TEST CASE GENERATION FROM CAUSE EFFECT GRAPHS

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Deniz KAVZAK UFUKTEPE

December 2016
İZMİR

We approve the thesis of **Deniz KAVZAK UFUKTEPE**

Examining Committee Members:

Asst. Prof. Dr. Tolga AYAV
Department of Computer Engineering, İzmir Institute of Technology

---

Asst. Prof. Dr. Tuğkan TUĞLULAR
Department of Computer Engineering, İzmir Institute of Technology

---

Asst. Prof. Dr. Mutlu BEYAZIT
Department of Computer Engineering, Yaşar University

27 December 2016

Asst. Prof. Dr. Tolga AYAV
Supervisor, Department of Computer Engineering
İzmir Institute of Technology

Assoc. Prof. Dr. Y. Murat ERTEN
Head of the Department of
Computer Engineering

Prof. Dr. Bilge KARAÇALI
Dean of the Graduate School of
Engineering and Sciences

ACKNOWLEDGMENTS

I would first like to thank my thesis advisor Asst. Prof. Dr. Tolga Ayav. The door to his office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right the direction whenever he thought I needed it.

I would also like to thank to Prof. Dr. Fevzi Belli for sharing his knowledge and supporting me both academically and spiritually. Also, I would like to thank to Asst. Prof. Dr. Tuğkan Tuğlular for helping me during all phases of my master studies. Without their support, this work wouldn't be completed as it is.

Finally, I must express my very profound gratitude to my parents and to my husband for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

ABSTRACT

TEST CASE GENERATION FROM CAUSE EFFECT GRAPHS

Cause-effect graphing is a well-known requirement based testing technique. However, since it was introduced by Myers in 1979, there seems not to have been any sufficiently comprehensive studies to generate test cases from these graphs. Yet there are several methods introduced to generate test cases from Boolean expressions. This thesis proposes to convert cause-effect graphs into Boolean expressions and find out the test sets using test input generation techniques for Boolean expressions, such as MI, MAX-A, CUTPNFP, MUMCUT, Unique MC/DC and Masking MC/DC. Generated test sets are compared by using mutation analysis according to their fault detection capabilities.
Myers' original test generation technique is also implemented and included in the mutation analysis. A tool is created which allows to generate test cases by using the implemented algorithms. The tool gets a ".graphml" file representing a cause- effect graph as an input and gives the generated test set as an output. In addition, mutation analysis can be done with the implemented tool. 14 Requirements of TCAS-II are used as an experiment. Results of the mutation testing for these requirements showed that MUMCUT technique has the highest mutant detection success for all fault types. Moreover, Unique MC/DC technique has detected highest number of mutants per test case.

ÖZET

NEDEN SONUÇ ÇİZGELERİNDEN TEST GİRİŞLERİNİN ÜRETİLMESİ

Neden-sonuç çizgeleri çok bilinen gereksinim tabanlı yazılım test yöntemlerinden biri olduğu halde Myers tarafından önerildiği 1979 yılından beri bu çizgelerden test girişleri üretilmesi konusunda yeterince kapsamlı çalışma yapılmamıştır. Ancak, Boole ifadelerden test girişlerinin üretilmesi için çeşitli yöntemler tanıtılmıştır. Bu tez çalışması, çizgelerin Boole ifadelerine dönüştürülmesini ve Boole ifadelerinden test girişlerinin oluşturulması için önerilmiş olan MI, MAX-A, CUTPNFP, MUMCUT, Unique MC/DC ve Masking MC/DC yöntemlerini kullanarak test giriş kümelerinin üretilmesini önermektedir. Üretilen test giriş kümeleri mutasyon analizi ile, hata yakalama başarıları açısından kıyaslanmıştır. Myers'ın orijinal test giriş üretme yöntemi de uygulanmış ve yapılan mutasyon analizine dahil edilmiştir. Uygulanan algoritmaları kullanarak test girişlerinin üretildiği bir araç yaratılmıştır. Bu araç, çizgeyi ifade eden ".graphml" dosyasını girdi olarak alır ve üretilen test girişleri kümesini çıktı olarak verir. Ayrıca, mutasyon analizi de bu araç ile yapılabilir. Deney için TCAS-II sistemine ait 14 gereksinim kullanılmıştır. Bu gereksinimler üzerinde yapılan mutasyon analizi sonuçları MUMCUT tekniğinin tüm hata tiplerinde en yüksek mutant yakalama başarısını elde ettiğini göstermiştir. Ayrıca, Unique MC/DC tekniği, test girişi başına yakalanan mutant sayısı bakımından en yüksek değeri vermiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CEG	Cause-Effect Graph
DNF	Disjunctive Normal Form
BOR	Boolean Operator
MI	Meaningful Impact
MC/DC	Modified Condition/Decision Coverage
RC/DC	Reinforced Condition/Decision Coverage
AST	Abstract Syntax Tree
UTP	Unique True Point
NFP	Near False Point
MVF	Missing Variable Fault
VNF	Variable Negation Fault
VRF	Variable Reference Fault
ORF	Operator Reference Fault
ENF	Expression Negation Fault
MUTP	Multiple Unique True Point
MNFP	Multiple Near False Point
CUTPNFP	Corresponding Unique True Point and Near False Point Pair

CHAPTER 1

INTRODUCTION

1.1. Aim of the Thesis and Objectives

Requirement based testing (RBT) techniques are very effective while dealing with critical systems such as health systems, avionics, etc., where the requirements are well defined and not likely to change. By these techniques, test cases can be created from the defined requirement sentences, Boolean expressions representing them, models or graphs representing them, etc. Cause-effect graphing is one of the RBT techniques which we can represent the system requirements as a cause-effect graph (CEG) and then generate test cases from that graph in early phases of the software development.

The aim of this thesis is to use some of the well-known test generation methods introduced for Boolean expressions in order to generate test cases from cause-effect graphs by using the representative equivalence of the cause-effect graphs and Boolean expressions. Another aim is to compare all these methods and Myers' method according to their fault detection capabilities for some fault classes. Thereby an evaluation can be made about which test generation methods are more successful on which fault classes and which methods can be combined together in order to get a better fault detection, etc. Also their costs can be estimated according to the total number of test cases that are generated by them. In the end, there will be an indicator for the selection of these techniques, respect to the fault types that are desired to be revealed.

A tool is implemented which allows to import a cause-effect graph and to export generated test cases by one of the implemented methods. Furthermore, this tool is used to see the detection capability of the test cases that are generated.

In order to achieve these, first, a cause-effect graph representation format is formed as a ".graphml" file, by making some additions to original graphml graph definitions. Afterwards, by using an open-source library named Gephi, the conversion from cause-effect graph into Boolean expressions is made. Since some of the methods require the Boolean expression to be in disjunctive normal form (DNF), the conversion of these expressions into DNF is also made. This DNF conversion is made by using truth tables, which provides a canonical DNF. A canonical DNF can be used for the methods Meaningful Impact (MI) and MAX-A. However, for the MUMCUT method, which is the union of the methods Multiple Unique True Point (MUTP), Multiple Near False Point (MNFP) and Corresponding Unique True Point and Near False Point Pair (CUTPNFP), Boolean expression should be in Irredundant DNF. Irredundant DNF can be defined as simplified DNF. Therefore, to be able to implement MUMCUT, an open-source library named jbool_expressions for DNF conversion is also used, which makes the conversion to a simplified DNF. Myers' technique, which is the original test case generation method for cause-effect graphs is implemented. Then, the test generation methods defined for Boolean expressions: MI, MAX-A, MUTP, MNFP, CUTPNFP, MUMCUT, Unique Modified Condition/Decision Coverage (MC/DC) and Masking Unique Modified Condition/Decision Coverage (MC/DC) are implemented. Test cases are generated for each of these methods. Then, mutants of nine different fault classes are created and the generated test cases are run on these mutants. The number of mutants that each test case has detected are recorded and evaluated together to be compared.

1.2. Organization of Thesis

In Chapter 2, related works on the test generation techniques from Boolean expressions and cause-effect graphs are summarized. In Chapter 3, needed background concepts and definitions are given. The definition of cause-effect graph and its properties are given in the Section 3.1. Implemented test generation techniques and needed definitions are given in the Section 3.2, and fault classes and mutation analysis are given in detail in the Section 3.3.

In Chapter 4, the proposed cause-effect model is explained in the Section 4.1, and the Disjunctive Normal Form (DNF) conversion algorithm is given in the Section 4.2. The implementation of the mutation analysis is explained in the Section 4.3.

In Chapter 5, the results of the mutation analysis on different techniques for different fault classes is shared.

Finally, in Chapter 6, interpretation of the results and the planned future works are explained.

CHAPTER 2

RELATED WORK

Cause-effect graph is developed in order to model a system's specifications with its components, their relationships and dependencies by Elmendorf (1973). Test cases are intended to be generated from the created model. Firstly, Myers (1979) proposed a systematic technique to create a refined decision table which guarantees to cover all decisions in the model, once. With this technique, the number of needed decision table combinations are reduced. Since, in big scale projects, the number of combinations can be too much, this reduction is really valuable.

In the paper of Tai et al. (1993), a new fault-based approach to generate tests for CEGs, called BOR (boolean operator) testing is presented. This method is based on the detection of Boolean operator faults. How to generate test cases by using BOR method is shown and an empirical study on a real-time boiler control and monitoring system is done. The BOR method is shown to have effective results.

In another work, Tai (1993), discussed the existing test generation techniques for simple predicates and explains the problems of them when the predicate is compound. Then, two fault-based testing strategies are proposed which are: Boolean operator (BOR) testing, which he defined for CEGs in the past, intends to detect Boolean operator faults, and Boolean and relation operator (BRO) testing, which intends to detect Boolean operator faults and relational operator faults. The approximate number of test cases needed for BOR testing is discussed and some experiment results are shared to show their success on detection of faults.

Weyuker et al. (1994), presented a new family of test case generation techniques from Boolean expressions. Several methods are proposed including Meaningful Impact Method (MI) and MAX-A. Type of faults which these methods are expected to detect are discussed. The methods defined are compared by their powers of detecting faults in five different fault classes (variable negation fault, expression negation fault, variable reference fault, operator reference fault, associative shift fault). All proposed methods are shown to did good on detecting these fault classes. Also, the costs of these methods are compared by the number of test cases they generate.

Paradkar (1994), did some empirical studies on testing CEGs of different real software systems. The problems occurred in these studies are explained and solutions are proposed to solve these problems. Also, the experiences earned from these evaluations are discussed. In one of his next works Paradkar (1995), the previously proposed method BOR is combined with the method MI defined by Weyuker et al. (1994) in order to solve the problems of BOR when the Boolean expression is not singular. BOR is applied to the singular components of the predicate and MI is applied to the rest. The number of high test cases coming from MI could also be reduced this way.

Nursimulu and Probert (1995), discussed the problems of Myers' cause-effect graphing method and proposed some solutions to overcome these problems. A new approach by using path sensitization techniques in order to solve the problems encountered in the construction of the decision table phase is proposed. Moreover, the way of CEG technique can be used to derive use case scenarios for validating requirements is explained.

Later, Tai (1996) defined Boolean and relational expression (BRE) testing method for compound predicates in addition to BOR and BRO testing methods he proposed earlier. This new approach is intended to detect Boolean operator faults, relational operator faults, and a type of fault involving arithmetical expressions. The algorithms are given in order to generate test cases with BOR, BRO and BRE. In order to overcome the problem when a Boolean expression has multiple occurrences of same variable, a combination with MI technique defined by Weyuker et al. (1994), is described.

Chen et al. (1999), studied the MUMCUT technique which combines the MUTP, MNFP and CUTPNFP techniques which were proposed in the past. This method is shown to detect faults in seven different fault classes. It is shown that the method is highly successful on the fault classes expression negation fault, literal negation fault, term omission fault, operator reference fault, literal omission fault, but not too good on the fault classes literal insertion fault and literal reference fault. The way of generating test cases by using this technique is defined and MUMCUT is compared with MAX-A and MAX-B methods which are also shown to very effective on same fault classes. The MUMCUT technique requires the Boolean expression on the form Irredundant DNF and both MAX-A,MAX-B methods from Weyuker et al. (1994) are defined on Boolean expressions which are in DNF. Irredundant DNF is the form where there is no simpler way of presenting that Boolean expression in DNF.

In another work, Paradkar et al. (1996), proposed another technique for generating test cases automatically from predicates. It was intended to extend one of the previous works of him Tai et al. (1993), where a method called BOR was defined in order to generate test cases from Boolean expressions. Using constraint logic programming (CLP) to automatically generate test data for predicates is proposed. An incremental approach is used in order to solve constraint systems with CLP.

Paradkar et al. (1997), later investigated a different approach on their previously defined method BOR as CEG_BOR, which works on CEGs. Informal specifications are converted into CEGs, then the BOR technique is applied. Some empirical studies are done and the test detection capabilities of the test set generated by CEG_BOR, with the test sets generated by exhaustive testing, constructed by using an extended finite state machine (EFSM) representation are compared.

Chilenski and Miller (1994), described the modified condition/decision coverage (MC/DC) and how to apply it in software testing. The advantages and disadvantages of this approach are evaluated. In another work of report, Chilenski (2001) defined and compared three forms of MC/DC in order to understand their strengths and weaknesses.

Chen and Lau (2001), compared different methods as MAX-B, CUTPNFP, MUTP and MNFP by their detection capabilities of the literal insertion fault and the literal reference fault, which are the fault classes that MUMCUT was not highly effective on. Later, the effectiveness of MUMCUT on general form Boolean expressions is examined by Chen et al. (2009), excluding the requirement of the Boolean expressions must be in Irredundant DNF.

Vilkomir and Bowen (2002), proposed a new testing criterion named Reinforced Condition/Decision Coverage (RC/DC) which is a further development of MC/DC. It is stated that this new method is more suitable for safety critical systems. A formal definition for RC/DC and examples with proofs of some features are given.

Chen et al. (2003) implemented a tool that generates test cases from Boolean expressions, by using the technique they proposed, named MUMCUT. Kaminski and Ammann (2009), studied on MUMCUT and proposed another, less costly method named Minimal-MUMCUT, which can detect same fault classes as MUMCUT. It covers the criteria of MUMCUT, however, the number of the test cases is decreased.

Singh et al. (2006), examined Elmendorf's technique, BOR, MC/DC and RC/DC together in order to evaluate these test generation techniques from Boolean expressions by their fault detection capabilities. Test cases and mutants were created by using branch statements. Mutants were created by making only one operator or operand faults at the same time. The experiments showed that BOR is effective for a subset of fault types. MCDC and RCDC gave better results on all classes. Elmendorf's (CEG) was the highest since it selects all possible test cases.

In his book Aditya Mathur (2008), in addition to explaining all concepts of software testing, explained in detail how to generate test cases by cause-effect graphing, BOR, BRO and BRE tests, MC/DC testing, etc. He also explained how to combine BOR with MI by constructing an Abstract Syntax Tree (AST) of the given Boolean expression and using this AST in order to generate test cases.

Srivastava et al. (2009) discussed the original Myers' cause-effect graphing technique and surveys how a CEG is converted into a decision table. Also, it is shown how the CEG method can be used to test a software fulfills its requirement specifications or not. It is aimed to find and show the problems of existing test generation algorithm and try to understand how these problems can be solved.

Fraser and Gargantini (2010), formalized the fault detection process as a logical Satisfiability (SAT) problem and compared the solution with MUMCUT technique. In their next work Gargantini and Fraser (2011), they extended their technique so that it can be used on general form Boolean expressions and they compared it with MUMCUT and MC/DC.

Paul and Lau (2012), created new uniform fault classes in order to ignore the differences on the fault detection success of different techniques caused by the form of the Boolean expression (DNF, general form, etc.). It is stated that different techniques with different constraints on Boolean expressions' form, can be compared in a better way with uniform fault classes.

Kaminski et al. (2013), used the ROR operator to create mutants on Boolean expressions, so that they eliminated the mutual mutants and reduced the total number of mutants needed to be created. The precision of MC/DC is increased and it is stated that minimal-MUMCUT is more effective than the more commonly used MC/DC in terms of detecting faults.

Sziray (2013) developed a new algorithm to generate test cases from cause-effect graphs by using decision trees with three-valued Boolean algebra. By using don't care values, the number of decisions are reduced. The algorithm proposed is applicable to any cause-effect graph.

Vilkomir et al. (2013) measured the effectiveness of t-wise test, which is one of the combinatorial testing approaches, on Boolean expressions by using mutation analysis. The results of mutation detection successes are compared with random test generation.

Paul and Lau (2014) examined the previously defined different MC/DC forms by doing a systematic literature review (SLR) and discussed the problems encountered when there are coupled conditions. A new MC/DC form is defined. which has higher success on detecting mutants when compared to previous MC/DC variants.

Arcaini et al. (2015), worked on the optimization of test generation by using SAT and Satisfiability Modulo Theories (SMT) problem solvers. The advantages and disadvantages of using SAT and SMT problem solvers are discussed. This approach is said to be an alternative to the classical algorithmic test generation methods defined before. Dealing with constraints are easier with these approaches, however the amount of time and space is higher, sometimes making it even infeasible. The optimization worked on is said to be promising in order to use these approaches in the future for test generation from Boolean expressions.

Chung (2014a), created new specified fault classes for cause-effect graphs and compared Myers' technique with combinatorial testing technique by their success on fault detecting on these new fault classes. Later, Chung (2014b), used a SAT solver to generate test cases by pairwise testing technique, which is again a combinatorial testing technique. In addition to his previous works, Chung (2015a), developed an automated tool to generate test cases from cause-effect graphs by using pairwise testing technique. Also, in another work, Chung (2015b), compared three different SAT problem solving approaches in order to generate test cases by pairwise testing technique.

Ayav and Belli (2015), proposed a formalization of generating test cases from cause-effect graphs by using MC/DC technique and Boolean differentiation. By this approach, the test cases are generated mathematically by calculating the derivatives of variables in the Boolean expressions of the graph. A mapping between Boolean expressions and calculus is made by this approach and it is easy to implement the MC/DC test generation by mathematical calculations.

Sun et al. (2015), gathered some of the test generation techniques on Boolean expressions (ONE, MIN, MUMCUT, and different forms of MUMCUT) and used the test cases generated by these techniques on randomly generated Boolean expressions. Mutation analysis is used in order to compare these methods' successes. The successes of these methods on general form Boolean expressions are also measured to see the effect of the constraint that Boolean expressions must be in DNF. As a result, it is stated that MUMCUT technique gives the best results and they made some extension on that technique.

Since CEGs are defined and used in software testing by Elmendorf (1973) and Myers (1979), few works are done in order to generate test cases from CEGs directly, as CEG_BOR. However, in order to generate test cases from Boolean expressions, there are many different techniques defined and studied, as MI, MC/DC, MUMCUT, etc. Also, comparisons for different methods are done in some of these works, but there is no work which considers the methods using different approachs. For example, the methods generating test cases from DNF Boolean expressions are compared as MI and MUMCUT together. However, no comparison of MC/DC and MI is done, which are using general form of Boolean expressions and DNF Boolean expressions, respectively.

CHAPTER 3

BACKGROUND

3.1. Cause-Effect Graph

Cause-effect graph is an undirected graph connecting the set of causes with the set of effects, which is used in software testing. The elements of the graph represent the Boolean expressions of the system with cause nodes as variables. The requirements are examined and the cause and effect nodes are determined so that: Cause nodes are the inputs to the system representing a state or an action and effect nodes represent a system state or a result effect, output from the system. All nodes in the graph can take binary values (true or false, 1 or 0 equivalently). It is important to write correct system requirements since the graph is drawn accordingly. In addition, it is important to draw the cause-effect graph right since all the test case generation process continues assuming that the graph represents the system correctly.

The formation of the nodes are from left to right as:

$$\textit{cause node} \rightarrow \textit{intermediate node} \rightarrow \textit{effect node}$$

The relationships between nodes can be the basic Boolean operations *AND, OR, NOT* which are shown in Figure 3.1.

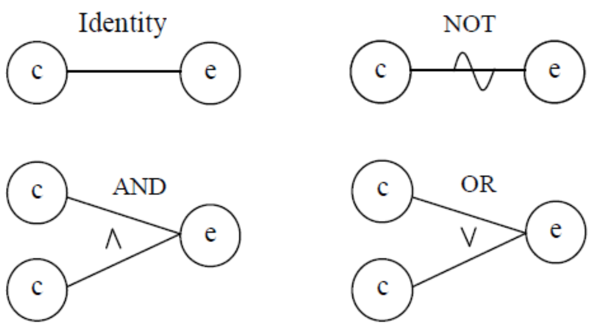


Figure 3.1. The basic operations which can be used on cause effect graphs.

In addition to the basic relationships, under the assumption of *a, b* being cause nodes and *e, f* being effect nodes, the constraints explained below and shown in Figure 3.2 can be defined:

- *Exclusive Or (E)*: At most one of the variables a and b can be true.
- *Inclusive Or (I)*: At least one of the variables a and b must be true.
- *Only-one (O)*: One and only one of the variables a and b must be true.
- *Required (R)*: In order to variable a to be true, variable b must be true.
- *Masking (M)*: When the effect e is true, effect f must be false.

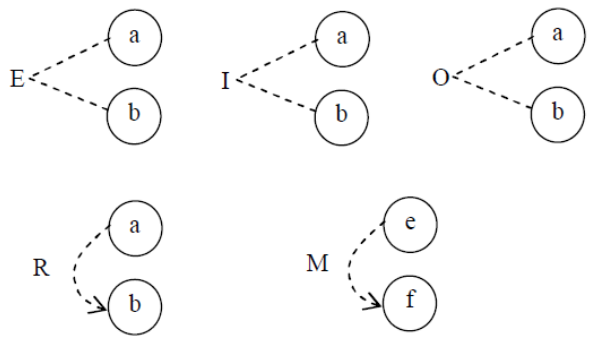


Figure 3.2. The constraints which can be defined on cause effect graphs.

Assume that we have a simple application which is also used in the book of Myers (1979), with the system specifications below:

The system has two inputs, first input must be one of the letters *'A'* or *'B'*. Second input must be a number. If both inputs are valid, a file update is made. If first input is invalid, an error message *'X12'*, if second input is invalid, an error message *'X13'* is given. The causes and effects interpreted from these specifications are given below in Table 3.1, and the cause-effect graph representing them is given in Figure 3.3.

Table 3.1. The causes and effects interpreted from the given specifications of the example.

Causes	Effects
1- First input is *'A'*	70- file update is made
2- First input is *'B'*	71- *'X12'* error message is given
3- Second input is a number	72- *'X13'* error message is given

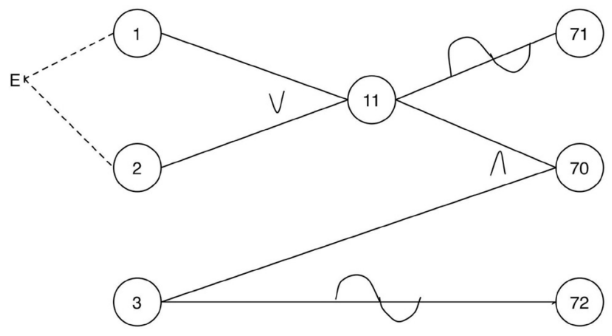


Figure 3.3. Cause effect graph example.

3.2. Test Generation Techniques

Definition Disjunctive Normal Form: In Boolean logic, a Boolean expression is in DNF, if it is a disjunction of conjunctive literals. A literal is a variable or its negation. In other words, it can be called as sum of products.

Example:

$$E = (B \wedge C) \vee (A' \wedge D) \vee (A' \wedge B' \wedge C \wedge D \wedge E')$$

Let E be a Boolean expression in DNF with n terms and m variables:

$$E = e_1 \vee e_2 \vee ... \vee e_n \tag{3.1}$$

Let all $e_i \ \ (1 \leq i \leq n)$, be a term with l_j number of variables. By using this definition; MI, MAX-A, CUTPNFP, MUMCUT, Unique MC/DC and Masking MC/DC testing techniques will be given in the following sections.

3.2.1. Myers' Technique

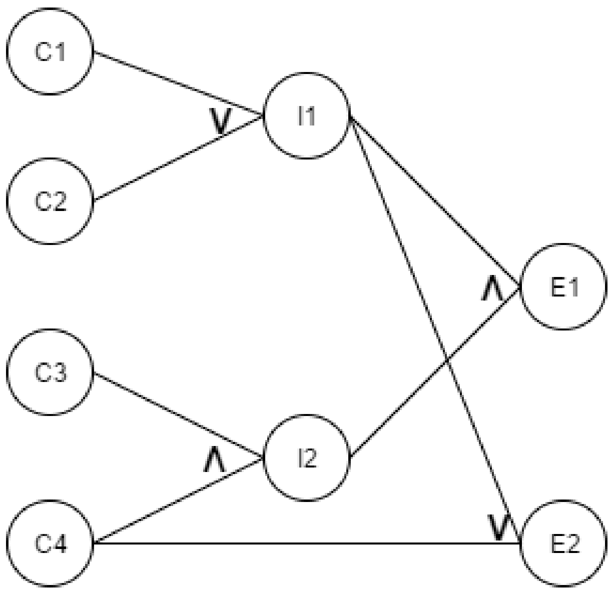


Figure 3.4. Example graph for Myers technique.

Myers (1979), defined a technique to generate test cases from a cause-effect graph. In this technique, a cause-effect graph or any form of (general form, DNF, CNF, etc.) Boolean expressions representing the graph is needed. The technique is given below:

1. An effect node is chosen and assumed to have true value. Nodes are traversed in the graph backwards towards the cause nodes.
2. All the combinations leading to the true value of chosen node are taken by considering the following:
   (a) If the node is an "OR" node and it must take true value, at most one of the nodes leading to this node must get true value. It is done in order to examine the influence of only one variables' effect at a time.
   (b) If the node is an "AND" node and it must take false value
      - only one case is chosen for each node getting false value
      - in cases where at least one node gets false value, only one combination is selected for all other nodes getting true value

Example: Generating test cases for the effect node E1 in graph given in Figure 3.4:

Corresponding Boolean expression: $E1 = I1 \wedge I2 = (C1 \vee C2) \wedge (C3 \wedge C4)$

1. $E1 = I1 \wedge I2$ gets value $1 \rightarrow$ all combinations make output true
   (a) $I1 = (C1 \vee C2)$ gets value $1 \rightarrow$ all combinations make output true, however, only one input is true at the same time $(0, 1, ,), (1, 0, ,)$.
   (b) $I2 = (C3 \wedge C4)$ gets value $1 \rightarrow$ all combinations make output true $(, , 1, 1)$.

$$E1 \text{ getting value } 1 \rightarrow I1 = 1, I2 = 1 \rightarrow (1, 0, 1, 1), (0, 1, 1, 1)$$

So, the test set generated by Myers:

$$\{(1, 0, 1, 1), (0, 1, 1, 1)\}$$

3.2.1.1. UTP

Unique True Point (UTP): UTP_i is the set of test inputs that makes term e_i true, and all other terms false in DNF Boolean expression (3.1). All unique true points for the Boolean expression: $UTP(E) = \bigcup_i UTP_i(E)$.

Example: For a simple example:

$$E = (C1 \wedge C2) \vee (C3 \wedge C4) \tag{3.2}$$

For $e_1 = C1 \wedge C2 \rightarrow UTP_1(E) = \{(1, 1, 0, 0), (1, 1, 0, 1), (1, 1, 1, 0)\}$

For $e_2 = C3 \wedge C4 \rightarrow UTP_2(E) = \{(0, 0, 1, 1), (0, 1, 1, 1), (1, 0, 1, 1)\}$

$UTP(E) = \{(1, 1, 0, 0), (1, 1, 0, 1), (1, 1, 1, 0), (0, 0, 1, 1), (0, 1, 1, 1), (1, 0, 1, 1)\}$

3.2.1.2. NFP

Near False Point (NFP): $NFP_{i,\bar{j}}$ is the set of test inputs that makes term $e_{i,\bar{j}}$ (which is formed by negating the jth variable of the term e_i) true, and all other terms false in DNF Boolean expression (3.1). All near false points for the term e_i of the Boolean expression: $NFP_i(E) = \bigcup_j NFP_{i,\bar{j}}(E)$. All near false points for the Boolean expression: $NFP(E) = \bigcup_i NFP_i(E)$.

Example: For the same Example (3.2):

For $e_{1,\bar{1}} = C1' \wedge C2 \rightarrow NFP_{1,\bar{1}}(E) = \{(0,1,0,0),(0,1,0,1),(0,1,1,0)\}$
For $e_{1,\bar{2}} = C1 \wedge C2' \rightarrow NFP_{1,\bar{2}}(E) = \{(1,0,0,0),(1,0,0,1),(1,0,1,0)\}$
For $e_{2,\bar{1}} = C3' \wedge C4 \rightarrow NFP_{2,\bar{1}}(E) = \{(0,0,0,1),(1,0,0,1),(0,1,0,1)\}$
For $e_{2,\bar{2}} = C3 \wedge C4' \rightarrow NFP_{2,\bar{2}}(E) = \{(0,0,1,0),(1,0,1,0),(0,1,1,0)\}$

$$\begin{aligned} NFP(E) = \{&(0,1,0,0),(0,1,0,1),(0,1,1,0),(1,0,0,0),\\ &(1,0,0,1),(1,0,1,0),(0,0,0,1),(0,0,1,0)\} \end{aligned}$$

3.2.2. Meaningful Impact (MI)

Meaningful Impact (MI) is defined by Weyuker et al. (1994), by studying it on the Boolean expressions of TCAS-II system specifications. Given DNF Boolean expressions, unique true point (UTP) and near false point (NFP) sets are found. By these sets, faults such as missing variable fault (MVF), variable negation fault (VNF), variable reference fault (VRF), operator reference fault (ORF) and expression negation fault (ENF) can be caught. However, random selection of the UTPs may cause missing some of these faults. A faulty term in a faulty Boolean expressions' UTP or NFP might include another terms' UTP or NFP in original Boolean expression. Furthermore, there might be same points in different UTP and NFP sets for different Boolean expressions of the system. Thus, it is not guaranteed to detect these faults in every case.

A formal algorithm of MI is given in Badhera et al. (2011):

1. For all $e_i,\ 1 \leq i \leq n$, Te_i sets are formed that makes e_i true.
2. For all $i \neq j, \quad TSe_i \cap TSe_j = \emptyset, \qquad TSe_i = Te_i - \bigcup\limits_{j=1, i\neq j}^{n} Te_j.$
3. S_E^t is formed by randomly choosing one element from each $TSe_i,\ 1 \leq i \leq n.$
4. For all e_i, $1 \leq i \leq n$ and $1 \leq j \leq l_j$, term e_i^j is formed by taking the term e_i and negating its' jth variable. Then, sets Fe_i^j are formed which make e_i^j true.
5. $FSe_i^j = Fe_i^j - \bigcup\limits_{k=1}^{n} Te_k.$
6. S_E^f set is formed to minimally cover each FSe_i^j set.
7. The needed set for E by MI is $S_E = S_E^t \cup S_E^f$.

First, for all term (e_i) of the Boolean expression (E), true test case sets (Te_i) are generated making that term true. From these sets, mutually exclusive subsets (TSe_i) are selected. Final true set (S_E^t) is formed by selecting one element from each of these subsets randomly.

Then, for all term (e_i), negation terms (e_i^j) are created for each variable of that term. False test case sets (Fe_i^j) are generated making these negation terms true. If there are mutual elements in these sets created with the true test sets (Te_i) created in the first step, these elements are removed to form subsets (FSe_i^j) of false test sets. This elimination is done in order to prevent the conflict of the aims of true and false sets. Then, a final false set (S_E^f) is created including all the eliminated subsets.

MI is the union of the final true set (S_E^t) and final false set (S_E^f).

Example: Generating test cases by MI for the effect $E3$ in system below:

$$\begin{aligned} &E1 = I1, \\ &E2 = I1' \wedge C3 \\ &E3 = I1' \vee C2 \vee C3 \qquad \text{,where } I1 = (C1 \vee C2) \end{aligned}$$

Effects converted to DNF:

$$\begin{aligned} &E1 = (C1 \vee C2), \\ &E2 = C1' \wedge C2' \wedge C3 \\ &E3 = (C1' \wedge C2') \vee C2 \vee C3 \end{aligned}$$

True test case sets Te_i for each term of $E3 = (C1' \wedge C2') \vee C2 \vee C3$:

$$Te_1 = \{\textbf{(0,0,1)}, (0,0,0)\} \rightarrow TSe_1 = \{(0,0,0)\}$$
$$Te_2 = \{(0,1,0), (1,1,0), \textbf{(0,1,1)}\} \rightarrow TSe_2 = \{(0,1,0), (1,1,0)\}$$
$$Te_3 = \{\textbf{(0,0,1)}, \textbf{(0,1,1)}, (1,0,1)\} \rightarrow TSe_3 = \{(1,0,1)\}$$

True test set for the effect $E3$: S^t_{E3} = {(0,0,0),(0,1,0),(1,0,1)}

False test case sets Fe^j_i for each term i and each variable j of each term of $E3 = (C1' \wedge C2') \vee C2 \vee C3$:

$$Fe^1_1 = \{\textbf{(1,0,1)}, (1,0,0)\} \rightarrow FSe^1_1 = \{(1,0,0)\}$$
$$Fe^2_1 = \{\textbf{(0,1,1)}, \textbf{(0,1,0)}\} \rightarrow FSe^2_1 = \emptyset$$
$$Fe^1_2 = \{\textbf{(1,0,1)}, (1,0,0), \textbf{(0,0,1)}\} \rightarrow FSe^1_2 = \{(1,0,0)\}$$
$$Fe^1_3 = \{\textbf{(1,1,0)}, (1,0,0), \textbf{(0,1,0)}\} \rightarrow FSe^1_3 = \{(1,0,0)\}$$

False test set for the effect $E3$: S^f_{E3} = {(1,0,0)}

$S_E 3 = S^t_{E3} \cup S^f_{E3} = \{(0,0,0), (0,1,0), (1,0,1), (1,0,0)\}$

3.2.2.1. MAX-A

For all terms of the Boolean expression, all points of TSe_i and FSe^j_i are selected. MAX-A set for the Boolean expression:

$$MAX - A(E) = TS(E) \cup FS(E)$$

Example: For the same simple Example (3.2):

$$\begin{aligned} MAX - A(E) = \{&(1,1,0,0), (1,1,0,1), (1,1,1,0), (0,0,1,1),\\ &(0,1,1,1), (1,0,1,1), (0,1,0,0), (0,1,0,1),\\ &(0,1,1,0), (1,0,0,0), (1,0,0,1), (1,0,1,0), (0,0,0,1), (0,0,1,0)\} \end{aligned}$$

3.2.3. MUMCUT

MUMCUT is the union of three different techniques: MUTP, MNFP and CUTP-NFP. The union of the test sets generated by MUTP, MNFP and CUTPNFP are taken in MUMCUT technique.

3.2.3.1. Multiple Unique True Point (MUTP)

Multiple Unique True Point (MUTP): For all terms e_i of the Boolean expression E, unique true points are selected from the set $UTP_i(E)$, which will cover all possible truth values of the variables not existent in that term.

Example: For the same simple Example (3.2):

For the selection of unique true points from $UTP_1(E)$; we cover all the possible truth values of every missing literal $e_1 = C1 \wedge C2$ (that is, $C3$ and $C4$). Similarly, for the selection of unique true points from $UTP_2(E)$; we cover all the possible truth values of every missing literal $e_2 = C3 \wedge C4$ (that is, $C1$ and $C2$).

Recall the sets from 3.2.1.1:

$$UTP_1(E) = \{(1,1,0,0), (1,1,0,1), (1,1,1,0)\}$$
$$UTP_2(E) = \{(0,0,1,1), (0,1,1,1), (1,0,1,1)\}$$

Therefore, we need $(1,1,0,1)$ and $(1,1,1,0)$ from $UTP_1(E)$ in order to cover all possible values of $C3$ and $C4$.

In addition, we need $(0,1,1,1)$ and $(1,0,1,1)$ from $UTP_2(E)$ in order to cover all possible values of $C1$ and $C2$.

Hence, the test set

.
$$MUTP(E) = \{(1,1,0,1), (1,1,1,0), (0,1,1,1), (1,0,1,1)\}$$

3.2.3.2. Multiple Near False Point (MNFP)

Multiple Near False Point (MNFP): For all possible i and all possible j for that term e_i of the Boolean expression E, near false points are selected from the set $NFP_{i,\bar{j}}(E)$, which will cover all possible truth values of the variables not existent in ith term.

Example: For the same simple Example (3.2):

For the selection of near false points from $NFP_{1,\bar{1}}(E)$; we cover all the possible truth values of every missing literal of $e_{1,\bar{1}} = C1' \wedge C2$ (that is, $C3$ and $C4$).

Also, for $NFP_{1,\bar{2}}(E)$; we cover all the possible truth values of every missing literal of $e_{1,\bar{2}} = C1 \wedge C2'$ (that is, $C3$ and $C4$).

Similarly, for the selection of near false points from $NFP_{2,\bar{1}}(E)$; we cover all the possible truth values of every missing literal of $e_{2,\bar{1}} = C3' \wedge C4$ (that is, $C1$ and $C2$).

Also, for $NFP_{2,\bar{1}}(E)$; we cover all the possible truth values of every missing literal of $e_{2,\bar{2}} = C3 \wedge C4'$ (that is, $C1$ and $C2$).

Recall the $NFP_{i,\bar{j}}(E)$ sets in 3.2.1.2:

$$NFP_{1,\bar{1}}(E) = \{(0,1,0,0),(0,1,0,1),(0,1,1,0)\}$$
$$NFP_{1,\bar{2}}(E) = \{(1,0,0,0),(1,0,0,1),(1,0,1,0)\}$$
$$NFP_{2,\bar{1}}(E) = \{(0,0,0,1),(1,0,0,1),(0,1,0,1)\}$$
$$NFP_{2,\bar{2}}(E) = \{(0,0,1,0),(1,0,1,0),(0,1,1,0)\}$$

So, we need $(0,1,0,1)$ and $(0,1,1,0)$ from $NFP_{1,\bar{1}}(E)$, $(1,0,0,1)$ and $(1,0,1,0)$ from $NFP_{1,\bar{2}}(E)$, $(1,0,0,1)$ and $(0,1,0,1)$ from $NFP_{1,\bar{1}}(E)$ and finally $(1,0,1,0)$ and $(0,1,1,0)$ from $NFP_{1,\bar{1}}(E)$,

Hence, the test set

$$MNFP(E) = \{(0,1,0,1),(0,1,1,0),(1,0,0,1),(1,0,1,0)\}.$$

3.2.3.3. Corresponding Unique True Point and Near False Point Pair (CUTPNFP)

Corresponding Unique True Point and Near False Point Pair (CUTPNFP): $\overrightarrow{u}$ and $\overrightarrow{v}$ test inputs pair is selected respectively from $UTP_i(E)$ and $NFP_{i,\bar{j}}(E)$ such that; they must only differ in the truth value of the jth term. This selection is done for each j value of each term and so, the $CUTPNFP(E)$ set is formed. The set is expected to detect faults as ORF, VNF and ENF.

Example: For the same simple Example (3.2):

For the $UTP_1(E)$ and $NFP_{1,\bar{1}}(E)$ pair,

$$\overrightarrow{u_1} = (1,1,0,0) \in UTP_1(E) \text{ and } \overrightarrow{v_1} = (0,1,0,0) \in NFP_{1,\bar{1}}(E)$$

can be selected. The only difference between $\overrightarrow{u}$ and $\overrightarrow{v}$ is first term's ($e_1 = C1 \wedge C2$) first variable's ($C1$) truth value.

Similarly,

$$\overrightarrow{u_2} = (1,1,0,0) \in UTP_1(E) \text{ and } \overrightarrow{v_2} = (0,1,0,0) \in NFP_{1,\bar{2}}(E)$$

can be selected for the $UTP_1(E)$ and $NFP_{1,\bar{2}}(E)$ pair. The only difference between $\overrightarrow{u}$ and $\overrightarrow{v}$ is first term's $e_1 = C1 \wedge C2$ second variable $C2$'s truth value.

As in this example, different pairs may contain same elements ($\overrightarrow{u_1}$ and $\overrightarrow{u_2}$ are the same). In fact, this is a desirable situation since it reduces the total number of test cases.

Continuing the same process for the $UTP_2(E)$ and $NFP_{2,\bar{1}}(E)$ pair we can chose

$$\overrightarrow{u_3} = (0,0,1,1) \in UTP_2(E) \text{ and } \overrightarrow{v_3} = (0,0,1,0) \in NFP_{2,\bar{1}}(E).$$

Finally, for the $UTP_2(E)$ and $NFP_{2,\bar{2}}(E)$ pair we can choose

$$\overrightarrow{u_4} = (0,0,1,1) \in UTP_2(E) \text{ and } \overrightarrow{v_4} = (0,0,0,1) \in NFP_{2,\bar{2}}(E).$$

Hence, the test set

$$CUTPNFP(E) = \{(1,1,0,0), (0,1,0,0), (1,0,0,0), (0,0,1,1), (0,0,1,0), (0,0,0,1)\}.$$

3.2.4. Modified Condition/Decision Coverage (MC/DC)

Modified Condition/Decision Coverage (MC/DC) aims simply selecting test cases to see each conditions' effect on the outcome of the Boolean expression independently. There are two different approaches for the technique.

3.2.4.1. Unique Modified Condition/Decision Coverage (Unique MC/DC)

Unique Modified Condition/Decision Coverage: In unique-cause approach, an pair of test cases is selected (named independence-pair) for each condition in the Boolean expression. The test cases must be the same, except the truth values of the condition under consideration and the outcomes of the test cases. Preserving the value of every other condition fixed ensures that the condition with changed value is the only condition affecting the value of outcome.

A truth table is often used to show the unique-cause approach. A truth table is created for the Boolean expression with new columns added on the right which indicate the possible independence pairs.

Example:

An example from CAST's positions paper (Cast) (2001) for the Boolean expression $Z = (A \vee B) \wedge (C \vee D)$:

Table 3.2. Unique cause approach to independence pairs of $Z = (A \vee B) \wedge (C \vee D)$

	Variables					Result		Independence-Pairs			
Test #	**A**	**B**	**C**	**D**		**Z**		**A**	**B**	**C**	**D**
1	F	F	F	F		F					
2	F	F	F	T		F		10	6		
3	F	F	T	F		F		11	7		
4	F	F	T	T		F		12	8		
5	F	T	F	F		F				7	6
6	F	T	F	T		T			2		5
7	F	T	T	F		T			3	5	
8	F	T	T	T		T			4		
9	T	F	F	F		F				11	10
10	T	F	F	T		T		2			9
11	T	F	T	F		T		3		9	
12	T	F	T	T		T		4			
13	T	T	F	F		F				15	14
14	T	T	F	T		T					13
15	T	T	T	F		T				13	
16	T	T	T	T		T					

In the Table 3.2, we see the truth table of Z on the left side of the table, and the possible pairs for each variable on the right side of the table. We can select any of the possible pairs for each variable.

Test pairs (2,10), (3,11) and (4,12) can be selected to show the independent effect of A, (2,6), (3,7), (4,8) can be selected for B, (5,7), (9,11), (13,15) can be selected for C, (5,6), (9,10), (13,14) can be selected for D.

3.2.4.2. Masking Modified Condition/Decision Coverage (Masking MC/DC)

The unique-cause approach guarantees the minimum tests for each condition when there is no strongly coupled conditions or repeated conditions exists. If there are repeated conditions in the expression (e.g. $Z = (A \vee B) \wedge (A \vee D)$), unique-cause approach will fail since it will force one of the same conditions to stay fixed and the other to change.

Masking means, in some specific cases, a value of a condition will determine the output of that part of the expression, independent from the other conditions' truth values. For instance, a false valued condition makes an AND operation's outcome as false, a true valued condition makes an OR operation's outcome as true.

Masking Modified Condition/Decision Coverage: The masking approach allows more than one condition's truth value to change in an independence pair, if that change does not change the truth value of the outcome under consideration. In other words, that changes must be masked and must not affect the outcome.

A truth table is often used to show the masking-cause approach, in the same manner as the unique-cause approach.

An example from CAST's positions paper (Cast) (2001) for the Boolean expression $Z = (A \vee B) \wedge (C \vee D)$:

*X represents $(A \vee B)$, Y represents $(C \vee D)$.

In the Table 3.3, we see the truth table of Z on the left side of the table, and the possible pairs for each variable on the right side of the table. We can select any of the possible pairs for each variable. For example;

Test pairs (2,10), (2,11), (2,12), (3,10), (3,11), (3,11), (4,10), (4,11), (4,12) can be selected to show the independent effect of A, (2,6), (2,7), (2,8), (3,6), (3,7), (3,8), (4,6), (4,7), (4,8) can be selected for B, (5,7), (5,11), (5,15), (7,9), (7,13), (9,11), (9,15), (11,13), (13,15) can be selected for C, (5,6), (5,10), (5,14), (6,9), (6,13),(9,10), (9,14), (10,13), (13,14) can be selected for D.

Table 3.3. Masking approach to independence pairs of $Z = (A \vee B) \wedge (C \vee D)$

	Term 1			Term 2			Result	Independence-Pairs			
Test #	**A**	**B**	**X***	**C**	**D**	**Y***	**Z**	**A**	**B**	**C**	**D**
1	F	F	F	F	F	F	F				
2	F	F	F	F	T	T	F	10, 11, 12	6, 7, 8		
3	F	F	F	T	F	T	F	10, 11, 12	6, 7, 8		
4	F	F	F	T	T	T	F	10, 11, 12	6, 7, 8		
5	F	T	T	F	F	F	F			7, 11, 15	6, 10, 14
6	F	T	T	F	T	T	T		2, 3, 4		5, 9, 13
7	F	T	T	T	F	T	T		2, 3, 4	5, 9, 13	
8	F	T	T	T	T	T	T		2, 3, 4		
9	T	F	T	F	F	F	F			7, 11, 15	6, 10, 14
10	T	F	T	F	T	T	T	2, 3, 4			5, 9, 13
11	T	F	T	T	F	T	T	2, 3, 4		5, 9, 13	
12	T	F	T	T	T	T	T	2, 3, 4			
13	T	T	T	F	F	F	F			7, 11, 15	6, 10, 14
14	T	T	T	F	T	T	T				5, 9, 13
15	T	T	T	T	F	T	T			5, 9, 13	
16	T	T	T	T	T	T	T				

3.3. Mutation Analysis

Mutation analysis is widely used in order to compare the fault detection success of test sets. Mutants are formed by making specific changes on the original Boolean expressions. These changes are made by certain rules that are inferred from the common mistakes made by programmers. Previous studies on mutant generation has defined that a generated mutant may refer one or more fault classes, based on the type of changes that have been made. A test set is said to kill a mutant, if it detects a mutant when it is run on that mutant. The success of a test set is measured by the number of the mutants it can kill. The aim of a good test case generation technique is killing as many mutants with fewer test case.

On an example Boolean expression as $S = (C0 \vee (C1 \wedge C2)) \wedge C3$, main fault types can be defined as follows Mathur (2008):

ORF Operator Reference Fault. "AND" operator is replaced by "OR", or "OR" operator is replaced by "AND". Ex.: $(C0 \vee (C1 \wedge C2)) \vee C_3$.

ENF Expression Negation Fault. A sub expression of the expression is negated. Ex.: $(C0 \vee (C1 \wedge C2)') \wedge C3$.

VNF Variable Negation Fault. A variable in the expression is negated.
Ex.: $(C0 \vee (C1' \wedge C2)) \wedge C3$.

MVF Missing Variable Fault. A variable is forgotten. Ex.: $(C0 \vee (C1 \wedge C2))$.

VRF Variable Reference Fault. A variable is changed by another variable.
Ex.: $(C1 \vee (C1 \wedge C2)) \wedge C3$.

CCF Clause Conjunction Fault. $a \wedge b$ is written instead of variable a.
Ex.: $(C0 \vee (C1 \wedge C3 \wedge C2)) \wedge C3$.

CDF Clause Disjunction Fault. $a \vee b$ is written instead of variable a.
Ex.: $(C0 \vee C3 \vee (C1 \wedge C2)) \wedge C3$.

SA0 Stuck at-0. A variable is always set to 0. Ex.: $(0 \vee (C1 \wedge C2)) \wedge C3$.

SA1 Stuck at-1. A variable is always set to 1. Ex.: $(C0 \vee (1 \wedge C2)) \wedge C3$.

After all mutants are created for all fault classes, running test sets on all these mutants can be infeasible in the means of time and sources, for large scale programs. According to Kuhn (1999)'s study, if a fault class can cover another fault class' faulty expressions, that fault class is considered as stronger than the other one. In Kuhn's study, it is shown that ENF is stronger than VNF, and VNF is stronger than VRF in a hierarchy.

Tsuchiya and Kikuno (2002) added missing expression fault class into the hierarchical structure which Kuhn (1999) introduced. Lau and Yu (2005) extended the hierarchy by analyzing and adding term and variable faults.

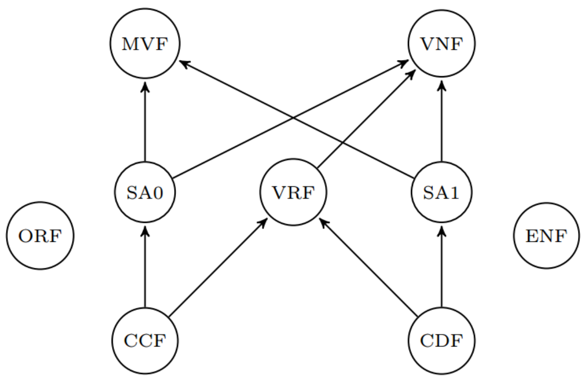


Figure 3.5. Fault class hiearchy.

In all studies mentioned above, analyzed Boolean expression should be given in DNF. Kapoor and Bowen (2007) reformed the hierarchical structure and added new fault classes, so that it can be viable for general Boolean expressions. Chen et al. (2011) extended Kapoor and Bowen (2007)'s study and presented the hierarchical structure given in the Figure 3.5. Arrows go from weaker to stronger. According to this structure, considering only four of nine fault classes is enough: ORF, CCF, CDF and ENF.

CHAPTER 4

PROPOSED MODEL AND IMPLEMENTED TOOL

4.1. CEG Model

In order to define and model a cause-effect graph, an XML based file format created to represent graphs named GraphML (2016) is used. GraphML's base has main components of a graph as node, edge, etc. and main properties on them as label, id, etc. This base can be extended by adding new node and edge properties if needed. To be able to define a proper cause-effect graph, it was sufficient to add the property of negation to the edges. Furthermore, the following properties are added to the nodes:

- grade - is used to draw the graph properly
- relation - is used to define the Boolean expression forming the node
- node type - cause, effect, intermediate, constraint nodes
- constraint result - is used to store the intermediate or effect node that a constraint is connected

Detailed explanation of how to create the corresponding ".graphml" file from a cause-effect graph is given in Appendix A.

A graph visualization desktop application named Gephi (2016) and its open source library are used in order to draw cause-effect graphs, import a ".graphml" file to see its corresponding graph and export drawn graphs in ".graphml" file format. By using the Java based Gephi library, the graph information is taken from an imported ".graphml" file. This information is transformed into the desired cause-effect model by the implemented tool, which is also developed in Java language. The class diagram of cause-effect graph model with Gephi toolkit can be seen in Figure 4.1

Boolean expressions are formed in general form for each effect node by using the relation information between nodes in the graph. In order to use these Boolean expressions for the techniques MI, MAX-A and CUTPNFP, they are converted into DNF.

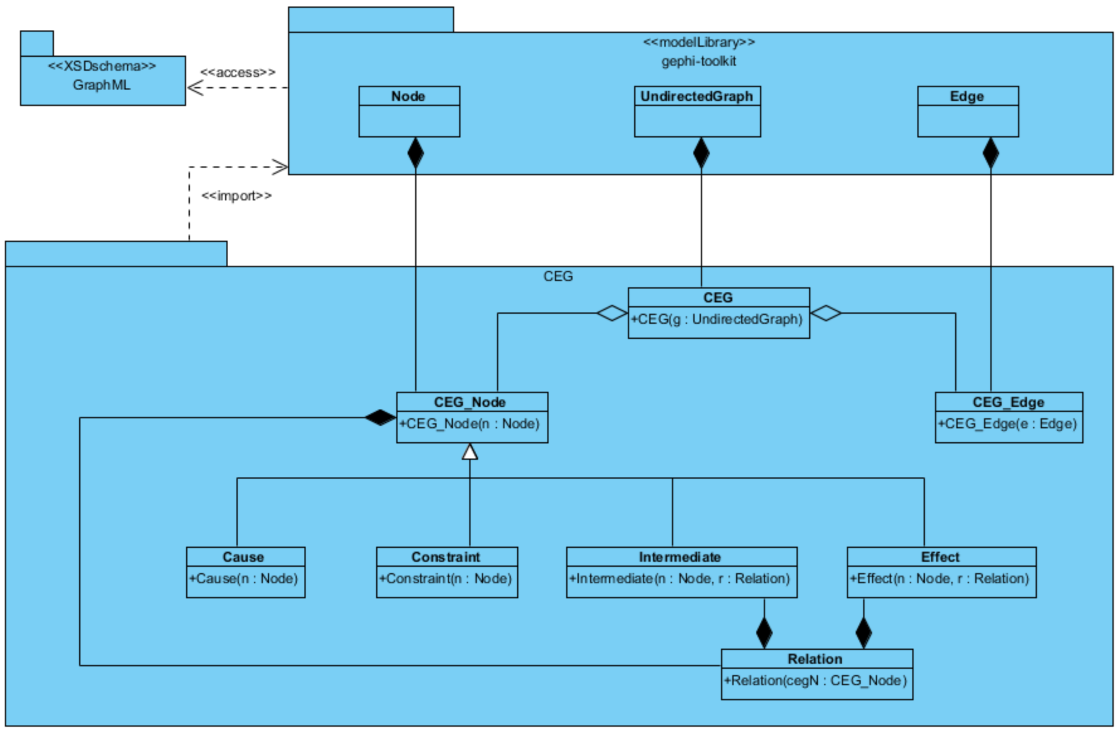


Figure 4.1. Class diagram of cause-effect graph model with Gephi toolkit.

This DNF is in canonical form since each term contains all variables. In order to apply MUTP, MNFP and MUMCUT methods the DNF Boolean expression cannot be canonical, since there must be some literals missing in terms, in order to apply these algorithms. For these methods, another open source tool named jbool_expressions developed by Podgursky (2016) is used. In order to create mutants, different mutation models are created for different fault classes. By using the original Boolean expression formed for each effect node, and the mutation model, mutants are created. The overall flow can be seen in Figure 4.2.

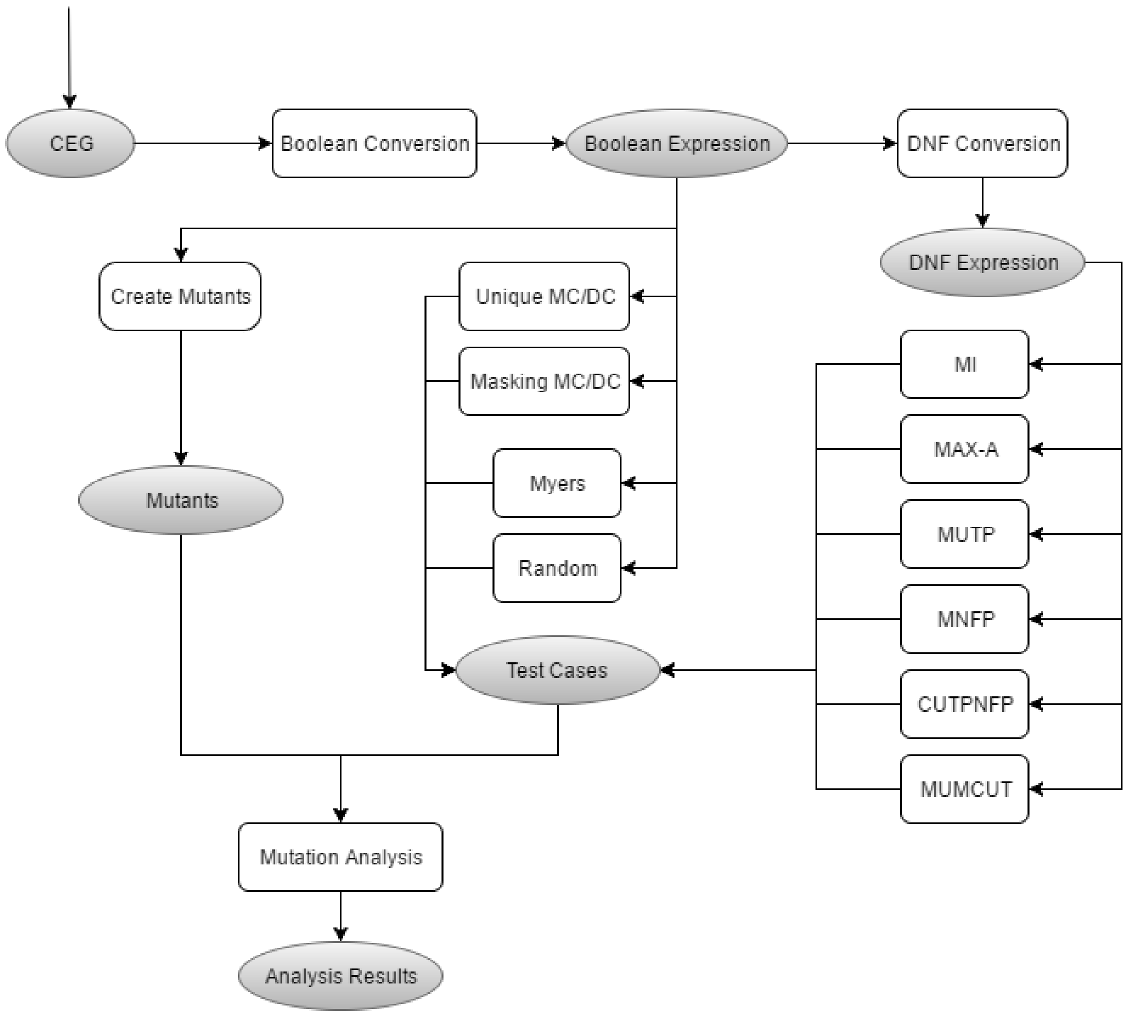


Figure 4.2. Architecture of Generating Test Cases From Cause Effect Graphs.

4.2. DNF Conversion

Recall that DNF is the disjunction of conjunctive literals from Section 3.2. These conjunctive literals form different terms of the expression. A term is called a minterm if it contains all variables once. Minterms also can be represented as truth table lines which has true result value, and vice versa. By using this property, we can convert a given Boolean expression into canonical DNF, by simply creating its corresponding truth table and then selecting true resulted lines.

All combinations for all variables are considered for each effect node. The lines that resulting in true value are chosen. These lines are examined if they have any constraint in between (exclusive or, inclusive or, required, etc.). Considering existent con-

straints, if the line conflicts with one of the constraints, that line is not chosen. Variables in these lines are taken as a term such as: a variable which has true value is directly selected, a variable which has false value is taken by its negation, then these variables are connected with "AND" operator. All chosen lines represent different terms of the new DNF Boolean expression. These terms are connected with "OR" operators, so that the DNF expression is formed.
Ex: Assume we have a Boolean expression as:
$E = (A \wedge B) \vee (((A \wedge C) \vee B) \wedge (B \wedge C))$

Table 4.1. Truth table for the expression $E = (A \wedge B) \vee (((A \wedge C) \vee B) \wedge (B \wedge C))$

A	**B**	**C**	**E**
T	T	T	T
T	T	F	T
T	F	T	F
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

Selecting the lines 1,2 and 5 in Table 4.1, we get the corresponding canonical DNF as:

$$E = (A \wedge B \wedge C) \vee (A \wedge B \wedge C') \vee (A' \wedge B \wedge C)$$

4.3. Mutation Analysis Implementation

Recall the fault classes in section 3.3: Operator Reference Fault (ORF), Expression Negation Fault (ENF), Variable Negation Fault (VNF), Missing Variable Fault (MVF), Variable Reference Fault (VRF), Clause Conjunction Fault (CCF), Clause Disjunction Fault (CDF) Stuck at-0 (SA0), Stuck at-1 (SA1). Although, only four of them are enough to cover the others, all nine fault classes are modeled in order to create mutants for each fault class from the original Boolean expression. Here, only one change is done on the original expression for each mutant created, mixed mutants are not used to examine individual faults. The class diagram of the mutant creation can be seen in Figure 4.3.

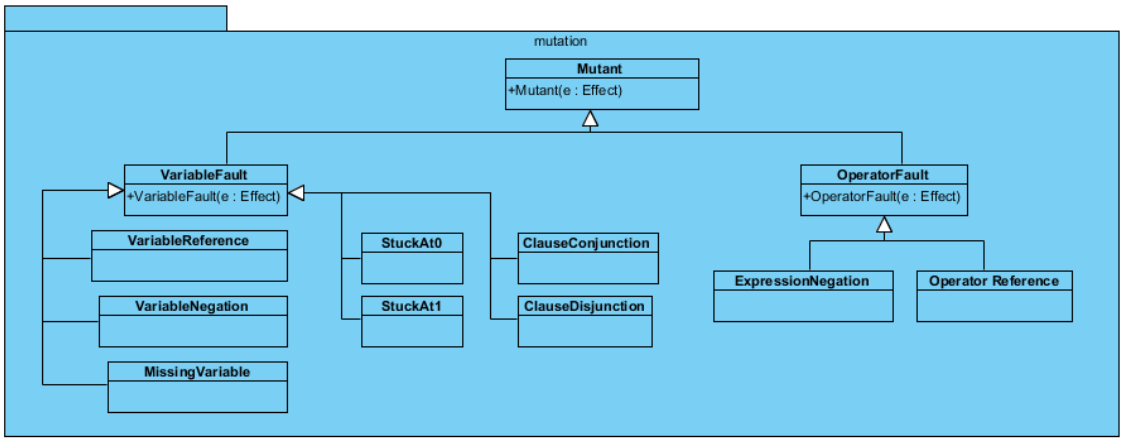
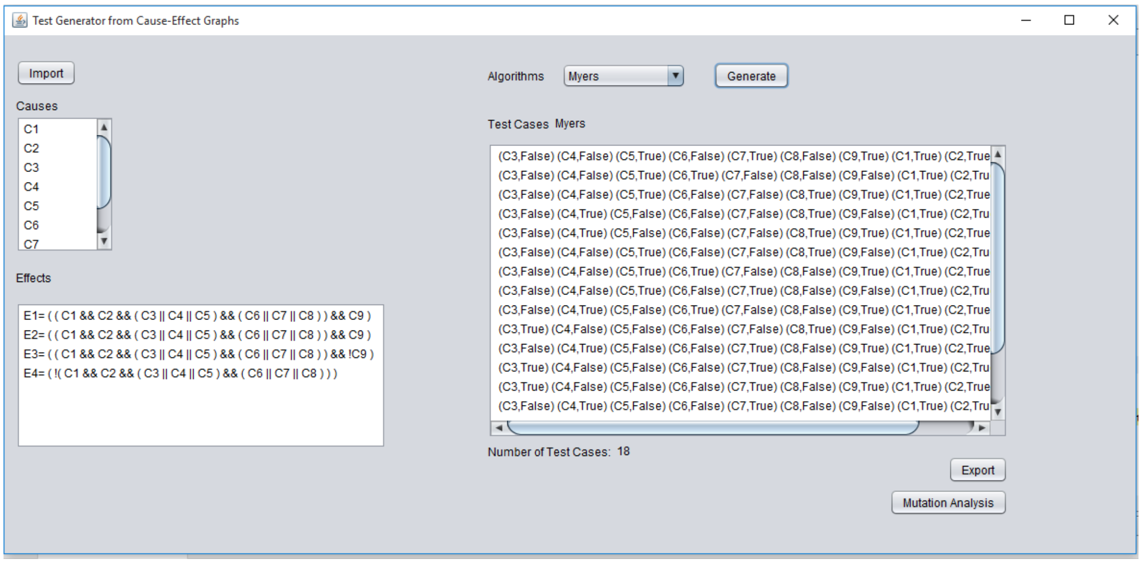


Figure 4.3. Class diagram of mutant creation.

4.4. Implemented Tool

Implemented tool allows user to import a cause-effect graph as a .graphml file. One of the implemented methods can be selected (Myers, MI, MC/DC, etc.). Generated test cases can be seen and can be exported. These can be done in the main page of the tool, can be seen in Figure 4.4.



Causes

```
C1
C2
C3
C4
C5
C6
C7
```

Effects

```
E1= ( ( C1 && C2 && ( C3 || C4 || C5 ) && ( C6 || C7 || C8 ) ) && C9 )
E2= ( ( C1 && C2 && ( C3 || C4 || C5 ) && ( C6 || C7 || C8 ) ) && C9 )
E3= ( ( C1 && C2 && ( C3 || C4 || C5 ) && ( C6 || C7 || C8 ) ) && !C9 )
E4= ( !( C1 && C2 && ( C3 || C4 || C5 ) && ( C6 || C7 || C8 ) ) )
```

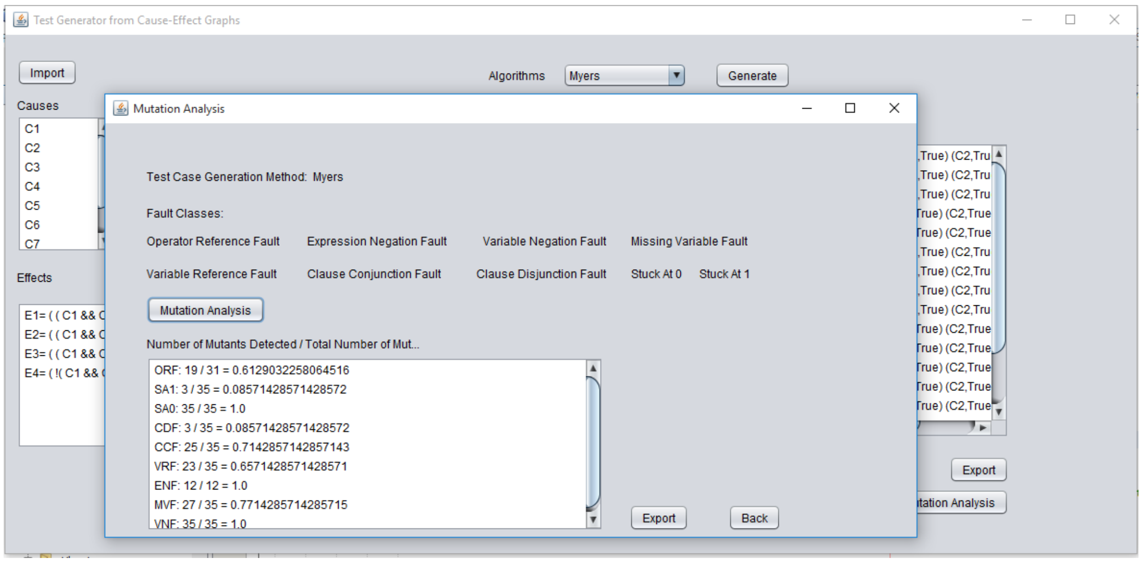
Test Cases Myers

```
(C3,False) (C4,False) (C5,True) (C6,False) (C7,True) (C8,False) (C9,True) (C1,True) (C2,True
(C3,False) (C4,False) (C5,True) (C6,True) (C7,False) (C8,False) (C9,False) (C1,True) (C2,Tru
(C3,False) (C4,False) (C5,True) (C6,False) (C7,False) (C8,True) (C9,True) (C1,True) (C2,True
(C3,False) (C4,True) (C5,False) (C6,False) (C7,False) (C8,True) (C9,False) (C1,True) (C2,Tru
(C3,False) (C4,True) (C5,False) (C6,False) (C7,False) (C8,True) (C9,True) (C1,True) (C2,True
(C3,False) (C4,False) (C5,True) (C6,False) (C7,False) (C8,True) (C9,False) (C1,True) (C2,Tru
(C3,False) (C4,False) (C5,True) (C6,True) (C7,False) (C8,False) (C9,True) (C1,True) (C2,True
(C3,False) (C4,False) (C5,True) (C6,False) (C7,True) (C8,False) (C9,False) (C1,True) (C2,Tru
(C3,False) (C4,True) (C5,False) (C6,True) (C7,False) (C8,False) (C9,True) (C1,True) (C2,True
(C3,True) (C4,False) (C5,False) (C6,False) (C7,False) (C8,True) (C9,False) (C1,True) (C2,Tru
(C3,False) (C4,True) (C5,False) (C6,False) (C7,True) (C8,False) (C9,True) (C1,True) (C2,True
(C3,True) (C4,False) (C5,False) (C6,False) (C7,True) (C8,False) (C9,False) (C1,True) (C2,Tru
(C3,True) (C4,False) (C5,False) (C6,False) (C7,True) (C8,False) (C9,True) (C1,True) (C2,True
(C3,False) (C4,True) (C5,False) (C6,False) (C7,True) (C8,False) (C9,False) (C1,True) (C2,Tru
```

Number of Test Cases: 18

Figure 4.4. Test Case Generator tool main page.

Mutation analysis can be done for all 9 fault types and also the results of the mutation analysis can be exported. Mutation analysis page of the tool can be seen in Figure 4.5.



Test Case Generation Method: Myers

Fault Classes:

Operator Reference Fault   Expression Negation Fault   Variable Negation Fault   Missing Variable Fault

Variable Reference Fault   Clause Conjunction Fault   Clause Disjunction Fault   Stuck At 0   Stuck At 1

Number of Mutants Detected / Total Number of Mut...

```
ORF: 19 / 31 = 0.6129032258064516
SA1: 3 / 35 = 0.08571428571428572
SA0: 35 / 35 = 1.0
CDF: 3 / 35 = 0.08571428571428572
CCF: 25 / 35 = 0.7142857142857143
VRF: 23 / 35 = 0.6571428571428571
ENF: 12 / 12 = 1.0
MVF: 27 / 35 = 0.7714285714285715
VNF: 35 / 35 = 1.0
```

Figure 4.5. Test Case Generator Mutation Analysis page.

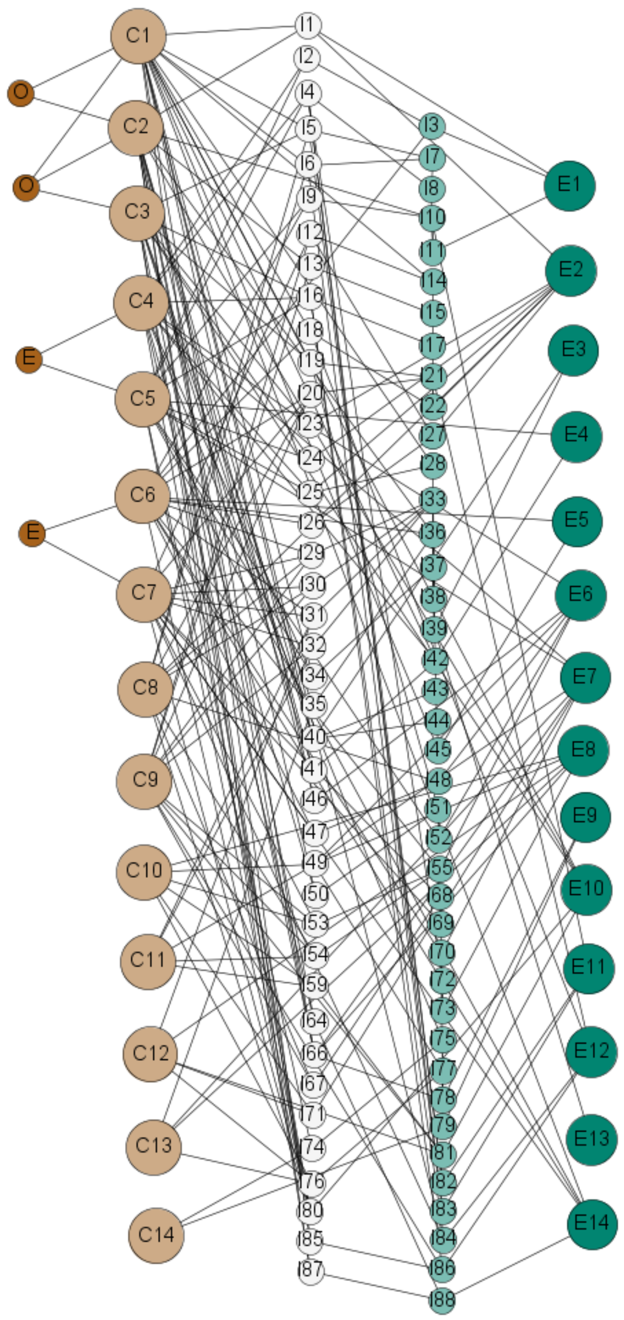


Figure 4.6. The drawn Cause-Effect Graph of 14 requirements of TCAS-II.

CHAPTER 5

EVALUATION

TCAS-II is an aviation collision avoidance system compatible with many aircraft types. It has been frequently used in literature on testing Lau and Yu (2005); Chen et al. (2009); Badhera et al. (2011); Weyuker et al. (1994); Gargantini and Fraser (2011). 14 requirements of TCAS-II are modeled as in Figure 4.6 and the corresponding graphml file of the graph is provided.

The cause-effect graph taken as graphml file is converted to Boolean expressions. From these expressions test cases are generated by using Myers' original method, Unique MC/DC and Masking MC/DC. Then these Boolean expressions are converted into canonical DNF, by using the method explained in Section 4.2. From these canonical DNF expressions, test cases are generated by using methods MI, MAX-A, MUTP, MNFP, CUTP-NFP and MUMCUT.

In order to use the given mutation analysis model in Section 4.3, for all nine fault classes mentioned, all possible mutants are created by making only one change on original expression. Generated test sets by different methods are run on created mutants and the number of mutants each test set can detect are listed. The Boolean expression evaluations are done by an open source library JBooleanExpression (2016).

Results of the experiments are given in tables below as: The number of test cases of methods are given in Table 5.1 and the number of total created mutants are given in Table 5.2. In Table 5.3, the total number of the mutants detected by the methods (success of methods) are given. Moreover, in Table 5.4, the detection success percentages of methods over total number of mutants are given. Lastly, in Table 5.5, the number of detected mutants per test case is given for each method. Furthermore, corresponding diagrams of the tables are given in Figures 5.1, 5.2, 5.3, 5.4, 5.5.

Comparing the number of test cases generated by different methods that uses Boolean expression in DNF: For MAX-A, since all true and false points are selected for each variable in each term of canonical DNF, the number of test cases is the highest. As mentioned before, MI and MAX-A are from the same family of test generation methods. However, the number of test cases generated by MI are less, because in MI

Table 5.1. The number of test cases for TCAS-II

	Number of Test Cases
MI	2125
MAX-A	4370
MUTP	1082
MNFP	1251
CUTPNFP	274
MUMCUT	2044
Unique MC/DC	87
Masking MC/DC	2304
Myers	708
Random1	87
Random2	300
Random3	750
Random4	1538
Random5	2200
Random6	3000
Random7	4370

Table 5.2. The number of created mutants for TCAS-II

	VNF+ENF	**ORF**	**SA0+SA1**	**VRF**	**CCF**	**CDF**	**MVF**	**All**
# Mutants	391	238	504	238	252	252	140	2029

method, only one test case is selected from true points, while all true points are selected in MAX-A. For MUTP and MNFP, the number of test cases is lesser, since the used DNF expressions were not canonical as MAX-A and MI. MUTP and MNFP focuses on true and false points, respectively. On the other hand, CUTPNFP has the least number of test cases, since it focuses on only specific types of faults. MUMCUT is the union of the test cases generated by MUTP, MNFP and CUTPNFP. However, the number of test cases of MUMCUT is lesser than the exact summation of the number of test cases of MUTP, MNFP and CUTPNFP. This difference occurs due to the existence of the common test cases generated for different effects in the graph. For instance, a test case that is generated for a specific effect by MUTP can be mutual with a test case generated for another effect by MNFP.

Comparing the number of test cases generated by the methods that use Boolean expression in general form: Unique MC/DC has generated the least number of test cases. This is because, Unique MC/DC directly selects independence pairs for each variable of each effect's expression. These pairs are selected in order to examine each variable's independent effect on the outcome of the corresponding effect. Masking MC/DC generated higher number of test cases. Since it selects different combinations of intermediate nodes, where these combinations does not change the independent effect of the variable under consideration on the outcome of the corresponding effect's expression. Finally, Myers generated a number of test cases that is in between Unique and Masking MC/DC.

For all types of faults, MUMCUT and MAX-A have the highest success percentage values. MUTP and MNFP have the next best values, since they are parts of MUMCUT. Other methods have close values among each other. Although, evaluating only the success percentages is not meaningful without considering the total number of test cases and the cost of processes needed in order to generate these test cases. MI and MAX-A have the highest cost since the Boolean expressions are converted into canonical DNF first. For MUTP, MNFP, CUTPNFP and MUMCUT, a conversion is done into DNF, however, it is done by another tool which converts the Boolean expressions into irredundant DNF. The cost difference between {MI, MAX-A} and {MUTP, MNFP, CUTPNFP and MUMCUT} comes from this DNF conversion difference. Normally, MI is expected to have a better result, because it uses a DNF Boolean expression and checks for each variable in each term. However, MI has a success percentage which is close to the methods Myers, Unique MC/DC and Masking MC/DC. The reason behind is the random selection of true points in MI method, since a randomly selected true point may not detect the considered fault. Test cases generated by Myers' and MC/DC types have similar costs since there is no conversion needed.

Comparing the successes on all types of faults except MVF, choosing MAX-A or MUMCUT seems better. However, considering their computational costs and total test cases, MUMCUT is a better choice. Although MUTP is the best choice for MVF according to the success percentage, it depends on the problem size and the time constraints are important. Using the MUMCUT method may not be feasible or the number of test cases may be too high to run under certain constraints. Under these circumstances, it may be better to use Unique MC/DC, Masking MC/DC or Myers methods, since they generate less number of test cases, even if their successes are lower.

Comparing the number of mutants killed per test case for each method, Unique MC/DC has the best result since it kills the highest number of mutants per test case. But again, this metric is not solely enough to make a choice between methods, since the result for Random test case generation is the same as MAX-A, but the percentage of their mutant killing success overall 0.31 and 0.85 respectively.

Table 5.3. The number of detected mutants for TCAS-II

	Number of Detected Mutants							
	VNF+ENF	ORF	SA0+SA1	VRF	CCF	CDF	MVF	All
MI	270	176	254	146	99	91	85	1119
MAX-A	344	214	438	215	220	218	85	1734
MUTP	318	198	399	196	201	200	119	1631
MNFP	339	209	418	206	206	200	108	1686
CUTPNFP	267	168	150	130	84	64	85	948
MUMCUT	344	214	440	217	220	220	119	1774
U. MC/DC	267	172	239	139	93	78	85	1073
M. MC/DC	270	177	254	148	99	91	85	1124
Myers	267	173	252	148	92	86	119	1137
Random1	78	50	49	27	18	14	19	255
Random2	118	78	76	44	30	21	34	399
Random3	147	99	103	64	44	31	41	529
Random4	151	102	109	68	46	36	43	555
Random5	163	114	117	72	50	34	47	597
Random6	190	127	145	89	60	43	54	708
Random7	224	148	181	107	76	58	65	859

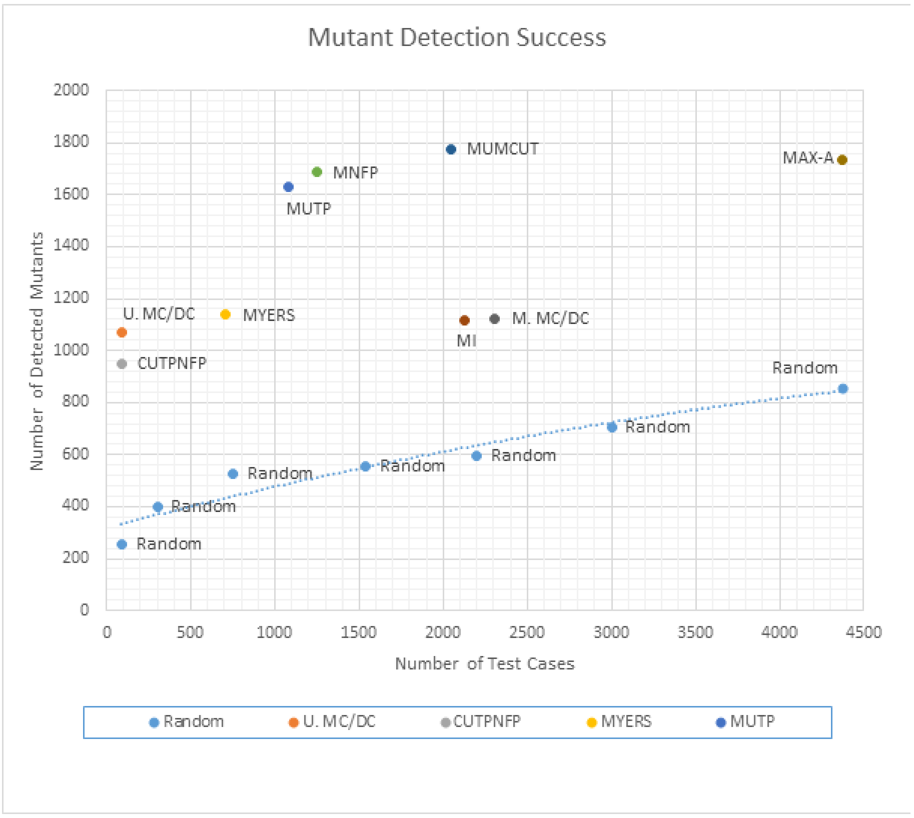


Figure 5.1. Mutant Detection Success of Methods.

Table 5.4. Percentage of detected mutants over all mutants for TCAS-II

	Percentage of Detected Mutants							
	VNF+ENF	**ORF**	**SA0+SA1**	**VRF**	**CCF**	**CDF**	**MVF**	**All**
MI	0.69	0.74	0.50	0.61	0.39	0.36	0.60	0.55
MAX-A	0.88	0.90	0.87	0.90	0.87	0.87	0.60	0.85
MUTP	0.81	0.83	0.79	0.82	0.80	0.79	0.85	0.80
MNFP	0.87	0.88	0.83	0.87	0.82	0.79	0.77	0.83
CUTPNFP	0.68	0.71	0.30	0.55	0.34	0.25	0.60	0.46
MUMCUT	0.88	0.90	0.87	0.91	0.87	0.87	0.85	0.87
U. MC/DC	0.68	0.73	0.47	0.58	0.37	0.31	0.60	0.53
M. MC/DC	0.69	0.74	0.50	0.62	0.39	0.36	0.60	0.55
Myers	0.68	0.73	0.50	0.62	0.36	0.34	0.85	0.56
Random1	0.20	0.21	0.09	0.11	0.07	0.05	0.13	0.12
Random2	0.30	0.33	0.15	0.18	0.12	0.08	0.24	0.20
Random3	0.37	0.41	0.20	0.27	0.17	0.12	0.29	0.26
Random4	0.39	0.43	0.22	0.28	0.18	0.14	0.31	0.27
Random5	0.42	0.48	0.23	0.30	0.20	0.13	0.33	0.29
Random6	0.48	0.53	0.29	0.37	0.24	0.17	0.38	0.35
Random7	0.57	0.62	0.36	0.45	0.30	0.23	0.46	0.42

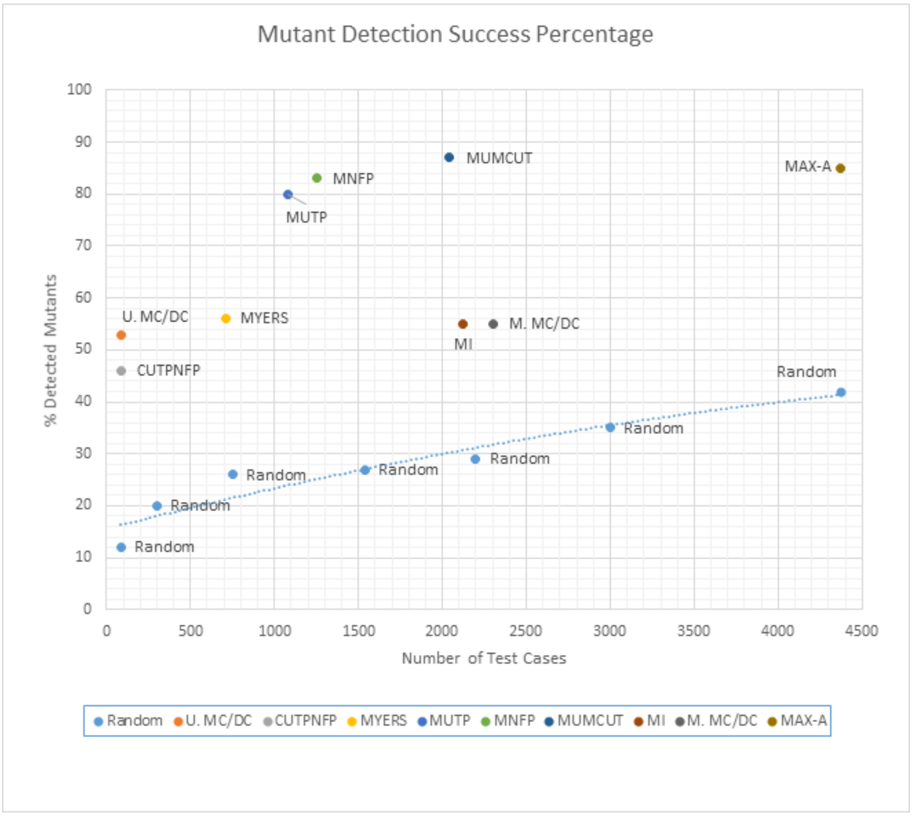


Figure 5.2. Mutant Detection Success Percentage of methods.

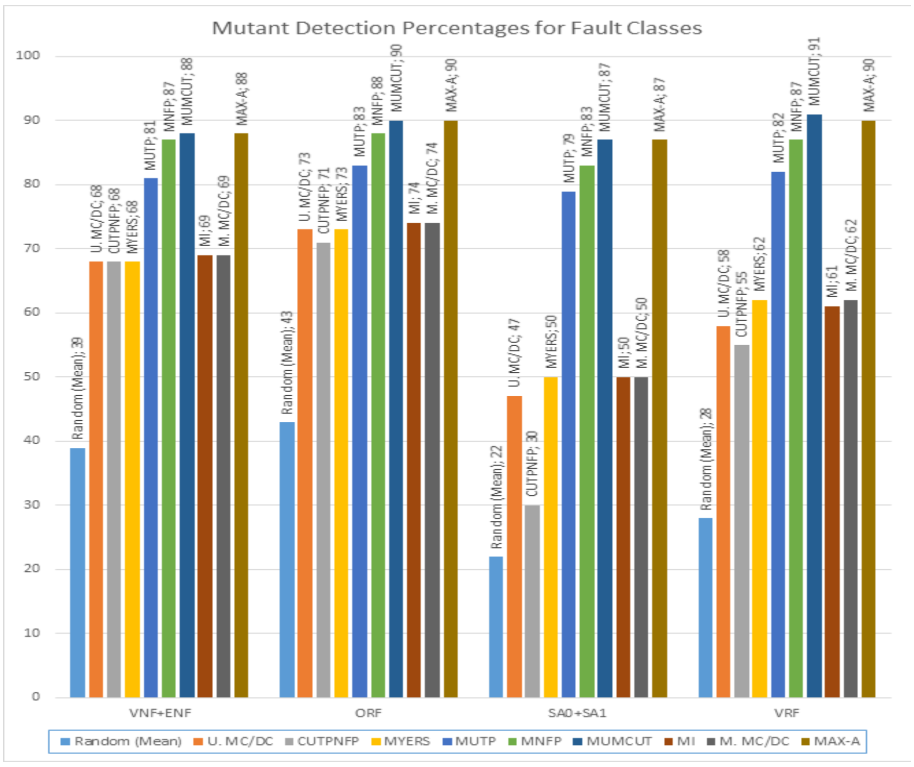


Figure 5.3. Mutant Detection Percentages for Fault Classses VNF+ENF, ORF, SA0+SA1 and VRF.

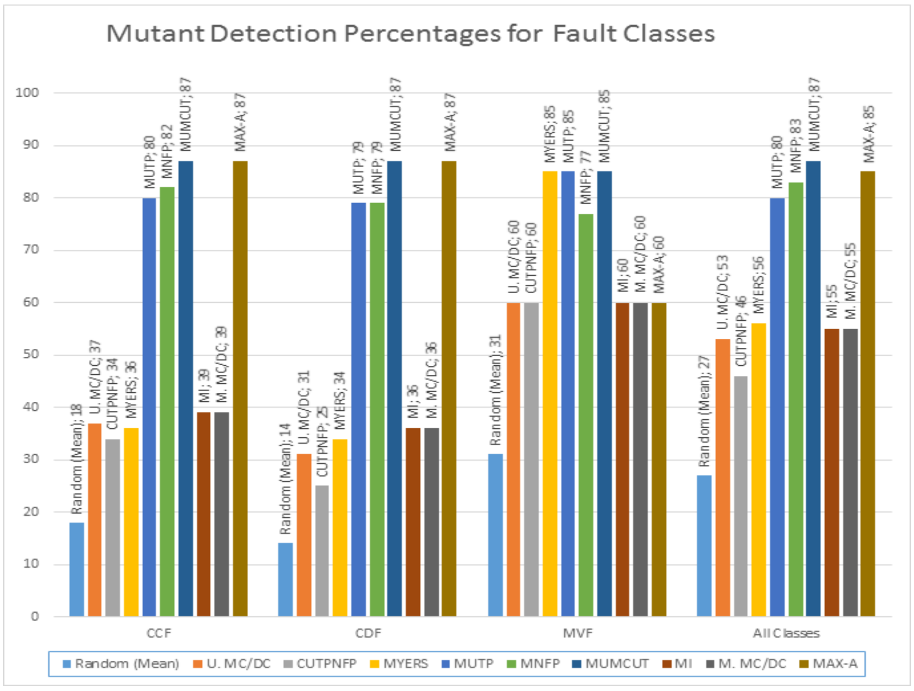


Figure 5.4. Mutant Detection Percentages for Fault Classses CCF, CDF, MVF and All Classes.

Table 5.5. The number of detected mutants per test case for TCAS-II

	Number of Detected Mutants per Test Case							
	VNF+ENF	**ORF**	**SA0+SA1**	**VRF**	**CCF**	**CDF**	**MVF**	**All**
MI	0.13	0.09	0.12	0.07	0.05	0.04	0.04	0.53
MAX-A	0.08	0.05	0.10	0.05	0.05	0.05	0.02	0.40
MUTP	0.29	0.18	0.37	0.18	0.18	0.18	0.11	1.51
MNFP	0.27	0.17	0.33	0.16	0.16	0.16	0.09	1.35
CUTPNFP	0.97	0.61	0.55	0.47	0.31	0.23	0.31	3.46
MUMCUT	0.17	0.10	0.22	0.11	0.11	0.11	0.06	0.87
U. MC/DC	3.07	1.98	2.75	1.60	1.07	0.90	0.98	12.3
M. MC/DC	0.12	0.08	0.11	0.06	0.04	0.04	0.04	0.49
Myers	0.38	0.24	0.36	0.21	0.13	0.12	0.17	1.60
Random1	0.90	0.57	0.57	0.31	0.20	0.16	0.22	2.93
Random2	0.39	0.26	0.25	0.15	0.10	0.07	0.11	1.33
Random3	0.19	0.13	0.14	0.08	0.06	0.04	0.05	0.70
Random4	0.09	0.06	0.07	0.04	0.03	0.02	0.03	0.36
Random5	0.07	0.05	0.05	0.03	0.02	0.01	0.02	0.27
Random6	0.06	0.04	0.05	0.03	0.02	0.01	0.01	0.23
Random7	0.05	0.03	0.04	0.02	0.01	0.01	0.01	0.20

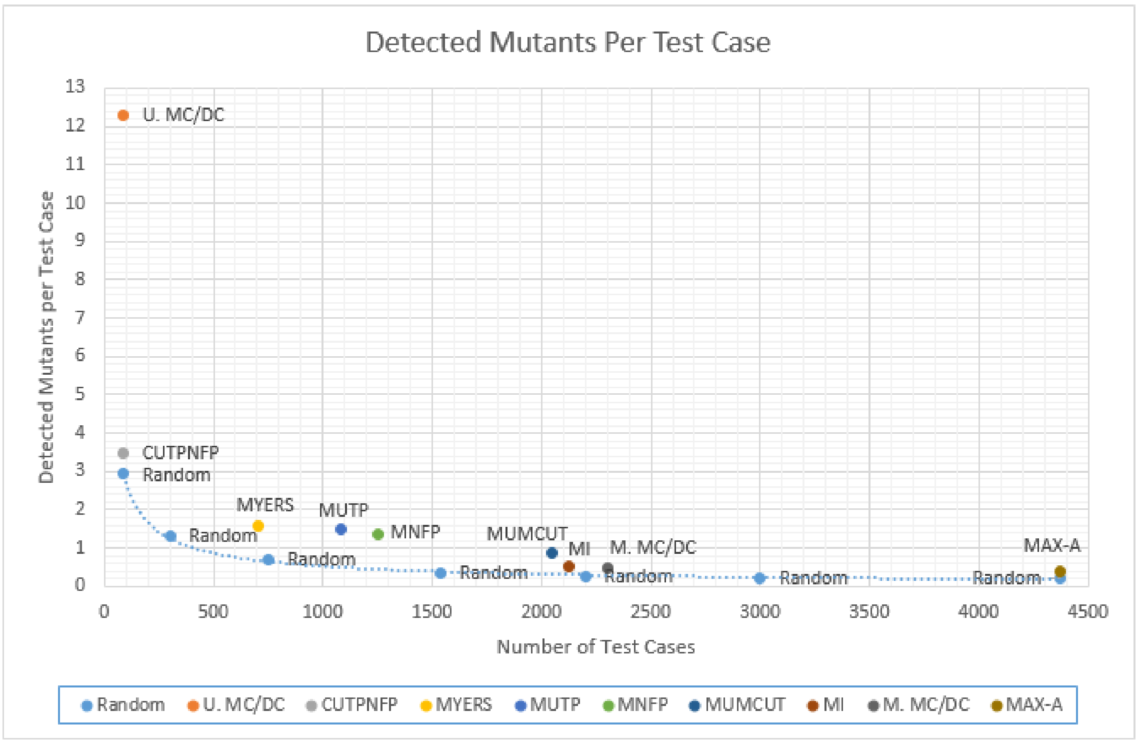


Figure 5.5. The number of detected mutants per test case for TCAS-II

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis; a cause-effect model is created where a cause-effect graph is represented in a .graphml file. By using this file, corresponding Boolean expressions are created. Then, these Boolean expressions are converted into DNF. Selected test generation techniques are implemented from the Boolean expressions or DNF expressions according to the requirements of these methods. Then, an experiment is done on some of the requirements of a real system. Mutants are created for selected fault types and mutation analysis is done on test cases generated by all implemented methods. As a result, it was seen that MUMCUT technique has the highest mutant detection success. Furthermore, Unique MC/DC detected the highest number of mutants per test case.

The conclusions made above should be verified by repeating the same experiments by using other cause-effect graphs. Results may vary in different types of programs under test. Moreover, when choosing a method over another, chosing one of them may be more logical or not according to the structure of the program and the related Boolean expressions of the effects. For example, using Masking MC/DC is meaningful for a system with expressions that have repetitive variables.

In the future, the ".graphml" model can be improved by directly deriving the relation attribute of the nodes. Furthermore, the tool can be extended in order to enable the user to import the graph with different file types that are supported by Gephi (GEXF, GDF, GML, etc.). Moreover, other test generation methods can be added into the tool as RC/DC, MIN-A, etc. In addition, a format for system requirements can be determined, so that the corresponding Boolean expressions and the representative ".graphml" file can be created automatically.

Furthermore, by using different cause-effect graphs, test cases need to be generated and compared in order to have more data to anaylse, in order to make a better conclusion on the selection of different methods. By considering the difference between the number of test cases and the probabilities of different methods' mutant detection possibilities, a better metric should be researched and used so that the comparison between methods can be more meaningful.

In mutation analysis, equivalent mutants can be examined in the mutant creation. Moreover, mutant detection success of different methods can be examined in the means of recording the exact mutants they detected. Using this knowledge, a guidance can be given in order to select method pairs. If two methods detect different mutants in the set of mutants, together these methods can be selected in order to have a better success on overall.

REFERENCES

Arcaini, P., A. Gargantini, and E. Riccobene (2015). How to optimize the use of sat and smt solvers for test generation of boolean expressions. *The Computer Journal 58*(11), 2900–2920.

Ayav, T. and F. Belli (2015). Boolean differentiation for formalizin myers' cause-effect graph testing technique. In *Software Quality, Reliability and Security-Companion*, pp. 138–143.

Badhera, U., P. G.N., and S. Taruna (2011). Fault based techniques for testing boolean expressions : A survey. *International Journal of Computer Science & Engineering 3*(1), 81–90.

(Cast), C. A. S. T. (2001). Rationale for Accepting Masking MC/DC in Certification Projects.

Chen, T., M. Lau, and Y. Yu (1999). Mumcut: A fault-based strategy for testing boolean specications. In *Asia-Pacic Software Engineering Conference*, pp. 606.

Chen, T. Y., D. D. Grant, M. F. Lau, S. P. Ng, and V. Vasa (2003). Beat:boolean expression fault-based test case generator. In *Information Technology:Research and Education Conference*, pp. 625–629.

Chen, T. Y. and M. F. Lau (2001). Test case selection strategies based on boolean specifications. *Software Testing, Verification, Reliability 11*, 165–180.

Chen, T. Y., M. F. Lau, K. Y. Sim, and C. Sun (2009). On detecting faults for boolean expressions. *Software Quality Journal 17*(3), 245–261.

Chen, Z., T. Y. Chen, and B. Xu (2011). A revisit of fault class hierarchies in general boolean specifications. *ACM Transactions on Software Engineering and Methodology 20*(3), 13:1–13:11.

Chilenski, J. and S. Miller (1994). Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal 9*(5), 193–200.

Chilenski, J. J. (2001). An investigation of three forms of the modified condition decision coverage (mcdc) criterion. Technical report, DTIC Document.

Chung, I. (2014a). Investigating effectiveness of software testing with cause-effect graphs. *International Journal of Software Engineering and Its Applications 8*(7), 41–54.

Chung, I. (2014b). Modeling pairwise test generation from cause-effect graphs as a boolean satisfiability problem. *International Journal of Contents 10*(3), 41–46.

Chung, I. (2015a). Cegpairgen: an automated tool for generating pairwise tests from cause–effect graphs. *International Journal of Software Engineering and Its Applications 9*(1), 53–66.

Chung, I. (2015b). Using boolean satisfiability solving for pairwise test generation from cause-effect graphs: Comparison of three approaches. *International Journal of Software Engineering and Its Applications 9*(9), 65–78.

Elmendorf, W. R. (1973). Cause-effect graphs in functional testing. Technical report, [Poughkeepsie, N.Y.] : IBM.

Fraser, G. and A. Gargantini (2010). Generating minimal fault detecting test suites for boolean expressions. In *Software Testing, Verification, and Validation Workshops*, pp. 37–45.

Gargantini, A. and G. Fraser (2011). Generating minimal fault detecting test suites for general boolean specifications. *Information and Software Technology 53*(11), 1263–1273.

Gephi (2016). Gephi: The open graph viz platform. `https://gephi.org/`. Online; Accessed: 2016-06-25.

GraphML (2016). The graphml file format. `http://graphml.graphdrawing.org/`. Online; Accessed: 2016-06-25.

JBooleanExpression (2016). Jbooleanexpression: Java boolean expression evaluator. `http://jboolexpr.sourceforge.net/`. Online; Accessed: 2016-06-25.

Kaminski, G., P. Ammann, and J. Offutt (2013). Improving logic-based testing. *Journal of Systems and Software 86*(8), 2002–2012.

Kaminski, G. K. and P. Ammann (2009). Using logic criterion feasibility to reduce test set size while guaranteeing fault detection. In *ICST'09: Proceedings of the 2nd International Conference on Software Testing Verification and Validation*, pp. 356–365.

Kapoor, K. and J. P. Bowen (2007). Test conditions for fault classes in boolean specifications. *ACM Transactions on Software Engineering and Methodology 16*(3), 10.

Kuhn, R. (1999). Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology 8*(4), 411–424.

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

Mathur, P. A. (2008). *Foundations of Software Testing* (First Edition ed.). Pearson Publication.

Myers, G. J. (1979). The art of software testing. *A Willy-Interscience Pub, pp.–1979.*

Nursimulu, K. and R. L. Probert (1995). Cause-effect graphing analysis and validation of requirements. In *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative research*, pp. 46. IBM Press.

Paradkar, A. (1994). On the experience of using cause-effect graphs for software specification and test generation. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, pp. 51. IBM Press.

Paradkar, A. (1995). A new solution to test generation for boolean expressions. In *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '95, pp. 48–. IBM Press.

Paradkar, A., K. C. Tai, and M. Vouk (1997). Specification-based testing using cause-effect graphs. *Annals of Software Engineering 4*, 133–157.

Paradkar, A., K. C. Tai, and M. A. Vouk (1996). Automatic test-generation for predicates. *IEEE Transactions on Reliability 45*(4), 515–530.

Paul, T. K. and M. F. Lau (2012). Redefinition of fault classes in logic expressions. In *2012 12th International Conference on Quality Software*, pp. 144–153. IEEE.

Paul, T. K. and M. F. Lau (2014). A systematic literature review on modified condition and decision coverage. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pp. 1301–1308. ACM.

Podgursky, B. (2016). jbool_expressions. `https://github.com/bpodgursky/jbool_expressions/`. Online; Accessed: 2016-06-25.

Singh, R. K., P. Chandra, and Y. Singh (2006). An evaluation of boolean expression testing techniques. *ACM SIGSOFT Software Engineering Notes 31*(5), 1–6.

Srivastava, P. R., P. Patel, and S. Hatrola (2009). Cause effect graph to decision table generation. *ACM SIGSOFT Software Eng.Notes 34*(2).

Sun, C.-A., Y. Zai, and H. Liu (2015). Evaluating and comparing fault-based testing strategies for general boolean specifications: A series of experiments. *The Computer Journal 58*(5), 1199–1213.

Sziray, J. (2013). Evaluation of boolean graphs in software testing. In *Computational Cybernetics (ICCC), 2013 IEEE 9th International Conference on*, pp. 225–230. IEEE.

Tai, K.-C. (1993). Predicate-based test generation for computer programs. In *Software Engineering, 1993. Proceedings., 15th International Conference on*, pp. 267–276. IEEE.

Tai, K.-C. (1996). Theory of fault-based predicate testing for computer programs. *IEEE Transactions on Software Engineering 22*(8), 552–562.

Tai, K.-C., A. Paradkar, H.-K. Su, and M. A. Vouk (1993). Fault-based test generation for cause-effect graphs. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pp. 495–504. IBM Press.

Tsuchiya, T. and T. Kikuno (2002). On fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology 11*(1), 58–62.

Vilkomir, S. and J. Bowen (2002). Reinforced condition/decision coverage (rc/dc): A new criterion for software testing. In *2nd International Conference, Formal Specification and Development in Z and B, Lecture Notes in Computer Science*, Volume 2272, pp. 295–313.

Vilkomir, S., O. Starov, and R. Bhambroo (2013). Evaluation of t-wise approach for testing logical expressions in software. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pp. 249–256. IEEE.

Weyuker, E., T. Goradia, and A. Singh (1994). Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering 20*(5), 353–363.

APPENDIX A

GRAPHML DESIGN

First, the cause and effect nodes are determined from the requirements of the program. Afterwards, the combinations of causes resulting in effect nodes are determined and the constraints are added if needed. Then, the corresponding graph can be drawn in any graph visualization tool and need to be exported as a ".graphml" file, Gephi is used in experiments for this thesis. Also, without using any tool, the ".graphml" file can be Written directly. There are same extra attributes need to be added to the nodes and edges in order to form a cause effect graph.

The attributes added to the nodes and how to define them:

- *grade*: This attribute is used only for visualization purposes. Cause constraints are started with grade 0, then each level represented by adding 1, i.e. Cause nodes are at grade 1, Intermediate nodes are at grade 2, etc.
- *label*: This attribute is directly the name of the node. Naming of the node labels is important. Different types of nodes must have certain labels as:
  - **Constraint** nodes must be named by their type directly, it is enumerated as: E, I, R, O, M
  - **Cause** nodes must be named starting with the letter "C" as: C1, C7, C12, etc.
  - **Intermediate** nodes must be named starting with the letter "I" as: I1, I5, etc.
  - **Effect** nodes must be named starting with the letter "E" as: E1, E8, etc.
- *relation*: Relation node represents the corresponding Boolean expression forming that node. Relation attribute exists for Intermediate and Effect nodes. For example, if an Intermediate node I5 is formed by $C1 \wedge C2 \wedge C3$;
  relation attribute must be: $C1 + C2 + C3 - AND$.
- *nodeType*: This attribute is also enumerated, represents the type of node in the cause effect graph. Can have the values: Cause, Intermediate, Effect, Constraint.
- *consInt*: This attribute is added for the Constraint nodes.

  - For the constraints connected to Cause nodes, these nodes are also connected to an Intermediate or Effect node. This attribute represents the resulting node of the Constraint. For example, if an Intermediate node I3 is formed by $C5 \vee C6 \vee C7$ with an E (Exclusive OR) Constraint; the consInt attribute in the corresponding Constraint node must be: I3.
  - For the constraint connecting Effect nodes, which is the M (Masking) Constraint; if the Effect $E3$ is Masking the Effect $E4$, the consInt attribute must be: $E3 - E4$.

- *neg*: The only attribute added to the edges is the negation attribute, which represents if the edge is negated or not. If an edge is negated, the attribute *neg* must be: NOT.

An example with the cause effect graph in Figure A.1 and the corresponding .graphml file can be found in Listing A below.

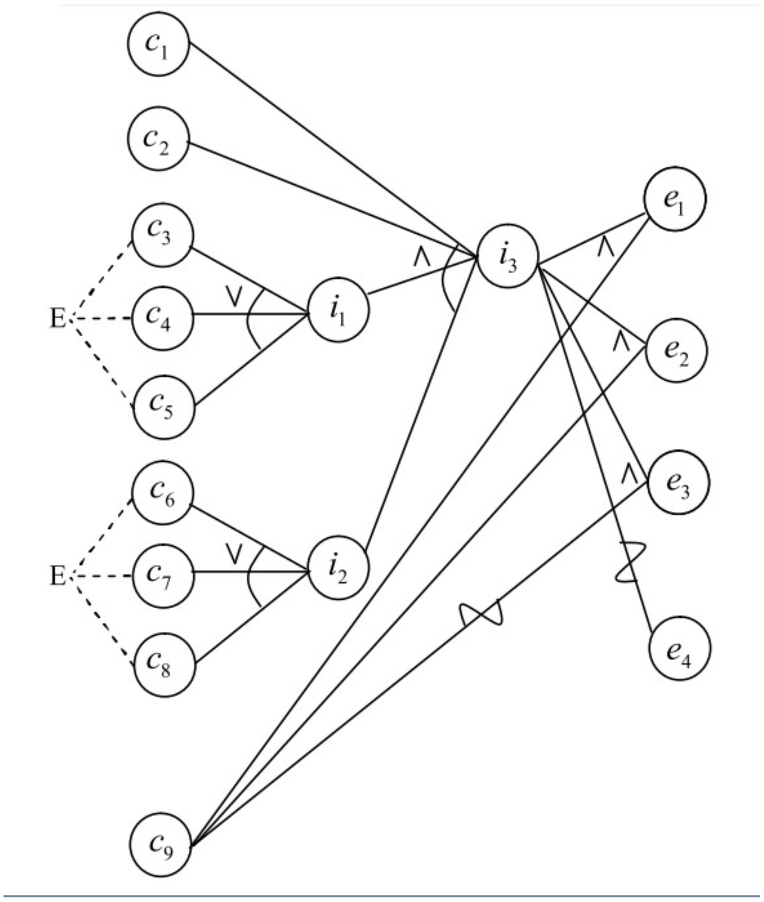


Figure A.1. Example cause effect graph.

Listing A.1 .graphml File Example

```
<?xml version="1.0" encoding = "UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
 http://graphml.graphdrawing.org/xmlns/1.1/graphml.xsd">

<!--Content: List of graphs and data-->

          <key id="grade" for="node" attr.name="grade"
           attr.type="int">
          <default>null</default>
          </key>

          <key  id="label" for="node" attr.name="label"
           attr.type="string">
          <default>null</default>
          </key>

          <key id="relation" for="node" attr.name="relation"
           attr.type="string">
          <default>null</default>
          </key>

          <key id="nodeType" for="node" attr.name="nodeType"
           attr.type="string">
          <default>null</default>
          </key>

          <key id="consInt" for="node" attr.name="consInt"
           attr.type="string">
          <default>null</default>
          </key>
```

```
<key id="neg" for="edge" attr.name="neg"
     attr.type="string">
     <default>null</default>
     </key>

     <graph edgedefault="undirected">

             <node id = "1">
                     <data key="label">C1</data>
                     <data key="grade">1</data>
                     <data key="nodeType">Cause</data>
             </node>
             <node id = "2">
                     <data key="label">C2</data>
                     <data key="grade">1</data>
                     <data key="nodeType">Cause</data>
             </node>
             <node id = "3">
                     <data key="label">C3</data>
                     <data key="grade">1</data>
                     <data key="nodeType">Cause</data>
             </node>
             <node id = "4">
                     <data key="label">C4</data>
                     <data key="grade">1</data>
                     <data key="nodeType">Cause</data>
             </node>
             <node id = "5">
                     <data key="label">C5</data>
                     <data key="grade">1</data>
                     <data key="nodeType">Cause</data>
             </node>
```

```
<node id = ”6”>
        <data key=”label”>C6</data>
        <data key=”grade”>1</data>
        <data key=”nodeType”>Cause</data>
</node>
<node id = ”7”>
        <data key=”label”>C7</data>
        <data key=”grade”>1</data>
        <data key=”nodeType”>Cause</data>
</node>
<node id = ”8”>
        <data key=”label”>C8</data>
        <data key=”grade”>1</data>
        <data key=”nodeType”>Cause</data>
</node>
<node id = ”9”>
        <data key=”label”>C9</data>
        <data key=”grade”>1</data>
        <data key=”nodeType”>Cause</data>
</node>
<node id = ”10”>
        <data key=”label”>E</data>
        <data key=”grade”>0</data>
        <data key=”nodeType”>Constraint</data>
        <data key=”consInt”>I1</data>
</node>
<node id = ”11”>
        <data key=”label”>E</data>
        <data key=”grade”>0</data>
        <data key=”nodeType”>Constraint</data>
        <data key=”consInt”>I2</data>
</node>
<node id = ”12”>
```

```
            <data key="label">I1</data>
            <data key="grade">2</data>
            <data key="nodeType">Intermediate</data>
            <data key="relation">C3+C4+C5 − OR</data>
        </node>
        <node id = "13">
            <data key="label">I2</data>
            <data key="grade">2</data>
            <data key="nodeType">Intermediate</data>
            <data key="relation">C6+C7+C8 − OR</data>
        </node>
        <node id = "14">
<data key="label">I3</data>
<data key="grade">3</data>
<data key="nodeType">Intermediate</data>
<data key="relation">C1+C2+I1+I2 − AND</data>
        </node>
        <node id = "15">
            <data key="label">E1</data>
            <data key="grade">4</data>
            <data key="nodeType">Effect</data>
            <data key="relation">I3+C9 − AND</data>
        </node>
        <node id = "16">
            <data key="label">E2</data>
            <data key="grade">4</data>
            <data key="nodeType">Effect</data>
            <data key="relation">I3+C9 − AND</data>
        </node>
        <node id = "17">
            <data key="label">E3</data>
            <data key="grade">4</data>
            <data key="nodeType">Effect</data>
        <data key="relation">I3+~C9 - AND</data>
</node>
<node id = "18">
        <data key="label">E4</data>
        <data key="grade">4</data>
        <data key="nodeType">Effect</data>
        <data key="relation">~I3 - NOT</data>
</node>

<edge source = "10" target = "3" />
<edge source = "10" target = "4" />
<edge source = "10" target = "5" />

<edge source = "11" target = "6" />
<edge source = "11" target = "7" />
<edge source = "11" target = "8" />

<edge source = "1" target = "14" />
<edge source = "2" target = "14" />

<edge source = "3" target = "12" />
<edge source = "4" target = "12" />
<edge source = "5" target = "12" />

<edge source = "12" target = "14" />

<edge source = "6" target = "13" />
<edge source = "7" target = "13" />
<edge source = "8" target = "13" />

<edge source = "13" target = "14" />

<edge source = "14" target = "15" />
```

```
                <edge source = ”14” target = ”16” />
                <edge source = ”14” target = ”17” />
                <edge source = ”14” target = ”18” >
                        <data key = ”neg”>NOT</data>
                </edge>

                <edge source = ”9” target = ”15” />
                <edge source = ”9” target = ”16” />
                <edge source = ”9” target = ”17”>
                        <data key = ”neg”>NOT</data>
                </edge>

        </graph>
</graphml>
```