ISSN 2320 - 2602
Volume 3, No.4, April 2014
International Journal of Advances in Computer Science and Technology
Available Online at http://warse.org/pdfs/2014/ijacst09342014.pdf

PERFORMANCE ANALYSIS OF PRIORITIZED TEST SUITES BASED ON FAULT DETECTION

Usha Badhera[1], **Annu Maheshwari**[2]

[1] Department of Computer Science, India, usha.badhera@gmail.com
[2] M.Tech Scholar, India, maheshwari.annu24@gmail.com

ABSTRACT

Specification based testing identifies test cases from software requirement specifications. This leads to better quality software which reduces effort and cost. Test cases generated for Boolean specifications have been widely used to specify requirements of safety critical softwares, avionics, medical and other control software. Various Boolean specification techniques have been proposed among them MC/DC and MUMCUT techniques are the popular testing techniques. Compliance of the MC/DC criterion has been mandated by Federal Aviation Administration for the approval of airborne software. According to Kaminski (2010) the Federal Aviation Administration requires the Minimal-MUMCUT criterion instead of MC/DC for Irredundant Disjunctive Normal Form (IDNF). The Minimal-MUMCUT criterion provides better logic fault detection .In this paper performance analysis of proposed prioritized test suite generated from Minimal MUMCUT has been done. The APFD of prioritized test suite is computed and compared with other possible prioritized test suites. Minimal MUMCUT identified testing techniques for Boolean specification for which various 2^n distinct Boolean functions with n variables can be formed. To distinguish one from all others using exhaustive testing, it would require 2^n distinct test cases. Test cases generated by Minimal MUMCUT are less than the test cases generated by MUMCUT Strategy. The proposed approach for prioritization of test cases generated by Minimal MUMCUT yields higher APFD and hence early detection of faults.

Key words: MUMCUT, MUTP, CUTPNFP, APFD

1. INTRODUCTION

Software testing and retesting occurs continuously during the software development life cycle to detect errors as early as possible. During the regression testing, a modified system needs to be retested using the existing test suite. Since the test suite may be very large, the better way is to prioritize it. Regression testing is a necessary but expensive process in the software lifecycle. One of the regression testing approaches, test case prioritization, aims at sorting and executing test case in order of potential abilities to achieve certain testing objective. Test Case Prioritization [8] schedule test cases for regression testing in an order that attempts to maximize some objective function. For example, testers might wish to schedule test cases in an order that achieves code coverage at the fastest rate possible, exercises features in order of expected frequency of use, or exercises sub-systems in an order that reflects their historical propensity to fail. When the time required to execute all test cases in a test suite is short, test case prioritization may not be cost effective, it may be most expedient simply to schedule test cases in any order. In the past decade, may testing criteria have been proposed for software characterized by complex logical decisions, such as those in safety-critical software[1],[2].[3]. In recent years, more sophisticated coverage criteria have been advocated, like BOR (Boolean OpeRator Testing Strategy), BMIS (Basic Meaningful Impact Strategy), modified condition/decision coverage (MC/DC) ([1] [2] [4]) and the MUMCUT criteria. [5]

MUMCUT strategy is to generate test cases that can guarantee detection of seven types of single faults provided that the original expression is in irredundant disjunctive normal form (IDNF) [6]. In this strategy, there is no restriction on the number and occurrence of variables in the given Boolean expressions. Minimal-MUMCUT [7] that improves the MUMCUT strategy by considering the feasibility problem of the three testing constituents of the MUMCUT strategy, It reduces the test suite size as compared to MUMCUT without compromising any fault detection capability. Thus, the extra tests required by the MUMCUT criterion are of little, if any, value based on the theoretical and empirical studies conducted [7].

2. TEST CASE PRIORITIZATION

Test case prioritization techniques schedule test cases in an execution order according to some criterion. Test case prioritization problem is defined [8] as follows:
Given: T, a test suite; PT, the set of permutations of T; f, a function from PT to the real numbers
Problem: Find T’ belongs to PT such that (for all T”) (T” belongs to PT) (T” $\neq$ T’) [f (T’) $\geq$ f (T”)]
Here, PT represents the set of all possible prioritizations (orderings) of T and f is a function that, applied to any such ordering, yields an award value for that ordering. The performance of the prioritization technique used is known as effectiveness. It is necessary to assess effectiveness of the ordering of the test suite. Effectiveness will be measured by the rate of faults detected. The following metric is used to calculate the level of effectiveness:

2.1 Average Percentage of Faults Detected (APFD) Metric

APFD (Average Percentage Fault Detected) metric is a measure of how rapidly a prioritized test suite detects faults, which measures the weighted average of percentage of faults detected over the life of a test suite. [9], [8].The APFD used in this paper is calculated by taking the weighted average of the number of faults detected during the run of the test suite. APFD can be calculated using the following notations:

Let T **-** The test suite under evaluation
m **-** The number of faults contained in the program under test P
n **-** The total number of test cases and
TFi **-** The position of the first test in T that exposes fault i.

$$\text{APFD} = 1- \frac{TF1+TF2+TF3+TF4+TF5 \ldots\ldots\ldots TFi}{m*n} + \frac{1}{2*n}$$

APFD can be calculated when prior knowledge of faults is available. APFD values ranges from 0 to 100; higher value implies faster (better) fault detection rates.

3. FAULT BASED PRIORITIZATION OF MINIMAL MUMCUT TESTS

Single faults of seven types mentioned in Section 4.1, are generated using JAVA eclipse and JAVA collection framework. For the given expression Minimal MUMCUT test cases are generated and a feasibility criterion is tested. Test cases are arranged according to the algorithm for fault based prioritization of Minimal MUMCUT test cases. Algorithm is given below:

Input: Test suite T and number of faults detected by a test case

Output: Prioritized Test suite T'.

```
1. Begin
2. Set T' empty
3. For each term X do
4.     If MUTP criteria is feasible for X
           Prioritize Multiple Unique True Points (U) followed by overlapping Near False Points (N)
5.     for each literal x in term X
6.         If CUTPNFP criteria is feasible for x
               Prioritize Unique True Points (U) followed by Corresponding Unique True Points Near False Points(C)
7.     End for
8. Else
       Prioritize Multiple Unique True Points (U) followed by Corresponding Unique True Points Near False Points(C) followed by overlapping Near False Points (N)
9. End For
10. End
```

4. PROPOSED WORK

4.1 Faults in Logical Expressions

A fault is an error in the original Boolean expression. A faulty implementation is referred to as single-fault expression if (1) it differs from the original expression by one syntactic change; and (2) it is not equivalent to the original expression. This study considers the following classes of simple faults for logical decisions. A decision *S* in *n* variables can always be written in *disjunctive normal form* (DNF) as a sum of product (Lau, Yu [2001]).

$$S = ab + \bar{c}d + e$$

Table1: Types of Faults

Fault	Description	Example
Expression Negation Fault (ENF)	The expression or sub-expression is negated	$\overline{ab + \bar{c}d} + \text{e}$
Term Negation Fault (TNF)	A term is negated	$\overline{ab} + \bar{c}d + e$
Term Omission Fault (TOF)	A term is omitted	$\overline{cd} + e$
Operator Reference Fault (ORF)	An OR operator(+) is implemented as the AND operator or vice versa	$ab.\bar{c}d + e$ or $a + b + \bar{c}d + e$
Literal Negation Fault (LNF)	A literal is negated	$\bar{a}b + \bar{c}d + e$
Literal Omission Fault (LOF)	A literal is omitted	$b + \bar{c}d + e$
Literal Insertion Fault (LIF)	A literal is inserted	$abc + \bar{c}d + e$
Literal Reference Fault (LRF)	A literal is implemented as another literal	$\overline{ac} + \bar{c}d + e$

4.2 Fault Generation

Fault generation is handled by a series of complex string manipulations on JAVA Eclipse using JAVA Collection Frameworks. The general methodology of generating faults starts with the expression, which is just an infix string at this point, being passed through a tokenizer. The tokens then are searched for the one that will have the fault inserted before or after.

Input: All the single faults take the original Boolean expression in IDNF form

Output: Give all the faulty expressions as output

4.2.1 Operator Negation Fault (ONF)

Step1: Count the number of operators in the expression. This number indicates how many derivatives will be created from this one expression and allows the allocation of storage for each result.

Step 2: From this point onwards, the string is tokenized and the tokens are copied to each of the resulting expressions.

Step 3: Do until all tokens have processed

(a) If the token is an AND operator (&) in the Boolean expression then it is replaced by the OR (|) operator and vice versa.

(b) If the token is an OR operator (|) in the
(c) the AND (&) operator and vice versa.

Step 4: With this process complete, the program returns the resulting array of faulty string expressions.

4.2.2 Expression Negation Fault (ENF):

Step 1: Insert a negation before each "opening parenthesis."
Step 2: The program recursively searches for groups of operands that are joined via an "and" operator, these groups can include other scoped parts of the expression or the entire expression itself.
Step 3: Each of the "& blocks" are surrounded by parenthesis and then the entire "and block" is negated.
Step 4: The resulting data is optimized into an array with no empty spaces and the program returns the resulting array of faulty string expressions.

4.2.3 Variable Negation Fault (VNF)

Step 1: Count the number of variable in the expression. This number indicates how many derivatives will be created from this one expression and allows the allocation of storage for each result.
Step 2: From this point onwards, the string is tokenized and the tokens are copied to each of the resulting expressions.
Step 3: Do until all tokens have processed

(a) If the token is a variable, then a negation operator(!) is inserted
(b) If the token is a negated variable, then the negation operator is removed

Step 4: With this process complete, the program returns the resulting array of faulty string expressions.

4.2.4 Term Negation Fault (TNF):

Step 1: Count the number of terms in the expression. This number indicates how many derivatives will be created from this one expression and allows the allocation of storage for each result.
Step 2: From this point onwards, the string is tokenized and the tokens are copied to each of the resulting expressions.
Step 3: Do until all tokens have processed

a) If the token is a term, then a negation operator (!) is inserted
b) If the token is a negated term, then negation operator is removed

Step 4: With this process complete, the program returns the resulting array of faulty string expressions.

4.2.5 Term Omission Fault (TOF)

Step 1: Count the number of terms in the expression. This number indicates how many derivatives will be created from this one expression and allows the allocation of storage for each result.
Step 2: From this point onwards, the string is tokenized and the tokens are copied to each of the resulting expressions.
Step 3: Do until all tokens have processed

Boolean expression then it is replaced by
(a) If the token is a term, then term is omitted

Step 4: With this process complete, the program returns the resulting array of faulty string expressions.

4.2.6 Literal Omission Fault (LOF)

Step 1: Count the number of literals in the expression. This number indicates how many derivatives will be created from this one expression and allows the allocation of storage for each result.
Step 2: From this point onwards, the string is tokenized and the tokens are copied to each of the resulting expressions.
Step 3: Do until all tokens have processed

(a) If the token is a literal, then that literal is omitted

Step 4: With this process complete, the program returns the resulting array of faulty string expressions.

4.2.7 Literal Insertion Fault (LIF)

Step 1: Count the number of literals in the expression. This number indicates how many derivatives will be created from this one expression and allows the allocation of storage for each result.
Step 2: From this point onwards, the string is tokenized and the tokens are copied to each of the resulting expressions.
Step 3: Do until all tokens have processed

(a) If the token is a term, then a literal (which is not present in that term) is inserted before that same token

Step 4: With this process complete, the program returns the resulting array of faulty string expressions.

4.2.8 Variable Reference Fault (VRF)

Step 1: Count the number of literals in the expression. This number indicates how many derivatives will be created from this one expression and allows the allocation of storage for each result.
Step 2: From this point onwards, the string is tokenized and the tokens are copied to each of the resulting expressions.
Step 3: Do until all tokens have processed

a) If the token is a literal, then this literal is replaced by all the other literals present in the expression

Step 4: With this process complete, the program returns the resulting array of faulty string expressions.

4.3 Total Number of Faults of Various Types for TCAS Boolean Expressions

Total number of faults of types ONF, TNF, TOF, LIF, LOF, ENF, VRF, and VNF are generated for TCAS Boolean expressions and a subset of some Boolean expressions. The result is tabulated in Table 2. According to these results the total number of generated faults is 9087 for TCAS 20 Boolean expressions with a subset of some Boolean expressions and for one expression value of single faults ranges from 52 to 1470.

Table2: Number of Generated Faults

S. N.	EXPRESSION	ONF	ENF	LIF	LOF	TNF	TOF	VNF	VRF	Total
1	T01	28	5	6	29	5	5	29	174	**281**
2	T02	105	13	12	106	13	13	106	848	**1216**
3	T04	6	3	8	7	3	3	7	28	**65**
4	T05	27	9	53	28	9	9	28	224	**387**
5	T06	57	6	8	58	6	6	58	580	**779**
6	T08	31	4	-	32	4	4	32	224	**331**
7	T09	13	1	-	14	2	2	14	84	**130**
8	T10	59	6	18	60	6	6	60	720	**935**
9	T11	62	9	54	63	9	9	63	756	**1025**
10	T12	-	-	-	-	-	-	-	-	-
11	T13	13	6	58	14	6	6	14	154	**271**
12	T14	15	6	26	16	6	6	16	96	**187**
13	T15	31	11	67	32	11	11	32	256	**451**
14	T16	81	23	189	87	23	23	87	957	**1470**
15	T17	31	6	34	32	6	6	32	320	**467**
16	T18	37	8	42	38	8	8	38	342	**521**
17	T19	19	4	12	20	4	4	20	140	**223**
18	T20	11	1	2	12	2	2	12	72	**114**
19	T21	3	1	8	4	2	2	4	28	**52**
20	T22	5	3	8	6	3	3	6	36	**70**
21	T23	5	3	6	6	3	3	6	28	**60**
22	T24	3	1	8	4	2	2	4	28	**52**
	TOTAL	**642**	**129**	**610**	**668**	**131**	**131**	**668**	**6095**	**9087**

*12th expression is not included due to missing right parenthesis

4.4 UTP & NFP Test Suite Size for Boolean Expressions

These test cases include Unique True Points and Near False Points. Test case generation aims at finding test cases that detect certain types of faults (illustrated in Section 4.1). The result is tabulated in Table 3. According to these results the number of test cases is increasing with the increase in number of variables in the expression, only exception are those expressions where variable are repeating in more than one term. Total number of test cases generated ranges from 9 to 2744.

Table 3: Size of Test Cases for TCAS 20 Boolean expressions

Expression	Number of Literal	UTP Test Cases	NFP TEST Cases	Total
T01	7	8	44	**52**
T02	9	16	129	**145**
T03	7	35	813	**848**
T04	5	15	19	**34**
T05	9	181	234	**415**
T06	11	10	94	**104**
T07	8	15	64	**79**
T08	8	4	32	**36**
T09	7	2	14	**16**
T10	13	12	120	**132**
T11	13	248	1592	**1840**
T13	12	1284	1460	**2744**
T14	7	33	81	**114**
T15	9	83	213	**296**
T16	12	750	1041	**1791**
T17	11	186	816	**1002**
T18	10	78	342	**420**
T19	8	24	120	**144**
T20	7	4	24	**28**
T21	3	3	6	**9**
T22	5	10	30	**40**
T23	3	4	8	**12**
T24	4	6	12	**18**

5. EXPERIMENTAL SETTINGS & RESULT

5.1 When Multiple Unique True Point (U) criteria is feasible for Boolean Expression

For the Boolean Expression (!a&b)|(c&d), MUTP criteria is feasible that is test suite includes the test cases which covers both values 0 and 1 for missing literals in both of the terms.

Table 4: All Test Cases for Boolean Expression (!(a&b)|(c&d)

Test Cases U followed by N		**Test Cases N followed by U**	
T1	0101	T5	1101
T2	0110	T6	0010
T3	0011	T1	0101
T4	1111	T2	0110
T5	1101	T3	0011
T6	0010	T4	1111

The comparison graph is drawn between APFD value of Boolean expression (!a&b)|(c&d) using UN order and NU order, which shows that value of APFD obtained using UN order is more than NU order.(See Figure 1 and 2)

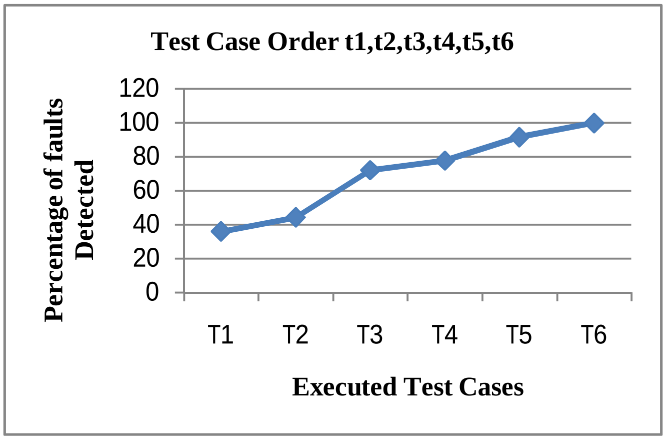

Figure 1: Graph for Boolean expression (! a&b)|(c&d) for UN ordered test cases with 62.01% APFD

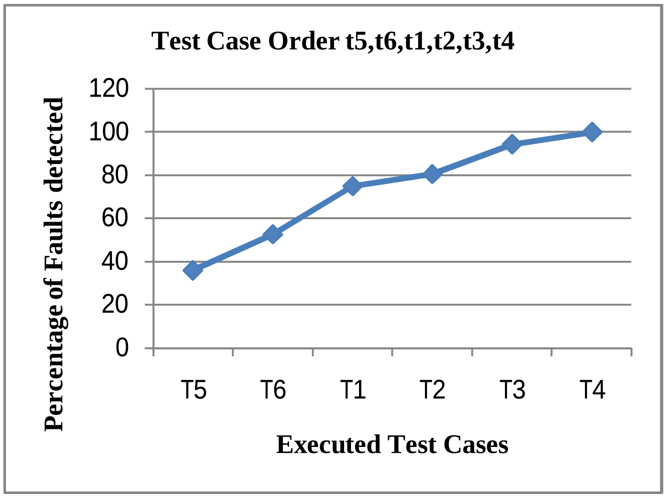

Figure 2: Graph for Boolean expression (! a&b)|(c&d) for NU ordered test cases with 45.8% APFD

5.2 When Multiple Unique True Point (U) criteria is infeasible for Boolean Expression

For the Boolean Expression (a&b)|(b&c) MUTP criteria is not feasible that is test suite does not include the test cases which covers both values 0 and 1 for missing literals in both of the terms.

Table 5: All Test Cases For Boolean expression (a&b)|(b&c)

Test Cases U followed by C		**Test Cases C followed by U**	
T1	110	T3	010
T2	011	T4	100
T3	010	T5	001
T4	100	T1	110
T5	001	T2	011

The comparison graph is drawn between APFD value of Boolean expression (!a&b)|(c&d) using UC order and CU order, which shows that value of APFD obtained using UC order is more than CU order(See Figure 3 and 4) .

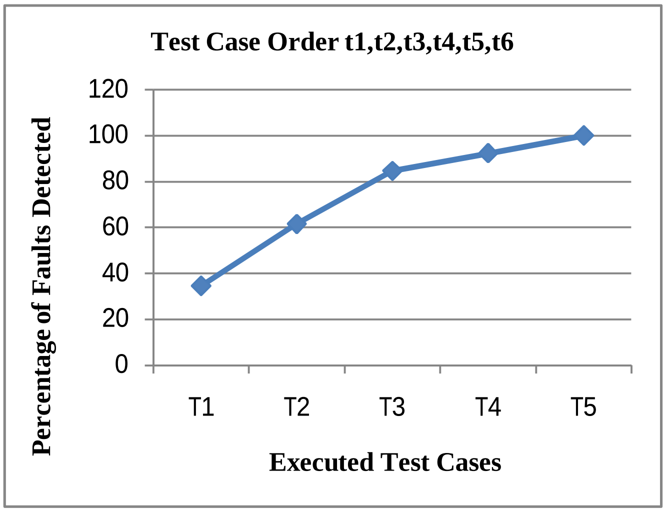

Figure 3: Graph for Boolean expression (!a&b)|(c&d) for UC ordered test cases with 67.86% APFD

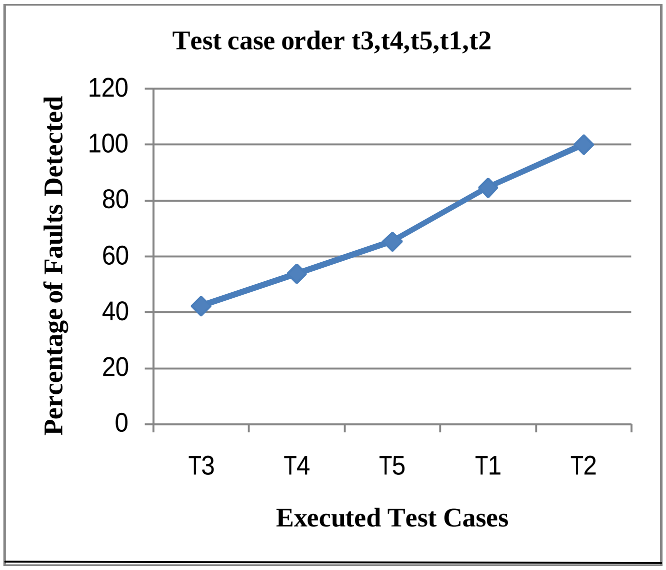

Figure 4: Graph for Boolean expression (! a&b)|(c&d) for CU ordered test cases with 59.29% APFD

Thus above experiment leads to some results which are listed in Table 6.

Table 6: Comparison of APFD for some Boolean expressions

<table>
<tr><th>SN.</th><th>Predicate</th><th>Feasibility Criteria</th><th colspan="2">APFD(%) with Proposed Approach</th><th colspan="2">APFD(%) with Random order</th></tr>
<tr><td>1</td><td>(a&!b&d)|(a&!c&d)|e)</td><td>MUTP in-feasible</td><td>UC Order</td><td>67.63</td><td>CU Order</td><td>59.5</td></tr>
<tr><td>2</td><td>(a&b)|(a&c)|(b&c)</td><td>MUTP in-feasible</td><td>UC Order</td><td>57.10</td><td>CU Order</td><td>55.09</td></tr>
<tr><td>3</td><td>(a&b&c)|(d&e)</td><td>MUTP Feasible</td><td>UN Order</td><td>61.08</td><td>NU Order</td><td>56.01</td></tr>
<tr><td>4</td><td>(a&b)|(b&!c)|(!b&c)</td><td>MUTP in-feasible</td><td>UC Order</td><td>60</td><td>CU Order</td><td>55.89</td></tr>
<tr><td>5</td><td>(!a&b)|(c&d)</td><td>MUTP Feasible</td><td>UN Order</td><td>62.01</td><td>NU Order</td><td>45.8</td></tr>
<tr><td>6</td><td>(a&b)|(b&c)</td><td>MUTP in-feasible</td><td>UC Order</td><td>67.86</td><td>CU Order</td><td>59.29</td></tr>
</table>

6. CONCLUSION & FUTURE WORK

This paper illustrates the comparison between proposed algorithm and the random approach for Prioritization of Minimal MUMCUT test cases in order to improve regression testing. In proposed study the experiments were done on Boolean expressions where MUTP criteria is feasible and MUTP criteria is not feasible and provided higher value of Average Percentage of Faults Detected metric with MUTP (U) test cases followed by MNFP (N) i.e. UN order and MUTP (U) test cases followed by CUTPNFP(C) test cases i.e. UC order, as compared to the random order NU Order and CU order. In future the experiment need to be conducted on the Boolean expression having more no of literals and order of prioritization need to be validated for high rate of fault detection.

REFERENCES

[1] Chilenski, J.J., Miller, S.P., **“Applicability of modified condition/decision coverage to software testing”**, Software Engineering Journal 9 (5), 193–229, 1994

[2] Dupuy, A., Leveson, N., **“An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software”**, In Proceedings of Digital Aviation Systems Conference (DASC), 2000

[3] Chilenski, J.J., **“An investigation of three forms of the modified condition decision coverage (MCDC) criterion”**, Federal Aviation Administration, US Department of Transportation, Washington, DC, Tech. Rep. DOT/ FAA/AR-01/18, 2001

[4] Jones, J.A., Harrold, M.J.,**”Test-suite reduction and prioritization for modified condition/decision coverage”**, IEEE Transactions on Software Engineering 29 (3), 195–209, (2003)

[5] Kaminski, G., & Ammann, P., **“Using a fault hierarchy to improve the efficiency of DNF logic mutation testing”**, In *Software Testing Verification and Validation, ICST’09. International Conference on* (pp. 386-395). IEEE, 2009

[6] Lau M.F., Chen T.Y, **“Test Case Selection strategies based on Boolean Specifications”** Software Testing, Verification and Reliability, 11(3), 165-180, 2001

[7] Yu, Y.T., & Lau M.F., **“Fault-based test suite prioritization for specification-based testing”** *Information and Software Technology, 54*(2), 179-202, 2012

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

[9] Malishevsky, A. G., Ruthruff, J. R., Rothermel, G., & Elbaum, S.**”Cost-cognizant test case prioritization”** ,*Department of Computer Science and Engineering, University of Nebraska-Lincoln, Technical Report,2006*

[10] Kaminski, G. K. **“Applications of Logic Coverage Criteria and Logic Mutation to Software Testing”** (Doctoral dissertation, George Mason University, 2010