AN ABSTRACT OF THE THESIS OF

Shalini Shamasunder for the degree of Master of Science in Computer Science presented on June 14, 2012.

Title: Empirical Study - Pairwise Prediction of Fault Based on Coverage

Abstract approved: _____

Alex Groce

Researchers/engineers in the field of software testing have valued *coverage* as a testing metric for decades now. There have been various empirical results that have shown that as coverage increases the ability of the test program to detect a fault also increases. As a result numerous coverage techniques have been introduced. Which coverage criteria co-relates better with fault detection? Which coverage criteria on the other hand have lower correlation with fault detection? In other words, does it make more sense to achieve a higher percentage of *c1* kind of coverage over a higher percentage of *c2* coverage to gain good fault detection rate. Do the popular block and branch coverage perform better or does path coverage outperform them? Answering these questions will help future engineers/researchers in generating more efficient test suites and in gaining a better metric of measurement. This also helps in test suite minimization. This thesis studies the relationship between coverage and mutant kill-rates over large, randomly generated test suites for statement, branch, predicate, and path coverage of two realistic programs to answer the above open questions. The experiments both confirm conventional wisdom about these coverage criteria and contains a few surprises.

Empirical Study - Pairwise Prediction of Fault Based on Coverage

by

Shalini Shamasunder

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 14, 2012
Commencement June 2013

Master of Science thesis of Shalini Shamasunder presented on June 14, 2012.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Shalini Shamasunder, Author

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude towards my advisor Dr. Alex Groce for his constant support, guidance and encouragement throughout my work. He has been a great source of inspiration in my academic life. His trust in me made me want to keep up the hard work. With his help, I have gained a completely new experience and developed a new perspective of thinking. I sincerely thank him for giving me this opportunity to work with him and learn from him.

I would also like to thank Chaoqiang Zhang aka Super who lend his helping hand all through the course of experiments and during my masters. He was a huge helping hand in helping me resolve the bugs and many issues in running the experiments. He was extremely patient and expanded my knowledge.

I must also thank all my friends who have directly or indirectly provided help at some of the craziest hours of work. I am indebted to the most important people in my life: my family, especially my parents, sister and all other family members. They gave me all the support and encouragement throughout my career. Nothing that I have accomplished would have been possible without them.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

1.1 Overview

Code coverage has gained high importance in the past few decades. "Code coverage is a measure/metric used in software testing that describes the degree to which the source code of a program has been tested and helps in knowing when to stop testing." [2] Coverage is basically divided into structural coverage, data flow coverage and logic coverage. Block, branch, path coverage fall under structural coverage while def-use, c-use, p-use falls under data flow coverage. Predicate coverage is a good example of logic coverage. These coverage criteria are explained in detail in the section that follows. Most commonly used coverage criteria in the industry by testers in general are block and branch coverage which are the very basic forms of coverage. Achieving a high percentage of coverage using these criteria is much simpler and more feasible than with other forms of coverage.

Code coverage has been proposed as an indicator of testing effectiveness and completeness by researchers before. The relationship between three important properties of test suites: size , structural coverage and fault finding effectiveness has been studied by many researchers under different circumstances be it under minimization or empirical studies in general. One such early empirical study by Frankl et al. [7] on test effectiveness indicated that the likelihood of detecting a fault increased sharply as very high coverage levels were reached but the magnitude of the increase seemed inconsistent. Was this increase just because of the increase in the size of the test suite? Namin and Andrews [11] used four coverage techniques: p-use, c-use, block , decision coverage and proposed that though both size and coverage together are both good predictors of test effectiveness; and that if size is kept constant coverage is correlated with effectiveness. They [7] also indicated that the testing based on coverage was more effective than random testing with no adequacy criteria meaning that coverage was definitely an important factor in determining the effectiveness of the test suite. Similar studies were performed by Wong et al. [20] and Frankl and Weiss [8] showing that that the correlation between test ef-

fectiveness and block coverage is high and so is the case with def-use and branch. Some other researchers went further to explore how this code coverage differs under different testing profiles and the benefit of coverage in real world for developers. Xia Cai and Lyu [6] proposed that the estimation of code coverage on testing effectiveness varies under different testing profiles and is not standard across every testing profile. [5] Experiments indicated that the coverage rate seemed to not go beyond 70 or 80 percent. One reason was reported to be that the developers were sufficiently confident about the quality of their tests and that they could not think of anything else to test and another reason being that getting higher rates often turned out to be very expensive because the systems architecture. It was also noted that even given a system with a reasonably testable architecture, the coverage of the tests would stall at a similar level, unless a code coverage analysis and visualization tool was used. Code coverage tools and the knowledge of code coverage hence is extremely essential to develop good test suites. Apart from these empirical studies that were directly related to coverage and its effectiveness, another literature that studied this in a different perspective was test suite minimization. Test suite minimization is achieved by identifying redundant test cases and removing them in order to minimize the size of the test suite. One measure of the suite quality after reduction is its fault detection capability. Empirical studies have shown that test suite minimizations reduces the effectiveness of fault detection [18] [21] [17]. In fact, a potential drawback observed in test suite reduction studies is that removal of test cases from a test suite may highly decrease the fault detection effectiveness of the remaining suite. Thus, the tradeoff between the time required to execute and manage test suites and their fault detection effectiveness should be considered when applying test suite reduction techniques. Therefore it is necessary to strike a balance between these trade offs and benefits. Knowing a good way to minimize test suites without effecting the quality is required.

Therefore to summarize the previous work in this field, we can state that there is sufficient empirical evidence that shows that there is a strong correlation between the code coverage and the ability to expose faults of test suites proving that coverage is definitely a good indicator of test effectiveness. This knowledge leads to some interesting research questions. Does having a test suite covering A mean that another test case covering A is less likely to add further mutant detection? Under which coverage measures/coverage profiles are correlations in mutant detection ability of test cases high and under which

low? If test cases A and B get the same coverage by criteria X, what's the correlation on criteria Y? Hence an important relation remains yet to be explored: relationship between different kinds of coverage and mutant kill rates as measured by the pairwise prediction of fault detection based on coverage. What does x percent of coverage c1 say about the test suite and its mutant detection capability in comparison with another coverage c2?

Let's take the following scenario: A researcher investigating automated testing methods applies methods A, B, and C to a large set of programs, giving each method 5 hours to generate tests. Examining the results, she hopes to announce which method is, on average, best - or at least to determine which method, for each different class of programs (simple tree structures, heap structures, file systems, array-based structures, numeric programs...) tends to be best. Unfortunately, she has a number of ways to evaluate each test suite. For some programs, method A may produce the best predicate coverage, method B the best path coverage, and method C the best shape coverage. She cannot simply average the rate of maximum coverage over programs, if she is considering predicate and path coverage, as the upper bound of reachable coverage is unknown.

An answer to a question like this would have important consequences on how we perform testing experiments. This thesis supports the work in the paper by Namin and Andrews [11] and attempts to report results that would be useful to both professional testers or practitioners as well as researchers. While professionals might be mostly interested in the experimental results and their implications for choosing a coverage criteria to develop better suites or in better minimization, researchers may find this experiment interesting in terms of its design and the potential future work that lies to be unfolded in this area. This thesis does not make any claim that the collection of the subject programs/languages represents all real world programs/software. But these results are definitely a step forward. The results are sound and hold a good value. It will definitely be interesting to see if the results are consistent across other languages in future work in this area.

The rest of the thesis is divided in sections as below: Section 3 does a literature review and describes some techniques and tools as background material. The section 3 that follows describes the experimental set up along with some other concepts and techniques used. The research questions are also listed in section 4. SGLIB [19] and YAFFS2 [15] are the two subject programs used. These are seen in detail in section 5 and 6 which explains data collection for both of these experiments and the analysis

of the data collection. After that in section 7 we combine the findings from the two experiments to come up with results to our research questions. We then describe the threats to validity and future work in the sections that follow.

Chapter 2: Background

Testingis the process of identifying defects, where a defect is any variance between actual and expected results. Test case is one execution of the program that may expose a bug. Test suite on the other hand is made up of test cases and is a set of executions of a program grouped together. A test suite or test case can be stated as effective when it is successful in detecting the faults in a program.

2.1 Coverage Criteria

Code coverage describes the degree to which the source code of a program has been tested. The last section clearly described the importance of coverage in the field of testing. It provides a bench mark or acts like an adequacy criteria in developing test suites. Having high coverage means better fault detection. In the experiment, four popular coverage techniques have been used: block, branch, path and predicate. Block and branch are popular coverage techniques used not just by researchers but also by software professions/practioners. These coverage techniques in particular are more basic forms of coverage and are more feasible when compared to other forms of coverage like path and predicate. Path and predicate coverage though not as commonly used as branch and block are still popular coverage techniques. The difference between the two being that path is structural coverage while predicate is logic coverage. They both do not subsume each other meaning that they are both independent, while path does subsume branch and block. The reverse does not hold true. Branch or block do not subsume path. Each of the coverage techniques are explained with diagrams for better understanding

Block/statement Coverage: Block coverage/statement coverage/ line coverage describes whether a block of code defined as not having any branch point within (the path of execution enters from the beginning and exits at the end) is executed or not. See images [1]

Branch Coverage: This coverage ensures that each possible branch from each decision point is executed at least once, thus ensuring that all reachable code is executed. (E.g,

an "if" statement) [1]

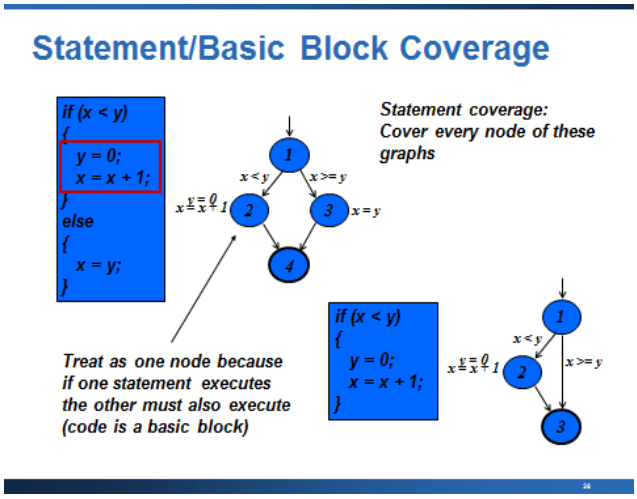
Path Coverage: Path coverage is a unique sequence of branches that the test case takes from entry to exit. This kind of coverage hence has the advantage of requiring very thorough testing. We use intra-procedural path meaning we consider path inside a function and eventually add up all path numbers in all functions [1] as path coverage.

Predicate Coverage: Predicate coverage is the most basic logical coverage criterion. It attempts to cover all values for all predicates in conditional expressions. It is to note that path does not subsume predicate or predicate does not subsume path so they are not in any subsumption relation [1]

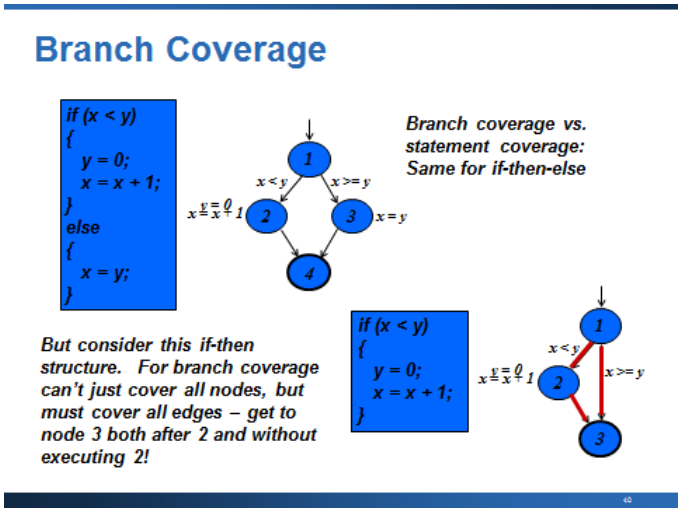
2.2 Instrumentation

Instrumentation basically adds program code that does not change functional behavior but collects information. In order to be able to measure the coverage of a program, we need to instrument the program. We have used a tool called CIL [13] in our case to instrument the two subject programs.

Figure 2.1: Block and Branch Coverage

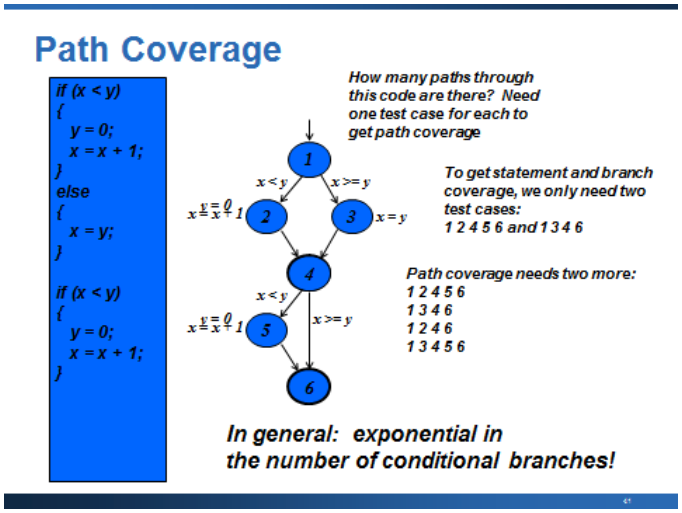


Block/statement coverage: Block coverage/statement coverage/ line coverage describes whether a block of code defined as not having any branch point within (the path of execution enters from the beginning and exits at the end) is executed or not.

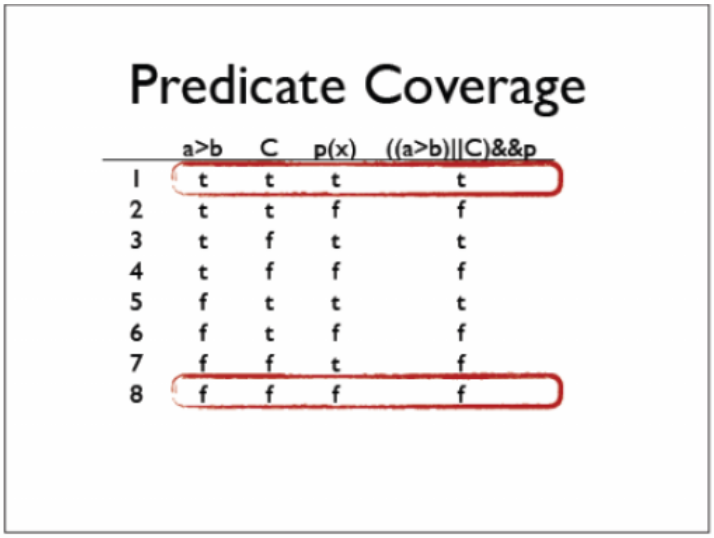


Branch coverage: This kind of coverage aims to ensure that each possible branch from each decision point is executed at least once, thus ensuring that all reachable code is executed. (E.g, "if" statement)

Figure 2.2: Path and Predicate Coverage



Path coverage: Path coverage is a unique sequence of branches that the test case takes from entry to exit. This kind of coverage hence has the advantage of requiring very thorough testing. Test cases might cover the same nodes but through different paths.



Chapter 3: Literature Review

We divide our related work section into categories to better understand how researchers have studied coverage in the past. This section also explains how we have arrived at some of our research questions and how they prove to be useful.

- General empirical studies on coverage and its influence on test suite effectiveness
- Comparison of coverage techniques
- Test suite minimization and its effect test suite effectiveness

3.1 General empirical studies on coverage and its comparison with size and effectiveness:

Wong et al. [20] in the early 1990's explored the relationship between size and fault-detection effectiveness. Their results showed that coverage was more correlated than size with fault detection effectiveness. They basically studied two relationships:

- Fault detection effectiveness with coverage
- Fault detection effectiveness with the size.

For this study block coverage was used as the coverage technique. Results did indicate that coverage and in this case block coverage was correlated with fault detection effectiveness. In fact it was a better indicator than size for the effectiveness. The study used quasi random testing for generating test sets, and they used manually seeded faults to determine the effectiveness.

Namin and Andrews [11] also studied the relationship between size, coverage and effectiveness of test set like in the Wong study. They used siemens suite (c language) and two other subject programs for the experiment and used four coverage techniques for evaluation: p-use, c-use, block , decision coverage and proposed that though both size and coverage together are good predictors of test effectiveness; and that if size is

kept constant coverage is correlated with effectiveness. This study did pick random tests but from a test pool of existing test cases and hence different from Wong et al. in that sense. Mainly, it did indicate that with size being kept constant coverage is an important factor in determining the test set effectiveness.

3.2 Comparison of coverage techniques:

Xia Cai and Lyu [6] proposed that the effects of code coverage in different testing profiles. They used random testing and functional testing to make a comparison between the two. They used four important coverage criteria: p-use, c-use, decision and block coverage in their experiment. They explored the following:

- The relationship revealed in normal operational testing versus exceptional testing: code coverage was found to be a better indicator in exceptional testing meaning that increasing code coverage would benefit such exceptional fault detection capability.
- The relationship revealed in functional testing versus random testing: Random testing was found to be a necessary complement to functional testing and also that code coverage worked well for both kinds of testing
- The relationship revealed in different combinations for various coverage metric: This one being particularly more of our interest. In general they did not report any significant difference between block/c-use and coverage/p-use under normal testing.

Though they have studied correlation of these coverage to some extent here, the study is very different from ours in the sense that it focuses more on the correlation of these coverage techniques with the different testing profiles begin random, functional, exceptional , normal and a comparison of how they differ under each of these circumstances. It focuses less on the correlation of these coverage techniques with effectiveness but more on their relation with these factors. This paper does not compare the coverage techniques in terms of the effectiveness of each coverage technique.

Hutchins et al.[10] conducted a study using du and edge coverage. They used random testing for generating test cases, used moderate size programs and seeded mutants into the programs for the study. Their study showed some interesting results. It indicated

that rather than having coverage as an adequacy criteria it servers better as an indicator of test inadequacy. Meaning that low coverage is a good reason to continue the testing efforts while a 80, 90 or 100% coverage does not really mean that all mutants are detected and we can stop testing. But it of course means that we have a good coverage of a lot of mutants. This study also showed that at the range of 90% to 100% coverage there seemed to be a rapid rise in the fault detection ratio. This paper though did not see a significant difference in the behavior of du vs edge coverage. This was also the case with Frankl et al.[7] . Frankl et al.[7] presented the results of the effectiveness of all use and decision or branch testing to random testing with no adequacy criteria. Their study was similar to Hutchins et al.[10], though they used a larger base/subject program code base. And in contrast to the earlier result, in this case the difference between the two coverage criteria wasn't very consistent. Hutchins et al.[10] though did find that there were differences in the way the coverage techniques behaved in the sense that during the 90% to 100% coverage range for each of these criteria, du seemed to have a dramatic increase in fault detection compared to edge, indicating that say 95% of du might not mean that the same number of mutants as 95% of edge. The coverage techniques do behave differently even at the same coverage level and studying this is definitely useful in knowing that achieving x% of which coverage criteria serves better.

Frankl and Weiss [8] performed an experiment comparing the effectiveness of all-uses and all-edges coverage criteria. They explored the answer for three research questions in their paper

- Are those test sets that satisfy x% the requirements induced by coverage c1 more likely to detect an error than those that satisfy y% of c2?
- For a fixed test size is c1 more likely to detect an error than c2?
- comparison of size vs coverage.

They generated a large number of tests using random testing. Their results indicated that all-uses proved to be significantly more effective than all-edges in most of the cases. Our experiments are similar in the sense that we are also comparing coverage criteria here. The difference being a) we use four coverage criteria instead of two for the comparison b) we also use reasonably much bigger subject programs in our study c) They use subject programs that had naturally occurring errors while we generate mutants using

an algorithm for our subject programs d) We explore additional research questions apart from 2 and 3 above with the coverage criteria.

3.3 Test suite minimization literature

Test suite minimization is a way of reducing the costs of saving and reusing test cases by eliminating redundant test cases from test suites. But how does this effect the fault detection capability of a reduced test suite?

This was studied in detail in some of the papers [17] [18] [21].

The early studies by Wong et al.[21] conclude from their experiments that if size of a test set is reduced by just removing test cases that are redundant in all-uses coverage then there is little or no reduction in the effectiveness of the reduced test test sets. They used randomly selected test cases from a pool of test cases for their experiment. They also manually seeded faults/mutants to measure effectiveness. But in general, reductions in fault detection is compromised to some extent.

A similar study by Rothermel et al.[18] indicate results similar to Wong study except that in this case they have used all-edge instead of all-uses and in this case removing test cases that are redundant in terms of all edge test cases does not still hold the effectiveness of the test suite. Effectiveness of the test suite is compromised even then. Though they used similar way of inducing mutants as in the Wong study, the difference was in the coverage criteria and the subject program used.

Hence, it is important to find a way of reducing test suites without reducing the effectiveness of the test suite and holding that as is. Both these used two different coverage techniques. Though we are not directly proving anything in terms of test suite minimization, we do believe that our results might prove to be useful to this field of researchers for some further study.

As seen above in the three categories, the relationship between coverage and effectiveness has been an important research topic and have been supported by a lot of researchers. Based on this background of the related work we go ahead and define our open research questions in the next section.

Chapter 4: Experimental Design

4.1 Research Questions

A look at our set of research questions would give an insight into what we are exploring in this paper and what makes this a worthwhile study and how answering this would help in better designing test suites.

- Under which coverage measures/coverage profiles are correlations in mutant detection ability high?
- Under which coverage measures/coverage profiles are correlations in mutant detection ability low?
- If test suites A and B get the same coverage by criteria X, what's the correlation on criteria Y?
- Would two test cases A and B with the same coverage (branch, block, path, predicate) detect the same number of mutants?

4.2 Experimental Setup

Our experiments uses two subject programs: SGLIB [19] and YAFFS2 [15]. Both of these are fairly large sized C programs. SGLIB has around 2k LOC and YAFFS2 around 14k on the other hand. For each of the two subject programs SGLIB [19]and YAFFS2 [15] we generate numerous test suites using random testing method which is explained in detail in section 4.3. The algorithm used in generating the random tests for them are explained in 5 and 6 respectively. At the same time, we create faulty versions of these subject programs by seeding mutants using mutation operators as explained in section 4.4. We call these as the faulty versions of the program and the untouched version is called the oracle version which is the program without any seeded mutants. The total number of mutants used for each program is mentioned in the table below as well:

Table 4.1: Subject Programs: SGLIB

SubjectProgram	LOC	TotalMutants	Block	Branch
SGLIB	2000			
a) List		88	87	72
b) SList		55	95	82
c) DLList		83	188	200
d) Rbtree		468	350	378
YAFFS	14000	1000	3518	4189

Now, we have the oracle and faulty versions of the program and also have a way of generating tests for these programs. In order to compute coverage of these tests, we have to instrument the program. So we use CIL [13] tool to instrument the program. This instrumentation helps in computation of coverage: branch, block, path and predicate coverage in our case. We have the overall number of branches, blocks and predicates mentioned in the table above.

After this, the tests are run on the subject programs, and the number of faults detected are determined by making a comparison of the oracle and the faulty version of the programs , therefore collecting the coverage information and the mutant information to perform correlation analysis.

4.3 Random Testing

Random testing [9] is performed by generating program inputs at random, drawn from some possibly changing probability distribution. We are using random testing to build a large test suite in our experiment. Since the test cases are randomly generated, each test case need not have a different coverage. Some of them might have the same path coverage, while some might cover the same nodes and some might be completely different. Random testing [9] [22] in our case is a much better approach than generating test suites with particular algorithms since that could be biased to constructing test suite that is weighing more towards one coverage technique rather than the other. In other terms, the results could be biased and hence less general. Also while generating very large test suites, random testing is a preferable method. Randomness means potentially a lot of test cases to examine, and thousands of variations of the same error and this works in our advantage as the perfect setting for our experiment. Along with random testing we

use the concept of feedback i,e., feedback directed random test generation for YAFFS since the parameter space is huge [16]. This therefore builds inputs incrementally by randomly selecting a method call to apply and finding arguments from among previously constructed inputs. The result determines if the input is illegal, redundant or useful based on which further inputs are generated. Therefore even though it is random testing this is still done systemically to some extent without still creating a bias towards one kind of coverage. It is also to note we are not randomly picking test cases from a test pool as in Namin and Andrews [11] paper that is the basis of this study but actually generating a random test suite meaning less bias towards a certain coverage criteria. The algorithm we used for generating test cases is in the section that follows.

4.4 Mutant Detection

Mutation Analysis injects/seeds artificial defects (mutants) into a program and checks whether the test suite is effective in finding the mutants. Andrews et al. [3] [4] showed that, despite the relative simplicity of the faults introduced by mutation operators, mutants behaved almost like real world faults and hence acts as a replacement for real world errors. In some of the empirical studies before, like [8] researchers manually seeded faults in subject software and measured the effectiveness. Frankl and Iakounenko on the other hand [7] used programs for which real faults had been identified during testing and these were seeded again to create the mutant version of the program. Performing large controlled experiments with real faults is much harder due to the difficulty of collecting a good number of faulty versions of the program. Some other researchers used mutation operators to seed faults in their experiments [11] [14] [3] [4]. To perform mutation analysis on a subject program, mutation operators are applied to the source code generating the mutant versions of the code. Each mutant represents a possibly faulty variant of the original program. So, we have multiple mutant versions of our subject programs. The advantage of using mutation operators over seeding faults manually or over using real faults is that it requires decreased effort and at the same time generates greater numbers of potentially faulty versions hence leading to better statistically significant results.

For our two subject programs, we created mutants using the tool implemented by Andrews et al. [3], which produces mutants based on a set of operators that have been shown to work well as fault proxies. For SGLIB all mutants were used but for YAFFS2

we sampled the mutants and randomly picked the mutants since the generated mutants was too large to finish our experiments in a reasonable amount of time. Hence we had to sample a random selection of mutants. Recent work shows that random selection of mutants can provide results comparable to selective mutation [22].

Chapter 5: SGLIB Program: Subject Software, Data Collection and Analysis

5.1 Subject Software

SGLIB [19] is a popular, open-source C library for data structures. It consists of a single C header file, sglib.h, with about 2000 lines of code consisting only of C macros. This file provides generic implementation of most common algorithms for arrays, lists, sorted lists, doubly linked lists, and red-black trees. We chose SGLIB as a subject program since its a widely used C library. The libraries that we have used for testing are

1) linked lists 2) sorted linked lists 3) double linked lists 4) red-black trees

A basic set of functions and macros are provided for each data structure like insertion, deletion, search, iterator traversal of elements, concatenation, reverse or sort. The list of these functions are established in the table below

The tables below shows some functionalities for SGLIB and YAFFS:

5.2 Data Collection

We began by generating, for each subject program in SGLIB: list, slist, dllist, rbtree - test suites of varying size till a stagnant pattern was seen. The sizes were varied starting from 10 test cases to until 10000 cases. All of the test suites were generated using random as in the algorithm below. It was not chosen randomly from a pool of test cases, but generated using random testing in itself to keep it more realistic and to remove any bias that maybe created when generating a test suite using an algorithm.

We chose the upper bound for the experiment because we observed that test suites of this size usually achieved very high coverage for our subject programs. For each test suite, the block, branch, path and predicate coverage were measured along with even coverage for individual test cases. This was done using CIL to instrument the program and then the coverage percentage was calculated.

Table 5.1: SGLIB: API Table

Functions/Macros	List	slist	dllist	Rbtree
add()	x	x	x	x
add-if-not-member()	x	x	x	x
concat()	x	-	x	-
delete()	x	x	x	x
delete-if-member()	x	x	x	x
is-member()	x	x	x	x
find-member()	x	x	x	x
len()	x	x	x	x
map-on-elements()	x	x	x	x
reverse()	x	-	x	-
sort()	x	-	x	-
find-member-or-place()	-	x	-	-
add-before()	-	-	x	-
add-after()	-	-	x	-
add-before-if-not-member()	-	-	x	-
add-after-if-not-member()	-	-	x	-
get-first()	-	-	x	-
get-last()	-	-	x	-

Table 5.2: YAFFS2: API Table

Functions/Macros	YAFFS2
chmod	x
close	x
closedir	x
fchmod	x
freespace	x
fstat	x
link	x
lseek	x
lstat	x
mkdir	x
open	x
opendir	x
read	x
write	x
readdir	x
readlink	x
rename	x
rewinddir	x
rmdir	x
stat	x
symlink	x
truncate	x
unlink	x

Table 5.3: SLIST: Data Table

TestLen	TestCase	Block	Branch	Path	Predicate	Mutants	TL*TC
5	5	25	18	12	141	26	25
5	10	25	20	13	141	20	50
10	10	35	29	34	188	41	100
5	50	35	29	24	188	40	250
10	50	35	30	49	188	43	500
10	100	35	30	51	188	43	1000
100	100	35	30	185	188	44	10000
100	500	35	30	215	188	44	50000
100	1000	35	30	227	188	44	100000

SGLIB DATA: Doubly linked list

Table 5.4: DLLIST: Data Table

TestLen	TestCase	Block	Branch	Path	Predicate	Mutants	TL*TC
5	5	61	52	14	617	35	25
5	10	75	67	20	698	44	50
10	10	96	98	32	987	56	100
5	50	89	88	28	962	54	250
10	50	108	115	60	1133	65	500
10	100	113	124	78	1187	68	1000
100	100	115	126	972	1206	72	10000
100	500	115	126	2168	1206	72	50000
100	1000	115	126	3312	1206	72	100000

For each of the subject program in SGLIB we used a mutant generation program first used by Andrews et al. [3] to generate mutants for code written in C. To generate mutants from a source file, each line of code was considered in sequence and each of four classes of mutation operators were applied. The first step was to generate and compile the mutants, and to run mutants and faulty versions on the entire test pool. We considered all the mutants. We ran all mutants on all test cases, and recorded which test cases detected which mutants, in the sense of forcing the mutant to give different output from the oracle version. For each of the test suites we generated, we computed the mutation detection number and this was also calculated for the overall test suite to see how many mutants each test suite detected.

```
SGLIB Test Generator Algorithm:
while( len < expected) {
randomly pick an api;
randomly generate parameters for it
add apicall(parameters) to test case
increase len;
}
```

The data table shows the data for sorted list, doubly linked list, list, and rbtree each along with all the coverage and mutant information. The first column here indicating the test case length (the number of function calls per test case is what this indicates), the second column is the number of test cases that are in a test suite. This has been varied to compute results at different sizes of the test suite and to obtain higher coverage. The third column is the block coverage per test suite, then the branch, path and predicate coverage per test suite. The final column indicates a multiple of the first two columns.

Table 5.5: RBTREE: Data Table

TestLen	TestCase	Block	Branch	Path	Predicate	Mutants	TL*TC
5	10	60	54	25	496	88	50
10	10	85	86	41	679	112	100
5	50	89	91	45	728	110	250
10	50	104	109	59	974	143	500
10	100	116	125	68	1135	146	1000
100	100	191	221	427	3194	197	10000
100	500	194	222	784	3353	202	50000
100	1000	200	224	1051	3410	202	100000
100	5000	202	230	2265	3554	202	500000

Table 5.6: LIST: Data Table

TestLen	TestCase	Block	Branch	Path	Predicate	Mutants	TL*TC
5	5	25	15	10	158	18	25
5	10	47	39	18	387	34	50
10	10	47	42	20	528	44	100
5	50	55	49	25	566	54	250
10	50	56	52	48	619	70	500
10	100	56	52	49	621	70	1000
100	100	56	52	586	635	74	10000
100	500	56	52	674	635	73	50000
100	1000	56	52	980	635	73	100000

5.3 Analysis

For analyzing our experimental results we used the analysis framework similar to the paper [11]. Experimental results were each first plotted on graphs as scatter plots to provide visualizations. After this graphs were created to plot the correlation between branch, block, predicate, path coverage with mutants. This was done in order to detect if there is any drastic variation between one coverage to the other with the same data and test set or if the results are more on a uniform pattern. Once these graphs were plotted, to go into more depth for analyzing the Pearson correlation coefficient, p-value were also calculated. This was done to indicate which coverage had a better correlation and then after this various regression models were constructed in order to obtain r^2 values

for better understanding of the relationship. The regression equation that described the relation between mutant and the particular coverage for each test program gave a good insight of the correlation between these variables.

5.3.1 Visualization using graphs

To better conceptualize the data gathered as the result of the experiments they were plotted as a scatter plot. The figure below shows the correlation between each of the coverage criteria with mutant detection. The x axis indicating the test suite sizes and the y axis showing the coverage and mutant detection. The graph below indicates how each coverage relates to mutant detection. From the graph for SGLIB: list, rbtree, slist and dllist it can be noted that though the base values are slightly different between each subject program the curves are almost similar. It does appear from the four graphs that :

- Branch, block seem to influence mutant detection proportionally. Though the graph is not linear the curves and how the graph turns out looks similar.
- Even predicate coverage seems to be more correlated to mutant detection than path.
- Amongst all these coverage criteria used, path seems to be the most unpredictable. It does not seem to correlate very well with mutant detection and knowing the path coverage of a program one might not be able to deduce its equivalent mutant killing rate.

Apart from studying the behavior of coverage with mutant detection, if we look at just coverage techniques themselves, it can be noted that:

- Branch , block , and predicate seem to correlate to each other. While path does not seem to correlate with these three coverage techniques. With two test suites of same branch coverage we can almost see that they would possibly have a very close block and predicate coverage as well, but the same can definitely not be said about path coverage.

- Path seems to increase in a more linear fashion with growing test suite size than the other coverage techniques.

To confirm these assumptions that we can draw from the graphs, we do further analysis which is described in more detail in the sections that follow.

5.3.2 Correlation of coverage and fault detection

With the data that has been plotted in graph there appears to be a positive correlation between coverage and mutant detection, Pearson correlation coefficients and p-values were calculated to affirm this. For every coverage curve vs mutant curve , the p-values and Pearson correlation coefficients were calculated. Below is the table for p-value and Pearson correlation coefficients for each of our subject programs and for each of the coverage vs mutant data. For Pearson coefficient correlation measure, the standard Guilford scale was used for verbal description; this scale describes correlation of under 0.4 as low, 0.4 to 0.7 as moderate, 0.7 to 0.9 as high, and over 0.9 as very high.

- For *sorted list* as in the table below, it can be seen that there is a "very high" correlation between block with mutant, predicate with mutant and branch with mutant i.e., the Pearson coefficient values are above 0.9 for all of these. But the path is "moderately" correlated with mutant detection.

- For *doubly linked list*, it can be seen that again there is a "very high" correlation of mutant detection with these coverage techniques: branch , block and then predicate. But here again the path is in the moderate range.

- Let us now see for *rbtree*: Branch, block and predicate seem to have a very high correlation again with mutant detection than the path coverage. Path coverage again falls under the moderate category.

- For *linked list*, Branch and predicate have a very high correlation and block has a high correlation with mutant detection while path seems to still show a moderate behavior. With each subject program in SGLIB , branch block and predicate seemed to have a very high correlation with mutant detection. Path on the other hand seemed to have a much lesser correlation in comparison.

Now using a similar scale to compare the correlation between the coverage techniques themselves. Block vs predicate, block vs branch, branch vs predicate : these have a "Very high" correlation among themselves. While the correlation between path and any of these techniques are not very high and fall in the moderate category and this is consistent across all of the subject programs in SGLIB: list, dllist, slist, rbtree and can be studied individually as above using the tables below for the coverage technique correlation study.

Thus it is clear that the correlation among the coverage techniques is high with these three coverage types for SGLIB program. In statistics, the p-value is the probability of obtaining a test statistic at least as extreme as the one that was actually observed, assuming that the null hypothesis is true. One often rejects the null hypothesis when the p-value is less than the significance level a (Greek alpha), which is often 0.05 or 0.01. So if the null hypothesis was that each of the coverage have no correlation with mutant detection, one can safely reject the null hypothesis in the case of branch , block and predicate coverage because of the lesser than significance p-value, therefore indicating the strong correlation between these coverage techniques and mutant detection. Though this null hypothesis is true in the case of path coverage for SGLIB since the p-value is not low.

Table 5.7: SLIST: Coverage Mutant Correlation

	Pearson Coeff	P-value	R^2 value
Block	0.973	0	94.8%
Branch	0.96	0	92.1%
Path	0.579	0.102	33.60%
Predicate	0.973	0	94.8%
TC-num	0.401	0.285	16.10%

5.3.3 Regression Analysis

Our correlation analysis showed us some interesting observations and a very standard behavior across the subject programs in SGLIB [19]. In statistics, the coefficient of determination r^2 is used in the context of statistical models whose main purpose is the prediction of future outcomes on the basis of other related information. It is the

Table 5.8: SLIST: Coverage Techniques Correlation

	Pearson Coeff	P-value	R^2 value
Block vs branch	0.991	0	98.10%
Block vs Path	0.483	0.188	
Block vs Predicate	1		100%
Branch vs Path	0.527	0.145	
Branch vs Predicate	0.991	0	98.10%
Path vs Predicate	0.483	0.188	

Table 5.9: DLLIST: Coverage Mutant Correlation

	Pearson Coeff	P-value	R^2 value
Block	0.996	0	99.10%
Branch	0.995	0	99.10%
Path	0.606	0.084	36.70%
Predicate	0.99	0	98%
TC-num	0.552	0.123	30.50%

Table 5.10: DLLIST: Coverage Techniques Correlation

	Pearson Coeff	P-value	R^2 value
Block vs branch	0.999	0	99.80%
Block vs Path	0.553	0.122	
Block vs Predicate	0.993	0	98.50%
Branch vs Path	0.562	0.116	
Branch vs Predicate	0.994	0	98.80%
Path vs Predicate	0.54	0.133	

Table 5.11: RBTREE: Coverage Mutant Correlation

	Pearson Coeff	P-value	R^2 value
Block	0.989	0	97.80%
Branch	0.99	0	97.90%
Path	0.68	0.03	46.30%
Predicate	0.968	0	93.80%
TC-num	0.503	0.139	25.30%

Table 5.12: RBTREE: Coverage Techniques Correlation

	Pearson Coeff	P-value	R^2 value
Block vs branch	1	0	99.90%
Block vs Path	0.733	0.016	
Block vs Predicate	0.992	0	98.30%
Branch vs Path	0.725	0.018	
Branch vs Predicate	0.991	0	98.10%
Path vs Predicate	0.754	0.012	

Table 5.13: LIST: Coverage Mutant Correlation

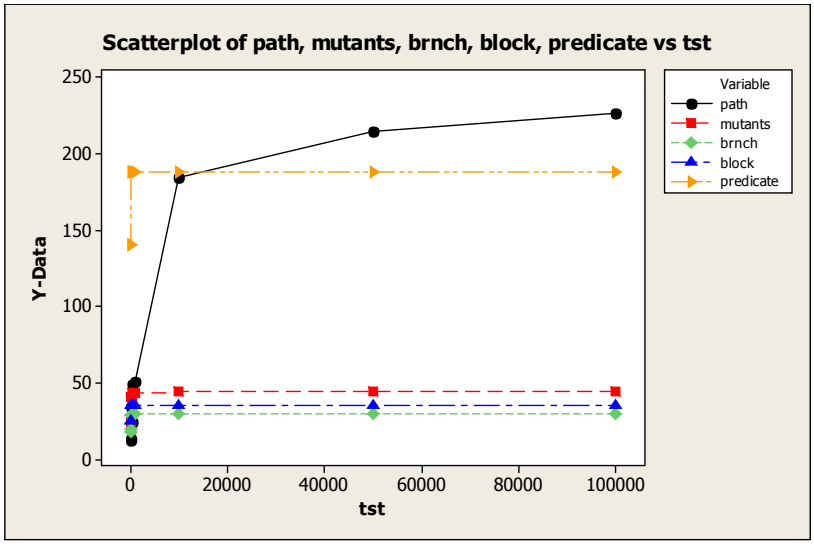
	Pearson Coeff	P-value	R^2 value
Block	0.9	0.001	81%
Branch	0.923	0	85.20%
Path	0.609	0.082	37%
Predicate	0.949	0	90%
TC-num	0.499	0.171	24.90%

Table 5.14: LIST: Coverage Techniques Correlation

	Pearson Coeff	P-value	R^2 value
Block vs branch	0.997	0	99.40%
Block vs Path	0.413	0.269	
Block vs Predicate	0.968	0	93.60%
Branch vs Path	0.435	0.242	
Branch vs Predicate	0.984	0	96.80%
Path vs Predicate	0.485	0.186	

proportion of variability in a data set that is accounted for by the statistical model. It provides a measure of how well future outcomes are likely to be predicted by the model. An r^2 of 1.0 indicates that the regression line perfectly fits the data. Using r^2s regression feature, various models were created for both the coverage relation with mutant and the correlation among coverage techniques. The third column in the tables below indicates the r^2 percentage. The regression line fits the data almost perfectly or very well in case of block , branch and predicate coverage. The values all above 90 or above 0.9 in decimal value. Even the regression line for the block vs predicate, block vs branch and branch vs block are almost perfectly fitting the regression line.

Scatter plot for SLLIST: Correlation of coverage with mutants



Scatter plot for DLLIST : Coverage vs mutants

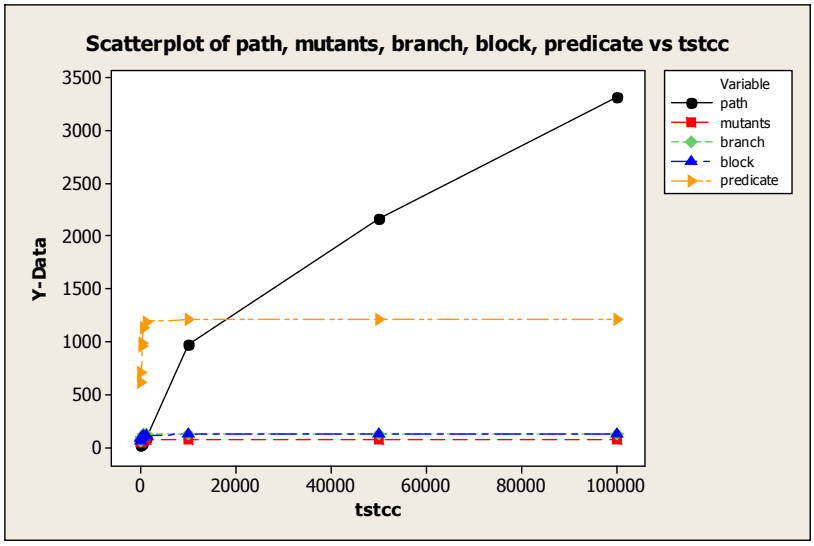
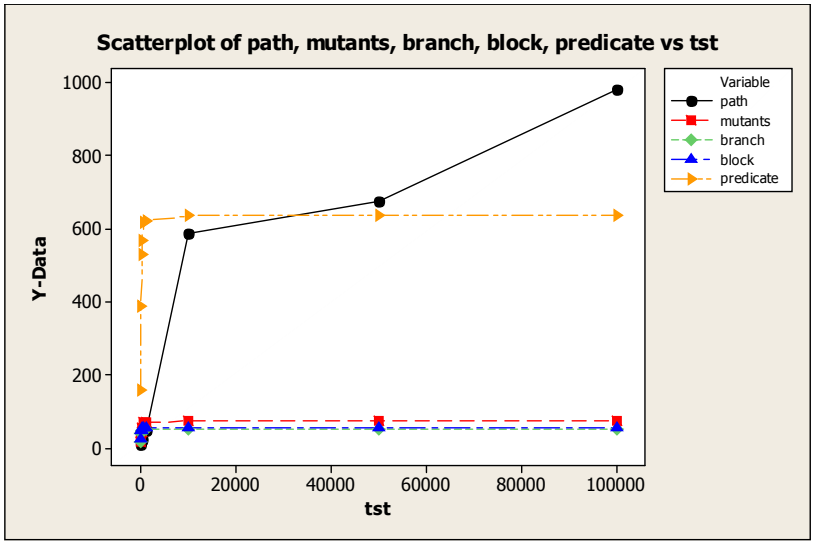


Figure 5.1: SGLIB coverage vs mutant graph - slist and dllist
fig:Note the x-axis tst/tsnum = testsuitesize x length of the testsuite]

Scatter plot for LIST: coverage vs mutants



Scatter plot for rbtree: coverage vs mutants

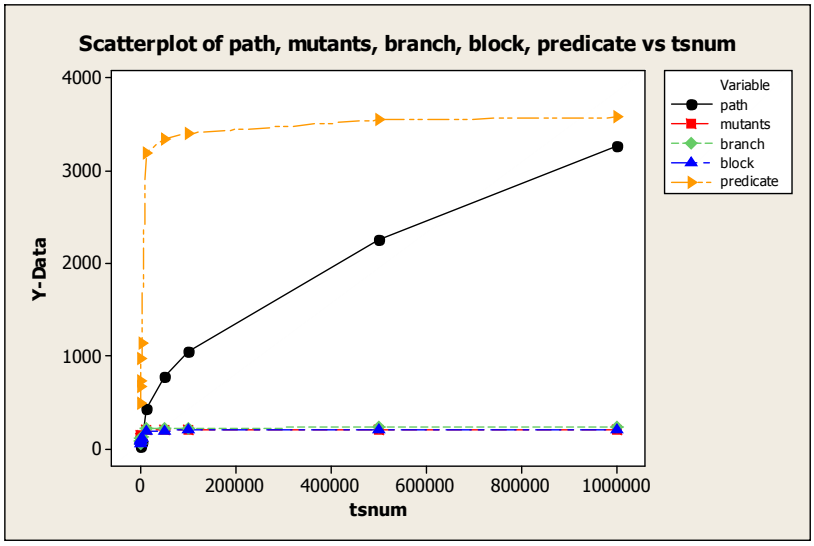


Figure 5.2: SGLIB coverage vs mutant graph - list and rbtree
fig:Note the x-axis tst/tsnum = testsuitesize x length of the testsuite]

Chapter 6: YAFFS2: Subject Software, Data collection and Analysis

6.1 Subject Software

YAFFS2 [15]stands for "yet another flash file system". It is designed specifically for use with NAND flash. It is used widely as an open-source flash file system for embedded use that serves as the default image format for Android. Its an open source project. This was our second subject program in hand.

6.2 Data collection

We began by generating, for each subject program, test suites of varying size till a stagnant pattern was seen. We varied sizes from 10 test cases to up to 100000 cases. All of the test suites were generated using random testing. It was not chosen randomly from a pool of test cases, but generated using random testing in itself to keep it more realistic and to remove any bias that maybe created when generating a test suite using an algorithm. Also feedback directed random testing was used for YAFFS2 since the parameter space is huge.

```
YAFFS2 test case generator algorithm: feedback-directed method
while(len < expected) {
randomly pick an api;
randomly generate parameters for it.
call this api with parameters and see if it succeed
if succeed
add api+ parameters to current test case
increase len;
}
```

For each test suite, we measured the block, branch, path and predicate coverage of the test suites and even coverage for individual test cases. This was done using CIL [13] to instrument the program and then the coverage percentage was calculated. For each subject program, we used a mutant generation program first used by Andrews et al. [3] to generate mutants for code written in C. To generate mutants from a source file, each line of code was considered in sequence and each of four classes of mutation operators were applied. The first step was to generate and compile the mutants, and to run mutants and faulty versions on the entire test pool. 1000 mutants were chosen randomly. We ran all mutants on all test cases, and recorded which test cases detected which mutants, in the sense of forcing the mutant to give different output from the oracle version. For each of the test suites we generated, we computed the mutation detection number and this was also calculated for the overall test suite to see how many mutants each test suite detected.

The data table below indicates the number of blocks, branches, paths and predicates covered over varying test suite sizes. The total number of LOC for YAFFS is 14k , the number of branches: 4189 , the total number of blocks is 3518. A length of 200 was chosen since a length lower than that was not showing a significant difference in result, considering the fact that for YAFFS we do have a larger code base , number of mutants than with SGLIB. And a length higher than 200 would not take a reasonable amount of time to finish the experiments. So by keeping the length at that rate we varied the test suite size to make some observations. For each of these test suites, we made a note of the number of mutants detected to make a correlation analysis.

Table 6.1: YAFFS2: Data Table

TestLen(TL)	TestCase(TC)	Block	Branch	Path	Predicate	Mutants	TL*TC
200	10	1376	1505	1027	32326	60	2000
200	50	1397	1535	2379	33104	64	10000
200	100	1397	1536	3457	33111	65	20000
200	500	1401	1540	9724	33270	68	100000
200	1000	1424	1570	16037	33855	75	200000

6.3 Analysis

For analyzing our experiment results we used the analysis framework similar to the paper by Namin et al. [11], since this made sense to our experiments as well. We were looking to find the correlation between these coverage and its effectiveness. Finding the correlation coefficient made sense and also finding the r^2 value to indicate its behavior in further depth. Experimental results were each first plotted on graphs as scatter plots to provide visualizations. After this graphs were created to plot the correlation between branch, block, predicate, path coverage with mutants. This was done in order to detect if there is any drastic variation between one coverage to the other with the same data and test set or if the results are more on a uniform pattern. Once these graphs were plotted, to go into more depth for analyzing the Pearson correlation coefficient, p-value were also calculated. This was done to indicate which coverage had a better correlation and then after this various regression models were constructed in order to obtain r^2 values for better understanding of the relationship. The regression equation that described the relation between mutant and the particular coverage for each test program gave a good insight of the correlation between these variables.

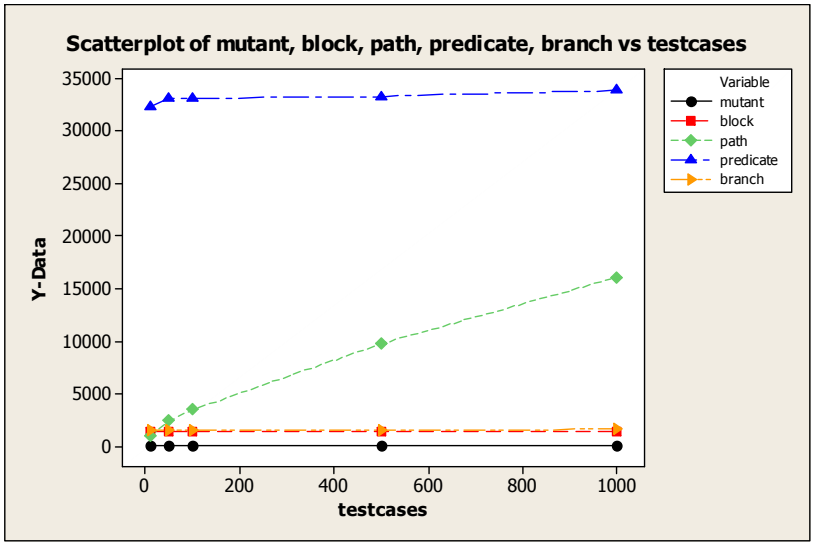
6.3.1 Visualization using graphs

To better conceptualize the data gathered as the result of the experiments they were plotted as a scatter plot. The figure below shows the correlation between each of the coverage criteria with mutant detection. The x axis indicating the test suite sizes and the y axis showing the coverage and mutant detection. The graph below indicates how each coverage relates to mutant detection. From the graph for YAFFS2 it can be noted that:

- Branch, block seem to influence mutant detection proportionally. Though the graph is not linear the curves and how the graph turns out looks similar. Path seems to have a higher correlation than predicate for YAFFS2.
- Apart from studying the behavior of coverage with mutant detection, if we look at just coverage techniques themselves, it can be noted that:Branch , block seem to correlate to each other. While to find the relation between path and predicate

Figure 6.1: YAFFS2: Graph

Coverage vs mutant



Correlation between different coverage criteria:

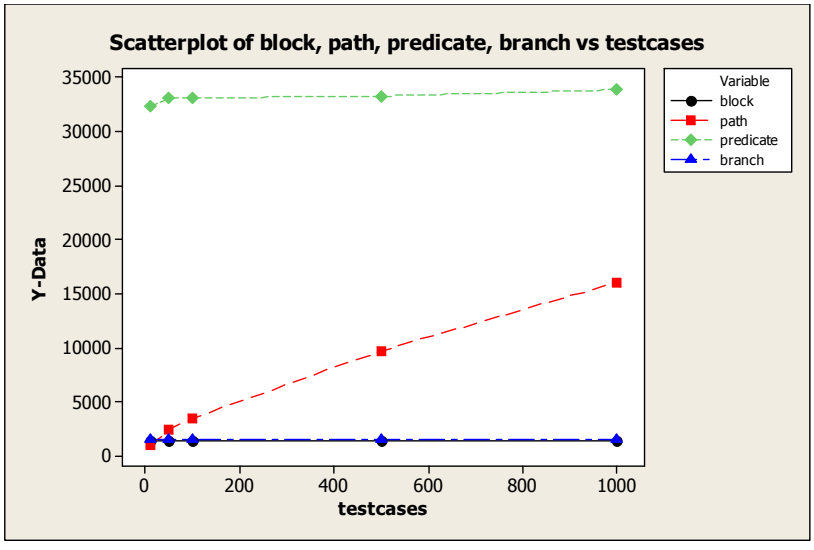


fig: Testcases(TC) of the x-axis= Number of TestCases in a TestSuite]

with each other and with each of branch and block we look more closely at the p-values.

To confirm these assumptions that we can draw from the graphs, we do further analysis which is described in more detail in the sections that follow.

6.3.2 Correlation of coverage and fault detection

With the data that has been plotted in graph there appears to be a positive correlation between coverage and mutant detection, Pearson correlation coefficients and p-values were calculated to affirm this. For every coverage curve vs mutant curve , the p-values and Pearson correlation coefficients were calculated. Below is the table for p-value and Pearson correlation coefficients for each of our subject programs and for each of the coverage vs mutant data. For Pearson coefficient correlation measure, the standard Guilford scale was used for verbal description; this scale describes correlation of under 0.4 as "low", 0.4 to 0.7 as "moderate", 0.7 to 0.9 as "high", and over 0.9 as "very high".

- Branch, block, path and predicate have a very high correlation with mutant detection since all of the values are above the 0.9 range.
- Strangely : Block vs predicate, block vs branch, branch vs predicate : these have a "Very high" correlation among themselves. While the correlation between path and any of these techniques are not very high even for the YAFFS2 data below. Thus it is clear that the correlation among the coverage techniques is high with these three coverage types. Though there is a correlation with path and these coverage techniques it is not as high as the others.

Table 6.2: YAFFS2: Coverage Mutant Correlation

	Pearson Coeff	P-value	R^2 value
Block	0.976	0.004	95.30%
Branch	0.968	0.007	93.70%
Path	0.971	0.006	94.30%
Predicate	0.957	0.011	91.60%
TC-num	0.962	0.009	92.5%

Table 6.3: YAFFS2: Coverage Techniques Correlation

	Pearson Coeff	P-value	R^2 value
Block vs branch	0.999	0	99.80%
Block vs Path	0.898	0.038	80.7%
Block vs Predicate	0.992	0.001	98.40%
Branch vs Path	0.882	0.048	77.9%
Branch vs Predicate	0.996	0	99.10%
Path vs Predicate	0.87	0.055	75.8%

6.3.3 Regression Models

Our correlation analysis showed us some interesting observations. In statistics, the coefficient of determination r^2 is used in the context of statistical models whose main purpose is the prediction of future outcomes on the basis of other related information. It is the proportion of variability in a data set that is accounted for by the statistical model. It provides a measure of how well future outcomes are likely to be predicted by the model. An r^2 of 1.0 indicates that the regression line perfectly fits the data. Using r^2s regression feature, various models were created for both the coverage relation with mutant and the correlation among coverage techniques. The third column in the tables below indicates the r^2 percentage. The regression line fits the data almost perfectly or very well in case of block , branch as with SGLIB as well. Though here path seems to perform slightly better than predicate. The values all above 90 or above 0.9 in decimal value. In regard to correlation between coverage techniques, the regression line for the block vs predicate, block vs branch and branch vs block are almost perfectly fitting the regression line.

Chapter 7: Discussion

The two subject programs were analyzed each as discussed below and this section combines the findings of each experiment in order to answer our initial research questions. The table below is divided into sections of coverage vs mutant and each row containing the values for each of our subject programs. Let us look into this in more detail. For the block coverage and mutant detection relation we can see a very consistent behavior across all our subject programs. Their P-values indicate that they have a very high correlation with mutant detection. This is with the four programs with SGLIB as well as YAFFS2. With branch coverage and predicate coverage we can see a similar consistent behavior. Another thing to note here is that Rbtree as we know contains recursive functions while the other subject programs don't. This does not still seem to change the coverage mutant relation. On the other hand path coverage does not behave consistently across our subject programs. It does not have a high correlation with mutant detection in 4 (SGLIB) out of 5 subject programs.

Table 7.1: Overall: Coverage vs Mutant

Subject Program	Pearson Coefficient	P-Value	R^2 percentage
Block vs mutant			
List	0.9	0.001	81%
Dllist	0.996	0	99.1%
Slist	0.973	0	94.8%
Rbtree	0.989	0	97.80%
Yaffs2	0.976	0.004	95.30%
Branch vs mutant			
List	0.923	0	85.20%
Dllist	0.995	0	99.10%
Slist	0.96	0	92.1%
Rbtree	0.99	0	97.90%
Yaffs2	0.968	0.007	93.70%
Path vs mutant			
List	0.609	0.082	37%
Dllist	0.606	0.084	36.70%
Slist	0.579	0.102	33.60%
Rbtree	0.68	0.03	46.30%
Yaffs2	0.971	0.006	94.30%
Predicate vs mutant			
List	0.949	0	90%
Dllist	0.99	0	98%
Slist	0.973	0	94.8%
Rbtree	0.968	0	93.8%
Yaffs2	0.957	0.011	91.60%
Numtc			
list	0.499	0.171	24.90%
Dllist	0.552	0.123	30.50%
slist	0.401	0.285	16.10%
Rbtree	0.503	0.139	25.30%
Yaffs2	0.962	0.009	92.50%

Two test cases in SGLIB with the same coverage were taken to see if they detect the same number of mutants. Here two test cases with the same coverage were considered (same branch , same block, predicate and path) and we saw the number of mutants it detected. It is to note that we did not consider the exact same branches covered just the overall coverage percentage to be the same between the two test cases and surprisingly even without the same branches, blocks or paths covered two test cases still appeared to determine almost same number of mutants on an average. The table indicates that. This might be because of a small subject program that we considered or it might be the small test case length itself. But we still found this behavior rather interesting and something that we look forward to explore in more detail in the future to see if the behavior is consistent with other programs.

Note the Pearson correlation coefficient for test cases with same coverage: Pearson correlation of two test cases with same block coverage : 0.999 (Its corresponding r^2 value is 99.8) Pearson correlation of two test cases with branch coverage: 0.999 (Its corresponding r^2 value is 99.8) Pearson correlation of two test cases with path coverage: 0.998 (Its corresponding r^2 value is 99.6) Pearson correlation of two test cases with predicate coverage: 0.998 (Its corresponding r^2 value is 99.6)

These results indicate that two test cases with similar coverage have a very high probability that they detect the same number of mutants.

Answering some of our original research questions after the analysis from both of our experiments.

- Under which coverage measures/coverage profiles are correlations in mutant detection ability high? Based on r^2 and p values, branch, predicate and block score well in four out of five subject programs

- Under which coverage measures/coverage profiles are correlations in mutant detection ability low? Path scored lowest everywhere in 4 out of 5 subject programs. Its rapid increase in coverage does not correlate with an equally rapid increase of the number of mutants after a certain point.

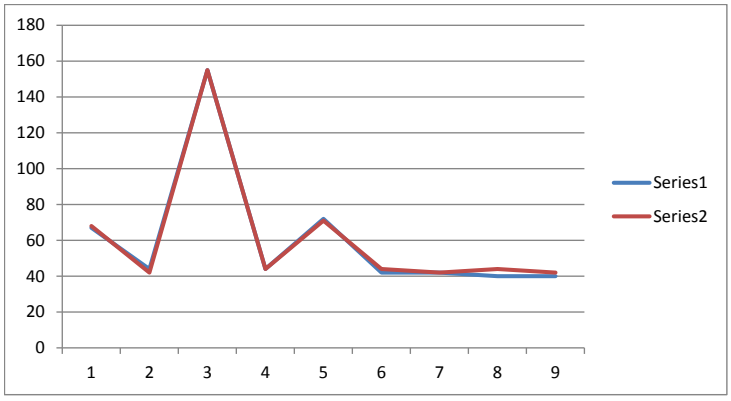
- If test suites A and B get the same coverage by criteria X, what's the correlation on criteria Y? The correlation between block-branch, block-predicate, branch-predicate is really high overall

SAME BRANCH		SAME BLOCK		SAME PATH		SAME PREDICATE	
TC A	TC B	TC A	TC B	TC A	TC B	TC A	TC B
67	68	68	67	68	68	67	69
44	42	44	42	149	168	158	159
155	155	155	155	63	66	62	70
44	44	44	44	44	44	40	41
72	71	72	71	44	42	43	44
42	44	42	44	42	44	42	44
42	42	42	42	42	42	42	42
40	44	40	44	40	44	40	44
40	42	40	42	40	42	40	42

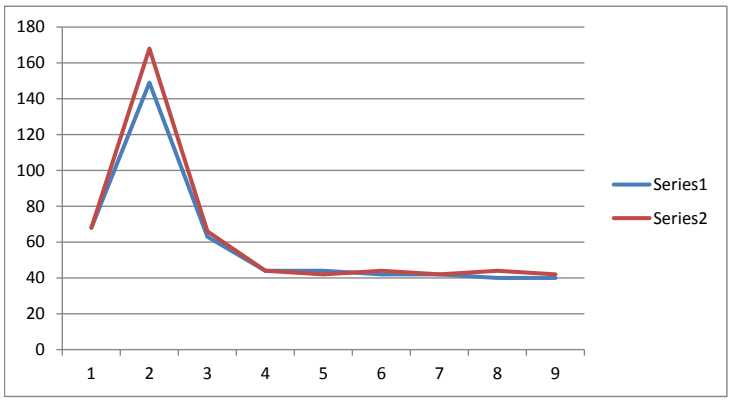
Note: Here TC A is Test Case A and TC is Test Case B

Figure 7.1: Comparison of two test cases with same coverage
fig:TC is for test case]

Figure 7.2: Same path and branch



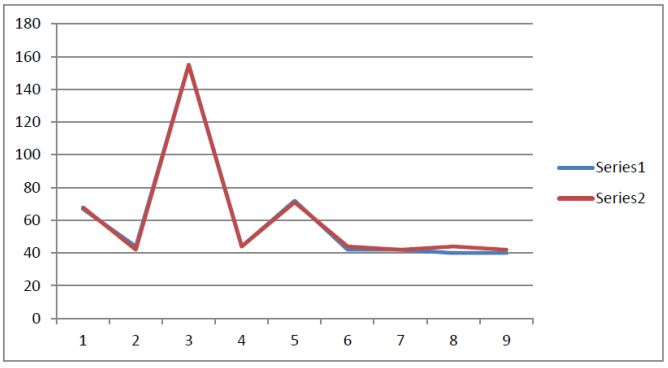
Same branch



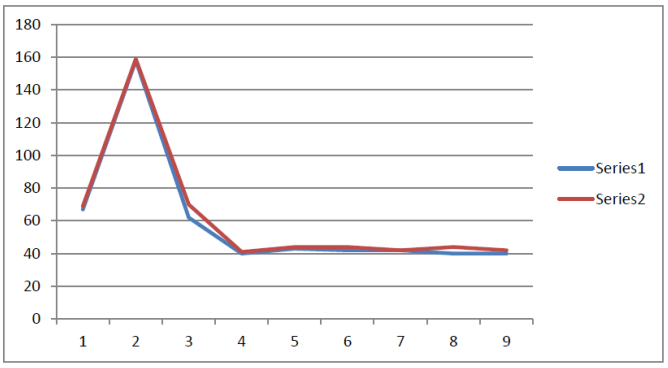
Same path

fig:"series 1= test case A" and "series 2= test case B"]

Figure 7.3: Same block and predicate



Same block



Same predicate

fig:"series 1= test case A" and "series 2= test case B"]

- Would two test cases A and B with the same coverage (branch, block, path, predicate) detect the same number of mutants? This did surprisingly prove to almost do predict the same number of mutants in most of the cases as discussed above. But this is more tougher to generalize since due to time constraints we could only consider SGLIB results.

Chapter 8: Threats to Validity

Threats to internal validity include the possibility of corrupted data or incorrect procedures; The most serious threats is to external validity. Our set of programs and test suites, while fairly large and representative of the C real world programs. In particular, we examined a large number of data structures. The results though might not extend to other languages and object oriented languages might behave differently in this aspect. We do plan to do future work in this aspect in order to extend our finds in this study. Mutation analysis is a well known method used by a lot of researchers in replacing real world faults. But this is sensitive to external threats caused by some influential factors including mutation operators, test suite size, and programming languages [12]. We also worked with randomly generated test suites as a way of making a more broader conclusions possible about the influence of various coverage techniques on effectiveness. However, if we had restricted our attention to particular algorithms for building test suites, our results may very well have been different, and while less general, may have applied directly to real world methods of test suite construction. Construct validity is primarily threatened by mistakes in our implementation of coverage info gathering; we doubt that any of these were sufficient to radically change any results.

Chapter 9: Conclusion and Future work

Our empirical study of coverage to answer our research questions did give some interesting results. We performed controlled experiments aimed at understanding the influence of different kinds of coverage on test suite effectiveness. Our results indicate that the very popular branch, block correlate better with fault detection than path coverage in a test suite with multiple test cases. Another interesting thing we noted in our experiment is that predicate performed better than path coverage in 4 out of 5 subject programs and almost did as well as the popular block and branch coverage. Hence designing a test suite with a higher percentage of branch and block not only seem to be feasible but also appear to correlate better with fault detection. We have used subject programs that are moderately sized. Block and branch coverage also seemed to have a high correlation among themselves, meaning that an x% of block coverage would in most cases ensure a particular y% of branch coverage as well based on the regression analysis and a regression equation and this also extended to predicate coverage. This was not true in the case of path coverage. Therefore knowing that there is x% branch coverage in a program we can to some extent predict the amount of block/predicate coverage that the program might have with the knowledge of branch but this did not extend to path. The correlation between path and the other coverage criteria did not show up as high.

For our future work, we would like to run similar experiments on programs from a broader range of programming languages like Java. In our present study we focussed on the four popular coverage techniques being branch, block, path and predicate. Another interesting way to expand this further would be to add a broader coverage techniques as well. This could very well be studied for def-use, prime-path, PCT for gathering more interesting results. We could vary our form of testing. We used random testing with feedback for our experiment with YAFFS2 and random testing for SGLIB but using adaptive testing or some other testing method could also prove to be interesting. We can see if the results remain consistent or vary from our present set up as we expand our study. Further studies in this area would be helpful for future software testers in optimizing the test suite creation or minimization. Another interesting thing that would

be worth exploring in the future is in the related field of statistical debugging - finding a way to predict when to use which coverage traces for statistical debugging.

Bibliography

[1] Groce Alex. This is a slide from a class@ONLINE, June 2009.

[2] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge Univ Pr, 2008.

[3] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments?[software testing]. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411. IEEE, 2005.

[4] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, 2006.

[5] S. Berner, R. Weber, and R.K. Keller. Enhancing software testing by judicious use of code coverage information. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 612–620. IEEE, 2007.

[6] X. Cai and M.R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *ACM SIGSOFT Software Engineering Notes*, pages 1–7. ACM, 2005.

[7] P.G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *ACM SIGSOFT Software Engineering Notes*, pages 153–162. ACM, 1998.

[8] P.G. Frankl and S.N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *Software Engineering, IEEE Transactions on*, pages 774–787, 1993.

[9] R. Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994.

[10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, 1994.

[11] A.S. Namin and J.H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 57–68. ACM, 2009.

[12] A.S. Namin and S. Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 342–352. ACM, 2011.

[13] G. Necula, S. McPeak, S. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction*, pages 209–265. Springer, 2002.

[14] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[15] A. One. Yaffs: Yet another flash file system, 2002.

[16] C. Pacheco, S.K. Lahiri, M.D. Ernst, and T. Ball. Feedback-directed random test generation. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 75–84. Ieee, 2007.

[17] G. Rothermel, M.J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 34–43. IEEE, 1998.

[18] G. Rothermel, M.J. Harrold, J. Von Ronne, and C. Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[19] M. Vittek, P. Borovansky, and P.E. Moreau. A simple generic library for c. *Reuse of Off-the-Shelf Components*, pages 423–426, 2006.

[20] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 230–238. IEEE, 1994.

[21] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pages 41–41. IEEE, 1995.

[22] L. Zhang, S.S. Hou, J.J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 435–444. ACM, 2010.