

Abstract

This dissertation is devoted to the evaluation of the Modified Condition / Decision Coverage (MC/DC) as a structural coverage metric for hardware verification. Initially, different approaches for MC/DC test suite calculation are highlighted. Afterwards, an experimental evaluation of the difference between Focused Expression Coverage (FEC) and MC/DC is presented. FEC as MC/DC checks that every condition has taken both true and false values under conditions that allow it to control the output. However, as it will be shown it is wrong to consider FEC and MC/DC equivalent. Furthermore, OBSRV procedure which is a novel proposal for MC/DC test suite generation is illustrated. Before this project, OBSRV procedure was in a pencil and paper manner, which is now translated to an automated software implementation. Using the implementation, bigger (in terms of the number of conditions) expressions can be evaluated and their corresponding test suites can be straightforwardly calculated. The dissertation provides a detailed description of the OBSRV implementation as well as a brief evaluation of the procedure. Finally, this dissertation provides a general MC/DC overview while providing an evaluation of the MC/DC as a structural coverage metric for hardware verification.

Table of Contents

1. MODIVATION, AIMS AND OBJECTIVES	
2. INTRODUCTION	1
3. BACKGROUND	1
3.1. Boolean algebra	4
3.2. The Verification Process	5
3.3. Definition and Role of Coverage	6
4. TYPES OF STRUCTURAL COVERAGE	8
4.1. Statement Coverage	8
4.2. Decision Coverage	8
4.3. Condition Coverage	8
4.4. Condition/decision coverage	9
4.5. Modified Condition/Decision Coverage (MC/DC)	9
4.6. Multiple Condition Coverage	9
5. MODIFIED CONDITION / DECISION COVERAGE (MC/DC) BACKGROUND	11
5.1. MC/DC usage as a structural coverage criterion	11
5.2. The three conventional MC/DC approaches	11
5.2.1. Unique-Cause Approach	12
5.2.2. Masking MC/DC approach	12
5.2.3. NASA: Gate-Level Evaluation Approach	12
5.2.3. a) Basic gates treatment in order to achieve MC/DC	13
5.2.3. b) NASA: Gate-Level Evaluation Approach – Analysis of the five steps	14
6. EXPERIMENTAL EVAVLUATION OF THE DIFFERENCE BETWEEN FEC AND MC/DC	18
6.1. Experimental setup	18
6.2. First Experiment: XOR gate (A xor B)	20
6.2.1. MC/DC test suite calculation	20
6.2.2. ModelSims' FEC coverage results	20
6.3. Second Experiment: Expression $F := (A \text{ and not } B) \text{ or } (C \text{ xor } D)$.	22
6.3.1. MC/DC test suite calculation	22
6.3.2. ModelSims' FEC coverage results	25
7. NOVEL MC/DC TEST GENERATION METHOD: OBSRV	26
7.1 OBSRV Procedure	26
7.1.1. Gate structure representation:	26
7.1.2. Computation of conditions' observability test vectors	26
7.1.3. Observability test vectors Ranking	27
7.1.4. Observability Matrix Construction	27
7.1.5. MC/DC test suite(s) Determination	27
7.2 Experimental evaluation of OBSRV	27
7.2.1 First example of the OBSRV usage	27
7.2.2 Another experiment using OBSRV algorithm	30
8. OBSRV IMPLEMENTATION	32
8.1. Requirements	32
8.2. Specification	32
8.3. Design	34

8.3.1	Textual representation of the Expression	34
8.3.2	Topological Traversal	37
8.3.3	Expression evaluation using the truth table of the expression	39
8.3.4	Internal data structure correctness – expression verification	40
8.3.5	D-value simulation	40
9.	TEST SUITE CACLULATION	44
9.1	Observability table processing	44
9.1.1	Observability table processing using online algorithm	44
9.1.2	Observability table processing using greedy algorithm	46
9.2	Test Case Selection	49
10.	DESIGN IMPLEMENTATION	52
10.1.	Observability matrix calculation	52
10.2.	Test suite calculation	54
10.2.1	Brief description of the functionality of the functions used for test suite calculation	
10.2.2.	Tree structure construction	56
10.2.3.	Pseudo code for the implementation	57
10.3.	Design verification	57
10.3.1.	Expression graph structure validation	58
10.3.2.	Tree structure validation	58
11.	RESULTS	60
11.1.	Test suites calculated under experimental expressions	60
11.2.	Differences identified between pencil and paper method presented in [17] and implementation results.	
11.3.	Complexity analysis of OBSRV implementation	62
12.	CONCLUSIONS AND FUTURE PLANS	63
	REFERENCES	64
	APPENTICES	66
	Appendix 1: OBSRV implementation source code of basic functions	67
	Appendix 2: Experimental evaluation of the difference between FEC and MC/DC	71
	Appendix 3: Poster	72

1. Motivation, Aims and Objectives

The inspiration to investigate the effectiveness of MC/DC criterion as structural coverage metric for hardware verification was sparked by a work proposed by a PhD student of the University of Bristol in the draft paper “**Modified Condition Decision Coverage Review: A Hardware Verification Perspective**” by Mohamed A. Salem (2010) [17]. The paper compiles a comprehensive review of the current MC/DC conventions and develops novel MC/DC insights through the conduction of experimental study for MC/DC in hardware verification spectrum. Also, it proposes an innovative method for MC/DC test suite generation named **OBSRV**. This method was presented in a theoretical-pencil and paper manner which after the completion of this project was converted in an automate software implementation. The OBSRV method has been experimentally evaluated on diverse base of logic combinations which emerge distinct MC/DC insights. These insights present novel MC/DC aspects, optimize the minimal MC/DC requirements and provide RTL design guidelines for MC/DC fulfillment.

The aims of this project are the following:

1. The first objective of this project is the experimental evaluation of difference(s) between Focused Expression Coverage (FEC) and MC/DC.
2. Investigate and re-evaluate OBSRV test suite generator, which is a novel proposal for MC/DC test generation.
3. Specify, design and implement the first (naive) version of OBSRV.
4. Iterate the OBSRV implementation in order to identify possible improvements/adaptations in the test suite generation process of OBSRV. More precisely it focuses on the test cases selection step and explores how this process can be automated.
5. Evaluation of MC/DC strength as structural coverage criterion.
6. Potential for publication.

2. Introduction

Verification and validation is the process of checking that a system fulfills its purpose and covers the specifications. This process is a critical and integral phase of the systems design/development cycle. Especially for critical, real-time applications such as commercial avionics this process is requisite and it is one of the most expensive activities employed during the development life-cycle. With testing the designers confidence that the system works correctly and meets the specifications is promoted. Code coverage is a measure in software testing which highlights the degree to which the source code of a system has been tested.

The testing of different modules of the system can be either structural (white box testing) or functional (black box testing) [32] or even a combination of the two. This project focuses on structural coverage analysis techniques where the internal structure of the system is used in order

to design proper test cases. The purpose of structural coverage analysis is to provide greater level of confidence by complementing the requirements-based testing. Moreover, structural coverage techniques inspect the code directly and provide evidence that the code structure is verified to the required level by demonstrating that all existing code is reachable and adequately tested.

In order for the aviation software to be approved by the Federal Aviation Administration (FAA)¹ [01], it needs to comply with the RTCA/DO-178B document Software Considerations in Airborne Systems and Equipment Certification [03]². In DO-178B the activities and objectives that must be implemented during the software life cycle are described. DO-178B is the document used by FAA as guidance to determine if the software will perform safely and reliably, in the critical airborne environment. It is required that testing of software must achieve **MC/DC** (Modified Condition/Decision Coverage) criteria for Level-A software; Level-A software is the software whose anomalous behavior could lead to catastrophic consequences.

MC/DC as described in DO-178B [03], is a structural coverage measure and a mandatory requirement for testing avionics software according to **FAA** [01]. MC/DC combines the requirements for decision coverage with those for condition coverage and in addition, it requires that each condition has been independently affecting the decision's outcome. In order to completely verify a decision with n conditions, 2^n test-cases are required (exhaustive testing). MC/DC criteria were developed to provide many of the benefits of Boolean expressions exhausting testing with fewer test cases; generally, using MC/DC for a decision with n conditions, $n+1$ test-cases are required.

MC/DC is a structural coverage measure which is strongly related to Boolean logic. In **section 3**, the basic background of Boolean algebra needed in order to understand the principles which *are widely used in* MC/DC techniques is presented. In the **subsection 3.2**, the role of verification and validation process in the software's life cycle is highlighted. After that, (in **subsection 3.3**) the role of coverage is explained and in the following section (**section 4**) the different types of structural coverage are presented. **Section 5** focuses on MC/DC where the three conventional approaches (Unique-Cause approach, Masking MC/DC approach and NASA: Gate-Level Evaluation) are presented.

Recently, MC/DC coverage has been introduced into the hardware verification process. However, not enough is known about the effectiveness of MC/DC coverage in this context; this motivates the aims and objectives of this project. In **section 6** a novel MC/DC test generation method called **OBSRV** is introduced. OBSRV is a novel proposal by a Bristol University PhD student which takes an expression as input and it calculates its MC/DC (~FEC) test suite. Firstly,

¹ The **Federal Aviation Administration (FAA)** is an agency of the United States Department of Transportation with authority to regulate and oversee all aspects of civil aviation in the U.S. (National Airworthiness Authority).[02]

² The **Radio Technical Commission for Aeronautics (RTCA)** develops guidance documents related to the FAA. RTCA is a not-for-profit corporation formed to advance the art and science of aviation and aviation electronic systems for the benefit of the public [03][13].

there is an introduction to OBSRV procedure which consists of five steps and then a more detailed analysis of OBSRV applied in real examples is presented. In **section 7**, the basic implementation techniques used are highlighted. In **section 8**, a comprehensive description of the MC/DC test suite calculation is presented using examples and figures. In **section 10**, the basic code segments and operations of the functions of OBSRV implementation are illustrated with a lot of details. **Section 11**, corresponds to the results section in which the test suites calculated using the implementation and a comparison table between the manual calculated and the generated results, are shown. In addition in **subsection10.3**, a brief complexity analysis of OBSRV implementation is highlighted. In the final section (**section 12**), a short overview of the work and the future plans are presented.

3. Background

3.1 . Boolean algebra

Most of the notation used in the report is based on the laws of Boolean algebra. Boolean algebra is the algebra with two values 1 and 0, although T (true) and F (false) are also used. Boolean algebra also consists of operations. Whereas elementary algebra is based on numeric operations such as addition $x + y$, multiplication xy and negation $\neg x$, Boolean algebra is based on logical operation similar to those operations, namely conjunction $x \wedge y$ (**AND**, “**x.y**” , “**x*y**”), disjunction $x \vee y$ (**OR** , “**x+y**”), and complement or negation $\neg x$ (**NOT**, “**x’** ”).

$$F = (A \text{ AND } B) \text{ OR } C$$

is an example of Boolean function, where A , B and C are called Boolean variables and “**AND**” and “**OR**” represent the Boolean operators. The logical AND is similar to binary multiplication; for example:

- Logical AND (“**•**”): $0 \cdot 0 = 0$, $0 \cdot 1 = 0$, $1 \cdot 0 = 0$, $1 \cdot 1 = 1$
- Logical OR (“**+**”): $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, $1 + 1 = 1$ ($\neq 2$)

Usually Boolean functions are express in a tabulate form which indicates each combination of every possible value of their expression.

Since in digital logic, Boolean expression only has two possible values: True and False, therefore, we can analyze any logical statement which contains a finite number of logical variables using a table which lists all possible values of the variables; this is called a **truth table**. Since each variable can take only two values, a statement with “ n ” variables requires a table with 2^n rows.

For example for the Boolean expression $F = (A \text{ AND } B) \text{ OR } C$ the truth table is the following:

Conditions:	A	B	C	F(Output)
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

A Boolean **condition** is a Boolean value that cannot be broken down into simpler Boolean expressions. For the expression used in the example before $F = (A \text{ AND } B) \text{ OR } C$; A , B and C are the conditions. Conditions are leafs in the parse tree of an expression. The parse tree for the expression is shown in the following figure [10].

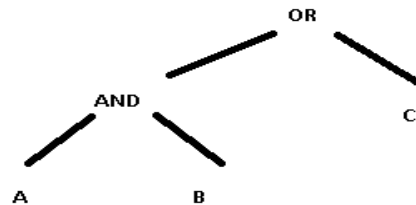


Figure1: Expression parse tree, conditions A, B and C are represented as leafs.

3.2. The Verification Process

Verification and testing of critical applications such as avionics, medical and banking systems is a complex, time consuming and expensive process during the electronic systems design/fabrication cycle. According to RTCA/DO-248A, the Second Annual Report for Clarification of DO-178B [04]:

"No product of human intellect comes out right the first time. We rewrite sentences, rip out knitting stitches, replant gardens, remodel houses, and repair bridges. Why should software be any different?" [04]

The role of software verification and testing is crucial during the development process. The verification process does not produce any software; however, its responsibility is to ensure that the produced software implements the intended functionality correctly according to the specifications. As can be seen in Figure 02, the verification process is integrated in the development process. The verification activities in the development cycle are intended to help "build in" quality at each step. Iterative testing promotes the designer's confidence that the design works correctly according to the specifications. In contrast, a final verification and testing step (instead of iterative testing) is impractical because first of all it will probably be complex and expensive in order to test the whole functionality and specifications. Also, maybe some of them would be impossible to be tested and in the case of bug detection the whole design/development cycle must be implemented again from the begging; this costs money and takes time, so small and more frequent testing/verification steps are more efficient.

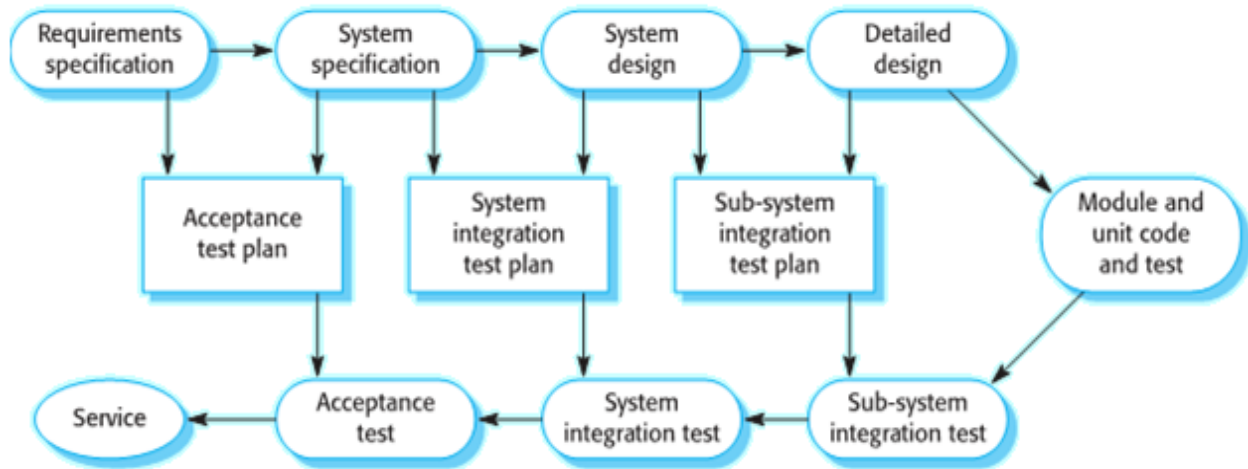


Figure 02: The V-model of development; Testing is integrated in the design cycle (modified from Ref. [05])

3.3. Definition and Role of Coverage

Coverage refers to the degree to which a verification/testing activity has satisfied its objectives. An appropriate coverage measure gives the verifier the sense that the verification is accomplished. In other words it is providing an indication for when testing is enough and reassures the designer that the system meets the specifications. Coverage is a measure, not a method or a test, so it is usually expressed as the percentage of an activity that is accomplished. Except from process assurance, coverage also conducts to process improvement due to its integration to the design cycle.

There are two specific measures of test coverage: **requirements coverage** (black box testing) and software **structure coverage** (also known as structural coverage) (white box testing). Requirements coverage analysis determines how well the requirements-based testing verifies the implementation of the software requirements (DO-178B, section 6.4.4.1) [03]. Structural coverage analysis determines how much of the code structure was executed by the requirements-based tests (DO-178B, section 6.4.4.2) [03].

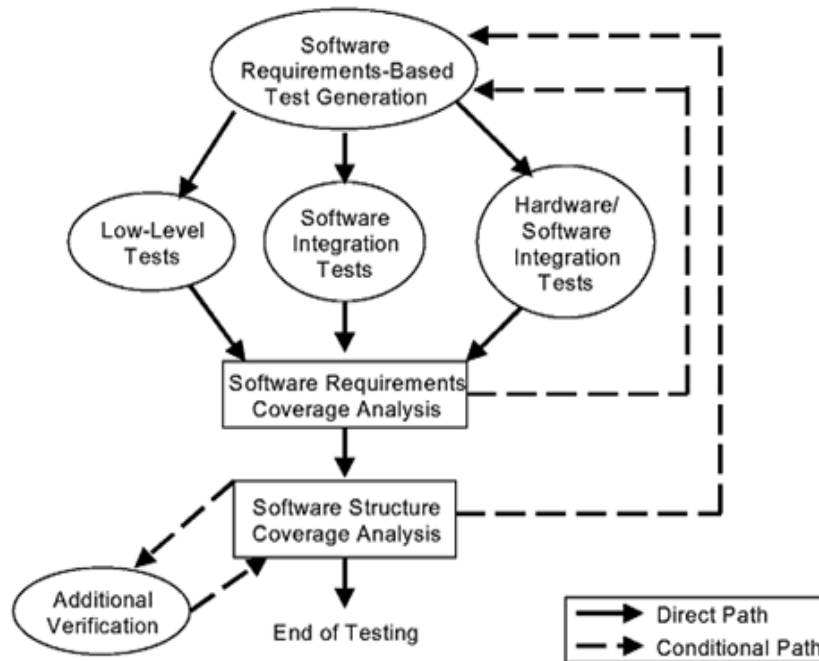


Figure 03: DO-178B Software testing activities (taken from Ref. [06])

This project is focused on structural coverage analysis techniques where the internal structure of the system is used in order to design proper test cases. The purpose of structural coverage analysis is to provide greater level of confidence by complementing the requirements-based testing. Moreover, it provides evidence that the code structure is verified to the required level by demonstrating that all existing code is reachable and adequately tested.

4. Types of Structural Coverage

The structural coverage criteria are divided into two types: **control flow** and **data flow** [07]. Control flow criteria measure the flow of control between the statements. In DO-178B standard, the structural coverage criterion is control flow. For control flow criteria, the degree of structural coverage achieved is measured in terms of statement invocations, Boolean expressions evaluated, and control conditions exercised.

As for data flow criteria, they measure the flow of data between variable assignments and references to the variables. Data flow metrics, involve analysis of the paths (or sub paths) between the definition of a variable and its subsequent use.

4.1. Statement Coverage

Statement coverage can only indicate that all code statements are reachable. This type is considered a weak criterion because it is insensitive to some control structures thus, the execution of a statement does not imply anything about correctness of the implementation. For example in the following code

```
if (x > 1) AND (y = 0) then
    z := z / x;
end if;
if (z = 1) OR (y > 7) then
    z := z + 1;
end if;
```

Let $x = 4$, $y = 0$, and $z = 4$ be the input to this code segment; then every statement is executed at least once. However, if there is an error and the “AND” decision (gate) in the first statement, is coded as an “OR” by mistake, then the test case will not detect any problem [08].

4.2. Decision Coverage

This type of structural coverage measures the control flow or branching execution of the code. It requires two test cases: one for a true and another for a false outcome of the decision statement. For the decision (A OR B), the test cases (A=T, B=F) and (A=F, B=F) will alert the decision outcome between true and false. However, with those tests the effect of B is not tested; B stays constant to the value F. For example the result could be the same if the expression was just “A” instead of “A OR B”.

4.3. Condition Coverage

Condition coverage requires that each condition in a decision takes on all possible outcomes at least once (this covers the previous problem of decision coverage), but does not require that the decision take on all possible outcomes at least once. For the previous example (A OR B), the test

cases (A=T, B= F) and (A=F, B=T) meet the condition coverage criterion, but do not cause the decision to take on all possible outcomes. [08]

4.4. Condition/Decision coverage

Condition/decision coverage requires that all conditions of a decision take all possible values and in addition they are switching the decision values between True and False. That way the requirements for condition coverage are combined with those for decision coverage. This indicates that there must be sufficient test cases to alter the decision outcome between true and false and to alter each condition value between true and false. Therefore, a minimum of two test cases is necessary for each decision. Using the same example (A AND B), the test cases (A=T, B= T) and (A=F, B= F) would meet the coverage requirement. However, using these two tests the correctness of expression cannot be distinguished. For example the expression A, the expression B or the expression (A and B) will have the same result.

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

According to DO-178B, MC/DC definition is the following [03]:

“Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision’s outcome. A condition is shown to independently affect a decision’s outcome by varying just that condition while holding fixed all other possible conditions.” (taken from Ref. [03]).

It is important to realize that MC/DC criterion enhances the condition/decision coverage criterion by requiring that **each condition is shown to independently affect the outcome of the decision**. That ensures that each condition effect is tested relative to the other conditions. MC/DC requires in general, a minimum of $n+1$ test cases for a decision with n inputs. For the previous example (A OR B), test cases (A=T, B=F), (A=F, B=T), and (A=F, B=F) provide MC/DC. MC/DC criteria were developed to provide many of the benefits of Boolean expressions exhausting testing with fewer test cases; generally $n+1$ test cases are required instead of 2^n where n represents the number of conditions. The MC/DC criterion will be analyzed further in the next sections.

4.6. Multiple Condition Coverage

Last but not least, multiple condition coverage requires exhaustive testing of the input combinations to a decision test case; that ensures that each possible combination of inputs to a decision is executed at least once. Ideally, multiple condition coverage is what should be done in order to be sure that the verification and validation state has actually investigated all possible cases. For a decision with n inputs, multiple condition coverage requires 2^n tests. Unfortunately,

this is not always possible, especially for the large circuits used in avionics domain, where complex Boolean expressions are common, this is almost impossible to be done. There are cases where decision can be found with 30 or more conditions, which mean 2^{30} tests required just for a single decision [06].

5. Modified Condition/ Decision Coverage (MC/DC) background

5.1. MC/DC usage as a structural coverage criterion

MC/DC is a structural coverage measure and a mandatory requirement for testing avionics software. The reason for MC/DC usage as a structural coverage criterion is mainly because it attempts to provide a cost-effective form of verification. Furthermore, MC/DC can be applied to any representation of logic (graphical, textual and mathematical) not just for logic verification, while the other forms of structural coverage cannot be expressed. Also, it can be applied in any stage of the development process/cycle. For example it can be applied to source code which is the lowest level or it can be applied at the requirements level which is the highest level of coverage.

On the other hand, as explained by Bhansali in The MC/DC paradox [09], the MC/DC is proven to miss the detection of common errors and because of this its usage as structural testing strategy for the software's used in commercial aviations, could not contribute to the detection of a possible failure of the system which will result in a catastrophic failure condition for the aircraft. Moreover, it is believed that covering the requirements to the MC/DC level, would also achieve the coverage of the source code. Unfortunately this is not guaranteed to happen (triangle problem [10]). As is analyzed in [10] Appendix A, since MC/DC has limited success in requirements-based verification on a simple problem (triangle problem), we should consider how limited the performance of MC/DC could have in a real system [18].

Results of a survey that took place by the aviation software industry in 1999 showed that more than 75% of the respondents, claimed that it was difficult to meet the MC/DC (DO-178B) requirement, and 74% of the respondents said the cost was substantial or nearly prohibitive [11]. Also many of the responders argue about the effectiveness of the MC/DC with respect to its cost. Furthermore, another study by Dupuy and Leveson [12] found that MC/DC criterion worth one's salt, even though it is relatively expensive, it is not significantly more expensive than achieving lower levels of code coverage. Moreover, with MC/DC important errors can be found by the additional test cases that are required to achieve MC/DC coverage.

5.2. The three conventional MC/DC approaches

Many forms of MC/DC exist; some of these interpretations (forms) result in a verification that is better at detecting errors than others. Other interpretations result in verification that is more costly than others. In the next paragraphs there are presented the three conventional MC/DC approaches and we discover their differences. The three conventional MC/DC approaches are: MC/DC as unique-cause, masking MC/DC and NASA evaluation approach. Before analyzing the three approaches two important terms need to be stated. **Controllability** means the ability of

testing each logical operator of an expression by setting the values of its inputs; this way the output value is under control. **Observability** is the ability to propagate the output of a logical operator under test to an observable point (usually output).

5.2.1. Unique-Cause Approach

An independence-pair is used to show the independent effect of each logic condition on the decisions' outcome. An independence-pair in unique-cause is consisted from two test cases that differ only in the values of the condition of interest and the decisions' outcome while all the other conditions are hold fixed. As a result, this guarantees that the only condition that influences the output is the condition of interest. For MC/DC, a minimum of one pair pre condition is needed in order to obtain the minimum tests for each logical operator in the decision.

5.2.2. Masking MC/DC approach

Masking MC/DC is another approach to ensure that no other condition influences the outcome of a decision, even if some conditions change their values [16]. The best way to understand the difference of masking MC/DC from Unique-cause is through an example. Let $F = A \text{ AND } (B \text{ OR } C)$. In this example, in order to show the independent effect of the condition A the sub term $(B \text{ OR } C)$ must be true. If the sub term $(B \text{ OR } C)$ is false then the decision outcome will be false whatever the value of A is. In Unique-cause the values of B and C must be fixed when the goal is to show the independence effect of A . However, in masking MC/DC this is not necessary, so B and C can change values in independence pair for A as long as the outcome of $(B \text{ OR } C)$ is true. Then, masking MC/DC consists of larger number of a certain condition's independence-pairs than unique-cause, therefore, it outputs an enlarge variety of MC/DC test suit. Masking MC/DC requires observability of each condition possible values; true and false at the decision output.

5.2.3. NASA: Gate-Level Evaluation Approach

This approach is using gate-level schematic representations of the decision in order to evaluate MC/DC. Initially it applies to the gate structure the identified requirements-based tests from a particular gate evaluation. Then, the masked tests whose outputs are not propagated to the output node (no way to observe) are eliminated from the test suite. Finally, the remaining tests are checked if they fulfill the minimum MC/DC tests for each gate [06].

In [14] a practical five-step approach for assessing MC/DC for aviation software products is presented, and an analysis of some types of errors expected to be caught when MC/DC is achieved, is highlighted. For MC/DC evaluation using a gate-level approach, each logical operator in a decision in the source code is examined to determine whether the requirements-based tests have observably exercised the operator using the minimum test criteria. According to [06] the five steps needed are the following:

- (1) Create a schematic representation of the source code.
- (2) Identify the test inputs used. Test inputs are obtained from the requirements-based tests of the software product.
- (3) Eliminate masked test cases; when its results are hidden from the observed outcome.
- (4) Determine MC/DC based on the minimum test criteria for each operator.
- (5) Confirm correct operation of the software by examining the outputs of the tests. The aim of this step is to confirm that the schematic representation of the source code provides the same results.

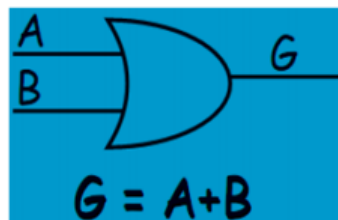
5.2.3. a) Basic gates treatment in order to achieve MC/DC

It is essential to understand how to test the basic logical operators (gates) “AND” and “OR”. The schematic representations and the corresponding truth tables for these operators are shown in the next Figure04.



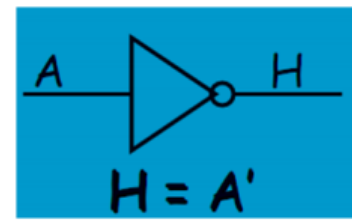
“AND” gate truth table

A	B	F = A · B
0	0	0
0	1	0
1	0	0
1	1	1



“OR” gate truth table

A	B	G = A + B
0	0	0
0	1	1
1	0	1
1	1	1



“NOT” gate truth table

A	H
0	1
1	0

Figure 04: Schematic representations of the three basic operators

- “AND” Gate testing

Minimum testing to achieve MC/DC for an n-input “AND” gate requires the following:

- (1) A single test case where all inputs are set to ‘true’ with the output observed to be true.
- (2) Test cases such that each input is set exclusively ‘false’ with the output observed to be false; in first test case, it is set only the first input to false value, then the second input and so on until all inputs have been set at least one time false value. This requires n test cases for each n-input and gate.

Any false input in an “AND” gate results a false output. In order to show independent effect we have to use the test case consisting of all true inputs with test cases that have only one input

false; this way we can ensure that each individual input is influencing the output. Hence, a specific set of $n+1$ test cases is needed to provide coverage for an n -input and gate.

- “**OR**” Gate testing

Similar analysis like “AND” gate is valid for an “OR” gate, but the “OR” gate is sensitive to true value. If an input is true then its output is true independent of what the values of the other inputs are. Here again, $n+1$ specific test cases are needed to test an n -input or gate. These specific $n+1$ test cases meet the intent of MC/DC by demonstrating that the “OR” gate is correctly implemented.

- “**NOT**” Gate testing

For “NOT” gate two test cases are needed; one for true input and one for false input value.

- “**XOR**” Gate testing

The “XOR” gate is different with respect to MC/DC from the “AND” and “OR” gates. According to [15] there are multiple minimum test set for XOR. Any three test sets from the possible exhaustive test can be selected (TT, TF, FT), (TF, FT, FF), (FT, TT, FF), or (TT, FF, TF). Note that in order to distinguish between an “OR” and an “XOR” gate, the test case with all inputs true (TT) should be present in the test set. Also, XOR operation may be represented by combination of “AND” and “OR” operations. The expression $A \text{ XOR } B$ can be rewritten $(A \text{ OR } B) \text{ AND } (\text{NOT } (A \text{ AND } B))$. This implementation requires four test cases (exhaustive testing) in order to provide MC/DC.

5.2.3. b) NASA: Gate-Level Evaluation Approach – Analysis of the five steps

Step one: Schematic representation of the source code

The first step is to represent the source code to gate schematic representation. For example $Z = (A \text{ OR } B) \text{ AND } (\text{NOT } (A \text{ AND } B))$ is represented in the next figure 05.

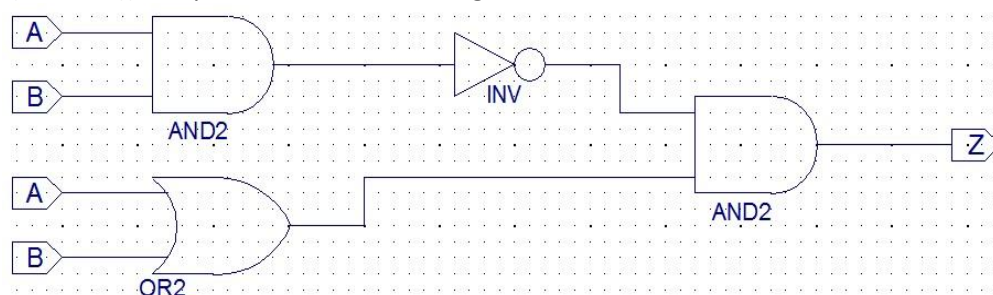


Figure 05: Schematic representation of the source code $Z = (A \text{ OR } B) \text{ AND } (\text{NOT } (A \text{ AND } B))$ ³

³ The figure was created using [Xilinx ISE Design Suite 10.1](#)

Step two: Identification of Test Inputs

In that step of the process, the inputs from the requirements-based test cases are taken and are mapped to the schematic representation. Using again the same example as in the previous step, the inputs and the expected observable outputs, which are considered the requirements-based test cases are shown in the following table:

Test Case Number	1	2	3
A	T	T	F
B	T	F	T
Z (Output)	F	T	T

Then, the test cases are represented in the schematic representation. In other words the values are injected at the inputs of the circuit. This help us to view the test cases and the source code in a more convenient format which help us to observe the intermedeate results. Having the intermidate values at the circuit lines assist in the determination of the test cases that do or do not contribute to valid MC/DC results. This way test cases where the output is masked (and then not observable) can be identified, so they do not contribute to achieve MC/DC.

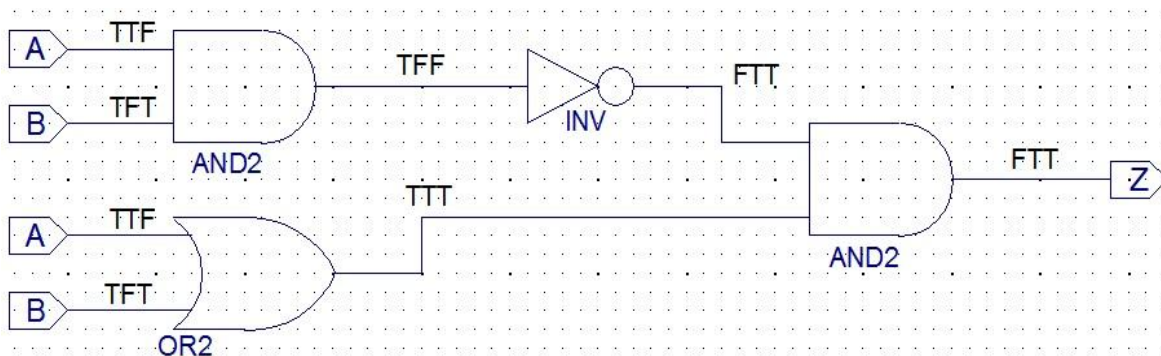


Figure 06: Schematic representation of the $Z = (A \text{ OR } B) \text{ AND } (\text{NOT } (A \text{ AND } B))$ by applying the requirements-based test cases⁴

Step three: Elimination of Masked Tests

The next step is to eliminate the requirements-based tests cases that do not contribute to achieving MC/DC. These tests can be identified by the previous figure. After elimination of the non observable (masked) cases, the remaining test cases can be compared to the minimum test criteria to determine if they are sufficient to meet the MC/DC criteria. One technique used here is to set (specific inputs) “**control inputs**” to a specific value in order to propagate the “**input of interest**” value to the output. This is important in order to propagate a particular input value to the output so it can be observed.

⁴ The figure was created using [Xilinx ISE Design Suite 10.1](#)

For an “AND” gate in order to propagate the input of interest to the output, the other input(s) of the “AND” gate must be true. The principle used is: $W \text{ AND } T(\text{true}) = W$. Using a similar approach for “OR” gate the second principle is the following: $W \text{ OR } F(\text{false}) = W$. For any “OR” gate in order to have a direct path from the input of interest to the output then, the other input(s) to the OR gate must be false.

For the previous example it is easier to use the schematic representation Figure 06 and starting from the output Z and working backwards (towards the inputs) the masked test cases can be identified. The false (F) input in the final “AND” gate (test case 1), masks the corresponding input coming from the “OR” gate. That is because the output of the “or” gate for the previous test case 1 cannot be observed in the output Z. Therefore, test case 1 should be eliminated for the “OR” gate.

Step four: Determination of MC/DC

The next step is to determine whether the remaining valid test cases are sufficient to provide MC/DC. This is done by examining the individual gates; taking each gate into consideration, the valid test cases are compared with the minimum test criteria for the particular gate; this is done for each gate in the circuit. For the first “AND” gate the test combinations needed are: TT, TF, and FT. The TT case is provided by test case 1, TF is provided by test case 2, and test case 3 provides the FT test case. Hence, test cases are sufficient to provide MC/DC for the “AND” gate. Next gate is the “OR” gate; the test cases provider are TF (test case 2) and FT (test case 3), however “OR” gate requires another test case (FF) which is not provider. “NOT” gate has the sufficient test cases required, therefore MC/DC is relevant to the NOT-gate. Finally, the second “AND” gate (just before the output) is checked against the minimum test requirements. Again here, there is a missing test case (TF), the other test cases FT (test case1) and TT (test case 2 and 3) are provider. To sum up, in the previous example $Z = (A \text{ OR } B) \text{ AND } (\text{NOT } (A \text{ AND } B))$ which is equivalent to XOR operation ($A \text{ XOR } B$), the sufficient test cases to provide MC/DC are missing. The next table highlights the missing test cases.

Gate	Valid Test Inputs	Missing Test Cases
AND (first)	TT Case 1 TF Case 2 FT Case 3	None
OR	TF Case 2 FT Case 3	FF
NOT	T Case 1 F Cases 2 or 3	None
AND (second)	TT Cases 2 or 3 FT Case 1	TF

Table 01: Comparison of Minimum Tests with Valid Tests

Step five: Output Confirmation

This is the final step of the evaluation process. This step is included as a reminder to show compliance with the MC/DC criterion as well as the determination of the proper requirements-based test results.

In the example used, the inputs which were used as test cases gave the expected results. It is obvious that this method is labor intensive, but is used by certification authorities and verification analysts for manual confirmation that the test cases and tools used have calculate the correct results. More details and more examples about the five steps process and important factors to consider in selecting and qualifying a structural coverage tool can be found in “A Practical Tutorial on Modified Condition/Decision Coverage” [06].

The **NASA** gate-Level evaluation approach applies in inputs of the schematic representation of the expression, the requirements-based test cases. Then, the masked tests are identified and eliminated from the test suite. Finally, the remaining tests are checked if they fulfill the minimum MC/DC tests for each gate. It is obvious that this method is labor intensive but is still used by certification authorities in order to verify that the software (Level A) used in commercial aviation is erroneous. In the next section, a novel proposal for MC/DC test generation is presented.

6. Experimental evaluation of the difference between FEC and MC/DC

FEC criterion is one of the default coverage criteria for Hardware Description Language (HDLs). HDL is a language for formal description of electronic circuits, especially digital logic. The main differences from the normal programming languages are that HDLs include an explicit notion of time and are the default notations for expressing concurrency (except for behavioral style), which are indispensable in Hardware. According to FAA/NASA SW and CEH Standardization Conference 2005 [34], FEC is equivalent to masking MC/DC. But as it was experimentally proved using some simple expressions this is not completely true for all test cases.

6.1. Experimental setup

The ModelSim HDL⁵ simulator from Mentor Graphics was used in all experiments. The experiments were focused particularly on Focused Expression Coverage (FEC) criterion offered. The experiments have been conducted using ModelSim6.5d simulator under Windows 7 64bits. The input was the expression under investigation (e.g $f := ((A \text{ AND } (B')) \text{ OR } (C \text{ XOR } D))$) and the MC/DC coverage test suite calculated manually using NASA approach (section 5.2.3.). The Verilog⁶ (.v) code illustrated in the next figure consists the generic design module (top.v), which will be customized for each experiment according to its MC/DC test suite and the particular expression. This generic design module was taken and customized by [17].

⁵ ModelSim - Advanced Simulation and Debugging tool by Mentor Graphics: <http://model.com/>

⁶ Verilog is a hardware description language (HDL) used to model electronic systems. It is commonly used in the design, verification, and implementation of digital logic chips at the register transfer level (RTL).

```

module top () ;
reg a, b, c, d, f= 1'b0;
reg [3:0] m;
integer i;
initial
begin
    for ( i = 0; i <= 15; i = i + 1)
    begin
        //MC/DC Test Suite
        if ((i == 3) || (i==5) || (i==6) || (i==8) || (i==12) )
        begin
            m = i ;
            a = m[3];
            b = m[2];
            c = m[1];
            d = m[0];

            f = ( a && (~b) ) + ( c ^ d ) ;    //Expression under investigation
        end
        #1;
    end
end
endmodule

```

Figure 07: Verilog code for the generic design module, which will be customized for each experiment according to its MC/DC test suite and the Expression.

Initially the conditions of the expression are declared (reg a, b, c, d, f = 1'b0;). After the conditions (reg [3:0] m;) which represent the corresponding test vector are declared. Notice that the test vector is represented as an array, in that case 4-bits (3 down to 0) array. The test cases then are not stored as integer values but are stored as the binary equivalence of the integer value (e.g the test case 13 will be represented as 1101 in m[3:0] which is the binary equivalence of 13). The if-statement represents the test suite elements of the expression which as stated in the comment in the figure 07 corresponds to the MC/DC test suite. The (i == 3) corresponds to the test case 3 which represents the binary value of 3 which in a 4-bit vector is 0011; the 0011 comprise the values at the inputs of the expression, for the expression $f = (A \text{ AND } (B')) \text{ OR } (C \text{ XOR } D)$ of the previous example the inputs for the test case 3 (0011) will be A=0, B=0, C=1, D=1. The expression is presented in one high level statement (f). It is obvious that this generic design module can be easily customized according to the particular expression of each experiment and according the particular MC/DC coverage test suite calculated. In order to get the FEC coverage results, initially the verilog code is compiled using: “*vlog -cover ec top.v*” (top.v is the filename) and then it is simulated using “*vsim -coverage -c top -do*” command in the Transcript window (or using the corresponding buttons from the menus). Then the design needs to be simulated (e.g for 1u sec) and then using the Tools->Coverage Report menu, the FEC report can be extracted in the file report.txt figure 14.

According to FAA/NASA software and CEH Standardization Conference 2005 [34], **FEC is equivalent to masking MC/DC**. But as stated in a previous stage there are reasons to believe that this idea is not completely true for all cases so in the next experiments, cases where MC/DC is not equivalent to FEC are explored.

6.2. First Experiment: XOR gate (A xor B)

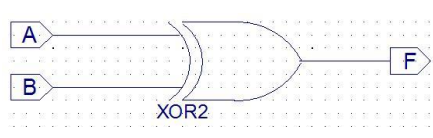
6.2.1 MC/DC test suite calculation

The first expression under investigation is $f := A \oplus B$ ($\oplus = \text{XOR}$).

Initially the MC/DC test suite is calculated. XOR gate is different compared to AND and OR gates with respect the MC/DC because there are multiple minimum test sets. According to [06] for a two-input XOR gate, is required any combination of three test cases out of four possible (exhausting test) test vectors, in order to provide MC/DC. Hence, for a two-input XOR gate the only test suites that meet the MC/DC criterion, are the following:

- test cases 0, 1, and 2 : (corresponding to binary vectors: 00, 01, 10)
- test cases 0, 1, and 3 : (corresponding to binary vectors: 00, 01, 11)
- test cases 0, 2, and 3 : (corresponding to binary vectors: 00, 10, 11)
- test cases 1, 2, and 3 : (corresponding to binary vectors: 01, 10, 11)

It's worth noting that in order to distinguish between an OR and a XOR gate test case 3 must be contained in the test suite (only with test case 3 can be detected if an OR gate which is incorrectly coded for an XOR gate; as shown with **red** colour in the table of the figure 08. Furthermore, the expression $F := A \text{ xor } B$ can be rewritten as $F := (A \text{ or } B) \text{ and not } (A \text{ and } B)$. This implementation of the two-input XOR gate requires four test cases (exhaustive testing), to provide MC/DC [06].



A	B	F(xor)	F(OR)
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1

Figure 08: Schematic representation of the expression $F := (A \text{ XOR } B)$, Expression truth table

6.2.2. ModelSims' FEC coverage results

Now using the same expression (two-input XOR gate) the generic design module was customized using the corresponding XOR expression $f := a \wedge b$ (the symbol \wedge corresponds to XOR in verilog programming language). In addition the MC/DC test suite was replaced with the test suite calculated in the previous step (0, 1, 2) or (0, 1, 3) or (0, 2, 3) or (1, 2, 3) [06]. Using

these test suites the output FEC coverage was 100% which was the expected one. That is an example where the MC/DC coverage is equal with FEC.

However, surprising after further experimentation with the two-input XOR gate it was discovered another test suite which results in 100% FEC coverage. The test suites (0,3) (= $\{0,0\}$, $\{1,1\}$) or the (1,2) (= $\{0,1\}$, $\{1,0\}$) achieve 100% FEC coverage for any N- input XOR gate. For example, test suites with all zeros followed by all ones (0,0,0,0,...,0,1,1,1,...,1) and test suites in the form (1,0,1,0,1,0,...,1,0) or (0,1,0,1,0,1,...,0,1) will also achieve 100%FEC. Those cases are complete violations of the theory of MC/DC because the output of the function has not change its value in both cases so the independent effect of each input on the output which is a requirement on MC/DC is missing. The verilog code after configuration of the generic design module in order to simulate FEC coverage results for the XOR gate is illustrated in the following figure 09.

```

module top () ;
reg a, b, f = 1'b0;
reg [3:0] m;
integer i;
initial begin
    for ( i = 0; i <= 3; i = i + 1)
        begin
            if ((i == 0) || (i==3)) //MC/DC Test Suite
                begin
                    m = i ;
                    a = m[1];
                    b = m[0];

                    f = ( a^b ) ; //Expression under investigation f= a XOR b
                end #1;
        end
    end
endmodule

```

Figure 09: Customization of the generic design module, for the FEC coverage of XOR gate

In the experiment the coverage tool of ModelSim 6.5d was used to get the FEC coverage results for the expression $f := a \wedge b$ as is illustrated in the previous figure 09 . As shown in the code, the test vectors 0 and 3 (if $((i == 0) \parallel (i==3))$) were used. Test vector 0 corresponds to (A=0,B=0 : binary equivalent) for a two input XOR gate, test vector 3 corresponds to (A=1, B=1 : binary equivalent) accordingly. The simulation outputs the following coverage report.

Coverage Report Summary Data by file					
File: top.v (F:=A xor B)					
Enabled Coverage	Active	Hits	Misses	% Covered	
-----	-----	----	-----	-----	
Conditions		6	6	0	100.0
Fec Conditions		8	8	0	100.0
Expressions		8	4	4	50.0
Fec Expressions		8	8	0	100.0

Figure: 10 FEC coverage report for F:=A xor B extracted using ModelSim 6.5d coverage simulator.

As can be seen from the coverage report the two inputs XOR gate is 100% FEC covered using only the test vectors '0' (A=0,B=0) and test vector '3' (A=1,B=1). In both cases the output of the XOR gate is **0-False** due to the truth table of a XOR gate where if the inputs of the gate are equal, then the output is False and if there is an odd number of 1 bits, then the output is **1-True**. Using the test vectors '0' and '3' the output of the expression remains stable in the value 0-False so the independent effect of each input on the output can not be observed. As a result we can conclude that even if this particular expression (two input XOR gate) is 100% FEC covered using the test vectors '0' and '3', with the same tests the same expression is **NOT** 100% MC/DC covered. This is the simplest example of an expression where FEC and MC/DC coverage provide different results under the same test cases.

The same expression, two input XOR gate (F:=A xor B), was simulated with the test vectors '**1**': (A=0,B=1) and '**2**': (A=1,B=0) which both conduct to a 1-True output value. The FEC coverage report shows again 100% FEC covered. This is again a violation of the MC/DC requirements where each input must independently affect the output. To sum up, 100 percent FEC coverage does not imply 100 percent MC/DC coverage; a simple example is when the output does not change value as it was shown in the previous experiment. An image of the experimental results in ModelSim can be found in Appendix 2. Furthermore, this example can be also extended for N-inputs XOR gate using test suites with all zeros (0,0,0,0,...,0) or all ones (1,1,1,1,...,1) and test suites in the form (1,0,1,0,1,...,1,0) or (0,1,0,1,0,1,...,0,1) and will also achieve 100%FEC coverage.

6.3 Second Experiment: Expression F:=(A and not B) or (C xor D)).

6.3.1. MC/DC test suite calculation

To evaluate MC/DC the NASA tutorial [06] steps need to be followed as they are presented in the **section 5.2.3**.

- **First step:** Create a schematic representation of the source code

The schematic representation of the expression F:= (A.B') + (C xor D).is the following:

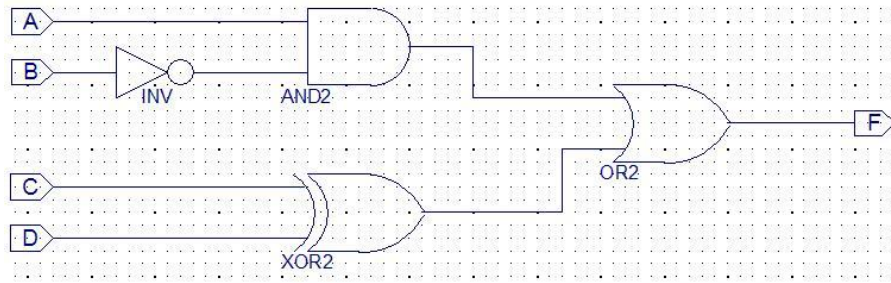


Figure 11: Schematic representation of the expression $F := (A \text{ and not } B) \text{ or } (C \text{ xor } D)$

- **The second step** is to identify the test inputs that will be used. Those inputs are obtained from the requirements-based tests of the software product.

The requirement based test cases for this experiment are illustrated in the following table:

Test Case Number:	0	1	2	3	4
Input A	T	T	F	F	F
Input B	T	F	T	T	T
Input C	F	F	F	F	T
Input D	F	F	F	T	F
Output F	F	T	F	T	T

Table: 02 Requirement based test cases

- **The third step** is the elimination of masked test cases.
An easy way to eliminate the masked test cases is to map the requirements-based test cases into schematic representation. That representation is a convenient format to view the test cases and the source code and it is an easier way to identify the masked test cases.

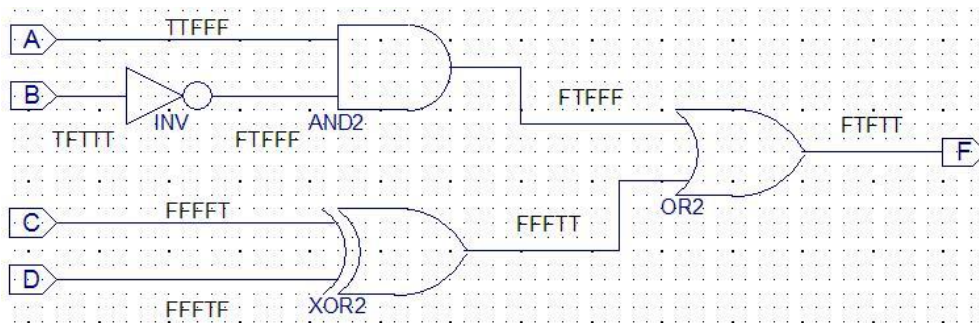


Figure 12: Map the requirement test cases with schematic representation of the expression

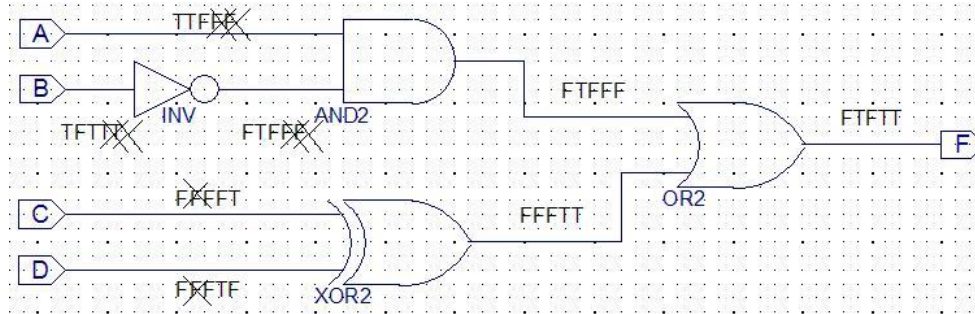


Figure 13: Identification of the masked test cases. Masked test cases are crossed out from the MC/DC test suite

- **The forth step** is the determination of a MC/DC based on the test cases required to test each individual gate. More details can be found on **section 5.2.3**.

Gate	Valid test inputs	Missing test cases
AND	TF Case 0 TT Case 1	FT
NOT	T Case 0 F Case 1	none
XOR	FF Case 0 or 2 FT Case 3 TF Case 4	none
OR	FF Case 0 or 2 TF Case 1 FT Case 3 or 4	none

Table: 03 Comparison of gate requirements with valid tests for this particular example

As shown in Table 03, a test case is missing from the cases required for AND gate; the test case F,T where the AND gate has input A false and gate input B' false. To ensure observability at the output F, the output of C xor D must also be false. Only with false value at the second input of the OR gate the input of interest (A and B) can be propagated to the output. One possible test case for the inputs (ABCD) is (FFTT) which is equivalent to (0011).

- **The final step** is the output confirmation. That step examines the output of the tests to confirm the correct operation of the software just to make sure that the expected result in the test cases match with the expected output based on the gate representation of the expression. If an error is identified then a check to the schematic representation is needed to make sure that everything is correct and also the expression declaration must be checked for typo-mistakes.

The complete MC/DC test suite for the expression $F := (A.B') + (C \text{ xor } D)$ is the following: 0, 1, 3, 4 and (0011). It is worth noting that test case 2 does not contribute to MC/DC. The truth table equivalent test cases are the following: 0:= TTFF(1100) => 12 , 1:= TFFF(1000)=> 8, 3:=

FTFT(0101) => 5, 4:= FTTF(0110) => 6 and the missing test case FFTT(0011) => 3. MC/DC test suite: (3, 5, 6, 8, 12).

6.3.2. ModelSims' FEC coverage results

Now using the same expression ($F := (A.B') + (C \text{ xor } D)$) and the MC/DC test suite calculated in the previous step Model Sims' coverage tool was used in order to show that the coverage results are the same. The customization of the generic design module is presented in the initial figure 07. The MC/DC test suite is (3, 5, 6, 8, 12) shown in red colour and the Expression ($F := (A.B') + (C \text{ xor } D)$) shown in blue colour.

The FEC coverage results are the following:

Coverage Report Summary Data by file					
File: top.v ($F := (A.B') + (C \text{ xor } D)$)					
Enabled Coverage	Active	Hits	Misses	% Covered	
-----	-----	----	-----	-----	
Conditions		6	6	0	100.0
Fec Conditions		10	10	0	100.0
Expressions		12	0	12	0.0
Fec Expressions		8	6	2	75.0

Figure: 14 FEC coverage report extracted using ModelSim 6.5d coverage simulator.

As can be seen the FEC coverage result was 75% using the MC/DC test suite calculated in the previous step, following the NASA MC/DC approach. This is another example where the MC/DC criterion and FEC provide different results and it clearly indicates that it **is wrong to consider MC/DC and FEC equivalent**.

General problems identified in FEC method are the following:

- 100% FEC coverage does not imply 100% MC/DC coverage; a simple example is the first experiment with the two input XOR gate where the output did not change value and still 100% FEC covered.
- FEC mainly focuses on observability requirements and ignores the output change and existence of sufficient independence pairs, which are basic requirements for MC/DC
- The FEC coverage tool sometimes allows incorrect FEC computation (e.g. unexpected begin, end), even if there are some errors in the verilog code; the tool will output wrong coverage results and the user will not notice anything.

7. Novel MC/DC Test generation method: OBSRV

OBSRV is a novel proposal for MC/DC test generation proposed by a Bristol University PhD student which takes as input a circuit and it calculates MC/DC (~FEC) test suite [17]. Adequate MC/DC and good test suite is not calculated easily and that is why OBSRV consists of several quite complex stages. Just like MC/DC criterion which is needed to show independent effect of each input condition to the decision output, OBSRV goal is to achieve the same objective in a simple method. It enforces that all possible values (true and false) for a particular input of concern are observable at the decisions output node in the logic structure. The other input conditions of the gate, are used as control inputs for the propagation of the input of concern to the output. It uses the principles defined previously (NASA MC/DC approach step3, **section 5.2.3. b**) for the basic gates “AND” and “OR”. For “AND” gate $W \text{ AND } T (\text{true}) = W$; in other words the control input should always be true in order to allow the input of interest to be observable at the output. For “OR” gate: $W \text{ OR } F (\text{false}) = W$; which means that the control input should always be false in order to allow the input of concern to be observable at the output. For “XOR” gate when the control input is *false* the gate acts as a **buffer** and when the control input is set to *true* the gate acts as an **inverter**.

7.1. Innovating OBSRV Procedure

The OBSRV MC/DC test generation procedure comprises of five steps whose objective is to deduce a set of test vectors which accomplish MC/DC test suite. The procedure consists of the following steps [17]:

7.1.1 Gate structure representation:

The first step is the representation of the decision statement in gate schematic representation. This step is similar with the first step of the NASA tutorial MC/DC criterion described previously (First step of NASA: Gate-Level Evaluation Approach in **section 5.2.3. b**).

7.1.2 Computation of conditions’ observability test vectors:

The goal in this step is to find a set of test vectors that will let the input condition of interest take all possible values (true I_1 and false I_0) and be observable at the output of the decision. This stage combines the value of the input of concern with the values of control inputs in test vectors. Note that there would be more than one test vector to achieve observability for an input of concern due to the fact that both true and false values of the input of interest must be observed at the output of the decision.

7.1.3 Observability test vectors Ranking:

In this step the test vectors or the test cases highlighted in step two (**section 7.1.2**) are ranked. The ranking defines the descending order of the number of the observable input conditions that the test vector achieved. The greater the number of observable conditions, the higher the ranking of the test vector. The ranking will be used in the selection of the optimized test suite for decision MC/DC.

7.1.4 Observability Matrix Construction

This step provides a visual map for the test vectors and its corresponding observable input conditions' values that will comprise the MC/DC test suite. In this step a table is constructed illustrating the mapping between the input conditions values to be observed, versus the possible test vectors and its ranking highlighted in step 3 (**section 7.1.3**).

7.1.5 MC/DC test suite(s) Determination

Finally, this step specifies the generation of the test suite that will be used in order to achieve the MC/DC criterion. The resulting test cases are the cases which let all possible input conditions values to be observed at the output. This step uses all the work done in previous steps, step three (**section 7.1.3**)-observability matrix and step two (**section 7.1.2**)-ranking of the test vectors, in order to generate the most possible optimized (in terms of the number of test vectors) test suite that form the final possible MC/DC OBSRV test suite for the particular decision statement.

7.2 Experimental evaluation of OBSRV

7.2.1 First example of the OBSRV usage

OBSRV algorithm can be better understood through an example on a real expression. The expression that will be investigated is $F = (A \text{ AND } B) \text{ OR } (C \text{ AND } D)$.

1. The first step of the algorithm is the gate structure representation which is illustrated in the following figure.

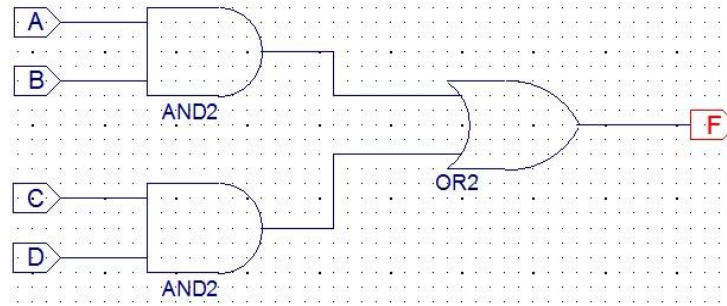


Figure 16: Gate structure representation of the expression $F = (A \text{ AND } B) \text{ OR } (C \text{ AND } D)$ ⁷ [17]

2. The second step is the computation of conditions observability test vectors. In this step the input of interest must take all possible values (true I_1 and false I_0). So it is required to identify the values of the control inputs, which will let the condition of interest to be observed at the output node. For this particular example in order to observe the A_0 , which means the condition of interest A has value=0 (false) the other input of the first **AND** gate must have value=1 (true). The other input of the **AND** gate is B and it is considered the control input that will let the input of concern (A) propagated to the output. So at this point the value of A_0 has been propagated to the **OR** gate. For the OR gate in order to propagate A_0 to the output, its other input must also be 0 (false), so the output of the second AND gate must be false in order to observe A_0 to the output; as a result in order to observe A_0 and similarly A_1 to the output the C and D conditions can have the values ({C=0, D=0} either {C=1, D=0} either {C=0, D=1}) combinations that will produce a false output at the second AND gate. Similar analysis should be done for all inputs conditions taking both values (true I_1 and false I_0).
3. The next step is to **rank** the observability test cases calculated in the previous step. The higher the ranking of the test vector, the higher the numbers of conditions observability it achieves. For the expression used in the example, the test vector {0, 0, 1, 1} (which corresponds to the input values {A=0, B=0, C=1, D=1} (number 3 on the following table)) achieves observability of C_1 and D_1 . Because it achieves observability of only two conditions the rank of the test vector is 2 (Rank=2).
4. The next step is the construction of the **observability matrix**. Observability matrix helps us to visualize the test vectors and their corresponding observable input conditions' values. Also, it illustrates the ranking of all the tests vectors calculated in the previous step and contributes in the MC/DC test suit(s) determination on which is the next step.

⁷ The figure was created using [Xilinx ISE Design Suite 10.1](#)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A0																
A1																
B0																
B1																
C0																
C1																
D0																
D1																
Rank	0	1	1	2	1	2	2	2	1	2	2	2	2	2	2	0

Figure 17: Observability matrix for $F = (A \text{ AND } B) \text{ OR } (C \text{ AND } D)$.⁸

The columns represent all the possible test cases from 0 to 15 (four conditions means $2^4 = 16$ possible values). Test case 0 corresponds to the test case where $A=0, B=0, C=0, D=0$; similarly test case 7 corresponds to the test case where $A=0, B=1, C=1, D=1$. The rows correspond to the conditions of the expression (having in mind that each condition must take both values T and F); A0 represents $A=0$, similarly A1 represents $A=1$ (true). The last row represent the ranking of the test vector and can be easily calculated by adding the number of observable conditions in each column.

5. The last step is the MC/DC test suite(s) determination. This step requires analysis of the observability matrix calculated in the previous step. A set of possible test suites that cover the observability requirements of all inputs need to be identified. It is basically a mix and match exercise on the matrix constructed in the previous step, by taking into account the most possible optimized test suite in terms of the number of test vectors for a particular decision statement. The cases for the test suite are selected by giving priority to the high rank test vectors. This helps OBRSRV to provide determination of an optimized test suite for MC/DC. From the previous table the test suite 3, 6, 9, 12 (corresponding to test vectors $\{0,0,1,1\}, \{0,1,1,0\}, \{1,0,0,1\}, \{1,1,0,0\}$) is an MC/DC test suite calculated using OBSRV. It is worth noting that only N test vectors are required to fulfill MC/DC which is less than N+1 which is the typical conventional minimum MC/DC requirements. The following sample set of test suites achieve the MC/DC requirements: (3,5,10,12), (3, 5, 10, 13), (3, 5, 10, 14), (3, 6, 9, 12), (3, 6, 9, 13), (3, 6, 9, 14), (7,5,10,12), (7, 5, 10, 13), (7, 5, 10, 14), (7, 6, 9, 12), (7, 6, 9, 13), (7, 6, 9, 14), (11,5,10,12), (11, 5, 10, 13), (11, 5, 10, 14), (11, 6, 9, 12), (11, 6, 9, 13), (11, 6, 9, 14) [17].

⁸ If the cell on the observability matrix is covered in blue colour that indicates that the corresponding test case (column number) contribute in the observability of the corresponding condition (row) e.g. A0 is observed by test cases 4, 5 and 6

7.2.2 Another experiment using OBSRV algorithm

Let $F = \text{NOT } (A \text{ AND } B) \text{ AND } (A \text{ OR } C)$

Initially the gate structure representation of the expression is constructed:

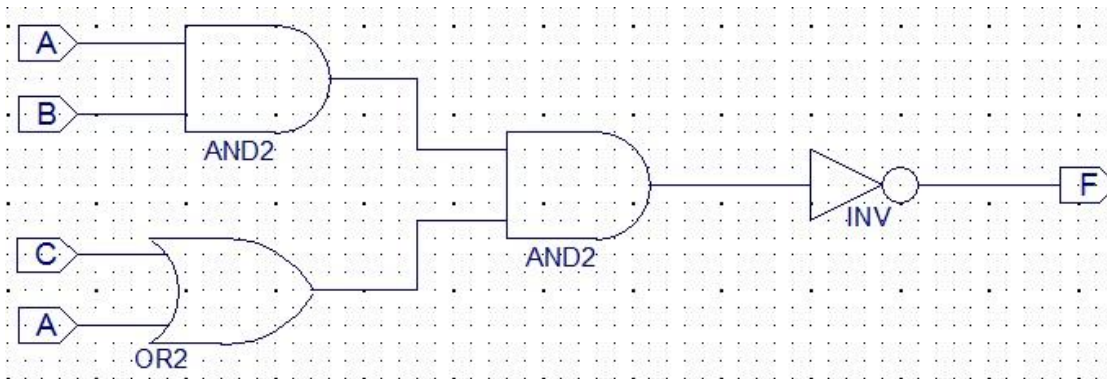


Figure 18: Gate structure representation of the expression $F = \text{NOT } (A \text{ AND } B) \text{ AND } (A \text{ OR } C)$ ⁹

The second step is the computation of conditions observability test vectors. All the inputs of interest must take both values (true and false) and also, control inputs values must be computed; control inputs conduct to the propagation of the value of input of *interest* to the output. Note that the number of inputs may differ from the number of conditions of a decision. For this example, the decision: $\text{NOT } (A \text{ AND } B) \text{ AND } (A \text{ OR } C)$ consist of A, B, and C Boolean variables, contains three inputs (A, B, and C) and four conditions (first A, B, second A and C) because each occurrence of A is considered as a unique condition.

In order to observe A_0 to the output (F), B must be true(1) so that only the condition A affects the output of the first AND gate; if B was false then independent of what the value of A is, the output of first AND gate will be false. Furthermore, the second input of the second AND gate must be true (1); as a result the output of the OR gate must be true (1) $(A+C) = 1$. But since the expression is consisting of a second A condition, there is a second path that A_0 can be propagated to the output. Regarding the second A, C must be false (due to the fact that it is an input to an OR gate; in the case that is true then, independent of what value has the other input of the OR gate, has its output will always be true) and the value of the second input of the second AND gate must also be true (1); the $(A.B)$ must be equal to 1 (true). This type of analysis should be done for A_1 and for all the other conditions.

The next step is the ranking of the test cases calculated in previous step according to the observability they offer. The ranking can be shown better on the observability matrix created in the following step.

⁹ The figure was created using [Xilinx ISE Design Suite 10.1](#)

The observability matrix calculated for the previous expression is the following:

	0	1	2	3	4	5	6	7
A0								
A1								
B0								
B1								
C0								
C1								
Rank	2	1	2	2	2	1	2	2

Figure 19: Observability matrix for $F := \text{NOT}(A \text{ AND } B) \text{ AND } (A \text{ OR } C)$.

The final step is to determine the MC/DC test suite(s). The observability requirements of all inputs need to be covered by choosing the test cases with the higher rank. The highly rank test cases are covering more inputs; that way OBRSRV aims to provide an optimized test suite for MC/DC. Some sample set of test suites that achieve the MC/DC requirements are the following: (0,3,4,5), (0,3,4,7), (2,3,4,6), (2,3,4,7), (0,1,4,6), (0,1,4,7), (2,1,4,6), (2,1,4,7).

8. OBSRV implementation

8.1 Requirements

The main task of this project was to convert a theoretical, pencil and paper procedure into an automated software implementation in order to confirm its correctness as procedure and analysis its complexity. Moreover, an automated procedure offers the opportunity to test straightforwardly and fast the procedure over several test cases (expressions), compared to the pencil and paper procedure which will probably take days in order to validate the same results and with high probability of incorrect results due to a human mistake/error. Furthermore, the automated procedure offers the opportunity to test bigger (in terms of the number of conditions) expressions which are more common in real avionics systems and automate calculate their test suites. Only with a real implementation of the OBSRV procedure it can be evaluated for the quality of its results and for its complexity.

8.2 Specification

The input of the OBSRV implementation is the textual representation of the expression stored in a .txt file. First of all the expression representation is parsed and the internal data structure Expression Graph is created. After construction of the expression Graph the software has a unique schematic representation of the expression. The Graph is the internal data structure which represents the expression in a more convenient, flexible format. Using the Graph the expression could be evaluated by injecting values at the inputs of the Graph and observing its output which represents the expressions' output. In order to create the observability matrix which is a tabular structure ideal for further processing, the implementation uses the truth table of the expression and by injecting into one input the value '**D**' (1/0: corresponds to the correct input value is '1'-True / but the actual erroneous input value is '0'-False) is checking if the '**D**' value is propagated to the output by evaluating the whole truth table of the expression. The D value is injected to every input of the expression (one by one) and for each input the whole truth table is evaluated; if the D value is observed at the output the test vector and the condition in which this is happening is marked (with '1') in the observability matrix [24]. After several iterations for all inputs and the evaluation of the whole truth table (e.g. for an expression with 3 conditions the truth table consist of $2^3=8$ test cases, and for each condition the whole truth table is evaluated, which means $3(\text{conditions}) \cdot 8(\text{test cases}) = 24$ iterations) the observability matrix is constructed. Then the observability matrix is processed in order to calculate the test suite which is the output of the OBSRV implementation.

For the test suite calculation the goal is to cover every condition with a test vector, but the selection order of the test vectors is important in order to end up with an optimal (in terms of

number of test vectors) test suite. For the test vectors selection a greedy algorithm was implemented as the goal was to have an optimal test suite.

The goal of the OBSRV procedure is to find a test suite which achieves the MC/DC requirements for a given expression. The expression is difficult to be interpreted correctly by software compared to humans because maybe a parenthesis or a three input AND gate can be miss-presented as two AND gates. (e.g. the expression $A.B.C$ could be interpreted as one three input AND gate or two two-input AND gate [$(A.B).C$]). That is the reason why it was chosen the expression textual representation as the input of the software; that way there are avoided erroneous interpretations and always the software will have a unique representation for every the expression.

The truth table is used to evaluate completely (all possible input combinations) the expression. The decision to pass the truth table to the software using a .txt file was taken to make the implementation more flexible; maybe not all the test cases will be possible in an expression so they can easily removed from the TruthTable.txt file. In the expression graph part it is important to separate the input, output and gates nodes of the graph. The input nodes are used to inject values into the graph, the output node is used to evaluate the expression and the gates are used to evaluate the sub-expressions and propagate the output value. The observability matrix used is a convenient structure proposes [17] which can be used to efficient calculation of the test suite which is the goal of the whole procedure. An overview of the OBSRV procedure is illustrated in the following figure:

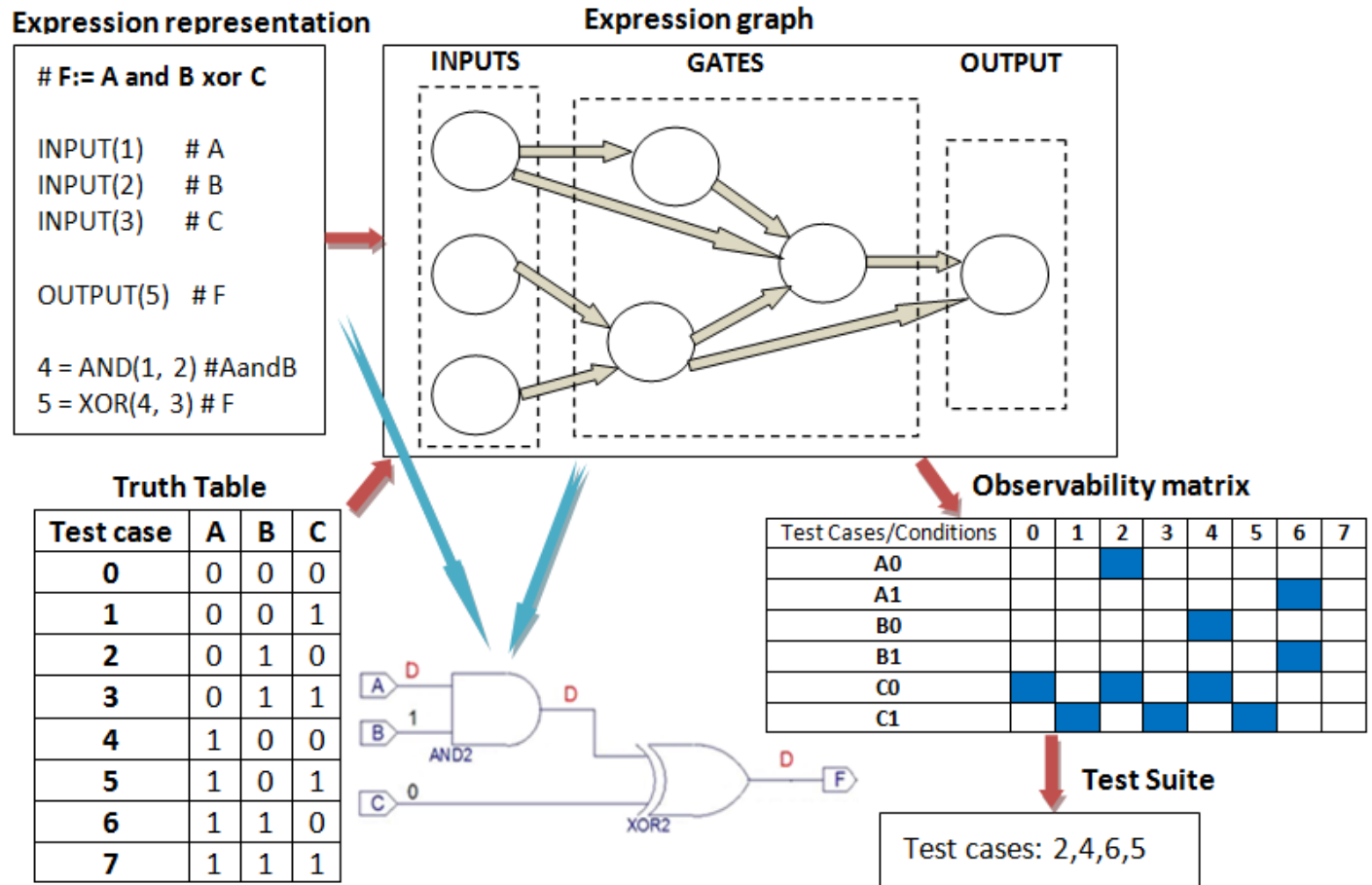


Figure 19: OBSRV procedure overview

8.3 Design

8.3.1 Textual representation of the Expression

The input of OBSRV procedure is the expression (written into a .txt file in a more structural understandable form) and the output of the procedure is the test suite that achieves the MC/DC requirements.

In order to be recognizable and has a more easily understandable structure the expression $F:=((A.B)+(C.D))$ is expressed in the following textual form:

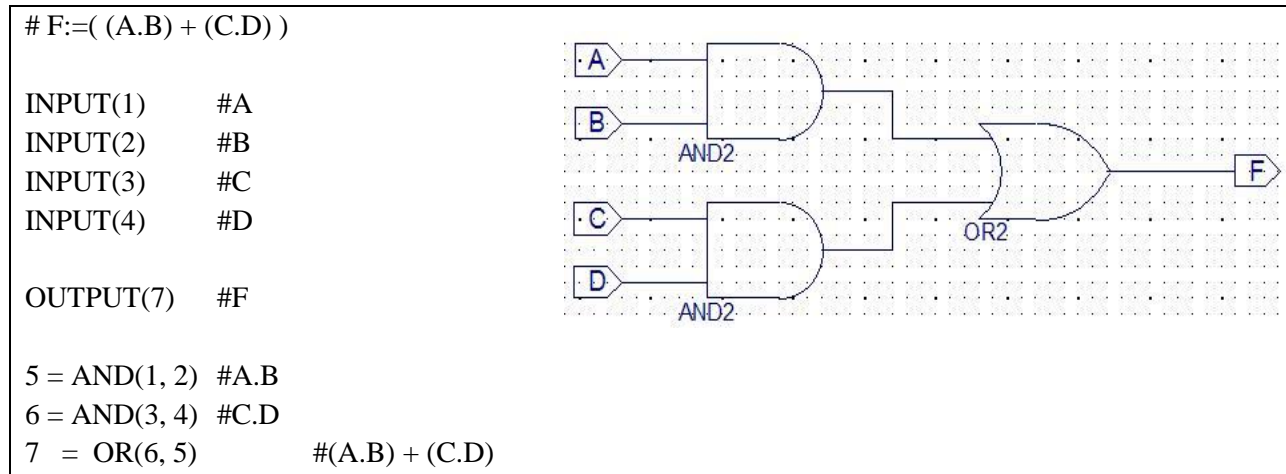


Figure 20: Textual structure form and Schematic representations for the expression $F := ((A.B) + (C.D))$

It normally starts with a comment line (everything after '#' is considered as a comment). Then the inputs are declared, for that particular example ($F := ((A.B) + (C.D))$) the expression has four inputs, that is the reason why there are four lines named (INPUT(1) to INPUT(4)). In that case the first input: INPUT(1) corresponds to the first condition in the expression (which is condition 'A'), the second input INPUT(2) corresponds to the second condition (condition 'B') and so on. The output is declared just after inputs. In that example the output OUTPUT(7) corresponds to the expression output 'F'. The lines following after output comprise the interconnection between inputs, outputs and gates. The $5 = \text{AND}(1, 2)$ represents an AND gate whose inputs are the conditions '1' and '2'; '1' corresponds to the condition A and '2' corresponds to the condition B. The output line of the gate is named/numbered '5'. The usage of the number is just to know where the AND gate output value ('input 1' AND 'input 2') will be propagated. In other words the line:

$5 = \text{AND}(1, 2)$ corresponds to the sub expression $Z = (A.B)$. The next line: $6 = \text{AND}(3, 4)$ with a similar analysis corresponds to the sub expression $Y = (C.D)$. The last line: $7 = \text{OR}(6, 5)$ is the interconnection between the two sub expressions $F := Z + Y$. The previous structure can be better understood by the schematic representation plus the textual structure numbering of the expression which is presented in the following figure:

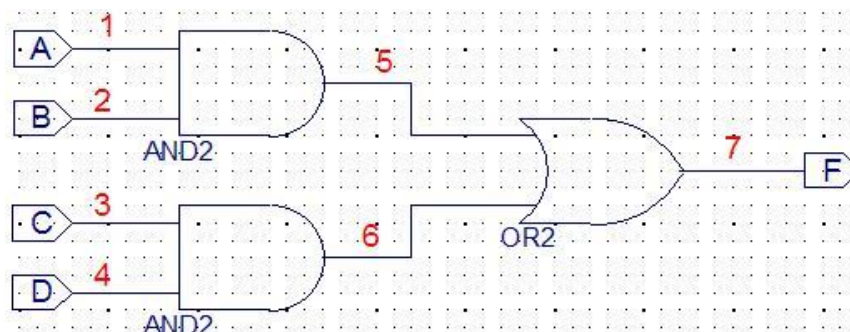


Figure 21: Schematic representation of the expression $F := ((A.B) + (C.D))$ plus the numbering of the lines as described in the textual representation of the expression.

The OBSRV implementation is parsing the textual schematic representation of the expression and the Expression Graph is created. Expression Graph is the internal data structure used to store the schematic representations of the expression; for simplicity the Expression Graph will be called just Graph for the rest of the report. Every input, output and gate is represented by nodes in the internal data structure. The lines/interconnection between input, output and gates comprise the interconnection between nodes. Each node of the Graph has GateIns and also GateOut. The GateIns represent the inputs e.g. of the gate. The GateIns of the first AND gate in the previous figure (figure 22) are 1 and 2; its GateOut is 5. However, 5 is also GateIn for the OR gate which is coming after the two AND gate. The GateIns and GateOuts comprise the way which the nodes of the Graph stay connected (interconnection of the graph) [19].

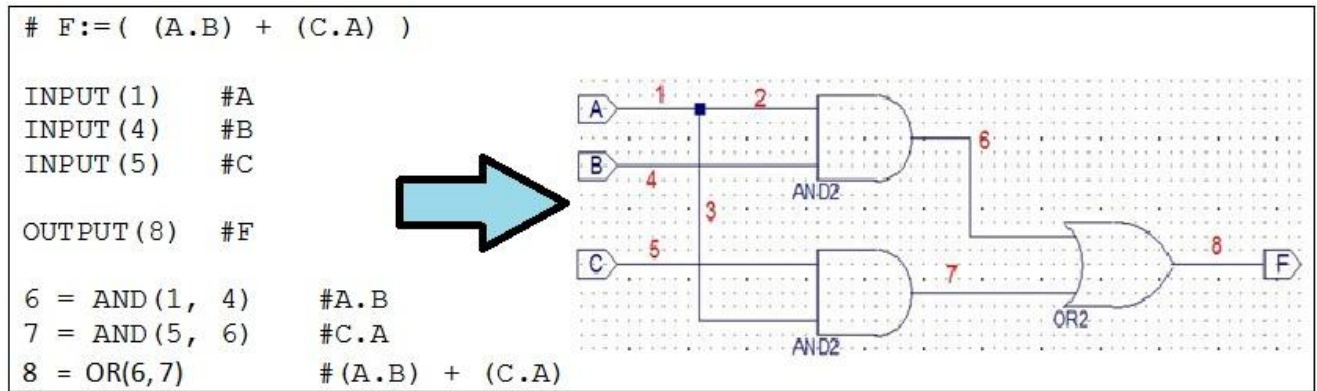


Figure 22: Textual representation converted to schematic representations with internal data structure numbering

It worth noting the way this schematic representation is expressed in the textual form. A special case expression is when one of the inputs or any other interconnection is connected to two different gates (e.g. $F := (A.B) + (C.A)$). When this is the case a branch is created. For example the following textual structure format implies (not explicitly defined) a branch node Figure 23:

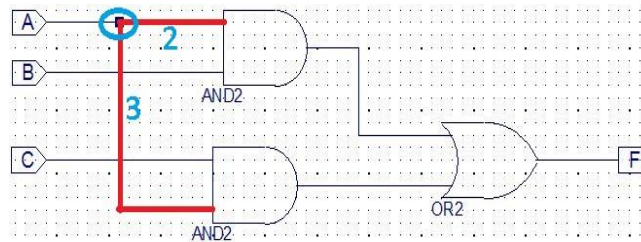


Figure 23: Schematic representation of the expression $F := (A.B) + (C.A)$ with branch noted.

The OBSRV implementation deals with braches by creating two new nodes in the Graph for each branch. Branches act like a buffer, whatever the value is at the input it is propagated to the output without a change/processing. Another important component about the creation of the textual form of the expression (Expression files) is the way the inputs and the gates outputs are numbered. The numbering is important in the traverse of the Graph during processing

(evaluation of the expression). The numbering is assigned in an ascending order such that number 5 will always be after 1,2,3 and 4 only. The reason numbering is important, is that during the traverse (**topological traversal** used as the traversal algorithm) of the Graph in order to evaluate the output of a gate, all the gates and lines before the currently processed gate need to be evaluated, otherwise the values of the inputs of the current gate maybe not available (updated), so the gate it cannot be evaluated.

8.3.2 Topological Traversal

Topological order traversal is ideal for processing the internal data structure created for the implementation of OBSRV. The Graph created represents the structure of the schematic representation of the expression. The only way to deal with the Graph is to inject (insert) values at the inputs and observe the value at the output. Topological order traversal is used in order to propagate the input values throw the interconnection of the Graph and evaluate the expressions' output. Furthermore, in order to evaluate the value at the output of a gate the values at the input of the gate should be present. So the values for the inputs of the gate must be present before the evaluation of the gates output. Topological order traversal requires that in order to visit a new node of a graph, **ALL** the predecessors of the new node must be visited. Furthermore, topological order traversal is used to count the number of possible paths in order to traverse the Graph. Every possible path is considered a possible way to propagate a value from the input to the output which in part of the OBSRV algorithm [20].

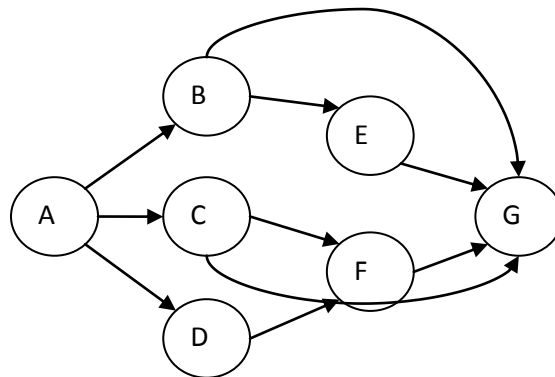


Figure 24: Topological order traversal graph example

For example one way of visiting the nodes of the graph of the previous figure using topological order traversal is: A,B,C,D,E,F,G. Node 'G' cannot be visited until the nodes 'B','E','F','C' are visited. But in order to visit node 'F', node 'C' and 'D' must be visited, and in order to visit node 'C' and 'D' node 'A' must be visited etc. The arrows in the graph indicate that in order to visit a node, its predecessors nodes must be visited first.

One way of implementing topological traversal is by using queues. A node is **Enqueue** (added in the queue) when it has been visited and a node is **Dequeue** when it has been processed. A node is processed if all its immediate successors have been visited [21].

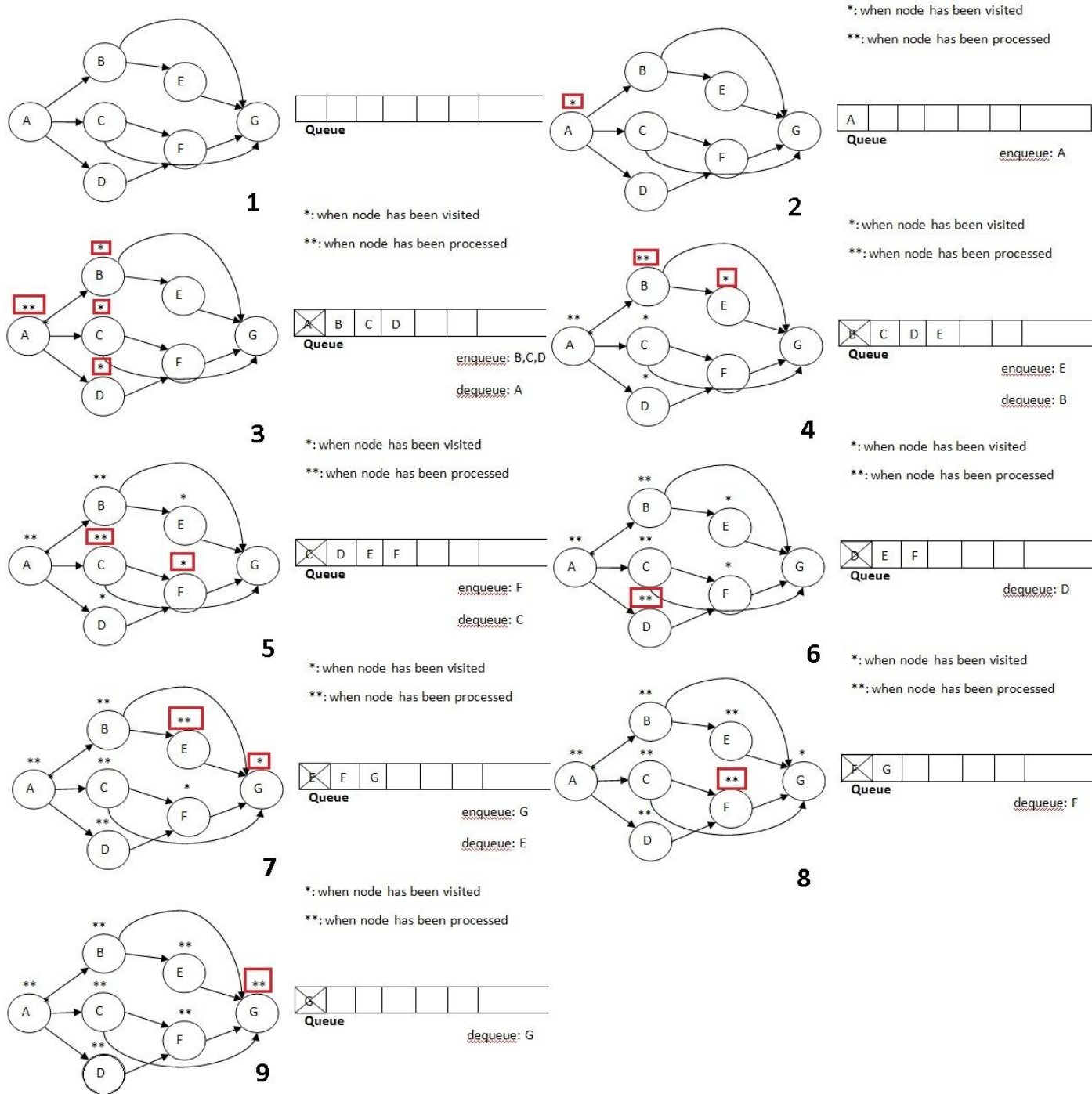


Figure 25: Topological Order Traversal - Algorithm using queue

Pseudo code for the topological order traversal algorithm using queues

Let Graph $G(V, E)$ with

V = Set of Vertices, E = Set of Edges

PI = Primary Inputs: set of nodes with no predecessors and $S(v)$ = set of immediate successors of node v

```

Topological_Traversal_Algorithm(G)
{
    Q : queue;
    v,p : node;
    Init_Queue(Q);           //initialize queue

    FOR every v ∈ V //for every vertice
    {
        IF ( v ∈ PI ) THEN           //if v is an Input
        {
            Enqueue(Q,v);
            mark v as visited;
        }
        ELSE
        { mark v as not-visited; }
    }
    WHILE ( Empty_Queue(Q) == False )
    {
        p = Dequeue(Q);
        FOR every node v ∈ S(p) //for all successors of node v
        {
            IF ( v can be processed ) THEN
            {
                Enqueue(Q,v);
                mark v as visited;
            }
        }
    }
}

```

8.3.3 Expression evaluation using the truth table of the expression

Using the internal data structure representation of the expression test vectors can be evaluated and the output is observed. Values can be injected at the conditions (inputs) of the expression and by using topological order traversal the impact of the inputs can be propagated and observed at the output of the expression. In other words the particular expression can be evaluated; the output of the expression is calculated using topological traversal to evaluate each node value of the graph. For example the expression $F := ((A.B) \text{ XOR } C)$, is composed by the conditions 'A', 'B' and 'C' which are the inputs of the schematic representation of the expression Figure 19. Then by injecting values at the inputs (A,B,C) the internal lines (interconnection) conduct in the propagation of the values in the inputs (conditions) to the gates and then the gates' output are evaluated until the final output of the expression is also evaluated; this is the value of the expression under the specific values at the conditions. Two examples are illustrated in the following figure 26; $F := (0 \text{ and } 1) \text{ xor } 1 = 1$, $F := (1 \text{ and } 1) \text{ xor } 1 = 0$.

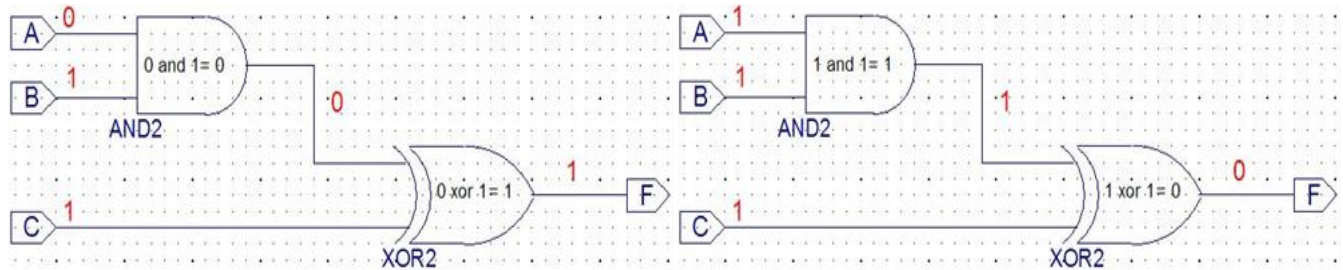


Figure 26: Examples of true value simulation using the schematic representation of the expression $F := ((A.B) \text{ XOR } C)$.

8.3.4 Internal data structure correctness – expression verification

After the implementation of the true value simulator the correctness of the internal data structure which is created can be validated. The validation has been done using the truth table of the expression. For the previous expression for example the truth table is the following Table 04

Condition/Test case	A	B	C	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Table 04: Truth table of the expression $F := ((A.B) \text{ XOR } C)$.

After evaluation the output value of the expression F (red column) has the expected values compared to the values calculated manually. The fact that the whole truth table, which means that all possible input combinations (2^n) have been tested, provide the confidence that the internal data structure and also the topological order traversal algorithm implemented, are representing the expression correctly (substitute the expression). The verification of the implementation have be done using several expressions with different types of gates (NAND, XOR, NOR, XNOR, NOT, AND, OR) and all have produced the expected results.

8.3.5 D-value simulation

After validating the correctness of the correctness of the internal structure, the next step was the creation of a D-value simulation. The idea derives from the paper [33] and **ATPG** (acronym for **A**utomatic **T**est **P**attern **G**eneration) [22] which is the method used to identify a test vector, that when applied to the inputs of a circuit and observe the output, the designer can distinguish

between the correct circuit behavior and the faulty circuit behavior. Basically the response (output) of the device under test after the injection of the test vector is compared with the expected response which is the response of a non erroneous circuit [23].

A fault is detected by a test pattern, when applying the pattern to the design inputs, any logic value observed at the circuit's output differs between the original design and the erroneous design. The ATPG process consists of two phases: *fault activation* and *fault propagation* [31]. Fault activation establishes a value at the design that is opposite of the value produced by the design (normal operation). Fault propagation moves the resulting signal value, or fault effect, forward by sensitizing a path from the fault site to the output. Path sensitization refers to the procedure of finding a path in the circuit that will allow an error to show up at an observable output of a device if it is faulty. For example, in a two-input AND gate, sensitizing the path of one input (input of interest), requires the other input to be set to '1' (control input). Similar case for an OR gate where sensitizing the path of one input (input of interest) requires the other input (control input) of the gate to be set to '0'. This idea derives from NASA: Gate-Level Evaluation Approach – Analysis of the five steps, Step three: Elimination of Masked Tests section **5.2.3. b)** and [06]

One example of ATPG algorithmic method is the D-Algorithm [33]. It refers to the notations “**D**” and “**D'**”. D stands for the true-'1' in a correct operation of the design and for a False-'0' in a faulty one. On the other hand, D', which is the opposite of D, stands for the False-'0' in the correct operation of the design and True-'1' in a faulty design. Thus, propagating a D or D' from the inputs of the expression Graph to its output, simply means applying (inject) a set of inputs in order to make its output exhibit an 'error' if a fault is detected. More specifically, by applying a D value to an input of the Graph (or in our case by injecting a D value to the condition of interest) and by evaluating the output of the Graph looking for observing the D value injected at one of the inputs (input of interest). The values of the other inputs except from the input of interest (input where the D values was injected) comprise the test vector which helps in the propagation of the input of interest value to the output.

To describe D-algebra [21] can best be done by presenting an example. The first example is the expression $F := (A.B) \text{ XOR } C$; its schematic representation is shown in the figure 26. Let's consider the condition A as the condition of interest. Then D (1/0) value is injected at the input A of the internal Graph representation of the expression (figure 26). Then by evaluating all possible input values combination is checked if the D value injected in the input A, is observed at the output. Particularly, the truth table of the expression is evaluated by injecting into the input A the value D when the truth table value of condition A is '0' and D' when the value of A is '1'; the other inputs (Conditions) take all possible input combinations according to truth table (Table 05).

Condition/Test case	A	B	C	F
0	D	0	0	0
1	D	0	1	1
2	D	1	0	D
3	D	1	1	D'
4	D'	0	0	0
5	D'	0	1	1
6	D'	1	0	D'
7	D'	1	1	D

(1)

Condition/Test case	A	B	C	F
0	0	D	0	0
1	0	D	1	1
2	0	D'	0	0
3	0	D'	1	1
4	1	D	0	D
5	1	D	1	D'
6	1	D'	0	D'
7	1	D'	1	D

(2)

Condition/Test case	A	B	C	F
0	0	0	D	D
1	0	0	D'	D'
2	0	1	D	D
3	0	1	D'	D'
4	1	0	D	D
5	1	0	D'	D'
6	1	1	D	D'
7	1	1	D'	D

(3)

Table 05: Truth tables of the expression $F := ((A.B) \text{ XOR } C)$ by injecting D or D' values in the input of interest and observing the output

If the D or D' which is injected at the input A is observed (D or D') at the output F, then the particular test case conducts in the propagation and the observability of the error at the output of the expression. If the value injected at the input of interest is D and also value D is observed at the output (marked with red colour in the table) then the test case in which D is detected, contributes in the observability of the False-'0' value of the particular condition (e.g. A0). Similarly, if the value injected and observed is D' then the particular test case contribute in the observability of the True-'1' value (e.g. A1). For example in the truth table 1 of the Table 05(1) in which the input of interest is the condition A, in test vector 2, D value is injected at A and also observed in the output which means that A0 (A=0) is observed using the test vector 2 (A=0,B=1,C=0). In addition, in test vector 6 (A=1, B=1, C=0) D' value is injected at the input A and also observed at the output F which means that A1 (A=1) can be observed using test vector 6, Table 05 (1) test vectors 2 and 6. [24] Those test cases are marked with blue shade in the observability matrix Figure 28.

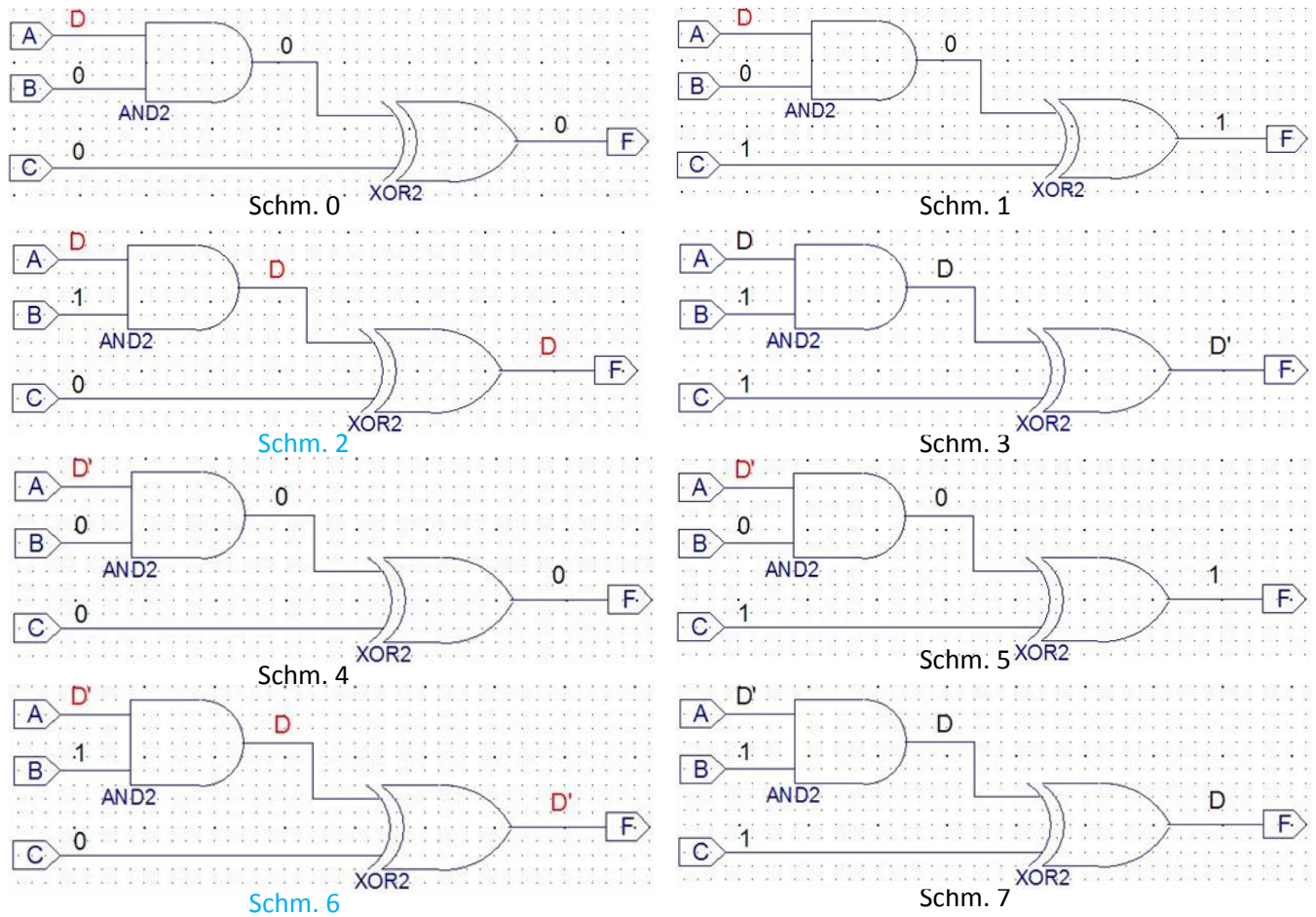


Figure 27: Evaluations of the expression $F := ((A.B) \text{ XOR } C)$ using D-algebra with input of interest the condition (input) A.

The same analysis is done for all the conditions (inputs) of the expression Figure 27 table 2 and table 3 and the results of the analysis is concentrated in a more convenient/understandable matrix called observability matrix. This matrix illustrates the results in terms of the observability requirements for each input condition possible value, true - '1' and false - '0', and the corresponding test vector(s) needed to achieve it. For the expression $F := ((A.B) \text{ XOR } C)$ of the previous example the observability matrix calculated is the following:

Test Case/Conditions	0	1	2	3	4	5	6	7
A0								
A1								
B0								
B1								
C0								
C1								

Figure 28: Observability matrix of the expression $F := ((A.B) \text{ XOR } C)$

Where the **blue shade** on a cell indicates that the particular condition (row) is observed by the particular test case (column).

9. Test suite calculation

9.1 Observability table processing

The final step after the calculation of the observability matrix is the calculation of the test suite. The output of the OBSRV procedure is the test suite which will provide 100% coverage. An example of an observability matrix can be found in the figure 20 . The matrix shows the results in terms of the observability requirements needed for each input condition possible value, (true e.g A0 and false e.g A1), and the corresponding test vector(s) needed to achieve it. The observability matrix shows also the ranking of the test cases to aggregate the OBSRV results in one table. The last row (Rank) represents the number of the observability requirements achieved by every test case. The higher the rank of a certain test case, the higher the number of observability requirements it achieves. For example a rank of 2 for a test case, implies that this test case covers two of the required observability requirements for MC/DC of the decision statement. For example in figure 21, test vector 6 has a rank of 2 as it covers the observability of conditions A0 and D0.

9.1.1. Observability table processing using online algorithm

The goal of the process is that every condition (row) on the observability matrix to be covered by at least one test case (test cases are represented by the columns numbers and correspond to the binary equivalent value at the inputs of the Graph. e.g. test case number (decimal) “10”, converted to binary “1010” which corresponds to A=1, B=0, C=1, D=0). There are two options for processing the observability matrix. The first one is to start from the first test case, detect which conditions it covers and then continue to the second test case and if it covers at least one different (extra) condition from the previous selected test cases then the second (current) test vector is added to the test suite. The same process goes on until all conditions (rows) are covered. The advantages of this method are that it is simple and it has low memory requirements. Basically this method is a configuration of the online algorithm. One example of the usage of this method on the observability matrix for the expression $F := (A.B) + (C.D)$ is shown in the following figures (figure 29: A-F).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A0					1	1	1									
A1												1	1	1		
B0								1	1	1						
B1												1	1	1		
C0		1			1			1								
C1			1				1				1					
D0			1			1				1						
D1				1			1				1					
Rank	0	1	1	2	1	2	2	2	1	2	2	2	2	2	2	0

A: Test suite: 1,

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A0					1	1	1									
A1												1	1	1		
B0								1	1	1						
B1												1	1	1		
C0		1			1			1								
C1			1				1				1					
D0			1			1				1						
D1				1			1				1					
Rank	0	1	1	2	1	2	2	2	1	2	2	2	2	2	2	0

B: Test suite: 1, 2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A0					1	1	1									
A1												1	1	1		
B0								1	1	1						
B1												1	1	1		
C0		1			1			1								
C1			1				1				1					
D0			1			1				1						
D1				1			1				1					
Rank	0	1	1	2	1	2	2	2	1	2	2	2	2	2	2	0

C: Test suite: 1, 2, 3

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A0					1	1	1									
A1												1	1	1		
B0								1	1	1						
B1												1	1	1		
C0		1			1			1			1					
C1			1				1					1				
D0			1			1				1						
D1				1			1				1					
Rank	0	1	1	2	1	2	2	2	1	2	2	2	2	2	2	0

D: Test suite: 1, 2, 3, 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A0					1	1	1									
A1												1	1	1		
B0								1	1	1						
B1												1	1	1		
C0		1			1			1								
C1			1				1				1					
D0			1			1				1						
D1				1			1				1					
Rank	0	1	1	2	1	2	2	2	1	2	2	2	2	2	2	0

E: Test suite: 1, 2, 3, 4, 8

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A0					1	1	1									
A1												1	1	1		
B0								1	1	1						
B1												1	1	1		
C0		1			1			1			1					
C1			1				1					1				
D0			1			1				1						
D1				1			1				1					
Rank	0	1	1	2	1	2	2	2	1	2	2	2	2	2	2	0

F: Test suite: 1, 2, 3, 4, 8, 12

Figures 29: Test cases selection using Online algorithm

In figures 29 the iterations of the online algorithm on the observability matrix of the expression $F := (A.B) + (C.D)$ are illustrated. In Figure 29 A the first test vector 1 (blue circle) is selected and is added to the test suite. All the conditions (rows) that are observed by this test vector are crossed out and also the test vector (column) its self is crossed out (orange horizontal and vertical lines). In Figure 29 B to F the same process takes place until all conditions are covered Figure 29 F. The test suite: 1, 2, 3, 4, 8, 12 is the output of the algorithm. It is comprised by 6 test vectors in order to cover all the conditions which is not the optimal solution. As it will be shown next, using a greedy algorithm solution the conditions can be covered by using only four test vectors.

9.1.2. Observability table processing using greedy algorithm

The other option is a **greedy algorithm** solution. The process starts with the selection of the test vector (column) which covers the biggest number of conditions (rows). Then the conditions which have been covered are removed from the table and the process continues with the selection of the test vector which covers the second in order biggest number of the remaining conditions. By selecting in each step the test vector which covers the greatest number of conditions, it is guaranteed that the output test suite will be optimal as for the number of test vectors needed to cover all the expressions. Some advantages of that algorithm are: its complexity is polynomial in the number of tests and the number of conditions and it offers the optimal test suite (as for the number of test vectors selected). One drawback is that it requires keeping the entire observability matrix in memory which for large expressions the observability matrix could be very large.

From the two previous algorithms, the second one (greedy) was selected. Since in real applications the target is to have a test suite which consist of the smaller number of test vectors. The main idea of the greedy algorithm is simple; since each (row) condition (its true e.g A_0 and false value e.g A_1) must be covered by a test vector, a test vector which covers the greater number of conditions must be selected. Then any other entries that are also covered by the selected condition are removed. That way if condition is covered by a selected test vector it will not affect the next decisions taken in the following steps and as a result it would not (probably) be covered again by another test vector (avoiding overlapping test vectors). This can be implemented by removing all the ones (1^s) across the row of the covered condition and recalculate the **Rank** row values. That way the Rank row will always indicate the correct rank of each test vector by avoiding the test vectors which have already been selected and conditions which have been covered.

Greedy algorithm solution: step by step execution. The input is the observability matrix for the expression $F := (A.B) + (C.D)$ figure 30 and the output (goal) is the test suite identification. The first step is to search the Rank row in order to identify the test vector (column) with the higher Rank value. In this example the higher Rank value is 2, then the algorithm selects the test vector 3 (blue circle in the figure 30)) which has Rank value 2 (note that it could be selected any test vector which has rank value 2). By selecting the test vector 3 the conditions **C1** and **D1** are observed (red circles in the figure 30).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A0				1	1	1										
A1												1	1	1		
B0								1	1	1						
B1												1	1	1		
C0		1			1				1							
C1			1				1				1					
D0			1			1				1						
D1			1				1				1					
Rank	0	1	1	2	1	2	2	2	1	2	2	2	2	2	2	0

Test suite: 3,

Figure 30: Test cases selection using greedy algorithm; selection of test case 3

The next step is to cross-out the column of test vector 3 and the rows of the conditions C1 and D1. In addition the Rank row is re-calculated taking into account the crossed out ones(1^s). The new values of Rank are shown in a subscript number next to the crossed out value in the figure 31. Moreover, the horizontal and vertical orange lines illustrate the columns or rows which are crossed out by the previous steps. The blue horizontal lines indicate the rows which are crossed out by the current step. The third step is to search again the Rank row searching again for the higher value. This value is again 2 and a test vector with Rank value 2 is selected. In that example test vector 5 (blue circle in the figure 31)) is selected. As can be seen in the figure 31 the selection of test vector 5 is crossing out the conditions A0 and C0. The Rank row values are now re-calculated taking into consideration the new rows which have been taken out in the previous steps.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A0					1	1	1									
A1												1	1	1		
B0								1	1	1						
B1												1	1	1		
C0		1			1					1						
C1			1				1				1					
D0			1			1				1						
D1			1				1				1					
Rank	0	1	1	2 ₀	1	2	2	2 ₀	1	2	2	2 ₀	2	2	2	0

Test suite: 3,5

Figure 31: Test cases selection using greedy algorithm; selection of test case 5

In the next step, is selected the test vector with the higher rank again; for that example test vector 10, which covers the conditions B0 and D0. The rows of B0 and D0 are crossed out and the ranking of the test vectors is re-calculated.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A0					±	±	±									
A1													1	1	1	
B0									1	1	1					
B1													1	1	1	
C0		±				±				±						
C1				±				±				±				
D0			1				1				1					
D1				±				±				±				
Rank	0	1 ₀	1	2 ₀	1 ₀	2 ₀	2 ₁	2 ₀	1	2 ₁	2	2 ₀	2	2	2	0

Test suite: 3,5,10

Figure 32: Test cases selection using greedy algorithm; selection of test case 10

As can be seen in the previous figure, only the conditions A1 and B1 left to be covered and can be covered by either test vectors 12, 13 or 14. Test vector 12 is selected and as it is highlighted in the figure 32 there are no left conditions to be covered, so the algorithm ends at this step and prints/writes out the test suite as an output. The final test suite which covers all conditions is: **3, 5, 10, 12**. Notice that using the same expression under the online algorithm the output test suite was: **1, 2, 3, 4, 8, 12** which is comprised by six test vectors instead of four which the greedy algorithm outputs. For larger expression this difference in the number of test vectors is very important because in those cases the difference could be in the magnitude of hundreds or thousands.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A0					±	±	±									
A1													1	1	1	
B0									±	±	±					
B1													1	1	1	
C0		±				±				±						
C1				±				±				±				
D0			±				±									
D1				±				±				±				
Rank	0	1 ₀	1 ₀	2 ₀	1 ₀	2 ₀	2 ₀	2 ₀	1 ₀	2 ₀	2 ₀	2 ₀	2	2	2	0

Test suite: 3,5,10,12

Figure 33: Test cases selection using greedy algorithm; selection of test case 12

This procedure is calculating only **one** test suite which achieves the MC/DC requirements; but there may be other test suites which may cover all the conditions with only four test vectors. The software was extended in that a way that it is calculating **all** the possible (optimal) test suites. Another innovation of this project is the transition of principles used in regression testing in the area of Hardware verification. The greedy algorithm for the selection of test cases is an idea customized in order to meet the demands of this project for an optimal test suite.

9.2 Test Case Selection

Having the observability matrix calculated as input, the next step is the targeted selection of the test cases which will cover all the conditions (rows of the matrix). At that point is good to notice the transition of principles used in the regression testing [25] in the area of hardware verification. Regression testing is an expensive maintenance process that is frequently used and its aim is to (re)verify the correct operation of modified software. One approach for improving regression testing processes is test case prioritization [26], [27], [28], which orders test cases so that the test cases with highest priority, according to some criterion (a specific metric), are executed first. At this project the order of test cases selection is done according to the Rank value of the test case. For the selection phase a greedy algorithm implemented; greedy as for the selections of test cases that cover the larger number of conditions (grater rank value).

In addition in order to calculate all the possible test suites a tree structure was constructed (Figure 34). The tree structure offers the opportunity to keep all possible (greedy) selections of test cases in each Level of the tree, so that all possible options can be explore in the next step of the procedure. Also using a tree structure, backtrack can be used in order to check all steps (test cases selections) taken in order to generate the output test suite.

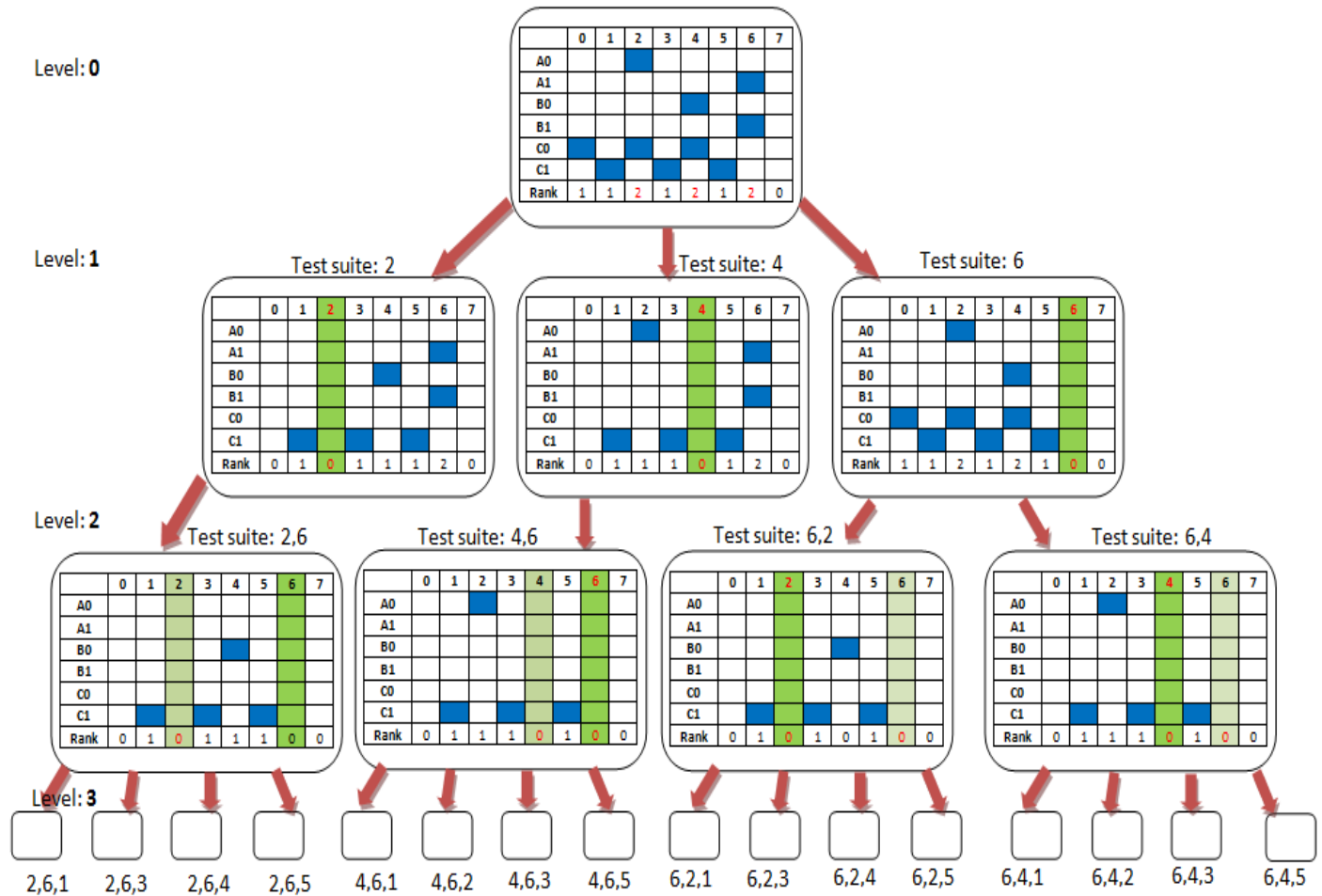


Figure 34: Tree Structure containing all steps executed for the calculation of the Test Suite

The figure 34 illustrates **the tree structure** created for the expression $F := [(A.B) \oplus C]$. In Level 0 the initial observability matrix it is shown; as it is indicated with red colour in rank row, the test cases 2, 4 and 6 all have the (same) higher rank value which is 2. So those test cases must be selected for level 1 of the tree structure. As it is shown after the selection of a test case, the column of the selected test case (highlighted with green colour on the figure 34) and the row(s) of the conditions observed by the particular test case are also removed from the observability matrix so they are not count for the next selections. The same procedure takes place and in the level 1 and the next steps of the tree structure. Test case 2 in Level 1 of the tree the higher rank value is 2 and is achieved in test case 6, therefore test case 6 is the following test case that must be selected and also added to the test suite extending it to test cases 2 and 6. In Level 1 for the test case 6 there are two possible test case selections with the same greater rank value 2, hence they are both selected in the Level 2 of the tree (the red arrows on the figure 34 illustrate the transition between the nodes of the tree). This procedure is executed sequentially until all the conditions (rows) are covered by at least one test case.

After the creation of the tree structure, the software implements a reverse DFS (Depth First Search) traversal procedure to visit the nodes of the tree and generate all the possible test suites for the expression. Starting form a leave node (node which has no children) and traveling backwards until the root (first node - initial node) of the tree is reach, then the node visited consists of the test vectors of the test suite. It is noticeable that usually there are more than one possible test suites which achieve MC/DC for an expression; this is because there are many different routes from a leave to the root of the tree. In addition it is common to have duplicate test suites which must be removed (using *RemoveDuplicatesTestSuites* Function) from the final output of the software.

The function which removes duplicates test suites (*RemoveDuplicatesTestSuites*) takes as parameter a double array *possibleTestSuite*[COL][ROW] and it removes all the duplicates test suites from the table. It begins from the first row of the table and it compares the elements of the test suite (index number 1) with all the other possible test suites in the table. If a duplicate test suite is found in the table it is removed (cross out) form the output-final MC/DC test suite. Duplicate row is also considered any row which has the same test vectors (elements) with the test suite under test and the elements are permuted. The previous steps are repeated for all the possible test suites in the table. The table 04 shows how *RemoveDuplicatesTestSuites* Function operates in the expression $F := A.B \text{ xor } C$; as it is noted there are a lot of duplicates test suites in this example, that need to be deleted (cross out from the table).

Index Number	MC/DC Test Suites:
1	2, 6, 1, 4
2	2, 6, 3, 4
3	2, 6, 4, 1
4	2, 6, 5, 4
5	4, 6, 1, 2
6	4, 6, 2, 1
7	4, 6, 3, 2
8	4, 6, 5, 2
9	6, 2, 1, 4
10	6, 2, 3, 4
11	6, 2, 4, 1
12	6, 2, 5, 4
13	6, 4, 1, 2
14	6, 4, 2, 1
15	6, 4, 3, 2
16	6, 4, 5, 2

Table 06: Remove Duplicates function execution using the expression F:=A.BxorC

10. Design Implementation

This part of the report highlights how the design of OBSRV procedure is converted into C++ code [29],[30]. The implantation consists of two main tasks. The first task is the observability matrix calculation and the second part is the test suite calculation using as input the observability matrix. The first part contains the creation of a graph data structure which corresponds to the schematic representation of the expression; using this graph structure the expression can be evaluated. Moreover, D algebra simulation is used in order to identify the test cases which conduct in the observability of each condition and marked them on the observability matrix. The second task begins with an input the observability matrix calculated in the first task and its goal is to identify the optimal (as for the number of test vectors needed) test suite. The test suite is generated using a greedy algorithm for the selection of test vectors form the observability matrix.

10.1. Observability matrix calculation

```
//data structure declaration
typedef struct
{
    unsigned long int ID;    // node id number
    enum NODETYPE TYPE;     // gate type
    bool PI;                // is the node PI?
    bool PO;                // is the node PO?
    int ConditionOfInterest; // is the node used in OBSRV or NOT
    bool VISITED;           // is the node visited?
    double PATH;            // number of paths
    queue_int NodeINS;       // NodeInputs list
    queue_int NodeOUTS;      // NodeOUTputs list
    int DataValue;          // value of the node (true-false or D)
} NODE;
```

Figure 35: C++ code representing a node of the Expression Graph created to represent the schematic of an expression

The calculation of observability table in OBSRV implementation is based on the internal data structure created (Graph) shown in Figure 19. The expression consists of the conditions which correspond to the inputs/output nodes and the gates which correspond to the gate nodes of the structure. As can be seen in the code fragment Figure 35 for each node it is stored an *ID* number, used to identified each node, the *TYPE* of the node e.g. *AND,OR,XOR,NOT,NAND...*, the *PI* and *PO* fields which are Boolean type are used to store if a node is a **P**rimary **I**ntput (PI) or a **P**rimary **O**utput (PO). The next field is the *ConditionOfInterest* which basically is used as a flag to indicate that a condition (input) is processed (value=1) or not (value=0). Then there is the *VISITED* field which is used during the topological order traverse of the Graph to indicate that a node is visited. The next node is *PATH* which is used as a counter to count the number of possible paths up to the current node. The next two fields are *NodeINS* and *NodeOUTS* which are type Queues in which the ID fields of the predecessors (NodeINS) and successors (NodeOUTS)

nodes from the current node are stored. Last field is the *DataValue* where it is stored the value of the node e.g. “0”=FALSE or “1”=TRUE or “9” = D(1/0) or “8” = D’(0/1).

The OBSRV implementation consists of several functions which make the implementation more flexible on changes, more understandable and easily tested for its correctness. The functions used are highlighted in the following figure. In addition, any function of the process can easily be replaced by a new one if it is found a better (more optimal) way of implementing the same procedure. This gives the opportunity to extent the implementation in any way the designer like or maybe the specifications change; e.g. any optimization (change) in OBSRV procedure can easily built-in in the current procedure:

```
//Functions Declaration
int InitializeDataStructure(NODE GRAPH[MAX]);
int ReadInputFile(char expressionfile[], NODE GRAPH[MAX]);
void PrintGraph(NODE GRAPH[MAX]);
void decoder(int type, int inputs, int outputs, FILE *outfile);
void topological_traversal(NODE GRAPH[MAX]);
int Evaluation(NODE GRAPH[MAX]);
int ReadTruthTableFile(char TruthTable[], NODE GRAPH[MAX]);
int InitializationTable(int Table[ROW][COL]);
int PrintTable(int Table[ROW][COL]);
int RANKING(int OBSRVtable[ROW][COL]);
int insertIndex(int Table[ROW][COL]);

//----- Funtions used for test suite caclulation -----
int copyDoubleArray(int array1[ROW][COL],int array2[ROW][COL]);
int FindMaxCol(int RankIndex, int Table[ROW][COL], int MaxIndexValue[2],int SameRankVectors[COL] );
int IsOtherPossibleTests(int maxRankcol,int Table[ROW][COL],int rankIndex);
int TestCaseSelection(int ObservabilityTable[ROW][COL],int SelectedColumnIndex,int indexRank);
int FilePrintTable(int Table[ROW][COL],int TableID);
int FindRankRowIndexNum(int ObservabilityTable[ROW][COL]);
int PossibleSelections(int ObservabilityTable[ROW][COL]);
int TreeRankLevelSol(int ObservabilityTable[ROW][COL]);
int TestSuiteStactureInitialization(TestCase TestSuiteArray[MAXNODES]);
int copyArray(int table1[COL],int table2[COL]);
int printSingleArray(int table[COL]);
int PrintNodes(TestCase TestSuiteArray[MAXNODES]);
int FilePrintNodes(TestCase TestSuiteArray[MAXNODES]);
int AddElementInArray(int element, int table[COL]);
int PrintTestSuite(TestCase TestSuiteArray[MAXNODES]);
int RemoveDuplicatesTestSuites(int OutTestSuite[COL][ROW]);
int checkDuplicateRow(int TempArray[ROW],int OutTestSuite[ROW][COL], int CurrentRowIndex );
int SearchArray(int Value, int Array[ROW]);
int CompareTwoArrays( int Array1[ROW] ,int Array2[ROW]);
int FilePrintSingleArray(int table[pred], FILE *PrintFile, int OneORmoreSols , int testCase, int TreeIDnum);
```

Figure 36: Declaration of the functions used in the OBSRV implementation

Starting from the *InitializeDataStructure* function which is usage is to loop throw all nodes of the internal data structure GRAPH[MAX] and initialize them with special initialization values. The *ReadInputFile* usage is for the parsing of the expression textual representation file and

creating the Expression Graph (internal data structure). The *PrintGraph* is used for debugging purposes; it writes (prints) the nodes of the Graph in a file (which has the same name as the input file of the expression). Decoder is used to decode the type of gate from the internal representation; e.g. 1 means the gate is an AND, 2 means is an NAND, 3 means is an OR etc. *topological_traversal* deals with the order of traversal of the Graph. Evaluation takes as input the Graph and calculates the data value of all nodes according to their type. For example by checking the data values at the inputs of an AND gate it calculates the output of the gate. *ReadTruthTableFile* function is used to parse the truth table of the expression and injecting the values read at the inputs of the expression. *InitializationTable* is used just to initialize the observability table with special value -1 (indicates not process yet) before the insertion of the calculated values. *PrintTable* is used for demonstration and debugging purpose by printing the observability table. The *RANKING* function is used to loop throw all column (test cases) elements of the observability matrix and count the number observed conditions (marked with value '1' in the table) and inject the count value at the end of the table in a new row, called Rank row. This row indicates the number of conditions observed by the particular test case. The *insertIndex* function is used just to add index numbers in the first row and the first column. The index in the first row indicates the index number of the test case (test vector number) which corresponds to the binary equivalent value on the inputs of the expression. The index in the first column indicates the index of conditions e.g. index: '0' corresponds to A=0, index: '1' corresponds to A=1 , index: '2' correspond to B=0, index: '3' correspond to B=1, index: '4' correspond to C=0 etc. The *copyDoubleArray* function is used by many other functions when a copy of the observability matrix is needed. It gets as parameters two double arrays and it copies all the non -1 elements of the second array (array2[ROW][COL]) to the first array (array1[ROW][COL]).

10.2. Test suite calculation

Having the observability matrix calculated by the previous function the next task is to find an optimal test suite by a targeted selection of test cases.

```

//data tree structure declaration
typedef struct
{
    int ID;
    int testcaseNum;
    int ContainsTestSuite;
    int Table[ROW][COL];
    int predecessors[pred];
} TestCase;

```

Figure 37: C++ code representing a node of the tree structure created in order to calculate all possible (optimal) test suites.

The previous part of code consist the declaration of a node of the tree structure. The first field is the ID which corresponds to the Identification of the node in the tree structure. The *testcaseNum* field corresponds to the number of test case vector it was just selected and processed in the particular node of the tree. The *ContainsTestSuite* field is used as a flag to indicate that from the particular node there are no other possible test cases selections so the expansion of the tree is stop to that node; this also means that if *ContainsTestSuite* has the value 1 then the particular node is a leave node of the tree. The Table[ROW][COL] corresponds to the current observability table stored in the node. The *predecessors[pred]* field contains all the previous nodes (selected test cases) before the current node. Notice that if *ContainsTestSuite* field of a node has the value 1 this imply that the nodes numbers stored in the predecessors[pred] array plus the particular test case number *testcaseNum* correspond to a possible MC/DC test suite.

10.2.1 Brief description of the functionality of the functions used for test suite calculation

The function *FindMaxCol* is loop throw all elements of the rank row and searching for the maximum rank value. If there are more than one test cases with the same maximum rank value then there index is stored in *SameRankVectors[COL]*. If the test suite have not found yet, in the *MaxIndexValue[0]* is stored the index of the max column. In *MaxIndexValue[1]* is stored the value of the maximum rank value. The function returns 1 if the test suite is found otherwise it returns 0. The *IsOtherPossibleTests* loops throw all the elements of the rank row and if there is more than one elements containing the same max rank value it return 1 otherwise it returns 0. The *TestCaseSelection* (**Appendix 1**) function takes as input the SelectedColumnIndex which corresponds to the selected test case and adds it to the *PossibleTestSuites[rows][Columns]* table. This table contains all the possible test suites before removing the duplicates. Then, the function removes all the elements from the selected column on the observability matrix and all the elements from the rows; conditions observed by the particular test case are also removed from the observability matrix. Basically this functions implements what is described in **section 9.1.2**. In addition the ranking of the observability matrix needs to be recalculated after the selection of

the particular test case. The function returns 1 if there exist more possible test cases selection otherwise it returns 0. The *FilePrintTable* function is just looping through all the elements of the observability table and writes (prints) all non -1 element values in the *TreeNodes.txt* file. The *FindRankRowIndexNum* function is used to find the Rank row index number. The *PossibleSelections* function returns 1 if all conditions of the observability matrix have been observed otherwise it returns 0. It loops through all the elements of the Rank row and is counting the number of zeros (0). If the number of zeros is equal to the number of test cases then the function returns 1.

10.2.2. Tree structure construction

The *TestSuiteStructureInitialization* function is used to initialize the fields' values on the nodes of the tree structure. The *TreeRankLevelSol* function (**Appendix 1**) creates the Tree structure with the test suites. It initializes the Tree structure and it calls/uses the functions: *FindMaxCol* , *IsOtherPossibleTests* until all conditions are covered ==>while(TestSuiteReady==1). The function adds nodes to the tree structure according to their order and it also adds their predecessor nodes to the predecessors array, the test case num, the ID etc. After the creation of the tree structure the structure is passed as parameter to the functions: *PrintTestSuite*(TestSuiteArray) and *FilePrintNodes*(TestSuiteArray) in order to print the test suite. The *copyArray* is used to copy all non -1 elements of the table 2 into the table 1. The *printSingleArray* and *FilePrintSingleArray* is used to print/write to file all the non -1 elements of an array and is often used to print the predecessors' values of a node which represent the test suite. The *AddElementInArray* function adds the element passed as parameter into the table of predecessors. It is used to add the new nodes (selected test cases) on the test suite. The *RemoveDuplicatesTestSuites* function loops through all rows of the *OutTestSuite[][]* array and searches row by row for duplicate rows. If a duplicate row is detected it is removed from the *OutTestSuite[][]* so the remaining test suites is the real final test suite. The *checkDuplicateRow* function is used as a sub function of *RemoveDuplicatesTestSuites* function and is implementing the comparison between the rows. The *CompareTwoArrays* function uses as sub-function the *SearchArray* and it implements the comparison between the rows of the *OutTestSuite[][]* array. Duplicate row is also considered a row which has permuted elements compared to the original row.

10.2.3. Pseudo code for the implementation

1. Read input file which contains a textual representation of the expression and create the expression Graph
2. Observability table calculation
 - 2.1. For each condition (**N** - conditions):
 - 2.1.1. Injecting D value in a condition
 - 2.1.2. Read truth table file which includes all the possible test vectors; the truth table size depends on the number of conditions e.g. 4 conditions means $2^4=16$ test cases.
 - 2.1.3. For each test vector on the truth table (2^N – test cases):
 - 2.1.3.1. Evaluate the output of the expression using as input values the values of each test case. Means traverse Graph and observe expressions' output (**K**-nodes)
 - 2.1.3.2. If the D value injected is observed at the output then the condition (row) and the particular test vector (column) is marked at the observability matrix (observability matrix[condition][test vector])
3. Test suite calculation (using greedy algorithm for test case selection) (**W**)
 - 3.1. Ranking of the conditions observed by each test vector
 - 3.2. Print observability table
 - 3.3. Create a tree structure according to the Rank level
 - 3.3.1. Find maxRank value (from rank row)
 - 3.3.2. For all test cases having same Rank value with the maxRank value, add a node on the Level 1 of the tree.
 - 3.3.3. For all nodes added on the previous step repeat steps 3.3.1 and 3.3.2 (with Level=Level+1) until all conditions are covered (maxRank value == 0)
4. Print test suite (Y selected test vectors)
 - 4.1. Use a Depth-first search (**DFS**) tree traversal in order to print all possible test suites

10.3. Design verification

It is obvious that OBSRV implementation is an advance software development project which consists of several functions and expensive conditional calculations. It was implemented in using C++ program language using a lot of the benefits that an object oriented language offers. In addition, C++ applications are fast compared to other higher language level applications and offer the necessary flexibility for covering the changing specifications. The implantation is nearly 3000 lines of C++ code spited among several functions. A structure way of implementing each part of OBSRV algorithm was the only way to accomplish such a complex implementation (consisting of a lot of loop operations). Each new module of the algorithm implemented was adequate tested before and after the integration to the existing implementation.

10.3.1. Expression graph structure validation

The first expensive task is the correct construction of the internal expression Graph. Each node of the graph has to be in the right order and connected to the correct other nodes. Every node which is created and added to the graph is also printed in a .txt file e.g. (AB+CD).txtGRAPH.txt which contains tracing facility of all the nodes created with their interconnections, ID number and data value.

```

NODE:1
NodeINS:No Entries
NodeOUTs:5
TYPE:PI
PI:1
PO:0
VISITED:1
PATHS:0
Data:3

```

Figure 38: Graph node fields for the input node 'A'(ID=1) of the expression. $F:=(AB+CD)$.

The *NodeINS* field corresponds to the inputs of the node while the *NodeOUTs* field corresponds to the output interconnection of the node. The *PI* field takes the value '1' which indicates that the particular node corresponds to an input of the expression. The *VISITED* is a flag used for the topological order traverse of the graph. The *PATHS* field indicates the number of possible paths form the input up to the specific graph node. The *Data* field is initialized with '3' which corresponds to the value of the node which in normal operation will take values '0' or '1'.

10.3.2. Tree structure validation

The second complex task of OBSRV implementation is the processing of the observability matrix. For the particular task a tree structure was created. All the nodes added to the tree structure are also written in the *TreeNodes.txt*

```

ID:0
TestCaseNum:-1
Predecessors:
0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
2 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
4 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
6 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0
7 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0
0 1 1 1 1 1 1 2 1 1 1 2 1 2 1 3 0

```

```

ID:1
TestCaseNum:14
Predecessors:
0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
6 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0
7 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0
0 1 1 1 1 1 1 2 1 1 1 2 1 2 1 0 0

```

ID:3	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
TestCaseNum:6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Predecessors: 14	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ID:59
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	TestCaseNum:9
7 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0	Predecessors: 14 6 10 9 12
0 0 1 0 1 0 1 0 1 0 1 0 1 1 1 1 1 0 0	0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
.....	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
.....	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ID:28	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
TestCaseNum:1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Predecessors: 14 12	4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

Figure 39: Nodes added to the tree structure for the expression $F:=(AB+CD)$.

Each node consists of the following fields: The *ID* field which corresponds to the order of which node is added to the structure. The *TestCaseNum* field which corresponds to the test case selected number and the *Predecessors:* field which corresponds in the test cases selected in previous step before the specific step. The following table corresponds to the observability table of the particular node.

During design stage, along with other programming debugging techniques, those files were used to debug and test the implementation. For any expression used the graph was constructed manually and compared with the generated expression graph in order to detect any missing node or erroneous interconnection. The generated graph is always stored in the EXPRESSION NAME.txtGRAPH.txt file (e.g. (AB+CD).txtGRAPH.txt), where EXPRESSION NAME corresponds to the input expressions' file name (e.g. (AB+CD).txt). The same process was followed for the validation of the generated Tree structure. The processing of the observability matrix has been calculated manually and compared with the nodes written in the *TreeNodes.txt* file. This method was expensive and a lot of time was needed to manually re-calculate the results in order to verify the correct operation of the implementation. Using this method several bugs were detected and we have the confidence that the implementation will operate correctly over any input expression used.

11. Results

The main task of this project was to convert an innovating proposed procedure in an automated software implementation and evaluate its performance and quality of the results. Only with a real implementation, a method can be evaluated completely. As will be presented the test suites calculated manually are also calculated straightforwardly by the implementation. As it is presented in section 10.2 some mistakes in the manual calculations have been identified. That is the reason why a real implementation is important; not only for the calculation speedup that the implementation offers but also for the confidence of its calculation correctness.

11.1. Test suites calculated under experimental expressions

The Table 07 highlights the experimental expressions used and the corresponding test suites calculated. The MC/DC test suites calculated using OBSRV implementation are verified by comparing them with the manual (pencil and paper) results calculated in [17]. For example the (2,6,1,4) which is the first test suite calculated for the expression $F := A \text{ and } B \text{ xor } C$ corresponds to the test vectors (the test vectors consist of three values e.g. "0,1,0", since the expression consist of three conditions) :

- 2: "0,1,0" → meaning: A='0',B='1',C='0'
- 6: "1,1,0" → meaning: A='1',B='1',C='0'
- 1: "0,0,1" → meaning: A='0',B='0',C='1'
- 4: "1,0,0" → meaning: A='1',B='0',C='0'

11.2. Differences identified between pencil and paper method presented in [17] and implementation results.

The existence of OBSRV implementation has contributed in the verification of the test suite results calculated using pencil and paper OBSRV method which has presented in the draft paper [17] and also in **section 6**. As it has been presented OBSRV procedure is a complex and time consuming process so the probability of inattention during the manual calculation of MC/DC test suites is high. As will be shown the OBSRV implementation has contributed in the detection of bugs in the manual calculation of test suites using OBSRV procedure which were presented in the draft paper [17]. The contribution of the OBSRV implementation in the experimental evaluation of OBSRV procedure can be shown in the Table 08 which is a table highlights the extra MC/DC test suites which were not calculated (by mistake) using the pencil and paper OBSRV procedure.

Expression	$F := [(A.B) \text{ xor } C]$	$F := [(A+B)(C+D)] \text{ xor } [(A.B) + (C.D)]$	$F := A+B+C+D$	$F := [((A.B).C) \text{ xor } D]$	$F := A.B.C.D$	$F := A \text{ xor } B$	$F := A.B+C.B$	$F := A+B+C+D$	$F := AB^1+A^1B$	$F := AB^1+C \text{ xor } D$	$F := A.B+A.C+B.C$
Calculated MC/DC Test Suites	(2,6,1,4), (2,6,3,4), (2,6,4,5)	(1,3,12,2), (1,3,12,4), (1,3,12,8), (1,5,10,2), (1,5,10,4), (1,5,10,8), (1,6,9,2), (1,6,9,4), (1,6,9,8), (2,3,12,4), (2,3,12,8), (2,5,10,4), (2,5,10,8), (2,6,9,4), (2,6,9,8), (4,3,12,8), (4,5,10,8), (4,6,9,8)	(1,2,4,8)	(14,6,1,10,12), (14,6,3,10,12), (14,6,5,10,12), (14,6,7,10,12), (14,6,9,10,12), (14,6,10,11,12), (14,6,10,12,13)	(15,7,11,13,14)	(0,1,2)	(3,5,10,12), (3,5,13,10), (3,5,14,10), (3,6,9,12), (3,6,13,9), (3,6,14,9), (3,7,13,6), (9,7,14,6), (9,11,6,12), (9,7,10,5), (13,9,7,6), (13,9,11,6), (13,10,7,5), (13,10,11,5), (14,5,7,10), (14,5,11,10), (14,6,11,9), (11,5,10,12), (12,5,7,10), (12,6,7,9)	(0,1,2,4,8)	(0,3),(1,2)	(0,8,1,2,12), (0,8,1,6,12), (0,8,1,14,12), (0,8,1,15,2), (0,8,2,5,12), (0,8,2,13,12), (0,8,5,6,12), (0,11,0,5,2), (12,11,0,6,1), (12,11,0,13,2), (12,11,0,14,1), (12,11,1,0,2), (12,11,1,3,2), (12,11,3,5,2), (12,11,3,6,1), (12,11,3,13,2), (12,11,3,14,1), (12,11,5,6,0), (12,11,5,14,0), (12,11,6,13,0), (12,11,13,14,0), (0,11,6,15,1), (0,11,13,15,2), (0,11,14,15,1), (0,11,15,1,2), (12,8,1,3,2), (12,8,3,5,2), (12,8,3,6,1), (12,8,3,13,2), (12,8,3,14,1), (12,8,5,14,0), (12,8,6,13,0), (12,8,13,14,0)	(1,2,3,5), (1,3,6,4), (1,3,5,4), (1,3,6,2), (1,5,6,2), (2,3,4,5), (2,6,4,3), (4,5,6,1)

Table 07: MC/DC test suites calculated using the OBSRV implementation under experimental expressions

Expression	$F := A.B+C.B$	$F := A.B+A.C+B.C$	$F := (A.B')+(C \text{ xor } D)$
Extra MC/DC test suites calculated by OBSRV but not calculated (by mistake) manually	(3,7,13,6), (9,7,10,5)	(1,5,6,2), (2,3,4,5), (2,6,4,3), (4,5,6,1)	(0,11,15,2,13), (0,8,1,15,2), (0,11,14,15,1), (0,11,13,15,2), (0,11,6,15,1), (0,11,15,14,1), (1,2,3,11,12), (2,3,5,11,12), (1,3,6,11,12), (12,11,3,13,2), (11,12,14,1,3), (12,11,5,14,0), (12,8,3,14,1), (12,8,3,13,2), (12,8,3,6,1), (12,8,3,5,2), (12,8,1,3,2)

Table 08: MC/DC test suites which were not detected (by mistake) using the OBSRV in a pencil and paper manner [17], but they have been detected using the automated software OBSRV implementation

11.3. Complexity analysis of OBSRV implementation

Having the OBSRV C++ implementation the next step is the evaluation of the process in terms of complexity (iterations needed) and quality of results. The numbers of conditions (**N**) of the expression play a very important role in the number of iterations needed for the fullness of the procedure. The reason why number of conditions is so important can be found in step 2 in the pseudo code procedure shown in **section 10.2.3**. Step 2 is the most time consuming step and most of the iterations of the procedure take place there. First of all in the truth table file (TruthTable.txt) the number of test vectors is in proportion with the number of conditions of the expression. For a condition with three inputs the truth table consist of $2^3=8$ test cases (**j**); for expressions with four conditions $2^4=16$ test cases (**j**) etc. In addition, for each condition a D-value is injected and the whole truth table of the expression is evaluated. Let **N** be the number of conditions of the expression, step 2 needs $N * 2^N * (K \text{ Nodes})$ iterations in order to calculate the observability matrix. The expression Graph consists of **K**-Nodes and in order to evaluate the expression, the Graph must be traverse and observe the output (step 2.1.3.2). Moreover, the number of conditions (**N**) also affects the size of the observability matrix. Observability matrix consist of $2*N$ rows in order to observe each conditions' true (e.g. A1) and false (e.g. A0) value and 2^N columns (same with the number of truth tables' test cases). The overall complexity of step 2 is $O(N * 2^N * K)$.

Step 3.1 – Ranking of the observability matrix complexity is: $O((2*N) * (2^N))$; $2*N$ represents the number of conditions true and false value and 2^N represents the number of test cases on the truth table. The step 3.3 - Test suite calculation complexity is exponential due to the fact that a tree structure is used which growth exponentially depending on the number of conditions (**N**) and the possible test case selection (2^N). The exponential growth of the tree is not considered too bad for the first implementation of the OBSRV procedure. The reason why is because principles used in regression testing are still applicable in expressions where the number of conditions is in the order of thousands, which make them also applicable in the field of hardware verification. Moreover, in critical avionics applications, where MC/DC is applied, basically are used expressions with number of conditions in decimal order (e.g. 3,8,10,13 etc.). As a result the Test cases tree of the implementation will growth in reasonable levels and it will not be huge so the implementation would have the ability to process any expression used in those type of software's. The other steps have negligible complexity compared to step 2 and step 3.3.

12. Conclusions and future plans

This project evaluates MC/DC criterion as a structural coverage metric for hardware verification. The MC/DC criterion has been analyzed and the steps needed for its calculation, its properties and its relationship to other criteria have been presented. MC/DC is an important coverage criterion because as it is illustrated, it provides better coverage of the underline code compared to other coverage techniques like condition/decision, statement coverage etc, and in addition it provides better computational time than multiple condition coverage (exhausting testing). MC/DC is a structural coverage measure and a mandatory requirement for testing avionics software according to **FAA** as described in D0-178B [03]. MC/DC and FEC were considered equivalent according to [34] however, this project has highlighted some cases in which MC/DC is not equal to FEC (**section 6**)

Recently, MC/DC coverage has been introduced into the hardware verification process. However, there is no adequate information about the effectiveness of MC/DC coverage in this context. This has motivated the aims and objectives of this project. A novel proposal for MC/DC test generation method called OBSRV has been presented. The five steps of OBSRV procedure have been described and experimental evaluation of the process has been illustrated (**Section 7**).

Moreover, a first naïve implementation of the OBSRV method has been created and evaluated (**Section 10**). This thesis had a significant contribution in the work of a PhD student translating a novel proposed procedure, into an automated implementation. The evaluation of OBSRV procedure as presented in the draft paper [17] is now evaluated experimentally and some mistakes in the manual procedure have been detected by the software (**section10.2** Table 06). The implementation is very flexible so any changes/improvements on OBSRV procedure can easily be injected in the software. OBSRV implementation transforms and integrates some of the basic principles used in the area of regression testing into the area of hardware verification. The greedy algorithm which was implemented for the test case selection (presented in **section 9.1.2**) derives from the area of regression testing in which it has been successfully used for years under huge expressions; this provides confidence that it can also be used successfully in a different area.

As demonstrated by the implementation of OBSRV, from now on using this implementation, the theoretical results calculated in a pencil and paper manner, can be automatically calculated and as a result, test suites of larger expressions can be evaluated straightforwardly. The OBSRV implementation created during this thesis is an initial step in the creation of an improved and extended application which can be used as a commercial application.

References

- [01] U. S. Department of Transportation, Federal Aviation Administration, guidelines for the approval of software changes in legacy systems using RTCA DO-178B, AC (Advisory Circular) 20-115B, RTCA, Inc. Document RTCA/DO-178B, dated January 11, 1993.
- [02] FAA - Federal Aviation Administration Official Site [Online] (Updated 30 Nov 2009), Available at: <http://www.faa.gov/> [Accessed 10 April 2010].
- [03] RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification, RTCA, Inc., Washington, D. C., December 1992.
- [04] RTCA/DO-248A, Second Annual Report for Clarification of DO-178B "Software Considerations in Airborne Systems and Equipment Certification". RTCA, Inc., Washington, D. C., September 13 2000.
- [05] Ian Sommerville, Software Engineering, Chapter 22: Verification and Validation, 8th Edition, 2006
- [06] Hayhurst, Kelly J., Dan S. Veerhusen, John J. Chilenski, Leanna K. Rierson, A Practical Tutorial on Modified Condition/Decision Coverage, NASA/TM-2001-210876 NASA Langley Technical Report Server, May 2001.
- [07] E. Jee, J. Yoo, S. Cha, Control and data flow testing on function block diagrams, in: Proceedings of the 24th International Conference on Computer Safety, Reliability and Security (SAFECOMP), LNCS 3688, Fredrikstad, Norway, September 28-30, 2005, pp. 67-80.
- [08] Glenford J. Myers (2004). The Art of Software Testing, 2nd edition, 2004.
- [09] P. V. Bhansali (software_der@yahoo.com), PhD, PE, The MC/DC paradox ,ACM SIGSOFT Software Engineering Notes Volume 32 Number 3, May 2007.
- [10] Chilenski , John Joseph, An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion, FAA Tech Center Report DOT/FAA/AR-01/18, April 2001.
- [11] Kelly J. Hayhurst, C. Michael Holloway, Cheryl A. Dorsey, John C. Knight, Nancy G. Leveson, G. Frank McCormick, Jeffrey C. Yang, NASA/TM-1998-207648 Streamlining Software Aspects of Certification: Technical Team Report on the First Industry Workshop (2007).
- [12] Dupuy, Arnaud, Nancy Leveson, An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software, Digital Avionics Systems Conference, October 2000.

- [13] RTCA - Radio Technical Commission for Aeronautics [Online] (Updated 09 April 2010), Available at: <http://www.rtca.org/> [Accessed 13 April 2010].
- [14] [Hayhurst Kelly J. Veerhusen Dan S.](#) A Practical Approach to Modified Condition/Decision Coverage, Technical Report: NASA-2001-20dasc-kjh, NASA Langley Research Center, Hampton, Virginia, Year of Publication: 2001.
- [15] J. J. Chilenski , John Joseph, Steven. P. Miller, Applicability of modified condition/decision coverage to software testing, Software Engineering Journal, September 1994, Vol. 7, No. 5, pp. 193-200.
- [16] FAA Certification Authorities Software Team (CAST) Position PaperCAST-6, Rationale for accepting masking MC/DC in certification projects (2001).
- [17] **Modified Condition Decision Coverage Review:** A Hardware Verification Perspective, draft paper that and will be published shortly.
- [18] Streamlining Software Aspects of Certification: Report on the SSAC Survey, NASA/TM-1999-209519. Langley Research Center Hampton, Virginia 23681-2199, August 1999.
- [19] Parag K. Lala, An Introduction to Logic Circuit Testing, Morgan & Claypool, Oct 2008
- [20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. [Introduction to Algorithms](#). MIT Press and McGraw-Hill. Second Edition, 2001, Section 22.4: Topological sort, pp.549–552.
- [21] Jarnagin, M. P. (1960), *Automatic machine methods of testing PERT networks for consistency*, Technical Memorandum No. K-24/60, Dahlgren, Virginia: U. S. Naval Weapons Laboratory.
- [22] Electronic Design Automation For Integrated Circuits Handbook, by Lavagno, Martin, and Scheffer, A survey of the field, from which the above summary was derived, with permission.
- [23] ATPG: Automatic Test Pattern Generation, <http://www.siliconfareast.com/atpg.htm> Copyright © 2005. SiliconFarEast.com.
- [24] Essentials of electronic testing for digital, memory, and mixed-signal VLSI ... By Michael Lee Bushnell, Vishwani D. Agrawal, Chapter 7.1.5
- [25] Zheng Li, Mark Harman, and Robert M. Hierons, Search Algorithms for Regression Test Case Prioritization, IEEE transactions on software engineering, VOL. 33, NO. 4, APRIL 2007

- [26] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold, Test Case Prioritization: An Empirical Study, Proc. Int'l Conf. Software Maintenance, pp. 178-189, 1999.
- [27] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold, Prioritizing Test Cases for Regression Testing, IEEE transactions on software engineering, vol. 27, no. 10, pp. 929-948, 2001.
- [28] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal, A Study of Effective Regression Testing in Practice, Proc. Eighth Int'l Symp. Software Reliability Eng., pp. 229-237, 1997.
- [29] Richard L Petersen, Introductory C: Pointers, Functions and Files, Second Edition, ACADEMIC PRESS, United States of America, 1997.
- [30] C++ web references used:
<http://www.cplusplus.com/reference/> , <http://www.cppreference.com/wiki/start> ,
http://www-control.eng.cam.ac.uk/~pcr20/C_Manual/booktoc.html,
<http://www.functionx.com/cpp/articles/cfileprocessing.htm>
http://www-control.eng.cam.ac.uk/~pcr20/C_Manual/chap13.html
- [31] S. Neophytou, M. K. Michael, and S. Tragoudas, “Functions for Quality Transition Fault Tests and their Applications in Test Set Enhancement,” IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, 2006.
- [32] Bruce Wile, John Goss and Wolfgang Roesner (IBM) Comprehensive Functional Verification Elsevier, MAY 2005
- [33] J.Paul Roth, Dignosis of automata failures: a calculus and a method, Journal of Research and Development, Volume 10, Issue 4, Pages: 278-291, July 1966
- [34] Santhanam, V. “Can Adaptive Flight Control Software be Certified to DO-178B Leve A?”, NASA and FAA Software and CEH Conference, Norfolk, VA, July 26-28, 2005.

Appendices

Appendix 1: OBSRV implementation source code of basic functions

```
//OBSRV procedure
//Stavros Hadjitheophanous sh9448
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "queue_int.h"
```

```
//Constants
#define MAX 3000 //Number of nodes of the graph
#define MAXLINE 80
#define MAXNODES 506 //Max number of nodes in the Tree Structure
#define ROW 128 //Observability matrix rows
#define COL 128 //Observability matrix columns
#define pred 20 //Max number of predecessor nodes

#define DemoFlag 1 // DemoFlag=1 : Used to show the creation of the observability matrix
// DemoFlag=2 : Used to show the creation of
the tree structure
```

```
int TreeRankLevelSol(int ObservabilityTable[ROW][COL])
{
    int i=0;
    int j=0;
    int ArrayIndex=0;
    int IndexMaxRankCol=-1;
    int MaxRankCol=-1;
    int TestSuiteReady=-1;
    int RankRowIndex=-1;
    int FindMaxIndexValue[2];
    int flagMorePossibleSol=-1;
    int SameRankVectors[COL];
    int TestSuiteFound=-1;
    int TempObservabilityTable[ROW][COL];
    int predecessorIndex=-1;
    int testSuiteIndex=-1;
    int stopLoop=0;
    int predecessorTestCase=-1;
    TestCase TestSuiteArray[MAXNODES];

    FILE *ExpressionFile;

    ExpressionFile = fopen ( "DemoFile.txt" , "a" );
```

```
fprintf(ExpressionFile,"\\nStarting creation of a tree structure\\n");
fprintf(ExpressionFile," .....\\n");
printf("\\nStarting creation of a tree structure\\n");
printf(" .....\\n");
```

```
// first NODE ---> initial observability matrix
TestSuiteStactureInitialization(TestSuiteArray);
InitializationTable(TestSuiteArray[ArrayIndex].Table);
```

```
copyDoubleArray((TestSuiteArray[ArrayIndex].Table),ObservabilityTable);
//copy OBSRV table
TestSuiteArray[ArrayIndex].testcaseNum=-1;
TestSuiteArray[ArrayIndex].ID=ArrayIndex;
TestSuiteArray[ArrayIndex].ContainsTestSuite=0;
```

```
testSuiteIndex=0;
```

```
RankRowIndex=FindRankRowIndexNum(ObservabilityTable); //Find and store
the Rank row index
```

```
while(stopLoop!=1)
{
    //fprintf(ExpressionFile,"-----ArrayIndex:%d-----
\\n",ArrayIndex);
    //printf("-----ArrayIndex:%d
,tastcase%d\\n",ArrayIndex,TestSuiteArray[ArrayIndex].testcaseNum);
```

```
IndexMaxRankCol=-1;
MaxRankCol=-1;
flagMorePossibleSol=-1;
TestSuiteReady=-1;

if (
(TestSuiteArray[ArrayIndex].ContainsTestSuite==1) || ArrayIndex==MAXNODES)
{
    break;
}
TestSuiteReady = FindMaxCol(RankRowIndex,
(TestSuiteArray[ArrayIndex].Table), FindMaxIndexValue , SameRankVectors);
{
    IndexMaxRankCol=FindMaxIndexValue[0];
    MaxRankCol=FindMaxIndexValue[1];
}

if (TestSuiteReady===-2) //Test suite not ready
{
```

```

        flagMorePossibleSol=
IsOtherPossibleTests(MaxRankCol,(TestSuiteArray[ArrayIndex].Table),RankRowIndex);

        if (flagMorePossibleSol==1)
        {
            predecessorIndex= ArrayIndex;
            predecessorTestCase =
TestSuiteArray[ArrayIndex].testCaseNum;

            InitializationTable(TempObservabilityTable);

            copyDoubleArray(TempObservabilityTable,(TestSuiteArray[ArrayIndex].Table));
            //copy OBSRV table

            for (i=0; i<COL; i++)
            {
                if
                (SameRankVectors[i]==-1)
                {

                    //fprintf(ExpressionFile,"-----break no more same rank solutions-----\n");

                    flagMorePossibleSol=0;

                    ArrayIndex++;
                    break;
                }
                else if
                ((SameRankVectors[i]!=-1))
                {

                    testSuiteIndex++;

                    TestSuiteFound=TestCaseSelection(TempObservabilityTable,SameRankVectors[i],R
ankRowIndex); //select first max rank

                    if
                    (TestSuiteFound==0)
                    {

                        InitializationTable(TestSuiteArray[ArrayIndex+testSuiteIndex].Table);

                        copyDoubleArray((TestSuiteArray[ArrayIndex+testSuiteIndex].Table),TempObservabilityTable);
                        //copy OBSRV table

                        TestSuiteArray[ArrayIndex+testSuiteIndex].testCaseNum=SameRankVectors[i];

                        TestSuiteArray[ArrayIndex+testSuiteIndex].ID=ArrayIndex+testSuiteIndex;

```

```

            copyArray(TestSuiteArray[ArrayIndex+testSuiteIndex].predecessors,TestSuiteArray
[predecessorIndex].predecessors);

            if
            (DemoFlag==2)
            {

                FilePrintSingleArray(TestSuiteArray[ArrayIndex+testSuiteIndex].predecessors,
ExpressionFile, 2, predecessorTestCase, ArrayIndex+testSuiteIndex); //demo

                printf("More than one possilbe solution, TestCase: %d ,Tree ID:
%d\n",predecessorTestCase,ArrayIndex+testSuiteIndex); //demo

                printf("Added to:");

                printSingleArray(TestSuiteArray[ArrayIndex+testSuiteIndex].predecessors);

            }

            AddElementInArray(predecessorTestCase,(TestSuiteArray[ArrayIndex+testSuiteInd
ex].predecessors)) ;

            TestSuiteArray[ArrayIndex+testSuiteIndex].ContainsTestSuite=0;
            }
            else
            {

                AddElementInArray(predecessorTestCase,(TestSuiteArray[ArrayIndex].predecessor
s));

                if
                (DemoFlag==2)
                {

                    FilePrintSingleArray(TestSuiteArray[ArrayIndex].predecessors, ExpressionFile,
2,SameRankVectors[i], ArrayIndex+testSuiteIndex); //demo

                    printf("More than one possilbe solution, TestCase: %d ,Tree ID:
%d\n",SameRankVectors[i],ArrayIndex+testSuiteIndex); //demo

                    printf("Added to:");

                    printSingleArray(TestSuiteArray[ArrayIndex].predecessors);

                }

```

```

));
AddElementInArray(SameRankVectors[i],(TestSuiteArray[ArrayIndex].predecessors
));

TestSuiteArray[ArrayIndex].ContainsTestSuite=1;

flagMorePossibleSol=0;

ArrayIndex++;

break;
}

InitializationTable(TempObservabilityTable);

copyDoubleArray(TempObservabilityTable,(TestSuiteArray[ArrayIndex].Table));
//copy OBSRV table
}
//for (i=0; i<COL; i++)
} //if (flagMorePossibleSol==1)
else
{

InitializationTable(TempObservabilityTable);

copyDoubleArray(TempObservabilityTable,(TestSuiteArray[ArrayIndex].Table));
//copy OBSRV table

predecessorIndex=
ArrayIndex;
predecessorTestCase
=TestSuiteArray[ArrayIndex].testCaseNum;

testSuiteIndex++;

TestSuiteFound=TestCaseSelection(TempObservabilityTable,IndexMaxRankCol-
1,RankRowIndex); //select first max rank

if
{
InitializationTable(TestSuiteArray[ArrayIndex+testSuiteIndex].Table);

```

```

copyDoubleArray((TestSuiteArray[ArrayIndex+testSuiteIndex].Table),TempObservabilityTable);
//copy OBSRV table

TestSuiteArray[ArrayIndex+testSuiteIndex].testCaseNum=IndexMaxRankCol-1;

TestSuiteArray[ArrayIndex+testSuiteIndex].ID=ArrayIndex+testSuiteIndex;

copyArray(TestSuiteArray[ArrayIndex+testSuiteIndex].predecessors,TestSuiteArray
[predecessorIndex].predecessors);
if
(DemoFlag==2)
{
FilePrintSingleArray(TestSuiteArray[predecessorIndex].predecessors, ExpressionFile, 1 ,
IndexMaxRankCol-1, ArrayIndex+testSuiteIndex);

printf("Only one possible solution, TestCase: %d ,Tree ID:
%d\n",IndexMaxRankCol-1,ArrayIndex+testSuiteIndex); //demo

printf("Added to:");

printSingleArray(TestSuiteArray[predecessorIndex].predecessors);
}

AddElementInArray(predecessorTestCase,(TestSuiteArray[ArrayIndex+testSuiteIndex].predecessors));

TestSuiteArray[ArrayIndex+testSuiteIndex].ContainsTestSuite=0;
}
else
{

AddElementInArray(predecessorTestCase,(TestSuiteArray[ArrayIndex].predecessors));
//add previous test case
if
(DemoFlag==2)
{
FilePrintSingleArray(TestSuiteArray[ArrayIndex].predecessors, ExpressionFile, 1 ,
IndexMaxRankCol-1, ArrayIndex+testSuiteIndex);

```



```

        printf("Only one possilbe solution, TestCase: %d ,Tree ID:
%d\n",IndexMaxRankCol-1,ArrayIndex+testSuiteIndex);          //demo

        printf("Added to:");

        printSingleArray(TestSuiteArray[ArrayIndex].predecessors);

    }

    AddElementInArray(IndexMaxRankCol-
1,(TestSuiteArray[ArrayIndex].predecessors)) ;          //add current test case

    TestSuiteArray[ArrayIndex].ContainsTestSuite=1;

    }

    ArrayIndex++;

    }
    //if (TestSuiteReady==2)          //Test suite not
ready
    else if (TestSuiteReady==1)
    {
        fprintf(ExpressionFile,"\nTEST SUITE FOUND\n");
        printf("\nTEST SUITE FOUND\n");
        stopLoop=1;
    }

    }          //-----while          loop-----
    //PrintNodes(TestSuiteArray);
    fprintf(ExpressionFile,"Tree structure created! successfully\n\n");
    printf("Tree structure created! successfully\n\n");

    PrintTestSuite(TestSuiteArray ); //Print Test Suie to Screen and also
print TestSuite to TestSuite.txt
    FilePrintNodes(TestSuiteArray);
    fclose(ExpressionFile);
    return(1);
}          //end function
//-----

```

```

int TestCaseSelection(int ObservabilityTable[ROW][COL],int SelectedColumnIndex,int
indexRank)
{

```

```

    int i=0;
    int j=0;
    int rows=0;
    int Columns=0;

    PossibleTestSuites[rows][Columns]=SelectedColumnIndex;
    SelectedColumnIndex=SelectedColumnIndex+1; // index of of observability matrix
is skiped

    //find condition covered by max rank COL //basic
remove (test case)column
    for (i=0;i<indexRank; i++)
    {
        if (ObservabilityTable[i+1][SelectedColumnIndex]==1)
        {
            //Observed[i]=1;

            ObservabilityTable[i+1][SelectedColumnIndex]=0; //remove coverd conditions
            for (j=0;j<COL; j++)
            //remove row
            {
                if
                (ObservabilityTable[i+1][j]!=-1)
                {
                    ObservabilityTable[i+1][j]=0;
                }
            } //for (j=0;j<COL; j++)
        } //if (ObservabilityTable[i][IndexMaxRankCol]==1)
    } //          for (i=0;i<indexRank; i++)

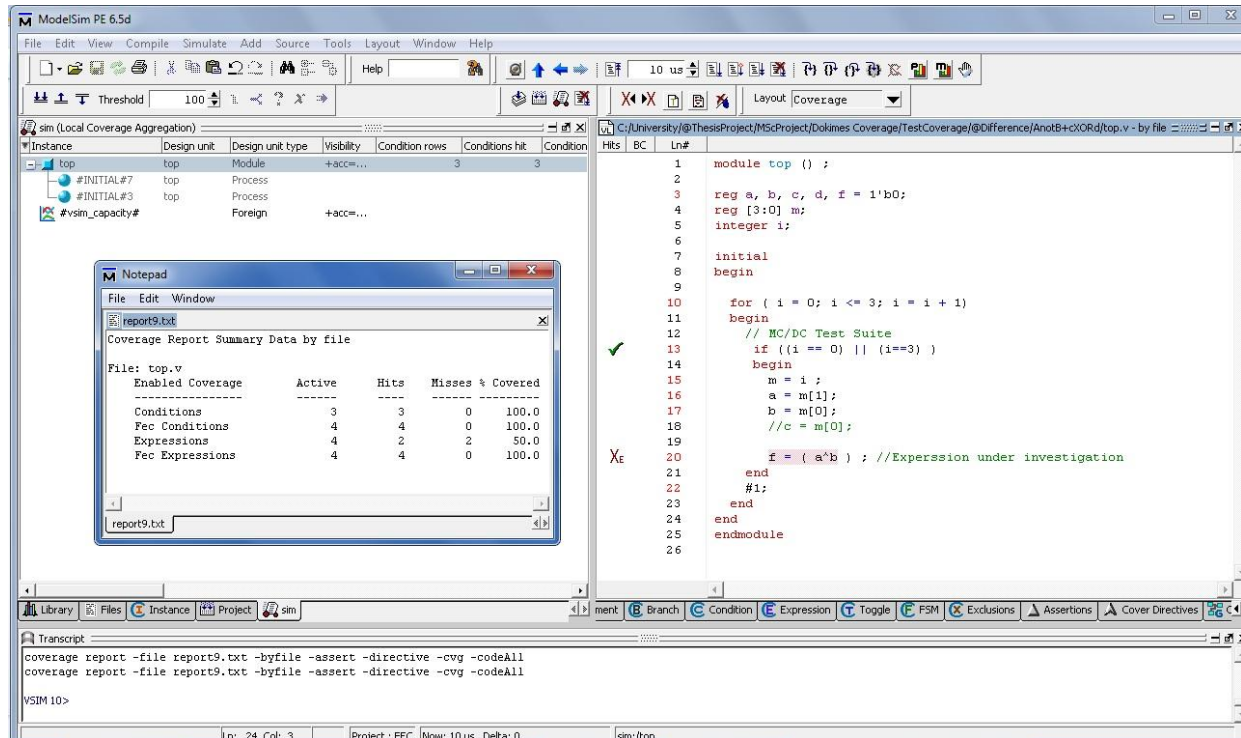
    //remove ranking, used for re-culation
    for(j=0;j<COL; j++)
    {
        ObservabilityTable[indexRank][j+1]=-1;
    }
    RANKING(ObservabilityTable);

    return (PossibleSelections(ObservabilityTable));

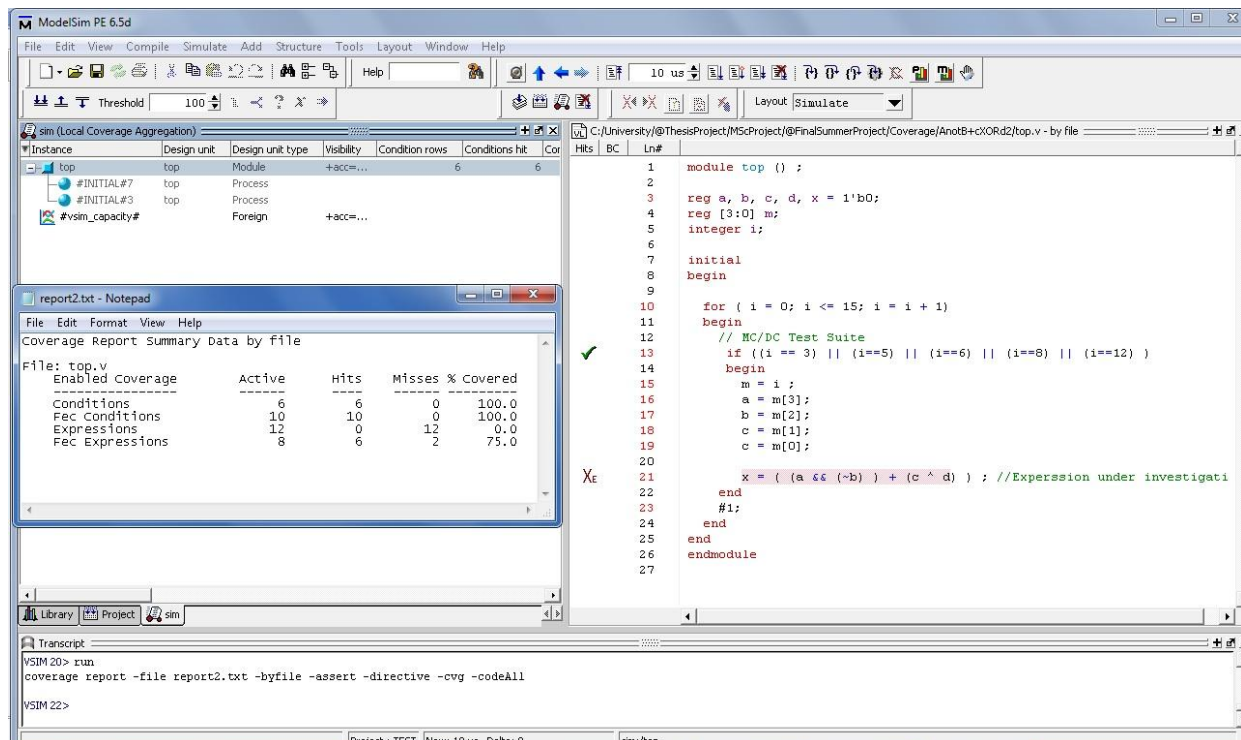
}

```

Appendix 2: Experimental evaluation of the difference between FEC and MC/DC



FEC coverage Report for two input XOR gate using ModelSim 6.5d coverage tool



FEC coverage Report for the expression $F = (A.B) + (C \text{ xor } D)$ using ModelSim 6.5d coverage tool

Appendix 3: Poster

Evaluation of MC/DC as a structural coverage metric for hardware verification



Aims and Context of the Project

The aim of this project is to research evaluate and explore **Modified Condition / Decision Coverage (MC/DC)** as a coverage metric for hardware verification. The main part of the project is the investigation of an innovative procedure for MC/DC test suite generation named **OBSRV** and covert it from a pencil and paper method to an automated software implementation. The robustness and reliability of the algorithm is evaluated through experiments.

Experimental evaluation of the difference between Focused Expression Coverage (FEC) and MC/DC

According to FAA/NASA SW and CEH Standardization Conference 2005, FEC is equivalent to masking MC/DC

- **Experiment 1:** Two input XOR gate $F = A \oplus B$
 - MC/DC test suites: (0, 1, 2), (0, 1, 3), (0, 2, 3) and (1, 2, 3)
 - Test suites (0,3) and (1,2) surprising achieve 100% FEC coverage on ModelSim 6.5d. Generally for an N-input XOR gate, test suites with all zeros followed by all ones (0,0,...,0,1,1,...,1) and test suites in the form (1,0,...,1,0) or (0,1,...,0,1) also achieve 100%FEC.
- ❖ This is complete violation of the theory of MC/DC (the output of the function does not change its value in both cases); the independent effect of each input on the output which is a requirement on MC/DC, is missing.
- **Experiment 2:** Expression $F = ((A \text{ and not } B) \text{ or } (C \text{ xor } D))$
 - MC/DC test suite is (3, 5, 6, 8, 12)
 - The experimental code coverage simulation results indicate **75% FEC** covered.
- ❖ This is another example where the MC/DC criterion and FEC provide different results which indicate that *it is wrong to consider MC/DC and FEC equivalent*.

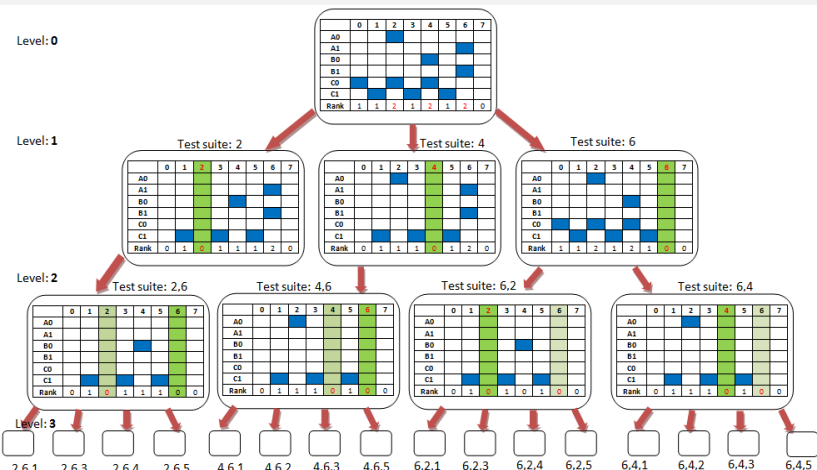
Observability matrix processing

D-value simulation:

- fault activation(injection)
- fault propagation

Expression: $F = [(A.B) \oplus C]$

A greedy algorithm solution for test case selection by creating a tree structure

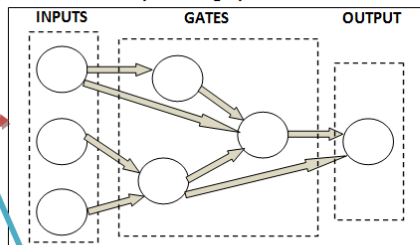


OBSRV Implementation Overview

Expression representation

F:= A and B xor C
INPUT(1) # A
INPUT(2) # B
INPUT(3) # C
OUTPUT(5) # F
4 = AND(1, 2) #AandB
5 = XOR(4, 3) # F

Expression graph



Truth Table

Test case	A	B	C
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Observability matrix

Test Cases/Conditions	0	1	2	3	4	5	6	7
A0								
A1								
B0								
B1								
C0								
C1								

Test Suite

Test cases: 2,4,6,5

Pseudo code for the implementation

1. Read input file with the textual representation of the expression and create the expression Graph
2. Observability matrix calculation:
 - 2.1. For each condition (**N** - conditions):
 - 2.1.1. Injecting D value in a condition (D-value simulation)
 - 2.1.2. Read truth table file which includes all the possible test vectors; the truth table size depends on the number of conditions.
 - 2.1.3. For each test vector on the truth table (**2^N** - test cases):
 - 2.1.3.1. Evaluate the output of the expression using as input values the values of each test case. (traverse Graph) (**K**-nodes)
 - 2.1.3.2. If the D value injected is observed at the output, then the condition (row)along with the particular test vector (column) is marked at the observability matrix (matrix[condition][test])
3. Test suite calculation (using a greedy algorithm for test case selection)
 - 3.1. Ranking of the conditions observed by each test vector
 - 3.2. Print observability table
 - 3.3. Create a tree structure according to the number of the selected test cases
 - 3.3.1. Find maxRank value (form rank row)
 - 3.3.2. For all test cases with the same Rank value as the maxRank value, add a node on the Level of the tree.
 - 3.3.3. For all (new) nodes added on the previous step repeat steps 3.3.1 and 3.3.2 (with Level=Level+1) until all conditions are covered (maxRank value == 0)
4. Print test suite (Y selected test vectors)
 - 4.1. Use a Depth-first search (**DFS**) tree traversal in order to print all possible test suites

Experiment Results

The following table highlights the expressions used and the corresponding test suites calculated

Expression	$F := [(A.B) \text{ xor } C]$	$F := [(A+B)(C+D)] \text{ xor } [(A.B) + (C.D)]$	$F := A+B+C+D$	$F := [(A.B.C) \text{ xor } D]$	$F := A.B.C.D$
Calculated MC/DC	(2,6,1,4), (2,6,3,4), (2,6,4,5)	(1,3,12,2), (1,3,12,4), (1,3,12,8), (1,5,10,2), (1,5,10,4), (1,5,10,8), (1,6,9,2), (1,6,9,4), (1,6,9,8), (2,3,12,4), (2,3,12,8), (2,5,10,4), (2,5,10,8), (2,6,9,4), (2,6,9,8), (4,3,12,8), (4,5,10,8), (4,6,9,8)	(1,2,4,8)	(0,14,6,1,10,12), (0,14,6,3,10,12), (0,14,6,5,10,12), (0,14,6,7,10,12), (0,14,6,9,10,12), (0,14,6,10,11,12), (0,14,6,10,12,13)	(15,7,11,13,14)
Test Suites					

Conclusion

This project evaluates MC/DC as a structural coverage criterion for Hardware verification. The main task of the project is the transition of a theoretical pencil and paper procedure, into a software implementation. MC/DC test suites from expressions from real applications can now be automatically calculated by using **OBSRV** implementation.

