The Impact of Test Case Prioritization on Test Coverage versus Defects Found

Ramadan Abdunabi
Computer Information Systems Dept
Colorado State University
Fort Collins, CO 80523
Ramadan.Abdunabi@colostate.edu

Yashwant K. Malaiya
Computer Science Dept
Colorado State University
Fort Collins, CO 80523
Yashwant.Malaiya@colostate.edu

***Abstract*—Prior studies demonstrate the importance of the relationship between code coverage and defects found to determine the effectiveness of test inputs. The variation of defect coverage with code coverage has been studied for a fixed execution sequence of test cases. We conducted an experiment to evaluate hypotheses expressed in two research questions. The first question addresses the relationship between defect coverage with code coverage for different execution sequences of test cases. The second research question evaluates the effectiveness and expensiveness of employing the prioritization techniques. The study confirms that altering the test cases execution order can affect the relationship between the code coverage and defects found. The results show that the optimal prioritization outperforms the st-total and the random prioritization.**

***Index Terms*—Defect Coverage, Statement Coverage, Empirical Study, Test Cases, Prioritization**

I. Introduction

The relationship between test coverage and the number of defects found is an indicator of software testing process. Horgan and London [9] showed that code coverage is an indicator of testing effectiveness and completeness. A high code coverage with low fault rate indicates a high software reliability.

In this work, we conducted an experiment to investigate the relationship between test coverage and number of defects found for varying execution order of test cases. The test cases are ordered in each execution sequence based on Test Case Prioritization (TCP) techniques defined in the study by Elbaum at al [7]. We propose an approach that can be used to improve the selection of the most cost-effective technique.

The research results provide insight into the tradeoffs between techniques, and the conditions underlying those tradeoffs, relative to the programs, test suites, and modified programs that we examine. If these results generalize to other workloads, they could guide the informed selection of techniques by practitioners. The analysis strategy we use demonstrably improves the prioritization technique selection process, and can be used by practitioners to evaluate techniques in a manner appropriate to their chosen testing scenarios.

Earlier experimental studies [4], [10] showed that at the beginning of testing, the number of detected faults grows slowly, which appears as *knee* shape in a plot of the defects found versus test coverage. Following the *knee*, when the fault detection has been already started, the number of detected faults grows linearly with test coverage. Saturation or acceleration is unlikely, and have not been encountered. However, all the studies to date presented in section 2 ignore the role of the ordering of test cases in the growth of number of detected faults and test coverage.

Different test cases might have different fault detecting ability and by altering the execution order of the test cases changes the curve of defects found versus test coverage. The first hypothesis in this study is that "the most effective test cases will give high test coverage and number of defects found than less effective test cases. However, we cannot distinguish between test cases unless we execute them and then investigate the reported data.

Another hypothesis is that altering the execution sequences of test cases will change the position of the *knee*. One other important expected results in this study is a better understanding of how to choose the test sequence that leads to maximum defect coverage growth. Further expectation of the possible effects is the change in the shape of the curve following the *knee* . The future goal of the study is to obtain a better insight into the behavior of the number of defects found with test coverage in order to propose more accurate faults prediction models.

An important consideration in our empirical study of fault detection is whether to use natural, manually seeded, or automatically seeded faults. Using automatically seeded faults was the only feasible option. Even apart from resource considerations, automatically seeded faults offer some advantages for experimentation: unlike hand-seeded faults, automatically seeded faults are not influenced by the person seeding the fault.

We created faulty versions using mutation-based fault injection to non-faulty programs. This was done to enlarge our data sets and because mutation-based faults have widely used for analyzing test effectiveness [6], [2], [13]. For all of our subject programs, any faulty versions that did not lead to at least one test case failure in our execution environment were excluded.

We initially had a concern that the effectiveness measure may not capture the artificially seeded faults and real effectiveness of a test suite and; its fault detection capability. Nevertheless, structural coverage and mutation score has been widely used as a successful surrogate of fault detection capability in software testing literature [6], [2], [13]. Andrews et al. [6] reported that faults generated

with mutation operators are similar to hand seeded faults for seven programs and natural faults in a space application.

MuJava, a mutation testing tool for Java programs, is used in our study to generate faulty version programs to simulate real faults. Mutant programs are commonly used as practical replacements for real faults. In these studies [14], [20], [1], a mutant program has shown that mutation faults can be representative of real faults.

The rest of the paper is organized as follows. $Section$ 2 presents the background material about the relationship between code coverage and defects found and Test Case Prioritization (TCP) techniques. The challenges in the study, experimental approaches, and data sets are discussed in $Section$ 3. $Section$ 4 analyzes the results with respect to the research questions. The potential threats to validity are discussed in $Section$ 5. Finally, $Section$ 6 summarizes the results and discusses future research work.

II. Related Work

Prior studies showed that the test coverage is a good estimator of the defects after achieving high coverage. Bishop [3] used test coverage to estimate the number of residual faults. This model is applied to a specific data set with known faults, and the results agreed well with the model. Cai et al. [5] proposed a reliability growth model that combines testing time and test coverage measures.

The Malaiya Li Bieman Karcich Skibbe *MLBKS* model [15] relates test coverage and defects found. This study demonstrated the early applicability of their model to describe the relationship between the test coverage and the defects found. Furthermore, the study showed that the initial defect density decides the position of the *knee* of the curve describing the model.

Cia and Lyu [4] studied the relationship between code coverage and fault detection using various scenarios: different test case coverage techniques, functional testing vs. random testing, normal operational testing vs. exceptional testing, and in different combinations of coverage metrics. The overall result showed code coverage is a good indicator for testing effectiveness.

In contrast to our results, these studies empirically showed that random test case prioritization (a.k.a. random ordering) can be ineffective. It has been a long tradition to deem random ordering as the lower bound control technique. If random ordering is indeed ineffective, we would like to ask the question: Why are other techniques not used to resolve tie cases?

Moreover, none of these studies are concerned about the relationship between code coverage and defect coverage under different execution sequences of test cases. In our study, the test cases are given particular order in each execution to examine the impact on the coverage and defect relationship. These prior studies have investigated small programs than the number of programs. Thus, we suspect that the results in these prior studies are limited and cannot be generalized.

Recently, a number of test case prioritization approaches have been proposed [11], [23]. Similar to our work, these approaches compare Test Case Prioritization (TCP) techniques in terms of their effectiveness, similarity, efficiency, and performance degradation. While these studies are considered among the most complete studies in terms of evaluation depth, they ignored the static techniques considered in this paper. Thus, our study is differentiated by the unique goal of understanding the relationships between purely static TCPs.

Zhang et al. [23] studied the regression prioritization and Fast Mutation Testing prioritization, which are similar to our study: to reorder the test cases to make regression/mutation testing faster. However, their mechanisms are different. Regression test prioritization aims to cover more program units faster, thus increasing the probability of revealing unknown faults earlier; whereas the locations of mutation faults (i.e., mutated statements) are known for mutation testing and a simple strategy can merely execute the test cases that reach the mutation faults, making the coverage-based regression test prioritization techniques are not suitable for mutation testing.

Rothermel et al. [19], [7] studied the 'Total' and 'Additional' approaches that utilize program dynamic coverage information. This work investigates test prioritization techniques for regression testing, Regression Test Prioritization (RTP), has different goals than ours. Our research aims at sorting a given set of tests into an expected most-productive order of execution and proposes which tests of a previously selected test suite are most advantageous to rerun with the same programs.

Total techniques, followed in this study, do not change values of test cases during the prioritization process, whereas additional techniques adjust values of the remaining test cases, taking into account the influence of already prioritized test cases. For example, a typical test-case prioritization technique reorders the tests to execute the effective tests earlier in order to speed up fault detection. Different from these traditional work; our work aims to improve the effectiveness of the existing tests rather than run them more efficiently.

Moreover, the study in [7] does not incorporate a testing time budget. Two graduate students of computer science are recruited to insert faults that were as realistic as possible based on their experience. Since there is usually a limited amount of time allowed for testing, in our work, faults seeding process as well as reordering the test cases are fully automated to reduce testing time.

III. The Experimental Study

We will address the following research questions:

RQ1: Does the alteration of test execution sequence change the position of *knee* in the defect coverage vs code coverage curve?

RQ2: Can the curves help us to determine which execution sequence is the best based on its effectiveness and expensiveness?

The approach of comparing Test Case Prioritization (TCP) techniques is to first obtain several non-faulty programs, mu-

tant faults, and test suites. Then, the prioritization techniques are applied to the test suites, the resulting ordered suites are executed, and measurements are taken of their effectiveness.

The subject programs vary both in terms of their size based on the LOC (lines of code) and functionality as shown in Table(1). This allows us to evaluate across a very broad spectrum of programs.

Assume P be a non-faulty program, T be a test suite for P, and testing is concerned with validating program P. To facilitate this, engineers often begin by reusing T, but reusing all of T (the retest-all approach) can be inordinately expensive. Thus, we aim to find an approach for rendering reuse most cost-effective test selection and test case prioritization.

Test Case Prioritization (TCP) techniques are one way to assist speeding up the effectiveness of testing process. Test case prioritization (TCP) techniques reorder the test cases in T such that testing objectives can be met more quickly. Our objective involves revealing faults, and find the TCP techniques capable of revealing faults more quickly. Because TCP techniques do not themselves discard (less effective) test cases, they can avoid the drawbacks that can occur with different test selection.

Alternately, in cases where discarding test cases is acceptable, test case prioritization can be used in conjunction with different test selection to prioritize the test cases in the selected test suite. Further, test case prioritization can increase the likelihood that, testing time will have been spent more beneficially than if test cases were not prioritized.

Mutation Testing is applied as a test case prioritization technique, measures how quickly a test suite detects the mutant in the testing process. Testing sequences are rescheduled based on the rate of mutant killing. Automating test case prioritization can effectively improve the rate of fault detection of test suites.

The tool MuJava [16] was used to seed mutation faults. Using MuJava, all possible faults within MuJava's parameters were generated for each sample program. Of these, faults that spanned multiple lines and faults in sample classes corresponded to events deliberately omitted. Faults not inside methods (i.e., in class-variable declarations and initialization) were also omitted, because their coverage is not tracked by EclEmma - a Free Java Code Coverage for Eclipse [12]. Equivalent mutants were not accounted for in this experiment, because it would have been unfeasible to examine every mutant to see if it could lead to a failure.

A test suite is generated using Raandoop - a free Automatic Unit Test Generation for Java [17], for each fault in the sample. For each test suite, fault pair, each test case is executed on the clean version of the application and, if it covered the line containing the fault on the faulty version. To determine whether a test suite covered a faulty line, the coverage report from EclEmma was examined.

A. Challenges in the Study

The success of an experiment relies on real applications with natural faults. However, usually the number of natural faults are not enough to factor into the experiment. Similarly, when there are no faults in the programs, researchers have to seed faults to produce faulty versions. These faults are seeded using artificial injecting faults tools such as MuJava.

Subject Program	Language	Type	Lines of Code	Downloads
Students Project	Java	Artificial	160	–
NanoXML	Java	Real	7,646	269
Jtopas	Java	Real	5,400	340

Tab. I: Subject Programs Characteristics

When Natural test suites are available, they usually do not give a good code coverage. To control this problem, more test cases need to be generated to obtain a high code coverage. The alternative solution is the commercial software applications, however, such applications are restricted to use. Thus, we used Raandoop tool to generate test suites for the subject programs.

The most well-known empirical approaches are the controlled experiments and case studies. The advantages of the controlled experiment is that the independent variable can be treated to identify their impact on the dependent variables. Therefore, the results won't depend on unknown factors. The disadvantages of such an approach are the threats to validity (see $Section\ 5$) due to the manufacturing of test cases and seeding defects. The advantages of the case studies are that they are performed on real objects and this reduces the cost of code inspection and artificiality injecting faults and generating test cases.

B. Data Sets

Three subject programs Students Project, Jtopas, and NanoXML are written in Java. The faults in the Students project (from a class project at CSU) are artificial faults caused by misconception of the project requirements. The test cases are developed based on the project specifications. The NanoXML and Jtopas are open source software applications downloaded from Software-artifact Infrastructure Repository (SIR) for experimentation [18].

For the Jtopas program, the test cases are available. However, there are no natural faults in the Jtopas program. Therefore, the faults are injected by using MuJava. For the NanoXML program, the faults are natural and the test cases are not available. Raandoop is used to generate the test cases for NanoXML. For all programs, test cases are developed in Java based on JUnit testing framework. Table I summarizes the characteristics of the object programs.

C. The Experimental Approach

This study is conducted to execute the test cases in different order and observe the effects on the relationship between the defect and code coverage. For each execution of test suites, the test cases are ordered based on different prioritization techniques as defined by Elbaum et al.[7]. The highest priority tests are executed first to find faults faster and reduce the cost of the of testing process. We order the test cases based on the statement coverage.

We investigate one version of three subject programs. Each version has multiple defects identified by either bug reports or code inspection, and it is associated with a set of test cases that expose these defects. The test cases are ordered based on Optimal, Random, and st-total prioritization

Tool	Eclipse	Eclemma	Raandoop	MuJava
Type	Free	Free	Free	Free
Language	Java and Others	Java	Java	Java
Purpose	JDK	Code Coverage	Test Case Generation	Artificially Seed Faults

Tab. II: Experimental Tool Attributes

techniques [7]. The optimal technique optimally order test cases in a test suite; it assumes that the faults are known. The random technique randomizes the order of test cases. St-total is an abbreviation of total statement technique in which the test cases sorted using statement coverage data for each test case.

The data is collected in a tabular manner such that each row comprises number of test cases, cumulative code coverage, and a cumulative number of defects found. The prioritization techniques are applied on the subject programs and the data is collected in Table III, Table IV , and Table V.

To get more insight about the relationship between defect coverage and statement coverage, the data is displayed in plots as shown on Figure 1, Figure 2, and Figure 3. Obviously, The curves in the plots help to compare the prioritization techniques.

The software development environment Eclipse IDE [8] is employed to compile and run the programs and test cases. The free Eclipse plug-in Java code coverage tool named Eclemma [22] is used to exhibit the fraction of covered code and the number of defects found for particular execution order of test cases. EclEmma is a coverage measurement of Java programs and it measures the statement coverage within JUnit testing framework. Table II summarizes the attributes of these software tools.

For Students and NanoXML programs, test cases are automatically generated for these programs, and in each execution sequence, the first test case is chosen and executed, then the data is collected. The subsequent test case is chosen with respect to the prioritization technique, then it is executed in combination with prior test cases. This process continues until all test cases in such execution sequence are executed. This approach is not followed for Jtopas program because the faults are artificially seeded.

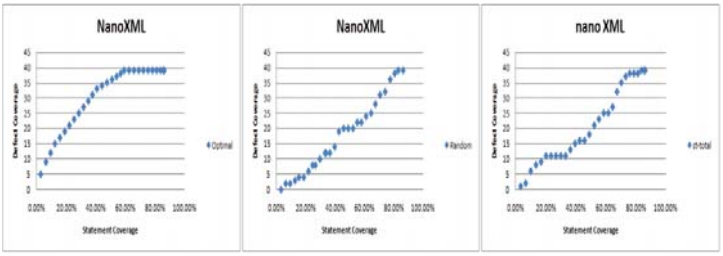
Then, the test cases are executed to kill the mutants and the mutant scores are recorded using MuJava. To obtain the coverage data, test cases are cumulatively executed and the data is recorded.

IV. Results and Discussion

The analysis of the data is driven by the answer of two questions, **RQ1** and **RQ2**. Overall, the results indicate that there is enough statistical evidence to eliminate the null hypothesis.

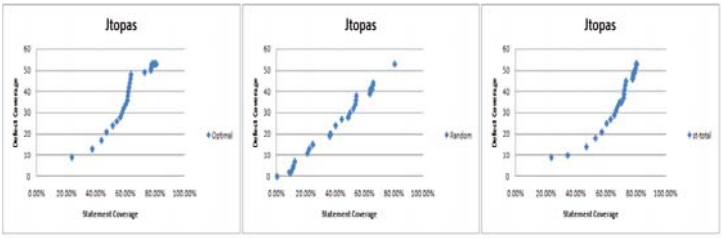
For question **RQ1**, the determination of the *knee* is conducted based on manual extrapolation and it is subjective. A straight line is fit to the part of the curve that has a maximum number of points. The straight line along the maximum number of points represents the high number of defects found and high coverage.

The point of intersection of that straight line with the curve at the lowest possible statement-coverage value is the *knee*. Tables VI show the statement-coverage values at the *knee* points of the three test execution sequences for NanoXML, Jtopas, and Students Project receptively. The results confirm the hypothesis expressed in **RQ1**; the position of the *knee* is different in each prioritization technique. After the *knee*, the curves that represent the relationship between statement coverage and defects found increase semi-linearly.



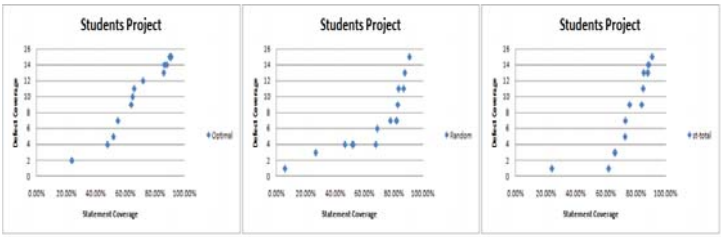
(a) Optimal (b) Random (c) St-Total

Fig. 1: Test Execution Sequences of the NanoXML



(a) Optimal (b) Random (c) St-Total

Fig. 2: Test Execution Sequences of the Jtopas



(a) Optimal (b) Random (c) St-Total

Fig. 3: Test Execution Sequences of the Students Project

Therefore, the highest defects found and statement-coverage is attained after the *knee*. Obviously, the position of the *knee* reacts to the starting point of exposing high defects and statement coverage. Furthermore, fewer faults in the early stage during the execution of each test sequence, causes the *knee* to pushed further along the statement-coverage axis. Hence, the position of the *knee* is a good measure of the effectiveness of prioritization techniques.

Building on the alteration of the *knee* position, the most effective test cases provide high coverage and high defect density. The *knee* position confirms that the optimal prioritization outperforms st-total and random prioritization in all programs. For optimal, the most effective test inputs are scheduled first causing most faults are exposed early, hence the *knee* occurs at a lower statement coverage value.

The random prioritization technique is the least desirable of all programs because the least effective test inputs are non-deterministic-ally executed near the beginning in the

Statement Coverage vs Defect Coverage						
N	Optimal		Random		st-total	
	SC	DC	SC	DC	SC	DC
1	2.90%	5	3.20%	0	3.50%	1
2	6.40%	9	6.40%	2	6.80%	2
3	9.60%	12	9.40%	2	10.20%	6
4	12.50%	15	12.70%	3	13.70%	8
5	15.90%	17	15.40%	4	17.10%	9
6	19.20%	19	18.70%	4	20.40%	11
7	22.30%	21	21.90%	6	23.70%	11
8	25.50%	23	24.70%	8	27.00%	11
9	28.70%	25	26.60%	8	30.20%	11
10	31.90%	27	29.70%	10	33.40%	11
11	35.10%	29	33.20%	12	36.50%	13
12	37.90%	31	33.90%	12	39.70%	15
13	40.90%	33	36.50%	12	42.80%	16
14	44.30%	34	39.80%	14	46.00%	16
15	47.70%	35	42.70%	19	49.20%	18
16	51.10%	36	45.90%	20	52.50%	21
17	54.20%	37	49.00%	20	55.60%	23
18	56.90%	38	52.30%	20	58.80%	25
19	59.30%	39	55.20%	22	61.80%	25
20	62.60%	39	58.00%	22	64.70%	27
21	65.90%	39	61.20%	24	67.60%	32
22	69.20%	39	64.60%	25	70.50%	35
23	72.30%	39	67.50%	28	73.40%	37
24	75.50%	39	70.70%	31	76.10%	38
25	78.50%	39	74.20%	32	78.90%	38
26	81.30%	39	77.60%	36	81.50%	38
27	83.90%	39	80.70%	38	83.90%	39
28	85.70%	39	83.10%	39	85.70%	39
29	86.40%	39	86.40%	39	86.40%	39

Tab. III: NanoXML TCP Techniques

Statement Coverage vs Defect Coverage						
N	Optimal		Random		st-total	
	SC	DC	SC	DC	SC	DC
1	23.80%	9	0.40%	0	23.80%	9
2	37.60%	13	8.70%	2	34.70%	10
3	43.80%	17	10.00%	2	47.20%	14
4	47.20%	21	11.20%	4	53.20%	18
5	51.40%	24	11.60%	5	57.40%	21
6	54.10%	26	12.50%	7	60.70%	25
7	56.50%	28	21.10%	11	63.30%	27
8	57.70%	30	22.40%	13	65.80%	29
9	58.70%	32	24.70%	15	66.90%	31
10	60.10%	34	24.90%	15	68.00%	33
11	61.30%	36	36.40%	19	69.30%	35
12	61.60%	38	36.60%	20	70.20%	35
13	61.90%	40	37.10%	20	70.70%	35
14	62.40%	42	40.40%	24	72.00%	37
15	62.90%	44	44.60%	27	72.30%	39
16	63.40%	46	48.90%	28	72.60%	41
17	63.70%	48	49.20%	28	73.10%	43
18	72.90%	49	50.20%	30	73.60%	45
19	77.10%	50	52.60%	32	77.80%	46
20	77.40%	51	53.90%	34	78.10%	46
21	77.60%	52	54.30%	36	78.30%	47
22	77.70%	53	54.60%	38	78.40%	48
23	78.60%	53	63.80%	39	78.60%	49
24	79.10%	53	64.10%	41	79.40%	49
25	79.40%	53	64.40%	41	79.50%	49
26	80.20%	53	65.40%	41	80.00%	51
27	80.30%	53	65.90%	43	80.30%	53
28	80.60%	53	66.10%	44	80.60%	53
29	80.80%	53	80.80%	53	80.80%	53

Tab. IV: Jtopas TCP Techniques

Statement Coverage vs Defect Coverage						
N	Optimal		Random		st-total	
	SC	DC	SC	DC	SC	DC
1	23.90%	2	6.00%	1	24.20%	1
2	48.00%	4	27.00%	3	62.00%	1
3	52.00%	5	47.00%	4	66.00%	3
4	55.00%	7	52.00%	4	66.40%	3
5	64.00%	9	52.60%	4	73.00%	5
6	65.00%	10	68.00%	4	73.20%	7
7	66.00%	11	69.00%	6	76.00%	9
8	72.00%	12	78.00%	7	84.00%	9
9	86.00%	13	82.00%	7	85.00%	11
10	86.50%	14	82.30%	7	85.40%	13
11	88.00%	14	83.00%	9	88.00%	13
12	90.00%	15	83.60%	11	88.20%	13
13	90.60%	15	87.00%	11	88.40%	14
14	90.80%	15	87.80%	13	88.80%	14
15	91.00%	15	91.00%	15	91.00%	15

Tab. V: Students Project TCP Techniques

test suite. The least effective test cases exhibit low coverage and defect density. The st-total technique outperforms the randomized technique and it is subjectively closer to the optimal technique in terms of the early occurrence of the *knee*.

The effectiveness of the prioritization techniques in the research question **RQ2** is confirmed in the answer of question **RQ1**. The expensiveness/cost of implementing prioritization techniques can be answered informally by investigation of Table III, Table IV, and Table V.

For instance, the number of defects found and obtained coverage by the execution of ten test cases for each technique in the Table III are (33.40%,27),(31.90%,11), and (29.70%,10). Therefore, the number of defects found is compared as follows: $27 > 11 > 10$ for optimal, st-total, and random respectively. This demonstrates that the optimal technique is less expensive than the st-total and the random techniques.

The coverage is compared as follows: $33.40\% > 31.90\% > 29.70\%$ for st-total, optimal, and random respectively. Therefore, there is a trade off between the optimal and st-total. Due to the fact that the number of defects found is more important than the coverage, the optimal technique is better than the st-total technique.

Definitely, the random technique is the most expensive. Following the same approach for each row in the three tables, we could conclude the following: the optimal and st-total prioritization technique significantly outperform the random technique. On average, the st-total technique is closer to the optimal technique.

Nevertheless, the above analysis of the results is performed based on subjective observation of the data. The results should be supported by some statistical evidences. Therefore, the analysis of variance ANOVA is applied to measure the differences between prioritization techniques in terms of number of exposed defects. To acquire more insight about the differences between techniques, the Tukey statistical model is used to compare the prioritization techniques.

NanoXML	
Execution Order	Coverage Value at the Knee
Optimal	14%
st-total	45%
Random	60 %

Jtopas	
Execution Order	Coverage Value at the Knee
Optimal	42%
st-total	49%
Random	53 %

Students Project	
Execution Order	Coverage Value at the Knee
Optimal	45%
st-total	60%
Random	67 %

Tab. VI: Positions of Knees

The analysis of variance shows no significant differences between prioritization techniques in case of all test cases (100%) are executed. This is true because when all the test cases are executed, we will get the same code coverage and defects found values. A prioritization technique is considered effective if the testing process is prematurely halted for some reasons, and a large number of defects is detected. More precisely, we can compare the prioritization techniques based on the number of defects covered after applying about 50% of the test cases.

Following this approach, the ANOVA and Tukey analysis are applied to compare the prioritization techniques after applying around 50% of test cases. The one way ANOVA analysis was conducted to compare the effect of execution order on defects found in optimal, st-total, and random conditions. As illustrated in the table, there was a significant effect of execution order on defects found at the $p < .05$ level for the three conditions $[F(2, 42) = 19.04, p = 0.000]$.

Defects versus Number of Test Cases					
Source	DF	SS	MS	F	P
Def	2	1795.2	897.6	19.04	0.000
Error	42	1979.9	47.1		
Total	44	3775.1			

Tab. VII: The ANOVA Analysis for NanoXML

And then, as illustrated in the table below, Post hoc comparisons using the Tukey HSD test indicated that the mean score for the optimal condition ($M = 22.33, SD = 9.40$) was significantly different than the random condition ($M = 7.73, SD = 5.37$). However, the st-total ($M = 10.60, SD = 4.89$) did not significantly differ from the random conditions.

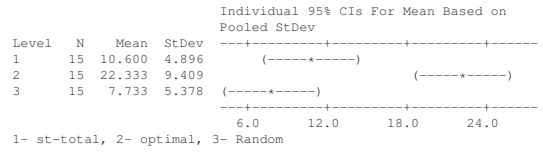


Fig. 4: Tukey HSD Test for NanoXML

Taken together, these results suggest that the optimal order really do have an effect on defects found. Specifically, our results suggest that when test cases are ordered as optimal, errors are exposed early.

The same statistical analysis has been conducted with Jtopas and students projects and the result is shown in the following tables. As a summary, there is enough statistical evidence on the differences between prioritization techniques.

The optimal technique is statistically more effective than the random technique in all programs. For the NanoXML, the optimal technique significantly outperforms the st-total. For the Jtopas and students programs, optimal is not statistically more effective than st-total; st-total is more effective than the random technique for Jtopas program, and they are closer in NanoXML and students projects. Finally the random technique exposes the smallest number of defects in all programs.

Defects versus Number of Test Cases					
Source	DF	SS	MS	F	P
Def	2	2435.4	1217.7	12.73	0.000
Error	42	4017.6	95.7		
Total	44	76453.0			

Tab. VIII: The ANOVA Analysis for Jtopas

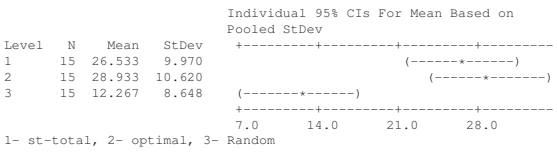


Fig. 5: Tukey HSD Test for Jtopas

Defects versus Number of Test Cases					
Source	DF	SS	MS	F	P
Def	2	81.7	40.8	3.19	0.057
Error	27	345.8	12.8		
Total	29	427.5			

Tab. IX: The ANOVA Analysis for Student's Project

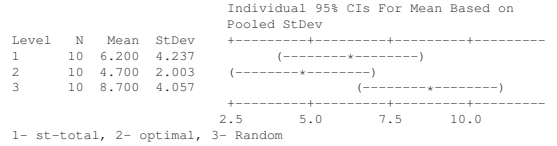


Fig. 6: Tukey HSD Test for Student's Project

To gain more statistical evidences about the differences between the techniques, we attempted to fit to the curves in the plots a collection of statistical models including: Regression Linear Model, Quadratic Model, Cubic Model, and Logistic Model, and then to compare the $R2$ values (The goodness of fit). However, for each program there was no model that fit all the curves and thus we could not compare the $R2$ values.

V. Threats to Validity

In many empirical studies, as the one proposed here, there will be potential threats to validity, both construct and external. In this section, we discuss threats to validity and explain how to reduce the chances of these threats.

A. Threats to Construct Validity

The inferences in the forgoing section can be affected by the following factors. Threats due to incorrect estimations of the total number of faults. For instance, the students project has artificial faults. These faults are detected by manual inspection of the source code and by designing new test cases. Such faults may not be the complete set of faults present in the program. However, it can be argued that the detected faults is a representative sample of the complete set of faults and any analysis with the detected set is also valid for the whole population.

Threats due to the presence of multiple faults. For students project, we only considered artificially occurring faults, all of which are present simultaneously in the program. Consider a test case that sensitizes two faults; both faults falsifies the condition stated in the assert statement of the JUnit test case.

When the JUnit test case fails, it is not possible to identify the fact that the failure was a result of sensitization of multiple faults. The failure will be counted as detection of one fault and the fault detection ability of the test case will be underestimated. One way to avoid this problem is to isolate the faults so that whenever there is a failure in the presence of a single fault, it is certain that the failure is due to that fault only.

Mutation testing resolves the problem of existing multiple faults in a program. Existence of multiple faults evolves extra effort to identify which fault is sensitized by exercising a particular test case.

Threats due to artificial faults; faults are artificially inserted in the Jtopas. The mutation system apply one mutation at a time and creates a new mutant program for every fault.

Moreover, the distribution of the artificial faults may not be same as that for the natural faults. This may result in completely different behavior of the defect coverage vs stament coverage for artificial faults. A code inspection and development documentations are required to gain more insight about the natural faults. However, a study by Smith et al. [21] demonstrated the efficiency of the MuJava tool, the mutant operators seed faults quite similar to natural faults.

Threats due to required prior knowledge. Optimal testing assumes prior knowledge of the defects and st-total strategy assumes prior knowledge of coverage of a test case. The two strategy are applicable only when such prior knowledge is available.

B. Threats to External Validity

The generalization of results can be affected by the following factors. Threats due to the object programs representativeness. The object programs are small and medium size. However, two of these objects are real applications. Furthermore, the fault patterns are natural in two objects.

Threats due to automaton test case generation. The test cases for NanoXML are generated using Randoop tool [17]. This tool automatically generate hundreds of test cases for a given set of classes within a limited time interval. As shown earlier, These test cases achieve a significant statement coverage.

Furthermore, a large number of test cases are doubled, such test cases do not factor in the study, these test cases are excluded. The elimination process requires the investigation of all test cases in order to remove insignificant test cases. There is no way for the tester to chose values for arguments to reduce the replication of test cases.

VI. Conclusion and Future Work

We conducted an experiment to evaluate how defect coverage varies with statement coverage for varying test execution sequences. Two prioritization techniques (optimal and st-total) relative to random testing are examined for altering the test cases execution order.

The result of the study indicates that the position of the *knee* is altered by changing the test cases execution order. We also showed a statistical evidence that the optimal prioritization technique outperforms the random and st-total techniques, and the random technique is the least effective technique.

The presented impact of time and synthesis of threats to validity open the door to future research. To cope with the threats to validity, we need to extend this study with more subject programs. Such subject programs should be associated with complete and accurate fault data and change log files. Preferably, we tend to investigate test cases developed by test engineers during the development and maintenance practices.

We also need to seek for an automatic tool, for given various metrics about programs, modifications, and test suites, we should be able to predict the prioritization technique most likely to succeed. The factors affecting prioritization success are, however complex, and interact in complex ways. We do not possess sufficient empirical data to allow creation of such a general prediction algorithm, and the complexities of gathering such data are such that it may be years before it can be made available. Moreover, even if we possessed a general prediction algorithm capable of distinguishing between existing prioritization techniques, such an algorithm might not extend to additional techniques that may be created.

References

[1] N.S. Akbar, J.H. Andrews, and D.J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 351–360, New York, NY, USA, 2008. ACM.

[2] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, page 411. ACM, 2005.

[3] P. Bishop. Estimating residual faults from code coverage. *Computer Safety, Reliability and Security*, pages 325–344.

[4] X. Cai and M.R. Lyu. The effect of code coverage on fault detection under different testing profiles. *ACM SIGSOFT Software Engineering Notes*, 30(4):7, 2005.

[5] X. Cai and M.R. Lyu. Software reliability modelling with test coverage: Experimentation and measurement with a fault-tolerant software project. 2007.

[6] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, 2006.

[7] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[8] The Eclipse Foundation. Eclips ide foundation, 2010. **http://www.eclipse.org/**.

[9] P.G. Frankl and E. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988.

[10] S. Goren and F.J. Ferguson. Test sequence generation for controller verification and test with high coverage. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 11(4):916–938, 2006.

[11] C. Henard, M. Papadakis, M. Harman, Y. Jia, and T.Y. Le. Comparing white-box and black-box test prioritization. In *Proceedings of the 38th International Conference on Software Engineering*, pages 523–534. ACM, 2016.

[12] M.R. Hoffmann, B. Janiczak, and E. Mandrikov. Eclemma-jacoco java code coverage library, 2011.

[13] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[14] R. Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 433–436, New York, NY, USA, 2014. ACM.

[15] Y.K. Malaiya, C. Braganza, and C. Sutaria. Early Applicability of the Coverage/Defect Model. In *Software Reliability Engineering*, pages 127–128, 2005.

[16] J. Offutt. java (mujava) - a mutation system for java programs, November 31 2008. **http://cs.gmu.edu/ offutt/mujava/**.

[17] C. Pacheco and M.D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816. ACM, 2007.

[18] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do. Software-artifact infrastructure repository (sir), 2010. **http://sir.unl.edu/portal/index.html**.

[19] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold. Test case prioritization. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[20] A. Schwartz and M. Hetzel. The impact of fault type on the relationship between code coverage and fault detection. In *2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST)*, pages 29–35, May 2016.

[21] B.H. Smith and L. Williams. An empirical evaluation of the MuJava mutation operators. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 193–202, 2007.

[22] W.E. Wong, J.R. Horgan, S. London, A.P. Mathur, H.N.S. Inc, and M.D. Germantown. Effect of test set size and block coverage on the fault detectioneffectiveness. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 230–238, 1994.

[23] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 192–201. IEEE, 2013.