CHAPTER 1

INTRODUCTION

1.1 Software Testing

In the software development life cycle software testing plays a very important role. Software Testing is a process which is aimed at evaluating the attribute or capability program or product/system and determining that meet its quality. Testing is an investigation conducted to provide customers with information about the quality of product/service under test. Testing also provide an objective, independent view of software to allow the business to appreciate and understand the risk of the software implementation.

Testing technique include the process of executing a program or application used to finding software bugs. It involves identifying bug, error, defect in a software without correcting it. Normally professionals with a quality assurance background are involved in bug's identification.

Software Testing is an activity in software development. It is an investigation performed against software to provide information about the quality of the software to stakeholders. There are variety of testing techniques are available to test the software product. Among these, White box testing is one of the software testing method in which the internal structure or design or implementation of the item being tested is known to the tester. It tests at very low levels. Unit testing is an example of White box testing. Mutation testing comes under Unit Testing.

Using the testing we can reduce the bugs given by the software application. Test cases are usually created manually which is an error prone and time consuming process. Path testing is an important testing method of white box testing. Path testing exercises only a basis set of execution paths through a program.

The basis path testing uses the Cyclomatic complexity and the mathematical analysis of control flow graphs to guide the software testing process. In flow graph, nodes represent statement or expressions and edges represent transfer of controls between the nodes

Test criteria apply some engineering rules to source or other software artifacts to create software requirements. A Test Requirement (TR) is a specific element that a test case must satisfy or cover the entire necessary path. Comparison of three graph coverage criteria- Edge Coverage (EC), Edge Pair Coverage (EPC), Prime Path Coverage (PPC) is performed. The most common two experimental comparison techniques are:

1. To count the number of actual Test Requirements-an estimation of cost
2. To count the number of faults found by test sets that satisfy the requirements-an estimation of effectiveness.

1.2 Objective

1. To compute the graph coverage criteria (EC, EPC, PPC) using Control Flow Graph.
2. To compare the graph coverage criteria
3. To evaluate the cost and effectiveness of the graph coverage criteria

1.3 Scope of the work

1. In this existing approaches they use test generation techniques, test set pruning and random testing to cover the faults.
2. In this, Mutation score and test requirements are used to compute the effectiveness and cost.

1.4 Flow diagram

The flow diagram for the software testing is given in fig 1.1.

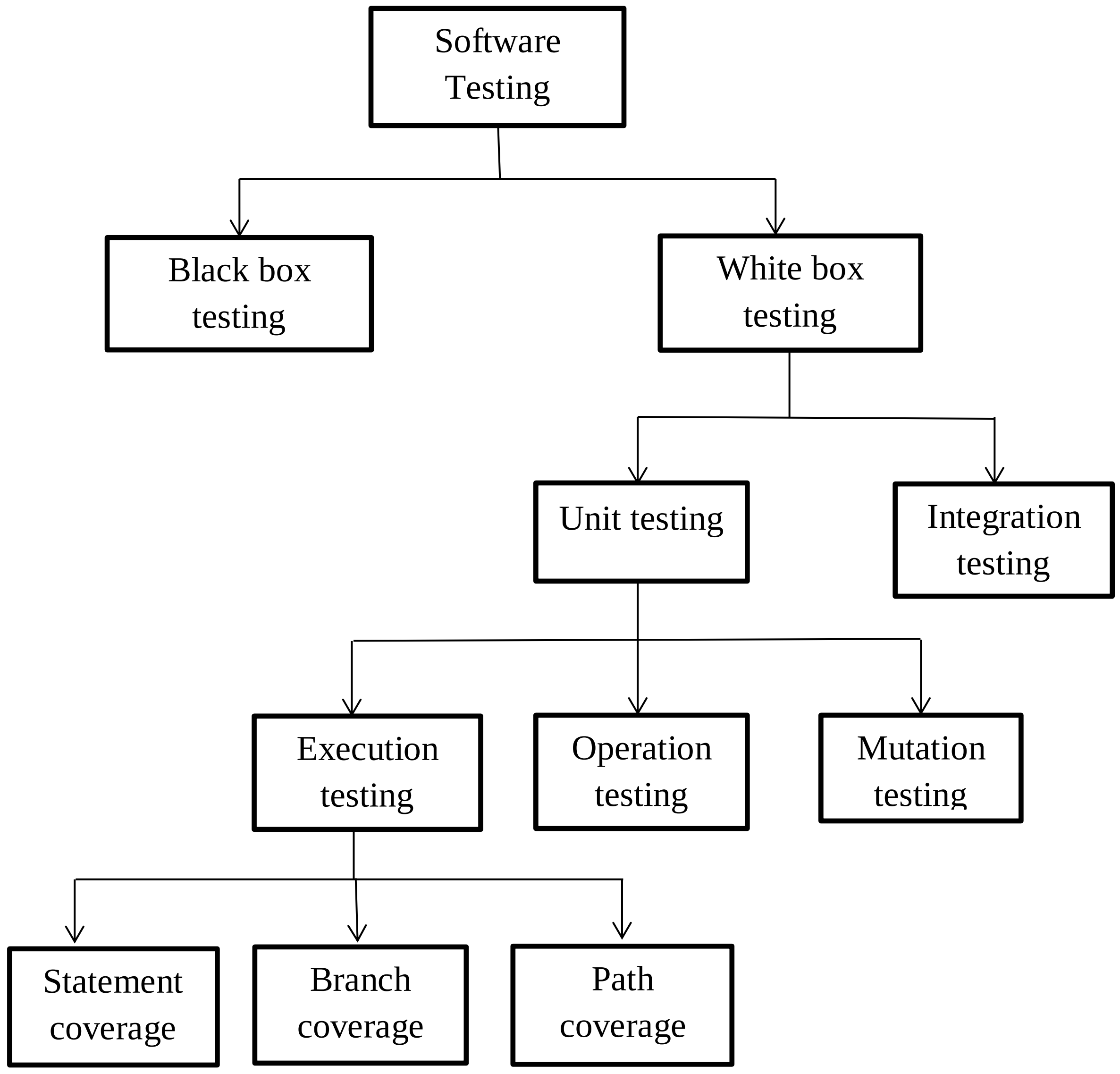


Fig 1.1 Flow diagram for software testing

Software testing has two types: black box testing and white box testing. Each would have come across several types of testing. Unit testing and integration testing are the types of white box testing. Mutation testing comes under the unit testing. Each type of testing has its own features, advantages and disadvantages as well.

1.5 Mutation testing

Mutation Testing is a type of software testing where mutate certain statements in the source and check if the test cases are able to find the errors. Mutants are used as proxies for faults. The goal of Mutation Testing is to assess the quality of the test cases which should be robust enough to fail mutant code.

As this method involves source code changes, it is not at all applicable for black box testing. It brings a whole new kind of errors to the developer’s attention. It is the most powerful method to detect hidden defects, which might be impossible to identify using the conventional testing techniques.

1.6 Organization of the chapter

Chapters are Organized are as follows. Chapter 2 describes the literature survey of the existing papers. Chapter 3 describes the domain of our project. Chapter 4 describes the data flow diagram and its levels. Chapter 5 describes the proposed technique of the given project. Chapter 6 describes the experimentation and analysis. Chapter 7 describes the result and discussion. Chapter 8 describes the Conclusion.

CHAPTER 2

LITERATURE SURVEY

2.1 Growing a Reduced Set of Mutation Operators

The paper [2] proposed a framework for reduced set of mutation operators. The effectiveness of mutation testing depends heavily on the specific mutation operators used. The goal is to find a small set of mutants that still selects a test set that can kill all or most of the mutants that would be generated if all operators were used. But there is a limitation present in this work. The tests were generated by hand and using different tests or a different tester might change the results. The test generation was extremely time consuming, so it would be impractical to create more than one test set.

The term complete set of operators to refer to using all mutation operators available. This is, of course, relative to the mutation system being used, and can vary quite a bit. The quality of a reduced set of operators can be measured relative to the complete set of operators. Specifically, tests are designed to kill all mutants created by a reduced set, then run against the complete set. The resulting mutation score is called the effectiveness score, and can be considered a measure of the effectiveness of using just those operators. Thus, effectiveness score is relative to a particular mutation system and a
Particular test set.

	STRATEGY S1			
Op	SSDL	OASN	ORBN	OAAN
MS	0.952	0.983	0.997	1.0
Mutants	38	60	84	128
Equiv	2	3	5	7
Tests	3	5	7	8

Table 2.1 Strategy 1 applied to program

	STRATEGY S2				
Op	SSDL	OARN	OASN	ORBN	VTWD
MS	0.952	0.979	0.983	0.997	1.0
Mutants	38	104	126	150	212
Equiv	2	2	3	5	7
Tests	3	4	5	7	8

Table 2.2 Strategy 2 applied to program

	STRATEGY S3				
Op	SSDL	Ccsr	OASN	VTWD	ORBN
MS	0.952	0.979	0.983	0.987	1.0
Mutants	38	93	115	177	201
Equiv	2	2	3	5	7
Tests	3	4	5	7	8

Table 2.3 Strategy 3 applied to program

When a test set is adequate relative to a set of mutation operators, those operators select those tests. We call a set of mutation operators highly effective if the tests they select have a very high effectiveness score. The effectiveness score to range from 99% to 100% on the subject programs. The effectiveness of mutation testing depends heavily on the specific mutation operators used. Good mutation operators will help testers design very strong tests, but poor mutation operators will lead to weak tests. This approach has the disadvantage of losing some tests, possibly very valuable tests.

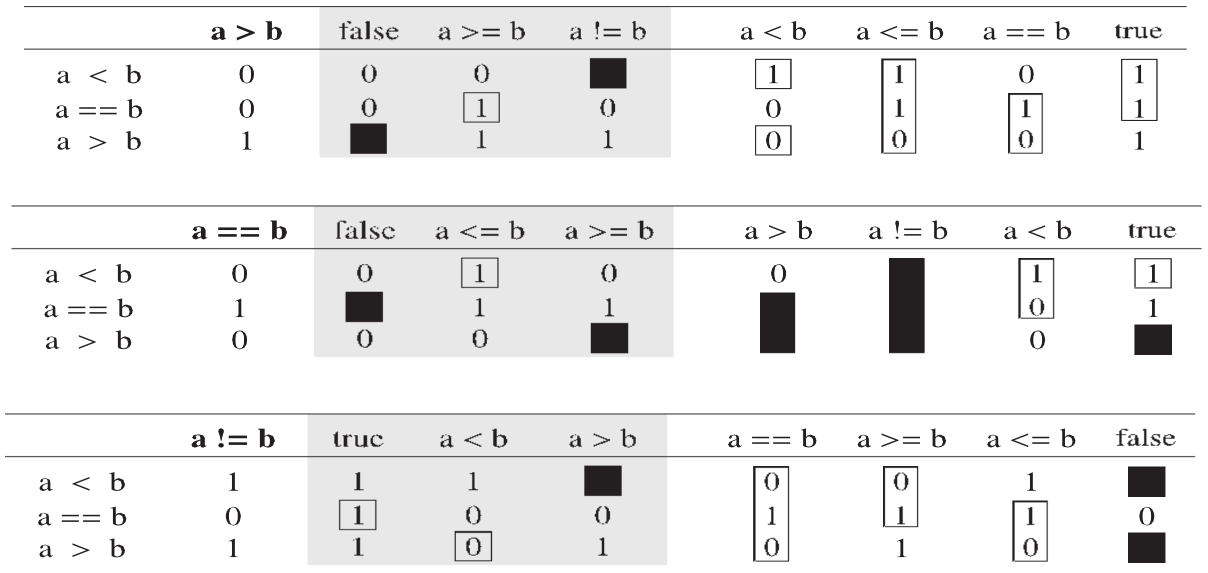
The cost of mutation testing also depends on the mutation operators. Using more mutation operators may increase the effectiveness of the test set, but can also create a large number of mutants that need to be executed, analyzed, and killed. The results confirm previous studies that indicate that with a large set of mutation operators such as with C, there is not a single way to select the best operators. This is an important contribution.

2.2 Efficient mutation analysis using non-redundant mutation operators

This paper [4] investigates how redundant mutants affect the efficiency and accuracy of mutation analysis. Focusing on three well-known mutation operators, namely the COR, UOI, and ROR mutation operators, it makes the following contributions.

In this method first develops a sub-sumption hierarchy and provides a sufficient set of non-redundant mutations for the COR, UOI, and ROR mutation operators. The method shows that 4 out of 10 COR mutations are sufficient and that the COR mutation operator subsumes the UOI mutation operator for Boolean expressions. The method also confirms prior results and shows that 3 out of 7 ROR mutations are sufficient.

Interval	Original version	Non-redundant ROR mutations	Subsumed ROR mutations



	a > b	false	a >= b	a != b	a < b	a <= b	a == b	true
a < b	0	0	0	■	$\boxed{1}$	$\boxed{1}$	0	$\boxed{1}$
a == b	0	0	$\boxed{1}$	0	0	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$
a > b	1	■	1	1	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	1

	a == b	false	a <= b	a >= b	a > b	a != b	a < b	true
a < b	0	0	$\boxed{1}$	0	0	■	$\boxed{1}$	$\boxed{1}$
a == b	1	■	1	1	■	■	$\boxed{0}$	1
a > b	0	0	0	■	■	■	0	■

	a != b	true	a < b	a > b	a == b	a >= b	a <= b	false
a < b	1	1	1	■	$\boxed{0}$	$\boxed{0}$	1	■
a == b	0	$\boxed{1}$	0	0	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	0
a > b	1	1	$\boxed{0}$	1	$\boxed{0}$	1	$\boxed{0}$	■

Table 2.4 Mutants applied to program

The empirical study in this paper demonstrates how redundant mutants affect the efficiency and accuracy of mutation analysis. Besides showing how prevalent COR, UOI, and ROR mutants are for real-world programmes, the study reveals that eliminating redundant mutants decreases the total mutation analysis run time by more than 20%. Moreover, the study shows that the inclusion of redundant mutants misleadingly overestimates the mutation score. Comparing mutation coverage with statement coverage, branch coverage, and the mutation score, the study reveals that mutation coverage has a very strong correlation with statement and branch coverage but only a moderate correlation with the mutation score.

Given the results reported in the empirical study, areas for future work include the determination of sufficient sets of non-redundant mutations for other mutation operators, such as the arithmetic operator replacement. The notion of determining a minimal distance between a mutation and its original version should be transferable to other mutation operators as well. In addition, this approach investigates intra-operator redundancies, but mutants derived from different mutation opera-tors might also exhibit redundancies. Therefore, the investigation of inter-operator redundancies is another area for future work

2.3 Establishing Theoretical Minimal Sets of Mutants

This paper [6] has presented a way to identify precisely how many mutants are needed in the context of a given test set. The size of this set is much smaller than delivered by current best-practice approaches to mutation. There is considerable scope for new approaches to mutation analysis that consider only relatively few mutants while at the same time thoroughly testing the underlying artifact.

Algorithm 1: Mutant set minimization

```
// Input: Mutant set M; Score function S
// Output: A minimal mutant set
remove live mutants from S
remove duplicate columns from S
minSet = remaining columns in S
subsumed = dynamically subsumed
mutants in minSet
return (minSet - subsumed);
```

Algorithm 1 describes first, live mutants are removed. Next, in distinguished mutants are removed. Finally, dynamically subsumed mutants are removed.

Mutation score is widely used in the literature to evaluate the quality of an approach to generating test cases. This approach has caused some disquiet in the research community due to the presence of redundant mutants. The results of this method suggest a different methodology for evaluating testing approaches. Rather than evaluating a given approach against all mutants generated by some set of operators, we propose that, in addition, the approach should be evaluated against a minimal set of mutants. Any approach as strong as the chosen mutation operators will achieve 100% in either case. Weaker approaches can still be compared against criteria such as random selection, but using a minimal set of mutants for comparison removes the problem of redundant mutants from the evaluation.

The minimization approach developed in this method focused on mutation analysis specifically to address the problem of redundant mutants. However, since the approach uses only the black-box score function, the model can also be applied to test requirements from any other coverage criterion, e.g., statement coverage, branch coverage, dataflow coverage, and so on. The eventual goal of this line of research is to make mutation testing cost-effective enough to use in practice. The dynamic subsumption approach to minimizing the number of mutants demonstrates that it is, indeed, possible to reduce the number of mutants needed to a very small number. We hope the theoretical structure presented in this paper will lead to practical applications to dramatically reduce the number of mutants generated by actual mutation systems.

2.4 An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing

This paper [10] described a comparing the effectiveness of randomly generated test sets that are adequate according to the all-edges, all-uses, and null test data adequacy criteria. Unlike previous experiments, this experiment was designed to allow for statistically meaningful comparison of the error-detecting ability adequacy criteria. It involved generating large numbers of test sets for each subject program, determining which test sets were adequate according to each criterion, and determining the proportion of adequate test sets that exposed at least one error.

The data was analysed rigorously, using well-established statistical techniques. The results of the experiment are presented below, organized into three subchapters corresponding to each of the three types of questions we asked initially:

1) Are those test sets that satisfy criterion C1 more likely to detect an error than those that satisfy C2?

2) For a given test set size, are those test sets that satisfy criterion C1 more likely to detect an error than those that satisfy C2?

3) How does the likelihood that the test set detects an error depend on the extent to which a test set satisfies the criterion?

In each of these the first group of questions we posed was whether C1 is more effective than C2 for each subject and for each pair of criteria. For five of the nine subjects, all-uses was significantly more effective than all-edges; for six subjects, all-uses was significantly more effective than the null criterion; for five subjects all-edges was more effective than null. Closer examination of the data showed that in several of the cases in which all-uses did well; it actually did very well, appearing to guarantee error detection. We also compared test sets that partially satisfied all-uses to those that partially satisfied all edges.

In six subjects, test sets that covered all but two definition-use associations were more effective than test sets that covered all but two edges. Thus, test sets that cover all (or almost all) dua's are sometimes, but not always, more likely to expose an error than those that cover all (almost all) edges.

The second group of questions limited attention to test sets of the same or similar size. All-uses adequate test sets appeared to be more effective than all-edges adequate sets of similar size in four of the nine subjects. For the same four subjects, all-uses adequate test sets appeared to be more effective than null adequate sets of similar size. In contrast, all edges adequate sets were not more effective than null adequate sets of similar size for any of the subjects. This indicates that in those cases where all-edges was more effective than the null criterion, the increased effectiveness was due primarily to the fact that all-edges test sets were typically larger than null-adequate test sets. In most of the cases where all-uses was more effective than all-edges or than the null criterion, the increased effectiveness appears to be due to other factors, such as the way the criterion concentrates failure-causing-inputs into subdomains.

The third group of questions investigated whether the probability that a test set exposes an error depends on the size of the test set and the proportion of definition-uses associations or edges covered. This is an important question because it is not uncommon for testers and testing researchers to assume implicitly that confidence in the correctness of a program should be proportional to the extent to which an adequacy criterion is satisfied. Logistic regression showed that in four of the nine subject programs the error-exposing ability of test sets tended to increase as these test sets covered more definition-use associations.

2.5 An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-uses and Prime Path Coverage

This paper [5] presents experimental data to try to help compare cost benefit trade-offs among four test criteria. Formal test criteria are used to choose specific test values to test with. Informally, a test criterion is a goal or stopping rule for testing. Test criteria make it more likely that testers will find faults in the program and provide greater assurance that the software is of high quality and reliability. Test requirements are specific elements of software artifacts that must be satisfied or covered. An example test requirement for statement coverage is “Execute statement.

A test criterion is a rule or collection of rules that, given a software artifact, imposes test requirements that must be met by tests. That is, the criterion describes the test requirements for the artifact in a complete and unambiguous manner. Although test criteria can be based on lots of software artifacts, including formal specifications, requirements, and design notations, most unit testing is based on the program source.

Many test criteria are based on graphs, including three of the four criteria compared in this paper. Numerous graph based criteria have been proposed, most notably on control flow analysis and data flow analysis Researchers have published theoretical studies and empirical comparisons among graph-based test criteria.

Figure [2.1] shows an example C method, its annotated control flow graph, and the edge-pairs and prime paths from the graph. Nodes 4 and 6 are final nodes, corresponding to the return statements. Node 2 is introduced to capture the for loop; it has no executable statements. The edge coverage, edge-pairs, prime paths are listed. Edge coverage Edge pairs and prime paths are given as sequences of nodes.

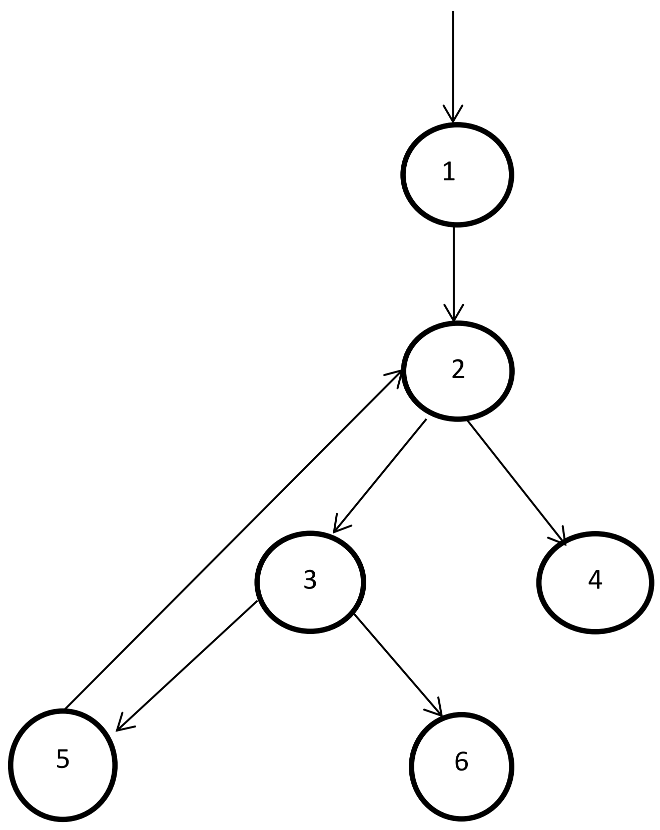


Fig 2.1 Example of CFG

V (G) = E-N+2

=7-6+2

=3

Path=[1, 2, 4], [1, 2, 3, 6], [1, 2, 3, 5]

Edge coverage = [1, 2], [2, 3], [2, 4], [3, 5], [3, 6], [5, 2]

Edge pair coverage = [1, 2, 3], [1, 2, 4], [2, 3, 5], [2, 3, 6], [3, 5, 2], [5, 2, 3], [5, 2, 4]

Prime path coverage = [1, 2, 4], [1, 2, 3, 6], [1, 2, 3, 5], [2, 3, 5, 2], [3, 5, 2, 3], [3, 5, 2, 4], [5, 2, 3, 6], [5, 2, 3, 5]

The faults that mutation did not catch can be instructive. It may be possible to design additional mutation operators that can detect these faults.

Of course this study has several limitations. As in all studies that use software as subjects, external validity is limited by the number of subjects and the fact that we have no way of knowing whether they are representative of the general population. Most of the classes were reasonably simple, and must leave it to a future replicated study to see if the results would be similar for larger and more complicated classes.

2.6 An experiment for evaluating effectiveness and efficiency of coverage criteria for software testing

In this paper [3] found that the predicate coverage criterion demonstrated best effectiveness but at more cost, i.e., it required more testing efforts than the other two. On the other hand, the block coverage criterion took least testing effort, but at lower effectiveness than the other two. The branch coverage criterion performed in between the two others in terms of the effectiveness and the testing-effort requirements. The efficiency of the block coverage criterion was found to be on the higher side but with greater variability. On the other hand, the efficiency for the branch coverage criterion was found to be slightly less than that of the block coverage but was relatively consistent.

The efficiency for the predicate test suites was observed to be lower than the other two. The overall results suggest that the branch test suites are likely to perform consistently in terms of the effectiveness and the efficiency whereas their effort requirements in terms of test suite size are smaller than that of the predicate test suites but larger than the block test suites. Based on our investigation for choosing a suitable criterion to test a given program, we observed that the branch coverage criterion is the best choice for getting better results with moderate testing efforts. The effectiveness can be further improved by considering predicate coverage for the methods containing composite conditionals but with increased effort requirements.

Systematic testing approaches are usually based on the models of the programs and tests can be generated by covering some specific aspects (i.e., elements) of those models. These models may be CFGs, state models, or formal specifications. The proposed approach facilitates comparison for all such coverage criteria for which test coverage information can be obtained. We can effectively compare coverage criteria not only in a family of a particular chapter of their classification, but also across their classification as well. With the help of proper tool support, for instance, we can use the proposed approach for comparing criteria in data-flow testing stream or in mutation testing. It is also interesting to see the performance of the same test cases when they are used in different contexts, as they are used in multiple coverage-adequate test suites for different criteria under study.

Figure 2.2 describes a schematic diagram for the experimental framework. The testing process for each test-program used in the experiment is carried out in two phases:

Phase-I: In this phase, a test-pool is constructed for a test-program which is sufficiently large enough to generate multiple coverage-adequate test suites for the coverage criteria under study. To meet this objective, existing test suites of the test-programs can be procured. Additionally, further test cases testing different aspects of the program are designed and included in the test-pool. A coverage tool is used to ensure the “sufficiently-large” condition of the test-pool. Coverage information for each test case in the test-pool for each coverage criterion in the study is obtained.

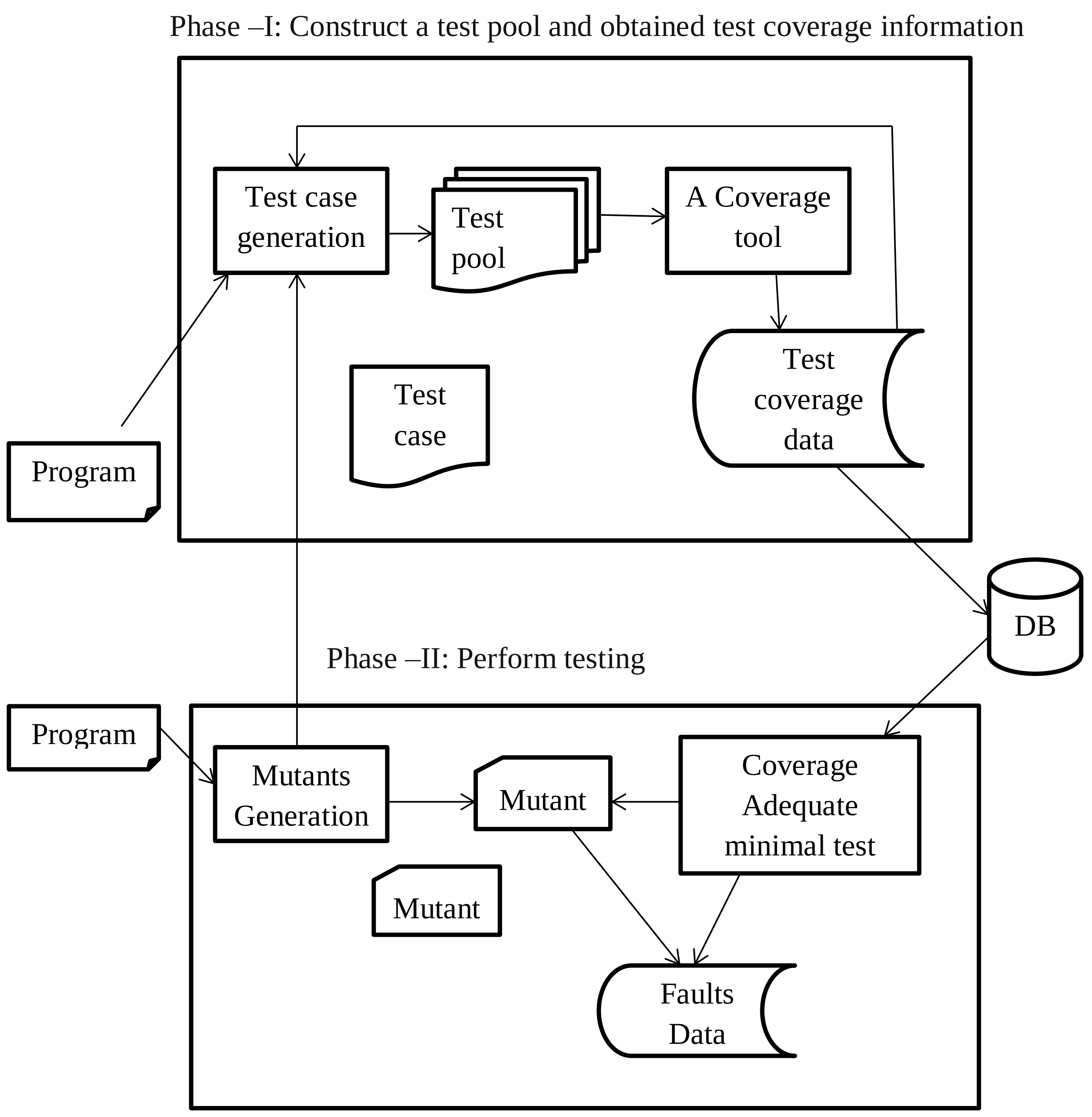


Fig 2.2 General purpose for experimental comparison of coverage criteria

Phase-II: In this phase, we generate a set of mutants for the test-program, and if required, generate additional test cases and include them in the test-pool to see that the test-pool covers the faults (i.e., the mutants) structure as well. Using the test-coverage information of each test case in the test-pool, we construct multiple coverage-adequate minimal test suites for each coverage criterion under study. The program's mutants then tested by the generated test suites so obtained and killed mutants information for each test suite is recorded.

The most important, also difficult to control, source of variability in a typical software testing experiment is the human participation (their roles and skills), which at the same time, makes it even more difficult to repeat (or replicate) such experiments. Use of automated testing in the proposed approach leads to much better control in the experiment process, and as desired, the experiment can be easily repeated and replicated with different set of faults, test-pool, or the test programs. This reduces the cost of the experimentation and provides excellent opportunities for the researchers to capitalize on their findings.

2.7 Limitations of the existing system

From the above all existing techniques they create the mutants using proteum tool but they are not efficient. It is used for measuring the effectiveness of selective mutation. Equivalent mutants are harmful to the runtime of the mutation analysis process because they can’t be detected by any test. Mutation analysis, may be prohibitively time consuming and computationally expensive. No adequate test set exist. In code coverage, there is a problem of finding all faults in a program. Efficiency has not been sufficiently addressed. In the existing system they use the control flow graph and it gives graph coverage criteria. Mutants are seeded by hand. Effectiveness is analysed using mutation score. Cost is analysed using infeasible Test Requirements. PPC has more number of Infeasible Test Requirements and it leads to increase the cost. So, there is a need for other technique that reduces the cost.

CHAPTER-3

MUTATION TESTING

3.1 Mutation Testing

A mutation is nothing but a single syntactic change that is made to the program statement. Each mutant program should differ from the original program by one mutation.

Execution process steps

Faults are introduced into the source code of the program [Table 3.1] by creating many versions called mutants. Each mutant should contain a single fault, and the goal is to cause the mutant version to fail which demonstrates the effectiveness of the test cases. Test cases are applied to the original program and also to the mutant program. A Test Case should be adequate, and it is tweaked to detect faults in a program. Compare the results of an original and mutant program.

If the original program and mutant programs generate the different output, then that the mutant is killed by the test case. Hence the test case is good enough to detect the change between the original and the mutant program. If the original program and mutant program generate the same output, Mutant is kept alive. Such mutants are called equivalent mutants. In such cases, more effective test cases need to be created that kill all mutants.

3.2 Mutation Operators:

A handle to seed faults in a test program in some specific context.

To insert the mutant’s one of the subject programs “check it” is given below.

Program

```
#include <stdio.h>
int main()
{
 int number, originalNumber, remainder, result = 0;
 printf("Enter a three digit integer: ");
 scanf("%d", &number);
 originalNumber = number;
 while (originalNumber != 0)
{
 remainder = originalNumber%10;
 result += remainder*remainder*remainder;
 originalNumber /= 10;
 }
 if(result == number)
printf("%d is an Armstrong number.",number);
else
printf("%d is not an Armstrong number.",number);
return 0;
}
```

Examples of Mutation Operators:

i. Statement deletion.
ii. Statement duplication or insertion

e.g., goto fail.

iii. Replacement of Boolean sub expressions with true or false.

iv. Replacement of arithmetic operations with others

e.g., + with *, - with /

v. Replacement of variables with others from the same scope.

vi. Replacement of some Boolean relations with others

e.g., > with >=, == and <=

vii. Return statement replacement

viii. Removing of else part in the if-else statement

ix. Adding or replacement of operators

x. Data Modification for the variables

xi. Modification of data types in the program

ORIGINAL PROGRAM	MUTANT PROGRAM

while (originalNumber != 0)	while (originalNumber >= 0)
	while (originalNumber <= 0)
	while (originalNumber = = 0)
	while (originalNumber > 0)
	while (originalNumber < 0)

Number originalNumber remainder result = 0	num, val, n number, onum rem, r res, R, ans
Else	Remove “else” part

Printf ("%d is an Armstrong number." ,number);
printf("%d is not an Armstrong number.",number);
	Printf ("%d is TRUE ",number);
	printf("%d is FALSE",number);

Table 3.1 Mutants seeded to the original program

Table 3.1 describes the possible mutants seeded in to the original program, it is possible to seed 104 mutants using the mutation operators. Mutation testing has long been used to compare test sets and criteria by using mutants as proxies for faults. The minimal set of mutants has no redundant mutants. Minimal mutant sets can be defined globally, for all possible inputs. Thus, the study in this paper uses both the full set of mutants as well as a minimal set.

3.3 Automation of Mutation Testing:

Mutation testing is extremely time consuming and complicated to execute manually. To speed up the process, it is advisable to go for automation tools. Automation tools reduce the cost of testing as well.

List of tools available -

1. Stryker
2. PIT Testing

3.4 Types of Mutation Testing

In Software Engineering, Mutation testing could be fundamentally categorized into 3 types.

Statement Mutation - developer cut and pastes a part of a code of which the outcome may be a removal of some lines

Value Mutation- values of primary parameters are modified

Decision Mutation- control statements are to be changed

3.5 Mutation Score:

The mutation score is defined as the percentage of killed mutants with the total number of mutants.

Full Mutation Score is calculated by the formula:

$$\text{Full Mutation Score}=\frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$

Minimal Mutation Score is calculated by the formula:

$$\text{Minimal Mutation Score}=\frac{\text{Number of minimal mutants killed}}{\text{Total number of mutants}}$$

3.6 Advantages of Mutation Testing:

1. It is a powerful approach to attain high coverage of the source program.
2. This testing is capable comprehensively testing the mutant program.
3. Mutation testing brings a good level of error detection to the software developer.
4. This method uncovers ambiguities in the source code and has the capacity to detect all the faults in the program.
5. Customers are benefited from this testing by getting a most reliable and stable system.

3.7 Disadvantages of Mutation Testing:

1. Mutation testing is extremely costly and time-consuming since there are many mutant programs that need to be generated.
2. Since its time consuming, it's fair to say that this testing cannot be done without an automation tool.
3. Each mutation will have the same number of test cases than that of the original program. So, a large number of mutant programs may need to be tested against the original test suite.
4. As this method involves source code changes, it is not at all applicable for Black Box Testing.

CHAPTER 4

SYSTEM ANALYSIS

4.1 Data Flow Diagram

A data flow diagram (DFD) is also known as data flow graph or bubble charts. A DFD serves the purpose of clarifying system requirements and identifying major transformations. A DFD is a graphical representation of the “flow” of data through an information system, modelling its process aspects. It is an important modelling tool that allows us to picture a system as a network of functional processes.

A DFD is often used as a preliminary step to create an overview of the system which can be later elaborated. DFD can also be used for visualization of data processing (structure design).

A DFD shows what kind of information will be input to output from the system, where the data will come from and go to, and where the data will be store. It doesn’t show information about the timing of process or information about whether processors will operate in sequence or parallel.

▢ → Input / Output

⬭ → Functions / Process

⟶ → Sequence of actions

Level 0

At the beginning the software programs are given as input. From this we can generate the Graph coverage criteria using CFG and from the generated EC, EPC, PPC compare the cost and effectiveness. The context level diagram is shown in fig 4.1.

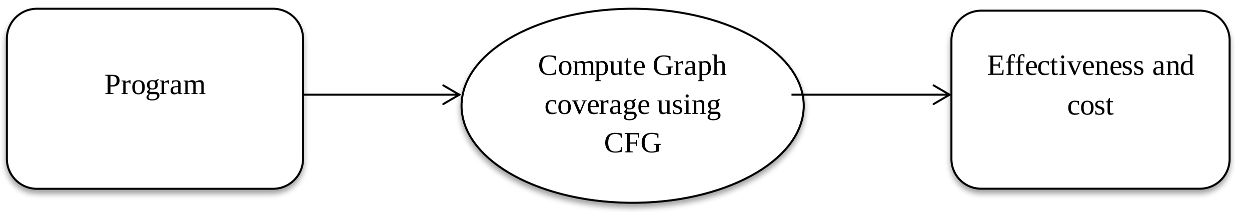


Fig 4.1 Context Level Diagram

Level 1

Software programs are given as an input. Unit testing is one of the important methods in the White Box Testing. By using mutation testing we can seed the mutants to the given program. The number of paths can be calculated using the Cyclomatic Complexity. From this number of paths, we extract the EC, EPC, and PPC. Cost and Effectiveness is computed using TR and Mutation score. Data flow diagram for path testing is shown in fig 4.2.

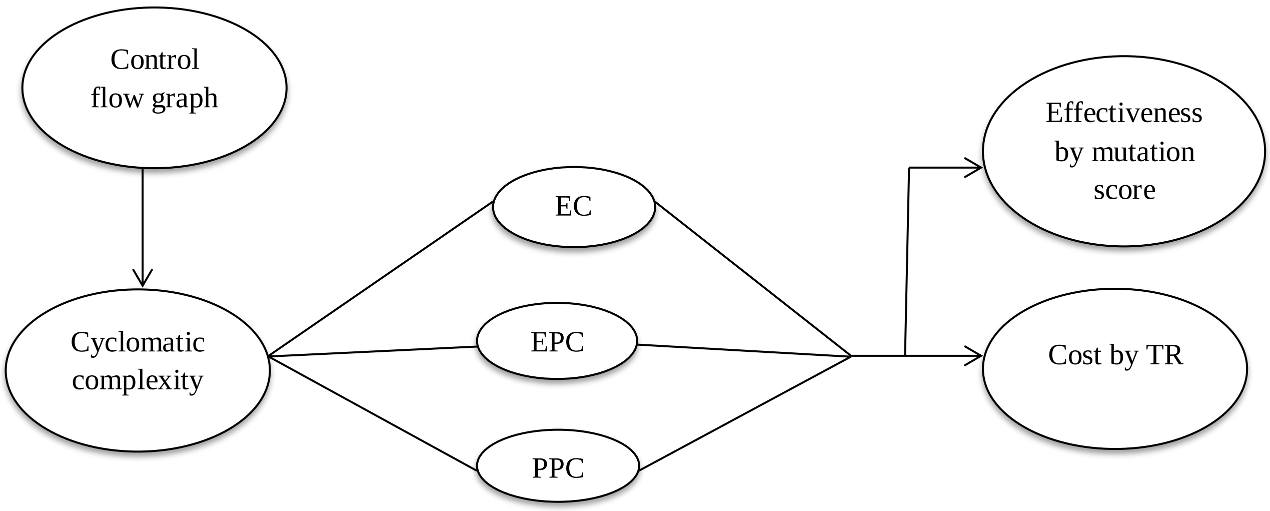


Fig 4.2 Level 1: Mutation Testing Diagram

4.2 Architecture diagram

The architectural diagram for our approach is given as follow in fig 4.5.

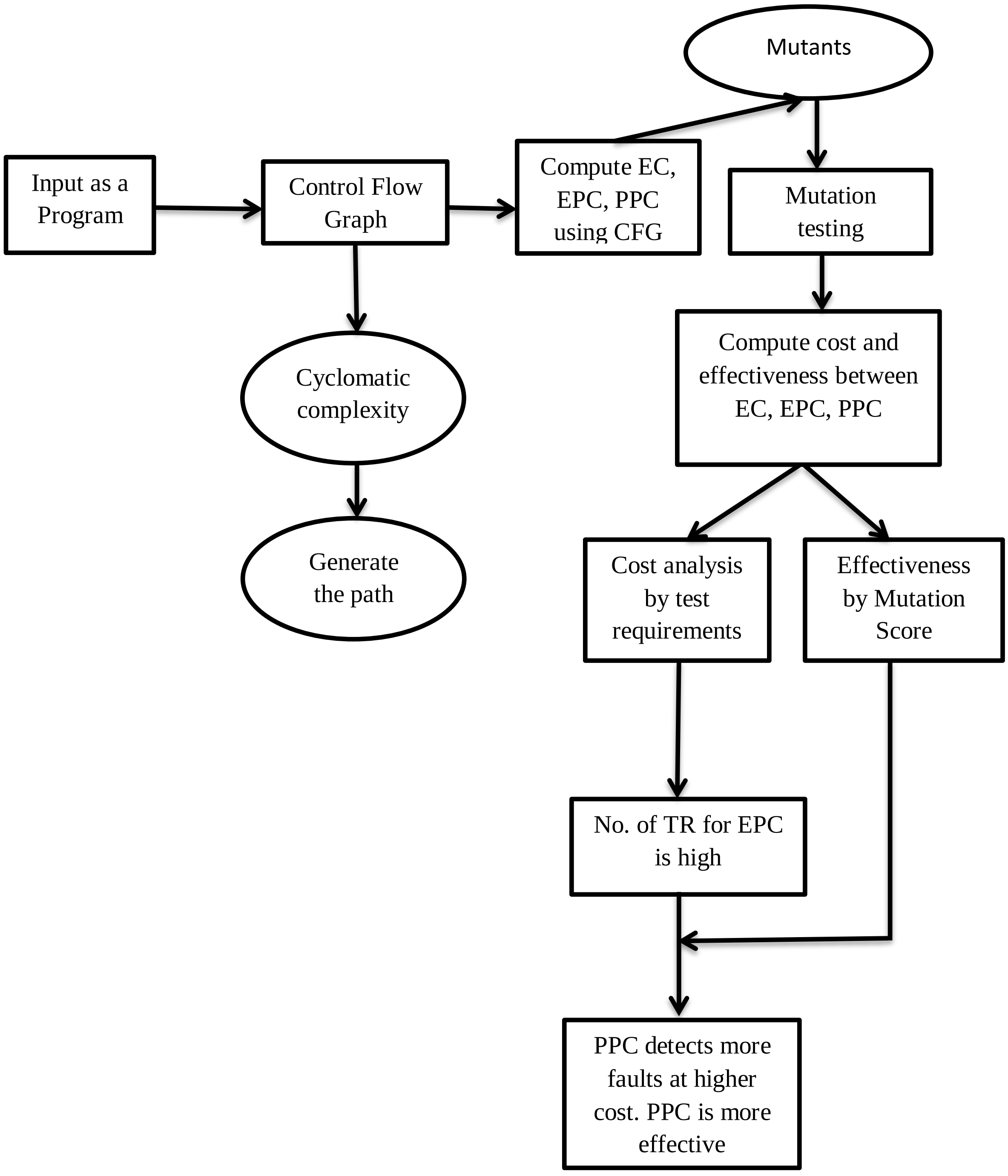

Mutants
Input as a Program
Control Flow Graph
Compute EC, EPC, PPC using CFG
Mutation testing
Cyclomatic complexity
Compute cost and effectiveness between EC, EPC, PPC
Generate the path
Cost analysis by test requirements
Effectiveness by Mutation Score
No. of TR for EPC is high
PPC detects more faults at higher cost. PPC is more effective

Fig 4.3 Architectural diagram

Software program is given as an input to the given program. Control Flow Graph is generated for the given program. From this control flow graph the various paths are generated using cyclomatic complexity. Then compute the graph coverage criteria: EC, EPC, PPC using CFG. After find out the EC, EPC, PPC seed the mutants to the program using mutation operators. Mutation testing is performed. Then finally cost and effectiveness are compared and evaluated.

Effectiveness is analyzed using Full Mutation Score and Minimal Mutation Score. Effectiveness is achieved between 95% to 99%. Cost is analyzed using Test Requirements. But the number of Infeasible TRs is higher in PPC.PPC detects more faults at higher cost. Among the graph coverage criteria, PPC is more effective.

CHAPTER 5

PROPOSED TECHNIQUE

In our proposed project, consider 10 subject C programs are taken as input. They are

1. Bounded Queue
2. Cal
3. Check it
4. Count positive
5. Digit reverser
6. Find last
7. Gaussian
8. Num zero
9. Odd or pos
10. Power

The control flow graph is generated for a subject program “check it”. From the control flow graph, path is generated using Cyclomatic complexity. Then EC, EPC, PPC are generated from the control flow graph. Compare the effectiveness and cost among the graph coverage criteria. Mutants are seeded into the program. The Efficiency can be measured by Mutation score. Cost is analysed using Test Requirements.

5.1 Test Cases

A Specific executable test that examines all aspects including inputs and outputs of a system and then provides a detailed description of the steps that should be taken, the results that should be achieved, and other elements that should be identified.

5.2 Path Testing

Path testing is an approach to testing where to ensure that every path through a program has been executed at least once. Here normally use a dynamic analyzer tool or test coverage analyzer to check that all of the code in a program has been executed.

Path testing is a structural testing strategy that uses programs control flow as a model. Path testing techniques are based on judiciously selecting a set of paths to the program. The set of paths chosen is used to achieve a certain measure of testing thoroughness. In Path testing, having the various approach to find the independent paths and then use the graphical representation for the flow of program.

5.3 Control flow Graph

A control flow graph (CFG) in computer science is representation, using graph notation, of all paths that might be traversed through a program during its execution.

In a control flow graph, each node in the graph represents a basic block, that is a straight line piece of code without any jumps or jump targets start a block, and jumps end a block. Directed edges are used to represents jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enter into the flow graph, and the exit block, through which all control flow leaves.

For this CFG, calculate the Cyclomatic complexity to find the number of paths can be obtained for the program. It is a software metric (measurement), used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code.

Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program. By McCabe, V (G) = E-N+2(P)

E – Number of Edges

N – Number of Nodes

P – Number of Sub-functions

5.4 Graph coverage criteria

Graph coverage criteria understand the concepts of simple paths and prime paths [Fig 5.1]. Graphs are the most commonly used structure for testing.

Graphs can come from many sources

i. Control flow graphs
ii. Design structure
iii. FSMs and state charts
iv. Use cases

Paths in Graphs

Path:

A sequence of nodes – [n1, n2, …, nM]. Each pair of nodes is an edge.

Length:

The number of edges. A single node is a path of length 0.

Tests:

Tests usually are intended to "cover" the graph in some way.

Test Path:

A path that starts at an initial node and ends at a final node.

Test paths represent execution of test cases

i. Some test paths can be executed by many tests
ii. Some test paths cannot be executed by any tests.

Test Requirements:

Test requirement is a specific element of a software artifact that a test case must satisfy or cover.

Infeasible Test Requirements:

i. An infeasible test requirement cannot be satisfied

ii. Unreachable statement (dead code)

iii. A subpath that can only be executed if a contradiction occurs (*X* > *0* and *X* < *0*)

TYPES OF GRAPH COVERAGE CRITERIA

1. Structural Coverage Criteria

2. Data flow Coverage Criteria

Structural Coverage Criteria: Defined on a graph just in terms of nodes and edges

Data Flow Coverage Criteria: Requires a graph to be annotated with references to variables.

$$\begin{tikzcd}
 & 1 \arrow[dl] \arrow[dr] & \\
2 \arrow[dr] & & 3 \arrow[dl] \\
 & 4 &
\end{tikzcd}$$

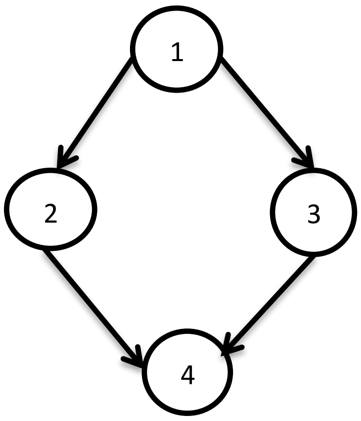


Fig 5.1.Example of graph coverage criteria

Figure 5.1 shows a control flow graph. It is used to calculate the graph coverage criteria EC, EPC, PPC for a sample program. Nodes represent a piece of code (1, 2, 3, and 4) and edges represent a flow from one node to another. The Test Requirements for the three structural criteria are: Edge Coverage, Edge Pair Coverage, Prime Path Coverage.

EDGE COVERAGE (EC):

TR contains each reachable path of length up to 1.

Thus the pair (1, 2) has length 1. Similarly the pairs (1, 3), (2, 4), (3, 4) has the same length 1. *So*

EC :{ (1, 2), (1, 3), (2, 4), (3, 4)}

EDGE PAIR COVERAGE (EPC):

TR contains each reachable path of length up to 2.

Thus the pair (1, 2, 4) has length 2. Similarly the pair (1, 3, 4) has the same length 2. *So*

EPC: {[1, 2, 4], [1, 3, 4] }

Prime Path:

A simple path that does not appear as a proper sub path of any other simple path. A simple, elegant and finite criterion that requires loops to be executed as well as skipped.

PRIME PATH COVERAGE (PPC):

TR contains each prime path

PPC: {[1, 2, 4], [1,3,4] }

Table 5.1 summarizes the subject programs used in this experiment. The table shows the number of C programs, their units, total LOC, number of mutants generated by all mutation operators, the number of minimal mutants, number of equivalent mutants, and number of test cases used in this experiment. Equivalent mutants were determined by hand analysis. Equivalent mutants are also called as non-killed mutants. Note that the minimal mutants contain no equivalent mutants, since they are eliminated as part of finding the minimal set. Test cases were determined by hand analysis.

The subject programs vary in size from one to 6 functions, totally 20 functions from 10 C programs and totally 165 lines of code (LOC). These programs were extracted from text books and the software testing literature.

Program	Units	LOC	Mut	Min Mut	Equiv	Test Case
Bounded queue	6	49	1121	7	99	13
Cal	1	18	891	7	71	8
Check it	1	9	104	7	3	9
Count positive	1	9	151	3	9	5
Digit reverser	1	17	496	3	43	5
Find last	1	10	198	5	17	6
Gaussian	6	23	1086	17	19	21
Num zero	1	10	151	3	17	5
Odd Or Pos	1	9	361	5	71	7
Power	1	11	268	5	12	9
Total	20	165	4827	59	361	88
Min	1	9	104	3	3	5
Max	6	49	1121	17	99	21
Average	2.0	16.5	482.7	5.9	36.1	8.8

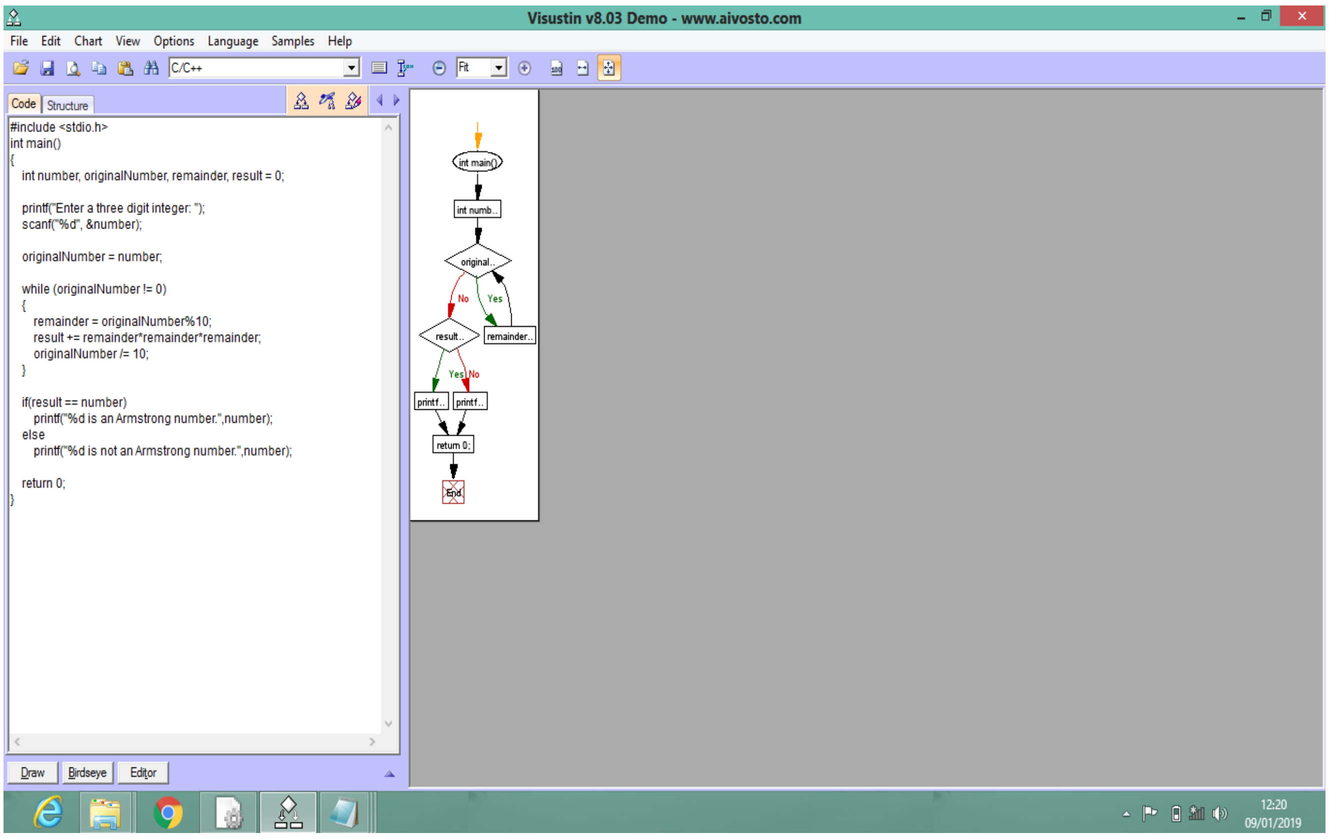
Table 5.1: Subject programs used in this experiment

Applying all mutant operators to all subject programs resulted in 4827 mutants. The number of mutants yielded from each subject ranged from 104 in check it to 1121 in bounded queue. By contrast, when considering minimal mutant sets, only a total of 59 mutants are needed. The smallest minimal sets have three mutants (count Positive, Digit Reverser and num Zero). The largest minimal set has 17 mutants (Gaussian).

CHAPTER 6

EXPERIMENTATION AND ANALYSIS

Consider one of our subject program "check it" given as the input to the control flow graph and convert it into the data flow graph. The independent paths are created for the check it program. The cyclomatic complexity is used to find the path. From the generated paths EC, EPC, PPC are computed. The cost and effectiveness are the metrics used. The program and graph for check it from visustin and manual representation is given as follow given in fig 6.1 & 6.2.



```
#include <stdio.h>
int main()
{
   int number, originalNumber, remainder, result = 0;

   printf("Enter a three digit integer: ");
   scanf("%d", &number);

   originalNumber = number;

   while (originalNumber != 0)
   {
      remainder = originalNumber%10;
      result += remainder*remainder*remainder;
      originalNumber /= 10;
   }

   if(result == number)
      printf("%d is an Armstrong number.",number);
   else
      printf("%d is not an Armstrong number.",number);

   return 0;
}
```

Fig 6.1 Control flow graph from visustin

Figure 6.1 represents a Control Flow Graph. CFG is obtained from the visustin tool. This tool is used to draw data flow graph from the C, C++, Java languages.

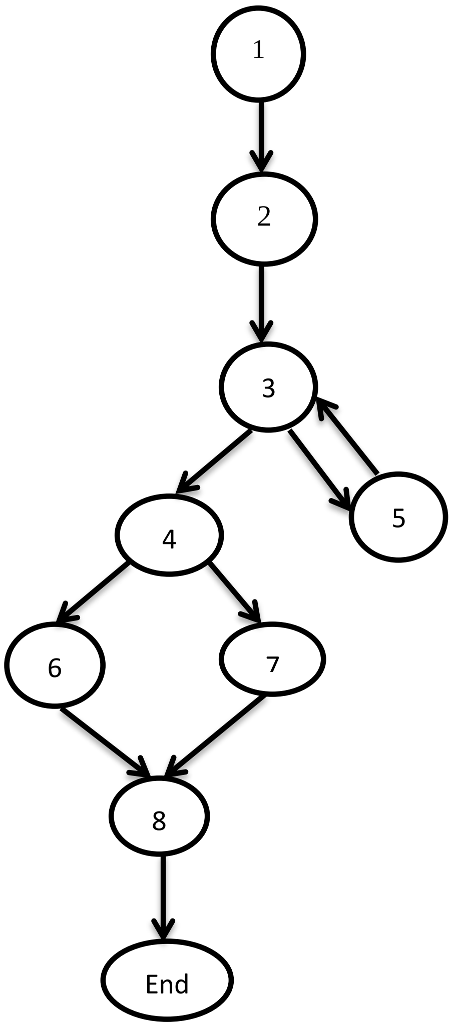


Fig 6.2 Control flow graph

Figure 6.2 represents the CFG. The control flow graph is drawn for the given software program. From this control flow graph by removing the statements the data flow graph is generated. And then independent paths are generated. Paths are generated by calculating cyclomatic complexity.

Cyclomatic Complexity V (G) = E-N+2

=10-9+2

=3

PATH

(1, 2, 3, 4, 6, 8, end)

(1, 2, 3, 4, 7, 8, end)

(1, 2, 3, 5, 3, 4, 7, 8, end)

Graph coverage criteria

EC

(1, 2)

(2, 3)

(3, 4)

(3, 5)

(5, 3)

(4, 6)

(4, 7)

(6, 8)

(7, 8)

(8, end)

EPC

(1, 2, 3)

(2, 3, 4)

(3, 4, 6)

(3, 5, 3)

(4, 6, 8)

(4, 7, 8)

(2, 3, 5)

(3, 4, 7)

(6, 8, end)

(7, 8, end)

PPC

(3, 5, 3)

(1, 2, 3, 5)

(5, 3, 4, 6, 8, end)

(5, 3, 4, 7, 8, end)

(1, 2, 3, 4, 6, 8, end)

(1, 2, 3, 4, 7, 8, end)

6.1 FULL MUTATION SCORE

Consider one of our subjects program “count positive”. In this program there are 151 mutants. PPC killed 148 mutants and EPC killed 146 mutants. Full mutation score has more number of redundant mutants. Test cases used by PPC are 4.100 and EPC uses 3.300 test cases.

Full Mutation Score is calculated by the formula:

$$\text{Full Mutation Score}=\frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$

$$=\frac{148}{151}$$

$$=0.980$$

Thus the full mutation score achieved by PPC is 98 %

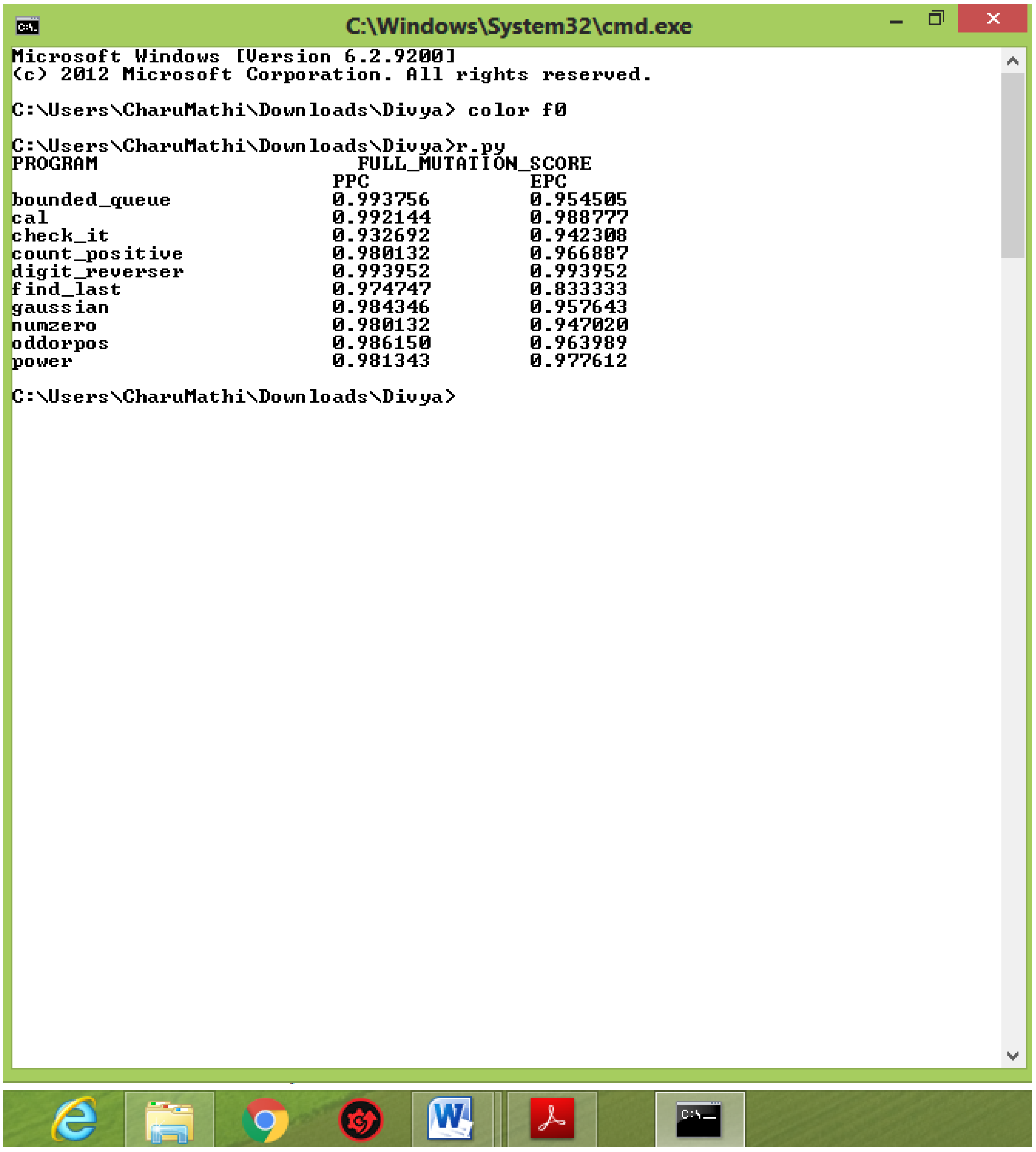
$$\text{Full Mutation Score}=\frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$

$$=\frac{146}{151}$$

$$=0.966$$

Thus the full mutation score achieved by EPC is 96 %

Similarly full mutation score is calculated for the remaining nine subject programs and the output is obtained. Thus the 10 subject programs are implemented and output is given as screenshot Figure 6.3. Effectiveness is compared using full mutation score. PPC is more effective than EPC.



```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Users\CharuMathi\Downloads\Divya> color f0

C:\Users\CharuMathi\Downloads\Divya>r.py
PROGRAM                          FULL_MUTATION_SCORE
                               PPC              EPC
bounded_queue                  0.993756         0.954505
cal                            0.992144         0.988777
check_it                       0.932692         0.942308
count_positive                 0.980132         0.966887
digit_reverser                 0.993952         0.993952
find_last                      0.974747         0.833333
gaussian                       0.984346         0.957643
numzero                        0.980132         0.947020
oddorpos                       0.986150         0.963989
power                          0.981343         0.977612

C:\Users\CharuMathi\Downloads\Divya>
```

Fig 6.3 Implementation to find out effectiveness using full mutation score

6.2 MINIMAL MUTATION SCORE

Minimal mutants are those mutants that contain no equivalent mutants.

Consider one of our subjects program “count positive”. In this program there are 151 mutants. PPC killed 133 minimal mutants and EPC killed 116 minimal mutants. EC killed 71 minimal mutants. Minimal mutation score has no redundant mutants. Test cases used by PPC are 4.100, EPC uses 3.300 test cases and Test cases used by EC are 1.900

Minimal Mutation Score is calculated by the formula:

$$\text{Minimal Mutation Score}=\frac{\text{Number of minimal mutants killed}}{\text{Total number of mutants}}$$

$$=\frac{133}{151}$$

$$=0.880$$

Thus the minimal mutation score achieved by PPC is 88 %

$$\text{Minimal Mutation Score}=\frac{\text{Number of minimal mutants killed}}{\text{Total number of mutants}}$$

$$=\frac{116}{151}$$

$$=0.768$$

Thus the minimal mutation score achieved by EPC is 76 %

$$\text{Minimal Mutation Score}=\frac{\text{Number of minimal mutants killed}}{\text{Total number of mutants}}$$

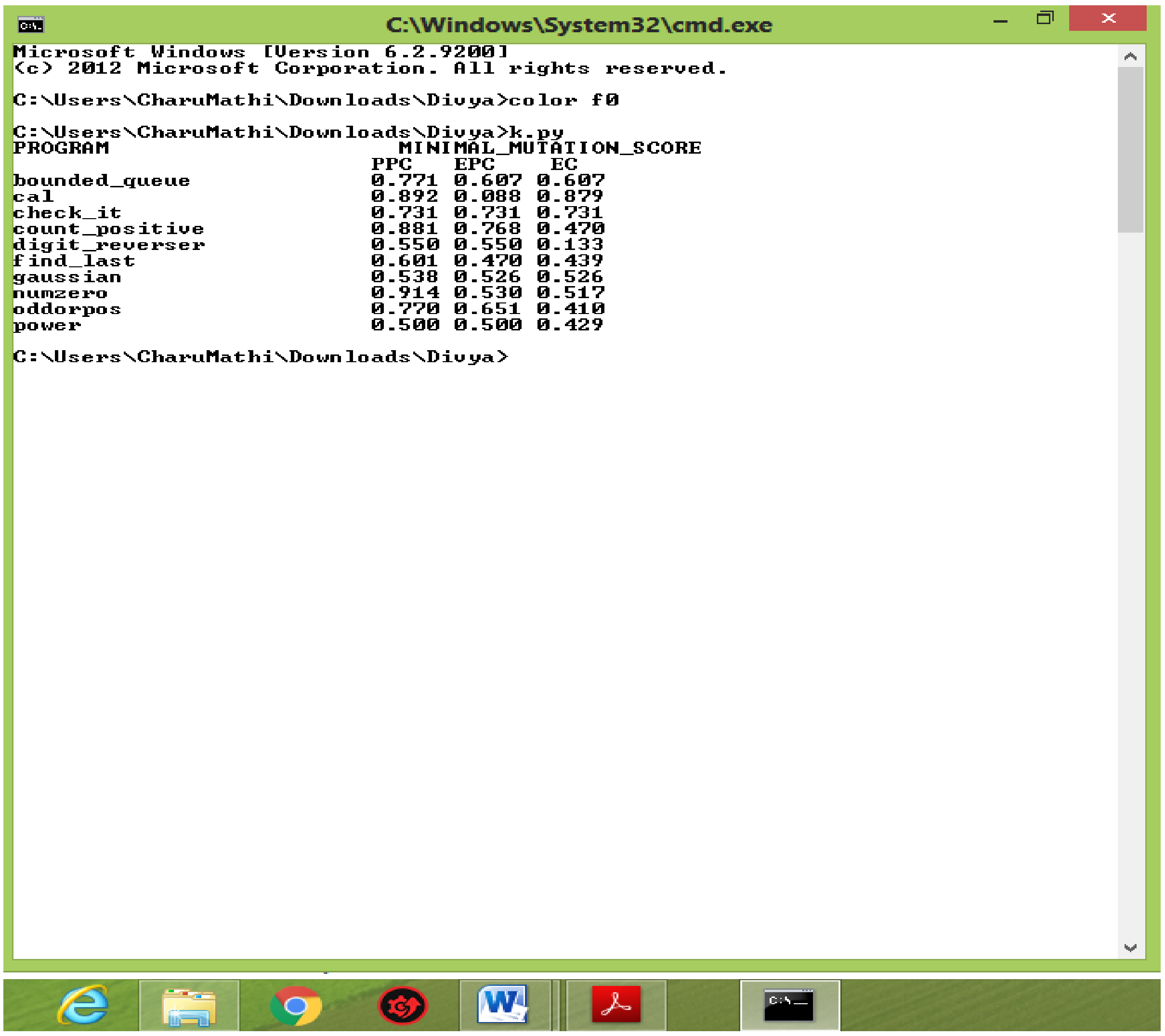
$$=\frac{71}{151}$$

$$=0.470$$

Thus the minimal mutation score achieved by EC is 47 %

Similarly minimal mutation score is calculated for the remaining nine subject programs and the output is obtained. Thus the 10 subject programs are implemented and output is given as screenshot Figure 6.4. Effectiveness is compared using minimal mutation score. PPC is more effective than EPC and EC. Minimal mutation is a more appropriate measure for this kind of comparative experiment. Minimal mutation score decreases in the order

PPC>EPC>EC



```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Users\CharuMathi\Downloads\Divya>color f0

C:\Users\CharuMathi\Downloads\Divya>k.py
PROGRAM                       MINIMAL_MUTATION_SCORE
                            PPC    EPC     EC
bounded_queue               0.771 0.607 0.607
cal                         0.892 0.088 0.879
check_it                    0.731 0.731 0.731
count_positive              0.881 0.768 0.470
digit_reverser              0.550 0.550 0.133
find_last                   0.601 0.470 0.439
gaussian                    0.538 0.526 0.526
numzero                     0.914 0.530 0.517
oddorpos                    0.770 0.651 0.410
power                       0.500 0.500 0.429

C:\Users\CharuMathi\Downloads\Divya>
```

Fig 6.4 Implementation to find out effectiveness using minimal mutation score

6.3 Cost complexity

To simplify the complexity analysis of *EPC* and *PPC,* we consider only three types of statements: sequential, selection (if statements), and loop (while statements). We also put restrictions on the number of incoming and outgoing edges to the nodes of the CFG. The number of incoming edges (indegree) a node can have is:

0. if it is an initial node and not a loop node

1. if it is a sequential statement node or an initial node with a loop statement

2. if it is a merging node for a selection statement or a node that begins a loop statement

3. if it is a merging node that also begins a loop

The number of outgoing edges (outdegree) a node can have is:

0. if it is an exit node

1. if it is a node with a sequential statement

2. if it is a node with a selection or loop statement.

Fig. 6.5 gives example graph. The notation x.y is used to describe nodes: x represents the indegree and y the outdegree. Thus a node of type x.y has indegree x and outdegree y, and therefore $x * y$ pairs of edges. So each node in the graph has from zero to two pairs of edges.

The total number of Edge Pairs is:

$$\textbf{Edge Pairs} = \sum_{\mathbf{I=0}}^{\mathbf{x*y}} \mathbf{i * Ni}$$

x* y represents pairs of Edges

N_i represents No. of Nodes

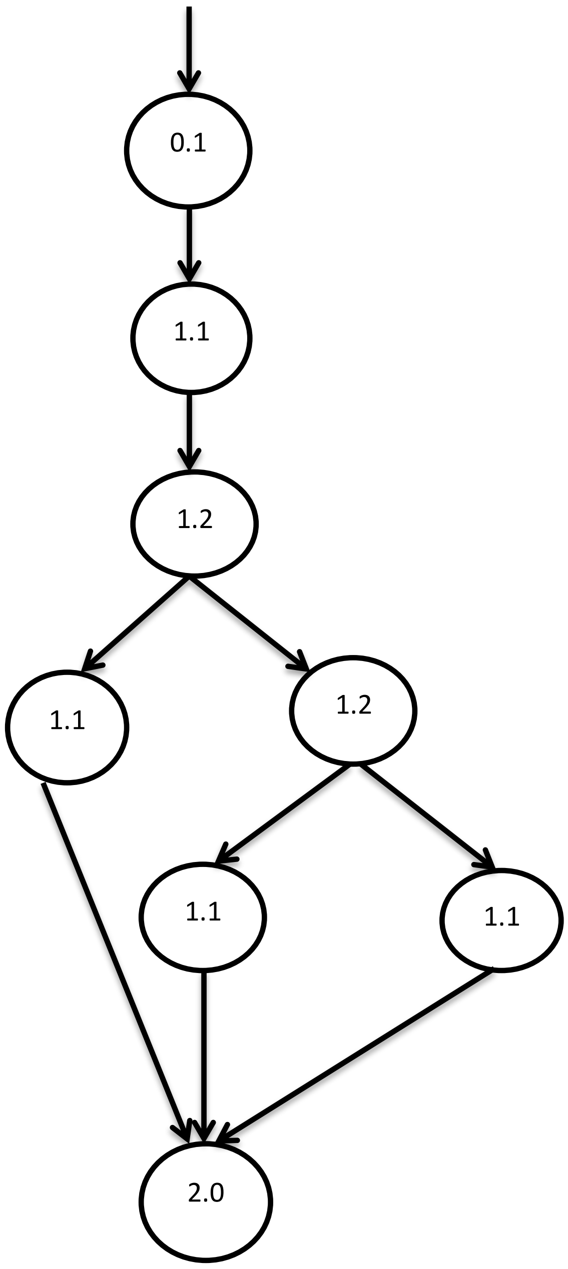


Fig 6.5 Example for CFGs for complexity analysis

Edge pairs =2*Ni

=2*8

=16

EP Requirements = Nodes +Edges +Edge Pairs

=8+10+16

=34

O(Nodes+Edges) =8+10

=18

This gives a complexity of O (Nodes +Edges) for the EPC criterion.

By using experimentation analysis, effectiveness is compared between graph coverage criteria using full mutation score and minimal mutation score. The program is implemented and output is obtained. Cost complexity is analysed between EPC and PPC.

CHAPTER 7

RESULT AND DISCUSSION

7.1 Implementation of graph using visustin

By selecting the format of the code that is C / C++, C#.... and then select the go button and then import the software program into the place of code typing. Using the menu for conversion of program into control flow graph. It will generate the separate graphs for separate sub-functions (fig 6.1).

7.2 Finding the independent paths from the graph

By using the Cyclomatic complexity of McCabe's calculate how many number of paths are derived

$$V(G) = E-N+2(P)$$

Where e -> number of edges

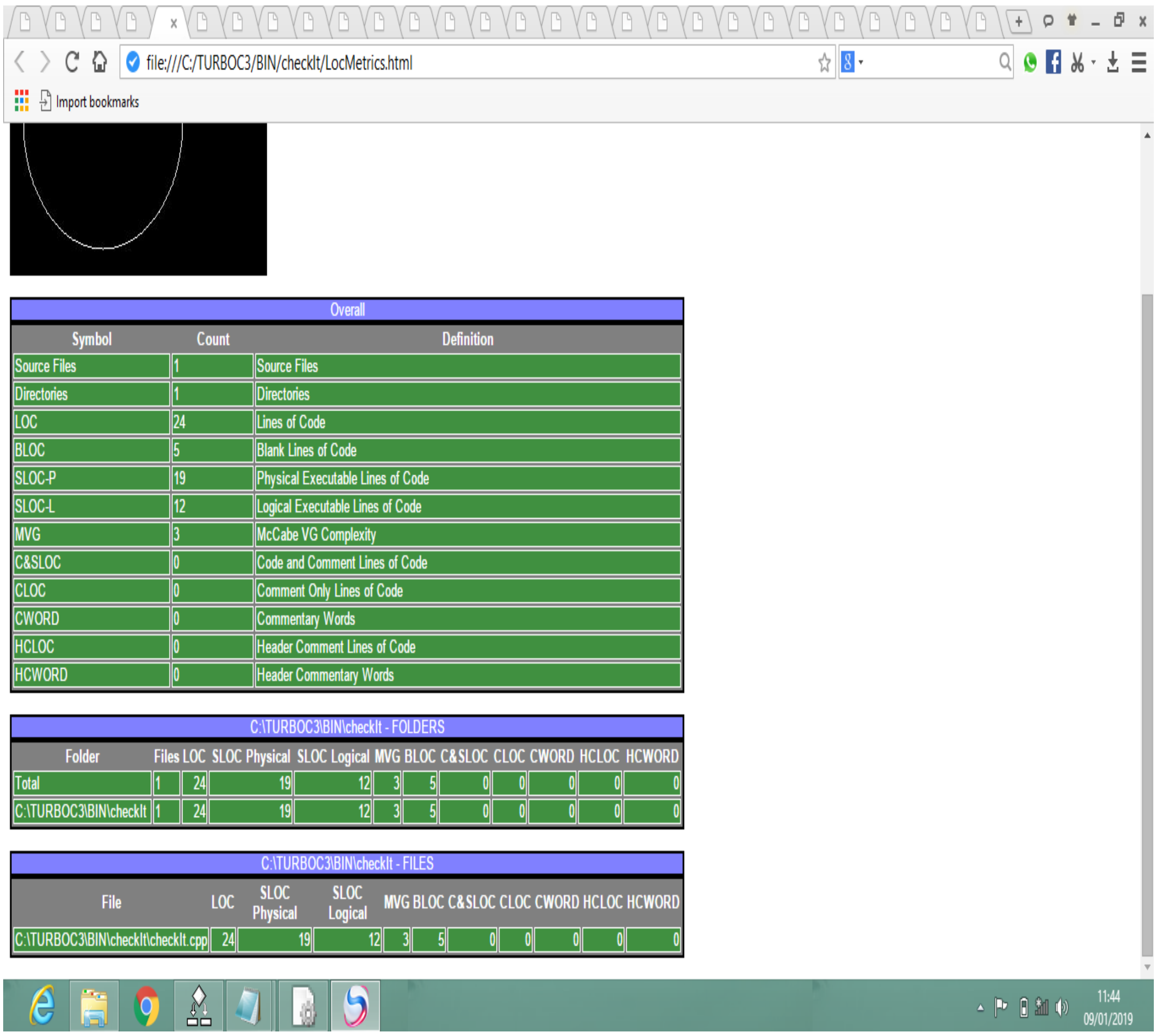
n ->number of nodes

7.3 Applying graph coverage criteria approach

Using the C coding to find out the independent paths. Giving paths as the input and compare the effectiveness and cost between the graph coverage criteria.

7.4 Implementation Works

The LOC Metric tool to calculate number of lines in code and McCabe's value. The LOC metric implementation is represented in fig 7.1.


file:///C:/TURBOC3/BIN/checkIt/LocMetrics.html

Import bookmarks

Overall		
Symbol	Count	Definition
Source Files	1	Source Files
Directories	1	Directories
LOC	24	Lines of Code
BLOC	5	Blank Lines of Code
SLOC-P	19	Physical Executable Lines of Code
SLOC-L	12	Logical Executable Lines of Code
MVG	3	McCabe VG Complexity
C&SLOC	0	Code and Comment Lines of Code
CLOC	0	Comment Only Lines of Code
CWORD	0	Commentary Words
HCLOC	0	Header Comment Lines of Code
HCWORD	0	Header Commentary Words

C:\TURBOC3\BIN\checkIt - FOLDERS

Folder	Files	LOC	SLOC Physical	SLOC Logical	MVG	BLOC	C&SLOC	CLOC	CWORD	HCLOC	HCWORD
Total	1	24	19	12	3	5	0	0	0	0	0
C:\TURBOC3\BIN\checkIt	1	24	19	12	3	5	0	0	0	0	0

C:\TURBOC3\BIN\checkIt - FILES

File	LOC	SLOC Physical	SLOC Logical	MVG	BLOC	C&SLOC	CLOC	CWORD	HCLOC	HCWORD
C:\TURBOC3\BIN\checkIt\checkIt.cpp	24	19	12	3	5	0	0	0	0	0

Fig 7.1 LOC Metric implementation

McCabe's calculation

Edges (E) = 11

Nodes (N) = 9

Sub-functions (P) = 1

V (G) = E – N + 2 (P)

= 11 – 9 + 2

V (G) =4 Paths.

7.5 Effectiveness Analysis

Table 7.1 shows data regarding the effectiveness of test set. The first Column gives the names of the programs. The next three columns give the full Mutation Score for PPC, EPC and EC. For instance, *PPC* for the first subject program (bounded Queue) killed an average (mean) of 99.350% of the mutants and none achieved a 100% mutation score. The next three columns, grouped under Min Mutation Score, the minimal sets have significantly fewer mutants. For instance, *PPC* for the first subject program (bounded Queue) killed on average 77.140% of the seven mutants in the minimal set.

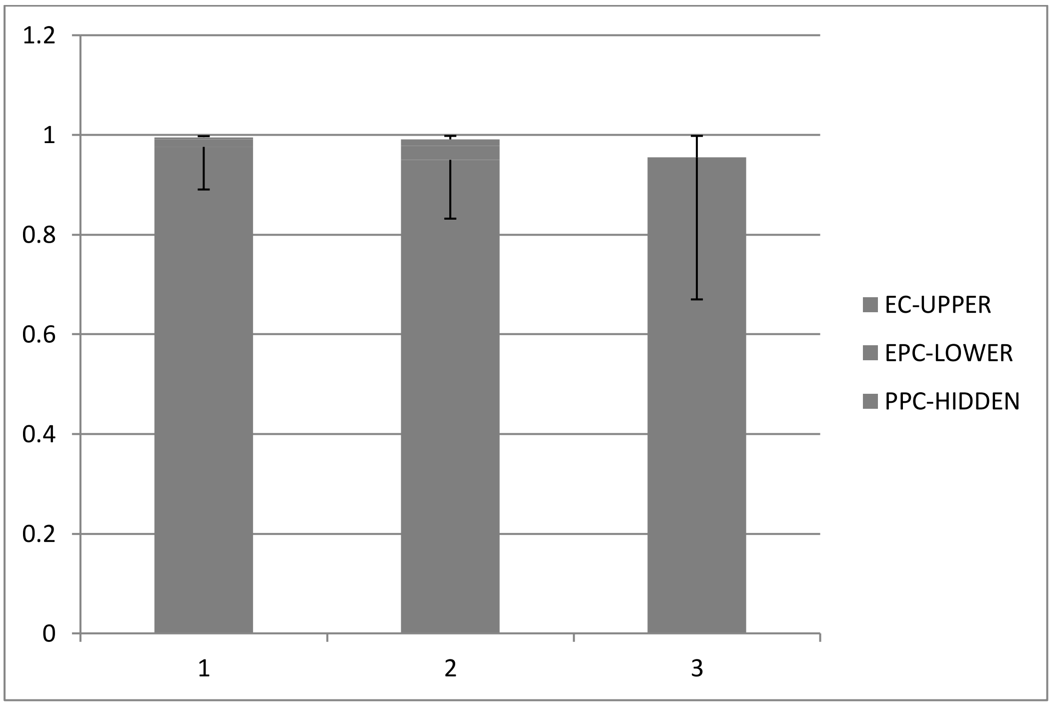


Fig 7.2: Box Plots for Full MS

Figure 7.2 represents a Mutation scores achieved by the PPC, EPC, and EC-adequate test sets. Gray boxplots summarize the mutation scores with all mutants.

Across 10 subject programs, the mean full mutation scores were 97.49% for PPC, 95.29% for EPC, and 93.08% for EC. Across all subject programs, the mean minimal-mutation scores were 71.51% for PPC, 62.18% for EPC, and 51.38% for EC. However, based on the minimal mutation scores, PPC clearly yields stronger test sets than EPC, EPC clearly yields stronger test sets than EC, and PPC is not nearly as strong as mutation. Thus we conclude that the minimal mutation scores are much more indicative of the true strength of the criteria, that is, minimal mutation is a more appropriate measure for this kind of comparative experiment**.**

Program	Full Mutation Score			Min Mutation Score		
	PPC	EPC	EC	PPC	EPC	EC
Bounded Queue	0.994	0.955	0.955	0.771	0.607	0.607
Cal	0.990	0.989	0.987	0.893	0.886	0.879
Check it	0.945	0.945	0.945	0.729	0.729	0.729
Count positive	0.995	0.967	0.895	0.883	0.767	0.467
Digit reverser	0.993	0.993	0.923	0.550	0.550	0.133
Find Last	0.892	0.832	0.815	0.600	0.470	0.440
Gaussian	0.974	0.958	0.958	0.538	0.526	0.526
Num zero	0.998	0.950	0.943	0.917	0.533	0.517
Odd Or Pos	0.994	0.964	0.918	0.770	0.650	0.410
Power	0.976	0.976	0.969	0.500	0.500	0.430
Min	0.892	0.832	0.815	0.500	0.470	0.133
Max	0.998	0.989	0.987	0.917	0.886	0.879
Average	0.9749	0.9529	0.9308	0.7151	0.6218	0.5138

Table 7.1: Effectiveness of Full and Minimal Mutant set.

7.6 Cost Analysis

Table 7.2 presents cost data from the study. The second through fourth columns show the number of TRs for PPC, EPC, and EC for each subject program. The next three columns show the number of infeasible TRs, and the final three show the number of test cases needed to cover all the TRs. The number of test cases is the average over the 20 test sets created for each program

Program	TR			Infeasible			Test cases		
	PPC	EPC	EC	PPC	EPC	EC	PPC	EPC	EC
Bounded Queue	13	46	35	2	0	0	9.700	7.950	7.950
Cal	5	21	18	3	0	0	7.250	7.200	6.950
Check it	6	10	10	0	0	0	6.400	6.400	6.400
Count positive	19	17	18	0	0	0	4.100	3.300	1.900
Digit Reverser	5	6	6	2	0	0	3.050	3.050	1.150
Find Last	15	15	13	0	1	0	3.550	2.700	2.550
Gaussian	36	48	42	0	0	0	11.400	11.200	11.200
Num zero	3	8	9	0	1	0	4.500	1.700	1.600
Odd Or Pos	2	7	7	11	0	0	5.750	4.350	2.500
Power	4	7	6	0	0	0	4.850	4.850	3.800
Min	2	6	6	0	0	0	3.050	1.700	1.150
Max	36	48	42	11	1	0	11.400	11.200	11.200
average	10.8	18.5	16.4	1.8	0.2	0	6.05	5.27	4.6

Table 7.2: Cost Data

CHAPTER 8

CONCLUSION

In this proposed work, three structural criteria, EC, EPC, and PPC are compared in terms of cost and effectiveness. The theoretical evaluation of the cost of EPC, showing its complexity in terms of number of TRs to be O (Nodes+Edges). In Full Mutation Score, PPC killed 98% of the mutants, EPC 97% and EC 94%. The use of minimal sets of mutants corrects this bias and produces a more reliable measure of effectiveness. The differences between the criteria are emphasized with the minimal mutant set. In our study, PPC killed 75% of the mutants, EPC killed 67%, and EC killed only 57%.

As for the cost of these criteria, we found that there is not much difference in terms of the number of TRs. We expected PPC to have the most TRs, so we were surprised to find that, on average, the number of TRs for EPC was highest. PPC tends to create many more infeasible TRs than the other two criteria. Thus, PPC has a much higher number of infeasible TRs, which may be an impediment for its practical use.

PPC can detect more faults, but at higher cost (it requires more infeasible TRs, although it does not generate the highest number of TRs). Thus, a practical tester can make an informed cost versus benefit decision. A better understanding of which structures in the programs contribute to the expense might help to choose when to use PPC. In future, there may be another technique will be developed to reduce the cost.

REFERENCES

1. J.H. Andrews, L.C. Briand, Y. Labiche, A.S. Namin, Using mutation analysis for assessing and comparing testing coverage criteria, IEEE Trans. Softw. Eng. 32(8) (2006) 608.
2. Marcio E. Delamaro,∗ Lin Deng, Nan Li, Vinicius H. S. Durelli,∗ and Jeff Offutt, Growing a Reduced Set of Mutation Operators, Brazilian Symposium on Software Engineering, 2014.
3. A. Gupta, P. Jalote, An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing, Int. J. Softw. Tools Technol. Transf. 10(2) (2008) 145–160.
4. R. Just, F. Schweiggert, Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators, Softw. Test. Verif. Reliab. 25(5–7) (2015) 490–507.
5. N. Li, U. Praphamontripong, J. Offutt, An experimental comparison of four unit test criteria: mutation, edge-pair, all-uses and prime path coverage, in: Fifth Workshop on Mutation Analysis, IEEE Mutation 2009, 2009.
6. P. Ammann, M.E. Delamaro, J. Offutt, Establishing theoretical minimal sets of mutants, in: 7th IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, 2014.
7. D. Tengeri, L. Vidács, A. Beszédes, J. Jász, G. Balogh, B. Vancsics, T. Gyimóthy, Relating code coverage, mutation score and test suite reducibility to defect density, in: International Workshop on Mutation Analysis Co-located with IEEE International Conference on Software Testing, Verification and Validation, ICST, 2016, pp.174–179. D.

8. Lin Deng, Jeff Offutt, Nan Li, Empirical Evaluation of the Statement Deletion Mutation Operator, IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013.
9. Y. Wei, B. Meyer, M. Oriol, Is branch coverage a good measure of testing effectiveness? In: Bertrand Meyer (Ed.), Empirical Software Engineering and Verification: Revised Tutorial Lectures, Springer, 2012, pp.194–212.
10. P.G. Frankl, S.N. Weiss, An experimental comparison of the effectiveness of branch testing and data flow testing, IEEE Trans. Softw. Eng. 19(8) (1993) 774–787.
11. M.M. Hassan, J.H. Andrews, Comparing multi-point stride coverage and dataflow coverage, in: Proceedings of the International Conference on Software Engineering, IEEE, 2013, pp.172–181.
12. H. Hemmati, How effective are code coverage criteria? In: IEEE International Conference on Software Quality, Reliability and Security, 2015, pp.151–156.
13. A. Schwartz, M. Hetzel, The impact of fault type on the relationship between code coverage and fault detection, in: IEEE/ACM International Workshop in Automation of Software Test, AST, 2016, pp.29–35.
14. M. Papadakis, C. Henard, M. Harman, Y. Jia, Y. Le Traon, Threats to the validity of mutation-based test assessment, in: Proceedings of the International Symposium on Software Testing and Analysis, ACM, 2016, pp.354–365.
15. R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, A. Groce, Measuring effectiveness of mutant sets, in: Twelfth IEEE Workshop on Mutation Analysis, Mutation 2016, Chicago, Illinois, USA, 2016, pp.132–141.
16. B. Kurtz, P. Ammann, M.E. Delamaro, J. Offutt, L. Deng, Mutant subsumption graphs, in: Tenth IEEE Workshop on Mutation Analysis, Mutation 2014, 2014.

17. B. Kurtz, P. Ammann, J. Offutt, Static analysis of mutant subsumption, in: Eleventh IEEE Workshop on Mutation Analysis, Mutation 2015, Graz, Austria, 2015.
18. R. Just, D. Jalali, L. Inozemtseva, M.D. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing? in: Proceedings of the Symposium on the Foundations of Software Engineering, FSE, 2014, pp.654–665.
19. B. Kurtz, P. Ammann, J. Offutt, M.E. Delamaro, M. Kurtz, N. Gökçe, Analyzing the validity of selective mutation with dominator mutants, in: 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE, Seattle Washington, USA, 2016.
20. P.G. Frankl, S.N. Weiss, C. Hu, All-uses versus mutation testing: an experimental comparison of effectiveness, J. Syst. Softw. 38(3) (1997) 235–253.
21. P.G. Frankl, Y. Deng, Comparison of delivered reliability of branch, data flow and operational testing: a case study, in: Proceedings of the 2000 Interna-tional Symposium on Software Testing, and Analysis, ISSTA ’00, 2000, pp.124–134.
22. J. Offutt, J. Pan, K. Tewary, T. Zhang, An experimental evaluation of data flow and mutation testing, Softw. Pract. Exp. 26(2) (1996) 165–176.
23. X. Cai, M.R. Lyu, The effect of code coverage on fault detection under different testing profiles, in: Proceedings of the International Workshop on Advances in Model-based Testing, ACM, 2005, pp.1–7.
24. M. Hutchins, H. Foster, T. Goradia, T. Ostrand, Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria, in: Proceed-ings of International Conference on Software Engineering, 1994, pp.191–200.
25. P.G. Frankl, O. Iakounenko, Further empirical studies of test effectiveness, in: Proceedings of the ACM SIGSOFT International

Symposium on Foundations of Software Engineering, ACM, 1998, pp.153–162.

26. M. Gligoric, A. Groce, C. Zhang, R. Sharma, M.A. Alipour, D. Marinov, Comparing non-adequate test suites using coverage criteria, in: Proceedings of the International Symposium on Software Testing and Analysis, ACM, 2013, pp.302–313.
27. M. Gligoric, A. Groce, C. Zhang, R. Sharma, M.A. Alipour, D. Marinov, Guidelines for coverage-based comparisons of non-adequate test suites, ACM Trans. Softw. Eng. Methodol. 24(4) (2015) 22:1–22:33.
28. T. Ball, A theory of predicate-complete test coverage and generation, in: International Symposium on Formal Methods for Components and Objects: Revised Lectures, Springer, 2005, pp.1–22.
29. A.S. Namin, J.H. Andrews, The influence of size and coverage on test suite effectiveness, in: Proceedings of the International Symposium on Software Testing and Analysis, ACM, 2009, pp.57–68.
30. L. Inozemtseva, R. Holmes, Coverage is not strongly correlated with test suite effectiveness, in: Proceedings of the International Conference on Software Engineering, ACM, 2014, pp.435–445.
31. L.C. Briand, D. Pfahl, Using simulation for assessing the real impact of test-coverage on defect-coverage, IEEE Trans. Reliab. 49(1) (2000) 60–70.
32. R. Gopinath, C. Jensen, A. Groce, Code Coverage for suite evaluation by developers, in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp.72–82.
33. P.S. Kochhar, F. Thung, D. Lo, Code coverage and test suite effectiveness: empirical study with real bugs in large systems, in: IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER, 2015, pp.560–564.

34. T.T. Chekam, M. Papadakis, Y.L. Traon, M. Harman, An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption, in: Proceedings of the International Conference on Software Engineering, ICSE, IEEE, 2017, pp.597–608.
35. S.H. Edwards, Z. Shams, Comparing test quality measures for assessing student-written tests, in: Companion Proceedings of the International Conference on Software Engineering, ICSE, ACM, 2014, pp.354–363.

PUBLICATION DETAILS

[1] M.Dhivyabharathi and R.Charumathi & Dr. C .P Indumathi, "**An Unified approach for comparing and evaluating graph coverage criteria**", presented the paper in the proceedings of ICANN-2019, "3rd International Conference on Applied Nanoscience and Nanotechnology (ICANN-2019)", Alagappa University, Karaikudi. ISBN: 978-81-937479-2-6 p.no:44. 18th& 19th March 2019