2015 IEEE International Conference on Software Quality, Reliability and Security - Companion

Boolean Differentiation for Formalizing Myers' Cause-Effect Graph Testing Technique

Tolga Ayav and Fevzi Belli
Department of Computer Engineering
Izmir Institute of Technology
Izmir, Turkey
{tolgaayav, fevzibelli}@iyte.edu.tr

***Abstract*—Cause-Effect Graph Testing is a popular technique used for almost four decades. Based on Boolean algebra, this technique assists deriving test cases from a given specification informally written in a natural language. The present paper suggests Boolean differentiation for formalizing this technique. The new approach is applied to an example, borrowed from G. Myers, for demonstrating and analyzing its features. Evaluations show that the new approach outperforms Myers' approach in terms of the detected faults per test cases.**

Keywords— Cause-Effect Graph; Software Testing; Formalization; Boolean Difference; MCDC analysis;

I. Introduction and Related Work

Testing is supposed to reveal weaknesses and flaws in a product. Critical parts in software that can cause such impairments are twofold. The calculation parts of a program that deliver the expected results, i.e., data flow on the one side, and selections that change the linearity of the program, i.e., control flow on the other side.

This paper is about testing control flow that represents the logic relations of data to enable decisions leading to different paths of calculation. Here, Boolean algebra delivers useful notions and operational means. Well-known techniques of this category include meaningful coverage metrics like Condition/Decision Coverage [6][7]. Other software testing techniques using Boolean algebra and related methods are Meaningful Impact strategy (MI), Branch Operator Strategy (BOR), BOR+MI and Multiple Unique True Point/Multiple Near False Point (MUMCUT) [6].

Requirements are in practice commonly informally specified. A popular test technique, introduced by G.J. Myers in the seventies of the last century, visualizes logic relations between causes and their effects in specifications by Boolean algebra-based operations and symbols. Myers enriches this visual model, called *cause-effect graph* (*CEG*), by constraints given by user requirements and then uses for deriving test cases.

CEG technique is straight-ahead and easy to understand, which explains its popularity. The difficult part is the production of CEG that needs considerably effort and practical experience. The analysis of the CEG for producing test is explained rather intuitional than algorithmically, although some work tried to form it more precisely [3]. This situation motivated the authors of this present paper to formalize CEG technique. Very briefly explained, the formalization suggested by this work makes use of Boolean differentiation method [8] for precisely defining logic relations and efficiently deriving test cases from CEG.

The approach introduced in this paper has two promising features. First, it is algorithmic, and thus enables automatizing a great part of the Myers' test technique. Second, as it is based on sound mathematics, it avoids the error-proneness and redundancy that often encounter manual, intuitional work. To demonstrate these both features, an example is borrowed from Myers' original work [4]. The analysis of the results of the experiments carried out with both test cases shows that the suggested formalization has also considerably good potential for cost saving by considerably decreasing the number of test cases.

The next section summarizes the CEG testing technique before Section 3 formally introduces Boolean algebraic notions used in the paper, including Boolean differentiation. Based on these notions, test case generation is explained also in this Section 3. Section 4 applies the approach introduced in this paper to a comprehensive example taken from Myers' original work. The comparison shows the merits of the formalization effort. Section 5 self-critically discusses the new approach and concludes the paper by giving also hints for the needs of future research.

II. Cause-Effect Graphs

Cause-Effect Graphs (CEG) can be considered as a formal language into which an informal specification written in a natural language is translated. Elements of CEG represent logic expressions in a visual notation as shown in Fig. 1.

A *cause* is binary-valued atomic sentence in the specification and represents a state or an event. An *effect* is also a binary-valued atomic sentence representing an output state or action. For each feature in the software specifications, we isolate causes that influence this feature's behaviors. The causes and effects are then connected with Boolean operators and so a graphical representation is derived. For a more

978-1-4673-9598-4/15 $31.00 © 2015 IEEE
DOI 10.1109/QRS-C.2015.31

detailed description and examples of cause-effect graphs, refer to Myers' book [4].

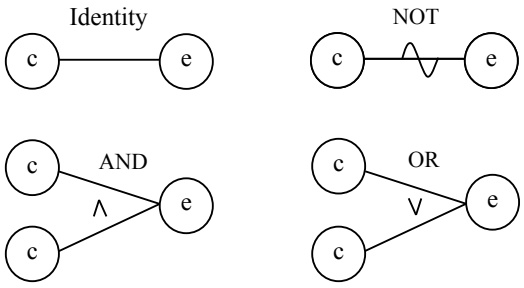

Fig. 1. Basic operations of cause-effect graphs.

The first step of CEG testing identifies causes and effects in the requirements, and lists all the causes, effects and constraints; connects those leading to the graph, considering the relationships. Analysis of CEG enables to generate the test cases. The algorithm Myers proposed [3] considers the structure of the graph to generate the test cases.

Apart from testing, CEG helps with identifying the inconsistencies in the requirements in the first place.

The graph can be augmented with *constraints* that describe the combinations of causes and effects that are impossible due to syntactic or environmental constraints. Finally, the graph is converted into a decision table where each column represents an input combination, i.e., a test case. The conversion of the graph into the decision table is the key. At this point, several heuristics can be applied to avoid combinatorial explosion of tests.

In most cases, certain combination of causes can be impossible due to syntactic or environmental considerations. Fig. 2 shows the possible constraints.

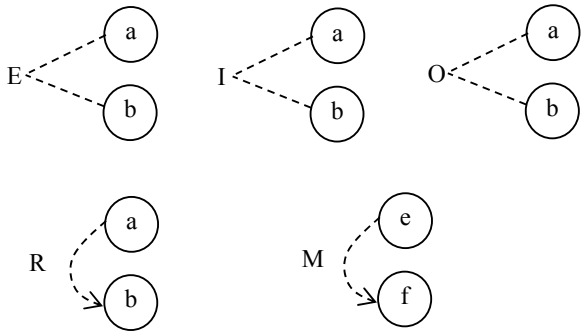

Fig. 2. Constraints on causes and effects.

The *E constraint* (Exclusive-OR) means that, at most, one of `a` and `b` can be true. The *I constraint* (Inclusive-OR) states that at least one of `a` and `b` must be true. The *O constraint* states that one and only one of `a` and `b` must be 1 (True). The *R constraint* states that for `a` to be 1, `b` must be 1, in other words, it is impossible for `a` to be 1 and `b` to be 0 (False). The *M constraint* is used to describe the relations among effects. It states that if effect `e` is 1, effect `f` is forced to 0.

III. Test Case Generation

Myers' test derivation method is shown to have some drawbacks by several studies [2] [3]. This problem can be overcome by exploiting the fault-based testing techniques for Boolean expressions [6], such as Meaningful Impact strategy (MI), Branch Operator strategy (BOR), and other strategies such as BOR+MI, Multiple Unique True Point/Multiple Near False Point (MUMCUT) and Modified Condition/Decision Coverage (MCDC). For example, Paradkar et. al. [3] propose the CEG-BOR strategy that integrates BOR strategy with CEG and analyzes the system comprehensively. This paper proposes a new methodology based on Masking MCDC criteria, which contributes to the derivation of test cases from a given CEG.

The approach proposed in this paper starts with extracting the Boolean expressions from the CEG. In particular, there exists one expression for each effect in the graph. Then MCDC analysis is performed to find the tests and finally the test suite will be optimized. Note that MCDC analysis can be defined using Boolean differentiation, which forms the mathematical basis of the methodology.

A. Boolean Algebra

In this work, Boolean variables and operations are noted and defined as follows:

$$x ::= 0 \,|\, 1 \,|\, x' \,|\, x_1 \Theta x_2, \qquad \text{where } \Theta = \{\cdot, +\}.$$

Operators, x', " $\cdot$ " and " $+$" denote negation, `AND` and `OR` respectively. Other operations can be derived from these three essential ones. For example, `XOR` operation can be defined such that $x \oplus y = xy' + x'y$. We define a Boolean function $\varepsilon : B^n \to B$ where $B = \{0,1\}$.

B. Boolean Differentiation

Let $\mathbf{c} = [c_1, c_2, \ldots, c_n]^T$ denote the vector of n causes. The differences (or derivatives) of a Boolean function $\varepsilon(\mathbf{c})$ with respect to $c_i \in \mathbf{c}$ is defined as [8]:

$$\frac{\partial \varepsilon}{\partial c_i}(\mathbf{c}) \;=\; \varepsilon(c_i = 0) \;\oplus\; \varepsilon(c_i = 1) \qquad (1)$$

For instance, for a given $y = \varepsilon(\mathbf{c}) = c_2 + c_3$, we can compute $\partial y / \partial c_2 = 1 \oplus c_3 = c_3'$. This result provides a useful insight into how c_3 sensitizes the output. If the derivative is 1, the output is sensible to c_2. If $c_3 = 0$ then the output is sensible to c_2 and vice versa. When computing Boolean derivatives, the chain rules given with Equation (2) and (3) are quite useful since we can write one Boolean function corresponding to each effect and also to intermediate nodes in the graph as can be

seen in the example of the next section. The chain rules may facilitate the computation of the derivatives of large graphs [9].

$$u = f \cdot g \quad \Rightarrow \quad \frac{\partial u}{\partial c_i} = f \frac{\partial g}{\partial c_i} \oplus g \frac{\partial f}{\partial c_i} \oplus \frac{\partial f}{\partial c_i} \frac{\partial g}{\partial c_i} \tag{2}$$

$$u = f + g \quad \Rightarrow \quad \frac{\partial u}{\partial c_i} = f' \frac{\partial g}{\partial c_i} \oplus g' \frac{\partial f}{\partial c_i} \oplus \frac{\partial f}{\partial c_i} \frac{\partial g}{\partial c_i} \tag{3}$$

The following rules are also very useful in calculations:

$$\frac{\partial f}{\partial c_i} = \frac{\partial f'}{\partial c_i} \tag{4}$$

$$a \oplus 0 = a, \qquad a \oplus 1 = a', \qquad ab \oplus b = a'b \tag{5}$$

C. Masking Modified Condition/Decision Coverage (MCDC) Analysis

MCDC states that "Every point of entry and exit in the program has taken on all possible outcomes at least once, and each condition has been shown to independently affect the decision's outcome". Masking MCDC is a form of MCDC that allows all possible forms of masking to be used to show a condition's independence. For all forms of MCDC together with their comparisons and detailed analyses, please refer to [7]. Among various forms, we use masking MCDC since it has been proved that it produces less test cases while the fault coverage remains high. MCDC is suitable to be defined formally with mathematics as shown next.

The Boolean derivative of effect $\varepsilon(\mathbf{c})$ with respect to c_i is *False* if toggling only c_i does not toggle the effect, and is *True* if toggling c_i does toggle the effect. Hence the test pairs can be found such that:

i. Cause i must be different in both tests ($x_i = y_i'$).

ii. The effect must be different in both tests ($f(\mathbf{x}) = f'(\mathbf{y})$).

iii. For test $\mathbf{x}$, cause i must change the effect ($\frac{\partial f}{\partial x_i}(\mathbf{x}) = 1$).

iv. For test $\mathbf{y}$, cause i must change the effect ($\frac{\partial f}{\partial y_i}(\mathbf{y}) = 1$).

The test pairs are also known as *independence pairs*. An independence pair is such a combination of truth vectors that the related cause toggles in two vectors while all other conditions are fixed or masked. For example, assume that an effect has the expression of $\varepsilon(\mathbf{c}) = c_1 c_2 c_3$. To find an independence pair for c_1, both c_2 and c_3 must be masked, i.e. they must be both True. In this case, $\varepsilon(\mathbf{c}) = c_1$, meaning that the effect will directly follow c_1.

D. Adding Constraints into Boolean Expressions

The graph may contain various constraints as presented in the previous section. These constraints must be incorporated into Boolean expressions before computing the derivatives. We demonstrate how the E constraint seen in Fig. 2 can be incorporated.

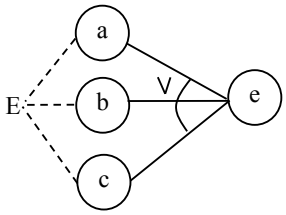

Fig. 3. Exclusive-OR Constraint.

The E constraint seen in Fig. 3 states that it must always be true that, at most, one of `a`, `b` and `c` can be 1. To identify this constraint, the following expression can be written:

$$\gamma_E(a,b,c) = a'b' + a'c' + b'c' \tag{6}$$

If we augment the Boolean expression with this constraint, the modified expression becomes:

$$\begin{aligned} e &= \varepsilon(a,b,c) = a + b + c \\ e^* &= \gamma_E(a,b,c)\, \varepsilon(a,b,c) \\ &= (a'b' + a'c' + b'c')\,(a + b + c) \end{aligned} \tag{7}$$

Applying the chain rule given in (2), the derivative of `e` with respect to cause `a` would be as follows:

$$\begin{aligned} \frac{\partial e^*}{\partial a} &= \varepsilon \frac{\partial \gamma_E}{\partial a} \oplus \gamma_E \frac{\partial \varepsilon}{\partial a} \oplus \frac{\partial \gamma_E}{\partial a} \frac{\partial \varepsilon}{\partial a} \\ &= bc' + b'c + b'c'. \end{aligned} \tag{8}$$

The analysis and formalization of the other constraints are carried out similarly. To keep the paper concise and easy to read the analysis, they are not included in the paper.

IV. Example – Comparison by Experiments

A well-known example, the "*change* subcommand" is borrowed from Myers' book. Refer to [3] for a detailed explanation of this example. The cause-effect graph of *change* is shown in Fig. 4. The graph has nine causes ($c_1, c_2, \cdots, c_9$) and four effects (e_1, e_2, e_3, e_4). The inner nodes are named as i_1, i_2, i_3. To apply our methodology, Boolean expression

equivalent to the graph must be determined. For example, the following Boolean expressions for effect e_1 can be written:

$$i_1 = c_3 + c_4 + c_5,\ \ i_2 = c_6 + c_7 + c_8,\ \ i_3 = c_1 c_2 i_1 i_2,$$

$$e_1 = i_3 c_9,$$

$$e_1 = \varepsilon_1(\mathbf{c}) = c_1 c_2 c_9 (c_3 + c_4 + c_5)(c_6 + c_7 + c_8).$$

$$\frac{\partial e_1}{\partial c_1} = c_9 \frac{\partial i_3}{\partial c_1} = c_9 c_2 i_1 i_2$$

$$= c_2 (c_3 + c_4 + c_5)(c_6 + c_7 + c_8) c_9.$$

If the exclusive-OR constraints $\gamma_E(c_3, c_4, c_5)$ and $\gamma_E(c_6, c_7, c_8)$ are added to the expression e_1, the derivative of it with respect to c_1 becomes:

$$\frac{\partial e_1^*}{\partial c_1} = (c_3' c_4' + c_3' c_5' + c_4' c_5')(c_6' c_7' + c_6' c_8' + c_7' c_8')$$
$$c_2 (c_3 + c_4 + c_5)(c_6 + c_7 + c_8) c_9.$$

The set of possible test vectors is the solution to the following problem:

$$\frac{\partial e_1^*}{\partial c_1} = 1 \qquad (9)$$

A simple calculation finds out that there exist nine different solutions, i.e. nine test pairs satisfying the above problem. Table I shows the entire solution set. For cause 2, we would have again nine test pairs, for cause 3, 4, 5, 6, 7 and 8 we would find three pairs for each and for cause 9, we would have nine test pairs. Therefore, the total number of possible test pairs is 45, i.e. leading to 90 tests. Please note that the maximum number of tests is $2^9 = 512$. However, many test vectors are supposed to be equivalent in these 90 test vectors. Since MCDC proposes to find one test pair for each cause, one must initially have 18 tests. We should select the smallest set out of 90 tests so that the total number of tests will be minimal.

Once all test pairs are computed for nine causes, the optimization minimizes the number of test cases. For example, among the set of possible test pairs given in Table I, the optimization decides to choose the first one, which constitutes the first two tests shown in Table II.

The CHANGE subcommand is used to modify a character string in the current line of the file being edited. Syntax: C /str1/str2. Cause1: The first nonblank character following "C". Cause2: The command contains exactly two "/" characters. Cause3: Str1 has length one, Cause2: Str1 has length 30, Cause3: Str1 has length 2-29. Cause4: Str2 has length zero, Cause5: Str2 has length 20, Cause6: Str2 has length 1-29. Effect1: The changed line is typed. Effect2: The first occurence of str1 is replaced by str2. Effect3: "Not Found" printed. Effect4: "Invalid Syntax" printed.

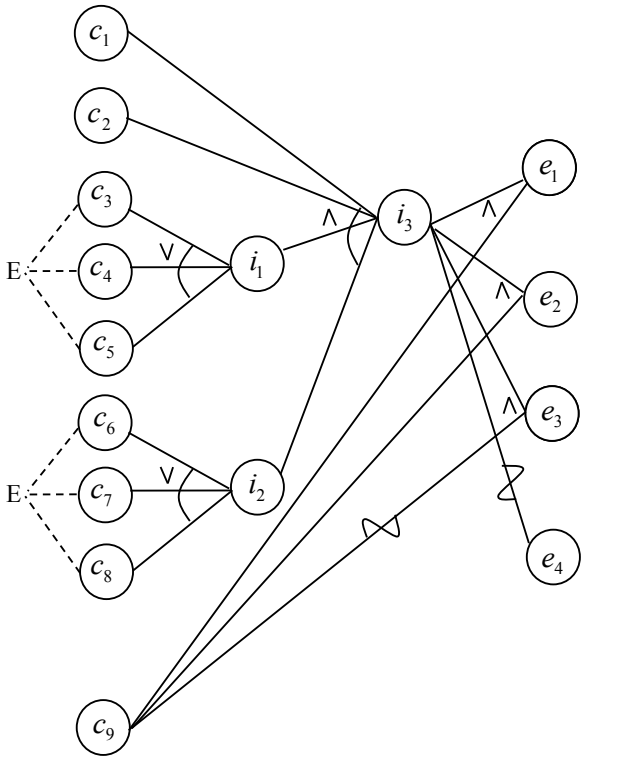

Fig. 4. Cause-effect graph of the *change* subcommand.

TABLE I. ALL POSSIBLE TEST PAIRS FOR TESTING CAUSE 1

	Causes									Effects			
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**1**	**2**	**3**	**4**
1	0	1	1	0	0	1	0	0	1	0	0	0	1
	1	1	1	0	0	1	0	0	1	1	1	0	0
2	0	1	0	1	0	1	0	0	1	0	0	0	1
	1	1	0	1	0	1	0	0	1	1	1	0	0
3	0	1	0	0	1	1	0	0	1	0	0	0	1
	1	1	0	0	1	1	0	0	1	1	1	0	0
4	0	1	1	0	0	0	1	0	1	0	0	0	1
	1	1	1	0	0	0	1	0	1	1	1	0	0
5	0	1	0	1	0	0	1	0	1	0	0	0	1
	1	1	0	1	0	0	1	0	1	1	1	0	0
6	0	1	0	0	1	0	1	0	1	0	0	0	1
	1	1	0	0	1	0	1	0	1	1	1	0	0

7	0	1	1	0	0	0	0	1	1	0	0	0	1
	1	1	1	0	0	0	0	1	1	1	1	0	0
8	0	1	0	1	0	0	0	1	1	0	0	0	1
	1	1	0	1	0	0	0	1	1	1	1	0	0
9	0	1	0	0	1	0	0	1	1	0	0	0	1
	1	1	0	0	1	0	0	1	1	1	1	0	0

TABLE II. TEST SUITE FOR THE *CHANGE* SUBCOMMAND EXAMPLE

	Causes									Effects			
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**1**	**2**	**3**	**4**
1	0	1	1	0	0	1	0	0	1	0	0	0	1
2	1	1	1	0	0	1	0	0	1	1	1	0	0
3	1	0	1	0	0	1	0	0	1	0	0	0	0
4	1	1	0	0	0	1	0	0	1	0	0	0	1
5	1	1	0	1	0	1	0	0	1	1	1	0	0
6	1	1	0	0	1	1	0	0	1	1	1	0	0
7	1	1	1	0	0	0	0	0	1	0	0	0	1
8	1	1	1	0	0	0	1	0	1	1	1	0	0
9	1	1	1	0	0	0	0	1	1	1	1	0	0
10	1	1	1	0	0	1	0	0	0	0	0	1	0

A. Comparison of Myers' Technique with the New Approach

Myers constructed **22** tests for the *change* subcommand in his book [3], which contains the complete table of these tests. The **10** test cases generated by the new approach (see Table II) are sufficient to test the system according to Masking Modified Cause/Effect Criteria.

It is desirable to keep the number of tests minimal while achieving higher fault coverage. The effectiveness of the test suites can be compared by means of two metrics, the *number of tests* and the *fault detection capability* of them. Two metrics are contradictory. The number of the tests depends on the test case generation method and the fault coverage highly depends on the fault assumption.

In the example above, the test suite derived using the proposed approach outperforms Myers' technique in terms of the number of tests. However, their fault coverage should be compared as well. *Fault coverage* effectiveness is generally determined by mutation analysis. A *mutant* can be created by slightly changing the original software, thus a mutant represents a fault prototype. A test case *kills* a mutant if it identifies this mutant.

There are various fault injection techniques to produce mutants as fault prototypes known from the literature [6]:

- *Operator Reference Fault (ORF)*: A binary logic operator '.' is replaced by '+' or vice versa.
- *Variable Negation Fault (VNF)*: An atomic Boolean literal is replaced by its negation.
- *Expression Negation Fault (ENF)*: A sub-expression in the statement is replaced by its negation.
- *Variable Reference Fault (VRF)*: A condition is replaced by another input that exists in the statement.
- *Stuck-at-0 (S-a-0)*: A condition is replaced by 0.
- *Stuck-at-1 (S-a-1)*: A condition is replaced by 1.

The following examples demonstrate how these faults are applied to the expressions derived from the graph. Below, one sample is given for each fault class:

$$i_1 = c_3 + c_4 + c_5 \xrightarrow{\text{ORF}} i_1 = c_3 c_4 c_5$$

$$i_2 = c_6 + c_7 + c_8 \xrightarrow{\text{VNF}} i_2 = c_6 + c_7' + c_8$$

$$e_1 = i_3 c_9 \xrightarrow{\text{ENF}} e_1 = i_3' c_9$$

$$i_3 = c_1 c_2 i_1 i_2 \xrightarrow{\text{VRF}} i_3 = c_9 c_2 i_1 i_2$$

$$e_1 = i_3 c_9 \xrightarrow{\text{s-a-1}} e_1 = i_3$$

$$i_1 = c_3 + c_4 + c_5 \xrightarrow{\text{s-a-0}} i_1 = c_3 + c_5$$

B. Setup for Evaluations

The above mentioned faults are systematically injected into the Boolean expression to create the mutants. The fault coverage performances of two test suites, produced by the new approach and Myers' approach, are compared against different fault classes. For VNF+ENF, there exist twelve nodes until the effects, hence the number of all possible mutants is $2^{12} - 1 = 4095$. In the CEG, there seems six Boolean operators, thus the maximum number of ORF faults is $2^6 - 1 = 63$. Stuck-at faults are applied to the aforementioned twelve nodes, so in these fault classes there can be 4095 mutants. In VRF class, two causes are exchanged in each mutant. Since there are nine causes, the total number of mutants can be $9 \times 8 \div 2 = 36$. When ORF and VRF are combined, $64 \times 36 = 2304$ mutants are created. Finally, for the combination of VRF, VNF and ENF, the number of mutants is $4096 \times 36 = 147456$.

C. Results of the Evaluations

Table III shows the evaluation results. The fault coverage percentages of the two approaches are exactly the same, but the number of test cases generated by the new approach is less than half of the number of the test cases generated by Myers' technique. Hence, we can conclude that our test suite considerably outperforms the Myers' suite in terms of the mutants killed (or faults detected) per test cases. Note that fault

coverage in VRF class is lower than other classes since VRF involves arbitrary exchange of causes, which is hard to detect.

TABLE III. EVALUATIONS FOR FAULT COVERAGE OF TWO METHODS

Fault Classes	# of Mutants	CEG-MCDC		Myers	
		Fault Coverage (Percent.)	# of Detected Faults per tests	Fault Coverage (Percent.)	# of Detected Faults per tests
VNF+ENF	4095	100	409.5	100	186.13
ORF	63	100	6.3	100	2.86
S-a-0	4095	100	409.5	100	186.13
S-a-1	4095	99.9969	409.4	99.9969	186.09
VRF	36	80.5556	2.9	80.5556	1.32
ORF+VRF	2304	99.6962	229.7	99.6962	104.41
VRF+VNF +ENF	147456	80.5556	118878.4	80.5556	5399.27

V. CONCLUSION, THREATS TO VALIDITY

This study proposes a methodology relying on MCDC analysis to derive test cases from CEGs. It has two benefits: 1) Masking MCDC produces less test cases with a good fault coverage compared to similar approaches, 2) The analysis is entirely mathematical. We demonstrate the method through an example borrowed from Myers' book. The evaluations show that the proposed technique outperforms the Myers' approach in terms of the detected faults per test cases.

The approach should be extended with further evaluations that compare the method with the other fault-based analyses such as BOR, MI and MUMCUT, and also include more applications in the range from small to large scale.

Since the approach has a formal basis, fault coverage and other metrics used in software testing can also be studied by means of mathematical proves rather than solely performing evaluations. Hence, future work will present further mathematical analyses supported also by extensive evaluations to prove the superiority of the method. A new CEG Testing tool is still under development for this purpose.

REFERENCES

[1] I. Chung, “Investigating Effectiveness of Software Testing with Cause-Effect Graphs,” International Journal of Software Engineering and Its Applications, Vol.8, No.7 (2014), pp.41-54.

[2] K. Nursimulu and R. L. Probert, , “Cause-Effect Graphing Analysis and Validation of Requirements,” In Proceedings of CASCON'95 , IBM Canada Ltd. and National Research Council, pp. 46, 1995.

[3] A. Paradkar, K.C. Tai and M.A. Vouk, “Specification-based Testing using Cause-Effect Graphs,” Annals of Software Engineering 4 (1997), pp. 133-157, J.C. Baltzer AG, Science Publishers.

[4] Myers, Glenford J. Art of Software Testing. First edition (1975). John Wiley and Sons, Inc.,New York, USA.

[5] John H. Reif, “Efficient VLSI fault simulation,” Computers & Mathematics with Applications, Volume 25, Issue 2, January 1993, Pages 15-32, ISSN 0898-1221, http://dx.doi.org/10.1016/0898-1221(93)90219-L.

[6] U. Badhera, G. N. Purohit and S. Taruna. “Fault Based Techniques for Testing Boolean Expressions: A Survey,” CoRR abs/1202.4836 (2012).

[7] J.J. Chilenski, “An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion,” Report No: DOT/FAA/AR-01/18, Boeing Commercial Airplane Group, April 2001.

[8] F.F. Sellers, M.Y. Hsiao, L.W. Bearnson, . “Analyzing Errors with the Boolean Difference,” IEEE Transactions on Computer, Vol. C-17, No:7 (1968), pp. 676-683.

[9] J.H. Reif, Efficient VLSI fault simulation, Computers & Mathematics with Applications, Volume 25, Issue 2, January 1993, pp. 15-32.