Dynamic Fault Visualization Tool for Fault-based Testing and Prioritization

Patrick Daniel
Faculty of Engineering, Computing and Science
Swinburne University of Technology
Kuching, Sarawak, Malaysia
pdaniel_koe@yahoo.com

Kwan Yong Sim
Faculty of Engineering, Computing and Science
Swinburne University of Technology
Kuching, Sarawak, Malaysia
ksim@swinburne.edu.my

Abstract—**Fault-based testing has been proven to be a cost effective testing technique for software logics and rules expressed in Boolean expressions. It can guarantee the elimination of common faults without exhaustive testing. However, average software testing practitioners may not have in-depth knowledge on Boolean algebra and complex logic derivations required to apply existing fault-based testing techniques. In this paper, a dynamic fault visualization tool has been proposed. This tool allows its user to visualize fault-based testing and prioritize test inputs with a simple greedy method. The performance evaluation of this tool has been done on Boolean expressions extracted from a real life aviation tool. The results show that it can achieve significant performance improvements compared to ordinary sequential order test execution and existing static technique. The proposed visualization tool could also identify possible faults to guide the debugging process.**

Keywords-component; Software Testing, Fault-based Testing, Fault Visualization.

I. Introduction

Software testing and debugging contribute up to 50% to 75% of the software development cost [1], making it the most expensive activities in the software development life-cycle. A software system with m input variables, each having n values, will have m^n possible inputs for testing. Hence, it is prohibitively expensive, if not impossible, to exhaustively test a software system for all its possible inputs. This problem is further amplified for software systems that require long computation time to run each test inputs or if the execution process requires user interactions that cannot be automated to run a large number of test inputs.

In recent years, fault-based testing [2][3][4] have emerged as an innovative approach to minimize the cost and time in testing software system. Fault-based testing techniques guarantee the detection of common types of faults committed by programmers by selecting only a small test suite (that is, a small set of test inputs) that target the detection these common faults. If software under test does not fail when tested with this small test suite, then it can be guaranteed that the software system is free from these common fault types. Fault-based testing has been proven to be particularly effective in testing software systems in which logics and rules are either specified or can be modeled into Boolean expressions [5].

Existing studies on fault-based testing for Boolean expressions have been focusing on the reduction of test suite size [2][5], automated generation of test cases [3][7][9] and the relationships among common fault-classes [4][8][10][11]. Even though these fault-based testing techniques are highly effective, they require knowledge on complex logic derivations and in-depth knowledge on Boolean algebra. Unfortunately, not all software testing practitioners have the required skills, hence, limits the applicability of these fault-based testing techniques.

In view of this problem, a new dynamic fault visualization technique and tool are proposed in this paper for fault-based testing of Boolean expressions. A colour image will be used to help software testers visualize and select the best test input to run in order to detect the most number of common faults. If no fault is detected by the selected test input, it implies that the software under test is free from the common faults detectable by this test input. The colour image is then updated to allow the testers to visualize and select the next best test input to detect the remaining common faults. This way, test inputs are prioritized according to their fault detection capability. Unlike existing techniques, this visual based technique and tool is simple and accessible to more software testing practitioners because it does not require complex logic derivations and in-depth knowledge of Boolean algebra.

In order to evaluate the performance of the proposed technique and tool developed, the aircraft Traffic Collision Avoidance System II (TCASII) [5] has been used as the subject program for testing. The experiment results show that the proposed interactive fault visualization technique and tool yield significant improvements in fault detection efficiency. Furthermore, the tool could also identify the possible faults occurred. This can be used to guide the debugging (fault localization) process.

The rest of this is organized as follows: Section II outlines the terminologies used in this paper, types of faults and the subject program used for testing. Section III describes the fault-based testing and prioritization strategy. Section IV introduces the proposed tool, dynamic fault visualization tool. Section V evaluates and compares the performance of the proposed dynamic fault visualization tool. Section VI discusses the findings and concludes this paper.

978-0-7695-4959-0/13 $25.00 © 2013 IEEE
DOI 10.1109/ACSAT.2012.55

II. Preliminaries

This section will outline the terminologies and notations used in Boolean expressions, types of common faults and the Boolean specifications of the subject program for testing, TCASII.

A. *Terminologies and Notations*

The Boolean operators and the truth values used in this paper are defined in TABLE I.

TABLE I. Boolean Operator and Truth Values

AND	•
OR	+
NOT	‾
TRUE	1
FALSE	0

The set of truth values are denote by **B** = {0, 1} and **B**^{n} denotes n-dimensional Boolean space. A variable in Boolean expression can be a positive literal or a negative literal. A positive literal denotes by original variable and a negative literal denotes by a bar "$^-$" at the top of original variable. For example, given Boolean expression $a\bar{b} + bc$, b occurs as a negative literal in the first term and positive literal in second term.

A Boolean expression in disjunctive normal form is said to be irredundant, if none of its terms can be omitted from the expression, and none of its literals may be omitted from any term in the expression without changing its meaning [12]. For instance, Boolean expression $a + b$ is the irredundant disjunctive normal form of Boolean expression $a\bar{b} + b$.

Let S denotes a program specification written in a Boolean expression in disjunctive normal form with m terms,

$$S = t_1 + \cdots + t_m$$
$$t_i = v_1^i \ldots v_{k_i}^i$$

where $t_i\ (i = 1, 2, 3 \ldots, m)$ is the i-th term in S. and $v_j^i (j = 1, 2, 3 \ldots, k_i)$ denotes the j-th variable in term t_i which has k_i variable. For example, if $S = a\bar{b} + bc$, then $t_1 = a\bar{b}$ where $v_1^1 = a$, $v_2^1 = \bar{b}$ and $t_2 = bc$ where $v_1^2 = b$, $v_2^2 = c$.

B. *Types of Faults*

In the process of implementing logics and rules represented by Boolean expressions in software requirement specifications into program codes, programmers could commit errors such as incorrect control predicates, omitted or additional conditions and paths as well as incorrect operators or operands. These errors result in omission, insertion or incorrect reference of Boolean operands or operators in Boolean specifications expressed in DNF form. Faults committed by programmers are commonly categorized into seven types in previous studies on Fault-based testing for Boolean expressions [2][4][8][9]. These common fault types are listed below.

1) LIF – *Literal Insertion Fault*

An insertion of single variable which exists in whole expression but does not exist in the tern it is inserted. For example:

Original Expression	: $ab + cd + e$
After Literal Insertion	: $abc + cd + e$

2) LNF – *Literal Negation Fault*

A modification of single variable from its original state to negation. For example:

Original Expression	: $ab + cd + e$
After Literal Negation	: $\bar{a}b + cd + e$

3) LOF – *Literal Omission Fault*

A variable is removed from its term. For example:

Original Expression	: $ab + cd + e$
After Literal Omission	: $b + cd + e$

4) LRF – *Literal Reference Fault*

A variable is replaced by another variable which exists in the whole expression but not in the term where literal replacement occurs. For example:

Original Expression	: $ab + cd + e$
After literal reference fault	: $cb + cd + e$

5) ORF – *Operator Reference Fault*

A switch of operator from 'AND' to 'OR', and vice versa. For example:

'AND' → 'OR'

Original Expression	: $ab + cd + e$
After Operator Switch	: $a + b + cd + e$

'OR' → 'AND'

Original Expression	: $ab + cd + e$
After Operator Switch	: $abcd + e$

6) TNF – *Term Negation Fault*

A conversion of original terms to its negation. De Morgan's Laws apply on this fault, where $\overline{ab} = \bar{a} + \bar{b}$. For Example:

Original Expression	: $ab + cd + e$
After Term Negation	: $\bar{a} + \bar{b} + cd + e$

7) TOF – *Term Omission Fault*

A term is removed from the whole expression. For example:

Original Expression	: $ab + cd + e$
After Term Omission	: $cd + e$

C. *Subject Program for Testing and Mutation*

The subject program for testing in this study is TCASII program (Traffic Collision Avoidance System II). TCASII is a real life aircraft collision avoidance system that is used to reduce the risk of mid-air collision between aircrafts. Logic and rules in the TCASII system are specified by 20 Boolean expressions, each having a different number of variables.

Figure 1. (on the last page of this paper) shows the original Boolean expression in general form extracted from TCASII. A mutation approach [3] is used to insert one fault at a time into Irredundant Disjunctive Normal Form of these Boolean expressions.

III. Fault-Based Test Input Selection and Prioritization Strategies

Fault-based testing techniques select a small set of test inputs to guarantee the detection of common faults identified. However, the fault detection capability of the test inputs selected may vary from one to the other. With the time and cost constraints in testing process, it is desirable to run test input that can detect more fault first to maximize the chance to detect fault as early as possible. Figure 2. illustrates the advantage of prioritization.

Let '000', '001', '010', ..., '111' represent the a test inputs, '1', '2', '3', ..., 10' represent the common faults identified, and 'X' indicates the capability of a test input to detect the fault. Let 000 → {1, 8} denotes test inputs '000' is capable to detect fault '1' and '8'.

Let *O* denotes a sequence of test inputs to detect all faults, t_i denotes test input *i*, and f_j^i denotes fault *j* can be detected by test input *i*. Let *F* denotes the sequence of faults detected by executed *O*.

$$O = t_1, t_2, t_3, \dots, t_i$$
$$tc_i = f_1^i, f_2^i, f_3^i, \dots, f_j^i$$

From Figure 2. , without prioritization, a *Sequential Order,* where the first test input is executed first followed by the second and subsequent test inputs, will require the first six test inputs to be executed to detect all 10 faults as required in fault-based testing.

$$O = \{000, 001, 010, 011, 100, 101\}$$
$$F = \{(1,8), (2,9), (3,4,5,6), (7), (\,), (10)\}$$

In sequence *O*, it can be observed that in Figure 2. , test input '100' is redundant because the faults detectable by this test inputs have been detected by test inputs executed earlier in the sequence. Therefore, executing test input '100' in Sequential Order is a waste of testing time and cost.

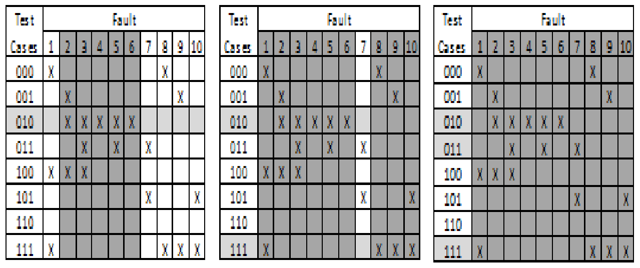
On the other hand, by using a simple greedy method to prioritize test inputs execution, the goal of detecting more faults as early as possible in the testing process could be achieved. In the greedy method, the test inputs that are able to detect the most number of faults were selected. Once this test input has been executed on the software under test without causing any failure, the faults (columns in Figure 2.) detectable by this test input can be removed from the table. The test input that can detect the most number of remaining faults is then selected as the next test input.By applying this greedy method on the test inputs in Figure 2. , the following test sequence can be obtained which only requires three test inputs to be executed to detect all possible faults in fault based testing.

$$O = \{010, 111, 011\}$$
$$F = \{(2,3,4,5,6), (1,8,9,10), (7)\}$$

Figure 3. shows how these faults are eliminated (shaded in grey) after each test input in this sequence executed on the software under test.

Test Inputs	Faults									
	1	2	3	4	5	6	7	8	9	10
000	X							X		
001		X							X	
010		X	X	X	X	X				
011			X		X		X			
100	X	X	X							
101							X			X
110										
111	X							X	X	X

Figure 2. Fault Detection Capabilities of Test Inputs.



Test Cases	Fault									
	1	2	3	4	5	6	7	8	9	10
000	X							X		
001		X							X	
010		X	X	X	X	X				
011			X		X		X			
100	X	X	X							
101							X			X
110										
111	X							X	X	X

Test Cases	Fault									
	1	2	3	4	5	6	7	8	9	10
000	X							X		
001		X							X	
010		X	X	X	X	X				
011			X		X		X			
100	X	X	X							
101							X			X
110										
111	X							X	X	X

Test Cases	Fault									
	1	2	3	4	5	6	7	8	9	10
000	X							X		
001		X							X	
010		X	X	X	X	X				
011			X		X		X			
100	X	X	X							
101							X			X
110										
111	X							X	X	X

Figure 3. Elimination of Faults Using Prioritized of Test Inputs.

First, test input 010 was selected to be executed because it is able to detect the most number of faults (five faults). Then, test input '111' is selected because it is able to detect the most number of remaining faults. Lastly, either test input '011' or '101' can be selected for execution as they can detect the same number of faults which is fault 7.

From the above examples, it can be observed that the *Sequential Order* requires six out of eight test inputs to be executed (75% of all test inputs) while the greedy method prioritization only requires three out of eight test inputs (37.5% of all test inputs). This reduction can significantly reduce the time and cost in testing.

As it is desirable to detect more common faults as early as possible in the testing process, the Average Percentage of Fault Detected (APFD) [13] has been used to measure the rates (how fast) both Sequential Order and the greedy method prioritization discover all faults. Let

TF_n = total number of fault discovered in n sequence
TTF = total number of fault test cases

$$APFD = \frac{TF_1 + TF_2 + TF_3 + \cdots + TF_n}{n \times TTF} \times 100\%$$

Based on the above formula, the APFD for Sequential Order and greedy method prioritization can be calculated as below. The higher the APDF value, the faster is the rate at which the sequences of test inputs are detecting faults.

APFD for Sequential Order:

$$APFD = \frac{2+4+8+9+9+10+10+10}{8 \times 10} \times 100\%$$

$$APFD = 77.5\%$$

APFD for greedy method prioritization:

$$APFD = \frac{5+9+10+10+10+10+10+10}{8 \times 10} \times 100\%$$

$$APFD = 92.5\%$$

From the APFDs, it is evident that the greedy method prioritization detects faults at significantly higher rate compared to the sequential method.

IV. Dynamic Fault Visualization Tool

The Dynamic Fault Visualization Tool has been developed to assist software testing practitioners to select and prioritize test inputs for execution using the greedy method described in Section III. With this Dynamic Fault Visualization Tool, software testing practitioners can visually identify the best test input, one that can detect the most number of remaining faults dynamically. To use this tool, firstly, the tool user needs to provide the Boolean expression to be tested. The tool user is able to select types of common faults they would like to detect. After that, the tool will automatically generate all possible the faults based on the fault types using mutation technique. Finally, the tool will generate a grayscale colour images to represent the number of faults each test input can detect). The UML Interaction Diagram of the Dynamic Fault Visualization Tool is shown in Figure 4.

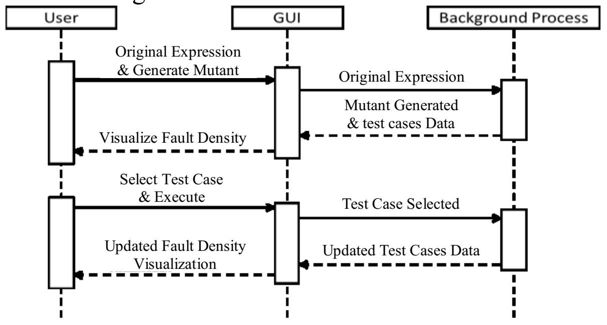
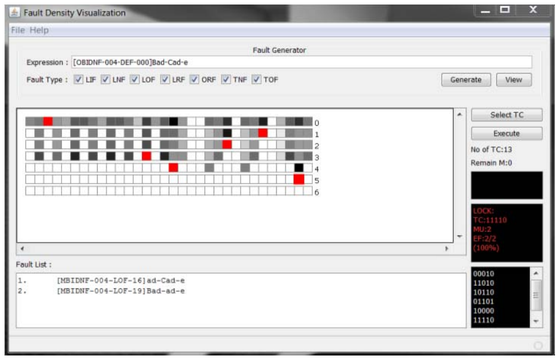

User
GUI
Background Process
Original Expression & Generate Mutant
Original Expression
Mutant Generated & test cases Data
Visualize Fault Density
Select Test Case & Execute
Test Case Selected
Updated Fault Density Visualization
Updated Test Cases Data

Figure 4. Elimination of Faults Using Prioritized of Test Inputs.



Expression : [OBIDNF-004-DEF-000]Bad-Cad-e

Fault Type : ☑ LIF ☑ LNF ☑ LOF ☑ LRF ☑ ORF ☑ TNF ☑ TOF

No of TC:13

Remain M:0

LOCK:
TC:11110
MU:2
EF:2/2
(100%)

```
1.      [MBIDNF-004-LOF-16]ad-Cad-e
2.      [MBIDNF-004-LOF-19]Bad-ad-e
```

```
00010
11010
10110
01101
10000
11110
```

Figure 5. Screenshots of the Dynamic Fault Visualization Tool.

Figure 5. shows the screenshot of the Dynamic Fault Visualization Tool that have been developed. The tool user is able to select which test input to be executed by clicking the boxes that represent test inputs on a grayscale colour image.

In Figure 5. , each row of boxes represents all possible test inputs. The numbers at the end of the row indicate the test input execution sequence as well as the number of test inputs that have been executed. Each box on the row is shaded based on a greyscale colour value which is calculated with the following formula.

$$T_n\ Color = 255 - \left(\frac{T_n\ No\ of\ Mutant}{Highest\ number\ of\ Mutant} \times 255 \right)$$

In this 8-bit greyscale colour scheme, 255 is the highest value which is white in colour, while 0 is the lowest value which is black in colour. Darker boxes on the row represent test inputs that can detect more faults while brighter boxes represent test inputs that can detect fewer faults. Executed test inputs are marked as red boxes. This grayscale image is updated dynamically after each test input is executed.

When all the boxes on the lowest row are white in colour, it means that all the faults have been covered by the executed test inputs.

As shown in Figure 5. , the fault list at the bottom of the visualization tool indicates the faults that can be detected by the currently selected test input. If execution of this test input causes a failure in the software under test, then it implies that the software under test contain one of the faults listed. This information can be used to locate the fault in the debugging process.

V. Performance Evaluation

To evaluate the performance of the Dynamic Fault Visualization Tool, the test has been performed on the 20 Boolean expressions extracted from TCASII shown in Figure 1. The “percentage of test inputs used to detect all faults” and APFD have been used as the performance metrics to evaluate the performance of greedy method prioritization implemented in this Dynamic Fault Visualization Tool. The performance Sequential Order and the *Static Fault Density Visualization* technique proposed in previous study [14] have been presented in TABLE II. for comparison.

TABLE II. shows the percentage of test inputs used to detect all faults. It can be observed that the proposed Dynamic Fault Visualization Tool significantly outperformed both Static Fault Density Visualization and Sequential Order. In average, it only requires 5.81% of the total test inputs to detect all seven types of common faults defined Section II, compared to 29.96% for Static Fault Visualization and 79.90% for Sequential Order.

The APFD (Average Percentage of Fault Detected) for the proposed Dynamic Fault Visualization Tool has also been compared to the Static Fault Density Visualization and Sequential Order. The results are shown in TABLE III.

From TABLE III. it can be observed that Dynamic Fault Visualization Tool consistently outperformed both Static Fault Density Visualization and Sequential Order. It has an average APFD of 98.8%. which is considerably higher than an average APFD of 94.1% for the Static Fault Density Visualization technique. On the other hand, Sequential Order performed poorly with an average APFD of 66.69%.

TABLE II. PERCENTAGES OF TEST INPUTS USED TO DETECT ALL FAULTS

Boolean Expressions	Sequential Order	Dynamic Fault Visualization	Static Fault Density Visualization
BE1	94.53%	14.06%	44.53%
BE2	89.26%	16.60%	99.22%
BE3	77.37%	1.61%	69.73%
BE4	96.88%	18.75%	18.75%
BE5	93.55%	4.10%	31.84%
BE6	92.33%	2.83%	64.45%
BE7	90.72%	2.54%	35.94%
BE8	93.36%	14.06%	58.59%
BE9	89.84%	12.50%	17.19%
BE10	83.70%	0.29%	1.57%
BE11	75.44%	0.98%	5.37%
BE12	93.80%	1.03%	15.97%
BE13	56.27%	0.27%	9.18%
BE14	70.31%	7.03%	25.00%
BE15	45.90%	2.73%	31.25%
BE16	60.93%	0.95%	30.84%
BE17	67.24%	0.59%	3.47%
BE18	68.46%	1.27%	14.84%
BE19	68.36%	4.69%	6.64%
BE20	89.84%	9.38%	14.84%
Average	**79.90%**	**5.81%**	**29.96%**

TABLE III. PERCENTAGES OF TEST INPUTS TO DETECT ALL FAULTS

Boolean Expressions	Sequential Order	Dynamic Fault Visualization	Static Fault Density Visualization
BE1	61.24%	97.35%	91.95%
BE2	61.00%	96.77%	83.56%
BE3	72.86%	99.71%	84.70%
BE4	62.56%	95.56%	94.44%
BE5	41.79%	99.13%	92.50%
BE6	61.32%	99.57%	95.23%
BE7	61.42%	99.46%	95.46%
BE8	66.09%	97.13%	87.03%
BE9	82.53%	97.52%	95.49%
BE10	54.32%	99.95%	99.62%
BE11	59.02%	99.83%	99.01%
BE12	75.47%	99.82%	96.78%
BE13	93.26%	99.94%	97.96%
BE14	63.45%	98.04%	92.73%
BE15	77.53%	99.29%	90.67%
BE16	76.61%	99.80%	91.86%
BE17	59.94%	99.85%	99.37%
BE18	60.94%	99.68%	97.21%
BE19	61.15%	99.08%	98.73%
BE20	81.22%	98.47%	98.01%
Average	**66.69%**	**98.80%**	**94.12%**

VI. CONCLUSION

In this paper, a fault-based testing technique has been proposed for Boolean expressions based on dynamic fault visualization. A simple greedy method is used to select and prioritize test inputs for fault-based testing. Hence, it is accessible to average software testing practitioners who do not have the in-depth knowledge in Boolean algebra and complex logics derivations required to implement existing fault-based testing techniques.

The Dynamic Fault Visualization Tool has been developed to implement the proposed fault-based testing techniques and generate fault list to guide the debugging process. To evaluate the performance of the Dynamic Fault Visualization Tool, the experiment has been done on it to test the Boolean expressions extracted from TCASII, a real life aviation software system.

The evaluation results show that the proposed Dynamic Fault Visualization Tool significantly outperformed both Static Fault Density Visualization and Sequential Order. By using the Dynamic Fault Visualization Tool, software testers only need to run an average of 6% of all possible test inputs to guarantee the detection of all common faults. This represent a significant saving in both time and cost of testing compared to Static Fault Density and Sequential Order which require 30% and 80% of test inputs respectively. Combined with the simplicity in method and debugging guide, the proposed dynamic fault visualization technique and tool would be very useful to software testing practitioners.

ACKNOWLEDGMENT

This work is supported by Malaysian Government MOHE FRGS (FRGS/2/2010/TK/SWIN/02/03).

REFERENCES

[1] B. Hailpern, P. Santhanam, Software debugging, testing, and verification, IBM Systems Journal, Vol. 40, No1. 2002.

[2] T. Y. Chen, M. F. Lau, and Y. T. Yu, MUMCUT: A fault-based strategy for testing Boolean Specifications. In Proceedings of Asia-Pacific Software Engineering Conference (APSEC ’99). 606–613. 1999.

[3] P. E. Black, V. Okun, and Y. Yesha, Mutation of model checker specifications for test generation and evaluation. In Proceedings of Mutation. 14–20. 2000.

[4] T. Tsuchiya, and T. Kikuno, On fault classes and error detection capability of specification-based testing. ACM Transaction on Software Engineering Method. 11, 1 (Jan.), 58–62. 2002.

[5] E. Weyuker, T. Goradia, A. Singh, Automatically generating test data from a Boolean specification, IEEE Transactions on Software Engineering, Vol. 20, No.5, pp353-363, 1994.

[6] A. Gargantini and E. Riccobene, Automatic model driven animation of SCR specifications. In Proceedings of the Sixth International Conference on Fundamental Approaches to Software Engineering (FASE 2003). Lecture Notes in Computer Science, vol. 2621. Springer, Berlin, Germany, 294–309. 2003.

[7] A. J. Offutt, S. Liu, A. Abdurazik and P. Ammann, Generating test data from state-based specifications. Software Testing, Verification and Reliability. 13, 1 (Mar.), 25–53. 2003.

[8] M. F. Lau and Y. T. Yu, An extended fault class hierarchy for specification-based testing. ACM Transaction on Software Engineering and Methodology, 14(3):247 – 276, 2005.

[9] Y. T. Yu, M. F. Lau, and T. Y. Chen, Automatic generation of test cases from Boolean specifications using the MUMCUT strategy. Journal of Systems and Software, 79(6):820–840, 2006.

[10] Z. Chen, T. Y. Chen, and B. Xu, A revisit of fault class hierarchies in general boolean specifications. ACM Trans. Softw. Eng. Methodol. 20, 3, Article 13, August 2011.

[11] Y.T. Yu, M.F. Lau, Fault-based test suite prioritization for specification-based testing, Information and Software Technology, Volume 54, Issue 2, Pages 179-202, ISSN 0950-5849, February 2012

[12] W. V. Quine, The problem of simplifying truth functions. American Mathematical Monthly, 59:521–531. 1952.

[13] S. Elbaum, A. G. Malishevsky, G. Rothermel, Test Case Prioritization: A Family of Empirical Studies, IEEE Transactions on Software Engineering, v.28 n.2, p.159-182, February 2002.

[14] K.Y. Sim, C. S. Low, and M. L. D. Wong. Visualization of Fault Density for Specification-based Testing. Proceedings of the 4th Malaysian Software Engineering Conference (MySEC'08), Kuala Terengganu, Malaysia, 2008.

BE1	$\overline{(ab)}\left(d\bar e\bar f+\bar d\bar e\bar f+\bar d\bar e\bar f\right)\left(ac(d+e)h+a(d+e)\bar h+b(e+f)\right)$
BE2	$\left(a\left((c+d+e)g+af+c(f+g+h+i)\right)+(a+b)(c+d+e)i\right)\overline{(ab)}\,\overline{(cd)}\,\overline{(ce)}\,\overline{(de)}\,\overline{(fg)}\,\overline{(fg)}\,\overline{(f\imath)}\,\overline{(gh)}\,\overline{(h\imath)}$
BE3	$\left(a\left(\bar d+\bar e+de\overline{(\bar f gh\bar\imath+\bar g h\imath)}\,\overline{(\bar f glk+\bar g\bar\imath k)}\right)+\overline{(\bar f gh\bar\imath+\bar g h\imath)}\,\overline{(\bar f glk+\bar g\bar\imath k)}(b+c\bar m+f)\right)\left(a\bar b\bar c+\bar a b\bar c+\bar a\bar b c\right)$
BE4	$a\left(\bar b+\bar c\right)d+e$
BE5	$a\left(\bar b+\bar c+bc\overline{(\bar f gh\bar\imath+\bar g h\imath)}\,\overline{(\bar f glk+\bar g\bar\imath k)}\right)+f$
BE6	$\left(\bar a b+a\bar b\right)\overline{(cd)}\left(f\bar g\bar h+\bar f g\bar h+\bar f\bar g\bar h\right)\overline{(\jmath k)}\left((ac+bd)e\left(f+\left(i(gj+hk)\right)\right)\right)$
BE7	$\left(\bar a b+a\bar b\right)\overline{(cd)}\,\overline{(gh)}\,\overline{(\jmath k)}\left((ac+bd)e\left(\bar\imath+\bar g\bar k+\bar\jmath\left(\bar h+\bar k\right)\right)\right)$
BE8	$\left(\bar a b+a\bar b\right)\overline{(cd)}\,\overline{(gh)}\left((ac+bd)e\left(fg+\bar f h\right)\right)$
BE9	$\overline{(cd)}\left(\bar e f\bar g\bar a\left(bc+\bar b d\right)\right)$
BE10	$a\bar b\bar c\bar d\bar e f\left(g+\bar g(h+i)\right)\overline{(\jmath k+\bar\jmath l+m)}$
BE11	$a\bar b\bar c\left(\overline{\left(f\left(g+\bar g(h+\imath)\right)\right)}+f\left(g+\bar g(h+i)\right)\bar d\bar e\right)\overline{(\jmath k+\bar\jmath l\bar m)}$
BE12	$a\bar b\bar c\left(f\left(g+\bar g(h+i)\right)(\bar e\bar n+d)+\bar n(jk+\bar\jmath l\bar m)\right)$
BE13	$a+b+c+\bar c\bar d ef\bar g\bar h+i(j+k)\bar l$
BE14	$ac(d+e)h+a(d+e)\bar h+b(e+f)$
BE15	$a\left((c+d+e)g+af+c(f+g+h+i)\right)+(a+b)(c+d+e)i$
BE16	$a\left(\bar d+\bar e+de\overline{(\bar f gh\bar\imath+\bar g h\imath)}\,\overline{(\bar f glk+\bar g\bar\imath k)}\right)+\overline{(\bar f gh\bar\imath+\bar g h\imath)}\,\overline{(\bar f glk+\bar g\bar\imath k)}\,(b+c\bar m+f)$
BE17	$(ac+bd)e\left(f+\left(i(gj+hk)\right)\right)$
BE18	$(ac+bd)e\left(\bar\imath+\bar g\bar k+\bar\jmath\left(\bar h+\bar k\right)\right)$
BE19	$(ac+bd)e(fg+\bar f h)$
BE20	$\bar e f\bar g\bar a(bc+\bar b d)$

Figure 1. Boolean expressions for Logics and Rules in TCASII.