DOT/FAA/AR-01/18

Office of Aviation Research
Washington, D.C. 20591

An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion

April 2001

Final Report

This document is available to the U.S. public through the National Technical Information Service (NTIS), Springfield, Virginia 22161.

U.S. Department of Transportation
Federal Aviation Administration

NOTICE

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
DOT/FAA/AR-01/18		

4. Title and Subtitle	5. Report Date
AN INVESTIGATION OF THREE FORMS OF THE MODIFIED CONDITION DECISION COVERAGE (MCDC) CRITERION	April 2001
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
John Joseph Chilenski	
9. Performing Organization Name and Address	10. Work Unit No. (TRAIS)
Boeing Commercial Airplane Group 2-122 Building, Door South 4 7701 14th Ave. South Seattle, WA 98108	11. Contract or Grant No.
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
U.S. Department of Transportation Federal Aviation Administration Office of Aviation Research Washington, DC 20591	Final Report
	14. Sponsoring Agency Code AIR-100

15. Supplementary Notes

The FAA William J. Hughes Technical Center COTRs were Mr. Peter Saraceni and Mr. Charles Kilgore.

16. Abstract

This report compares three forms of Modified Condition Decision Coverage (MCDC). MCDC is a structural coverage criterion used to assist with the assessment of the adequacy of the requirements-based testing process. This level of coverage is required for Level A software. The purpose of these comparisons is to provide data to enable a rational choice for what form of structural coverage is required for Level A software. This report provides justification why structural coverage, in general, and MCDC in particular, should be part of the software system development process. Definitions for three forms of MCDC are given, along with extensions for relational operators. These three forms of MCDC are compared theoretically and empirically for minimum probability of error detection performance and ease of satisfaction. Conclusions from the data are drawn and limitations of the study methodology are identified.

17. Key Words	18. Distribution Statement
Modified Condition Decision Coverage (MCDC), Software fault injection, Software mutation, Specification, Structural coverage, Testing, Verification	This document is available to the public through the National Technical Information Service (NTIS) Springfield, Virginia 22161.

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	214	

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

DEFINITIONS, ACRONYMNS, ABBREVIATIONS, AND SYMBOLS

Boolean Expression — An expression which evaluates to one of two possible (Boolean) outcomes traditionally known as False and True. These are generally abbreviated as *F* for *False* and *T* for *True*. Sometimes, *0* is used for *False* and *1* for *True*.

Boolean Function — A function that returns a Boolean value (False, True). It may be either user defined or implementation defined. The relational operators operating on non-Booleans are examples of implementation defined Boolean functions.

Boolean Object — Something that holds a Boolean value (False, True). It may be either constant or variable.

Boolean Operator — Operators operating on Booleans. These can be:

Class of Boolean Operators	Boolean Operators
Unary infix operators	NOT
Binary infix normal form operators	AND, OR, XOR
Binary infix short-circuit form operators	AND (AND-THEN), OR (OR-ELSE)
Binary relational operators operating on Booleans	“=”, “≠”, “<”, “≤”, “>”, “≥”

Branchpoint — A point in a computer program at which one of two or more alternative sets of program statements is selected for execution.

Condition — The operand(s) of a Boolean operator (Boolean functions, objects and operators). Generally this refers to the lowest level conditions (i.e., those operands which are not Boolean operators themselves), which are normally the leaves of an expression tree (see the example at the Expression Tree definition). Note that a condition is a Boolean (sub) expression.

Condition Combination — The Boolean values that the conditions in a Boolean expression may assume. The Boolean values that the operands of a Boolean operator may assume. One or both of those operands may be operators themselves, in which case the condition combination requires a certain outcome from the subexpression the operator operand represents.

For the expression

A and (B or C)

one condition combination for the expression is (TFF) in *(A, B,C)*.
For the expression

A and (B or C)

one condition combination for the AND operator is (TF) in *(A,(B or C))*, where *B or C* represents a subexpression, and the OR is an operator operand of the AND.

Condition combinations are generally identified by a (decimal) number generated by interpreting the condition combinations as a binary number (interpreting a False as 0, a True as 1, and a nonexecution due to short-circuit operators as 0). Hence (TFF) is identified as 4:(TFF) (from $(100)_2 = 4$). In some analyses, only the number is used.

Coupled Conditions

Two or more conditions where changing one condition can cause the other condition(s) to change.

Strongly coupled conditions are those conditions where changing one condition always changes the others. For example, in the expression

(X = 0 and A) or (X ≠ 0 and B)

the conditions *X=0* and *X≠0* are strongly coupled. Changing the value of *X* between *0* and *non-0* (in either direction) always changes both conditions.

Weakly coupled conditions are those conditions where changing one condition sometimes (but not always) changes the others. For example, in the expression

X = 0 or X = 1 or X = 3

the conditions *X = 0*, *X = 1* and *X=3* are weakly coupled. Changing the value of *X* from *0* to *2* only changes the first condition, while changing the value of *X* from *0* to *1* changes the first two conditions.

Coupling

The relationship between two or more conditions such that they are not free to assume any possible condition combination between them (i.e., they cannot be varied independently in all circumstances).

Coverage Set

For an expression, a set of condition combinations such that an independence pair for each condition in the expression is present. Condition combinations are generally represented by their decimal number. For the expression

A or (B and C)

one Coverage Set would be:

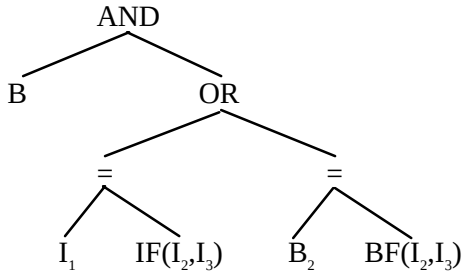
(1,2,3,5) where

(1,5) is the independence pair for *A*
(1,3) is the independence pair for *B*
(2,3) is the independence pair for C

Expression Tree

A graphical (tree) representation of an expression where operators and operands occupy the nodes of the tree, and the edges (arcs) connect operands to their operators. Below is an expression tree for the expression

B_1 and ((I_1 = IF(I_2,I_3)) or (B_2 = BF(I_2,I_3)))



In the above expression tree, where *B* is for *Boolean*, *I* is for *Integer*, and *F* is for *Function*, the conditions are

B_1	{LHS operand of AND}
(I_1 = IF(I_2,I_3)) or (B_2 = BF(I_2,I_3))	{RHS operand of AND}
I_1 = IF(I_2,I_3)	{LHS operand of OR}
B_2 = BF(I_2,I_3)	{RHS operand of OR}
B_2	{LHS operand of "="}
BF(I_2,I_3)	{RHS operand of "="}

Note that conditions can be nested within other conditions.

Independence

A condition possesses independence if there exists two truth vectors such that:

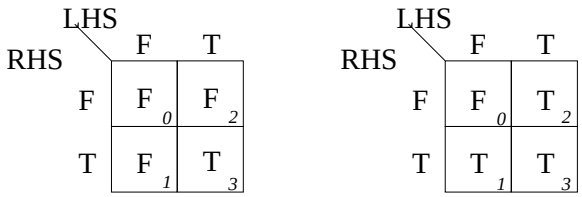
a. both truth vectors have differing results (i.e., one is True and the other is False);

b. the condition of interest has differing values (i.e., in one truth vector the condition is True and in the other it is False); and

c. all other conditions either do not change value or are masked.

Independence Pair — The combination of two truth vectors, one True and one False, that together demonstrate the independence of a condition in a Boolean expression. This condition will change values between the two truth vectors, and all other conditions either will not change or will be masked.

Karnaugh Map — A specific method for formatting a truth table such that adjacent cells in the map differ by one condition difference (T vs. F) in the condition combination. Also known as a KV-Map. Karnaugh maps for the AND and OR operators are presented below:



KV-Map — An abbreviation for a Karnaugh-Veitch map. Also known as a Karnaugh Map.

LHS — Left-Hand Side. The operand which appears on the left-hand side of a binary infix operator.

LRM — Ada Language Reference Manual.

LRU — Line replaceable unit.

MCDC — Modified Condition Decision Coverage.

Masking — The process of setting the RHS (LHS) operand of an operator to a value such that changing the LHS (RHS) operand of that operator does not change the value of the operator.

For an AND operator, masking of the RHS (LHS) can be achieved by holding the LHS (RHS) False. Recall from Boolean algebra that

X AND False = False AND X = False

no matter what the value of *X* is.

For an OR operator, masking of the RHS (LHS) can be achieved by holding the LHS (RHS) True. Recall from Boolean algebra that

X OR True = True OR X = True

no matter what the value of *X* is.

Masking MCDC — A form of MCDC that allows all possible forms of masking to be used to show a condition's independence.

Normal-Form Boolean Operator — A binary Boolean operator (AND, OR, XOR) which evaluates both the LHS and RHS before deciding an outcome.

RHS — Right-Hand Side. The operand which appears on the right-hand side of either a unary or binary infix operator.

Short-Circuit Form Boolean Operator — A binary Boolean operator (AND, OR) which evaluates the LHS first, and then conditionally evaluates the RHS.

For the short-circuit AND, if the LHS is False, then False is returned and the RHS is not evaluated. If the LHS is True, then the RHS is evaluated and that result is returned.

For the short-circuit OR, if the LHS is True, then True is returned and the RHS is not evaluated. If the LHS is False, then the RHS is evaluated and that result is returned.

Truth Vector — For a Boolean function or expression, a specific condition combination and the function/expression result. For example, for the expression

A and (B or C)

one truth vector is *4:F-tff*, where the condition combination *4:(tff)* applied to the expression results in an *F* (False).

Unique Cause MCDC — A form of MCDC which allows for masking to be used only in the case of coupled conditions to show a condition's independence. Otherwise, only the condition of interest is allowed to change between the two truth vectors of the independence pair. The condition's change is (generally) the unique cause of the change in the expression's outcome, hence the name.

EXECUTIVE SUMMARY

This report compares three forms of the Modified Condition Decision Coverage (MCDC) criterion. MCDC is a structural coverage criterion used in the "Software Considerations in Airborne Systems and Equipment Certification Document," to assist with the assessment of the adequacy of the requirements-based testing process. This level of coverage is required for level A software per DO-178B. The purpose of these comparisons is to provide data to support a rational choice for what form of structural coverage to require for level A software.

The three forms of MCDC investigated include:

- Unique-Cause MCDC: This form is the one defined in "Software Considerations in Airborne Systems and Equipment Certification Document."

- Unique-Cause+Masking MCDC: This form is the one suggested in "Applicability of Modified Condition Decision Coverage to Software Testing," that addresses the coupling problem. This form is included because it has been used in airplane certifications and as a reference by automated coverage analysis tool vendors.

- Masking MCDC: This is the weakest possible form of MCDC in that it allows for the independence of a condition to be determined strictly by the Boolean Difference Function discussed in *Switching and Finite Automata Theory*. The other forms place additional constraints on an independence pair beyond those imposed by the Boolean Difference Function. This form is included because if it is found acceptable, then all other stronger forms will also be acceptable.

The comparisons performed in this study address the following four major questions:

1. Why is there a need for structural coverage in general?

   It is well known that all forms of verification could miss important features of the system being implemented. Examples in this report demonstrate requirements-based verification missing important features of the implementation. Structural coverage in the software system development process is a check and balance on requirements-based verification. Structural coverage helps ensure that the requirements-based verification process has paid attention to certain features of the implementation.

2. Why is there a need for MCDC in particular?

   This study shows that MCDC is a form of structural coverage providing equivalence class and boundary value coverage of the implementation. MCDC provides this coverage by ensuring that the verification process executes each side of the subdomain partitions defined by a decision's conditions in a significant manner.

3. There are known problems/limitations with the current definition of MCDC. How should these be addressed?

The known (claimed) problems addressed in this study are:

a. MCDC doesn't find errors.

   This study shows that test sets satisfying all forms of MCDC are capable of detecting certain types of errors, if they are present. However, as with all forms of nonexhaustive testing, there is a probability associated with the detection of errors, and MCDC isn't guaranteed to detect all errors. This study shows that MCDC has a high probability of error detection for the cost incurred (number of tests), especially when compared with other coverage criteria specified in DO-178B (Statement Coverage, Decision Coverage).

b. MCDC, as defined, can't be achieved for all expressions. How should these be addressed?

   A MCDC solvable expression is a Boolean expression for which there is at least one coverage set given a specific definition for MCDC. This study shows that Unique-Cause MCDC is solvable for a small portion of the theoretical Boolean expression space. Unique-Cause+Masking MCDC is shown to have wider applicability and Masking MCDC is shown to have even wider applicability. It is shown empirically, with expressions extracted from five level A line replaceable units (LRUs), that this problem with MCDC may not be as large a problem as some have claimed.

c. MCDC, as defined, ignores the relational operator and operand part of the logic space (i.e., MCDC applies only to Boolean objects).

   Extensions to MCDC for relational operators operating on non-Boolean operands are defined. It is shown how to incorporate these extensions into the definition of MCDC and apply them. Empirical data is provided supporting the magnitude of this problem.

4. Multiple interpretations of MCDC exist and have been used in airplane certifications. Which one (if any) should be standardized on?

   To address this question, theoretical and empirical answers to the following questions were obtained for the three forms of MCDC:

   a. What is the minimum number of tests necessary to satisfy the three different forms?

      The theoretical analysis shows that Masking MCDC would be the easiest form to satisfy as it required fewer tests than the other forms of MCDC. Two empirical investigations, one using logic expressions extracted from five LRUs and the

other using generated expressions to cover the entire singular Boolean expression (SBE) space, confirmed the theory.

b. What is the minimum probability of logic error detection for the three different forms?

A model was developed for the error detecting capabilities of any coverage criterion. This model defined an error as having an incorrect Boolean function in the implementation (i.e., the implementation was not what was specified). Using this model, a theoretical analysis shows that even though Masking MCDC could allow fewer tests than Unique-Cause MCDC, its performance in detecting incorrect Boolean functions was not that much different. An empirical analysis performed against logical expressions extracted from five LRUs not only confirmed the theory, but also showed that the difference was smaller than the theory predicted. An additional empirical analysis using generated expressions covering the SBE space and injected faults using the rules of mutation also showed that the performance of the three forms of MCDC was nearly identical from the probability of error detection viewpoint.

c. How many independence pairs per condition are allowed by the three different forms?

An empirical investigation shows that there were more independence pairs for Masking MCDC than for either of the unique-cause forms.

d. How many minimal test sets exist on average for the three different forms?

An empirical investigation shows that Masking MCDC is satisfied by a greater number of coverage test sets.

Based on the results from all the different analyses performed during this study, it was concluded that Masking MCDC should be the preferred form of MCDC. Masking MCDC requires equivalent numbers of tests to the currently defined Unique-Cause MCDC, which allows it to provide equivalent error detection. However, Masking MCDC allows for more independence pairs per condition and more coverage test sets per expression, which allows it to be applied more cost-effectively.

1. INTRODUCTION.

This is the final report for a study into different forms of Modified Condition Decision Coverage (MCDC). MCDC is a structural coverage criterion used in DO-178B [1] to assist with the assessment of the adequacy of the requirements-based testing process. This level of coverage is required for Level A software [DO-178B pg. 74, table A-7].

This study provides detailed data that supports a rational decision about what level of structural coverage to require for what has been identified as high-integrity or safety-critical software. This study was undertaken in response to a number of issues:

a. It is not generally understood what structural coverage in general, and MCDC in particular, is supposed to be doing (i.e., why is it done? why is it needed?).

   It has been claimed that MCDC is nonvalue added (i.e., it does not find errors).

   In section 2, with support from appendices A and B, background material was provided on coverage and MCDC. The primary purpose of structural coverage is not to find errors per se. Instead, structural coverage is a check and balance on the requirements-based verification process. This check and balance is necessary because the requirements-based verification can miss important features of the implementation. The examples that support this are in appendix A. It will also be shown that MCDC is a weak measure [2] of equivalence class and boundary value coverage. The support material for this is contained in appendix B. Section 4 shows how to extend the definition of MCDC so that it is a strong measure [2] of equivalence class and boundary value coverage. When one takes all three together (section 2, appendices A and B), it will be shown that the most important function of structural coverage is for process assurance and improvement.

   Section 8 investigates the adequacy of the different forms of MCDC in the software mutation (or fault injection) domain. The data from this investigation shows that test sets that satisfy all forms of MCDC *are* capable of detecting certain types of errors, if they are present. The data also shows that test sets that satisfy MCDC are not capable of detecting all errors. The issue of whether test sets satisfying MCDC are cost-effective for the errors they do find is not addressed in this report. As was stated previously, this report takes the view that structural coverage, in general, and MCDC in particular, is for process assurance. As is addressed in sections 5 and 6, the main advantage of MCDC is its cost effectiveness from the probability of error detection viewpoint.

b. It has been claimed that MCDC is difficult to understand (i.e., what does it mean?).

   Section 3, with support from one of the references that is on the web [3], clarifies MCDC's definition. A mathematical definition is provided which applies to the (NOT, AND, OR) Boolean operators. Then a graph coloring definition is provided which applies to the complete set (NOT, AND, OR, XOR, $=, \neq, <, \leq, >, \geq$) of Boolean operators. This definition allows you to overlay two colored graphs for an expression and two test

cases. If yellow plus blue make green in the overlay, then you have MCDC. Appendix B also provides some help here, as it shows *what* MCDC does.

c. MCDC, as defined, cannot be achieved for all expressions. How is this dealt with?

MCDC solvable expression is referred to as a Boolean expression for which there is at least one coverage set. Section 2.4 shows that only a small percentage of theoretically possible Boolean expressions are solvable for MCDC as defined in DO-178B. Section 3 defines three different forms of MCDC, one of which is the DO-178B form. Section 7 addresses the issue of MCDC solvability in general. Section 7.3 shows that there are forms of MCDC that have wider applicability than the DO-178B form. Appendix C contains an abstract form of the expressions extracted from the Ada source code of five Level A systems. Using this empirical data from appendix C, the problem with MCDC, as defined, may not be as large a problem as some have claimed.

d. MCDC, as defined, ignores the relational operator and operand part of the logic space (i.e., MCDC applies only to Booleans, what is to be done about non-Booleans?).

Note: In this report, the term Booleans is used as a noun for any Boolean condition, expression, function, object, operand, or operator.

Section 3 defines three different forms of MCDC. These definitions apply to relational operators operating on Boolean operands. Section 4 defines extensions to MCDC for relational operators operating on non-Boolean operands. It will also be shown how to incorporate these extensions into the definition of MCDC and how to apply them. Appendix C provides empirical data that supports the magnitude of this problem.

e. Multiple interpretations of MCDC exist and have been used in airplane certifications. Which one (if any) should be standardized on? How does one go about making that choice?

Section 3 defines three different forms of MCDC. In sections 5, 6, and 8, a number of theoretical and empirical analyses are performed into the performance of the different MCDC forms. The minimum number of tests in a test set, the minimum probability of logic error detection, the number of independence pairs per condition, and the average number of coverage compliant minimal test sets per expression have been investigated. The methodology defined therein can be used to evaluate other alternative coverage criteria. The analyses contained in this report suggest that Masking MCDC, as defined in section 3.2.3, should be the definition which is standardized on.

As has been mentioned several times, three different forms of MCDC are defined in section 3. One of those forms is the one defined in DO-178B, which is referred to as Unique-Cause MCDC. Including this form makes sense, since that is in the current guidelines. One of those forms is the one suggested in reference 4 that deals with the coupling problem, which is referred to as Unique-Cause + Masking MCDC. Including this form makes sense, as it has been used in airplane certifications and as a reference by automated coverage analysis tool vendors. One of

those forms is the weakest possible form of MCDC, which is referred to as Masking MCDC. This form is the weakest in that it allows for the independence of a condition to be determined strictly by the Boolean Difference Function [5]. The other forms place additional constraints on an independence pair beyond those imposed by the Boolean Difference Function. Inclusion of this form makes sense since, if it is found acceptable, then all other stronger forms will also be acceptable. Other stronger forms of MCDC can be defined. This report only looked at the three defined.

2. BACKGROUND ON COVERAGE.

This section provides background material on coverage. First, a justification is given as to why coverage of any kind should be a part of the standard software development process, in particular the verification process. The discussion is specialized down to the issue of structural coverage. Appendix A provides a detailed example supporting the structural coverage position. The discussion is specialized down to the MCDC itself. Appendix B provides the detailed examples that support the MCDC position. Finally, the need for looking at multiple definitions for MCDC is discussed.

2.1 WHY COVERAGE?

This section addresses the question *why should* coverage *be part of the verification process?* There are three major reasons for using coverage, the first two of which are interrelated:

a. The primary point of coverage, and its measurement, is to help manage risk by giving the people doing, managing, and auditing the verification activities an empirical sense of the extent of verification accomplished (i.e., adequacy). Note that there are a large number of different classes of coverage and associated measures, with different levels of thoroughness within the classes.

b. Secondarily, measurement of coverage provides an exit criteria for when to stop the verification process (i.e., completion).

c. Finally, coverage and its measurement supports process assurance, along with process improvement.

The first two above are interrelated in that they are saying an exit criteria is needed for the verification process. Figure 1 shows a simplified flow diagram for the Verification Process model.

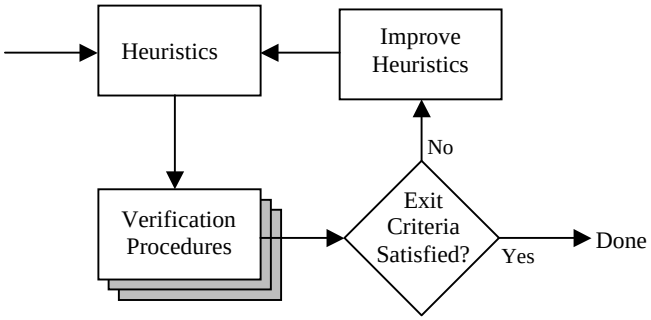


FIGURE 1. SIMPLIFIED VERIFICATION PROCESS

This simplified verification process can be logically defined in three steps, as described below:

a. First, a well-defined set of *heuristics* is used as a strategy to develop a set of verification procedures. Early in the development life cycle, the verification procedures can be requirements for any combination of inspections, analyses, or tests to be applied against some artifact of the development process (which will be produced later in the life cycle). Later in the life cycle, the verification procedures can be the actual analysis methods, inspection checklists, or test cases to be applied against the artifacts of the development process.

b. Second, *exit-criteria* are used to determine when you have an adequate set of verification procedures. This is stated in the plural because two different reasons were given earlier, adequacy and completion, for coverage and they may have different criteria. After developing what is believed to be an adequate set of verification procedures, coverage is then determined against the *exit-criteria* to determine if it is done. It helps tremendously if the *exit-criteria* are objective.

c. Third, if coverage is not satisfied (i.e., it is low, less than 100% of the target) the *heuristics* used must be improved to create the verification procedures and develop additional procedures until the *exit-criteria* are satisfied. Additionally, determine why the coverage was low and use that as feedback for process improvement. For example, coverage was low because verification of error conditions was superficial. This implies that the development process needs to be strengthened in the area of considering errors.

Notice that in the above discussion, coverage was used in a very broad sense as being applicable to multiple forms of verification. In most cases, when the topic of coverage comes up, it usually applies only to testing, which is the most popular form of verification currently applied. The broader form of applicability is intended within this report.

2.2 WHY STRUCTURAL COVERAGE?

This section addresses the question *why should* structural *coverage be part of the verification process?* The assumption behind this question is that if the requirements are adequately covered with requirements-based verification, then the implementation (details) should be unimportant.

Unfortunately, both requirements-based verification and the corresponding requirements coverage criteria have some fundamental problems:

a. They are mostly based on heuristics (i.e., rules of thumb/opinion). This means that they will mean different things to different people. This also means that they will have different effectiveness on different projects.

b. They are mostly subjective (an opinion) as opposed to objective. This makes them unrepeatable and potentially unmeasurable. What is adequate in one instance may be inadequate in another, and overkill in yet another instance.

c. They tend to ignore the implementation because they are built on abstract models of the system. This may cause them to miss important features in the implementation (i.e., the tests are not telling you what you think they are). This is demonstrated with the type-of-triangle problem detailed in appendix A.

The analysis in appendix A shows that requirements-based verification may or may not be good enough for particular implementations. It also shows that verification that is good enough for one implementation may or may not be good enough for another. So what was the major thing that happened in appendix A? The answer lies at the heart of verification theory. There were two different approaches taken to the problem. These differences are shown in figure 2.

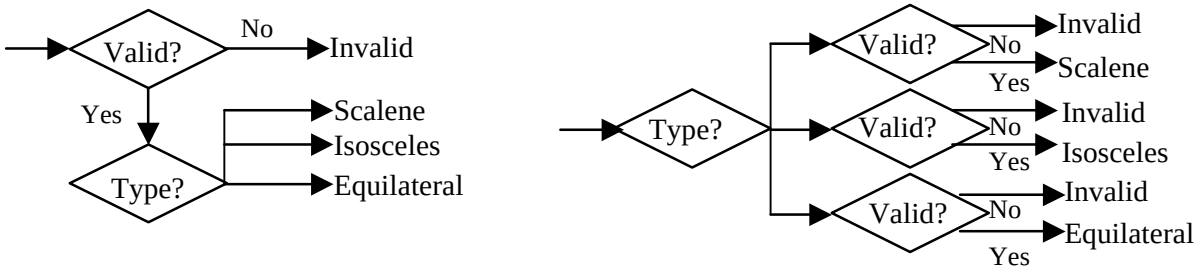


FIGURE 2. ALTERNATIVE APPROACHES FOR TYPE-OF-TRIANGLE PROBLEM

The first approach shown in the left of figure 2 is that of determining validity first, and then if the triangle is valid determining its type. This is the approach taken in the statement of the requirements, as well as some of the implementations. It should be no surprise then that it appears that requirements-based verification would work well for those implementations. The second approach shown in the right of figure 2 is that of determining type first, and then determining validity. As the diagrams show, it is not to be expected that the verification for the validity first check would work well for the type first check, and indeed this is the case for the implementation that uses the second approach. The requirements-based verification has a completely different set of assumptions behind it, which are not valid in the implementation. The opposite is also true, verification which is good for the second approach may not do very well for implementations following the first approach, again because of the assumption mismatch.

The above discussion shows that an expanded verification process model is needed, one that employs both requirements coverage as well as structural coverage. Figure 3 shows this expanded verification process model. In this model there are two sets of exit-criteria to satisfy. First, the coverage of the requirements must be satisfied. Once that has been achieved, then the coverage of the software structure must be satisfied.

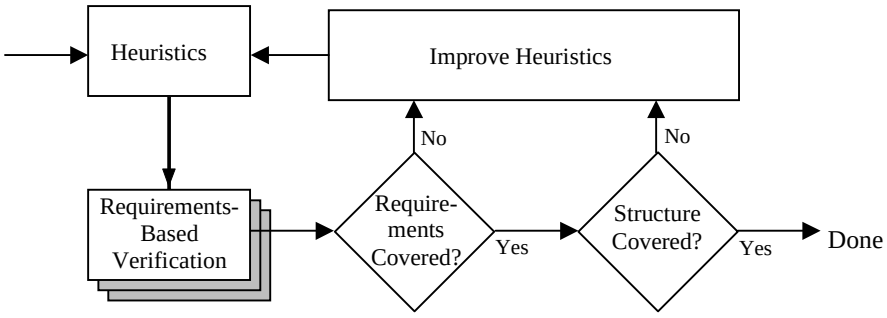


FIGURE 3. EXPANDED VERIFICATION PROCESS

Notice that a process for generating new verification directly off the uncovered structure was not added, even though that will be the first temptation. Instead, what is shown is that if the normal heuristics does not get to 100% structural coverage, then improve (i.e., fix) the heuristics. Generating new verification directly off the structure in order to satisfy structural coverage is not as effective as improving the requirements-based verification. Improving the requirements-based verification actually improves our understanding of how the system is *supposed* to perform, while generating new verification off the structure only demonstrates how the system *does* perform.

In truth, the coverage exit criteria in figure 3 go on somewhat in parallel. That is because failures in the structural coverage area generally point to failures in the development process, and insufficient coverage of the requirements domain.

2.3 WHY MCDC?

This section addresses the question *why should* MCDC *be part of the verification process?* There are four steps in answering this question:

a. All structural coverage measures investigate/spotlight some aspect of the implementation relationships present within the system. DO-178B has decided on three:

- Statement Coverage
- Decision Coverage
- Modified Condition Decision Coverage

These can be abstracted in the following manner:

- Every statement present within the system represents some functionality in the real world that the development process (Requirements Process, Design Process, Coding Process, etc.) felt the system had to provide. If statements are left uncovered by the requirements-based verification process, then that process failed to consider some aspect(s) of the systems implemented functionality. Statement coverage ensures that the verification process has considered sufficient operational scenarios to execute every statement in at least one operational context.

- Every decision present within the system represents a situation in the real world that the development process felt had to be handled specially. This is because each decision partitions (divides) the real world (operational space) into two special cases, one to be handled one way and the other to be handled differently. If branches (decision outcomes) are left uncovered by the requirements-based verification process, then that process failed to consider the special cases the development process thought were important. Decision coverage ensures that the verification process has considered sufficient operational scenarios to execute every special case the system was designed for in at least one operational context.

- Every condition present within the system is supposed to uniquely affect the choice between special cases under the right circumstances (i.e., each condition guards a boundary between different equivalence classes, multiple equivalence class boundaries are potentially present within a multicondition decision). If the requirements-based verification process does not demonstrate a condition's independence, then that process failed to consider those situations where that condition alone was significant to determining how the system would respond. In essence, the condition's effect on the correct operational behavior of the system has not been demonstrated. MCDC ensures that the verification process has considered sufficient operational scenarios to demonstrate that each condition is able to correctly affect the operational behavior of the system in at least one operational context.

b. These implementation relationships are supposed to be significant within the real world the system is to operate within. The implementation process is not supposed to introduce functionality or special cases just for the sake of building it. In many cases, real-time systems have enough trouble delivering the functionality and handling the special cases they are supposed to without adding additional bells and whistles. When these relationships are left uncovered, one obvious question is *were they really significant enough to be present within the system?*

c. These implementation relationships are supposed to operate/function correctly. If indeed the functionality and special cases reflect what the system is to do, then it should do it correctly. Nobody likes it when their systems misbehave. This is especially true when system misbehavior can result in unacceptable loss, which is what distinguishes a high-integrity/safety-critical system.

d. Whenever the requirements-based verification fails to achieve coverage, there are problems within the development process which need investigation. The most prevalent problem noticed by the author of this report is undocumented implementation (i.e., design and/or coding) decisions. Most of these decisions should have resulted in derived (i.e., low level) requirements, which the requirements-based verification process should have addressed. It is the presence of all the "shoulds" in the previous statement that points to a flawed development process. Fixing those flaws is considered in today's software engineering environment to be of paramount importance. In essence, the major usage of coverage is for process assurance and improvement, not for finding errors in the product.

MCDC should be used as a structural coverage criterion because it attempts to provide a cost-effective form of logic verification. In essence, MCDC can be thought of as a (weak) measure of the coverage of equivalence classes and boundary values. Full details of this are contained in appendix B.

Unlike some other forms of structural coverage, MCDC can be applied to any representation (graphical or textual, mathematical or not) where logic is expressed. MCDC can also be applied anywhere in the development process (i.e., to any life cycle artifact). MCDC applied to the source code is the lowest level of coverage, while MCDC applied at the requirements would be the highest level.

One would hope that if you covered the requirements to the MCDC level, then coverage of the source code would also be achieved. Unfortunately this is not guaranteed to happen, as the analyses in appendix A for implementations Nos. 2 (see A.4) and 4 (see A.6) show. Given the limited success requirements-based verification obtained on something as simple as the triangle problem, consider how limited the performance could be on a real system. This is just further justification for the necessity of structural coverage and something beyond Statement Coverage and Decision Coverage.

2.4 WHICH MCDC?

Many interpretations exist for a coverage criterion satisfying the intent of MCDC. Some of these interpretations result in verification that is weaker at detecting errors than others. Some of these interpretations result in verification that is more costly than others. The intent of this report is to provide data supporting a choice between the different interpretations (forms) studied herein. Other possibilities than those studied exist, but their analysis will have to be performed elsewhere.

One may ask why there is a need to look at these different interpretations for MCDC. To understand the major motivation behind this study, consider the following. MCDC is defined in reference 1 through the following (excerpted) definitions:

- Modified Condition Decision Coverage — 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 [pg. A-10].

- Decision — A Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. If a condition appears more than once in a decision, each occurrence is a distinct condition [pg. A-8].
- Condition — A Boolean expression containing no Boolean operators [pg. A-7].

From the above definitions the following can be derived:

a. Every Boolean expression in the program will need to evaluate to both True and False (since they can only assume those two values, they constitute every possible outcome).

b. Every condition in the program will need to evaluate to both True and False (same reasoning as (a.) above).

c. Conditions are either Boolean valued objects (variables and constants) or functions which return Boolean results. These functions can either be user defined, language defined, or the (language defined) relational operators (for objects of non-Boolean types).

d. Conditions that occur more than once in a decision need each occurrence to demonstrate its independence.

It is with this last derivation that a problem appears. The problem concerns the inability to test certain expressions under the above set of definitions. Notice that this definition sequence for MCDC is requiring a unique cause for a change in decision outcome due to a single condition change to show that condition's independence. This is from the requirement that all other conditions are held fixed while the condition of interest is changed (hence, the term unique-cause).

Consider the following expression: (A and B) or (A and C)

By the definitions given above, this MCDC decision consists of the

- first occurrence of A (in (A and B)),
- only occurrence of B,
- second occurrence of A (in (A and C)), and
- only occurrence of C.

The problem comes about when testing the first occurrence of A. By the definitions in DO-178B, the first occurrence of A must be toggled between True and False, changing the outcome of the expression, while holding all other possible conditions (B, second occurrence of A, C) fixed. It is not possible to hold the second occurrence of A fixed

while changing the first occurrence, so (B, C) needs to be held fixed. However, DO-178B provides no guidance on how to hold (B, C) fixed in order to demonstrate the independence of each occurrence of (A) individually.

This limits the application of this definition for MCDC to singular Boolean expressions (SBEs). Boolean functions can be categorized into three classes:

a. Degenerate functions—these are functions which are not a function of all the conditions present in the expression describing the function (i.e., some of the conditions in the expression are redundant, therefore the expression can be reduced/simplified to an expression with fewer conditions) [5]. For example, consider the following expression:

(A and B) or (not A and C) or (A and C and D) or (C and D and E)

Table 1 is a partial truth table for the above expression showing the condition combinations when each of the subexpressions in the above expression will cause the expression to evaluate to True. There is a dot in the subexpression's column for those condition combination rows that cause the subexpression to return True. Looking at the column for the subexpression *(C and D and E)*, notice that none of the rows where dots are present are unique (i.e., are not covered by another of the subexpression). This means that this subexpression can be removed from the expression, and the resulting (simplified/reduced) expression will still describe the same Boolean function. Since this subexpression is the only one in which the condition *E* occurs, this means that the Boolean function described by this expression is a function of the conditions (*A, B, C, D*) only. Since this Boolean function is a function of four conditions, not five, it is identified as a degenerate function.

TABLE 1. PARTIAL TRUTH TABLE FOR THE EXPRESSION
(A and B) or (not A and C) or (A and C and D) or (C and D and E)

		A and B	*not A and C*	*A and C and D*	*C and D and E*
4	FFTFF		●		
5	FFTFT		●		
6	FFTTF		●		
7	FFTTT		●		●
12	FTTFF		●		
13	FTTFT		●		
14	FTTTF		●		
15	FTTTT		●		●
22	TFTTF			●	
23	TFTTT			●	●
24	TTFFF	●			
25	TTFFT	●			
26	TTFTF	●			
27	TTFTT	●			
28	TTTFF	●			
29	TTTFT	●			
30	TTTTF	●		●	
31	TTTTT	●		●	●

b. Singular Boolean Expressions—these are functions which are not degenerate and can be described by expressions where all of the conditions have a single occurrence only.

c. Nonsingular Boolean Expressions—these are nondegenerate functions which cannot be described by expressions where every condition has a single occurrence only. For example, consider the following expression:

(A and B and C) or (not A and not B and not C)

Table 2 is a partial truth table for the above expression. This table shows that either subexpression cannot be removed. Additionally, since both of the subexpressions are irreducible minterms [5], either subexpression cannot be removed. Hence, no singular Boolean expression exists for this Boolean function.

TABLE 2. PARTIAL TRUTH TABLE FOR THE EXPRESSION
(A and B and C) or (not A and not B and not C)

		A and B and C	*not A and not B and not C*
0	FFF		●
7	TTT	●	

As stated previously, the DO-178B definition for MCDC is restricted to SBEs. This limits the usefulness of this definition if the number of SBEs represent a small fraction of the nondegenerate functions. This is indeed the case as figure 4 shows. Figure 4 is a semilog plot showing the number of Boolean functions which are degenerate, which are not degenerate, and can be described by a SBE, and which are not degenerate and cannot be described by an SBE. The percentage of SBE functions is growing increasingly insignificant as the number of conditions grows.

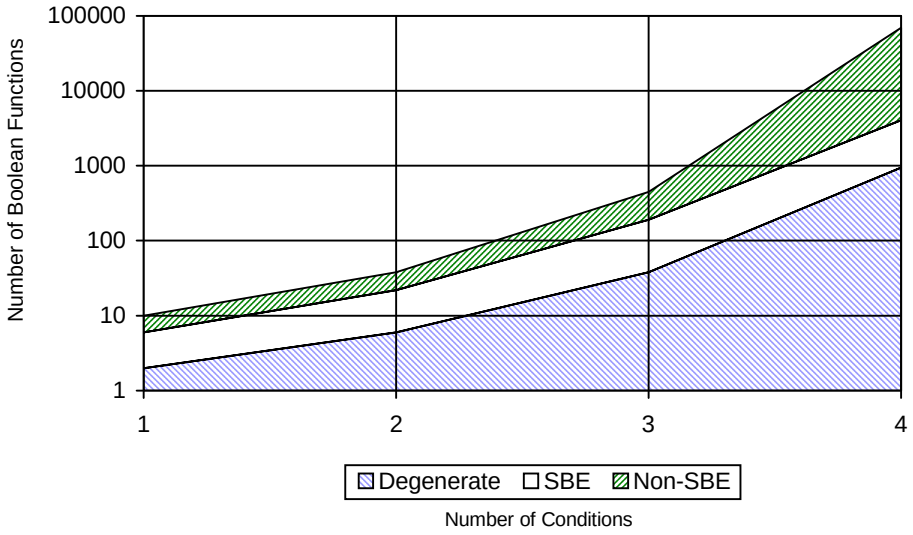


FIGURE 4. BOOLEAN FUNCTION CLASSIFICATION VS NUMBER OF CONDITIONS

Table 3 provides the same data as figure 4 but is extended to include the numbers for five conditions.

TABLE 3. BOOLEAN FUNCTION CLASSIFICATION PER CONDITION LEVEL

No. Conditions	No. Degenerate	No. SBE	No. Non-SBE
1	2	2	
2	6	8	2
3	38	114	104
4	942	2,154	62,440
5	325,262	19,286	4,294,622,748

From the theoretical point of view, the above data indicates that the DO-178B definition for MCDC has a very limited range of applicability. This suggests that an alternative applicable to the non-SBEs needs to be found. Further analysis is provided in section 7 of this report.

3. DEFINITIONS FOR MODIFIED CONDITION DECISION COVERAGE.

This section defines the three forms of MCDC studied for this report. The forms of MCDC studied herein are known as

- Unique-Cause MCDC,
- Unique-Cause + Masking MCDC, and
- Masking MCDC.

First, start with a base definition for MCDC itself. The justification for this definition is presented in reference 12. MCDC is defined as

- every statement in the program has been invoked at least once.
- every point of entry and exit in the program has been invoked at least once.
- every control statement (i.e., branchpoint) in the program has taken all possible outcomes (i.e., branches) at least once.
- every nonconstant Boolean expression in the program has evaluated to both a true and a false result.
- every nonconstant condition in a Boolean expression in the program has evaluated to both a true and a false result.
- every nonconstant condition in a Boolean expression in the program has been shown to independently affect that expression's outcome.

The difference between the forms of MCDC that are studied is in the definition for showing independence (final clause in the base definition above). Therefore, the definitions given in section 3.2 concentrate on what it means to show a condition's independence. The independence

criterion was also concentrated on since the empirical data used in this study consists of Boolean expressions extracted from the source code of five airborne systems (appendix C) and Boolean expressions built according to rules of mutation (appendix D).

3.1 DEFINITION OF INDEPENDENCE PAIRS.

A part of MCDC is defined as showing the independence of each condition within a Boolean expression. This means that the expression's outcome (i.e., result evaluation) will be toggled (between True and False) as a result of each condition being toggled (between True and False) in a way that the condition of interest is the only condition which has influence on the expression's outcome during the toggling.

This results in a need for two tests, known as an independence pair, for each condition within a decision. This is not to say that these two tests are each unique to the condition. Generally, one or both of these tests can be paired with other tests to form an independence pair for a different condition.

This section provides the definition for what constitutes legal MCDC test pairs to show a condition's independence (independence pairs). This definition is given in two parts.

The first part is a formal mathematical definition. This definition is not complete in that the mathematics used do not recognize either the XOR operator (it is not considered a primitive Boolean operation) or the relational operators (=, ≠, <, ≤, >, ≥) operating on Boolean objects. This document does not extend the mathematics to properly handle the XOR and relational operators. Instead an equivalent alternative which handles the XOR and relational operators properly is given in part two.

The second part is a graph-coloring algorithm. This algorithm also has mathematics behind it, which properly handle the XOR and relational operators. This is conceptually the easiest definition to understand.

3.1.1 Mathematical Definition.

This section defines an independence pair in terms of the Boolean Difference Function [5]. This function works for expressions that are written in terms of Boolean variables and the Boolean operators (NOT, AND, OR). The extensions needed for the XOR operator and the relational operators (=, ≠, <, ≤, >, ≥) operating on Boolean objects are not made here.

The independence pair can be defined through the following derivations.

Let F() be a (Boolean) function of n Boolean conditions. This function will be represented by a Boolean expression in the software (either in a program or its documentation) under analysis. Note that a Boolean function can be represented by multiple Boolean expressions.

- Let $c = (c_1, ..., c_i, ..., c_n)$ be a vector of n Boolean values (conditions) (i.e., a Boolean n-vector, also known as a test).

- The Boolean Difference of F(c) with respect to c_i, denoted $\delta F(c)/\delta c_i$, is defined as [5]: $\delta F(c)/\delta c_i = F(c_1, ..., c_i, ..., c_n) \oplus F(c_1, ..., \neg c_i, ..., c_n)$

The Boolean difference of F(c) with respect to c_i is False if toggling just c_i does not toggle (change) the outcome and is True if toggling just c_i does toggle (change) the outcome. Using the Boolean difference, a set of constraints can be calculated which must be satisfied if the independence of a condition c_i are to be shown.
For Masking MCDC, the independence pair can be formally defined as follows:

- Let x and y be two (test) vectors
- x and y form a Masking MCDC independence pair for the i^{th} condition (c_i) IFF

  - $(x_i = \neg y_i)$ and
  - $(F(x) = \neg F(y))$ and
  - $(\delta F(x)/\delta x_i)$ and
  - $(\delta F(y)/\delta y_i)$

What the formal mathematics for the Masking MCDC independence pair is saying is the

a. condition c_i must toggle (between True and False) between the two tests;

b. expression must return different results for the two tests (i.e., toggle between True and False);

c. condition c_i must have influence on the outcome of the expression when the first test (x) is applied; and

d. condition c_i must have influence on the outcome of the expression when the second test (y) is applied.

For Unique-Cause MCDC, the independence pair can be formally defined as follows:

- Let x and y be two (test) vectors
- x and y form a Unique-Cause MCDC independence pair for the i^{th} condition (c_i) IFF

  - (for all j in 1 .. n, $j \neq i$, $x_j = y_j$) and
  - $(x_i = \neg y_i)$ and
  - $(F(x) = \neg F(y))$ and
  - $(\delta F(c)/\delta c_i)$, where $c_j = x_j = y_j$, for all j in 1 .. n, $j \neq i$.

Note that under the first two clauses, $(\delta F(x)/\delta x_i) = (\delta F(y)/\delta y_i)$, so the fourth clause could have read either "$\delta F(x)/\delta x_i$" or "$\delta F(y)/\delta y_i$." The Boolean Difference Function only needs to be calculated once since all other conditions are held fixed.

The changes between the two forms of MCDC are highlighted by the mathematical definitions. Masking MCDC allows any number of conditions to change so long as only the condition of interest has influence on the outcome of the expression. This generally allows for more coverage test sets (i.e., test sets that satisfy a coverage criterion).

For example:

Let *F() = A or (B and C)*

Then

δF(A or (B and C))/δA =
(False or (B and C)) XOR (True or (B and C)) =
(B and C) XOR True =
not (B and C)

To show *A*'s independence, the subexpression must ensure that "*B and C*" remains False while toggling *A* between True and False.

The Unique-Cause form of MCDC will require that *B* and *C* remain fixed at one of the condition combinations (0:(FF), 1:(FT), 2:(TF)) while *A* is toggled. This results in three test sets:

- (0:(FFF), 4:(TFF))
- (1:(FFT), 5:(TFT))
- (2:(FTF), 6:(TTF))

Masking MCDC allows *B* and *C* to change between any two of the condition combinations ((FF), (FT), (TF)) while *A* is toggled. This results in nine test sets:

- (0:(FFF), 4:(TFF))
- (0:(FFF), 5:(TFT))
- (0:(FFF), 6:(TTF))
- (1:(FFT), 4:(TFF))
- (1:(FFT), 5:(TFT))
- (1:(FFT), 6:(TTF))
- (2:(FTF), 4:(TFF))
- (2:(FTF), 5:(TFT))
- (2:(FTF), 6:(TTF))

Section 6 provides the empirical data substantiating the claim of more coverage test sets. This larger number of coverage test sets is advantageous in that requirements-based testing has a larger target to hit for satisfying structural coverage. This should reduce both the difficulty and cost of satisfying MCDC.

3.1.2 Graph-Coloring Definition.

This section defines an independence pair in terms of coloring a graph. Unlike a Boolean function, a Boolean expression can be represented in a parse tree, i.e., an expression tree. This data structure is more properly known as a graph, and a graph can be colored (or decorated) with annotations. The annotations that will be used are the values of execution derived from applying a specific pair of tests. The differences between the colors of the two graphs representing each test result in a colored graph identifying the condition(s) that influenced the change in the expression's outcome.

The independence pair can be defined as follows.

- Let *F()* be a Boolean function (expression) of n Boolean conditions.
- Let $c = (c_1, ..., c_i, ..., c_n)$ be a vector of n Boolean values (conditions) (i.e., a Boolean n-vector, also known as a test).
- Let *EF()* be an expression tree for the Boolean expression of *F()*.

An expression tree *EF(c)* is evaluated for a test c according to the following rules:

a. Set the (leaf) conditions of the tree to c, and then

b. walk the values up to the root through the operators according to the rules of Boolean algebra. (Note: for short-circuit forms, the values on the right-hand side (RHS) might all be changed to Not-Executed depending on the value of the left-hand side (LHS), i.e., if the LHS of an And-Then is False, the RHS is changed to Not-Executed, if the LHS of an Or-Else is True, the RHS is changed to Not-Executed.)

Given a Boolean function *F()* and two truth vectors *x* and *y* for the expression,

- Construct an expression tree for each truth vector, *EF(x)* and *EF(y)*
- Construct an influence tree

$$IE(F(x),F(y)) = EF(x)\ tree\_xor\ EF(y)$$

by applying the tree_xor function between corresponding annotations in the two expression trees to the corresponding annotation in the influence tree. The tree_xor function is an extension of the standard XOR function designed to handle the nonexecutions of the RHS possible with short-circuit operators, and is defined in figure 5.

Let the Influence_Set be the set of conditions in the influence tree that have a path of all Trues to the root (i.e., these are the conditions which changed between the two tests and had the effect of that change make it to the outcome). If there is a single condition c_i in the influence set, then *x* and *y* form a Masking independence pair for the condition (i.e., *x* and *y* show c_i's independence).

If additionally c_i is the only condition which has a True in the influence tree (i.e., only c_i changed between the two tests), then *x* and *y* form a Unique-Cause independence pair for the condition.

LHS / RHS	F	T	NE
F	F	T	F
T	T	F	F
NE	F	F	F

where NE = not executed

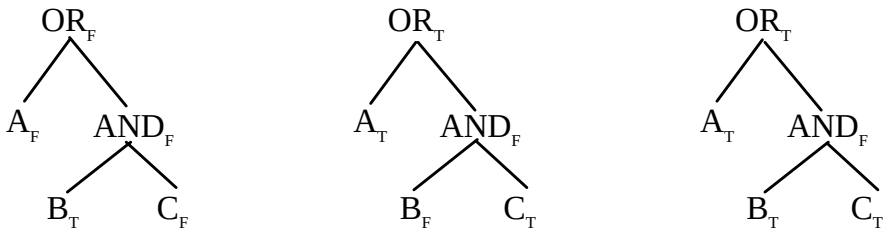
FIGURE 5. DEFINITION OF tree_xor

The following examples show how the tree-coloring algorithm works for Masking MCDC. The analysis for Unique-Cause MCDC is the same as that for Masking MCDC with the additional constraint that only one single condition is allowed to change value. Masking MCDC is used in these examples since the analysis for Unique-Cause MCDC is much simpler and more straightforward (i.e., did only a single condition change along with the expression results).

The first example will use the expression "*A or (B and C)*", presented in figure 6, and the second example will use the expression "*A or else (B and C)*", presented in figure 7. Both examples will use the condition combinations in (A, B, C) of 2:(FTF), whose evaluated Expression Tree is presented in figures 6(a) and 7(a), and 5:(TFT), whose evaluated Expression Tree is presented in figures 6(b) and 7(b). The annotations (or colors) are shown in the trees as subscripts where the following are the subscript meanings:

T - True
F - False
X - Not-Executed

The Influence Trees are presented in figures 6(c) and 7(c). Recall that the annotations for the Influence Tree are obtained by tree_xoring the corresponding annotations in the Expression Trees.



(a) (b) (c)

FIGURE 6. GRAPH COLORING FOR *A or (B and C)*

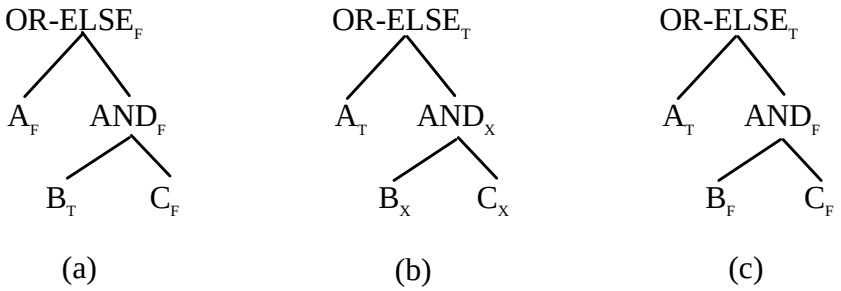


FIGURE 7. GRAPH COLORING FOR *A or else (B and C)*

For example, in figure 6 the influence annotation for the OR operator is obtained by (False tree_xor True = True). Since the OR operator is the decision operator, the influence annotation must be True if the pair of tests is to demonstrate the independence of *any* condition. Note that the decision operator changing value between the two tests is a necessary (by the definition for MCDC), but not sufficient, condition for a pair of tests to be an independence pair (i.e., not every pair of tests which change the decision outcome will form an independence pair). The annotations for the rest of the nodes in the Influence Tree can be obtained in like manner to how the OR was done.

The Influence Set is then obtained by finding all of those conditions that have a path of True annotations all the way up the Influence Tree.

Looking at figure 6(c), it can be determined that the Influence Set for these two condition combinations is (*A*), even though all three conditions changed value between the two Expression Trees (figures 6(a) and 6(b)). Conditions *B* and *C* are prevented from being members of the Influence Set by the AND operator remaining False in figures 6(a) and 6(b), and therefore being False in the Influence Tree (figure 6(c)). The AND operator blocked the effects of *B*'s and *C*'s changes from influencing the decision change.

Looking at figure 7(c), it can be determined that the Influence Set for these two condition combinations is again just (*A*), even though all three conditions changed value. Conditions *B* and *C* are prevented from being members of the Influence Set since they (and the AND operator to which they are conditions) were executed only once. Therefore they and the AND operator are False in the Influence Tree (figure 7(c)). Recall that for a condition to have influence, it must toggle between True and False in two executions.

The following three figures demonstrate that the graph-coloring algorithm works with XOR operators (figure 8), relational operators (figure 9), and coupled conditions (figure 10). They are presented without detailed explanations as the reasoning and format is the same as that used in figures 6 and 7. However, unlike figures 6 and 7 where *A*'s independence is shown, *B*'s independence will be shown in figures 8, 9, and 10. Figures 8 and 9 will use the condition combinations in (A, B, C, D) of 10:(TFTF) presented in figures 8(a) and 9(a); and 13:(TTFT) presented in figures 8(b) and 9(b). Figure 10 will use the condition combinations in (A, B, C) of 4:(TFF) presented in figure 10(a) and 6:(TTF) presented in figure 10(b).

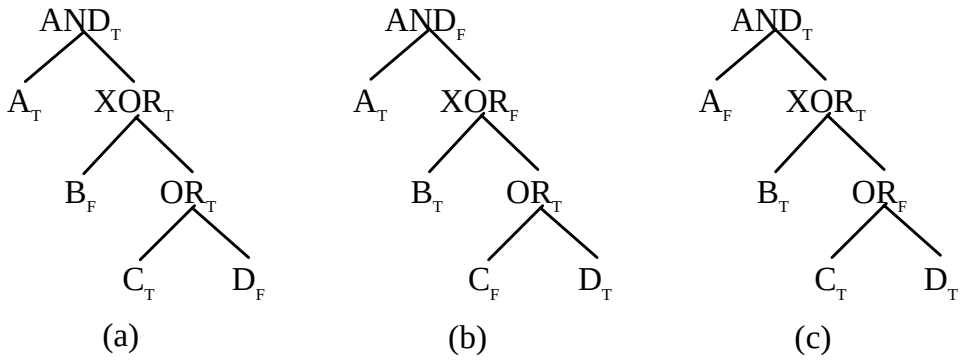


FIGURE 8. GRAPH COLORING FOR *A and (B XOR (C or D))*

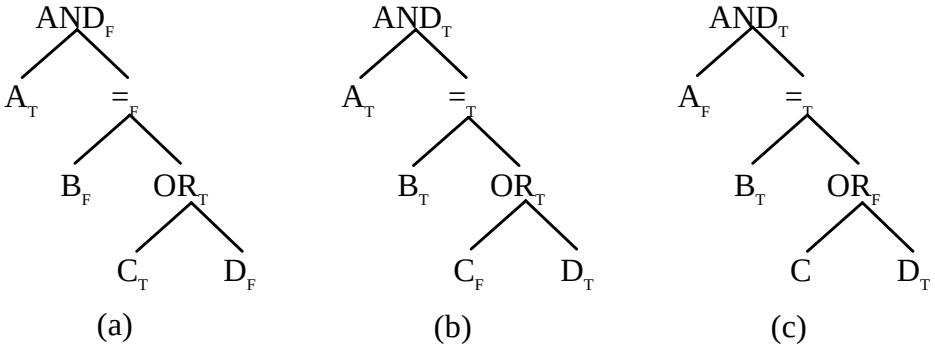


FIGURE 9. GRAPH COLORING FOR *A and (B = (C or D))*

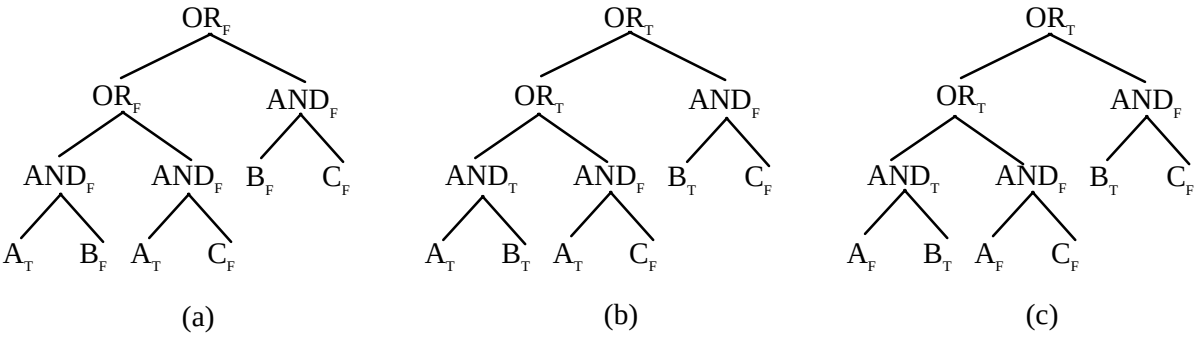


FIGURE 10. GRAPH COLORING FOR *(A and B) or (A and C) or (B and C)*

Notice in figure 10 that the test set (4:(TFF) and 6:(TTF)) demonstrates the independence of the first occurrence of *B.*

3.2 THE THREE WORKING DEFINITIONS.

Recall that the three forms of MCDC that we will study are Unique-Cause MCDC, Unique-Cause + Masking MCDC, and Masking MCDC. They are defined in the following subsections.

3.2.1 Unique-Cause MCDC.

Unique-Cause MCDC will require a unique cause (toggle a single condition and change the expression result) for all possible (uncoupled) conditions. In the case of strongly coupled conditions, no coverage set is possible as DO-178B provided no guidance on how to cover these conditions. Fortunately, as the data in appendix C shows, expressions with strongly coupled conditions are quite rare in airborne software (72 of 20,256 expressions).

3.2.2 Unique-Cause + Masking MCDC.

Unique-Cause + Masking MCDC will require a unique cause (toggle a single condition and change the expression result) for all possible (uncoupled) conditions. In the case of strongly coupled conditions, masking will be allowed for that condition only (i.e., all other (uncoupled) conditions will remain fixed). This definition complies with the suggestion made in reference 4.

3.2.3 Masking MCDC.

Masking MCDC, as its name implies, allows masking in all cases. This is an extension beyond the suggestion made in reference 4 that masking be allowed for strongly coupled conditions only. This extension to allow masking for all conditions was motivated by the Boolean Difference Function. As the example given in section 3.1.1 showed, when demonstrating the independence of *A* in *"A or (B and C),"* all that was required was that the subexpression *"B and C,"* remain False.

4. EXTENSION OF MCDC FOR RELATIONAL OPERATORS.

Currently, MCDC concerns itself entirely with Boolean operators and operands. This means that MCDC is ignoring relational operators and non-Boolean operands. This presents the possibility that MCDC is ignoring a significant portion of the logic verification domain. The data in tables 4 and 5 suggests that perhaps MCDC should be providing some assurance for the relational operators between non-Booleans, as it does for the Boolean operators. Unfortunately, MCDC as defined does not address what needs to be done in order to cover these relational operators (i.e., provide some assurance that they are correct). This section proposes some extensions to the basic MCDC definition to cover the relational operators (applied to non-Booleans).

Recall that appendix C contains all the logic expressions extracted from five examples of airborne software source code. Table 4 presents a condition profile by class from the expressions contained in appendix C using the operand terminology from that appendix (e.g., *Iv* means *integer-object*). The data in table 4 shows that approximately 68% of the conditions occurring in the profiled airborne software are discrete Boolean values. This leaves 32% of the conditions

consisting of some form of relational operator (the Ada membership operator (in, not in) is including as a relational operator).

TABLE 4. CONDITION OCCURRENCES BY CLASS

Class of Relational Operators	Total	Percentage
Generic Objects (?v)	21	0.074
Access Objects (Av)	284	1.003
Boolean Array Objects (Bav)	11	0.039
Boolean Objects (Bv)	19,229	67.875
Character Array Objects (Cav)	12	0.042
Character Objects (Cv)	88	0.311
Enumerated Array Objects (Eav)	3	0.011
Enumeration Objects (Ev) Membership	65	0.229
Enumeration Objects (Ev)	3,212	11.338
Floating Point Objects (Flv)	197	0.695
Fixed Point Objects (Fxv)	971	3.428
Integer Array Objects (Iav)	1	0.003
Integer Objects (Iv) Membership	76	0.268
Integer Objects (Iv)	4,083	14.412
Record Objects (Rv)	43	0.152
String Array Objects (Sav)	3	0.011
String Objects (Sv)	31	0.109
	28,330	100

In table 4, the first column identifies which base Ada type is operated upon by the relational (or membership) operators (i.e., Integer Objects (Iv) means *integer-object relational-operator integer-object*) or the Boolean objects appearing as conditions (i.e., Boolean objects (Bv)). Note that relational operators between Boolean objects are defined to be Boolean operators by the MCDC definition, therefore the Boolean objects operated upon by the relational operators are identified as conditions, not the relational operators (as is the case for non-Boolean objects). The second column identifies the total number of occurrences of conditions of that class. The third column identifies the percentage.

Table 5 presents an expression profile by class for each of the condition levels of the expressions in appendix B. The data in table 5 show that the majority of expressions occurring in airborne software utilize Boolean discretes as the only conditions (67% for expressions with 1 through 76 conditions, 58% for expressions with 2 through 76 conditions). The data also show that a significant number of expressions have no Booleans for conditions (29% for expressions with 1 through 76 conditions, 24% for expressions with 2 through 76 conditions).

In table 5, the first column identifies the number of unique conditions which appear in the expression(s), except for the last two rows, which show two different levels of totals: one for all the condition levels and one for only the multicondition levels (> 1 condition). The second

TABLE 5. BOOLEAN EXPRESSION PROFILE

Number of Conditions	Boolean Only	Mixed	Relational Only	Total
1	11,430		5,061	16,491
2	1,369	300	593	2,262
3	403	169	113	685
4	231	63	97	391
5	79	33	19	131
6	34	20	25	79
7	15	21	16	52
8	16	14	15	45
9	10	8	17	35
10	4	4		8
11		3		3
12	2	2		4
13	5	2	5	12
14	2	2	5	9
15	2		5	7
16	6	22	1	29
17	2	1		3
18			3	3
19		1		1
21	1			1
25	1			1
28	1			1
33	1			1
49		1		1
76		1		1
1..76 Total	13,614	667	5,975	20,256
2..76 Total	2,184	667	914	3,765

column shows the number of expressions that consist of only Boolean conditions. The third column shows the number of expressions that consist of mixed Boolean and relational conditions. The fourth column shows the number of expressions that consist of relational conditions only (i.e., no Booleans). The fifth column shows the total number of expressions.

The extensions defined in the following two subsections are to ensure that the correct relational operator (*rop*) as well as the correct operands (*A*, *B*) are in place in the relational expression (*A rop B*). As with MCDC, these extensions are not intended to guarantee the absence of all errors, just an acceptably high probability of detecting errors if they occur. Note that the rules for MCDC already cover relational operators between Booleans.

4.1 OPERATOR ASSURANCE EXTENSIONS.

To properly demonstrate that the correct relational operator is in place (either in the requirements or in the implementation) requires two things. The first thing that is required is that the verification set (analyses/tests), which when applied to the operator, is guaranteed to distinguish between the correct operator and the erroneous ones. This means that at least one of the analyses/tests must return a different result between the correct operator and all possible incorrect ones. Recall that there are three relational states between any two objects (*A*, *B*) of the same type:

- *A* equals *B* (equality, $A = B$),
- *A* is greater than *B*, *B* is less than *A* (greater than, $A > B$), and
- *A* is less than *B*, *B* is greater than *A* (less than, $A < B$).

These three basic states then should form the basis for the definition of the verification set. The objective is to find the smallest number of these three basic states such that every relational operator returns a different value for at least one of them. Table 6 shows that all three of these basic states (shown on the left of the table) are necessary to distinguish the six relational operators (shown across the top of the table). For any pair of states, at least two of the relational operators will return the same set of results, therefore more than two states are required. The need for all three basic states was proven by Howden [2].

TABLE 6. RESPONSE PROFILES FOR *A rop B*

Basic States	$A = B$	$A \neq B$	$A > B$	$A \geq B$	$A < B$	$A \leq B$
$A = B$	T	F	F	T	F	T
$A > B$	F	T	T	T	F	F
$A < B$	F	T	F	F	T	T

The second thing that is required to properly demonstrate that the correct relational operator is in place is that the distinguishing analyses/tests be applied in such a way that the results of the analyses/tests are not masked. In essence, this means that the relational operator analyses/tests must be applied in such a way that all three demonstrate the relational operator's independence. Note that this requirement will require three tests for each relational operator, where independence (from the Boolean perspective) normally only requires two. However, from the Boolean expression perspective, that third test does not necessarily tell anything more, since the same truth vector from one of the earlier tests may result (e.g., when verifying the expression $A = B$, only two truth vectors result from the three analyses/tests: (0:F and 1:T)).

4.2 OPERAND ASSURANCE EXTENSIONS.

The next extension to consider for relational operators is providing some assurance that the expressions that represent the operands (objects *(A, B)*) are correct. This is especially important when one of the operands to the relational operator is a constant. Verifying the constant is equivalent to boundary value analysis of the partition it forms in the operational space.

To do this, the following relations are needed between the operands in the verification set:

a. Both operands equal to each other *(A = B)*.

b. The left operand equal to the successor of the right operand in the computer number representation (*A = Succ(B)*).

c. The left operand equal to the predecessor of the right operand in the computer number representation (*A = Pred(B)*).

For example, consider the expressions *A = B* versus *A = C* when *C > B*. The analysis is laid out in table 7. In the first column, the relations that hold between the values for *A* and *B* are specified (e.g., for the second row *A equals B*). The second column gives the response for the (correct) function *A = B* when given the data from the first column. The third column gives the response for the (incorrect) function *A = C* when *C = Succ(B)*. For this case, the incorrect function is considered to be *A = Succ(B)*. For the case when *A = B*, this function gives an incorrect response, as it does when *A = Succ(B)*. These two entries will show up as bold italics in table 8. The fourth column gives the response for the incorrect function *A = C* when *C > Succ(B)*. For this case, the incorrect function is considered to be *A = Succ(B)* + δ, where the δ is some nonzero value. Notice that this function only gives an incorrect response when *A = B*. This entry is the same as the one for the third column, so it will appear in bold italics in table 8 as a single entry (***F***). The entries for *A = Succ(B)* disagree, so the double entry (***T***/F) is shown in table 8.

TABLE 7. RESPONSE PROFILES EXAMPLE

	A = B	*A = C {C = Succ(B)}* *A = Succ(B)*	A = C *{C > Succ(B)}* *A = Succ(B) + δ*
A = B	T	F	F
A= Succ(B)	F	T	F
A = Pred(B)	F	F	F

TABLE 8. RESPONSE PROFILES FOR *A rop B* VS *A rop C, C > B*

	A = C	*A ≠ C*	*A > C*	*A ≥ C*	*A < C*	*A ≤ C*
A = B	***F***	***T***	F	***F***	***T***	T
A = Succ(B)	***T***/F	***F***/T	***F***	T/***F***	F/***T***	***T***
A = Pred(B)	F	T	F	F	T	T

Tables 8 and 9 show that these relations will detect the incorrect operand *C* in the implementation for the specified operand *B*. The first column of the table identifies the desired relation between *A* and *B* for the corresponding row. The second through seventh columns represent the incorrect expression in *(A,C)*, identified by the expression in the first row (i.e., at the top of the column). The entries in the table are the response given by the incorrect expression when the relation between A and B is used. The bold italicized entries in the table are where an incorrect response is given (i.e., F*(A, B)* ≠ F*(A,C)*), therefore the error is detected.

TABLE 9. RESPONSE PROFILES FOR *A rop B* VS *A rop C, C < B*

	$A = C$	$A \neq C$	$A > C$	$A \geq C$	$A < C$	$A \leq C$
$A = B$	***F***	***T***	***T***	T	F	***F***
$A = Succ(B)$	F	T	T	T	F	F
$A = Pred(B)$	***T***/F	***F***/T	F/***T***	***T***	***F***	T/***F***

Table 8 shows that these relations will detect the incorrect operand *C* in the implementation for the specified operand *B*, when $C > B$. The *F/T* entries are due to the differences between when $C = Succ(B)$ versus when $C > Succ(B)$.

Table 9 shows that these relations will detect the incorrect operand *C* in the implementation for the specified operand *B*, when $C < B$. The *F/T* entries are due to the differences between when $C = Pred(B)$ versus when $C < Pred(B)$.

Examination of tables 8 and 9 shows that each of the operand relations are necessary to guarantee the detection of errors in the operands, especially when one is a constant (*B* in the tables). Table 8 shows that the relation $A = B$ is needed for the incorrect ($=, \neq, \geq, <$) expressions, and the relation $A = Succ(B)$ is needed for the incorrect ($>, \leq$) expressions. Table 9 shows that the relation $A = B$ is needed for the incorrect ($=, \neq, >, \leq$) expressions, and the relation $A = Pred(B)$ is needed for the incorrect ($\geq, <$) expressions.

4.3 DEFINITION EXTENSION.

This section gives a suggestion for how to incorporate the operator and operand extensions into the definition for MCDC. First, notice that the operand extensions given in section 4.1 satisfy the operator extensions given in section 4.2. This is demonstrated in table 10, where the equivalent relations are shown.

TABLE 10. OPERATOR RELATION EQUIVALENCE TO OPERAND RELATION

Operator Relation	Operand Relation
$A = B$	$A = B$
$A > B$	$A = Succ(B)$
$A < B$	$A = Pred(B)$

This suggests that the definition for MCDC given in section 3 could be extended by another clause, stating that when the independence tests are run for the relational operator condition, that the appropriate operand relations be used for the verification/test data per table 11.

To demonstrate how this would work, consider the expression *(A = B) and (C /= D) and E* where *(A, B, C, D)* are non-Booleans and *(E)* is Boolean. The coverage tests from the Boolean perspective are given in table 12. Notice that the rows are labeled with their condition codes. The independence tests for an AND operator are to submit all Trues, and then to cycle a single False through all the conditions.

TABLE 11. RELATIONAL OPERATOR INDEPENDENCE TEST DATA

Expression	True Result	False Result
$A = B$	$A = B$	$A = Succ(B), A = Pred(B)$
$A \neq B$	$A = Succ(B), A = Pred(B)$	$A = B$
$A > B$	$A = Succ(B)$	$A = B, A = Pred(B)$
$A \geq B$	$A = B, A = Succ(B)$	$A = Pred(B)$
$A < B$	$A = Pred(B)$	$A = B, A = Succ(B)$
$A \leq B$	$A = B, A = Pred(B)$	$A = Succ(B)$

TABLE 12. INDEPENDENCE TEST SET FOR *(A = B) and (C /= D) and E*

Test	*(A = B)*	*(C /= D)*	*E*
7	True	True	True
6	True	True	False
5	True	False	True
3	False	True	True

Expanding table 12 per table 11, results in table 13, where the required relations have been supplied between *(A, B)* and *(C, D)* in their respective columns. Notice that test 7 has been expanded into two tests: 7a and 7b. This is because the independence pair for *(C /= D)* is (tests 5 and 7), and test 7 is the one where the Pred and Succ relations return the required results. Test 3 was expanded to accommodate the relations for *(A = B)* for similar reasons.

TABLE 13. EXPANDED INDEPENDENCE TEST SET FOR *(A = B) and (C /= D) and E*

Test	*(A = B)*	*(C /= D)*	*E*
7a	$A = B$	$C = Succ(D)$	True
7b	$A = B$	$C = Pred(D)$	True
6	$A = B$	$C \neq D$	False
5	$A = B$	$C = D$	True
3a	$A = Succ(B)$	$C \neq D$	True
3b	$A = Pred(B)$	$C \neq D$	True

Notice that for tests 6 and 3, all that is required of the *(C /= D)* condition is that it be True. It does not matter what the difference between *C* and *D* is, only that there be one. The Pred and Succ relations could be used in these tests, but their usage would not satisfy the requirements of table 11 and would not prevent the expansion of test 7 into 7a and 7b. Similar reasoning applies to tests 6 and 5 and the *(A = B)* condition.

Notice that in moving from table 12 to table 13 two more tests have been added to the test set (one for each relational operator). From the Boolean perspective, these two new tests have not added anything to the verification. This is because tests 7a and 7b return the same Boolean results for the function and have identical Boolean condition values. The same applies to tests 3a and 3b. Whether these extended tests provide value commensurate with their cost is an issue that was not addressed during this study.

5. THEORETICAL COMPARISONS.

We will now look at the performance of the different forms of MCDC. This set of comparisons will consider only the theoretical side of things. It may be the case that, in practice, MCDC will always perform above these levels. However, it will always be the case that MCDC cannot perform below these levels. The first comparison concerns the minimum number of tests that the different forms of MCDC allows. The expectation is that the fewer tests that are required, the worse the criteria will perform in detecting errors. This will be the concern of the second comparison performed.

5.1 MINIMUM NUMBER OF TESTS VS EXPRESSION SIZE.

One of the major questions concerning any structural coverage criterion is *how many tests must be run to satisfy coverage?* A second related question concerns the benefits accrued for the costs of those tests. The second question is addressed in section 5.2. To determine a minimum number of tests, return to the definition for MCDC. MCDC requires a pair of tests for each condition, known as an independence pair. These two tests need to return a different value (one True and one False). This could be laid out graphically in an Independence Graph, as in figure 11. In an independence graph, the tests are the nodes of the graph, and the edge that connects them is labeled for the condition whose independence is shown by those two tests (i.e., condition *A*). The nodes are colored differently, indicating that they return different values. Which color represents True and which False is unimportant for this discussion. The independence graph is an adaptation of an n-cube representation for a Boolean function [5].

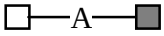


FIGURE 11. INDEPENDENCE GRAPH FOR CONDITION *A*

Figure 11 can now be extended for an additional condition *B* in one of two ways. In order to show *B*'s independence, two tests are needed to form its independence pair. One of the tests (at most) can be used from *A*'s independence pair and couple it with a new test for *B*. Notice that if another new test is not introduced, then both *A* and *B* are changing values between the same pair of tests, and therefore those two tests cannot form an independence pair for either condition. The two possible extensions for *B*'s independence are shown in figure 12.

FIGURE 12. INDEPENDENCE GRAPHS FOR ADDING CONDITION *B*

The two independence graphs in figure 12 can each be extended in three ways for an additional condition *C*. Either an additional dark node can be added, connected to an already existing light node, or vice-versa. The two existing nodes of the same color cannot be connected for condition *C*'s independence, as this would violate the rule for independence. One of the possible extensions for each of the independence graphs in figure 12 are shown in figure 13. In this case, *C* has been extended off of the center node in each of the graphs. The other possibilities would connect *C* with one of the outer nodes (and change the "*C* " node's color).

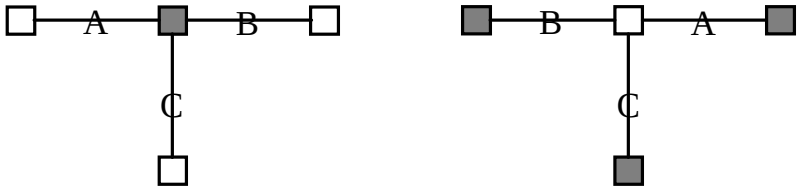


FIGURE 13. INDEPENDENCE GRAPHS FOR ADDING CONDITION *C*

What the proceeding has shown is that for up through three conditions, any form of MCDC will require a minimum of $N + 1$ tests, where N is the number of conditions. At this point, that will now change. In the preceding analyses, it was not important to know the states of the conditions at the nodes (i.e., their values). For the next step in the analysis, it will be important to know this because the rules for making transitions differ between the different forms of MCDC.

To begin the analysis for four and more conditions, first start with a discussion of graphically represented functions, starting with the *n-cube* representation [5]. The vertices of the *n-cube* represent the minterms of the function (i.e., the condition combinations), while the edges represent a transition between two vertices that differ in just one value. The *true vertices* (those for which the function returns True) are colored (dot added), while the *false vertices* are not. The edges between *true vertices* are also colored (heavier line). Figure 14 is a two-cube representation for the Boolean function which has the expression *not (A and B)*. The representation on the left side is the traditional mathematical representation, while the representation on the right side has been modified to use condition combinations. You may recognize that this is an alternate form of a KV-map [5].

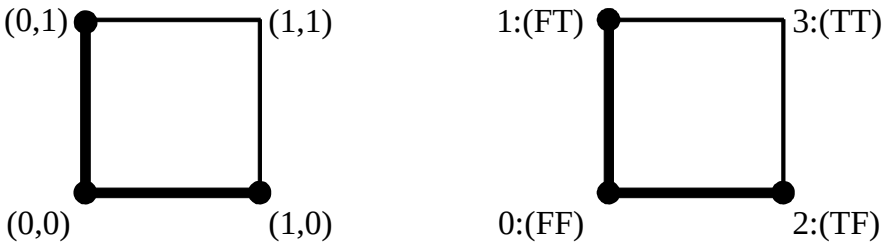


FIGURE 14. TWO-CUBE REPRESENTATION OF *not (A and B)*

To show the independence of either of the conditions will require that the truth vector 3:F-tt be one of the members of the independence pair as that is the only condition combination for which this function returns False. To show the independence of *A*, change between the condition combination 3:(TT) to the combination 1:(FT) and have the function return True for the latter combination. This is possible as the two-cube representation of this expression shows (figure 14). To show the independence of *B*, change between the condition combination 3:(TT) to the combination 2:(TF) and have the function return True for the latter combination. This is possible as the two-cube representation of this expression shows (figure 14). For this function, the change between 3:(TT) to 0:(FF) is not allowed to show independence. This is because both conditions change value, so neither one can be said to have shown independent effect.

Now consider a function with three conditions. Figure 15 is a three-cube representation for the Boolean function with the expression *A and (B or C)*.

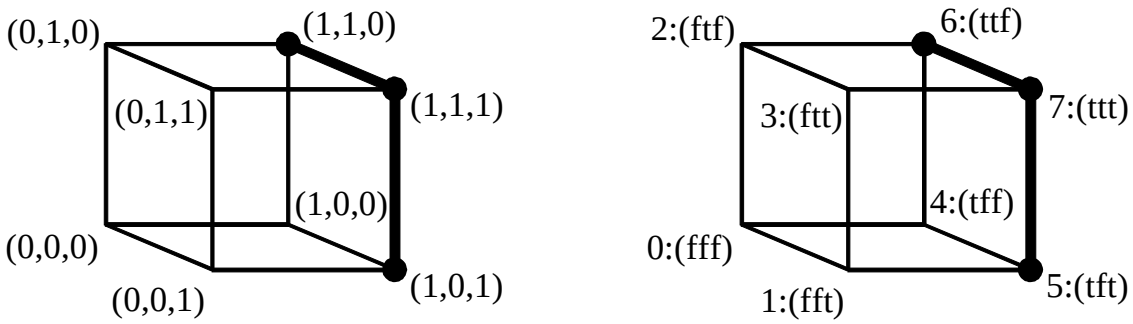


FIGURE 15. THREE-CUBE REPRESENTATION OF *A and (B or C)*

The independence analysis for the expression *A and (B or C)* is presented in table 14 (tree analysis left as an exercise for the reader). The first column lists the condition under consideration, the second column identifies one of the independence pairs for the condition, and the final column identifies the transition type. The first type of transition is along an edge of the three-cube. This is the only transition allowed by Unique-Cause MCDC (by definition, only the condition of interest is allowed to change). The second type of transition is a diagonal (i.e., not along an edge). These transitions are allowed by Masking MCDC. There are two types of diagonals, one that travels along a cube face (e.g., 1 to 7), and one that travels internally between opposite corners of the cube (e.g., 1 to 6). Each of these different types is identified. Notice that there is at least one edge transition independence pair for each condition, so there is a Unique-Cause MCDC coverage set for this expression. Notice that there is at least one independence pair (of any transition type) for each of the conditions, so there is a Masking MCDC coverage set for this expression. Also notice that there are diagonal transitions for condition *A*, so there are more coverage sets for Masking MCDC than there are for Unique-Cause MCDC (remember that one independence pair is needed for each condition to form a coverage set).

TABLE 14. INDEPENDENCE ANALYSIS FOR *A and (B or C)*

Condition	Independence Pair	Transition Type
A	(1,5)	Edge
	(1,6)	Internal Diagonal
	(1,7)	Face Diagonal
	(2,5)	Internal Diagonal
	(2,6)	Edge
	(2,7)	Face Diagonal
	(3,5)	Face Diagonal
	(3,6)	Face Diagonal
	(3,7)	Edge
B	(4,6)	Edge
C	(4,5)	Edge

Two independence graphs are presented in figure 16. The graph on the left is for a Unique-Cause MCDC coverage set, while the one on the right is for Masking MCDC. The nodes have been annotated with the condition codes. Notice that in the Unique-Cause graph, only a single condition changes between the nodes (as expected). With the Masking graph, the transition from 6 to 1 changes all three conditions. This is allowed because the changes in conditions *(B, C)* are masked at the *or* operator (i.e., the *or* remained True).

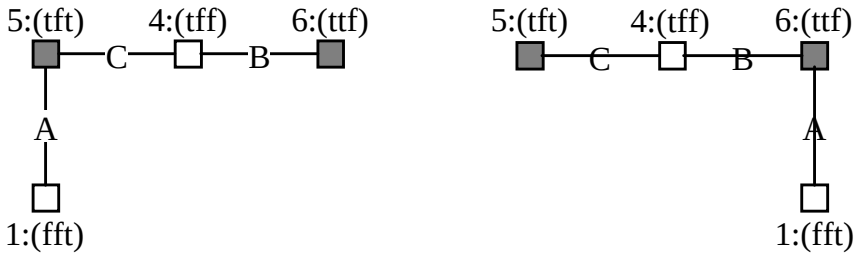


FIGURE 16. INDEPENDENCE GRAPHS FOR ADDING CONDITION *C*

Theorem 1. If a coverage set exists for an expression with *N* unique conditions with a single occurrence each, then both *Unique-Cause MCDC* and *Unique-Cause + Masking MCDC* require a minimum of *N* + 1 tests. See section 7 for a discussion of why a coverage set may not exist for certain expressions.

To understand why this is so, use a modified form of the independence graph as shown in figures 17 through 20. The modified independence graph annotates the nodes in a different manner than previously used. In this case, annotate the nodes with an indication of which condition changed in the transition from the previous/following truth vector. Figure 17 shows the annotated independence graph for a single condition (*A*). For this annotation, pick one of the nodes to be a baseline and annotate all the conditions with a 0 subscript. Then when the transition is made for that condition which shows its independence, change the annotation to a 1 subscript. Figure 17 is annotated two ways to show that it does not matter which node to start with, nor which color to start with.

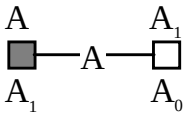


FIGURE 17. ANNOTATED INDEPENDENCE GRAPH FOR A SINGLE CONDITION (*A*)

Figure 18 shows the annotated independence graph for two conditions (*A, B*). As with figure 17, this graph is also annotated two ways to show that it does not matter which node to start with (outer, inner), nor which way the independence chain was built (*(A, B)* off of base, *(A, B)* off of *(B, A)*).

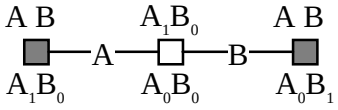


FIGURE 18. ANNOTATED INDEPENDENCE GRAPH FOR TWO CONDITIONS (*A, B*)

Figure 19 shows two annotated independence graphs for three conditions (*A, B, C*). The two different styles of graphs show again that it does not matter which node is used first to start with, nor how the independence chain was built.

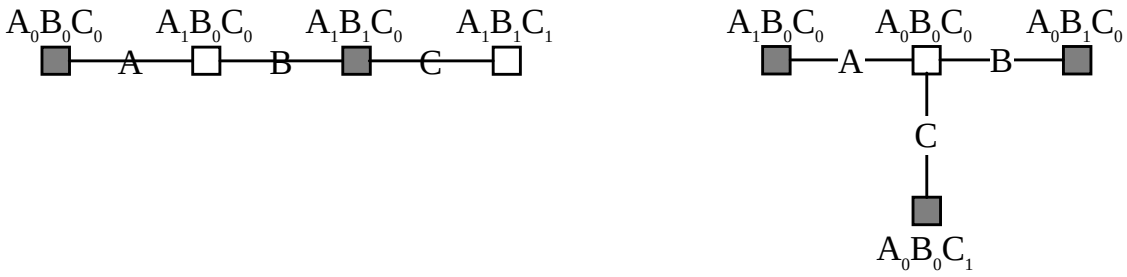


FIGURE 19. ANNOTATED INDEPENDENCE GRAPHS FOR THREE CONDITIONS (*A, B, C*)

Finally, figure 20 shows one annotated independence graph for four conditions *(A, B, C, D)*. These graphs have shown that starting with each condition annotated with a zero subscript, sufficient tests are needed to change the subscript from a 0 to a 1. It is also known that this subscript change will occur only once for each condition (assuming there is an independence pair for the condition). Given that *N* annotation changes are needed for *N* conditions, plus the base case of all 0 annotations are needed, this requires a minimum of *N + 1* tests.

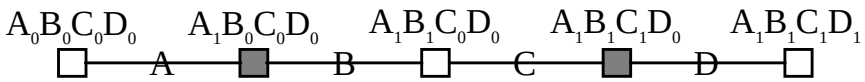


FIGURE 20. INDEPENDENCE GRAPH FOR FOUR CONDITIONS (*A, B, C, D*)

A way to paraphrase the above is to consider that the smallest test set can be constructed if every condition past the first one is able to add a single test to the existing set. Since the first condition required two tests for its independence, and each condition is adding exactly one new test to the set, this results in *N + 1* tests for *N* conditions.

Theorem 2. If a coverage set exists for an expression with *N* unique conditions with *M* total occurrences, then *Unique-Cause + Masking MCDC* requires a minimum of *M* + 1 tests. See section 7 for a discussion of why a coverage set may not exist for certain expressions.

To understand why this is so, rewrite the expression in terms of *M* unique conditions and then follow the argument for *N* conditions each with a unique occurrence given previously. The fact that some of these conditions will be strongly coupled (due to being identical) has no effect on the size of the minimum coverage set. However, coupling does have an effect on whether a coverage set exists, and whether the theoretical minimum is achievable.

Theorem 3. If a coverage set exists for an expression with N unique conditions with a single occurrence each, then Masking MCDC requires a minimum of *RUTW(2*SQRT(N))* tests, where *RUTW* stands for round up to whole. See section 7 for a discussion of why a coverage set may

not exist for certain expressions. The rounding up is necessary when the quantity *2*SQRT(N)* is not a whole number, since a fractional number of tests is not possible.

To understand why this is so, consider that a minimum number of tests can be achieved if every node in an independence graph can have an arc to every node in that graph in a different color. Figure 21 shows two such independence graphs, one with four conditions and four tests, and one with six conditions and five tests.

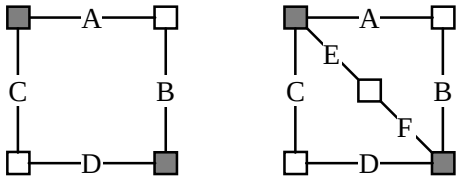


FIGURE 21. INDEPENDENCE GRAPHS SHOWING MINIMUM NUMBER OF TESTS

The minimum number of tests for the maximum number of conditions can be achieved when the independence graph consists of an equal number of nodes of each color, with each pair of different colored nodes supporting the independence of a condition. This is known as a perfect square, and some examples are shown in figure 22. These independence graphs have been formatted differently from those used previously. Here the nodes are shown in a matrix, with the dark nodes forming the columns and the light nodes forming the rows. The condition whose independence is shown by a pair of dissimilar colored nodes is shown at the intersection of the two node's column and row.

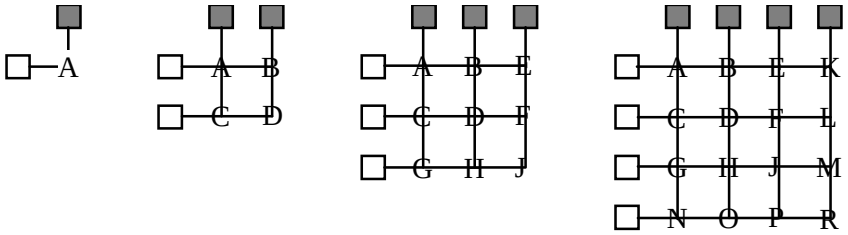


FIGURE 22. INDEPENDENCE GRAPHS SHOWING PERFECT SQUARES

Figure 22 proceeds from two tests (for one condition), to four tests (for four conditions), to six tests (for nine conditions), and finally to eight tests (for sixteen conditions). In every case, when another new test for a new condition is needed (from two tests for just A, to three tests for (A, B)), first extend the dark-colored nodes first, followed by the light-colored nodes. In all cases, the maximum number of conditions is achieved when the square is filled. For example, consider the extension from two conditions (A, B) to three conditions (A, B, C) depicted in figure 23. If another dark node is added to the graph for condition C, four tests would have been used to cover three conditions (as with the Unique-Cause form of MCDC). By adding a second light node to the graph, four tests are still used to cover three conditions, but there is a space in the independence graph to add a fourth condition without having to add another test.

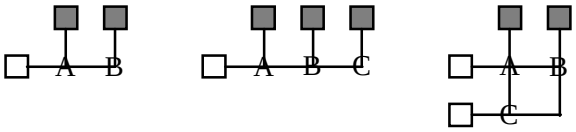


FIGURE 23. INDEPENDENCE GRAPHS FOR A TWO- TO THREE-CONDITION EXTENSION

Achieving maximum node cover with minimum tests being a perfect square means that the number of nodes of one color should equal the square root of the number of conditions. Since an equal number of the other colored nodes are needed to form a perfect square, this leads to 2**SQRT(N)* tests for *N* conditions when *N* is expressible as a perfect square. What about the case when *N* is not a perfect square? To deal with this case, consider the extension from six to seven conditions depicted in figure 24.

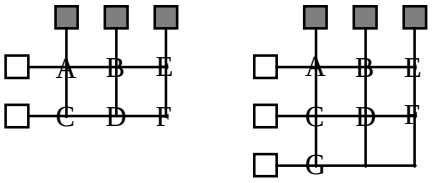


FIGURE 24. INDEPENDENCE GRAPHS FOR SIX AND SEVEN CONDITIONS

For six conditions, the above formula leads to 2**SQRT*(6) = 4.90 tests. There cannot be 0.90 tests, since tests are discrete things. Now if this number is rounded to 5, the correct number of tests can be obtained. However, for seven conditions, the formula leads to 2**SQRT*(7) = 5.29 tests. In this instance, rounding this to 5 tests gets an incorrect answer, as the graphs in figure 24 show. In this instance, rounding up to the nearest whole number, which is 6, is required.

Theorem 4. If a coverage set exists for an expression with *N* unique conditions with *M* total occurrences, then Masking MCDC requires a minimum of *RUTW(2*SQRT(M))* tests. See section 7 for a discussion of why a coverage set may not exist for certain expressions.

To understand why this is so, rewrite the expression in terms of *M* unique conditions and then follow the argument for *N* conditions each with a unique occurrence given previously. The fact that some of these conditions will be strongly coupled (due to being identical) has no effect on the size of the minimum coverage set. However, coupling does have an effect on whether a coverage set exists, and whether the theoretical minimum is achievable.

Two equations have now been developed that predict the minimum number of tests for each of the forms of MCDC. Whether there are expressions which can actually realize this number is investigated later in this report. For now though, it is assumed that there are expressions that can realize these minimum numbers. Figure 25 plots the performance of these two equations for expressions consisting of 1 through 76 conditions. Since the number of tests is the closest thing this study has for costs, examination of the two curves shows that Masking MCDC will be much cheaper for larger expressions than either of the other forms (especially for 76 conditions, 18 versus 77 tests).

The reason for the stepping in the Masking MCDC curve in figure 25 is because of the number of conditions which can be covered by adding a new test to the perfect square. As figure 24 shows, when a new test was added for seven conditions, up to nine conditions was also covered for that same number of tests. This means that (seven, eight, and nine) conditions have the same number of tests (as the curve shows). Notice that the steps get longer with the addition of more conditions. This is because the size of the square of tests is getting longer the further out in condition space the test goes.

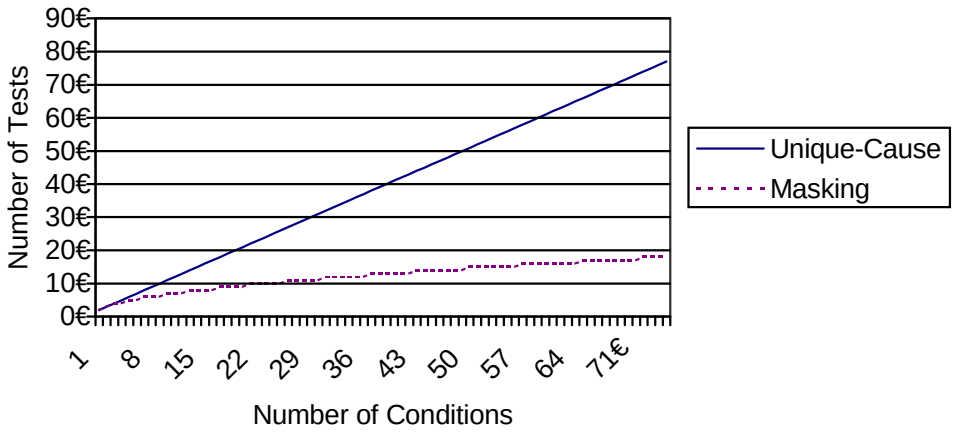


FIGURE 25. MINIMUM NUMBER OF TESTS VS NUMBER OF CONDITIONS

5.2 PROBABILITY OF LOGIC ERROR DETECTION.

As mentioned previously, another of the major questions concerning any structural coverage criterion is *what are the benefits accrued for the costs of the tests run to satisfy coverage?* Two approaches can be taken to provide an answer to that question. In the first approach, investigated here, the minimum probability of detecting logic errors will be discussed. These logic errors are abstracted to the ability of the MCDC test set to detect if an incorrect Boolean function has been implemented. The second approach, which this study was unable to fully address, would determine the mutation sensitivity of the different forms of MCDC applied to the expressions in appendix C.

The sensitivity of the different forms of MCDC to errors in the implementation of a Boolean function can be quantified in the following manner. For a Boolean function of N conditions there are 2^N possible combinations of the condition values. Table 15 shows the possible condition combinations when there are two conditions. Notice that the combinations are identified with the corresponding condition code numbers.

TABLE 15. POSSIBLE COMBINATIONS FOR TWO CONDITIONS

	$Combination_0$	$Combination_1$	$Combination_2$	$Combination_3$
$Condition_1$	False	False	True	True
$Condition_2$	False	True	False	True

For each of these condition combinations, there is the possibility of two responses (False, True) since we are talking about Boolean functions. The responses to the condition combinations can be used to identify the Boolean function (i.e., used as condition codes). For these *M* combinations, ($M = 2^N$) there are 2^M possible responses (in essence, there is a higher order Boolean function for the *M* conditions). This means that for *N* conditions, there are 2^{2^N} possible Boolean functions [5]. Table 16 shows the possible functions (response profiles) for two conditions, and their respective combinations.

Recall that for a function of *N* operands, there are 2^N possible tests (i.e., combinations of the condition values) which can be run against the function. In order to completely distinguish the function wanted from all of the other possible functions (2^{2^N} - 1), all 2^N tests need to be run. When *N* starts to get large, this becomes impractical (i.e., this becomes multiple-condition coverage [6]). When less than 2^N tests are run, the function cannot be distinguished from what is wanted from some other number of functions. This number of undistinguished functions can be described by the number of functions allowed by the combinations that have not been tested. This is described as follows: given *M* tests, then 2^N- *M* combinations were not tested. This leaves $2^{\left(2^N - M\right)}$ functions that were not distinguished by the tests. Since one of those functions is the one that is wanted, this means that for a given function of *N* operands and any *M* distinct tests there are $2^{\left(2^N - M\right)} - 1$ other functions that are indistinguishable, i.e., produce the same outcome, for the *M* tests.

TABLE 16. POSSIBLE BOOLEAN FUNCTIONS FOR TWO CONDITIONS

	$Combination_0$	$Combination_1$	$Combination_2$	$Combination_3$
$Function_0$	False	False	False	False
$Function_1$	False	False	False	True
$Function_2$	False	False	True	False
$Function_3$	False	False	True	True
$Function_4$	False	True	False	False
$Function_5$	False	True	False	True
$Function_6$	False	True	True	False
$Function_7$	False	True	True	True
$Function_8$	True	False	False	False
$Function_9$	True	False	False	True
$Function_{10}$	True	False	True	False
$Function_{11}$	True	False	True	True
$Function_{12}$	True	True	False	False
$Function_{13}$	True	True	False	True
$Function_{14}$	True	True	True	False
$Function_{15}$	True	True	True	True

For example, if run ($Combination_0$, $Combination_1$) gets the response (False, False), then the data in table 16 shows that it cannot tell which of ($Function_0$, $Function_1$, $Function_2$, $Function_3$) are

being tested. If $Function_1$, is wanted, then there are three other functions that the tests have failed to distinguish between. This means that given any *M* distinct tests, the probability $P_{(N,M)}$ of detecting an error in an incorrect implementation of a Boolean expression with *N* conditions is given by

$$P_{(N,M)} = 1 - \left[\frac{2\,(2^N - M)\ - 1}{2\,2^N} \right]$$

It is important to remember that this probability applies to the detection of an incorrect Boolean function. This mainly applies to the placement of Boolean operators and operands (i.e., conditions). If some of the conditions are composed of relational operators between non-Booleans, this equation does not apply to the discovery of errors in those relational (sub)expressions. This means that given the expression *(X ≥ 0) and (X ≤ 100)*, this probability applies to the detection of the Boolean function *A and B*. In section 4, extensions for the relational operators were suggested.

If the number of tests *M* is fixed at *N* + 1 (*N* being the number of conditions), the probability of distinguishing between incorrect functions grows exponentially with *N*, *N* > 3. If *M* is fixed at *RUTW*(2**SQRT(N)*), the probability grows in a jumpier progression, representing the steps in the minimum number of tests discussed previously. Figure 26 shows the growth of both of these curves for 1 through 32 conditions.

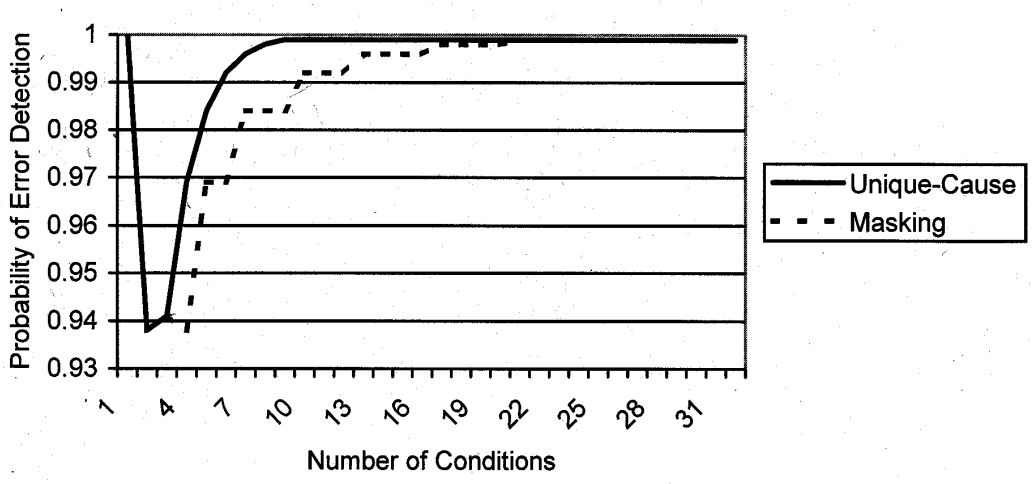


FIGURE 26. MINIMUM PROBABILITY OF LOGIC ERROR DETECTION VS NUMBER OF CONDITIONS—MCDC

Examination of figure 26 shows that the difference between the Unique-Cause approach and the Masking approach to MCDC is not that great for the detection of incorrect Boolean functions. This is especially true when the number of conditions is large.

In figure 27, the same analysis is used to compare the two forms of MCDC to Statement and Decision Coverage. For this analysis, the Statement Coverage will require a minimum of one test when the Boolean expression occurs in a branchpoint. Hence, one test is able to be used as

the lower bound for the performance of Statement Coverage. Because *M* is fixed at 1, the probability of distinguishing between incorrect functions drops (as opposed to grows) exponentially and asymptotically approaches 0.50.

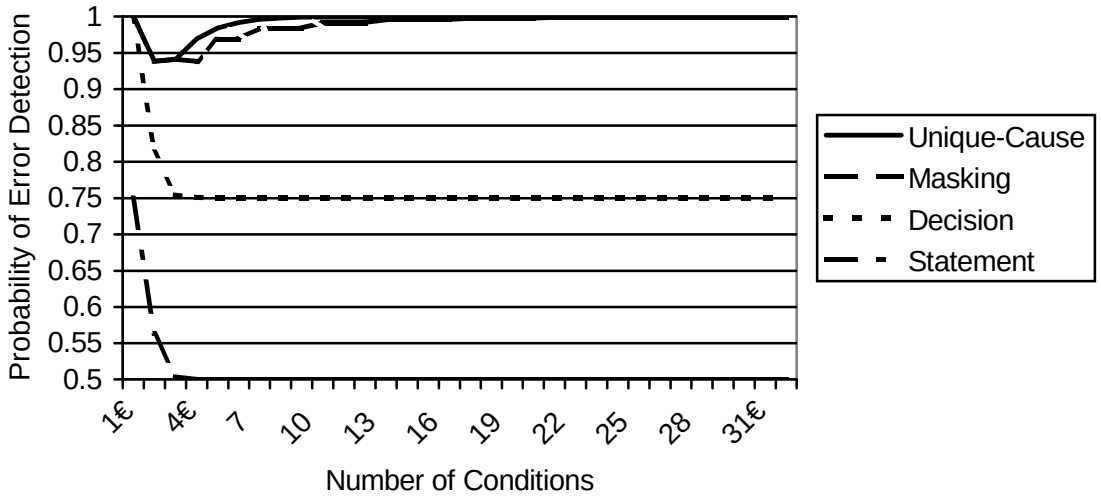


FIGURE 27. MINIMUM PROBABILITY OF LOGIC ERROR DETECTION VS NUMBER OF CONDITIONS—ALL COVERAGE LEVELS

It is also known that Decision Coverage will require a minimum of two tests when the Boolean expression occurs in a branchpoint. Hence, two tests are able to be used as the lower bound for the performance of Decision Coverage. Because *M* is fixed at 2, the probability of distinguishing between incorrect functions drops (as opposed to grows) exponentially and asymptotically approaches 0.75.

Examination of figure 27 shows that either form of MCDC performs significantly better than either Statement Coverage or Decision Coverage. The differences between the two forms of MCDC pale by comparison to the differences between Decision Coverage and Statement Coverage.

It is important to note that the performance of MCDC in this model is based only on the *number* of tests, not on *which* tests were run. This means that any testing which applies $N + 1$ (Boolean) tests to a Boolean expression will have the performance given in figures 26 and 27. Recall from appendix B that MCDC is concentrating on visiting each side of a partitioning plane in a significant way, which is a form of equivalence class partitioning and boundary value analysis. It is unknown if the MCDC selection rules would perform significantly better on real expressions with real errors than just randomly chosen tests of the same number. To help resolve this issue would require either an error analysis of a real development or a mutation analysis, both of which this study was unable to address.

6. EMPIRICAL COMPARISONS.

In section 5, comparisons were considered between the different forms of MCDC in an abstract theoretical sense. That analysis presented what should be the worst case since the assumption that all Boolean functions could be realized by an expression allowing the minimum number of tests is known to be false. As the probability of error detection equation showed, the smaller the number of tests, the lower the probability of error detection. Therefore, the minimum number of tests always determines the minimum probability of error detection.

In this section, comparisons were performed between the different forms of MCDC using actual logic expressions from airborne software (appendix C). These comparisons provide data to allow for a choice between the different forms of MCDC. The comparisons that were performed, are

- The minimum number of tests in a coverage test set versus expression size (section 6.1) (number of independent conditions). This comparison is used to confirm the theoretical analysis in section 5.1.

- The probability of error detection (section 6.2) given the minimum coverage test set versus expression size derived in section 6.1. This comparison is used to confirm the theoretical analysis in section 5.2.

- The average number of independence pairs (section 6.3) per condition versus expression size. This comparison is used to determine one aspect of the cost-effectiveness of the different forms of MCDC. It is assumed that the larger the number of independence pairs per condition, the easier the attainment of coverage will be.

- The average number of tests in a minimal coverage test set (section 6.4) versus expression size. This comparison is used to determine one aspect of the cost-effectiveness of the different forms of MCDC. It is assumed that the smaller the minimal coverage test set, the easier the attainment of coverage will be. However, this smaller test set also carries with it a lower probability of error detection.

- The average number of minimal test sets (section 6.5) versus expression size. This comparison is used to determine one aspect of the cost-effectiveness of the different forms of MCDC. It is assumed that the larger the number of acceptable test sets, the easier the attainment of coverage will be.

Because of the analysis methods used in some of these comparisons, they are limited to expressions with conditions one through six. These analyses are performed in two different ways: context free and context dependent. The context free analysis assumes that all condition combinations are possible (i.e., that there is no coupling) while the context dependent analysis uses only the feasible condition combinations.

In order to perform the first form of analysis, the expressions from appendix C were abstracted into their pure Boolean form (e.g., the expression *(X ≥ 0) and (X ≤ 100)* becomes the expression *(A and B)* for this analysis). This form of the analysis ignored coupling since coupling will never

decrease the size of a test set if one exists, only increase it. The use of the smaller-sized (though infeasible) test set provides a worst-case analysis.

The second form of analysis takes into account the coupling in the original expressions (e.g., the expression *(X ≥ 0) and (X ≤ 100)* does not allow for the condition combination 0:FF since it is not possible for *X* to be negative (< 0) and greater than 100 (> 100) simultaneously).

6.1 MINIMUM NUMBER OF TESTS VS EXPRESSION SIZE.

The minimum number of tests allowed by the encodings of Boolean expressions will be computed from the logic expressions in appendix C. These expressions, both in their pure Boolean form as well as their context-coupled forms, were exhaustively examined (i.e., all permutations and combinations of condition combinations were computed) to determine the absolute smallest test set that would satisfy each of the MCDC definitions. This analysis confirms the theoretical analysis in section 5.1.

6.1.1 Boolean (Context Free) Analysis.

Figure 28 plots the data for Unique-Cause MCDC for one through six conditions. For those expressions for which at least one test set exists, the number of expressions with that number of tests as the minimum test set size is shown beside each dot (e.g., for 4 conditions, there are 382 expressions that require 5 tests and 2 expressions that require 7 tests). For those expressions for which there is no test set possible, the number of expressions is also shown beside each dot (e.g., for four conditions, there are seven expressions that have no coverage set). See section 7 for a discussion of why a coverage set may not exist for certain expressions.

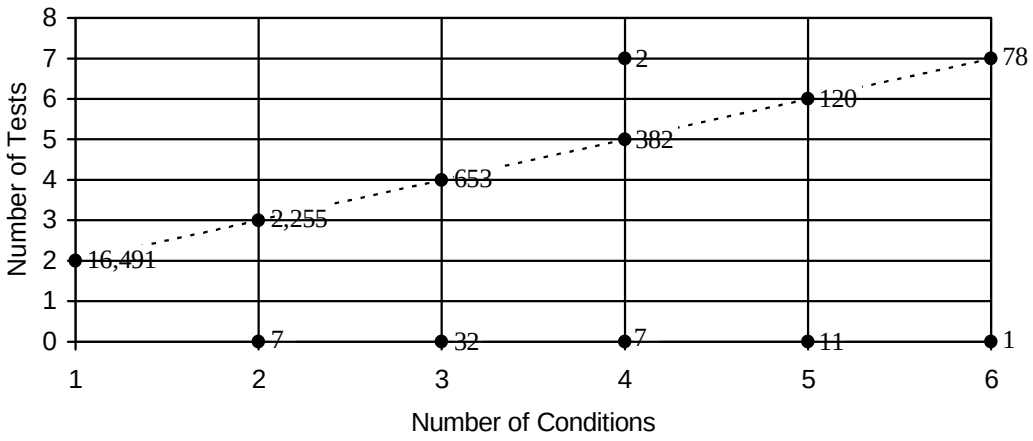


FIGURE 28. MINIMUM NUMBER OF TESTS VS NUMBER OF CONDITIONS—UNIQUE-CAUSE (CONTEXT FREE)

Notice that for four conditions, there are two expressions that require more tests than the theoretical minimum. These two expressions utilized short-circuit forms with coupled conditions. Normally, Unique-Cause MCDC would not be able to handle these type of expressions (repeated conditions). However, in this instance the short-circuit forms masked the

cases when the repeated conditions both change value at the same time (i.e., only one occurrence of the repeated condition was evaluatable at a time, therefore the identical coupling does not show up in the independence analysis). Other than these two anomalies, when a coverage set exists, it requires exactly $N + 1$ tests (as is indicated by the majority of the data points being on the $N + 1$ line).

Figure 29 plots the data for Unique-Cause + Masking MCDC for one through six conditions. Notice that moving from Unique-Cause to Unique-Cause + Masking allowed for a few more expressions to have a coverage set. Also notice that this move did not allow for all expressions to have a coverage set. See section 7 for a discussion of why a coverage set may not exist for certain expressions. The other thing to notice about this data is that in all cases where Unique-Cause + Masking allows for a coverage set where Unique-Cause does not, the new coverage sets are always $N + 1$ or larger (with the majority being larger).

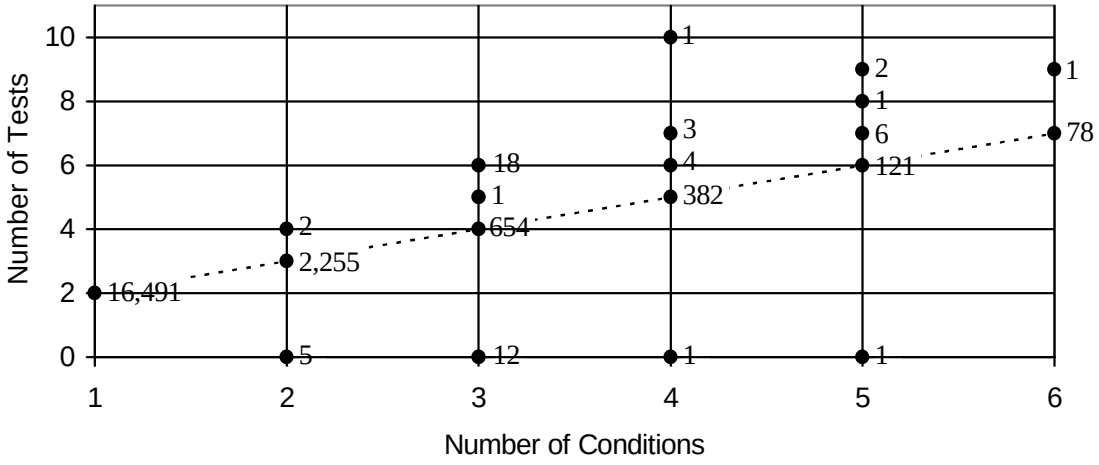


FIGURE 29. MINIMUM NUMBER OF TESTS VS NUMBER OF CONDITIONS—UNIQUE-CAUSE + MASKING (CONTEXT FREE)

Figure 30 plots the data for Masking MCDC for one through six conditions. Notice that moving from Unique-Cause + Masking to Masking allowed for a few more expressions to have a coverage set. Also notice that this move still did not allow for all expressions to have a coverage set. See section 7 for a discussion of why a coverage set may not exist for certain expressions. Another thing to notice about this data is that in all cases where Masking allows for a coverage set where Unique-Cause + Masking does not, the new coverage sets are always $N + 1$ or larger (with the majority being larger). Finally, the final thing to notice about this data is that the majority of test sets are still of size $N + 1$. A few expressions did go below the $N + 1$ line, but not very many (85 of 20,030). None of the coverage sets went below the *RUTW*(2**SQRT(N)*) line.

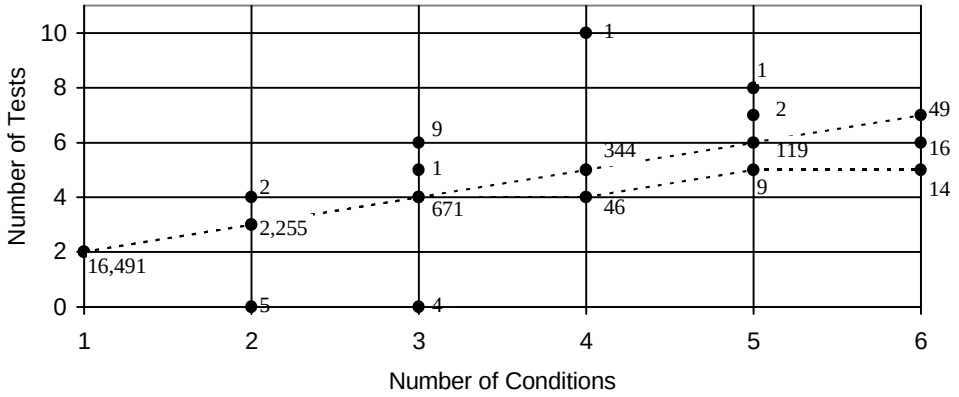


FIGURE 30. MINIMUM NUMBER OF TESTS VS NUMBER OF CONDITIONS—MASKING (CONTEXT FREE)

6.1.2 Coupling (Context Dependent) Analysis.

Figure 31 plots the data for Unique-Cause MCDC for one through six conditions. Comparing the data in figures 28 (context free) and 31 (context dependent) show that bringing in the context information into the Unique-Cause MCDC analysis did not change the picture very much. Only a single expression at the four-condition level needed more tests when context dependency was brought in. Surprisingly, the context information did not cause any new expressions to not have a coverage set.

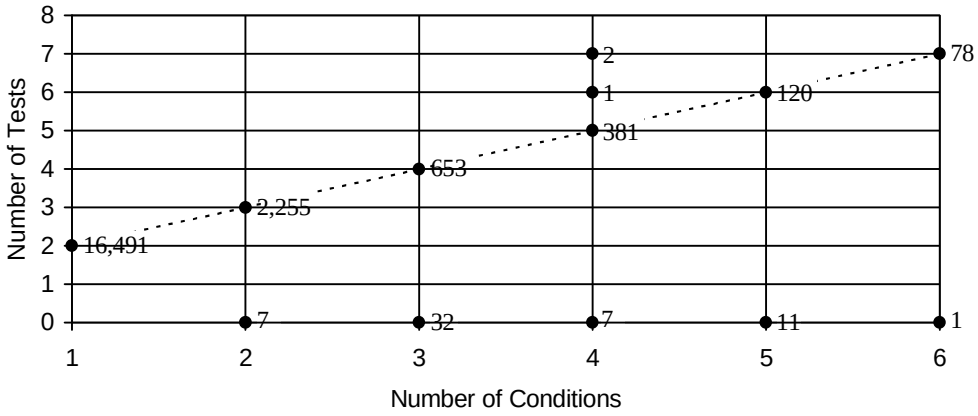


FIGURE 31. MINIMUM NUMBER OF TESTS VS NUMBER OF CONDITIONS—UNIQUE-CAUSE (CONTEXT DEPENDENT)

Figure 32 plots the data for Unique-Cause + Masking MCDC for one through six conditions. Comparing the data in figures 29 (context free) and 32 (context dependent) shows that bringing in the context information changes the picture more for Unique-Cause + Masking than it did for Unique-Cause (compare figures 28 and 31). In particular, there is one expression each at the three-, four-, and five-condition levels that do not have coverage sets in the context dependent analysis that did have coverage sets in the context free analysis. This is because the independence pairs that would have allowed coverage are not feasible in the implementation.

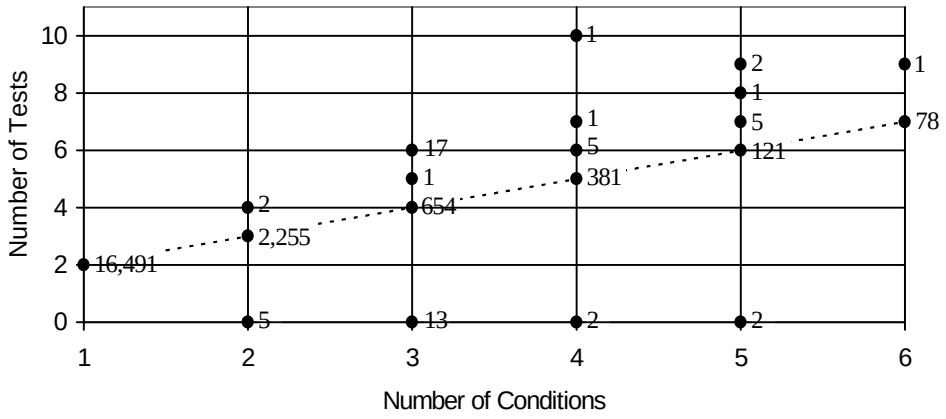


FIGURE 32. MINIMUM NUMBER OF TESTS VS NUMBER OF CONDITIONS—UNIQUE-CAUSE + MASKING (CONTEXT DEPENDENT)

Taking a closer look at the four-condition level between figure 29 and figure 32, one expression which required seven tests in the context free form has no solutions in the context dependent form. This decreased the number of expressions at (4,7) and increased the expressions at (4,0). Another expression at the four-condition level required five tests when context free, but now requires six tests. This decreased the expressions at (4,5) and increased the expressions at (4,6). This is because the independence pairs that allowed the smaller test set are infeasible.

Comparing figures 31 and 32, once again it can be seen that Unique-Cause + Masking allows for more expressions to have coverage sets than Unique-Cause does. In addition, each of the expressions that have coverage under Unique-Cause + Masking has $N + 1$ or greater tests.

Figure 33 plots the data for Masking MCDC for one through six conditions. Comparing figure 30 (context free) with figure 33 (context dependent) once again demonstrates that the context information made some expressions no longer have a coverage set (see three and four conditions), while increasing the size of others (see six conditions).

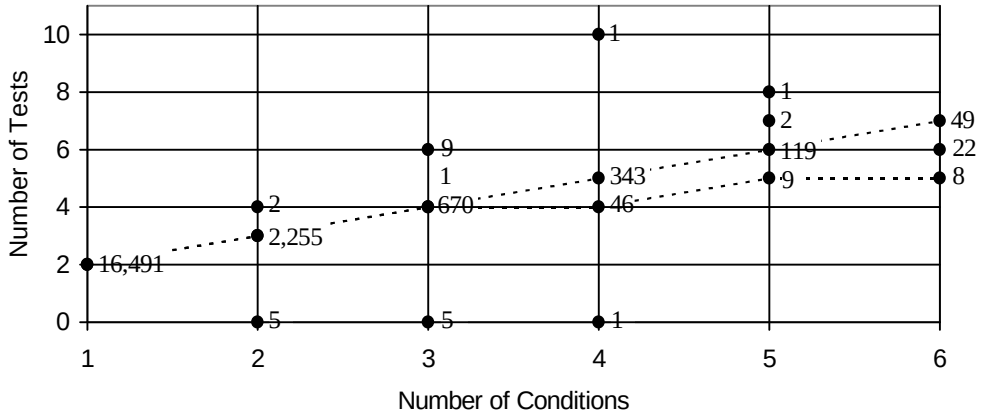


FIGURE 33. MINIMUM NUMBER OF TESTS VS NUMBER OF CONDITIONS—MASKING (CONTEXT DEPENDENT)

Once again, comparing figures 32 and 33 shows that moving from Unique-Cause + Masking to Masking allowed for a few more expressions to have a coverage set, though still not all. Notice that in all cases where Masking allows for a coverage set where Unique-Cause + Masking does not, the new coverage sets are always $N + 1$ or larger (with the majority being larger). Finally, the final thing to notice about this data is that the majority of test sets are still of size $N + 1$. A few expressions did go below the $N + 1$ line, but not very many (85 of 20,028). None of the coverage sets went below the *RUTW(2*SQRT(N))* line.

6.1.3 Comparisons Across All Forms.

This section presents the data in figures 28 through 33 in tabular form in order to compare and contrast what is happening with the expression forms. In tables 17 through 22, the following data is provided. The first column identifies the number of expressions that have the signature of the number of tests for each form of MCDC. The next three columns (columns two through four) identify the minimum number of tests needed to satisfy the three forms of MCDC when the expressions are considered in their pure Boolean context free form. Column two is for Unique-Cause, column three is for Unique-Cause + Masking, and column four is for Masking. The next three columns (columns five through seven) identify the minimum number of tests needed to satisfy the three forms of MCDC when the expressions are considered in context dependent form. Column five is for Unique-Cause, column six is for Unique-Cause + Masking, and column seven is for Masking.

Table 17 presents the data for the single condition expressions. As expected, all expressions require two tests to satisfy any form of MCDC. Recall that at this expression level, all forms of MCDC are equivalent and require exhaustive testing (both tests) for coverage.

TABLE 17. TEST SET SIZE FOR ONE-CONDITION EXPRESSIONS

Number of Expressions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
16,491	2	2	2	2	2	2

Table 18 presents the data for the two-condition expressions. Here there are some definite differences from the single-condition expressions. First off, notice that there are five expressions for which there are no coverage sets under any of the interpretations of MCDC. Analysis for some of these expressions is provided in section 7.1. There are also two expressions for which there are no coverage solutions for Unique-Cause MCDC, but for which there are solutions for the other interpretations. These are expressions which have (identically) coupled (i.e., repeated) conditions. It is interesting to note that for these expressions, exhaustive testing was needed to achieve coverage. Finally, the vast majority of expressions require three tests for coverage, no matter what the interpretation for MCDC is. As the analysis in sections 6.3 and 6.5 shows, some of these expressions have larger numbers of independence pairs and coverage sets under the Masking interpretation than under the Unique-Cause interpretation.

TABLE 18. TEST SET SIZE FOR TWO-CONDITION EXPRESSIONS

Number of Expressions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
5	0	0	0	0	0	0
2	0	4	4	0	4	4
2,255	3	3	3	3	3	3

Table 19 presents the data for the three-condition expressions. Here there are some definite differences from the single-condition expressions. As with the two-condition analysis, there are expressions for which there is no coverage set under any interpretation of MCDC. There are also expressions that are only MCDC testable under the Masking interpretation. Finally, as was pointed out earlier in the discussion for figure 32, there is one expression which has MCDC coverage sets under the context free analysis, but does not when the context dependent information is factored into the analysis. Again, as was the case previously, the vast majority of the expressions require $N + 1$ tests to achieve coverage under any interpretation of MCDC.

TABLE 19. TEST SET SIZE FOR THREE-CONDITION EXPRESSIONS

Number of Expressions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
4	0	0	0	0	0	0
8	0	0	4	0	0	4
1	0	4	4	0	4	4
1	0	5	5	0	5	5
1	0	6	4	0	0	0
8	0	6	4	0	6	4
9	0	6	6	0	6	6
653	4	4	4	4	4	4

Table 20 presents the data for the four-condition expressions. Unlike what has been seen previously, there is a context-free masking coverage set for all of these expressions. As has been seen previously, there are expressions that have coverage sets under the context-free analysis but have none under the context-dependent analysis. At this level we also see that there are expressions for which there is a smaller test set under the context-free analysis than there is under the context-dependent analysis. This occurred for the Unique-Cause and the Unique-Cause + Masking interpretations for MCDC. Finally, there are expressions that require more than $N + 1$ tests for coverage under both of the unique interpretations. Again, as was the case previously, the vast majority of the expressions require $N + 1$ tests to achieve coverage under any interpretation of MCDC.

Table 21 presents the data for the five-condition expressions. As with the four-condition expressions, there is a context-free Masking coverage set for all of these expressions. There is also a context-dependent Masking coverage set for all expressions. Again, as was the case

previously, the vast majority of the expressions require $N + 1$ tests to achieve coverage under any interpretation of MCDC.

TABLE 20. TEST SET SIZE FOR FOUR-CONDITION EXPRESSIONS

Number of Expressions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
1	0	0	5	0	0	5
4	0	6	5	0	6	5
1	0	7	5	0	0	0
1	0	10	10	0	10	10
45	5	5	4	5	5	4
1	5	5	4	6	6	4
336	5	5	5	5	5	5
2	7	7	5	7	7	5

TABLE 21. TEST SET SIZE FOR FIVE-CONDITION EXPRESSIONS

Number of Expressions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
1	0	0	6	0	0	6
1	0	6	6	0	6	6
1	0	7	6	0	0	6
5	0	7	6	0	7	6
1	0	8	8	0	8	8
2	0	9	7	0	9	7
9	6	6	5	6	6	5
111	6	6	6	6	6	6

Table 22 presents the data for the six-condition expressions. Again, as was the case previously, the vast majority of the expressions require $N + 1$ tests to achieve coverage under any interpretation of MCDC.

TABLE 22. TEST SET SIZE FOR SIX-CONDITION EXPRESSIONS

Number of Expressions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
1	0	9	6	0	9	6
8	7	7	5	7	7	5
6	7	7	5	7	7	6
15	7	7	6	7	7	6
49	7	7	7	7	7	7

6.2 PROBABILITY OF ERROR DETECTION.

This section computes the probability of error detection given the minimum number of tests determined in section 6.1, and the probability of error detection formula given in section 5.2.

6.2.1 Boolean (Context Free) Analysis.

Figure 34 plots the probability of error detection results from the size of test sets data in figure 28 for Unique-Cause MCDC for one through six conditions. For those expressions for which at least one test set exists, the number of expressions with that probability is shown below and to the left of each dot. For those expressions for which there is no test set possible, no data is plotted. As can be expected, since the majority of expressions required $N + 1$ tests, the majority of the expressions fall on the minimum probability of error detection line for Unique-Cause MCDC.

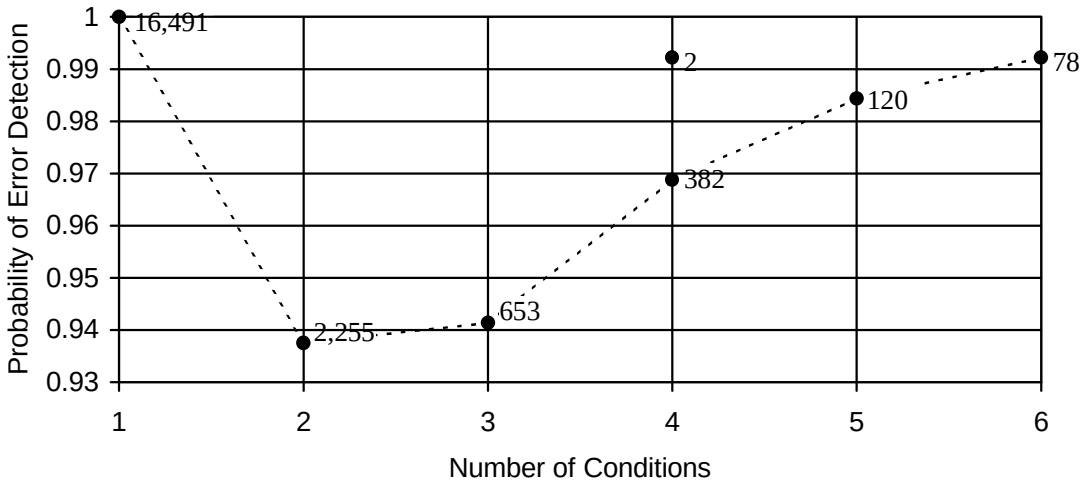


FIGURE 34. PROBABILITY OF ERROR DETECTION—UNIQUE-CAUSE (CONTEXT FREE)

Figure 35 plots the probability of error detection results from the size of test sets data in figure 29 for Unique-Cause + Masking MCDC for one through six conditions. As can be expected, since the majority of expressions required $N + 1$ tests, the majority of the expressions fall on the minimum probability of error detection line for Unique-Cause MCDC.

Figure 36 plots the probability of error detection results from the size of test sets data in figure 30 for Masking MCDC for one through six conditions. As can be expected, since the majority of expressions required $N + 1$ tests, the majority of the expressions fall on the minimum probability of error detection line for Unique-Cause MCDC. None of the expressions fall below the Masking MCDC line.

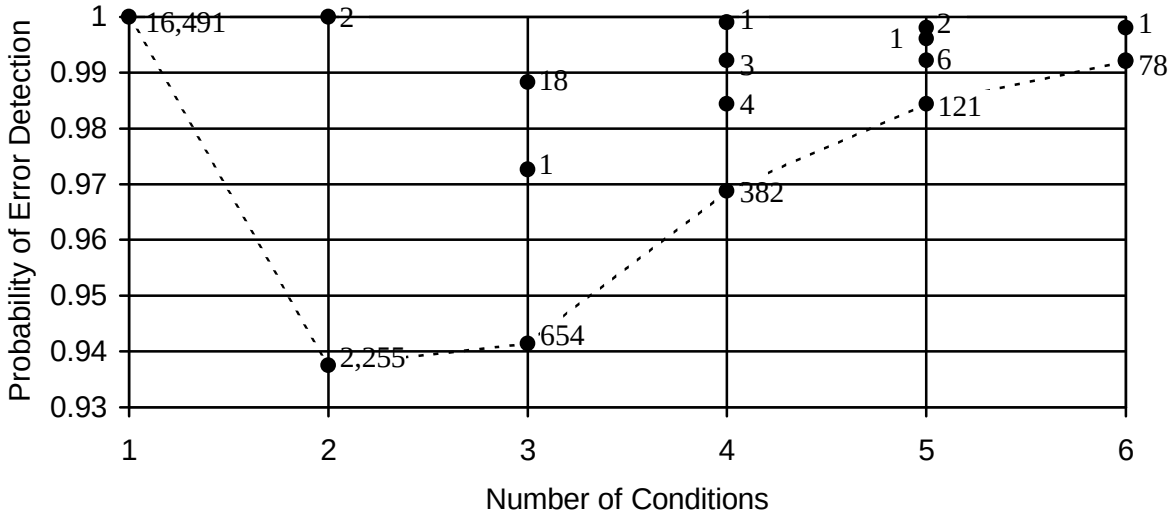


FIGURE 35. PROBABILITY OF ERROR DETECTION—UNIQUE-CAUSE + MASKING (CONTEXT FREE)

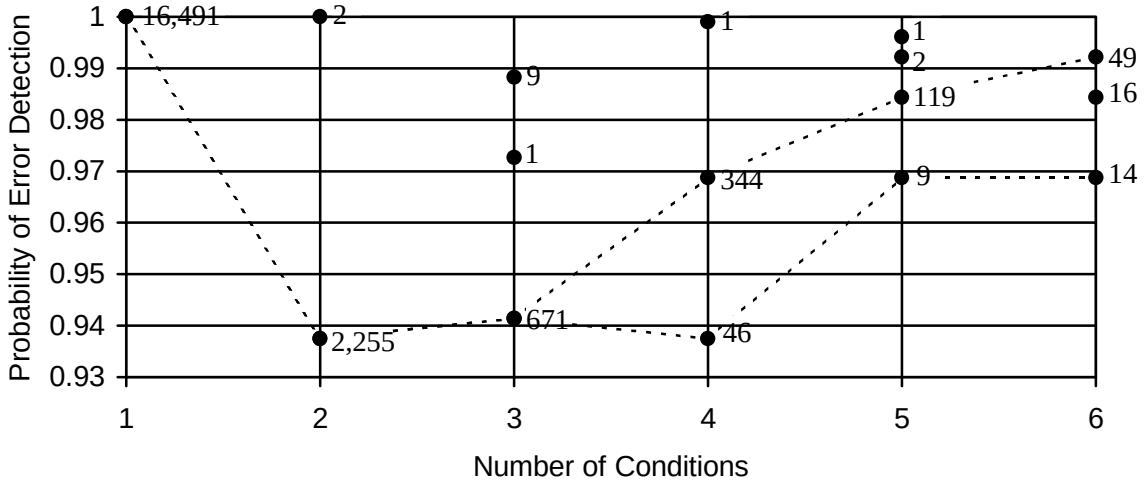


FIGURE 36. PROBABILITY OF ERROR DETECTION—MASKING (CONTEXT FREE)

6.2.2 Coupling (Context Dependent) Analysis.

Figure 37 plots the probability of error detection results from the size of test sets data in figure 31 for Unique-Cause MCDC for one through six conditions. As can be expected, since the majority of expressions required $N + 1$ tests, the majority of the expressions fall on the minimum probability of error detection line for Unique-Cause MCDC.

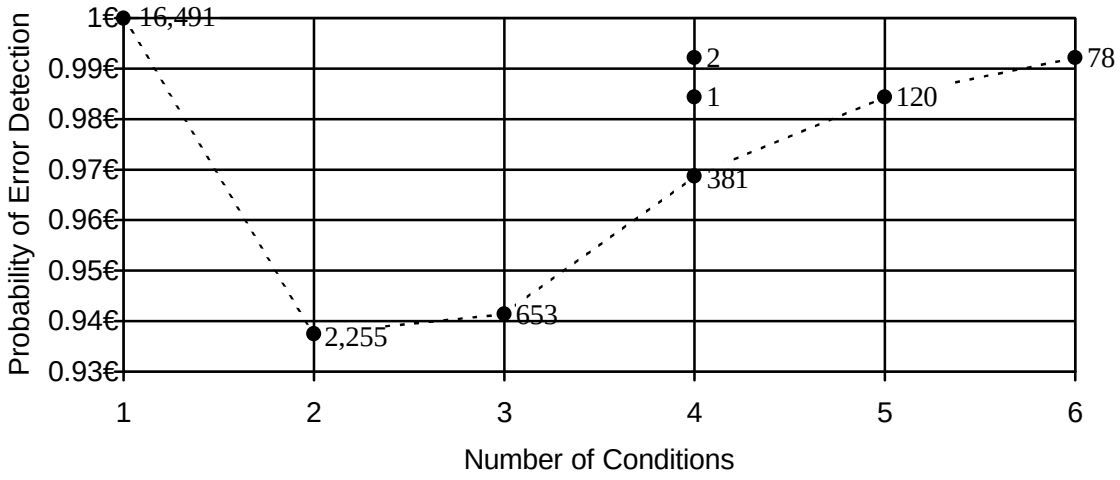


FIGURE 37. PROBABILITY OF ERROR DETECTION—UNIQUE-CAUSE (CONTEXT DEPENDENT)

Figure 38 plots the probability of error detection results from the size of test sets data in figure 32 for Unique-Cause + Masking MCDC for one through six conditions. As can be expected, since the majority of expressions required $N + 1$ tests, the majority of the expressions fall on the minimum probability of error detection line for Unique-Cause MCDC.

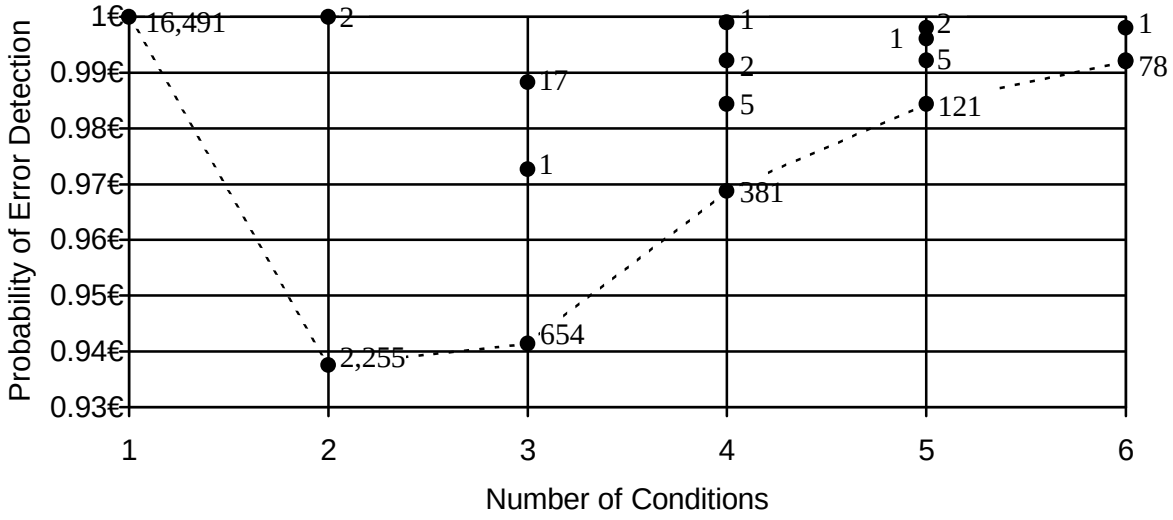


FIGURE 38. PROBABILITY OF ERROR DETECTION—UNIQUE-CAUSE + MASKING (CONTEXT DEPENDENT)

Figure 39 plots the probability of error detection results from the size of test sets data in figure 33 for Masking MCDC for one through six conditions. As can be expected, since the majority of expressions required $N + 1$ tests, the majority of the expressions fall on the minimum probability of error detection line for Unique-Cause MCDC. None of the expressions fall below the Masking MCDC line.

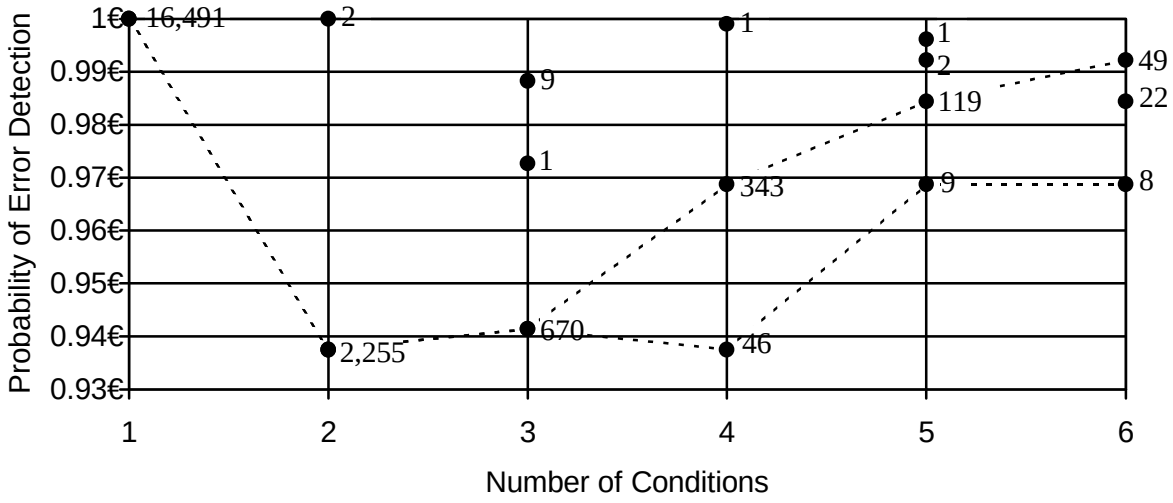


FIGURE 39. PROBABILITY OF ERROR DETECTION—MASKING (CONTEXT DEPENDENT)

6.3 NUMBER OF INDEPENDENCE PAIRS.

As was shown in section 5.1, one of the things that can happen when moving from Unique-Cause MCDC to Masking MCDC is that the number of independence pairs grows. More independence pairs mean a greater probability of achieving coverage for each condition. Tables 23 through 25 present the empirical data for the average number of independence pairs allowed by each definition of MCDC for the expressions contained in appendix C with one through six conditions.

Table 23 provides the averages over all of the conditions, including those for which there are no solutions. This analysis disfavors Unique-Cause over the others since fewer conditions are solvable for this form of MCDC, and those conditions contributed to bringing the average down. However, the data show the trend that is expected: that as it moves from Unique-Cause to Masking, the number of independence pairs for each condition will tend to rise. This rise starts to take off from four conditions and up.

TABLE 23. AVERAGE NUMBER OF INDEPENDENCE PAIRS PER CONDITION—ALL CONDITIONS

Number of Conditions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
1	1.0	1.0	1.0	1.0	1.0	1.0
2	1.0507	1.0532	1.0543	1.0507	1.0532	1.0543
3	1.1656	1.2102	1.7581	1.1490	1.1921	1.6874
4	1.5139	1.5247	3.4798	1.4507	1.46081	3.1075
5	1.9277	2.0147	9.2434	1.8245	1.9100	8.2257
6	4.2521	4.2689	35.571	3.2227	3.2395	24.655

Table 24 provides the averages over all of the conditions for which there are solutions. This analysis favors Unique-Cause over the others since fewer conditions are solvable for this form of MCDC, and those missing conditions contributed to keeping the average up. Based on this analysis, one would expect that Unique-Cause MCDC would be the better criterion to use over Unique-Cause + Masking MCDC since it allows for more independence pairs per condition. Notice that even though Unique-Cause MCDC is favored by this analysis, it still did worse than Masking MCDC.

TABLE 24. AVERAGE NUMBER OF INDEPENDENCE PAIRS PER CONDITION—ALL SOLVABLE CONDITIONS

Number of Conditions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
1	1.0	1.0	1.0	1.0	1.0	1.0
2	1.0549	1.0548	1.0554	1.0549	1.0548	1.0554
3	1.2273	1.2217	1.7614	1.2098	1.2041	1.6914
4	1.5323	1.5285	3.4798	1.4683	1.4654	3.1094
5	2.0327	2.0267	9.2434	1.9238	1.9242	8.2257
6	4.2881	4.2689	35.571	3.2500	3.2395	24.655

Table 25 provides the averages over all of the conditions that have solutions under all the MCDC forms and context sensitivities. This analysis reduces Unique-Cause and Unique-Cause + Masking into the same criterion. This analysis again shows that Masking MCDC allows more independence pairs for the conditions in an expression.

TABLE 25. AVERAGE NUMBER OF INDEPENDENCE PAIRS PER CONDITION—COMMON SOLVABLE CONDITIONS

Number of Conditions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
1	1.0	1.0	1.0	1.0	1.0	1.0
2	1.0550	1.0550	1.0550	1.0550	1.0550	1.0550
3	1.2149	1.2149	1.7448	1.2001	1.2001	1.6810
4	1.5286	1.5286	3.5039	1.4701	1.4701	3.1377
5	2.0017	2.0017	9.4450	1.9167	1.9167	8.5233
6	4.2650	4.2650	35.641	3.2179	3.2179	24.538

All of the analyses performed in this section tend to favor Masking MCDC as the form easiest to satisfy by virtue of providing more independence pairs for each condition.

6.4 SIZE OF MINIMAL TEST SETS.

As was shown in section 5.1, one of the things that can happen as it moves from Unique-Cause MCDC to Masking MCDC is that the size of the coverage test sets shrinks. Fewer required

tests means a greater probability of achieving coverage for each expression. As was seen in section 5.2, fewer required tests also means a decreased probability of detecting errors. Tables 26 through 28 present the empirical data for the average number of tests in a coverage test set allowed by each definition of MCDC for the expressions contained in appendix C with one through six conditions.

Table 26 provides the averages over all of the expressions, including those for which there are no solutions. This analysis disfavors Unique-Cause over the others since fewer expressions are solvable for this form of MCDC and those expressions contributed to bringing the average down.

TABLE 26. AVERAGE COVERAGE TEST SET SIZE—ALL EXPRESSIONS

Number of Conditions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
1	2.0	2.0	2.0	2.0	2.0	2.0
2	2.9907	2.9943	2.9943	2.9907	2.9943	2.9943
3	3.8131	3.9839	4.0044	3.8131	3.9752	3.9985
4	4.9207	5.0256	4.8951	4.9233	5.0102	4.8824
5	5.4962	6.0611	5.9618	5.4962	6.0076	5.9618
6	6.9114	7.0253	6.4430	6.9114	7.0253	6.5190

Based on this analysis, one would expect that Unique-Cause + Masking MCDC would be the better criterion to use for error detection since it requires more tests per expression. Which form to choose for ease of coverage is not clear from this analysis since Masking MCDC is more closely following the Unique-Cause curve of figure 26 than it is the Masking curve.

Table 27 provides the averages over all of the expressions for which there are solutions. This analysis favors Unique-Cause over the others since fewer expressions are solvable for this form of MCDC and those missing conditions contributed to keeping the average down with respect to Unique-Cause + Masking and up with respect to Masking. Based on this analysis, one would expect that Unique-Cause + Masking MCDC would be the better criterion to use for error detection since it requires more tests per expression. Which form to choose for ease of coverage is not clear from this analysis since Masking MCDC is very close to Unique-Cause.

TABLE 27. AVERAGE COVERAGE TEST SET SIZE—ALL SOLVABLE EXPRESSIONS

Number of Conditions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
1	2.0	2.0	2.0	2.0	2.0	2.0
2	3.0	3.0009	3.0009	3.0	3.0009	3.0009
3	4.0	4.0550	4.0279	4.0	4.0521	4.0279
4	5.0104	5.0385	4.8951	5.0130	5.0360	4.8949
5	6.0	6.1077	5.9618	6.0	6.1008	5.9618
6	7.0	7.0253	6.4430	7.0	7.0253	6.5190

Table 28 provides the averages over all of the expressions that have solutions under all the MCDC forms and context sensitivities. This analysis reduces Unique-Cause and Unique-Cause + Masking into the same criterion. This analysis again shows that the Unique-Cause and Masking forms of MCDC require essentially the same number of tests per expression.

TABLE 28. AVERAGE COVERAGE TEST SET SIZE—COMMON SOLVABLE EXPRESSIONS

Number of Conditions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
1	2.0	2.0	2.0	2.0	2.0	2.0
2	3.0	3.0	3.0	3.0	3.0	3.0
3	4.0	4.0	4.0	4.0	4.0	4.0
4	5.0104	5.0104	4.8802	5.0130	5.0130	4.8802
5	6.0	6.0	5.9250	6.0	6.0	5.9250
6	7.0	7.0	6.4487	7.0	7.0	6.5256

All of the analyses performed in this section tend not to favor any MCDC form as easiest to satisfy by virtue of providing smaller coverage test sets for each condition. This also means that none are favored from the probability of error detection viewpoint either.

6.5 NUMBER OF MINIMAL TEST SETS.

As was shown earlier in section 5.1, one of the things that can happen as it moves from Unique-Cause MCDC to Masking MCDC is that the number of coverage test sets grows. More allowed test sets means a greater probability of achieving coverage for each expression. Tables 29 through 31 present the empirical data for the average number of minimal coverage test sets allowed by each definition of MCDC for the expressions contained in appendix C with one through six conditions.

Table 29 provides the averages over all of the expressions, including those for which there are no solutions. Based on this analysis, one would expect that Masking MCDC would be the better criterion to use since it allows more coverage test sets per expression.

TABLE 29. AVERAGE NUMBER OF COVERAGE TEST SETS—ALL EXPRESSIONS

Number of Conditions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
1	1.0	1.0	1.0	1.0	1.0	1.0
2	1.1614	1.1622	1.1622	1.1614	1.1622	1.1622
3	1.3489	1.4175	1.7723	1.3372	1.3912	1.7168
4	2.2174	2.2711	5.1407	2.1637	2.1790	4.6777
5	3.8550	4.3206	32.466	3.7634	4.0534	30.069
6	15.722	15.747	1330.4	9.9747	10.0	790.04

Table 30 provides the averages over all of the expressions for which there are solutions. This analysis disfavors Unique-Cause over the others since fewer expressions are solvable for this form of MCDC and those missing conditions contributed to keeping the average down. Based on this analysis, one would expect that Masking MCDC would be the better form to use since it allows more coverage test sets per expression.

TABLE 30. AVERAGE NUMBER OF COVERAGE TEST SETS—ALL SOLVABLE EXPRESSIONS

Number of Conditions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
1	1.0	1.0	1.0	1.0	1.0	1.0
2	1.1650	1.1648	1.1648	1.1650	1.1648	1.1648
3	1.4150	1.4428	1.7827	1.4028	1.4182	1.7294
4	2.2578	2.2769	5.1407	2.2031	2.1902	4.6897
5	4.2083	4.3583	32.466	4.1083	4.1163	30.069
6	15.923	15.747	1330.4	10.103	10.000	790.04

Table 31 provides the averages over all of the expressions that have solutions under all the MCDC forms and context sensitivities. This analysis reduces Unique-Cause and Unique-Cause + Masking into the same criterion. This analysis again shows that Masking MCDC would be the better form to use since it allows more coverage test sets per expression.

TABLE 31. AVERAGE NUMBER OF COVERAGE TEST SETS—COMMON SOLVABLE EXPRESSIONS

Number of Conditions	Context Free			Context Dependent		
	Unique Cause	Unique Masking	Masking	Unique Cause	Unique Masking	Masking
1	1.0	1.0	1.0	1.0	1.0	1.0
2	1.1650	1.1650	1.1650	1.1650	1.1650	1.1650
3	1.4150	1.4150	1.7335	1.4028	1.4028	1.6907
4	2.2578	2.2578	5.1458	2.2031	2.2031	4.7240
5	4.2083	4.2083	33.900	4.1083	4.1083	31.558
6	15.923	15.923	1342.2	10.103	10.103	794.83

7. SOLVABLE EXPRESSIONS.

As was pointed out in reference 4, not all expressions have an MCDC coverage set. Coupling between conditions in an expression is the mechanism by which this occurs. There are two forms of coupling that need to be addressed:

1. Repeated conditions (i.e., strong coupling, Boolean coupling)
2. Context coupling

The Boolean coupling mechanism is discussed in section 7.1, while the context coupling mechanism is discussed in section 7.2. Section 7.3 looks at the solvable expression problem from a different perspective. Nonsingular Boolean expressions (non-SBEs) are briefly addressed and how masking helps with the solvability of expressions for these functions.

7.1 BOOLEAN COUPLING.

Those expressions with repeated coupled conditions are sometimes structured such that one or more of the coupled conditions cannot be masked when one of the other condition instances is being toggled to demonstrate its independence. This can be demonstrated with the expression *(A and B) or (A and not B)*. Figure 40 is a two-cube representation (see section 5.1) of the function this expression represents. The two-cube on the left is the traditional mathematical representation, while the two-cube on the right has been annotated with condition codes.

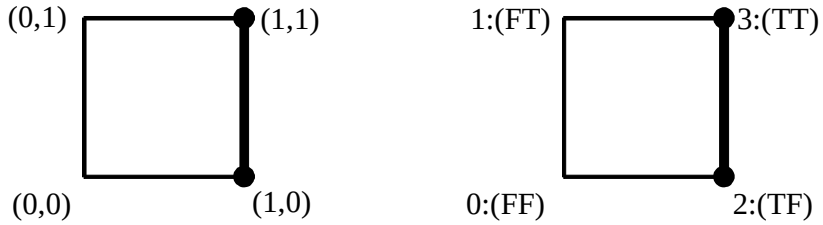


FIGURE 40. TWO-CUBE REPRESENTATION OF *(A and B) or (A and not B)*

The expression *(A and B) or (A and not B)* is in conjunctive normal form. This means that each of the subexpressions (*(A and B), (A and not B)*) represent minterms, and therefore are represented by vertices of the two-cube in figure 40. The minterm *(A and B)* corresponds to the (1,1)/3:(TT) vertex, while the minterm *(A and not B)* corresponds to the (1,0)/2:(TF) vertex.

In order to show the independence of the first occurrence of *A*, a transition from (1,1) to (0,1) must be done with a change in the function between false and true (expression tree analysis left as an exercise for the reader). This is possible as the two-cube in figure 40 shows. In order to show the independence of the first occurrence of *B*, a transition from (1,1) to (1,0) must be done with a change in the function between false and true. This is not possible as the two-cube in figure 40 shows. This expression cannot be MCDC tested under any of the definitions. Note that this function can be simplified to the expression *A*, in which case, it is now MCDC testable under all of the definitions.

In the above example, *B* was not a significant variable in the function (i.e., the function did not depend on *B*, or the function was not a function of *B*). In the next example, *B* is significant to the function (i.e., the function is a function of *B*). Figure 41 is a two-cube representation of the function that the expression *(not A and not B) or (not A and B) or (A and not B)* represents. This expression is again in conjunctive normal form.

With this function, it is possible to show the independence of the second occurrence of *A* (transition from (0,1) to (1,1)) and the third occurrence of *B* (transition from (1,0) to (1,1)). The independence of the other conditions cannot be shown. Note that this expression can be

simplified to either *not A or not B*, or *not (A and B)*, both of which are MCDC testable under all of the definitions.

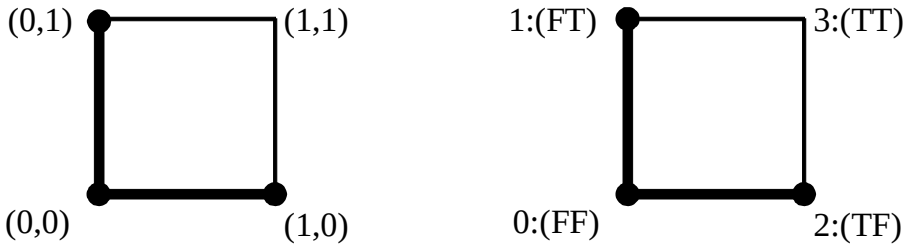


FIGURE 41. TWO-CUBE REPRESENTATION OF *(not A and not B) or (not A and B) or (A and not B)*

A function with three different expressions for it will be considered. Figure 42 is a three-cube representation for the function with the three expressions.

A and (B or C);
(A and B) or (A and C); and
(A and not B and C) or (A and B and not C) or (A and B and C).

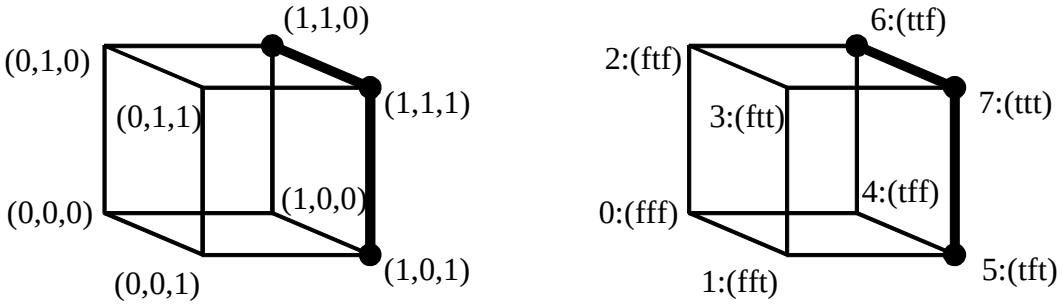


FIGURE 42. THREE-CUBE REPRESENTATION OF *A and (B or C)*

The independence analysis for the expression *A and (B or C)* is presented in table 32. The first column lists the condition under consideration, the second column identifies one of the independence pairs for the condition, and the final column identifies the transition type (see section 5.1).

Notice that there is at least one edge transition independence pair for each condition, so there is both a Unique-Cause MCDC and Unique-Cause + Masking MCDC coverage set for this expression. Notice that there is at least one independence pair (of any transition type) for each of the conditions, so there is a Masking MCDC coverage set for this expression. Also notice that there are diagonal transitions for condition *A*, so there are more coverage sets for Masking MCDC than there are for Unique-Cause/Unique-Cause + Masking MCDC (remember that one independence pair is needed for each condition to form a coverage set).

TABLE 32. INDEPENDENCE ANALYSIS FOR *A and (B or C)*

Condition	Independence Pair	Transition Type
A	(1, 5)	Edge
	(1, 6)	Internal Diagonal
	(1, 7)	Face Diagonal
	(2, 5)	Internal Diagonal
	(2, 6)	Edge
	(2, 7)	Face Diagonal
	(3, 5)	Face Diagonal
	(3, 6)	Face Diagonal
	(3, 7)	Edge
B	(4, 6)	Edge
C	(4, 5)	Edge

The independence analysis for the expression *(A and B) or (A and C)* is presented in table 33. In this table, the independence pairs are separated by the subexpression they are a part of (since *A* has multiple occurrences and each must demonstrate its independence). Because of the strongly coupled condition (*A* appearing twice) present in this expression, there is no Unique-Cause MCDC test set. Notice that there is at least one edge transition independence pair for each condition, so there is a Unique-Cause + Masking MCDC coverage set for this expression. Notice that there is at least one independence pair (of any transition type) for each of the conditions, so there is a Masking MCDC coverage set for this expression.

TABLE 33. INDEPENDENCE ANALYSIS FOR *(A and B) or (A and C)*

Condition	Independence Pair	Transition Type
(A and B)		
A (1st occurrence)	(2, 6)	Edge
	(3, 6)	Face Diagonal
B	(4, 6)	Edge
(A and C)		
A (2nd occurrence)	(1, 5)	Edge
	(3, 5)	Face Diagonal
C	(4, 5)	Edge

The independence analysis for the expression *(A and not B and C) or (A and B and not C) or (A and B and C)* is presented in table 34. Notice that there are no independence pairs (of any transition type) for some of the conditions. This means that there is no MCDC coverage set for any of the types of MCDC of this study.

What the examples show is that when there is no transition from a *true vertex* to a *false vertex* allowed by the MCDC definition for a condition, there is no (complete) MCDC coverage set. This is as expected since MCDC requires an expression-level transition between true to false as a result of a condition-level transition between true to false in such a way that the transition condition is the only condition to affect the expression transition.

These examples also show that an expression can have coverage sets for one form of MCDC that differ from those for another, in particular between Masking MCDC and one of the Unique-Cause variants. This is because Masking MCDC allows diagonal transitions in an n-cube representation of the expression. They have also shown that certain expressions will have no (complete) coverage set for any form of MCDC. This is because certain expressions allow neither edge nor diagonal transitions for certain conditions.

TABLE 34. INDEPENDENCE ANALYSIS FOR *(A and not B and C) or (A and B and not C) or (A and B and C)*

Condition	Independence Pair	Transition Type
(A and not B and C)		
A (1st occurrence)	(1, 5)	Edge
B (1st occurrence)		
C (1st occurrence)B	(4, 5)	Edge
(A and B and not C)		
A (2nd occurrence)	(2, 6)	Edge
B (2nd occurrence)	(4, 6)	Edge
C (2nd occurrence)		
(A and B and C)		
A (3rd occurrence)	(3, 7)	Edge
B (3rd occurrence)		
C (3rd occurrence)		

It is this property of not allowing transitions that can be used to characterize which functions and expressions have (complete) MCDC coverage sets under different definitions for independence. Unfortunately, this study was not able to pursue this analysis very far. A small analysis was pursued and is documented in section 7.3.

7.2 CONTEXT COUPLING.

As was pointed out in section 6.1.2 of this report, the context coupling of Boolean expressions incorporating relational operators on non-Booleans can also cause expressions to not have an MCDC coverage set. This section will examine, in detail, the following line replaceable unit (LRU) expression from appendix C in order to identify the circumstances under which the infeasibility occurs. The expression is

(Bv and (Fxv > Fxv2 - url)) or (not Bv and (Fxv > Fxv2))

The analysis will start by looking at the context-free version of this expression. Table 35 is the independence pairs analysis for the expression *(A and B) or (not A and C)*. In this table, the independence pairs are separated by the subexpression they are a part of (since *A* has multiple occurrences and each must demonstrate its independence). Because of the strongly coupled condition (*A* appearing twice) present in this expression, there is no Unique-Cause MCDC test set. Notice that there is at least one edge transition independence pair for each condition, so there is a Unique-Cause + Masking MCDC coverage set for this expression, and there is at least one

independence pair (of any transition type) for each of the conditions, so there is a Masking MCDC coverage set for this expression.

TABLE 35. INDEPENDENCE ANALYSIS FOR *(A and B) or (not A and C)*

Condition	Independence Pair	Transition Type
(A and B)		
A (1st occurrence)	(2, 6)	Edge
	(2, 7)	Face Diagonal
B	(4, 6)	Edge
	(4, 7)	Face Diagonal
	(5, 6)	Face Diagonal
	(5, 7)	Edge
(not A and C)		
A (2nd occurrence)	(1, 5)	Edge
	(1, 3)	Edge
C	(0, 1)	Edge
	(0, 3)	Face Diagonal
	(1, 2)	Face Diagonal
	(2, 3)	Edge

The condition *B* in our context-free analysis corresponds to the subexpression *(Fxv > Fxv2 - url)*, and the condition *C* corresponds to the subexpression *(Fxv > Fxv2)*. These two conditions are weakly coupled. If *(Fxv > Fxv2 - url)* is false (i.e., *Fxv* ? *Fxv2 - url*), then *(Fxv > Fxv2)* cannot be true. This is because a number (*Fxv*) cannot be both less than another number (*Fxv2*) with a nonnegative decrement (*url*) and greater than the same number (*Fxv2*) at the same time. This means that condition combinations (1) and (5) are infeasible. If these are removed from table 35, what is left is the independence table for *(Bv and (Fxv > Fxv2 - url)) or (not Bv and (Fxv > Fxv2))*, presented in table 36.

TABLE 36. INDEPENDENCE ANALYSIS FOR *(Bv and (Fxv > Fxv2 - url)) or (not Bv and (Fxv > Fxv2))*

Condition	Independence Pair	Transition Type
(A and B)		
Bv (A) (1st occurrence)	(2, 6)	Edge
	(2, 7)	Face Diagonal
Fxv > Fxv2 – url (B)	(4, 6)	Edge
	(4, 7)	Face Diagonal
(not A and C)		
Bv (A) (2nd occurrence)		
Fxv > Fxv2 (C)	(0, 3)	Face Diagonal
	(2, 3)	Edge

Examination of table 36 shows that the second occurrence of *A* (*Bv1*) has no independence pairs in the context-dependent analysis. This is because the infeasible condition combinations were essential to showing that condition's independence since condition combination (1) was a

member of both independence pairs. Comparing tables 35 and 36 shows that the second and fourth conditions also lost independence pairs, but not all of them.

7.3 MASKING AND NONSINGULAR BOOLEAN EXPRESSIONS (NON-SBE).

As was pointed out in section 2.4, some expressions do not have a Unique-Cause MCDC test set by virtue of the fact that they cannot be represented by an SBE. Section 5.1 pointed out that certain Boolean expressions can have different coverage sets for the different forms of MCDC. Section 6 showed that certain expressions without a Unique-Cause coverage test set could have test sets under the masking forms. Section 7.1 showed that certain Boolean functions could be represented by multiple Boolean expressions, some of which were solvable under one form of MCDC while others were not. This section briefly visits the issue of whether or not a masking coverage set exists for all non-SBEs.

Recall that the data in table 3 indicates that the DO-178B definition for MCDC has a very limited range of applicability. As part of this study, a brief but incomplete investigation was conducted into the question of whether one of the masking forms of MCDC would be applicable to all non-SBEs.

For the three-condition case, there were 104 non-SBEs. For all of these Boolean functions, a Boolean expression exists which can be solved for both Unique-Cause + Masking MCDC and Masking MCDC.

For the four-condition case, there were 62,440 non-SBEs. For the 25,520 functions that were able to be analyzed within the bounds of this study, a Boolean expression was found which could be solved for both Unique-Cause + Masking MCDC and Masking MCDC.

It is not known if there are Boolean functions for which only Masking MCDC coverage sets exist for all solvable expressions. It is also not known if there are Boolean functions for which no MCDC solvable expression exists. This analysis would need to be extended within another study.

Examination of the data contained in appendix C gives a different story than the data in table 3. Of the 20,256 expressions extracted from the five systems, only 72 are non-SBEs. From the empirical point of view, the inapplicability of the DO-178B MCDC definition to non-SBEs is presently not an item of great concern.

8. MUTATION/FAULT INJECTION INVESTIGATION.

Section 6.2 conducted an empirical comparison of the probability of error detection given MCDC test sets for the expressions in appendix C. This analysis used the probability of error detection model developed in section 5.2. An alternative approach would be to determine the probability of error detection given faulty code and the MCDC test sets for that code. Unfortunately, examples of faulty airborne software with documented faults were not available for this study, so an alternative had to be found. That alternative is to inject faults into software and see how the test sets perform. This alternative, known alternatively as software mutation originally or more

recently known as software fault injection, is one that has been used within the software testing research community to evaluate the effectiveness of new testing methods and coverage criteria.

This section uses mutation/fault injection to perform comparisons between the different forms of MCDC using generated logic expressions and fault injections. The details of the study methodology, along with some problems that were discovered with mutation theory itself, are documented in appendix D. These comparisons are intended to provide data to allow for a choice between the different forms of MCDC. Note that the methodology used for this study did not generate expressions with coupled conditions, so Unique-Cause and Unique-Cause + Masking MCDC are identical. The comparisons performed are

- The average number of tests in a minimal coverage test set (section 8.1) versus expression size (number of independent conditions). This comparison is used to determine one aspect of the cost-effectiveness of the different forms of MCDC. It is assumed, as in section 6.4, that the smaller the minimal coverage test set, the easier the attainment of coverage will be. However, this smaller test set also carries with it a lower probability of error detection.

- The average number of minimal test sets (section 8.2) versus expression size. This comparison is used to determine one aspect of the cost-effectiveness of the different forms of MCDC. It is assumed, as in section 6.5, that the larger the number of acceptable test sets, the easier the attainment of coverage will be.

- The probability of error detection (section 8.3) (mutation kill) given the minimum MCDC test sets versus expression size. This analysis was conducted in two forms: one for the mutants themselves, including the redundancy, and one for the spanned functions (i.e., the nonredundant mutations).

Because of the analysis methods used in these comparisons, they are limited to expressions with one through four conditions. These results are exhaustive because the complete expression, mutation, and minimal test sets spaces were examined. A partial analysis of the expressions with five conditions was completed and the results are included in the corresponding tables in italics. For the five-condition expressions, 71% of the expressions dominated by the (AND, ≤, ≥) operators were analyzed. These expressions were chosen because in the two- through four-condition analyses, expressions dominated by these operators had results very near the average, and there was insufficient time in this study to perform the complete five-condition analysis. Care should be taken with these five condition results as the analysis of expressions was not exhaustive as they were for one through four conditions.

8.1 AVERAGE SIZE OF MINIMAL TEST SETS.

This section determines the average size of the minimal MCDC compliant test sets allowed by the Boolean expressions generated according to the methodology detailed in appendix D. Each expression was exhaustively examined (i.e., all permutations and combinations of condition combinations were computed) to determine the absolute smallest test set that would satisfy each of the MCDC definitions.

Table 37 provides the averages over all the expressions. This analysis differs from that performed in section 6 in that all expressions generated were examined, not just the subset represented by those presented in appendix C. Since all of the expressions are SBEs, the test sets for Unique-Cause MCDC are always the size predicted by the $N + 1$ rule developed in section 5.1. The data also agrees with the development in section 5.1 that any expression up through three conditions always requires $N + 1$ tests for any form of MCDC. Because of this, only the data for four and five conditions is really significant for the differentiation between the different forms of MCDC. As was the case in section 6.4, this analysis did not establish a clear preference between the different forms of MCDC.

TABLE 37. AVERAGE SMALLEST NUMBER TEST SETS SIZE VS EXPRESSION SIZE

Number of Conditions	Unique Cause	Unique Masking	Masking
1	2.0	2.0	2.0
2	3.0	3.0	3.0
3	4.0	4.0	4.0
4	5.0	5.0	4.7990
5	*6.0*	*6.0*	*5.5112*

8.2 AVERAGE NUMBER OF MINIMAL TEST SETS.

This section determines the average number of MCDC compliant minimal test sets allowed by the Boolean expressions generated according to the methodology detailed in appendix D. Each expression was exhaustively examined (i.e., all permutations and combinations of condition combinations were computed) to determine all of the nonredundant minimal test sets that would satisfy each of the MCDC definitions.

Table 38 provides the averages over all the expressions. This analysis differs from that performed in section 6 in that all expressions generated were examined, not just the subset represented by those presented in appendix C. As was the case in section 6.5, this analysis shows that the Masking form of MCDC is preferable since it allows significantly more coverage sets per expression.

TABLE 38. AVERAGE NUMBER OF MINIMAL NONREDUNDANT TEST SETS VS EXPRESSION SIZE

Number of Conditions	Unique Cause	Unique Masking	Masking
1	1.0	1.0	1.0
2	2.0000	2.0000	2.0000
3	7.3333	7.3333	9.3333
4	41.2222	41.2222	67.4444
5	*73.1314*	*73.1314*	*460.139*

8.3 PROBABILITY OF MUTATION (ERROR) DETECTION.

This section determines the probability of detecting mutation errors for the Boolean expressions generated according to the methodology detailed in appendix D. Each expression was exhaustively examined by injecting each mutant into the expression, and then determining if each minimal MCDC compliant test set would detect (kill) the error (mutant) or not.

This analysis was conducted in two different ways. In the first analysis, the raw mutants were used, and the results for that analysis are in table 39. In the second analysis, the redundant mutants were removed leaving only the spanned functions, and the results for that analysis are in table 40. The reason for these two analyses is because the mutants tend to cluster (many mutants representing the same underlying Boolean function), and the probability of error detection model developed in section 5 considered all Boolean functions as the potential error needing detection. By conducting both versions of the analysis, it tends to minimize skewing the results by mutants clustering in areas where MCDC is insensitive.

TABLE 39. PROBABILITY OF MUTATION DETECTION VS EXPRESSION SIZE—MUTANTS

Number of Conditions	Unique Cause	Unique Masking	Masking
1	1.0	1.0	1.0
2	0.93959	0.93959	0.93959
3	0.94331	0.94331	0.94508
4	0.94786	0.94786	0.94540
5	*0.91969*	*0.91969*	*0.91222*

The results in table 40 show that the probability of detecting (mutation) errors does not follow the curve predicted by the probability of error detection model developed in section 5. This is because mutation is not able to encompass the entire Boolean function space. However, since this flaw applies equally to all forms of MCDC, mutation can be used as a yardstick for comparison.

TABLE 40. PROBABILITY OF MUTATION DETECTION VS EXPRESSION SIZE—SPANNED FUNCTIONS

Number of Conditions	Unique Cause	Unique Masking	Masking
1	1.0	1.0	1.0
2	0.92857	0.92857	0.92857
3	0.93160	0.93160	0.93394
4	0.93748	0.93748	0.93468
5	*0.89990*	*0.89990*	*0.88652*

The analyses performed in this section tend not to favor any form of MCDC from the probability of error detection viewpoint. As was established in sections 5.2 and 6.2, and corroborated here, the probability of error detection between all forms of MCDC is nearly identical.

9. CONCLUSIONS AND FURTHER WORK.

Based on the results from all the different analyses performed during this study, it was concluded that Masking MCDC should be the preferred form of MCDC. The progression from the start of this study through its conclusion is described in the following paragraphs.

Coming into this study, it was known that all forms of verification could miss important features of the system being implemented. Examples in this report demonstrate requirements-based verification missing important features of the implementation. Structural coverage in the software system development process is a check and balance on requirements-based verification. Examples in this report demonstrate that structural coverage is not a complete answer, only one part of the answer.

MCDC is a form of structural coverage providing equivalence class and boundary value coverage of the implementation. MCDC provides coverage of equivalence classes by ensuring that the verification process visits each side of the subdomain partitions in a significant manner. Extensions to cover the verification of relational operators and operands were defined that would strengthen MCDC in the boundary value coverage of non-Booleans.

Examples demonstrate that multiple degrees of rigor could be applied to MCDC resulting in different forms of MCDC. This study investigated three different forms of MCDC. These investigations are intended to support a rational choice between which form of MCDC should be preferred. The investigative methods could also be used to evaluate other proposed alternatives to those investigated by this study. In fact, one of the investigative methods was used to compare the performance of MCDC against Statement Coverage and Decision Coverage.

The first investigation was into the minimum number of tests required by each of the forms of MCDC. This analysis showed that Masking MCDC would be the easiest form to satisfy, as it required fewer tests than the other forms of MCDC. To support the theoretical analysis, an empirical analysis was performed against logical expressions extracted from five LRUs. An additional empirical investigation was conducted using generated expressions to cover the entire SBE space. The empirical data confirmed the theory, and also showed that, in practice, Masking MCDC required a number of tests equivalent to that of the other forms of MCDC.

The second investigation was into the theoretical minimum probability of logic error detection. For this analysis a model was developed for the error detecting capabilities of any coverage criterion. This model defined an error as having an incorrect Boolean function in the implementation (i.e., the implementation was not what was specified). The error model is based on the *number* of tests, not the *kind* of tests. This means that all tests were assumed equal, hence, any coverage method that provides an equal number of tests to MCDC would have the same performance. Another study would need to be conducted to ascertain if the MCDC selection rules were superior to just randomly selecting the equivalent number of tests. This other study could also address errors in the relational space in order to determine if the extensions defined in this report would be worth while (e.g., cost-effective).

Using the above model, a theoretical analysis of the performance of the three forms of MCDC was conducted. This analysis showed that even though Masking MCDC could allow fewer tests than Unique-Cause MCDC, its performance in detecting incorrect Boolean functions was not that much different. To support the theoretical analysis, an empirical analysis was performed against logical expressions extracted from five LRUs. The empirical data not only confirmed the theory, but also showed that the difference was smaller than the theory predicted. This is mainly due to the fact that the number of tests was equivalent. An additional empirical analysis was performed using generated expressions covering the SBE space and injected faults using the rules of mutation. These empirical results did not follow the theory because of some problems discovered with mutation theory itself (see appendix D). These problems did not compromise using mutation as a yardstick to compare the three forms of MCDC. The results of the mutation analysis also showed that the performance of the three forms of MCDC was nearly identical from the probability of error detection viewpoint.

The theoretical analysis was extended to compare the performance of MCDC with Statement Coverage and Decision Coverage. This analysis showed that the differences between Unique-Cause MCDC and Masking MCDC were insignificant compared with the differences between MCDC and Decision Coverage and Statement Coverage. Validation of these results would require the analysis of actual subprograms extracted from airborne software and actual faults from a real development process. These were not available for this study.

The third investigation was into the average number of independence pairs that met the requirements of each form of MCDC. This investigation was entirely empirical, and showed that there were more independence pairs at all levels for Masking MCDC than for either of the unique-cause forms. It is assumed that the larger the number of independence pairs, the easier coverage would be to attain. This easier attainment should result in less costly verification since coverage would be satisfied more easily by requirements-based verification. This is where better empirical data would have helped. Validation of these results would also require the analysis of actual subprograms extracted from airborne software, and actual faults from a real development process. Unfortunately, this study was limited to the extraction of logical expressions from airborne software.

On the issue of the empirical data, it could also be improved in other ways. Perhaps the best improvement that could be implemented is a design of experiments to determine exactly what sort of empirical data is actually needed. This study used the data at hand for the analyses that were performed. Another way for the data to be improved is to use the airborne software from different supplier's LRUs. This study was limited to a single supplier. The final improvement would be to obtain software in multiple languages. This study was limited to the use of Ada. It is not known if other languages would show equivalent results to those for Ada.

The fourth investigation was into the average number of minimally sized coverage test sets. This investigation was also entirely empirical, and showed that Masking MCDC is satisfied by a greater number of coverage test sets. It is assumed that the larger the number of coverage test sets, the easier coverage would be to attain. That easier attainment should result in less costly verification.

The final (partial) investigation was into the applicability of the different forms of MCDC to different expression types. Theoretically, it was shown the number of Boolean functions for which Unique-Cause MCDC was applicable was a small portion of the Boolean function space. It was also shown that the number of Boolean functions for which Masking MCDC was applicable was a larger portion of the Boolean function space. It could not be determined if Masking MCDC can be applied to all Boolean functions because the analysis could not be completed. This would need to be the focus of yet another study.

Combining the different analyses leads one to the conclusion that Masking MCDC should be the preferred form of MCDC. It requires equivalent numbers of tests to the currently defined Unique-Cause MCDC, but allows for more independence pairs per condition and more coverage test sets per expression.

10. REFERENCES.

1. RTCA Document, "Software Considerations in Airborne Systems and Equipment Certification," Document No. RTCA/DO-178B, RTCA, Inc., December 1, 1992.

2. Howden, W.E., *Functional Program Testing and Analysis*, McGraw-Hill, New York, 1987.

3. Chilenski, J.J. and Richey, L.A., "Definition for a Masking form of Modified Condition Decision Coverage (MCDC)," <http://www.boeing.com/nosearch/mcdc/MCDC.pdf>, May 12, 1997.

4. Chilenski, J.J. and Miller, S.P., "Applicability of Modified Condition Decision Coverage to Software Testing," Software Engineering Journal, Vol. 7, No. 5, September 1994, pp. 193-200.

5. Kohavi, Z., *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978.

6. Myers, G.J., *The Art of Software Testing*, John Wiley & Sons, New York, 1979.

APPENDIX A—TYPE-OF-TRIANGLE ANALYSIS

In this appendix the type-of-triangle problem is used to demonstrate that some form of structural coverage analysis is needed as a check and balance on requirements-based verification, since that (requirements-based) verification may miss important features of an implementation. This analysis will use testing as the verification method, but any other verification method could have been used.

In the triangle problem, a subprogram is supplied three integer values that represent the lengths of three sides of a potential triangle. The subprogram is to return an indication of whether the sides cannot represent a valid triangle, or if valid, the type of triangle they represent. This problem is chosen because it is simple enough to be discussed in a small space but complex enough to illustrate the points needed to be made. This discussion is directed through the following progression:

1. First, the requirements for the problem are laid out with an Ada (package) specification to which an implementation must adhere in order to solve this problem (section A.1).

2. Second, the specifications for tests that Myers gives in the front of his book [1] are discussed and a set of test data based on these specifications is derived (section A.2).

3. Third, four different solutions (i.e., implementations) to the requirements are then examined (sections A.3 through A.6). Each solution is given in an Ada (package) body (implementation) which could be compiled under the required specification. All of the implementation bodies have been coded in Ada with the closest fidelity to the original implementations as could be managed. This will make comparisons between the different implementations easier. It should be noted that if the implementations had been done in Ada originally, the specification and bodies would be a bit different (e.g., the interface would have required that three positive values be supplied). However, the original codings are kept since some of the discussions require it. Any failures to meet the intent of the original algorithms with these codings are entirely the fault of the author of this report. The implementations are compared against the test specifications and test data to see what coverage is provided (data) and what coverage could have been potentially provided (specifications).

4. Fourth, the implementations are compared against each other to compare and contrast how a coverage set adequate for an implementation performs against the others (section A.7).

The points made in the above progression are given in a conclusions section (section A.8). In addition, some further points are made about coverage in general.

Decision tables are a natural way to formally express what is happening in both the requirements and implementations presented in this appendix, since a simple function is being discussed that is entirely driven by logic. For the decision tables which are used in this appendix, the following terminology is used:

T : True
F : False
\- : Don't Care
X : Can't Execute (necessary when discussing implementations)
S_1 : Length of Side 1
S_2 : Length of Side 2
S_3 : Length of Side 3

A.1 Requirements

The requirements for the type-of-triangle problem are as follows:

1. The subprogram shall accept as input three integer values that represent three lengths.

2. The subprogram shall determine if a valid triangle can be formed from the sides of the given lengths. A valid triangle is one where the length of each side is smaller than the sum of the other two.

3. If the three lengths represent an invalid triangle, then the subprogram shall return an indication that the triangle is invalid.

4. If the three lengths represent a valid triangle, then the subprogram shall determine what class of triangle is represented by those sides.

5. If the three lengths represent a valid equilateral triangle, then the subprogram shall return an indication that the triangle is equilateral. An equilateral triangle is one where all three sides are of equal length.

6. If the three lengths represent a valid isosceles triangle, then the subprogram shall return an indication that the triangle is isosceles. An isosceles triangle is one where only two sides are of equal length.

7. If the three lengths represent a valid scalene triangle, then the subprogram shall return an indication that the triangle is scalene. A scalene triangle is one where none of the sides are of equal length.

These requirements can be represented in a decision table. A decision table is composed of two parts: upper and lower. The upper part lists the controlling conditions at the left of the table. Columns are formed listing the combinations of states under which different actions are to be taken by the function. The lower part lists the actions to be taken by the function. In this case, there is only one action to be taken: returning the type of the triangle.

Table A-1 is a decision table (like) representation for the type-of-triangle requirements. The columns have been given a name at the top of the table that will be used in various analyses throughout this appendix. Note that this name does not correlate with the numbers of the textual requirements. This is partly because the textual requirements contain things that cannot be represented in the table and because one textual requirement can be responsible for multiple

columns in the table (e.g., textual requirement 6 corresponds to columns Requirement 5, 6, and 7).

TABLE A-1. TYPE-OF-TRIANGLE REQUIREMENTS DECISION TABLE

	Requirements							
	1	2	3	4	5	6	7	8
$S_1<S_2+S_3$	F	-	-	T	T	T	T	T
$S_2<S_3+S_1$	-	F	-	T	T	T	T	T
$S_3<S_1+S_2$	-	-	F	T	T	T	T	T
$S_1=S_2$	-	-	-	F	F	F	T	T
$S_2=S_3$	-	-	-	F	F	T	F	T
$S_3=S_1$	-	-	-	F	T	F	F	T
return =	Invalid	Invalid	Invalid	Scalene	Isosceles	Isosceles	Isosceles	Equilateral

The Ada (package) specification in figure A-1 is used for all of the implementations discussed in this appendix. This code will be using the Ada name *Length_Of_Side*_No. for the decision table name $S_{\#}$, and the Ada name *Not_A_Triangle* for the decision table name *Invalid*.

```
package Triangles is

    type Triangle_Type is (Not_A_Triangle, Equilateral, Isosceles,
Scalene);

    function Type_Of_Triangle(
        Length_Of_Side_1 : in Integer;
        Length_Of_Side_2 : in Integer;
        Length_Of_Side_3 : in Integer) return Triangle_Type;

end Triangles;
```

FIGURE A-1. TRIANGLES PACKAGE SPECIFICATION

One can now see where the use of Ada would have allowed for slightly different implementations than those provided in this appendix. For instance, the parameters to the Type_Of_Triangle function would probably have been of subtype positive, instead of integer. This would have made certain internal checks performed by different implementations unnecessary.

A.2 Myers' Tests

This section will investigate how the tests that Myers specifies at the beginning of his book [1] compare to coverage of the requirements. These tests will be compared against the coverage they might provide for each of the implementations within the implementation subsections of this appendix. Myers' tests are being used as a convenient point of comparison as they are

readily available. Myers himself acknowledges that these are not a complete set of test specifications, and it is not implied that they are. The comparisons based on these specifications are here to make a point only (i.e., that requirements-based verification, in particular testing, in the absence of structural coverage can be insufficient). The mathematical equations given to define each of Myers' tests will be over specified (i.e., there will be redundant clauses in them). This will make filling out decision tables in the rest of the appendix easier.

The first test that Myers calls for is that of a valid scalene triangle. The valid scalene triangle can be defined as

$$S_1<S_2+S_3 \wedge S_2<S_3+S_1 \wedge S_3<S_1+S_2 \wedge S_1\neq S_2 \wedge S_2\neq S_3 \wedge S_3\neq S_1 \qquad \text{(Myers' test 1)}$$

The second test that Myers calls for is that of a valid equilateral triangle. The valid equilateral triangle can be defined as

$$S_1<S_2+S_3 \wedge S_2<S_3+S_1 \wedge S_3<S_1+S_2 \wedge S_1=S_2 \wedge S_2=S_3 \wedge S_3=S_1 \qquad \text{(Myers' test 2)}$$

The third test that Myers calls for is that of a valid isosceles triangle. The valid isosceles triangle can be defined as

$$S_1<S_2+S_3 \wedge S_2<S_3+S_1 \wedge S_3<S_1+S_2 \wedge ((S_1=S_2 \wedge S_2\neq S_3 \wedge S_3\neq S_1) \vee (S_1\neq S_2 \wedge S_2=S_3 \wedge S_3\neq S_1) \vee (S_1\neq S_2 \wedge S_2\neq S_3 \wedge S_3=S_1)) \qquad \text{(Myers' test 3)}$$

The fourth test that Myers calls for is three valid isosceles triangles such that each permutation of the two equal sides is tried. Notice that satisfaction of any of these permuted isosceles triangles automatically satisfies Myers' test 3. Equation 3 could be removed, for minimality, as it is not telling anything new over the following equations. The valid permuted isosceles triangles can be defined as

$$S_1<S_2+S_3 \wedge S_2<S_3+S_1 \wedge S_3<S_1+S_2 \wedge S_1=S_2 \wedge S_2\neq S_3 \wedge S_3\neq S_1 \qquad \text{(Myers' test 4a)}$$

$$S_1<S_2+S_3 \wedge S_2<S_3+S_1 \wedge S_3<S_1+S_2 \wedge S_1\neq S_2 \wedge S_2=S_3 \wedge S_3\neq S_1 \qquad \text{(Myers' test 4b)}$$

$$S_1<S_2+S_3 \wedge S_2<S_3+S_1 \wedge S_3<S_1+S_2 \wedge S_1\neq S_2 \wedge S_2\neq S_3 \wedge S_3=S_1 \qquad \text{(Myers' test 4c)}$$

The fifth test that Myers calls for is one where one of the sides is of zero length. This can be interpreted in one of two ways: either exactly one side has zero length or at least one side has zero length. These two interpretations will each be defined. Notice that satisfaction of the first interpretation automatically satisfies the second. However, satisfaction of the second does not automatically satisfy the first (e.g., when two sides are zero). These invalid zero length triangles can be defined as follows

$$(S_1=0 \wedge S_2\neq 0 \wedge S_3\neq 0) \vee (S_1\neq 0 \wedge S_2=0 \wedge S_3\neq 0) \vee (S_1\neq 0 \wedge S_2\neq 0 \wedge S_3=0) \qquad \text{(Myers' test 5a)}$$

$$S_1=0 \vee S_2=0 \vee S_3=0 \qquad \text{(Myers' test 5b)}$$

The sixth test that Myers calls for is one where one of the sides is of negative length. This can be interpreted in one of two ways: either exactly one side has negative length or at least one side has negative length. These two interpretations will each be defined. Notice that satisfaction of the first interpretation automatically satisfies the second. However, satisfaction of the second does not automatically satisfy the first (e.g., when two sides are negative). These invalid negative length triangles can be defined as follows

$$(S_1<0 \wedge S_2\geq 0 \wedge S_3\geq 0) \vee (S_1\geq 0 \wedge S_2<0 \wedge S_3\geq 0) \vee (S_1\geq 0 \wedge S_2\geq 0 \wedge S_3<0) \quad \text{(Myers' test 6a)}$$

$$S_1<0 \vee S_2<0 \vee S_3<0 \quad \text{(Myers' test 6b)}$$

The seventh test that Myers calls for is one where all of the sides are positive and one of the sides is of equal length to the sum of the other two. Notice that with three positive numbers, if one is equal to the sum of the other two, then both of the smaller sides (one of the other two) cannot be equal to the large side (the side which equals the sum of the other two) plus the remaining side. This invalid length triangle can be defined as

$$S_1>0 \wedge S_2>0 \wedge S_3>0 \wedge (S_1=S_2+S_3 \vee S_2=S_3+S_1 \vee S_3=S_1+S_2) \quad \text{(Myers' test 7)}$$

The eighth test that Myers calls for is three of the invalid length triangles called for in Myers' test 7 such that all three permutations of the long side is tried. Notice that satisfaction of any of these permutations will automatically satisfy Myers' test 7. Myers' test 7 could removed, for minimality, as it adds nothing over the following definitions. These invalid length permuted triangles can be defined as

$$S_1>0 \wedge S_2>0 \wedge S_3>0 \wedge S_1=S_2+S_3 \quad \text{(Myers' test 8a)}$$

$$S_1>0 \wedge S_2>0 \wedge S_3>0 \wedge S_2=S_3+S_1 \quad \text{(Myers' test 8b)}$$

$$S_1>0 \wedge S_2>0 \wedge S_3>0 \wedge S_3=S_1+S_2 \quad \text{(Myers' test 8c)}$$

The ninth test that Myers calls for is one where all of the sides are positive and one of the sides is of greater length than the sum of the other two. This invalid length triangle can be defined as

$$S_1>0 \wedge S_2>0 \wedge S_3>0 \wedge (S_1>S_2+S_3 \vee S_2>S_3+S_1 \vee S_3>S_1+S_2) \quad \text{(Myers' test 9)}$$

The tenth test that Myers calls for is three of the invalid length triangles called for in Myers' test 9 such that all three permutations of long side (i.e., the side which is greater than the sum of the other two) is tried. Notice that satisfaction of any of these permutations will automatically satisfy Myers' test 9. Meyers' test 9 could be removed, for minimality, as it adds nothing over the following definitions. These invalid length permuted triangles can be defined as

$$S_1>0 \wedge S_2>0 \wedge S_3>0 \wedge S_1>S_2+S_3 \quad \text{(Myers' test 10a)}$$

$$S_1>0 \wedge S_2>0 \wedge S_3>0 \wedge S_2>S_3+S_1 \quad \text{(Myers' test 10b)}$$

$$S_1>0 \wedge S_2>0 \wedge S_3>0 \wedge S_3>S_1+S_2 \quad \text{(Myers' test 10c)}$$

The eleventh test that Myers calls for is one where all sides are of zero length. This invalid length triangle can be defined as

$$S_1=0 \wedge S_2=0 \wedge S_3=0 \qquad \text{(Myers' test 11)}$$

The final two tests that Myers calls for concern errors not permitted in Ada, so they will not be considered as part of this analysis. The first error concerns calling the classification subprogram with objects of the wrong type. This would be detected by the compiler and no executable code would be generated for the offending unit, so no erroneous data could be sent. The second error concerns calling the classification subprogram with the wrong number of arguments. Again, this would be detected and rejected by the compiler.

Table A-2 is a pseudo decision table identifying the overlap of the Myers' test specifications with the requirements. Note that for this decision table, the dot "●" will be used to indicate that Myers' test specification(s) may potentially provide coverage. The requirements are identified using the names that were used in table A-1. Myers' test specifications are identified by the numbering used in the previous specifications (e.g., Myers 4a). To make this more of a proper decision table, the specification rows from table A-1 could be inserted in this table below the specification names.

TABLE A-2. TYPE-OF-TRIANGLE REQUIREMENTS VS MYERS' TESTS

Myers' Tests	Requirements							
	1	2	3	4	5	6	7	8
1				●				
2								●
3					●	●	●	
4a							●	
4b						●		
4c					●			
5a	●	●	●					
5b	●	●	●					
6a	●	●	●					
6b	●	●	●					
7	●	●	●					
8a	●							
8b		●						
8c			●					
9	●	●	●					
10a	●							
10b		●						
10c			●					
11	●	●	●					

The reason it is said that a specification may potentially provide coverage is because it is possible for a test specification to be broad enough that it covers parts of multiple requirements. For

example, consider the overlap between Myers' test specification 5a (exactly one side is of zero length) with requirements specifications 1, 2, and 3 (invalid side(s)). This overlap is graphically demonstrated in figure A-2 with Venn diagrams. A Venn diagram is a graphical way to show the intersections (overlaps) of different sets. Each of these intersections forms a subdomain that can be described by a mathematical equation. The subdomains within the left Venn diagram have been labeled with condition codes which correspond to the three side checks (i.e., ($S_1<S_2+S_3$, $S_2<S_3+S_1$, $S_3<S_1+S_2$)).

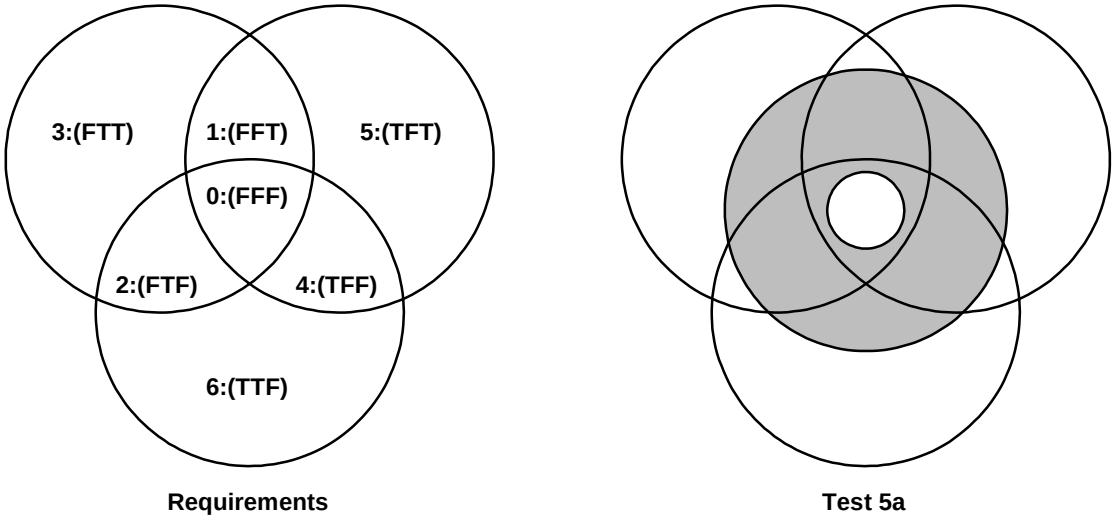


FIGURE A-2. VENN DIAGRAMS FOR INVALID TRIANGLES AND TEST 5a

The three requirements specifications themselves overlap each other as is shown in the left side of the figure (e.g., the invalid triangle (0,0,0) satisfies all three requirements). In all seven subdomains it is possible to have invalid triangles with exactly one of the lengths equal to zero as well as none of the lengths equal to zero (e.g., the invalid triangle (-1,0,-1) and the invalid triangle (-1,-1,-1) are both in subdomain 0:(FFF)). This is shown by the "donut" shaded region in the right-hand side of the figure. The shaded region represents those invalid triangles with only one side of zero length. Each subdomain has shaded and unshaded regions. Where the test data is chosen for 5a determines which one(s) of the requirements are covered.

A check of table A-2 shows that Myers' test specifications provide (simple) coverage of the requirements. This observation is based on the fact that every requirement has at least one dot in its column, and each of Myers' specifications has at least one dot in its row, and most importantly, each requirement (column) has at least one specification (row) where coverage is uniquely provided (i.e., the specification provides coverage of only that requirement, no overlaps with other requirements). If this had not been the case, then it would be possible to provide actual test data that would not cover one or more of the requirements by covering the other overlapping specification in its unique part of the domain overlap. For example, a test satisfying specification 3 could cover either requirements 5, 6, or 7. If specifications 4a, 4b, and 4c were not present, then one could choose test data that would satisfy specification 3 in the 4a part of its domain, which would provide coverage for requirement 7 only.

For other comparison purposes, it is necessary to have a test data set to compare against code since the specification overlap will not always provide at least one test region for each implementation section of code. This clouds the analysis of whether Myers' tests specifications are sufficient to provide coverage of the code. For that reason, the test set detailed in table A-3 was chosen. This table identifies which of Myers' specification(s) are satisfied by the potential triangle test tuple (triple actually), the test tuple itself, and which of the requirements specifications are covered by the test. The test set was chosen to be the smallest possible with the smallest overlap of tests to requirements except for Myers' tests 5 and 6. It is possible to submit one test case for all four specifications (5a, 5b, 6a, and 6b), but this was not done as it did not seem to be the proper approach to coverage of the requirements. Other test sets are possible, and each provides different coverages. Remember that this is being used for comparison purposes only.

TABLE A-3. TYPE-OF-TRIANGLE REQUIREMENTS VS MYERS' TEST SPECIFICATIONS VS TEST TUPLES

Myers' Specification	Test Tuple	Covered Requirements
(1)	(2, 3, 4)	(4)
(2)	(1, 1, 1)	(8)
(3, 4a)	(2, 2, 1)	(7)
(3, 4b)	(1, 2, 2)	(6)
(3, 4c)	(2, 1, 2)	(5)
(5a, 5b)	(0, 2, 1)	(2)
(6a, 6b)	(-1, 3, 1)	(2)
(7, 8a)	(3, 1, 2)	(1)
(7, 8b)	(1, 3, 2)	(2)
(7, 8c)	(1, 2, 3)	(3)
(9, 10a)	(4, 1, 2)	(1)
(9, 10b)	(1, 4, 2)	(2)
(9, 10c)	(1, 2, 4)	(3)
11, 5b)	(0, 0, 0)	(1, 2, 3)

A.3 Implementation No. 1

The first implementation is one suggested by Dyer of IBM [2]. The Ada package body for this implementation is in figure A-3. In order to build a decision table for this implementation, the individual conditions present within the code need to be isolated. This results in the following list for unique conditions:

- Length_Of_Side_1 < Length_Of_Side_2 + Length_Of_Side_3
- Length_Of_Side_2 < Length_Of_Side_3 + Length_Of_Side_1
- Length_Of_Side_3 < Length_Of_Side_1 + Length_Of_Side_2
- Length_Of_Side_1 = Length_Of_Side_2
- Length_Of_Side_2 = Length_Of_Side_3
- Length_Of_Side_3 = Length_Of_Side_1

```
package body Triangles is

    function Type_Of_Triangle(
        Length_Of_Side_1 : in Integer;
        Length_Of_Side_2 : in Integer;
        Length_Of_Side_3 : in Integer) return Triangle_Type is
    begin -- Type_Of_Triangle
        if   (Length_Of_Side_1 < Length_Of_Side_2 + Length_Of_Side_3) and then
             (Length_Of_Side_2 < Length_Of_Side_3 + Length_Of_Side_1) and then
             (Length_Of_Side_3 < Length_Of_Side_1 + Length_Of_Side_2)
        then
            if   (Length_Of_Side_1 = Length_Of_Side_2) and then
                 (Length_Of_Side_2 = Length_Of_Side_3)
            then
                return Equilateral;
            elsif     (Length_Of_Side_1 = Length_Of_Side_2) or else
                 (Length_Of_Side_2 = Length_Of_Side_3) or else
                 (Length_Of_Side_3 = Length_Of_Side_1)
            then
                return Isosceles;
            else
                return Scalene;
            end if;
        else
            return Not_A_Triangle;
        end if;
    end Type_Of_Triangle;

end Triangles;
```

FIGURE A-3. TRIANGLES IMPLEMENTATION NO. 1 BODY

It turns out for this implementation that there are the same numbers of conditions with the same structure as the requirements (adjusting for name equivalence). The decision table for this implementation, using the decision table names for the equivalent Ada names, is shown in table A-4. Note that this table has been annotated in accordance with how the implementation would execute.

Comparing tables A-1 and A-4, a few differences are noticed. The first difference is that the implementation (represented in table A-4) specifies *Can't Execute* in places where the requirements (represented in table A-1) specify *Don't Care*. Most of these occur in the part of the tables where the equality of sides in invalid triangles are compared. This is obviously not a problem, since there is no problem not executing (evaluating) something you do not care about. The other instances occur in the part of the tables where the length of sides in invalid triangles

TABLE A-4. TYPE-OF-TRIANGLE IMPLEMENTATION NO. 1 DECISION TABLE

	Implementation Number							
	1-1	1-2	1-3	1-4	1-5	1-6	1-7	1-8
$S_1<S_2+S_3$	F	T	T	T	T	T	T	T
$S_2<S_3+S_1$	X	F	T	T	T	T	T	T
$S_3<S_1+S_2$	X	X	F	T	T	T	T	T
$S_1=S_2$	X	X	X	F	F	F	T	T
$S_2=S_3$	X	X	X	F	F	T	F	T
$S_3=S_1$	X	X	X	F	T	X	X	X
return =	Invalid	Invalid	Invalid	Scalene	Isosceles	Isosceles	Isosceles	Equilateral

are compared. You will also notice that in this part of the tables some of the requirements *Don't Cares* have been turned into implementation *Trues*. This is not a problem as long as the implementation covers all of the cases required by the requirements.

As the Venn diagrams in figure A-4 show, the implementation is in compliance with the requirements. The subdomains within each of the Venn diagrams have been labeled with condition codes which correspond to the three side checks (i.e., ($S_1<S_2+S_3$, $S_2<S_3+S_1$, $S_3<S_1+S_2$)). Notice that Implementation subdomain 0:(FXX) completely covers the requirements subdomains (0:(FFF), 1:(FFT), 2:(FTF), and 3:(FTT)). This Implementation subdomain covers all cases where the first side is invalid. Implementation subdomain 5:(TFX) completely covers the requirements subdomains (4:(TFF), and 5:(TFT)). This Implementation subdomain covers all cases where the first side is valid and the second side is invalid. Implementation subdomain 6:(TTF) is identical to the requirement subdomain 6:(TTF). This subdomain covers all cases where the first two sides are valid and the third is invalid.

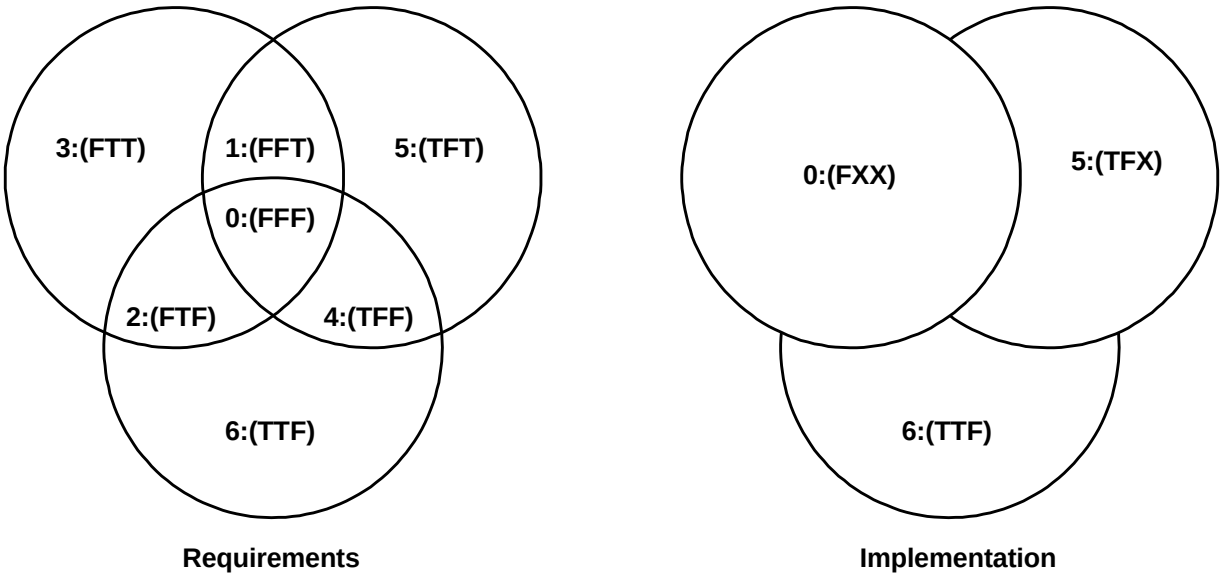


FIGURE A-4. VENN DIAGRAMS FOR INVALID TRIANGLES

Given that it has now been shown that the implementation is in compliance with the requirements, the question arises *how well would requirements-based verification have covered Implementation No. 1?* The analysis for implementation coverage of Myers' tests specifications is given in table A-5. Notice that table A-5 (Myers' Test Specifications vs Implementation No. 1) is identical to table A-2 (Type-Of-Triangle Requirements vs Myers' Tests). This means that Myers' test specifications are guaranteed to provide coverage of this implementation, no matter what test data is chosen to satisfy those specifications. This should come as no surprise since the decision table for the requirements and the decision table for this implementation were essentially the same.

TABLE A-5. MYERS' TEST SPECIFICATIONS VS IMPLEMENTATION NO. 1

Myers' Test	Implementation Number							
	1-1	1-2	1-3	1-4	1-5	1-6	1-7	1-8
1				●				
2								●
3					●	●	●	
4a							●	
4b						●		
4c					●			
5a	●	●	●					
5b	●	●	●					
6a	●	●	●					
6b	●	●	●					
7	●	●	●					
8a	●							
8b		●						
8c			●					
9	●	●	●					
10a	●							
10b		●						
10c			●					
11	●							

The final question that will be explored is *how well do the requirements tests cover Implementation No. 1?* The test specification analysis in table A-5 has already told us that complete coverage is provided by any test data set that satisfies the specifications. However, for this implementation we are interested in the degree of over-coverage (i.e., redundant testing). This analysis is presented in table A-6. As the analysis shows, the test data provide redundant coverage for Implementation Nos. 1-1, 1-2, and 1-3. As table A-5 shows, with Myers' specifications there was no way to avoid this redundancy for this implementation. The results for some of the other implementations will be different.

TABLE A-6. TYPE-OF-TRIANGLE TESTS VS IMPLEMENTATION NO. 1

Test Tuple	Implementation Number							
	1-1	1-2	1-3	1-4	1-5	1-6	. 1-7	1-8
(2, 3, 4)				●				
(1, 1, 1)								●
(2, 2, 1)							●	
(1, 2, 2)						●		
(2, 1, 2)					●			
(0, 2, 1)		●						
(-1, 3, 1)		●						
(3, 1, 2)	●							
(1, 3, 2)		●						
(1, 2, 3)			●					
(4, 1, 2)	●							
(1. 4. 2)		●						
(1, 2, 4)			●					
(0, 0, 0)	●							

A.4 Implementation No. 2

The second implementation is one suggested by the author of this report. The Ada package body for this implementation is in figure A-5.

In order to build a decision table for this implementation, first the individual conditions present within the code need to be isolated. This results in the following list for unique conditions:

- Length_Of_Side_1 > 0
- Length_Of_Side_2 > 0
- Length_Of_Side_3 > 0
- Length_Of_Side_1 > Length_Of_Side_3. Length_Of_Side_2
- Length_Of_Side_2 > Length_Of_Side_1. Length_Of_Side_3
- Length_Of_Side_3 > Length_Of_Side_2. Length_Of_Side_1
- Length_Of_Side_1 = Length_Of_Side_2
- Length_Of_Side_2 = Length_Of_Side_3
- Length_Of_Side_3 = Length_Of_Side_1

For this implementation, there are a different number of conditions with a different structure from those in the requirements (adjusting for name equivalence). The decision table for this implementation, using the decision table names for the equivalent Ada names, is shown in table A-7. Note that this table has been annotated in accordance with how the implementation would execute.

```
package body Triangles is

    function Type_Of_Triangle(
        Length_Of_Side_1 : in Integer;
        Length_Of_Side_2 : in Integer;
        Length_Of_Side_3 : in Integer) return Triangle_Type is
    begin -- Type_Of_Triangle
        if   (Length_Of_Side_1 > 0) and then
             (Length_Of_Side_2 > 0) and then
             (Length_Of_Side_3 > 0) and then
             (Length_Of_Side_1 > Length_Of_Side_3.  Length_Of_Side_2) and then
             (Length_Of_Side_2 > Length_Of_Side_1.  Length_Of_Side_3) and then
             (Length_Of_Side_3 > Length_Of_Side_2.  Length_Of_Side_1)
        then
            if   (Length_Of_Side_1 = Length_Of_Side_2) and then
                 (Length_Of_Side_2 = Length_Of_Side_3)
            then
                 return Equilateral;
            elsif          (Length_Of_Side_1 = Length_Of_Side_2) or else
                 (Length_Of_Side_2 = Length_Of_Side_3) or else
                 (Length_Of_Side_3 = Length_Of_Side_1)
            then
                 return Isosceles;
            else
                 return Scalene;
            end if;
        else
            return Not_A_Triangle;
        end if;
    end Type_Of_Triangle;

end Triangles;
```

FIGURE A-5. TRIANGLE IMPLEMENTATION NO. 2 BODY

After comparing table A-7 for Implementation No. 2 with table A-1 for the requirements, the first question one is likely to have is: *does this implementation satisfy the requirements?* That question really does not come to mind with Implementation No. 1 since that implementation's decision table was nearly identical to the requirement's decision table. The overlap between the requirements and Implementation No. 2 is presented in table A-8. Notice that every column and row has at least one entry within it. This would imply that the implementation is in conformance with the requirements.

To understand better what is happening with Implementation No. 2, first observe that the invalid side tests are present, though in a different format and order from those of the requirements. This is demonstrated in table A-9.

TABLE A-7. TYPE-OF-TRIANGLE IMPLEMENTATION NO. 2 DECISION TABLE

	Implementation Number										
	2-1	2-2	2-3	2-4	2-5	2-6	2-7	2-8	2-9	2-10	2-11
$S_1>0$	F	T	T	T	T	T	T	T	T	T	T
$S_2>0$	X	F	T	T	T	T	T	T	T	T	T
$S_3>0$	X	X	F	T	T	T	T	T	T	T	T
$S_1>S_3-S_2$	X	X	X	F	T	T	T	T	T	T	T
$S_2>S_1-S_3$	X	X	X	X	F	T	T	T	T	T	T
$S_3>S_2-S_1$	X	X	X	X	X	F	T	T	T	T	T
$S_1=S_2$	X	X	X	X	X	X	F	F	F	T	T
$S_2=S_3$	X	X	X	X	X	X	F	F	T	F	T
$S_3=S_1$	X	X	X	X	X	X	F	T	X	X	X
return=	Inv.	Inv.	Inv.	Inv.	Inv.	Inv.	Scal.	Isos.	Isos.	Isos.	Equ.

TABLE A-8. TYPE-OF-TRIANGLE REQUIREMENTS VS IMPLEMENTATION NO. 2

Implementation Number	Requirements							
	1	2	3	4	5	6	7	8
2-1	●	●	●					
2-2	●	●	●					
2-3	●	●						
2-4			●					
2-5	●							
2-6		●						
2-7				●				
2-8					●			
2-9						●		
2-10							●	
2-11								●

TABLE A-9. IMPLEMENTATION NO. 2 VS REQUIREMENTS CONDITIONS TABLE

Implementation No. 2 condition No. 4: $S_1>S_3-S_2$	Requirements condition No. 3: $S_3<S_1+S_2$
Implementation No. 2 condition No. 5: $S_2>S_1-S_3$	Requirements condition No. 1: $S_1<S_2+S_3$
Implementation No. 2 condition No. 6: $S_3>S_2-S_1$	Requirements condition No. 2: $S_2<S_3+S_1$

Second, notice that the side equality tests are the same in both the implementation and the requirements. So any triangles which make it down to the invalid side tests and equality tests will be correctly categorized (i.e., ignoring the first three columns and rows of the table, we have something which looks like Implementation No. 1 and the requirements). All this leaves is the first three implementation conditions. These three exclude any triangle that has any side less than or equal to zero. This is a correct thing to do, since any side that is not positive will cause at least one of the valid sides tests to fail. The proof of this is given in the following case analysis.

Only the basic relations between S_1, S_2 and S_3 will be given. The other relations can be formed by exchanging the subscripts through all the possible combinations.

1. $S_1 = S_2 = S_3 = 0$

   For this case, all the valid side tests will fail since $0 = 0 + 0$ (i.e., $S_1 = S_2 + S_3$).

2. $S_1 = S_2 = 0, S_3 < 0$

   For this case, the valid side tests for S_1 and S_2 will fail since 0 is greater than any negative number (S_3).

3. $S_1 = S_2 = 0, S_3 > 0$

   For this case, the valid side test for S_3 will fail since 0 is less than any positive number (i.e., $S_3 > 0 + 0$).

4. $S_1 = 0, S_2 < 0, S_3 < 0$

   For this case, the valid side test for S_1 will fail since 0 is greater than any negative number ($S_2 + S_3$).

5. $S_1 = 0, S_2 < 0, S_3 > 0$

   For this case, the valid side test for S_3 will fail since any positive number (S_3) is greater than any negative number (S_2).

6. $S_1 = 0, S_2 > 0, S_3 > 0, S_2 \leq S_3$

   For this case, the valid side test for S_3 will fail since any positive number (S_3) cannot be less than itself or any lesser positive number (S_2).

7. $S_1 < 0, S_2 < 0, S_3 > 0$

   For this case, the valid side test for S_3 will fail since any positive number (S_3) is greater than any negative number ($S_1 + S_2$).

8. $S_1 < 0, S_2 > 0, S_3 > 0, S_2 \leq S_3$

   For this case, the valid side test for S_3 will fail since any positive number (S_3) cannot be less than any lesser number ($S_1 + S_2$).

9. $S_1 < 0, S_2 < 0, S_3 < 0$

   We consider three subcases for this case

9a. $S_1 = S_2 = S_3$

   For this case, all the valid side tests will fail since any negative number is greater than twice itself.

9b. $S_1 = S_2 \neq S_3$

For this case, the valid side tests for S_1 and S_2 will fail since any negative number is greater than itself and a nonzero decrement.

9c. $S_1 < S_2 < S_3$

For this case, the valid side test for S_3 will fail since any negative number (S_3) is greater than any lesser negative number ($S_1 + S_2$).

Now that it has been shown that this implementation satisfies the requirements, why does it look so different from the first implementation that also satisfies the requirements? What this second implementation has done is to guard against the possibility of a constraint error due to overflow. Notice that if two of the sides are of such a length that their sum exceeds the maximum (minimum) representable integer value, Ada will raise the Constraint_Error exception. This is the only implementation that is not susceptible to this error. Notice that the test set chosen for this example does not expose this failure mode of the other three implementations. This is one instance where the test set is not complete.

The next question to explore is *do Myers' test specifications cover this implementation?* The answer to this question is not as obvious as it was with Implementation No. 1. The overlap analysis is presented in table A-10.

TABLE A-10. MYERS' TEST SPECIFICATIONS VS IMPLEMENTATION NO. 2

Myers' Test	Implementation Number										
	2-1	2-2	2-3	2-4	2-5	2-6	2-7	2-8	2-9	2-10	2-11
1							●				
2											●
3								●	●	●	
4a										●	
4b									●		
4c								●			
5a	●	●	●								
5b	●	●	●								
6a	●	●	●								
6b	●	●	●								
7				●	●	●					
8a					●						
8b						●					
8c				●							
9				●	●	●					
10a					●						
10b						●					
10c				●							
11	●										

At first glance of table A-10, it appears that Myers' test specifications would cover this implementation. However, only Myers' specifications (3, 7, and 9) apply to multiple columns simultaneously. Myers' specifications (5a, 5b, 6a, and 6b) can only apply to a single column with a single test. This means that unless one is careful in choosing those tests, implementation (Nos. 2-2 and 2-3) may not be covered if the tests chosen for those specifications cover Implementation No. 2-1.

The final question to explore is *do the requirements tests cover Implementation No. 2?* This analysis is presented in table A-11. As the analysis shows, the test data did not provide coverage for implementation (Nos. 2-2 and. 2-3). Recall from the previous analysis (table A-10) that if different tests had been chosen more carefully, coverage of these two implementation conditions could have been achieved.

TABLE A-11. TYPE-OF-TRIANGLE TESTS VS IMPLEMENTATION NO. 2

Test Tuple	Implementation Number										
	2-1	2-2	2-3	2-4	2-5	2-6	2-7	2-8	2-9	2-10	2-11
(2, 3, 4)							●				
(1, 1, 1)											●
(2, 2, 1)										●	
(1, 2, 2)									●		
(2, 1, 2)								●			
(0, 2, 1)	●										
(-1, 3, 1)	●										
(3, 1, 2)					●						
(1, 3, 2)						●					
(1, 2, 3)				●							
(4, 1, 2)					●						
(1. 4. 2)						●					
(1, 2, 4)				●							
(0, 0, 0)	●										

A.5 Implementation No. 3

The third implementation is one suggested by Hedley and Hennell [2,3]. The Ada package body for this implementation is shown in figure A-6. This implementation differs from the previous two in that there is a temporary variable that is used to keep track of the number of matching pairs of sides it finds. Note that there are zero matching pairs of sides for a scalene triangle, one matching pairs of sides for an isosceles triangle (either (S_1, S_2) or (S_2, S_3) or (S_3, S_1)), and three matching pairs of sides for an equilateral triangle. The Ada variable cannot really take on the value of two unless there has been an error in the program. Here is where an extension to utilize the features of Ada would have been appropriate. However, in keeping with the original implementation, the case statement tests for zero and one and defaults for the others.

```
package body Triangles is

    function Type_Of_Triangle(
        Length_Of_Side_1 : in Integer;
        Length_Of_Side_2 : in Integer;
        Length_Of_Side_3 : in Integer) return Triangle_Type is
        Number_Of_Matching_Pairs : Integer range 0..3 := 0;
    begin -- Type_Of_Triangle
        if    (Length_Of_Side_1 < Length_Of_Side_2 + Length_Of_Side_3) and then
              (Length_Of_Side_2 < Length_Of_Side_3 + Length_Of_Side_1) and then
              (Length_Of_Side_3 < Length_Of_Side_1 + Length_Of_Side_2)
        then
            if (Length_Of_Side_1 = Length_Of_Side_2) then
                Number_Of_Matching_Pairs := Number_Of_Matching_Pairs + 1;
            end if;
            if (Length_Of_Side_2 = Length_Of_Side_3) then
                Number_Of_Matching_Pairs := Number_Of_Matching_Pairs + 1;
            end if;
            if Length_Of_Side_3 = Length_Of_Side_1 then
                Number_Of_Matching_Pairs := Number_Of_Matching_Pairs + 1;
            end if;
            case Number_Of_Matching_Pairs is
                when 0 =>
                    return Scalene;
                when 1 =>
                    return Isosceles;
                when others =>
                    return Equilateral;
            end case;
        else
            return Not_A_Triangle;
        end if;
    end Type_Of_Triangle;

end Triangles;
```

FIGURE A-6. TRIANGLES IMPLEMENTATION NO. 3 BODY

In order to build a decision table for this implementation, the individual conditions present within the code need to be isolated. This results in the following list for unique conditions:

- Length_Of_Side_1 < Length_Of_Side_2 + Length_Of_Side_3
- Length_Of_Side_2 < Length_Of_Side_3 + Length_Of_Side_1
- Length_Of_Side_3 < Length_Of_Side_1 + Length_Of_Side_2
- Length_Of_Side_1 = Length_Of_Side_2
- Length_Of_Side_2 = Length_Of_Side_3
- Length_Of_Side_3 = Length_Of_Side_1
- Number_Of_Matching_Pairs = 0
- Number_Of_Matching_Pairs = 1
- Number_Of_Matching_Pairs in 2..3

For this implementation, there are a different number of conditions from those in the requirements (adjusting for name equivalence). The first six conditions are identical between the requirements and the implementation. The final three conditions are unique to the implementation. However, these are not independent conditions; therefore, they do not change the decision table in any significant way (other than to add rows). The decision table for this implementation, using the decision table names for the equivalent Ada names, is shown in table A-12. Note that this table has been annotated in accordance with how the implementation would execute.

TABLE A-12. TYPE-OF-TRIANGLE IMPLEMENTATION NO. 3 DECISION TABLE

	Implementation Number							
	3-1	3-2	3-3	3-4	3-5	3-6	3-7	3-8
$S_1<S_2+S_3$	F	T	T	T	T	T	T	T
$S_2<S_3+S_1$	X	F	T	T	T	T	T	T
$S_3<S_1+S_2$	X	X	F	T	T	T	T	T
$S_1=S_2$	X	X	X	F	F	F	T	T
$S_2=S_3$	X	X	X	F	F	T	F	T
$S_3=S_1$	X	X	X	F	T	F	F	T
M=0	X	X	X	T	X	X	X	X
M=1	X	X	X	X	T	T	T	X
M in 2..3	X	X	X	X	X	X	X	T
return =	Invalid	Invalid	Invalid	Scalene	Isosceles	Isosceles	Isosceles	Equilateral

After comparing table A-12 for Implementation No. 3 with table A-1 for the requirements, the first question one is likely to have is *does this implementation satisfy the requirements?* To answer that question, consider that the final three rows in table A-12 do not add any new information, so they could be removed from the table. If that is done, it was found that the decision table for Implementations No. 1 (table A-4) and No. 3 (table A-12) are equivalent. They are not identical in that Implementation No. 1 does not execute the final side equality test ($S_3 = S_1$) in the final three columns. However, if we were to put the values that would be computed by Implementation No. 1 into table A-4, then the two tables would be identical. This means that this implementation does satisfy the requirements, exactly as Implementation No. 1 does. It also has the same test specification overlap and test case overlap as Implementation No. 1 does. The code, however, looks entirely different between the two implementations.

A.6 Implementation No. 4

The fourth and final implementation is one suggested by TRW [2,4]. The Ada package body for this implementation is shown in figure A-7. Notice that this code has been formatted differently from the previous examples in order to fit on a single page. This implementation has a test to check that all sides are positive in accordance with the assumption made by TRW that only positive sides would be submitted. This is the only extension made to this implementation.

```
package body Triangles is

    function Type_Of_Triangle(
        Length_Of_Side_1 : in Integer;
        Length_Of_Side_2 : in Integer;
        Length_Of_Side_3 : in Integer) return Triangle_Type is
        Match : Integer range 0..6 := 0;
    begin -- Type_Of_Triangle
        if    (Length_Of_Side_1 > 0) and then
              (Length_Of_Side_2 > 0) and then
              (Length_Of_Side_3 > 0)
        then
            if Length_Of_Side_1 = Length_Of_Side_2 then Match := Match + 1; end if;
            if Length_Of_Side_2 = Length_Of_Side_3 then Match := Match + 2; end if;
            if Length_Of_Side_3 = Length_Of_Side_1 then Match := Match + 3; end if;
            case Match is
                when 0 =>
                    if    (Length_Of_Side_1 >= Length_Of_Side_2 + Length_Of_Side_3) or else
                          (Length_Of_Side_2 >= Length_Of_Side_3 + Length_Of_Side_1) or else
                          (Length_Of_Side_3 >= Length_Of_Side_1 + Length_Of_Side_2)
                    then return Not_A_Triangle;
                    else return Scalene;
                    end if;
                when 1 =>
                    if Length_Of_Side_3 >= Length_Of_Side_1 + Length_Of_Side_2
                    then return Not_A_Triangle;
                    else return Isosceles;
                    end if;
                when 2 =>
                    if Length_Of_Side_1 >= Length_Of_Side_2 + Length_Of_Side_3
                    then return Not_A_Triangle;
                    else return Isosceles;
                    end if;
                when 3 =>
                    if Length_Of_Side_2 >= Length_Of_Side_3 + Length_Of_Side_1
                    then return Not_A_Triangle;
                    else return Isosceles;
                    end if;
                when others =>
                    return Equilateral;
            end case;
        else
            return Not_A_Triangle;
        end if;
    end Type_Of_Triangle;

end Triangles;
```

FIGURE A-7. TRIANGLES IMPLEMENTATION NUMBER 4 BODY

This implementation is similar to Implementation No. 3 in that it also has a temporary variable that is used to keep track of the matching pairs of sides it finds. Unlike Implementation No. 3

where the variable was a counter of how many matches it found, in Implementation No. 4 the variable is used to keep track of which combinations of matching pairs were found. Note that there are zero matches of sides for a scalene triangle; one, two or three matches of sides for an isosceles triangle (either one for (S_1, S_2) or two for (S_2, S_3) or three for (S_3, S_1)) and six matches of sides for an equilateral triangle. The Ada variable cannot really take on the value of four or five unless there has been an error in the program. Here is where an extension to utilize the features of Ada would have been appropriate. However, in keeping with the original implementation, the case statement tests for zero, one, two, and three matches and defaults for the others.

In order to build a decision table for this implementation, first the individual conditions present within the code need to be isolated. This results in the following list for unique conditions

- Length_Of_Side_1 > 0
- Length_Of_Side_2 > 0
- Length_Of_Side_3 > 0
- Length_Of_Side_1 = Length_Of_Side_2
- Length_Of_Side_2 = Length_Of_Side_3
- Length_Of_Side_3 = Length_Of_Side_1
- Match = 0
- Match = 1
- Match = 2
- Match = 3
- Match in 5..6
- Length_Of_Side_1 >= Length_Of_Side_2 + Length_Of_Side_3
- Length_Of_Side_2 >= Length_Of_Side_3 + Length_Of_Side_1
- Length_Of_Side_3 >= Length_Of_Side_1 + Length_Of_Side_2

For this implementation, there are a different number of conditions with a different structure from those in the requirements (adjusting for name equivalence). The decision table for this implementation, using the decision table names for the equivalent Ada names, is shown in table A-13. Note that we have annotated this table in accordance with how the implementation would execute.

After comparing table A-13 for Implementation No. 4 with table A-1 for the requirements, the first question one is likely to have is *does this implementation satisfy the requirements?* This question is not as easy to answer for this implementation as it has been for the previous three implementations. The overlap between the Requirements and Implementation No. 4 is presented in table A-14. Notice that every column and row has at least one entry within it. This would imply that the implementation is in conformance with the requirements.

To understand better what is happening with Implementation No. 4, first observe that just as in Implementation No. 2, Implementation No. 4 screens out those potential triangles where not all three sides are positive (i.e., > 0). It correctly identifies these as being invalid triangles (reference the argument in A.4 for Implementation No. 2). Second, the side equality tests are present, though in a different processing sequence than previously seen. Implementation No. 4

TABLE A-13 TYPE-OF-TRIANGLE IMPLEMENTATION NO. 4 DECISION TABLE

	Implementation Number													
	4-1	4-2	4-3	4-4	4-5	4-6	4-7	4-8	4-9	4-10	4-11	4-12	4-13	4-14
$S_1>0$	F	T	T	T	T	T	T	T	T	T	T	T	T	T
$S_2>0$	X	F	T	T	T	T	T	T	T	T	T	T	T	T
$S_3>0$	X	X	F	T	T	T	T	T	T	T	T	T	T	T
$S_1=S_2$	X	X	X	F	F	F	F	F	F	F	F	T	T	T
$S_2=S_3$	X	X	X	F	F	F	F	F	F	T	T	F	F	T
$S_3=S_1$	X	X	X	F	F	F	F	T	T	F	F	F	F	T
M=0	X	X	X	T	T	T	T	X	X	X	X	X	X	X
M=1	X	X	X	X	X	X	X	X	X	X	X	T	T	X
M=2	X	X	X	X	X	X	X	X	X	T	T	X	X	X
M=3	X	X	X	X	X	X	X	T	T	X	X	X	X	X
M in 4..6	X	X	X	X	X	X	X	X	X	X	X	X	X	T
$S_1\geq S_2+S_3$	X	X	X	F	F	F	T	X	X	F	T	X	X	X
$S_2\geq S_3+S_1$	X	X	X	F	F	T	X	F	T	X	X	X	X	X
$S_3\geq S_1+S_2$	X	X	X	F	T	X	X	X	X	X	X	F	T	X
return=	Inv.	Inv.	Inv.	Scal	Inv.	Inv.	Inv.	Isos	Inv.	Isos	Inv.	Isos	Inv.	Equ.

TABLE A-14 TYPE-OF-TRIANGLE REQUIREMENTS VS IMPLEMENTATION NO. 4

Implementation Number	Requirements Number							
	1	2	3	4	5	6	7	8
4-1	●	●	●					
4-2	●	●	●					
4-3	●	●						
4-4				●				
4-5			●					
4-6		●						
4-7	●							
4-8					●			
4-9		●						
4-10						●		
4-11	●							
4-12							●	
4-13			●					
4-14								●

first checks for all positive sides, then categorizes the type of the triangle second, and then determines if it is valid third. In all previous implementations, the validity check was done first, followed by the type classification. These differences are shown in figure A-8, where the approach taken in the first three implementations is shown on the left, while the alternative is shown on the right.

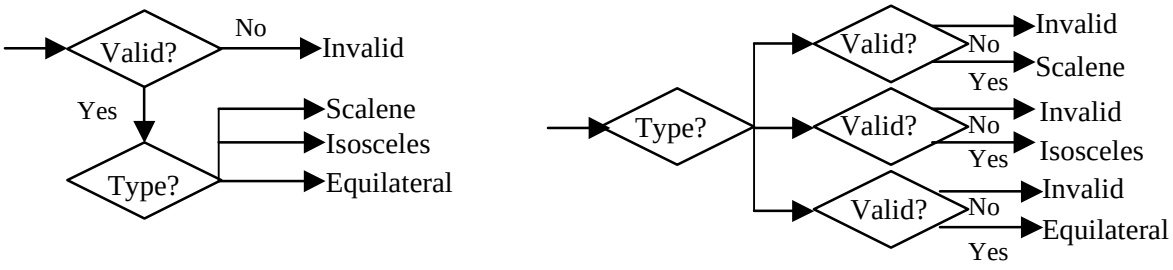


FIGURE A-8. ALTERNATIVE IMPLEMENTATION APPROACHES FOR TYPE-OF-TRIANGLE PROBLEM

What needs to be determined at this point is if implementation (No. 4) will channel the right type of triangle to the right kind of validity check. This breaks down into two questions

1. Does the type classification occur correctly?
2. Are the validity checks correct for each triangle type?

It is easy to see that the type classification will occur correctly, since the same side equality conditions are used in both the requirements and the implementation (i.e., no sides equal is scalene, two sides equal is isosceles, all sides equal is equilateral). The implementation goes a bit beyond the requirements in that it treats each of the isosceles cases (i.e., $S_1=S_2$, $S_2=S_3$, $S_3=S_1$) individually.

Given that the type classification is working correctly, the final point to consider is are the correct side invalidity checks present for the different triangle types (scalene, isosceles$_1$, isosceles$_2$, isosceles$_3$, and equilateral). First, notice that the invalid sides tests are present, though in a different format from those of the requirements. This is demonstrated in table A-15.

TABLE A-15. IMPLEMENTATION NO. 4 VS REQUIREMENTS CONDITIONS TABLE

Implementation No. 4 condition No. 12: $S_1 \geq S_2+S_3$	Requirements condition No. 1: $S_1<S_2+S_3$
Implementation No. 4 condition No. 13: $S_2 \geq S_3+S_1$	Requirements condition No. 2: $S_2<S_3+S_1$
Implementation No. 4 condition No. 14: $S_3 \geq S_1+S_2$	Requirements condition No. 3: $S_3<S_1+S_2$

Notice that the side validity tests are the converse of each other. Also notice that if the implementation conditions are True, the triangle is classified as invalid. This conforms to the requirements since if any of those (requirements) conditions fail (i.e., are False), the triangle is classified as invalid. All three-side validity checks are present for scalene triangles, so scalene triangles will be properly classified. No side validity checks are needed (since all sides are positive) nor present for equilateral triangles, so equilateral triangles are classified correctly. This leaves the three forms of isosceles triangles. All that is necessary for them is to ensure that the nonequal side is less than the sum of the two equal sides. These checks are present where they need to be. Therefore, it can be concluded that this implementation does indeed satisfy the requirements.

The next question to explore is *do Myers' test specifications cover this implementation?* The answer to this question is not as obvious as it was with any of the previous implementations. The overlap analysis is presented in table A-16.

TABLE A-16. MYERS' TEST SPECIFICATIONS VS IMPLEMENTATION NO. 4

Myers' Test	Implementation Number													
	4-1	4-2	4-3	4-4	4-5	4-6	4-7	4-8	4-9	4-10	4-11	4-12	4-13	4-14
1				●										
2														●
3								●		●		●		
4a												●		
4b										●				
4c								●						
5a	●	●	●											
5b	●	●	●											
6a	●	●	●											
6b	●	●	●											
7					●	●	●		●		●		●	
8a							●				●			
8b						●			●					
8c					●								●	
9					●	●	●		●		●		●	
10a							●				●			
10b						●			●					
10c					●								●	
11	●													

At first glance of table A-16, it appears that Myers' test specifications would cover this implementation. However, only Myers' specifications 3, 7, and 9 apply to multiple columns simultaneously. Myers' specifications 5a, 5b, 6a, and 6b can only apply to a single column with a single test. This means that unless one is careful in choosing those tests, Implementation Nos. 4-2 and 4-3 may not be covered if the tests chosen for those specifications cover Implementation No. 4-1. Myers' specifications 8a and 10a can only apply to a single column with a single test. This means that unless one is careful in choosing those tests, only one of Implementation Nos. 4-7 and 4-11 will be covered. The same applies for Myers' 8b and 10b and Implementation Nos. 4-6 and 4-9 as well as Myers' 8c and 10c and Implementation Nos. 4-5 and 4-13.

The final question to explore is *do the requirements tests cover Implementation No. 4?* This analysis is presented in table A-17. As the analysis shows, the test data did not provide coverage for Implementation Nos. 4-2, 4-3, 4-9, 4-11, and 4-13. Recall from the previous analysis (table A-16) that if different tests had been chosen more carefully, coverage of these two implementation conditions could have been achieved.

TABLE A-17 TYPE-OF-TRIANGLE TESTS VS IMPLEMENTATION NO. 4

	Implementation Number													
	4-1	4-2	4-3	4-4	4-5	4-6	4-7	4-8	4-9	4-10	4-11	4-12	4-13	4-14
(2, 3, 4)				●										
(1, 1, 1)														●
(2, 2, 1)												●		
(1, 2, 2)										●				
(2, 1, 2)								●						
(0, 2, 1)	●													
(-1, 3, 1)	●													
(3, 1, 2)							●							
(1, 3, 2)						●								
(1, 2, 3)					●									
(4, 1, 2)							●							
(1. 4. 2)						●								
(1, 2, 4)					●									
(0, 0, 0)	●													

A.7 Implementation Versus Implementation Coverage

This section of the appendix, will look at the question *given a covering set for one implementation, what degree of coverage of the other implementations will be achieved?* Implementations Nos. 1 and 3 can be considered to be identical, since they have the same essential decision table that specifies them. This means that implementations Nos. 1, and 2, Nos. 1, and 4), and Nos. 2, and 4 will be compared.

The overlap table for implementations Nos. 1, and 2 is presented in table A-18. Examination of the table shows that if tests are generated for Implementation No. 1, Implementation No. 2 cannot get coverage. This makes sense as Implementation No. 1 only requires eight tests, while Implementation No. 2 requires eleven. If tests are generated for Implementation No. 2, coverage of Implementation No. 1 is guaranteed and, in fact, will over cover Implementation No. 1 (i.e., some of the tests will provide redundant coverage). Recall that only simple coverage of the decision table are being discussed (i.e., one test for every column).

TABLE A-18. IMPLEMENTATION NO. 1 VS IMPLEMENTATION NO. 2

Implementation Number	Implementation Number										
	2-1	2-2	2-3	2-4	2-5	2-6	2-7	2-8	2-9	2-10	2-11
1-1	●	●	●		●						
1-2	●		●			●					
1-3	●	●		●							
1-4							●				
1-5								●			
1-6									●		
1-7										●	
1-8											●

The overlap table for Implementations Nos. 1 and 4 is presented in table A-19. Examination of the table shows that if tests are generated for Implementation No. 1, Implementation No. 4 we cannot get coverage. This makes sense as Implementation No. 1 only requires eight tests, while Implementation No. 4 requires fourteen. If tests are generated for Implementation No. 4, coverage of Implementation No. 1 is guaranteed and, in fact, will over cover Implementation No. 1 (i.e., some of the tests will provide redundant coverage).

TABLE A-19. IMPLEMENTATION NO. 1 VS IMPLEMENTATION NO. 4

Implementation Number	Implementation Number													
	4-1	4-2	4-3	4-4	4-5	4-6	4-7	4-8	4-9	4-10	4-11	4-12	4-13	4-14
1-1	●	●	●				●				●			
1-2	●		●			●			●					
1-3	●	●			●								●	
1-4				●										
1-5								●						
1-6										●				
1-7												●		
1-8														●

The overlap table for implementations Nos. 2 and. 4 is presented in table A-20. Examination of the table shows that if tests are generated for Implementation No. 2, Implementation No. 4 cannot get coverage. This makes sense as Implementation No. 2 only requires eleven tests, while Implementation No. 4 requires fourteen. If tests are generated for Implementation No. 4, coverage of Implementation No. 2 is guaranteed and, in fact, will over cover Implementation No. 2 (i.e., some of the tests will provide redundant coverage).

TABLE A-20. IMPLEMENTATION NO. 2 VS IMPLEMENTATION NO. 4

Implementation Number	Implementation Number													
	4-1	4-2	4-3	4-4	4-5	4-6	4-7	4-8	4-9	4-10	4-11	4-12	4-13	4-14
2-1	●													
2-2		●												
2-3			●											
2-4					●								●	
2-5							●				●			
2-6						●			●					
2-7				●										
2-8								●						
2-9										●				
2-10												●		
2-11														●

A.8 Conclusions

The analysis in this appendix has shown that structural coverage is needed to assist with requirements-based testing, even for something as small as a single function. For this example, it would have been possible to choose a test set that would have covered all of the implementations. That was not done to drive home the point that verification that is good for the requirements is not necessarily good for an implementation. It was also shown that verification that is good for one implementation is not necessarily good for another. This was actually shown in two directions: either the verification could be shy of what was actually needed or it could be overkill. Since the verification should be cost-effective, over-verification (especially testing!) should be avoided as much as possible. A verification that is adequate for every possible implementation does not need to be conducted when the implementation only needs a fraction of what every possible implementation requires.

It was discovered that Implementation No. 2 had, in essence, a derived requirement to avoid overflow. It was shown that the test set in use was not adequate to detect this error. It would have been possible to have chosen test data for both sides of the problem (exceeding the minimum and maximum representable integers) without increasing the size of the test set we used. This was not done in order to drive home another point. There cannot be verification (especially testing) which is sensitive to all errors. This is especially true when no knowledge of the implementation is used, since some errors are not possible in certain implementations.

In this case, coverage helped to uncover a missing requirement. This requirement could either be a requirement on the system itself, or a verification requirement. Coverage is not guaranteed to do that in all cases, so it should not be relied on for that task. Coverage will only uncover missing requirements when those requirements correspond to uncovered code. Coverage will be of no help when the missing requirements correspond to missing code. However, what coverage did show is that our requirements-based verification was deficient in one area, and that deficiency should be examined. Perhaps the requirements themselves were at fault? What other ways could they be at fault due to the same form of oversight? Was the verification process itself at fault? How could this be prevented in the future?

It is through this directed and focused investigation of the development process started by inadequate coverage that the greatest contribution of coverage can be made. Coverage is not about finding errors in the product per se, although it may do that. Coverage is about finding deficiencies in the development process (why did the verification not get here?) and weakly assessing the adequacy of the verification (only 80% of the coverage conditions was satisfied).

A.9 References

1. Myers, G.J., “The Art of Software Testing,” John Wiley & Sons, New York, 1979.

2. Dyer, M., “The Cleanroom Approach to Quality Software Development,” John Wiley & Sons, New York, 1992.

3. Headley, D. and Hennell, M.A., "The Causes and Effects of Infeasible Paths in Computer Programs," Proceedings of the 8th International Conference on Software Engineering (ICSE85), 1985, pp. 259-266.

4. Brown, J.R. and Lipow, M., "Testing for Software Reliability," Proceedings – 1975 International Conference on Reliable Software, SIGPLAN Notices, Vol. 10, No. 6, June 1975, pp. 518-527.

APPENDIX B—MCDC AS A (WEAK) MEASURE OF EQUIVALENCE CLASS COVERAGE

In this appendix it is shown how MCDC can be thought of as a weak measure of equivalence class coverage and boundary value analysis. To understand this, consider that every condition in an expression is dividing the input space into two regions. If the expression consists of a single input variable and *N* conditions of that variable, then the expression divides (partitions) the input space into at least $N + 1$ regions (subdomains). MCDC requires each side of these partitions to be visited.

B.1 Examples

Figures B-1 and B-3 demonstrate this with a computer-math number line. Recall that a computer can only represent a finite set of numbers, hence the bounds between MinInt (minimum representable integer) and MaxInt (maximum representable integer), and the use of the term computer-math. In all of the figures in this appendix, a dashed line will be used for partition boundaries. This differs from the mathematical convention of a solid line for a closed boundary and a dashed line for an open boundary.

In figure B-1 there is a single condition which divides the number line into two regions, the one which satisfies the condition $X \geq 0$ (dark) and the one that does not (light). The regions have been labeled as True and False respectively.

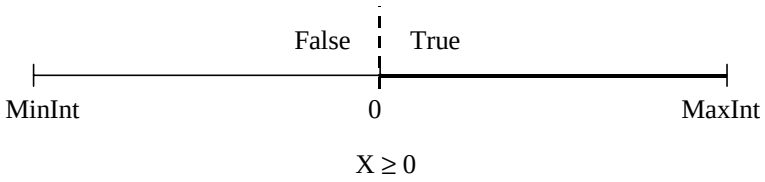


FIGURE B-1. SINGLE VARIABLE SUBDOMAIN PARTITIONING DUE TO A SINGLE CONDITION

Note that Decision Coverage and all forms of MCDC will require both of the subdomains in figure B-1 to be visited. Statement Coverage is not guaranteed to visit both regions unless the decision containing the expression uniquely leads to a statement that requires execution. To demonstrate this, consider the code fragments in figure B-2. The code on the left only requires the decision to be True to achieve Statement Coverage, hence it only needs to visit one of the subdomains in figure B-1, while the code on the right requires both subdomains to be visited.

```
if condition then
    do_something;
end if;
```

```
if condition then
    do_something;
else
    do_something_else;
end if;
```

FIGURE B-2. CODE STRUCTURES FOR COVERAGE COMPARISONS

In figure B-3 the number line is divided into three regions by two conditions. For this example, there are two regions which do not satisfy the condition: one where X is less than the allowed range ($False_1$) and one where X is greater than the allowed range ($False_2$). There is also the subdomain where X is in the allowed range (True, dark line segment).

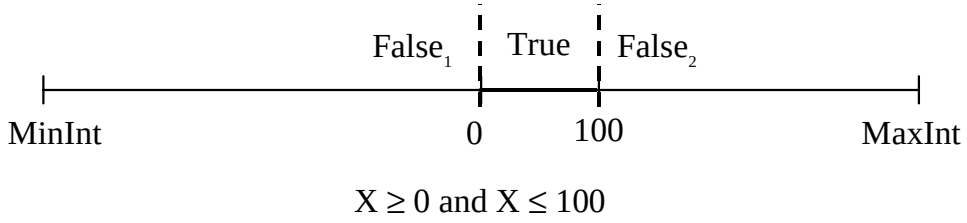


FIGURE B-3. SINGLE VARIABLE SUBDOMAIN PARTITIONING DUE TO TWO CONDITIONS

Note that all forms of MCDC will require all three of the subdomains in figure B-3 to be visited. Visiting both sides of the (True, $False_1$) boundary demonstrates the independence of the ($X \geq 0$) condition. This is because only that condition changes value when we cross the boundary (i.e., X is still ≤ 100 when it becomes negative). Similarly, visiting both sides of the (True, $False_2$) boundary demonstrates the independence of the ($X \leq 100$) condition. Decision Coverage will only require the visitation of the True subdomain and one of the False ($False_1$, $False_2$) subdomains. In this example, Decision Coverage will only demonstrate a single condition's independence. This is not always guaranteed, as the following examples demonstrate. Statement Coverage for the left code fragment of figure B-2 only requires the decision to be True to achieve Statement Coverage, hence it only needs to visit one of the subdomains in figure B-3, while the code fragment on the right of figure B-2 requires Decision Coverage.

Figure B-4 shows subdomain partitioning for two variables due to two conditions. As the figure shows, there are now four subdomains. These subdomains have been labeled with condition codes that correspond to the two conditions (i.e., ($X \geq 10$, $Y \geq 10$)). The one True region has been shaded, while the three False regions are left unshaded.

Note that all forms of MCDC will require that subdomains (1:(FT), 2:(TF), and 3:(TT)) be visited in figure B-4. Notice that the (3:(TT) and 1:(FT)) pair demonstrates the independence of the ($X \geq 10$) condition, while the (3:(TT) and 2:(TF)) pair demonstrates the independence of the ($Y \geq 10$) condition. Decision Coverage will require that two subdomains be visited, the True (3:(TT)) subdomain and one of the False subdomains (0:(FF), 1:(FT) and 2:(TF)). For this example, Decision Coverage is not guaranteed to demonstrate any condition's independence as (3:(TT) and 0:(FF)) is an acceptable test set. Statement Coverage in the weakest sense will only require that one of the subdomains be visited.

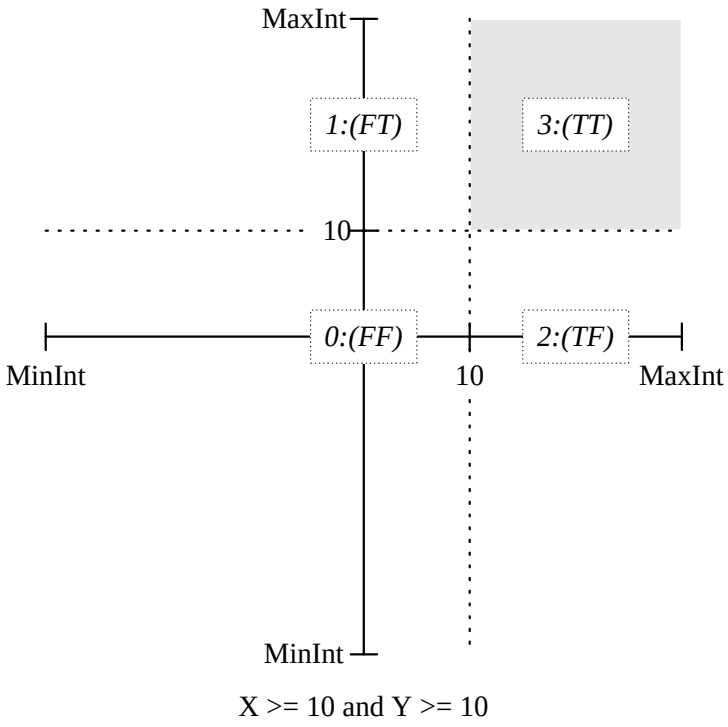
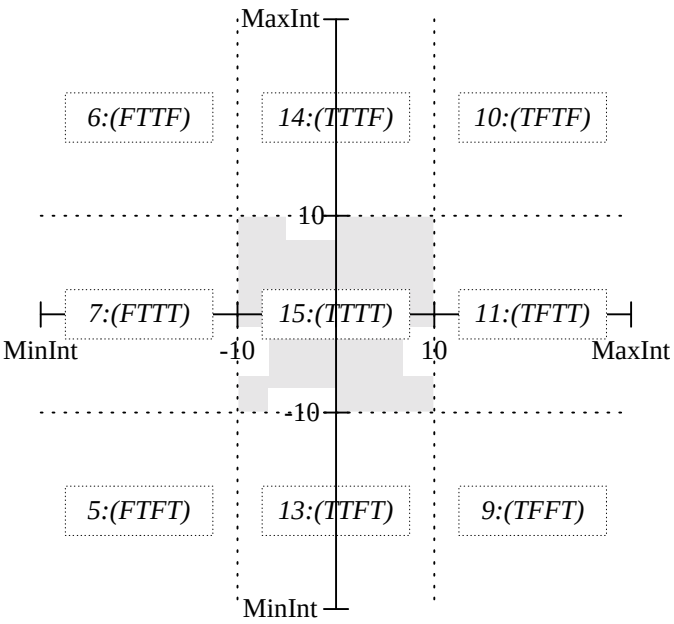


FIGURE B-4. DOUBLE VARIABLE SUBDOMAIN PARTITIONING DUE TO TWO CONDITIONS—TRUE REGION

Figure B-5 shows subdomain partitioning for two variables due to four conditions resulting in a single True region. As the figure shows, there are now nine subdomains. These subdomains have been labeled with condition codes that correspond to the four conditions (i.e., ($X \geq -10$, $X \leq 10$, $Y \geq -10$, $Y \leq 10$)). The one True region has been shaded, while the eight False regions are left unshaded.

Note that all forms of MCDC will require that subdomains (7:(FTTT), 11:(TFTT), 13:(TTFT), 14:(TTTF), and 15:(TTTT)) be visited in figure B-5. Decision Coverage will require that two subdomains be visited, the True (15:(TTTT)) subdomain and one of the False subdomains. Note that the False subdomain does not have to be one of the ones MCDC would visit as (5:(FTFT), 6:(FTTF), 9:(TFFT), and 10:(TFTF)) are all acceptable for Decision Coverage but not MCDC. Statement Coverage in the weakest sense will only require that one of the subdomains be visited, generally the True.

Recall that MCDC requires a “significant” visitation (test) on either side of each of the partition boundaries. This means that the number of subdomains is not the factor that drives our MCDC coverage, the number of partitions is what drives coverage. The next two examples will help to clarify this point.



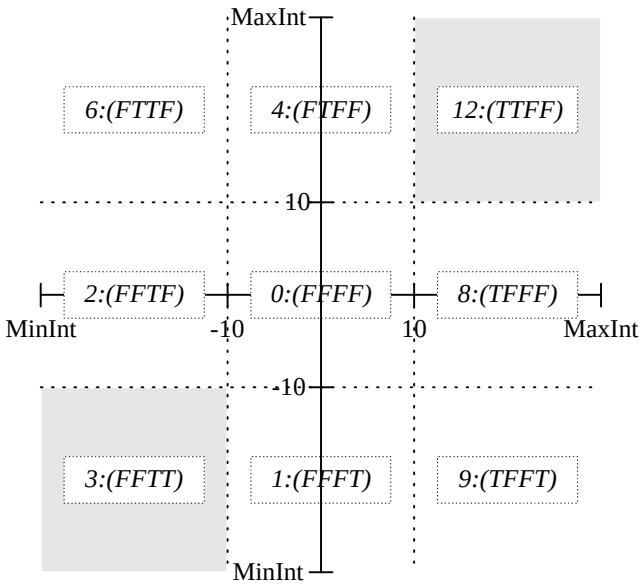
-10 <= X and X >= 10 and -10 <= Y and Y >= 10

FIGURE B-5. DOUBLE VARIABLE SUBDOMAIN PARTITIONING DUE TO FOUR CONDITIONS—ONE TRUE REGION

Figure B-6 shows subdomain partitioning for two variables due to four conditions resulting in two True regions. Just as with figure B-5, there are nine subdomains in figure B-6. These subdomains have been labeled with condition codes that correspond to the four conditions (i.e., ($X \geq 10$, $Y \geq 10$, $X \leq -10$, $Y \leq -10$)). The two True regions have been shaded, while the seven False regions are left unshaded.

Note that all forms of MCDC will require that both True subdomains (3:(FFTT) and 12:(TTFF)) be visited in figure B-6. Which False subdomains are to be visited now depends on which form of MCDC is used.

If the Unique Cause form is being used, then only a single partition boundary may be “crossed” at a time in order to show the underlying condition’s independence. This means that the True subdomain (3:(FFTT)) will need to visit False subdomains (1:(FFFT) and 2:(FFTF)) in order to cross the ($X \leq -10$) and ($Y \leq -10$) partition boundaries respectively. The True subdomain (12:(TTFF)) will need to visit False subdomains (4:(FTFF) and 8:(TFFF)) to cross the ($X \geq 10$) and ($Y \geq 10$) partition boundaries respectively. This results in six tests being needed for this expression. Note that this is larger than the $N + 1$ formula given at the beginning of this appendix. This is because coupling between the conditions did not allow for the full truth table (only nine of sixteen combinations are possible), and the combinations which would have allowed $N + 1$ tests are not present.



(X >= 10 and Y >= 10) or (X <= -10 and Y <= -10)

FIGURE B-6. DOUBLE VARIABLE SUBDOMAIN PARTITIONING DUE TO FOUR CONDITIONS—TWO TRUE REGIONS

If the Masking form is being used, then only a single *significant* partition boundary may be crossed at a time. This means that the True subdomain (3:(FFTT)) may visit False subdomains (6:(FTTF) and 9:(TFFT)) in order to cross the ($X \leq -10$) and ($Y \leq -10$) partition boundaries respectively. Note that the (3:(FFTT) and 6:(FTTF)) pair crosses two partitions (3:(FFTT) - 2:(FFTF)) and (2:(FFTF) - 6:(FTTF)). Masking regards only the (3:(FFTT) - 2:(FFTF)) crossing as significant as there is no change in result due to the (2:(FFTF) - 6:(FTTF)) crossing. In similar fashion, the True subdomain (12:(TTFF)) may visit False subdomains (6:(FTTF) and 9:(TFFT)) to cross the ($X \geq 10$) and ($Y \geq 10$) partition boundaries respectively. This results in four tests (3:(FFTT), 6:(FTTF), 9:(TFFT) and 12:(TTFF)) being needed for this expression.

Masking MCDC will not allow the (3:(FFTT) and 0:(FFFF)) pair to be used even though it too crosses two partitions. This is because both partitions are crossed simultaneously, and neither one is uniquely responsible for the change in the result. In similar fashion, the (3:(FFTT) and 8:(TFFF)) pair is also not allowed because it again simultaneously crosses the same two partitions as the (3:(FFTT) and 0:(FFFF)) pair did.

Decision Coverage will require that two subdomains be visited, one of the True subdomains and one of the False subdomains. Note that the False subdomain does not have to be one of the ones MCDC would visit (e.g., it could use 0:(FFFF)). Also note that the (True, False) subdomain pair does not have to be one of the ones that MCDC would visit. For example, the pair (3:(FFTT) and 8:(TFFF) is unacceptable for MCDC but is acceptable for Decision Coverage while the pair (12:(TTFF) and 8:(TFFF)) is acceptable for both MCDC and Decision Coverage. Statement Coverage in the weakest sense will only require that one of the subdomains be visited, generally one of the Trues.

Figure B-7 shows subdomain partitioning for two variables due to four conditions resulting in four True regions. Just as with figures B-5 and B-6, there are nine subdomains in figure B-7. These subdomains have been labeled with condition codes that correspond to the four conditions (i.e., ($X \leq -10$, $X \geq 10$, $Y \leq -10$, $Y \geq 10$)). The four True regions have been shaded, while the five False regions are left unshaded.

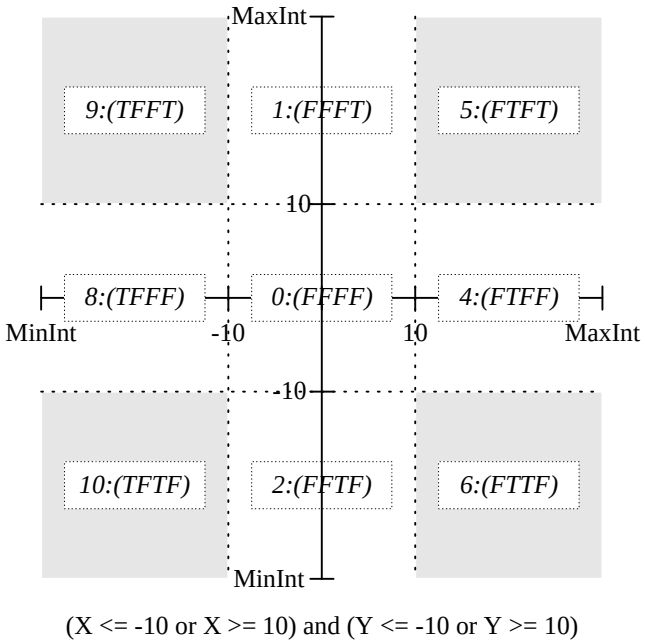


FIGURE B-7. DOUBLE VARIABLE SUBDOMAIN PARTITIONING DUE TO FOUR CONDITIONS—FOUR TRUE REGIONS

Recall that MCDC needs a test on either side of the four partition boundaries, just as in figures B-5 and B-6. For the ($X \leq -10$) condition, either the (1:(FFFT) and 9:(TFFT)) or (2:(FFTF) and 10:(TFTF)) pairs can be used to show its independence. For the ($X \geq 10$) condition, either the (1:(FFFT) and 5:(FTFT)) or (2:(FFTF) and 6:(FTTF)) pairs can be used. The least number of visits (tests) to show both conditions independence is obtained if either the upper three subdomains (1:(FFFT), and 5:(FTFT), and 9:(TFFT)) or the lower three subdomains (2:(FFTF), 6:(FTTF), and 10:(TFTF)) are used. Similarly, the ($Y \leq -10$) and ($Y \geq 10$) conditions use either the left three subdomains (8:(TFFF), 9:(TFFT), and 10:(TFTF)) or the right three subdomains (4:(FTFF), 5:(FTFT), and 6:(FTTF)). Combining one of the *X* subsets (horizontals) with one of the *Y* subsets (verticals) results in five tests being needed for this expression for MCDC.

Decision Coverage will require that two subdomains be visited, one of the True subdomains and one of the False subdomains. Just as with the expression in figure B-6, neither the False subdomain nor the (True, False) subdomain pair has to be one of the ones that MCDC would visit. Statement Coverage in the weakest sense will only require that one of the subdomains be visited, generally one of the Trues.

B.2 Conclusions

All of the examples to this point have used Integers and relational operators on those Integers. This analysis can be extended to Booleans by using *n-cubes* (see section 7). Also, all of the examples have used at most two variables and four conditions. This can be extended to any number of variables and conditions, but the graphical presentation starts to become increasingly complex. The examples contained in this appendix are sufficient to support the points we wish to make.

Notice that in all cases, MCDC requires that each side of a condition's partition boundary be visited. This will (weakly) verify each of the partitions imposed by the conditions. MCDC does not however require the visitation of all the subdomains (e.g., 0:(FF) in figure B-4). This is why it was said earlier that MCDC is a weak measure of equivalence class coverage. MCDC also does not require that values on the boundaries and just on either side of the boundaries be used. It only requires that one value on either side of the boundaries be used. This is why it was said earlier that MCDC is a weak measure of boundary value coverage. Section 4 proposes some extensions to MCDC that would make it a stronger measure of equivalence class and boundary value coverage (but still incomplete).

To generalize the above, every condition present within a logic expression partitions the space for the input variables into two regions (subdomains). MCDC will require that each side of this partitioning plane be visited by the verification process. It will not, however, require that every subdomain formed by all of the partitions be visited. This would be equivalent to multiple condition coverage [1], which is essentially exhaustive testing in the logic domain. This would assure that no logic errors are present, but it comes at a prohibitively high cost. Multiple condition coverage requires 2^N tests or the analysis equivalent for an expression with N uncoupled conditions. This rapidly becomes impractical, as well as infeasible for larger expressions (e.g., the 76-condition expression in appendix C). What MCDC is trying to do is find a more cost-effective approach for logic verification.

MCDC does not allow just any visitation across the partition boundary. It requires the visitation be such that on one side of the partition a different result is returned for the entire logic expression than is returned for the visitation on the other side. Going no further than this however, would be doing the equivalent of Decision Condition Coverage [1]. The problem with this is that you do not know which boundary crossing was responsible for the change in the logic outcome (e.g., between 0:(FF) and 3:(TT) in the expression *A and B*, two boundaries are crossed simultaneously). Therefore, MCDC requires that there be only a single "significant" boundary crossing (e.g., between 1:(FT) and 3:(TT) in the expression *A and B*, only a single boundary is crossed). This requires that the condition have an (independent) effect on the expression. The differences between the different forms of MCDC concern the definition for what forms a significant boundary crossing (independence).

B.3. References

1. Myers, G.J., "The Art of Software Testing," John Wiley & Sons, New York, 1979.

APPENDIX C—AIRBORNE SOFTWARE LOGIC PROFILE

This appendix contains all the logic expressions (20,256) extracted from the airborne software (Ada source code) of five different Line Replaceable Units (LRUs, also known as black boxes). These five LRUs belong to five different airborne systems across two different airplane models (two from one model, three from the other).

Table C-1 contains the summary codes used in the profiling of the logic expressions to classify the objects appearing within the expressions. In those expressions where more than one name of the same type was used (e.g., two Boolean variables (*A,B*) in the expression *A and B*), the original name was abstracted to a summary code without a numeric suffix, while succeeding names were differentiated with the number 2 and upwards (e.g., *Bv and Bv2*). When a name appeared multiple times in an expression, the same summary code was used for all occurrences (e.g., *(Bv and Bv2) or (Bv and Bv3) or (Bv2 and Bv3)*).

TABLE C-1. EXPRESSION PROFILING OBJECT SUMMARY CODES

Code	Meaning
?c	A generic constant object of a user-defined type. All of the generic parameter types profiled in this study were private.
?v	A generic variable object of a user-defined type.
Ac	A constant object of a user-defined access type (i.e., a pointer).
Av	A variable object of a user-defined access type (i.e., a pointer).
Bac	A constant object of a user-defined array type with Boolean components.
Bav	A variable object of a user-defined array type with Boolean components.
Bv	A variable object of Boolean type.
Cac	A constant object of a user-defined array type with components of the predefined character enumerated type. The user-defined array type with character components was not a subtype of the predefined String type.
Cal	A literal of a user-defined array type with components of the predefined character enumerated type. All of these were aggregates. The user-defined array type with character components was not a subtype of the predefined String type.
Cav	A variable object of a user-defined array type with components of the predefined character enumerated type. The user-defined array type with character components was not a subtype of the predefined String type.
Cc	A constant object of the predefined character enumerated type.
Cl	A literal of the predefined character enumerated type.
Cv	A variable object of the predefined character enumerated type.
Eac	A constant object of a user-defined array type with components of a user-defined enumerated type.
Eav	A variable object of a user-defined array type with components of a user-defined enumerated type.
Ec	A constant object of a user-defined enumerated type.
El	A literal of a user-defined enumerated type.

TABLE C-1. EXPRESSION PROFILING OBJECT SUMMARY CODES (Continued)

Code	Meaning
Et	A user-defined enumerated type. These were mostly used in membership tests and for attributes.
Ev	A variable object of a user-defined enumerated type.
False	Predefined literal of the predefined Boolean type.
Flc	A constant object of a user-defined floating point type.
Flv	A variable object of a user-defined floating point type.
Fxc	A constant object of a user-defined fixed point type.
Fxv	A variable object of a user-defined fixed point type.
Iav	A variable object of a user-defined array type with components of an integer type. The integer type could be one of the predefined integer types (e.g., Short_Integer, Integer, Long_Integer), subtypes (e.g., Natural, Positive), or a user-defined integer type or subtype of a user-defined type.
Ic	A constant object of either a predefined or user-defined integer type.
It	Either a predefined or a user-defined integer type. These were mostly used in membership tests and for attributes.
Iv	A variable object of either a predefined or user-defined integer type.
null	Universal predefined access literal, applicable to all user-defined access types.
Rc	A constant object of a user-defined record type.
Rl	A literal of a user-defined record type. All of these were aggregates.
Rv	A variable object of a user-defined record type.
Sac	A constant object of a user-defined array type with components of the predefined String type.
Sav	A variable object of a user-defined array type with components of the predefined String type.
Sc	A constant object of the predefined string type.
Sl	A literal of the predefined string type. All of these are in essence aggregates.
Sv	A variable object of the predefined string type.
True	Predefined literal of the predefined Boolean type.
uil	A universal integer literal (e.g., *0*).
uinn	A universal integer named number (e.g., *No_Deflection : constant := 0*).
url	A universal real literal (e.g., *0.0*).
urnn	A universal real named number (e.g., *PI : constant := 3.14159*).

Table C-2 contains the summary codes used in the profiling of the logic expressions to identify what kind of a statement the logic expression resided in. Examples are given with the relevant part of the statement italicized.

The expressions are grouped into subsections by the number of unique conditions within the expression. Within the groupings, the expressions are given in alphabetic order. The expression data is formatted into three pieces, each piece within its own column.

TABLE C-2. EXPRESSION PROFILING STATEMENT SUMMARY CODES

Code	Statement Type	Example
()	Boolean index dereference.	ev := eav(*bv*);
'b	attribute expression.	iv := Boolean'pos(*bv and bv2*);
:=	RHS of a Boolean assignment statement.	bv := *bv2 or bv3*;
:=>	member of an aggregate expression.	rv := (brc => (*bv and bv2*), erc => et'first);
::=	RHS of a Boolean initialization expression.	bv : Boolean := *bv2 or bv3*;
=>	actual parameter expression.	flf(flv, flv2, *bv*)
exit	The condition part of exit-statement.	exit when *av = null*;
if	The condition part of either an if-clause or elsif-clause part of an if-statement.	if *bv* then elsif *bv and bv2* then
return	The Boolean expression part of a return-statement for a Boolean function.	return *iv > ic*;
while	The condition part of while loop-statement.	while *av /= null and then av.all >= 0* loop

The first piece of data in the first column is the (profiled) text of the expression itself. This text uses the summary codes which appear in table C-1. The use of parenthesis in the text is what was used in the original source code. If there are constraints imposed on any of the objects in the expression which are significant for a coverage analysis, they occur within curly braces ({}) immediately following the expression text in the first column.

The second piece of data in the second column identifies the kinds of locations within the Ada source code that the expression appeared within. The locations are identified with the summary codes that appear in table C-2. If the expression appeared within multiple kinds of locations, each location is listed on its own line.

The final piece of data in the third column identifies how many times the expression appeared within the Ada location.

For example, the following entry means that the expression abstracted by *(Av = Av2)* appeared five times within a Boolean assignment statement, and sixteen times within an if statement.

Expression	Statement	Occurs
(Av = Av2)	:=	5
	if	16

The following entry means that the expression abstracted by *((Flv < url) and (Flv >= url2))* appeared once within an if statement. In addition, there is a constraint on *url* and *url2* such that *url>url2*.

Expression	Statement	Occurs
((Flv < url) and (Flv >= url2)) {url>url2}	if	1

C.1 Expressions With One Condition

C.1.1 One-Condition Expressions With the Response Profile 1:FT

Expression	Statement	Occurs
(?v = ?v2)	if	4
(Av /= Av2)	if	2
(Av /= null)	if	16
(Av = Av2)	:=	5
	if	16
(Bv = true)	if	44
(Bv)	=>	71
	if	113
(Cav = Cal)	if	5
(Cv = Cl)	if	4
(Ec < Ev)	:=	8
(Ec > Ev)	:=	8
(Ev /= El)	=>	1
	if	71
(Ev /= Ev2)	=>	2
	exit	11
	if	5
(Ev = Ec)	if	6
(Ev = El)	:=	5
	exit	1
	if	341
(Ev = Ev2)	if	4
(Ev in El..El2)	if	5
(Ev in Et)	if	10
(Flv < Flc)	if	2
(Flv > Flc)	if	1
(Fxv < Fxc)	:=	2
	=>	4
	if	47
(Fxv < Fxv2)	if	23
(Fxv < url)	:=	2
(Fxv < urnn)	:=	1
(Fxv <= Fxc)	:=	2
	if	56
(Fxv <= Fxv2)	if	2
(Fxv > Fxc)	:=	2
	=>	8
(Fxv > Fxv2)	:=	2
	if	24

Expression	Statement	Occurs
(Fxv >= Fxc)	:=	2
	if	16
(Fxv >= Fxv2)	if	3
(Ic = Iv)	:=	152
(Iv /= Ic)	if	33
(Iv /= Iv2)	=>	4
	exit	5
	if	13
(Iv /= uil)	if	95
(Iv < Ic)	:=	4
	if	15
(Iv < Iv2)	if	12
(Iv < uil)	if	32
(Iv < uinn)	if	2
(Iv <= Et()'pos)	if	2
(Iv <= Ic)	if	9
(Iv <= It'last)	if	2
(Iv <= uil)	exit	1
	if	2
(Iv = Ic)	:=	28
	exit	8
	if	63
(Iv = It'first)	if	4
(Iv = It'last)	exit	2
(Iv = Iv2)	:=	11
	exit	3
	if	33
(Iv = uil)	:=	11
	exit	2
	if	77
(Iv = uinn)	if	5
(Iv > Ic)	if	19
(Iv > Iv2)	if	3
(Iv > uil)	if	40
(Iv > uinn)	if	2
(Iv >= Ic)	:=	4
	if	24
(Iv >= Iv2)	:=	4
	if	24
(Iv >= uil)	if	5
(Iv in Ic..Ic2)	if	15
(Rv /= Rv2)	if	2
(Rv = Rc)	if	6

Expression	Statement	Occurs
(Rv > Rv2)	if	1
(Sv /= Sl)	if	2
(uinn = Iv)	:=	2
?c = ?v	if	1
?v /= ?c	while	1
?v = ?c	if	2
?v = ?v2	if	4
Av /= Ac	if	21
Av /= null	if	49
	while	14
Av = Ac	if	8
Av = Av2	if	29
Av = null	if	40
Bav = Bac	if	3
Bv	()	32
	::=	19
	:=	1806
	:=>	29
	=>	5011
	b’	70
	exit	4
	if	2723
	return	9
	while	39
Bv /= false	:=	1
Bv = true	if	227
Cav = Cac	if	1
Cav = Cav2	if	1
Cv = Cc	if	3
Cv = Cl	if	8
Eav = Eac	if	3
Ec /= Ev	:=	6
Ec < Ev	:=	2
Ec = Ev	:=	6
Ec > Ev	:=	2
	if	3
El = Ev	if	20
Ev /= Ec	if	2
Ev /= El	exit	1
	if	54
	while	2

Expression	Statement	Occurs
Ev /= Et'last	if	9
Ev /= Ev2	:=	2
	exit	14
	if	8
Ev < Ev2	if	6
Ev = El	:=	7
	=>	7
	exit	2
	if	769
	while	2
Ev = Et'first	if	3
Ev = Et'last	if	3
Ev = Ev2	:=	27
	if	8
Ev > Ev2	=>	2
	if	6
Ev >= El	if	2
Ev >= Et'last	if	1
Ev in El..El2	if	8
Ev in Et	if	22
Ev not in Et	if	4
Flc < Flv	:=	2
Flv /= Flv2	if	1
Flv /= url	if	1
Flv < Flc	if	14
Flv < Flv2	:=	2
	if	6
Flv < url	:=	2
	if	8
Flv <= Flc	if	12
Flv <= Flv2	if	7
Flv <= url	if	2
Flv = Flv2	if	2
Flv = url	if	1
Flv > Flc	if	3
Flv > Flv2	if	4
Flv > url	if	4
Flv >= Flc	if	16
	while	12
Flv >= Flv2	if	6
	while	5
Fxc >= Fxv	if	3
Fxv /= url	if	1

Expression	Statement	Occurs
Fxv < Fxc	:=	9
	=>	5
	if	37
Fxv < Fxv2	:=	3
	=>	3
	if	60
Fxv < url	if	53
Fxv < urnn	if	15
Fxv <= Fxc	if	12
Fxv <= Fxv2	=>	1
	if	116
Fxv <= url	if	13
Fxv = Fxv2	if	4
Fxv = url	if	1
Fxv > Fxc	:=	39
	=>	2
	if	35
Fxv > Fxv2	:=	7
	=>	3
	if	57
Fxv > url	:=	2
	=>	1
	if	10
Fxv > urnn	:=	3
	if	15
Fxv >= Fxc	=>	4
	if	22
Fxv >= Fxv2	:=	2
	if	26
Fxv >= url	if	20
Ic /= Iv	if	2
Ic = Iv	:=	210
	if	15
Ic > Iv	if	4
Iv /= Ic	if	35
Iv /= Iv2	:=	2
	=>	2
	if	46
Iv /= uil	if	90
Iv /= uinn	if	3
Iv < Ic	:=	2
	if	18
Iv < It’last	if	1

Expression	Statement	Occurs
Iv < Iv2	:=	6
	if	47
	while	13
Iv < St'last	if	3
Iv < uil	:=	1
	if	54
Iv < uinn	:=	3
	if	32
Iv <= Ic	if	25
Iv <= It'first	if	1
Iv <= It'last	if	6
Iv <= Iv2	:=	2
	if	2
Iv <= St'last	if	3
Iv <= uil	if	12
Iv <= uinn	if	12
Iv = Iav'last	exit	12
Iv = Ic	if	128
Iv = It'first	if	2
Iv = It'last	exit	4
	if	12
Iv = Iv2	=>	2
	if	156
Iv = uil	exit	42
	if	246
Iv = uinn	if	24
Iv > Ic	:=	2
	if	30
	while	2
Iv > It'first	if	1
Iv > Iv2	:=	6
	if	53
	while	2
Iv > uil	if	112
Iv > uinn	if	5
Iv >= Ic	:=	4
	if	43
Iv >= It'last	if	3
Iv >= Iv2	:=	2
	if	34
Iv >= uil	exit	6
	if	17
Iv >= uinn	if	31

Expression	Statement	Occurs
Iv in It	if	4
Iv in uil..Ic	if	17
Rv /= Rc	if	6
Rv /= Rv2	if	5
Rv = Rv2	if	1
Rv >= Rc	if	1
Sav /= Sac	:=	3
Sv /= Sl	if	3
Sv = Sl	if	6
Sv'length < Iv	if	1
Sv'length <= Iv	if	2
uinn < Iv	if	2
uinn = Iv	if	3

C.1.2 One-Condition Expressions With the Response Profile 2:TF

Expression	Statement	Occurs
(Bv = false)	=>	19
	if	62
(not (Bv = true))	if	1
(not (Bv))	while	2
(not (Ev = El))	if	1
(not Bv)	:=	1
	=>	4
	if	36
	while	1
Bv /= true	:=	1
	if	8
Bv = false	exit	7
	if	222
not (Bv)	:=	18
	if	131
not (Ev = El)	:=	4
	if	3
not (Iv > uil)	if	6
not (Iv in Ic..Ic2)	if	3
not Bv	::=	16
	:=	141
	exit	6
	if	554
	while	2

C.2 Expressions With Two Conditions

C.2.1 Two-Condition Expressions With the Response Profile 1:FFFT

C.2.1.1 Response Profile 1:FFFT

Expression	Statement	Occurs
((Bv = True) and (Ev = El))	if	1
((Ev /= Ev2) and (Ev = Ev3))	=>	2
(Av /= null) and (Av /= Av2)	while	10
(Bv = True) and (Bv2 = True)	if	4
(Bv = True) and (Iv = Ic)	if	1
(Bv = True) and (Iv = uil)	if	8
(Bv = True) and (Iv >= uil)	if	1
(Bv and (Bv2 = True))	if	1
(Bv and (Ev = El))	if	1
(Bv and (Ev > El))	:=	2
(Bv and (Fxv < url))	=>	1
(Bv and (Fxv >= url))	=>	1
(Bv and (Iv /= Iv2))	while	8
(Bv and (Iv >= Ic))	:=	2
(Bv and Bv2)	:=	22
	=>	49
	exit	1
	if	22
(Bv and Iv < uil)	if	1
(Bv) and (Ev = El)	if	1
(Cav /= Cac) and (Cv /= Cc)	if	2
(Cv = Cl) and (Cv2 = Cl)	if	1
(Ev /= Ev2) and (Iv = uil)	if	14
(Ev < Ev2) and Bv	:=	2
(Ev = Ec) and (Bv = True)	if	3
(Ev = Ec) and (Iv /= uil)	if	3
(Ev = El and Ev2 = El)	if	1
(Ev = El and Ev2 = El2)	if	3
(Ev = El and Iv = uil)	if	3
(Ev = El) and (Bv = True)	if	1
(Ev = El) and (Bv)	if	2
(Ev = El) and (Ev2 /= El2)	if	1
(Ev = El) and (Ev2 = El2)	:=	1
	if	8
(Ev = El) and (Iv /= It'last)	if	2
(Ev = El) and (Iv = Iv2)	if	2
(Ev = El) and (Iv = uil)	if	2
(Ev = El) and (Iv > uil)	if	3

Expression	Statement	Occurs
(Ev = El) and (Iv >= uil)	if	2
(Ev = El) and Bv	:=	2
	if	3
(Ev = El) and Fxv > Fxc	:=	1
(Ev in El..El2) and (Ic = Iv)	if	1
(Ev not in El..El2) and Iv > uil	if	2
(Flv <= Flv2) and (Flv2 <= Flv3)	if	6
(Fxv /= url) and (Fxv2 <= Fxv)	if	6
(Fxv <= Fxc) and Bv	:=	2
(Fxv > urnn) and Bv	:=	1
(Fxv >= Fxv2) and (Fxv2 >= Fxv3)	:=	6
(Iv /= Ic) and (Bv = True)	if	2
(Iv /= Iv2) and (Ev = El)	if	1
(Iv /= uil) and (Bv = True)	if	1
(Iv /= uil) and (Iv < Ic)	if	3
(Iv /= uil) and (Iv2 /= uil)	:=	1
(Iv < Ic and Ev /= El)	if	2
(Iv < Iv2 and Ev /= El)	if	3
(Iv <= Iv2) and (Iv /= uil)	if	1
(Iv <= uil) and (Iv2 = Ic)	if	1
(Iv <= uinn) and (Iv > uil)	if	2
(Iv = Ic) and (Bv = True)	if	2
(Iv = Ic) and (Iv >= Iv2)	if	2
(Iv = Ic) and (Iv2 = Ic2)	if	4
(Iv = Ic) and (Iv2 = uil)	if	1
(Iv = Ic) and Bv	:=	2
(Iv = It’last) and (Iv2 = It2’last)	if	2
(Iv = Iv2) and (Iv2 = Iv3)	if	1
(Iv = Iv2) and (Iv3 = Iv4)	if	1
(Iv > uil) and (Ev = El)	if	3
(Iv > uil) and (Iv < Ic)	if	3
(Iv >= Iv2) and (Iv <= Iv3)	if	3
(Iv >= Iv2) and (Iv3 < Iv4)	if	1
(Iv >= uil) and (Iv2 = uil2)	if	1
(Sv = Sv2) and (Sv = Sv3)	if	6
Bv = True and Bv2 = True	if	3
Bv = True and Bv2	if	1
Bv = True and Iv <= Ic	if	1
Bv = True and Iv >= Iv2	if	1
Bv and (Bv2)	:=	2
Bv and (Ev /= El)	if	5
Bv and (Ev = El)	if	4
Bv and (Iv /= uil)	:=	5

Expression	Statement	Occurs
Bv and (Iv = uil)	if	2
Bv and (Iv > uil)	if	9
Bv and (Iv >= Ic)	:=	2
Bv and (Iv >= uil)	if	9
Bv and Bv2	:=	75
	if	55
Bv and Ev /= Ec	if	1
Bv and Ev = El	if	17
Bv and Flv >= Flv2	if	1
Bv and Iv /= uil	if	1
Cv = Cl and Cv2 = Cl	if	2
Ev /= El and Iv /= uil	if	2
Ev = El and Bv	if	4
Ev = El and Ev2 = El	exit	1
	if	1
Ev = El and Ev2 = El2	if	3
Ev = Ev2 and (Ev3 = Ev4)	if	1
Ev not in El .. El2 and Iv > uil	if	2
Flv >= Flc and Bv	if	1
Fxv < Fxc and Fxv2 > Fxc	if	1
Fxv = url and Fxv2 = url	if	3
Iv = Iv2 and Iv2 /= uil	if	2
Iv = uil and (Ev = El)	if	1
Iv = uil and Iv2 = uil	exit	4
Iv = uil and Iv2 = uinn	if	2
Iv > Ic and Bv	if	2
Iv >= It'first and Iv <= Iv2	if	1
Iv >= Iv2 and Bv	if	1

C.2.1.2 Response Profile 1:F-FT

Expression	Statement	Occurs
Ev /= El and then Ev = El2	if	6

C.2.1.3 Response Profile 1:FfFT

Expression	Statement	Occurs
((Ev = El) and then (Iv = Ic))	if	6
(Av /= null) and then (Iv <= Iv2)	while	5
(Av /= null) and then (Iv >= Iv2)	while	5
(Bv) and then (Flv > Flv2)	if	1
(Flv > url and then Flv2 > Flv3)	if	1
(Iv < uinn) and then (Iv2 >= Iv3)	if	1
(Iv = uil) and then (Bv)	if	1
(Iv >= uinn) and then (Ev < Et'last)	if	1
(Iv >= uinn) and then (Iv2 < Iv3)	if	2
(Iv >= uinn) and then (Iv2 > Iv3)	if	1
Av /= null and then Iv = Iv2	if	1
Bv and then (Iv = uil)	if	2
Bv and then Bv2	if	3
Bv and then Ev = El	if	5
Bv and then Flv > Flv2	if	1
Bv and then Fxv > Fxc	:=	4
Bv and then Iv > uil	if	3
Cv = Cl and then Cv2 = Cl	if	4
Ev = Et'last and then Ev2 /= Et2'last	if	4
Ev = Et'last and then Ev2 = Et2'last	if	2
Flv = url and then Flv2 > url	if	1
Flv >= Flv2 and then Flv3 >= Flv4	if	2
Iv /= Ic and then Ev = El	if	2
Iv = Ic and then Sv = Sc	if	1
Iv = Iv2 and then Iv = Ic	if	1
Iv >= Ic and then Bv	if	2

C.2.1.4 Response Profile 1:-FFT

Expression	Statement	Occurs
((Ev /= El) and (Ev /= El2))	if	9
((Flv < url) and (Flv >= url2)) {url>url2}	if	1
((Fxv > url) and (Fxv < url2)) {url<url2}	=>	1
((Iv /= uil) and (Iv /= uil2))	if	3
((Iv <= Ic) and (Iv > Ic2)) {ic>ic2}	if	6
((Iv <= Ic) and (Iv >= Ic2)) {ic>ic2}	if	5
(Ev /= El) and (Ev /= El2)	if	7
(Ev <= El) and (Ev >= El2) {el>el2}	if	3
(Ev >= El) and (Ev <= El2) {el<el2}	if	1
(Flv < Flc) and (Flv > Flc2) {flc>flc2}	if	2
(Flv < url) and (Flv > url2) {url>url2}	if	1
(Flv > url) and (Flv < url2) {url<url2}	if	2
(Flv >= Flc) and (Flv <= Flc2) {flc<flc2}	if	3
(Ic >= Iv and Iv >= Ic2) {ic>ic2}	=>	36
(Iv /= Ic and Iv /= Ic2)	if	3
(Iv /= uil) and (Iv /= uil2)	if	3
(Iv <= Ic) and (Iv >= Ic2) {ic>ic2}	if	2
(Iv <= uil) and (Iv >= uil2) {uil>uil2}	if	1
(Iv > uil) and (Iv < uil2) {uil<uil2}	if	1
(Iv >= uil) and (Iv < uil2) {uil<uil2}	if	2
?v >= ?c and ?v <= ?c2 {?c<?c2}	if	1
Ev /= El and Ev /= El2	if	6
Ev >= El and Ev <= El2 {el<el2}	if	4
Ev >= Et'first and Ev <= Et'last	if	2
Ic >= Iv and Iv >= Ic2 {ic>ic2}	:=	4
Iv < uil and Iv /= uil2 {uil>uil2}	if	1
Iv >= Ic and Iv <= Ic2 {ic<ic2}	if	5
Iv >= uil and Iv <= uil2 {uil<uil2}	if	1
Iv in It and Iv /= It'last	if	1

C.2.1.5 Response Profile 1:-fFT

Expression	Statement	Occurs
Ev /= El and then Ev /= El2	if	6

C.2.2 Two-Condition Expressions With the Response Profile 2:FFTF

C.2.2.1 Response Profile 2:FFTF

Expression	Statement	Occurs
((Ec /= Ev) and (not Bv))	=>	2
((Fxv <= Fxv2) and not Bv)	:=	4
((Fxv > Fxc) and (not Bv))	=>	1
((Ic /= Iv) and not (Bv))	while	2
((Iv <= Iv2) and not (Bv))	while	2
(Bv = True and Bv2 = False)	if	8
(Bv and not (Bv2))	=>	2
	if	1
(Bv and not Bv2)	:=	1
	=>	5
	if	13
(Bv) and (not Bv2)	if	1
(Ev /= El) and (not Bv)	if	3
(Ev = El) and (Bv = False)	if	1
(Ev = El) and (not Bv)	if	10
(Ev = El) and not Bv	:=	1
(Fxv < Fxc) and not Bv	:=	1
(Fxv <= Fxc) and not Bv	:=	1
	if	1
(Fxv > Fxc) and (not Bv)	:=	2
(Iv < uil) and (Bv = False)	if	1
(Iv = uil) and (not Bv)	if	4
Bv = True and Bv2 = False	if	1
Bv and (not Bv2)	:=	44
	if	9
Bv and not (Bv2)	:=	4
	if	6
Bv and not Bv2	:=	45
	if	61
Ev /= El and not (Bv)	if	3
Ev < Et'last and Bv = False	while	1
Ev = El and Bv = False	if	2
Ev = El and not Bv	if	2
Flv > Flv2 and not Bv	if	1
Iv = uinn and not Bv	if	1

C.2.2.2 Response Profile 2:FfTF

Expression	Statement	Occurs
Bv and then not Bv2	if	3

C.2.3 Two-Condition Expressions With the Response Profile 4:FTFF

C.2.3.1 Response Profile 4:FTFF

Expression	Statement	Occurs
((Bv = False) and Bv2)	=>	4
((not Bv) and Bv2)	=>	1
	if	3
(Bv /= True) and (Iv /= uil)	while	2
(Bv = False and Bv2)	=>	27
(Bv = False) and (Bv2 = True)	if	2
(Bv = False) and (Iv = Ic)	if	1
(not Bv and (Iv >= uinn))	=>	1
(not Bv and Bv2)	=>	19
	if	6
(not Bv and Fxv < url)	=>	1
(not Bv and Iv >= uinn)	=>	1
(not Bv) and (Bv2)	:=	2
(not Bv) and (Ev = El)	if	6
(not Bv) and (Ev in El..El2)	if	1
(not Bv) and (Flv /= Flv2)	if	1
(not Bv) and (Iv >= uil)	if	4
(not Bv) and Bv2	:=	12
	if	1
Bv = False and Bv2 = True	if	8
Bv = False and Bv2	if	2
not (Bv) and Bv2	:=	2
	if	2
not Bv and (Iv >= Ic)	:=	2
not Bv and (Iv >= uil)	if	5
not Bv and Bv2	:=	16
	if	14
not Bv and Ev = El	if	9

C.2.3.2 Response Profile 4:FTFf

Expression	Statement	Occurs
(not Bv) and then (Flv > Flv2)	if	1
not Bv and then Bv2	if	1

C.2.4 Two-Condition Expressions With the Response Profile 6:FTTF

Expression	Statement	Occurs
(((not Bv) and Bv2) or (Bv and (not Bv2)))	:=	2
(Bv /= Bv2)	=>	4
	if	1
(Bv = not (Bv2))	if	2
(Bv xor Bv2)	=>	1
Bv /= Bv2	:=	1
	if	16
Bv xor Bv2	:=	9
	if	2

C.2.5 Two-Condition Expressions With the Response Profile 7:FTTT

C.2.5.1 Response Profile 7:FTTT

Expression	Statement	Occurs
((Ev = El) or (Bv = True))	exit	1
((Iv = Iv2) or (Bv))	if	2
(?v /= ?v2 or Ev = El)	if	5
(Av = Av2) or (Av = Av3)	if	4
(Bv = True) or (Bv2 = True)	if	4
(Bv = True) or (Ev = El)	if	1
(Bv = True) or (Iv <= uil)	if	1
(Bv = True) or (Iv >= uil)	if	2
(Bv or (Iv /= Iv2))	if	2
(Bv or Bv2)	:=	3
	=>	20
	exit	2
	if	28
(Bv or Fxv >= Fxv2)	=>	1
(Ev <= El) or Bv	if	1
(Ev = El or Ev2 = El)	if	1
(Ev = El or Rv /= Rv2)	if	5
(Ev = El) or (Ev2 = El)	:=	3
	if	15
(Ev = El) or (Ev2 = El2)	if	3
(Ev = El) or (Iv = uil)	if	3
(Ev = El) or Bv	:=	1
(Fxv < urnn) or (Fxv2 < urnn2)	:=	2
(Fxv > Fxc) or (Fxv2 > Fxc2)	:=	2
(Fxv > Fxv2) or (Fxv < Fxv3)	:=	1
(Fxv > urnn) or (Fxv2 < urnn2)	:=	2
(Iv /= Ic) or (Iv2 /= Ic2)	if	8

Expression	Statement	Occurs
(Iv /= Iv2) or Bv	:=	3
(Iv /= uil) or Bv	:=	2
(Iv < Ic) or (Iv2 < Ic)	if	11
(Iv <= Ic) or (Iv2 <= Ic)	if	8
(Iv = Ic) or Bv	if	2
(Iv = Iv2) or (Iv = Iv3)	if	1
(Iv = uil) or (Ev = El)	if	2
(Iv = uil) or (Iv2 > Ic)	if	2
(Iv = uinn) or (Iv2 <= Ic)	if	2
(Iv > Ic) or (Iv2 > Ic)	if	11
(Iv >= Ic) or (Iv2 >= Ic)	if	8
(Iv >= uil) or (Flv > url)	if	1
(Rv /= Rv2 or Ev = El)	if	6
Bv = True or Bv2 = True	if	1
Bv or (Iv /= Ic)	:=	6
Bv or (Iv /= Iv2)	:=	12
Bv or Bv2	:=	415
	if	71
Bv or Ev = El	if	2
Ev /= El or (Ev = El and Iv = Ic)	if	3
Ev = El or Bv	if	5
Ev = El or Ev2 = El2	if	1
Flv >= Flv2 or Bv = True	if	2
Fxv > Fxc or Bv	if	2
Iv /= Iv2 or Iv3 /= Iv4	if	6
Iv > Iv2 or Iv3 > Iv4	if	1
Iv > uil or Iv2 > uil	if	15

C.2.5.2 Response Profile 7:FTT-

Expression	Statement	Occurs
((Ev = El) or (Ev = El2))	if	1
((Iv < uil) or (Iv > uil2)) {uil<uil2}	if	1
((Iv = Ic) or (Iv = Ic2))	if	1
(Ev = El) or (Ev = El2)	if	66
(Ev in El..El2) or (Ev = El3) {el1<el2<el3}	if	1
(Ev in Et) or (Ev = El) {el not in et}	if	1
(Flv > url) or (Flv < url2) {url>url2}	if	3
(Fxv > Fxc) or (Fxv < Fxc2) {fxc>fxc2}	if	3
(Fxv > url) or (Fxv < url2) {url>url2}	:=	1
(Fxv > url) or (Fxv <= url2) {url>url2}	:=	1
(Iv < uil) or (Iv > uil2) {uil<uil2}	if	4
(Iv = Ic) or (Iv = Ic2)	if	12
(Iv = uil) or (Iv = uil2)	if	1
(Iv in uil..uil2) or (Iv in uil3..uil4) {uil<uil2<uil3<uil4}	if	3
?v = ?c or ?v = ?c2	if	1
Ev = El or Ev = El2	if	21
Ev = El or Ev >= El2 {el<el2}	if	1
Flv < Flc or Flv > Flc2 {flc<flc2}	if	1
Fxv > url or Fxv < url2 {url>url2}	:=	3
Iv < Ic or Iv > Ic2 {ic<ic2}	if	2

C.2.5.3 Response Profile 7:FTTt

Expression	Statement	Occurs
((Av = null) or else (Iv /= uil))	exit	5
(Av = null) or else (Iv < Iv2)	exit	4
Bv or else Rv >= Rc	if	1
Iv < uinn or else Iv2 < uinn	if	1

C.2.6 Two-Condition Expressions With the Response Profile 8:TFFF

C.2.6.1 Response Profile 8:TFFF

Expression	Statement	Occurs
(Bv = Bv2) and (Bv2 = False)	if	1
(not (Bv or Bv2))	=>	1
(not (Bv) and not (Bv2))	:=	3
(not (Bv)) and (not (Bv2))	:=	2
(not Bv and not Bv2)	=>	1
	if	4
(not Bv) and (not Bv2)	:=	3
	if	2
Bv = False and Bv2 = False	if	6
not (Bv or Bv2)	:=	26
	=>	1
	if	6
not (Bv) and not (Bv2)	:=	2
not Bv and not (Ev = El)	:=	1
not Bv and not Bv2	:=	9
	if	8

C.2.6.2 Response Profile 8:TFF-

Expression	Statement	Occurs
not (Ev = El or Ev = El2)	if	1

C.2.7 Two-Condition Expressions With the Response Profile 9:TFFT

Expression	Statement	Occurs
(Bv = Bv2)	=>	2
	if	5
Bv = Bv2	:=	1
	if	78
not (Bv xor Bv2)	:=	2

C.2.8 Two-Condition Expressions With the Response Profile 11:TFTT

C.2.8.1 Response Profile 11:TFTT

Expression	Statement	Occurs
((Fxv >= Fxc) or not Bv)	if	2
(Bv or (not (Bv2)))	if	2
(Bv or not Bv2)	:=	1
(Fxv < Fxc or (not Bv))	:=	2
(Fxv >= Fxc) or not (Bv)	if	2
Bv or not Bv2	:=	8
	if	3
Flv < Flv2 or not Bv	if	1

C.2.8.2 Response Profile 11:TFTt

Expression	Statement	Occurs
(Iv /= Iv2) or else not Bv	exit	3

C.2.9 Two-Condition Expressions With the Response Profile 13:TTFT

Expression	Statement	Occurs
((not Bv) or Bv2)	if	2
(Bv = Bv2) or (Bv2 = True)	if	1
(Bv = False) or (Iv /= Ic)	if	1
(Bv = False) or (Iv = Ic)	if	12
(not (Bv) or Bv2)	:=	6
(not Bv) or Bv2	:=	3
not (Bv and not Bv2)	if	1
not (Bv) or Bv2	:=	2
	if	3
not Bv or (Iv > uil)	if	8
not Bv or Bv2	:=	3
	if	1

C.2.10 Two-Condition Expressions With the Response Profile 14:TTTF

C.2.10.1 Response Profile 14:TTTF

Expression	Statement	Occurs
(Bv = False) or (Bv2 = False)	if	4
not ((Iv >= Iv2) and (Iv <= Iv3))	if	3
not (Bv and Bv2)	:=	3
	if	8
not (Ev = El and Bv)	if	2
not Bv or not Bv2	:=	7
	if	1

C.2.10.2 Response Profile 14:TtTF

Expression	Statement	Occurs
not Bv or else not Bv2	if	1

C.3 Expressions With Three Conditions

C.3.1 Three-Condition Expressions With the Response Profile 1:FFFFFFFT

C.3.1.1 Response Profile 1:FFFFFFFT

Expression	Statement	Occurs
(Bv = True) and (Bv2 = True) and (Bv3 = True)	if	2
(Bv = True) and (Ev = El) and (Bv2 = True)	if	1
(Bv and Bv2 and Bv3)	:=	1
	=>	3
	if	4
(Bv) and (Bv2) and (Bv3)	:=	2
(Ev /= El) and (Bv = True) and (Iv = Ic)	if	2
(Ev /= Ev2) and (Ev3 = Ev4) and (Ev3 = Ev5)	if	1
(Ev = Ec) and (Ev2 = Ec) and (Ev3 = Ec)	if	8
(Ev = El) and (Bv) and (Ev2 = El2)	if	2
(Ev = El) and (Ev2 = El) and (Bv = True)	if	1
(Ev = El) and (Ev2 = El) and (Ev3 = El)	if	1
(Ev = El) and (Iv = Iv2) and (Iv3 = Iv4)	if	3
(Ev = Ev2) and (Ev = Ev3) and (Ev = Ev4)	if	1
(Iv = Ic) and (Iv2 = Ic) and (Iv3 = Ic)	if	3
(Iv = uil) and (Ev = El) and (Iv2 = uil)	if	4
(Iv > uil) and (Iv2 = uil) and (Iv3 /= uil)	if	3
(Sv = Sv2) and (Sv = Sv3) and (Sv = Sv4)	if	2
Bv and (Ev = El) and (Ev2 = El2)	:=	1
Bv and (Ev = El) and Bv2	:=	1

Expression	Statement	Occurs
Bv and (Ev = Ev2) and (Ev3 = Ev4)	:=	2
Bv and (Fxv > Fxc) and Bv2	:=	2
Bv and Bv2 and (Ev /= El)	if	3
Bv and Bv2 and (Ev = El)	if	16
Bv and Bv2 and (Fxv > Fxc)	:=	2
Bv and Bv2 and Bv3	:=	21
	if	6
Bv and Iv /= uil and Ev = El	if	2
Ev = El and Iv = Iv2 and Iv3 = Iv4	if	3
Iv = uil and Iv2 /= uil and Iv3 = uil	if	2

C.3.1.2 Response Profile 1:FfffFfFT

Expression	Statement	Occurs
Cv = Cl and then Cv2 = Cl and then Cv3 = Cl	if	3

C.3.1.3 Response Profile 1:-FFF-FFT

Expression	Statement	Occurs
(Ev = Ec) and (Iv /= uil) and (Iv /= uil2)	if	6
(Ev = El) and ((Ev2 /= El2) and (Ev2 /= El3))	if	2

C.3.1.4 Response Profile 1:--FFFFFT

Expression	Statement	Occurs
((Ev /= El) and (Ev /= El2)) and (Bv)	if	4
(Ev /= El) and (Ev /= El2) and (Bv)	if	4

C.3.1.5 Response Profile 1:---F-FFT

Expression	Statement	Occurs
(Ev /= El) and (Ev /= El2) and (Ev /= El3)	if	3
Ev /= El and Ev /= El2 and Ev /= El3	exit	2

C.3.1.6 Response Profile 1:-fff-fFT

Expression	Statement	Occurs
Bv and then (Flv >= Flc) and then (Flv <= Flc2) {flc<flc2}	if	1

C.3.2 Three-Condition Expressions With the Response Profile 2:FFFFFFTF

Expression	Statement	Occurs
(Bv and (Bv2 and not Bv3))	if	2
(Bv and Bv2 and (not Bv3))	:=	1
	if	4
(Bv and Bv2 and not Bv3)	=>	2
(Ev = El) and (Ev2 = El2) and (not Bv)	if	2
Bv and Bv2 and (not Bv3)	:=	1
Bv and Bv2 and not Bv3	:=	7
	if	17

C.3.3 Three-Condition Expressions With the Response Profile 4:FFFFFTFF

Expression	Statement	Occurs
(Bv = True and Bv2 = False and Bv3 = True)	if	1
(Bv = True and Bv2 = False and Bv3)	if	1
(Bv and (not Bv2) and Bv3)	if	3
(Bv and not Bv2 and Bv3)	=>	2
Bv and not (Bv2) and Bv3	:=	1
Bv and not Bv2 and Fxv > url	:=	3
Iv = Iv2 and not Bv and Iv2 /= uil	if	1

C.3.4 Three-Condition Expressions With the Response Profile 7:FFFFFTTT

C.3.4.1 Response Profile 7:FFFFFTTT

Expression	Statement	Occurs
(Bv) and ((Ev = El) or (Iv = Ic))	if	1
(Iv = Ic) and (Bv or Bv2)	if	2
Bv and (Bv2 or Bv3)	:=	1
Iv = Ic and (Iv2 = Ic2 or Iv3 = uil)	if	3
Iv = Iv2 and (Iv3 = uinn or Iv4 = Iv3)	if	5

C.3.4.2 Response Profile 7:FFF-FTT-

Expression	Statement	Occurs
((Ev = El) and ((Ev2 = El2) or (Ev2 = El3)))	if	1

C.3.4.3 Response Profile 7:FfffFTTT

Expression	Statement	Occurs
(Ev in Et and then Bv) or (Ev in Et and then Bv2)	:=	1
Bv and then (Bv2 or Bv3)	if	1

C.3.5 Three-Condition Expressions With the Response Profile 8:FFFFTFFF

Expression	Statement	Occurs
(Bv and (not Bv2) and (not Bv3))	if	1
(Bv and not Bv2 and not Bv3)	=>	1
(Bv) and (not Bv2) and (not Bv3)	:=	1
	if	1
(Ev = El) and not (Bv or Bv2)	if	1
Bv and not (Bv2 or Bv3)	:=	18
	if	5
Bv and not (Bv2) and not (Bv3)	:=	2
	if	1
Bv and not Bv2 and not Bv3	:=	5
	if	6

C.3.6 Three-Condition Expressions With the Response Profile 11:FFFFTFTT

Expression	Statement	Occurs
Bv and (Iv = It'last or not Bv2)	if	2

C.3.7 Three-Condition Expressions With the Response Profile 13:FFFFTTFT

Expression	Statement	Occurs
Bv and ((not Bv2) or (Fxv >= url))	:=	1
Bv and not (Bv2 and not Bv3)	:=	1

C.3.8 Three-Condition Expressions With the Response Profile 14:FFFFTTTF

Expression	Statement	Occurs
(Bv and (not (Bv2 and Bv3)))	=>	2
(Ev /= El) and not (Bv and Bv2)	if	4
(Ev = El) and ((Bv = False) or (Bv2 = False))	if	2
(Iv = uinn) and not (Bv and Bv2)	if	2
Bv and (not Bv2 or not Bv3)	:=	2
Bv and not (Bv2 and Bv3)	if	2
Bv and not (Ev = El and Bv2)	if	2

C.3.9 Three-Condition Expressions With the Response Profile 16:FFFTFFFF

Expression	Statement	Occurs
((not Bv) and Bv2 and Bv3)	if	4
(Bv = False and Bv2 and Bv3)	=>	2
(not Bv and (Bv2) and Bv3)	if	1
(not Bv and Bv2 and Bv3)	=>	1
(not Bv and Bv2) and Bv3	:=	2
(not Bv and not Bv2 and (Fxv < url))	=>	2
(not Bv) and (Bv2) and (Ev = El)	if	3
(not Bv) and (Bv2) and (Flv < url)	if	1
(not Bv) and (Bv2) and Bv3	if	2
(not Bv) and Bv2 and (Ev = El)	if	16
not (Bv) and Bv2 and Bv3	:=	1
not Bv and Bv2 and (Iv = uinn)	if	1
not Bv and Bv2 and Bv3	:=	2
	if	1

C.3.10 Three-Condition Expressions With the Response Profile 20:FFFTFTFF

Expression	Statement	Occurs
((Bv /= Bv2) and Bv3)	=>	2
(Bv = not (Bv2)) and (Iv > Ic)	while	4

C.3.11 Three-Condition Expressions With the Response Profile 21:FFFTFTFT

C.3.11.1 Response Profile 21:FFFTFTFT

Expression	Statement	Occurs
((Iv > Iv2) or (Iv3 = uil)) and (Iv4 = uil)	exit	2
(Bv or Bv2) and (Bv3)	if	1
(Bv or Bv2) and (Fxv < Fxv2)	if	1
(Bv or Bv2) and Bv3	:=	6
	if	1
(Bv or Bv2) and Iv <= Ic	if	1

C.3.11.2 Response Profile 21:FFFTFT--

Expression	Statement	Occurs
((Ev = El or Ev = El2) and Ev2 = El3)	if	5
((Ev = El) or (Ev = El2)) and (Bv = True)	:=	1
(Ev = El or Ev = El2) and (Ev2 = El3)	if	2
(Ev = El or Ev = El2) and Ev2 = El3	if	1

C.3.12 Three-Condition Expressions With the Response Profile 23:FFFTFTTT

Expression	Statement	Occurs
((Bv = True) and (Bv2 = True)) or ((Bv = True) and (Bv3 = True)) or ((Bv2 = True) and (Bv3 = True))	if	1
((Ev = El) and (Ev2 = El)) or ((Ev = El) and (Ev3 = El)) or ((Ev2 = El) and (Ev3 = El))	:=	1
((Ev = El) and (Ev2 = El2)) or ((Ev = El) and (Ev3 = El3)) or ((Ev2 = El2) and (Ev3 = El3))	:=	2
((Flv < url) and (Flv2 < url)) or ((Flv2 < url) and (Flv3 < url)) or ((Flv < url) and (Flv3 < url))	:=	1
(Bv and Bv2) or (Bv and Bv3) or (Bv2 and Bv3)	:=	1

C.3.13 Three-Condition Expressions With the Response Profile 31:FFFTTTTT

C.3.13.1 Response Profile 31:FFFTTTTT

Expression	Statement	Occurs
(Bv = True) or ((Ev = El) and (Bv2 = True))	if	1
(Bv) or (Ev = El and Ev2 = El2)	if	3
(Iv > uinn) or ((Iv2 > Iv3) and (Iv3 /= uil2))	if	2
(Iv >= Ic) or ((Ev /= El) and (Bv))	if	2
(Iv >= Ic) or (Bv and Bv2)	if	2
Bv or ((Fxv > url) and (Fxv2 > url2))	if	1
Bv or (Bv2 and Bv3)	:=	12
	if	5
Bv or (Bv2 and Ev = El)	:=	2

C.3.13.2 Response Profile 31:FFFTTT--

Expression	Statement	Occurs
(Ev = El) or ((Ev = El2) and Bv)	if	2

C.3.13.3 Response Profile 31:FFFTTt--

Expression	Statement	Occurs
(Ev = El) or else ((Ev = El2) and Bv)	if	4
Ev = El or else (Ev = El2 and Bv)	if	6

C.3.13.4 Response Profile 31:FFFTTttt

Expression	Statement	Occurs
Bv = True or else (Bv2 and Bv3)	if	1

C.3.13.5 Response Profile 31:-FFT-TTT

Expression	Statement	Occurs
(Ev /= El) or ((Ev2 /= El) and (Ev2 /= El2))	:=	1

C.3.14 Three-Condition Expressions With the Response Profile 32:FFTFFFFF

Expression	Statement	Occurs
((not Bv) and Bv2 and (not Bv3))	if	1
(Bv = False and Bv2 and Bv3 = False)	=>	1

C.3.15 Three-Condition Expressions With the Response Profile 36:FFTFFTFF

Expression	Statement	Occurs
(Bv /= Bv2 and Bv3 /= Bv2)	=>	1
(Bv xor Bv2) and (not (Bv xor Bv3))	if	1
(Bv xor Bv2) and not (Bv xor Bv3)	if	1

C.3.16 Three-Condition Expressions With the Response Profile 40:FFTFTFFF

Expression	Statement	Occurs
(Bv xor Bv2) and not Bv3	:=	1

C.3.17 Three-Condition Expressions With the Response Profile 42:FFTFTFTF

Expression	Statement	Occurs
(Bv or Bv2) and (not Bv3)	:=	2
(Bv or Bv2) and not Bv3	:=	2
	if	2

C.3.18 Three-Condition Expressions With the Response Profile 43:FFTFTFTT

Expression	Statement	Occurs
(Bv and Bv2) or (Bv and not (Bv3)) or (Bv2 and not (Bv3))	:=	1

C.3.19 Three-Condition Expressions With the Response Profile 47:FFTFTTTT

Expression	Statement	Occurs
Bv or (Bv2 and not Bv3)	:=	2
	if	1

C.3.20 Three-Condition Expressions With the Response Profile 53:FFTTFTFT

Expression	Statement	Occurs
(not Bv and Bv2) or (Bv and Bv3)	:=	1

C.3.21 Three-Condition Expressions With the Response Profile 59:FFTTTFTT

Expression	Statement	Occurs
(Bv and not Bv2 and not Bv3) or Bv2	:=	1

C.3.22 Three-Condition Expressions With the Response Profile 64:FTFFFFFF

Expression	Statement	Occurs
((not Bv) and (not Bv2) and Bv3)	if	3
(not (Bv) and not (Bv2) and Bv3)	if	2
(not Bv) and (not Bv2) and Bv3	:=	2
not Bv and not Bv2 and Bv3	:=	2
	if	2

C.3.23 Three-Condition Expressions With the Response Profile 65:FTFFFFFT

Expression	Statement	Occurs
(Bv = Bv2) and (Iv > Ic)	while	4

C.3.24 Three-Condition Expressions With the Response Profile 66:FTFFFFTF

Expression	Statement	Occurs
(not (Bv xor Bv2)) and (Bv xor Bv3)	if	1
not (Bv xor Bv2) and (Bv xor Bv3)	if	1

C.3.25 Three-Condition Expressions With the Response Profile 69:FTFFFTFT

Expression	Statement	Occurs
(Bv or not Bv2) and Bv3	:=	2

C.3.26 Three-Condition Expressions With the Response Profile 79:FTFFTTTT

Expression	Statement	Occurs
Bv or (not Bv2 and Bv3)	:=	2

C.3.27 Three-Condition Expressions With the Response Profile 81:FTFTFFFT

Expression	Statement	Occurs
(Bv and (Bv2 and Bv3)) or (not Bv and Bv3)	:=	2
not (Bv and not Bv2) and Bv3	:=	1

C.3.28 Three-Condition Expressions With the Response Profile 83:FTFTFFTT

C.3.28.1 Response Profile 83:FTFTFFTT

Expression	Statement	Occurs
((Bv and Bv2) or (not (Bv) and Bv3))	if	2
(Bv and Bv2) or (Bv = False and Bv3)	if	2

C.3.28.2 Response Profile 83:FTF-FFT-

Expression	Statement	Occurs
((Bv) and (Ev = El)) or ((not Bv) and (Ev = El2))	:=	1
(Bv = True and Ev = El) or (Bv = False and Ev = El2)	if	1

C.3.28.3 Response Profile 83:F-FTF-TT

Expression	Statement	Occurs
(Bv and (Fxv > (Fxv2-url))) or (not Bv and (Fxv > Fxv2))	:=	1

C.3.29 Three-Condition Expressions With the Response Profile 84:FTFTFTFF

Expression	Statement	Occurs
((not (Bv and Bv2)) and Bv3)	=>	2

C.3.30 Three-Condition Expressions With the Response Profile 87:FTFTFTTT

C.3.30.1 Response Profile 87:FTFTFTTT

Expression	Statement	Occurs
((Bv and Bv2) or Bv3)	=>	4
	if	2
(Bv and Bv2) or Bv3	:=	7
	if	6
(Bv and Iv = Iv2) or (Iv <= Ic)	if	2
(Bv and Iv > uil) or Iv = Iv2	if	2
(Iv = uinn and Iv2 = uinn2) or Iv3 = Iv4	if	1

C.3.30.2 Response Profile 87:FTFTF-T-

Expression	Statement	Occurs
((Iv = Iv2) and (Iv3 < Iv4)) or (Iv+uil = Iv2)	if	5

C.3.31 Three-Condition Expressions With the Response Profile 91:FTFTTFTT

Expression	Statement	Occurs
(Bv and (Bv2 or (not Bv3))) or ((not Bv) and Bv3)	:=	1

C.3.32 Three-Condition Expressions With the Response Profile 93:FTFTTTFT

C.3.32.1 Response Profile 93:FTFTTTFT

Expression	Statement	Occurs
(Bv and (not Bv2)) or Bv3	:=	2
(Bv and Bv2 = False) or Bv3 = True	if	1
(Bv and not Bv2) or Bv3	:=	1

C.3.32.2 Response Profile 93:FTFTT-F-

Expression	Statement	Occurs
(Ev = El and not Bv) or Ev = El2	if	1

C.3.33 Three-Condition Expressions With the Response Profile 112:FTTTFFFF

C.3.33.1 Response Profile 112:FTTTFFFF

Expression	Statement	Occurs
((Bv = False) and (Bv2 or Bv3))	=>	1
(not Bv) and (Bv2 or Bv3)	:=	1
not (Bv) and (Bv2 or Bv3)	:=	2
not Bv and (Bv2 or Bv3)	:=	3
	if	1

C.3.33.2 Response Profile 112:FTT-FFF-

Expression	Statement	Occurs
(Bv = False) and ((Iv = Ic) or (Iv = Ic2))	if	1
not Bv and (Ev = El or Ev = El2)	exit	8

C.3.34 Three-Condition Expressions With the Response Profile 117:FTTTFTFT

Expression	Statement	Occurs
(not (Bv) and Bv2) or Bv3	:=	2
(not Bv and Bv2) or Bv3	:=	2

C.3.35 Three-Condition Expressions With the Response Profile 127:FTTTTTTT

C.3.35.1 Response Profile 127:FTTTTTTT

Expression	Statement	Occurs
(Bv = True) or (Bv2 = True) or (Bv3 = True)	if	1
(Bv or Bv2 or (Bv3))	if	4
(Bv or Bv2 or Bv3)	:=	2
	=>	3
	if	10
(Ev = El or Bv or Bv2)	if	2
(Ev = El) or (Ev2 = El) or (Ev3 = El)	if	5
(Ev = El) or Bv or Bv2	if	1
(Ev = Ev2) or (Ev3 = Ev2) or (Ev4 = Ev2)	if	3
(Iv < Iv2) or (Iv < Iv3) or (Iv < Iv4)	if	1
(Iv = Ic) or (Iv2 = Ic) or (Iv3 = Ic)	if	1
(Iv = Iv2) or (Iv = Iv3) or (Iv = Iv4)	if	1
(Iv > Iv2) or (Iv = uil) or (Ev /= El)	if	1
(Iv > uil) or (Iv2 > uil) or Bv	:=	2
Bv or Bv2 or (Ev = El)	:=	2
Bv or Bv2 or Bv3	:=	72
	if	9
Ev /= El or Ev2 /= El or Bv	:=	1
Ev = El or Bv or Ev2 = El2	if	3
Iv /= Iv2 or Iv3 /= Iv4 or Iv5 /= Iv6	if	6

C.3.35.2 Response Profile 127:FTTTTT--

Expression	Statement	Occurs
(Ev = El) or (Ev = El2) or (Bv = True)	if	3

C.3.35.3 Response Profile 127:FTT-TTT-

Expression	Statement	Occurs
((Ev = El) or (Iv = Ic) or (Iv = Ic2))	if	3
(Ev = El) or ((Ev2 in El2..El3) or (Ev2 = El4)) {el4<el2<el3}	if	2

C.3.35.4 Response Profile 127:FTT-T---

Expression	Statement	Occurs
((Ev = El) or (Ev = El2) or (Ev = El3))	if	1
(Ev = El) or (Ev = El2) or (Ev = El3)	if	1
(Iv = Ic) or (Iv = Ic2) or (Iv = Ic3)	if	4
(Iv = uil) or (Iv = uil2) or (Iv = uil3)	if	3

C.3.36 Three-Condition Expressions With the Response Profile 128:TFFFFFFF

C.3.36.1 Response Profile 128:TFFFFFFF

Expression	Statement	Occurs
((not Bv) and (not Bv2) and (not Bv3))	if	2
(not Bv) and (not Bv2) and (not Bv3)	:=	1
Bv = False and Bv2 = False and Bv3 = False	if	1
not (Bv or Bv2 or Bv3)	:=	5
	=>	1
	if	1
not (Bv or Bv2) and not (Bv3)	:=	1
not Bv and not Bv2 and not Bv3	:=	1
	if	1

C.3.36.2 Response Profile 128:TFF-F---

Expression	Statement	Occurs
not (Ev = El or else Ev = El2 or else Ev = El3)	if	1

C.3.37 Three-Condition Expressions With the Response Profile 129:TFFFFFFT

Expression	Statement	Occurs
((Bv = Bv2) and (Bv = Bv3))	if	1
(Bv = Bv2) and (Bv = Bv3)	if	2

C.3.38 Three-Condition Expressions With the Response Profile 159:TFFTTTTT

Expression	Statement	Occurs
Bv or (not (Bv2 xor Bv3))	if	8

C.3.39 Three-Condition Expressions With the Response Profile 168:TFTFTFFF

Expression	Statement	Occurs
not ((Bv and (Fxv >= Fxc)) or Bv2)	if	1

C.3.40 Three-Condition Expressions With the Response Profile 178:TFTTFFTF

Expression	Statement	Occurs
(not Bv and Bv2) or (not Bv3 and not Bv) or (Bv2 and not Bv3)	:=	2

C.3.41 Three-Condition Expressions With the Response Profile 186:TFTTTFTF

C.3.41.1 Response Profile 186:-FTT-FTF

Expression	Statement	Occurs
((not Bv and (Flv >= url) and (Flv < url2)) or (Flv >= url2)) {url<url2}	if	1

C.3.42 Three-Condition Expressions With the Response Profile 202:TTFFTFTF

Expression	Statement	Occurs
not (((not Bv) and Bv2) or (Bv and Bv3))	:=	1

C.3.43 Three-Condition Expressions With the Response Profile 208:TTFTFFFF

Expression	Statement	Occurs
not (Bv or (Bv2 and not Bv3))	:=	1
not Bv and (not Bv2 or Bv3)	:=	3

C.3.44 Three-Condition Expressions With the Response Profile 224:TTTFFFFF

Expression	Statement	Occurs
(not Bv) and (not Bv2 or not Bv3)	:=	1

C.3.45 Three-Condition Expressions With the Response Profile 234:TTTFTFTF

Expression	Statement	Occurs
not (Bv or Bv2) or not Bv3	:=	1

C.3.46 Three-Condition Expressions With the Response Profile 235:TTTFTFTT

Expression	Statement	Occurs
(Bv = Bv2) or not (Bv3)	if	2

C.3.47 Three-Condition Expressions With the Response Profile 239:TTTFTTTT

Expression	Statement	Occurs
Bv or not (Bv2) or not (Ev = El)	:=	2

C.3.48 Three-Condition Expressions With the Response Profile 242:TTTTFFTF

Expression	Statement	Occurs
(not Bv) or (Bv2 and not Bv3)	if	1

C.3.49 Three-Condition Expressions With the Response Profile 247:TTTTFTTT

Expression	Statement	Occurs
not (Bv) or Bv2 or Bv3	:=	1

C.3.50 Three-Condition Expressions With the Response Profile 248:TTTTTFFF

C.3.50.1 Response Profile 248:Ttt-TFF-

Expression	Statement	Occurs
not (Iv = uil and then (Ev = El or Ev = El2))	exit	1

C.3.51 Three-Condition Expressions With the Response Profile 253:TTTTTTFT

Expression	Statement	Occurs
not Bv or not Bv2 or (Iv > uil)	if	8

C.3.52 Three-Condition Expressions With the Response Profile 254:TTTTTTTF

Expression	Statement	Occurs
not (Bv and Bv2 and Bv3)	:=	6
	if	1
not Bv or not Bv2 or not Bv3	:=	1

C.4 Expressions With Four Conditions

C.4.1 Four-Condition Expressions With the Response Profile 1:FFFFFFFFFFFFFFFT

C.4.1.1 Response Profile 1:FFFFFFFFFFFFFFFT

Expression	Statement	Occurs
((Bv = True) and (Ev = Ec)) and ((Bv2 = True) and (Ev2 = Ec))	if	3
((Iv = uil) and (Rv = Rl) and (Ev = El) and (Iv2 = uil))	if	3
(Bv and Bv2 and Bv3 and Bv4)	if	2
(Ev = El) and (Ev2 = El) and (Ev3 = El) and (Ev4 = El)	if	1
(Ev = El) and Bv and Bv2 and Bv3	:=	1
(Ev = Ev2) and (Ev3 in El..El2) and (Ev4 = El3) and (Bv = True)	if	1
Bv and Bv2 and Bv3 and Bv4	:=	3
	if	2
Ev = El and Ev2 /= Ev3 and Iv = uil and Iv2 = uil	if	3
Ev = El and Ev2 = El and Ev3 = El and Ev4 = El	:=	1
Ev = El and Ev2 = El and Ev3 = El and Ev4 = El2	:=	1

C.4.1.2 Response Profile 1:FffffffFFFFFFFT

Expression	Statement	Occurs
(Ev = El) and then ((Iv = uil) and (Ev2 = El2) and (Iv2 = uil))	if	4

C.4.1.3 Response Profile 1:FffffffFfffFfFT

Expression	Statement	Occurs
Bv = True and then Bv2 = True and then Bv3 = True and then Flv > Flc	if	1
Bv and then Bv2 and then Bv3 and then Bv4	if	1
Cv = Cl and then Cv2 = Cl and then Cv3 = Cl and then Cv4 = Cl	if	3
Iv = Iv2 and then Iv3 = Ic and then Iv4 = Ic2 and then Iv5 = Iv6	if	1

C.4.1.4 Response Profile 1:--FFFFFF--FFFFFT

Expression	Statement	Occurs
((Iv /= uil) and (Iv2 /= uil2) and (Iv2 /= uil3) and (Ev = El))	if	1
(Bv) and (Ev /= El) and (Ev /= El2) and (Bv2)	if	4

C.4.1.5 Response Profile 1:----FFFFFFFFFFT

Expression	Statement	Occurs
Ev /= El and Ev /= El2 and Iv = uil and Iv2 = uil	if	3

C.4.1.6 Response Profile 1:------FF--FFFFTF

Expression	Statement	Occurs
Ev /= El and Ev /= El2 and Ev /= El3 and not (Bv)	:=	3

C.4.1.7 Response Profile 1:-------F---F-FFT

Expression	Statement	Occurs
Iv /= uil and Iv /= uil2 and Iv /= uil3 and Iv /= uil4	if	1

C.4.1.8 Response Profile 1:-------f---f-fFT

Expression	Statement	Occurs
Iv /= Ic and then Iv /= Ic2 and then Iv /= Ic3 and then Iv /= Ic4	if	4

C.4.2 Four-Condition Expressions With the Response Profile 2:FFFFFFFFFFFFFTF

Expression	Statement	Occurs
(Bv and (Fxv > url) and (Ev /= El) and not Bv2)	=>	1
(Bv and Bv2 and Bv3) and not Bv4	:=	1
Bv and Bv2 and Bv3 and (not Bv4)	:=	2
Bv and Bv2 and Bv3 and not Bv4	:=	2
Ev = El and Bv and Bv2 and not Bv3	if	2

C.4.3 Four-Condition Expressions With the Response Profile 4:FFFFFFFFFFFFTFF

Expression	Statement	Occurs
Bv and Bv2 and not Bv3 and Bv4	:=	1

C.4.4 Four-Condition Expressions With the Response Profile 7:FFFFFFFFFFFFTTT

Expression	Statement	Occurs
(Ev = El) and Bv and (Bv2 or Bv3)	if	1
Bv and Bv2 and (Bv3 or Bv4)	:=	1

C.4.5 Four-Condition Expressions With the Response Profile 8:FFFFFFFFFFFTFFF

Expression	Statement	Occurs
(Bv and Bv2 and not Bv3 and not Bv4)	:=	1
(Ev = El) and Bv and (not Bv2) and (not Bv3)	if	2
Bv and Bv2 and not (Bv3 or Bv4)	if	2

C.4.6 Four-Condition Expressions With the Response Profile 13:FFFFFFFFFFFTTFT

Expression	Statement	Occurs
(Bv and Bv2) and not (Bv3 and not Bv4)	:=	2

C.4.7 Four-Condition Expressions With the Response Profile 32:FFFFFFFFFTFFFFF

Expression	Statement	Occurs
Bv and not (Bv2) and Bv3 and not (Bv4)	:=	2
Bv and not Bv2 and (Iv <= uil) and not Bv3	:=	2
Bv and not Bv2 and Bv3 and not Bv4	:=	2

C.4.8 Four-Condition Expressions With the Response Profile 64:FFFFFFFFTFFFFFF

Expression	Statement	Occurs
Bv = True and Bv2 = False and Bv3 = False and Bv4 = True	:=	1
Bv and not Bv2 and not Bv3 and (Fxv < url)	:=	1
Bv and not Bv2 and not Bv3 and Iv <= uil	:=	1

C.4.9 Four-Condition Expressions With the Response Profile 127:FFFFFFFFFTTTTTTT

C.4.9.1 Response Profile 127:FFFFFFFFFTTTTTTT

Expression	Statement	Occurs
Bv and (Bv2 or Bv3 or Bv4)	:=	1

C.4.9.2 Response Profile 127:FFF-FFF-FTT-TTT-

Expression	Statement	Occurs
(Ev = El) and ((Ev2 = El2) or (Iv = Ic) or (Iv = Ic2))	if	3

C.4.9.3 Response Profile 127:FFF-F---FTT-T---

Expression	Statement	Occurs
Bv and (Iv = Ic or Iv = Ic2 or Iv = Ic3)	if	2

C.4.10 Four-Condition Expressions With the Response Profile 128:FFFFFFFFTFFFFFFF

Expression	Statement	Occurs
Bv = True and Bv2 = False and Bv3 = False and Bv4 = False	:=	1
Bv and not (Bv2 or Bv3 or Bv4)	:=	3
Bv and not Bv2 and not Bv3 and not Bv4	:=	2

C.4.11 Four-Condition Expressions With the Response Profile 256:FFFFFFFTFFFFFFFF

C.4.11.1 Response Profile 256:FFFFFFFTFFFFFFFF

Expression	Statement	Occurs
(Bv = False and Bv2 and Bv3 and Bv4)	=>	2
(not Bv and Bv2 and Bv3 and Bv4)	=>	2
not Bv and Bv2 and Bv3 and Bv4	:=	1

C.4.11.2 Response Profile 256:-FFF-FFT-FFF-FFF

Expression	Statement	Occurs
not Bv and Bv2 and (Fxv < url) and (Fxv >= url2) {url>url2}	if	1

C.4.12 Four-Condition Expressions With the Response Profile 273:FFFFFFFTFFFTFFFT

C.4.12.1 Response Profile 273:FFFFFFFTFFFTFFFT

Expression	Statement	Occurs
((Flv > Flv2) or (Flv3 > Flv2)) and (Flv4 >= url) and (Ev = El)	:=	1
((Iv = uil or Ev = El) and Ev2 /= El2 and Bv)	if	3
(Bv or Bv2) and (Bv3 and Bv4)	if	2
(Bv or Bv2) and Bv3 and Bv4	:=	7

C.4.12.2 Response Profile 273:FFFFFFFTFFFT----

Expression	Statement	Occurs
((Ev = El or Ev = El2) and Ev2 = El3 and Iv < Ic)	if	5

C.4.13 Four-Condition Expressions With the Response Profile 341:FFFFFFFTFTFTFTFF

Expression	Statement	Occurs
(Bv or (Bv2 and Bv3)) and Bv4	:=	2
(Bv xor (Bv2 and Bv3)) and Bv4	if	2

C.4.14 Four-Condition Expressions With the Response Profile 512:FFFFFFTFFFFFFFFF

Expression	Statement	Occurs
((not Bv) and Bv2 and (Ev >= El) and (not Bv3))	:=	2
(not Bv) and (Flv < Flv2) and (Flv3 <= url) and (not Bv2)	:=	2
not Bv and Bv2 and Bv3 and not Bv4	:=	3

C.4.15 Four-Condition Expressions With the Response Profile 1024: FFFFFTFFFFFFFFF

Expression	Statement	Occurs
not (Bv) and Bv2 and not (Bv3) and Bv4	:=	2
not Bv and Bv2 and not Bv3 and Bv4	:=	2

C.4.16 Four-Condition Expressions With the Response Profile 1279:FFFFFTFFTTTTTTTT

Expression	Statement	Occurs
Bv or (Bv2 and not Bv3 and Bv4)	:=	1

C.4.17 Four-Condition Expressions With the Response Profile 1792: FFFFFTTTFFFFFFFF

Expression	Statement	Occurs
(not Bv) and (Bv2) and (Bv3 or Bv4)	:=	2

C.4.18 Four-Condition Expressions With the Response Profile 1911:FFFFFTTTFTTTFTTT

Expression	Statement	Occurs
((Bv) or (Bv2)) and ((Ev = El) or (Iv = Ic))	if	4
(Bv or Bv2) and (Bv3 or Bv4)	:=	4

C.4.19 Four-Condition Expressions With the Response Profile 2176:FFFFTFFFTFFFFFFF

Expression	Statement	Occurs
(Bv xor (Iv = Ic)) and not (Bv2) and not (Bv3)	:=	2

C.4.20 Four-Condition Expressions With the Response Profile 2184:FFFFTFFFTFFFTFFF

Expression	Statement	Occurs
(Bv or Bv2) and not Bv3 and not Bv4	:=	2

C.4.21 Four-Condition Expressions With the Response Profile 2303:FFFFTFFFTTTTTTTT

Expression	Statement	Occurs
Bv or (Bv2 and (not Bv3) and (not Bv4))	:=	2

C.4.22 Four-Condition Expressions With the Response Profile 3328:FFFFTTFTFFFFFFFF

Expression	Statement	Occurs
not (Bv) and Bv2 and not (Bv3 and not (Bv4))	:=	2

C.4.23 Four-Condition Expressions With the Response Profile 4096:FFFTFFFFFFFFFFF

Expression	Statement	Occurs
not Bv and not Bv2 and Bv3 and (Iv < Ic)	:=	4

C.4.24 Four-Condition Expressions With the Response Profile 4373:FFFTFFFTFFFTFTFT

Expression	Statement	Occurs
(((Bv = True) and (Bv2 = True)) or (Bv3 = True)) and (Iv = Ic)	if	1

C.4.25 Four-Condition Expressions With the Response Profile 4383:FFFTFFFTFFFTTTTT

C.4.25.1 Response Profile 4383:FFFTFFFTFFFTTTTT

Expression	Statement	Occurs
(Bv and Bv2) or (Bv3 and Bv4)	:=	12
	if	1
(Iv /= Iv2 and Iv /= Iv3) or (Iv4 /= Iv5 and Iv4 /= Iv6)	:=	3

C.4.25.2 Response Profile 4383:FFFTF-F-FF--T---

Expression	Statement	Occurs
(Fxv > url and Fxv2 < url) or (Fxv < url and Fxv2 > url)	:=	1

C.4.25.3 Response Profile 4383:FfFTffftFfFTTttt

Expression	Statement	Occurs
(Bv and then Ev = El) or else (Bv2 and then Ev2 in Et)	if	1

C.4.25.4 Response Profile 4383:--FT--F-FFFTT-T-

Expression	Statement	Occurs
((Fxv <= url) and (Fxv2 < url)) or ((Fxv >= url) and (Fxv2 > url))	if	4

C.4.25.5 Response Profile 4383:-----F---FFT-T--

Expression	Statement	Occurs
((Fxv > url) and (Fxv <= url2)) or ((Fxv > url3) and (Fxv <= url4)) {url<url2<url3<url4}	:=	1

C.4.26 Four-Condition Expressions With the Response Profile 4415:FFFTFFFTFFTTTTTT

C.4.26.1 Response Profile 4415:FFfTffftF-T-T-t-

Expression	Statement	Occurs
(Iv = Ic and then Iv2 = Iv3) or else (Iv = Ic and then Iv2 = Ic2) or else (Iv = Ic3 and then Iv2 = Ic2)	if	3

C.4.27 Four-Condition Expressions With the Response Profile 5461:FFFTFTFTFTFTFTFT

C.4.27.1 Response Profile 5461:FFFTFTFTFTFTFTFT

Expression	Statement	Occurs
((Bv = True) or (Bv2 = True) or (Bv3 = True)) and (Iv = uil)	if	1
((Bv or Bv2 or Bv3) and Bv4)	:=	1
((Iv = Ic) or (Iv2 = Ic) or (Iv3 = Ic)) and (Ev = El)	if	1
(Bv or Bv2 or Bv3) and (Flv > Flc)	if	1

C.4.27.2 Response Profile 5461:FFFTFT--FT------

Expression	Statement	Occurs
((Iv = Ic) or (Iv = Ic2) or (Iv = Ic3)) and (Bv = True)	if	1

C.4.28 Four-Condition Expressions With the Response Profile 8191:FFFTTTTTTTTTTTTT

Expression	Statement	Occurs
(Fxv > url) or (Fxv2 > url2) or (Bv and Bv2)	:=	1
Bv or Bv2 or (Bv3 and Bv4)	:=	3
	if	3

C.4.29 Four-Condition Expressions With the Response Profile 10922:FFTFTFTFTFTFTFTF

Expression	Statement	Occurs
(Bv or Bv2 or Bv3) and not Bv4	:=	2

C.4.30 Four-Condition Expressions With the Response Profile 12287:FFTFTTTTTTTTTTTT

Expression	Statement	Occurs
Ev /= El or Ev2 /= El or (Bv and not (Bv2))	:=	1

C.4.31 Four-Condition Expressions With the Response Profile 16384:FTFFFFFFFFFFFFFF

Expression	Statement	Occurs
(not Bv) and (not Bv2) and (not Bv3) and (Bv4)	:=	1

C.4.32 Four-Condition Expressions With the Response Profile 16576:FTFFFFFFTTFFFFFF

C.4.32.1 Response Profile 16576:FTfFfFfFTTFFFFFF

Expression	Statement	Occurs
(Bv and then (not Bv2 and not Bv3)) or (Bv4 and then (not Bv2 and not Bv3))	if	2

C.4.33 Four-Condition Expressions With the Response Profile 18431:FTFFFTTTTTTTTTTT

C.4.33.1 Response Profile 18431:F-F-FTTTT-T-----

Expression	Statement	Occurs
(Fxv < url) or ((Fxv > url2) and (Ev /= El)) or ((Fxv > url3) and (Ev = El)) {url<url2<url3}	:=	1

C.4.34 Four-Condition Expressions With the Response Profile 20292:FTFFTTTTFTFFFTFF

Expression	Statement	Occurs
((not Bv and Bv2) or (not Bv3 and Bv4))	if	8

C.4.35 Four-Condition Expressions With the Response Profile 21844:FTFTFTFTFTFTFTFF

Expression	Statement	Occurs
not (Bv and Bv2 and Bv3) and Bv4	if	2

C.4.36 Four-Condition Expressions With the Response Profile 21847:FTFTFTFTFTFTFTTT

C.4.36.1 Response Profile 21847:FTFTFTFTFTFTFTTT

Expression	Statement	Occurs
(Bv and Bv2 and Bv3) or Bv4	:=	2

C.4.36.2 Response Profile 21847:------FT--FTFTTT

Expression	Statement	Occurs
(Ev /= El and Ev /= El2 and Ev /= El3) or Bv	if	3

C.4.37 Four-Condition Expressions With the Response Profile 22357:FTFTFTTTFTFTFTFT

Expression	Statement	Occurs
((not Bv) and Bv2 and (Ev = El)) or Bv3	:=	2

C.4.38 Four-Condition Expressions With the Response Profile 22391:FTFTFTTTFTTTFTTT

Expression	Statement	Occurs
((Bv or Bv2) and Bv3) or Bv4	:=	2

C.4.39 Four-Condition Expressions With the Response Profile 22527:FTFTFTTTTTTTTTTT

Expression	Statement	Occurs
(Ev = El or (Bv and Ev2 /= El2) or Bv2)	if	1
Bv or (Bv2 and Bv3) or Bv4	:=	2

C.4.40 Four-Condition Expressions With the Response Profile 24063:FTFTTTFTTTTTTTTT

Expression	Statement	Occurs
Bv or (Bv2 and not Bv3) or Bv4	:=	4

C.4.41 Four-Condition Expressions With the Response Profile 24576:FTTFFFFFFFFFFFFF

Expression	Statement	Occurs
not Bv and not Bv2 and (Bv3 xor Bv4)	if	2

C.4.42 Four-Condition Expressions With the Response Profile 28627:FTTTFFFFFFFFFFF

Expression	Statement	Occurs
not (Bv or Bv2) and (Bv3 or Bv4)	:=	1

C.4.43 Four-Condition Expressions With the Response Profile 28672:FTTTFFFFFFFFFFF

Expression	Statement	Occurs
not Bv and not Bv2 and (Bv3 or Bv4)	:=	3

C.4.44 Four-Condition Expressions With the Response Profile 30576:FTTTFTTTFTTTFFF

Expression	Statement	Occurs
((Bv = False) or (Bv2 = False)) and ((Ev = El) or (Ev2 = El))	if	1
(not Bv or not Bv2) and (Bv3 or Bv4)	:=	1

C.4.45 Four-Condition Expressions With the Response Profile 30591:FTTTFTTTFTTTTTT

C.4.45.1 Response Profile 30591:FTTTFTTTFTTTTTT

Expression	Statement	Occurs
((Fxv > Fxc) and Bv) or (Bv2 or Bv3)	if	1

C.4.45.2 Response Profile 30591:----FTTTFTTTTTT

Expression	Statement	Occurs
((Fxv < url) and (Fxv >= url2)) or Bv or Bv2 {url>url2}	:=	1

C.4.45.3 Response Profile 30591:----FTT-F---T---

Expression	Statement	Occurs
((Ev >= El) and (Ev <= El2)) or (Ev = El3) or (Ev = El4) {el3<el4<el<el2}	if	1

C.4.46 Four-Condition Expressions With the Response Profile 32512:FTTTTTTTFFFFFFF

C.4.46.1 Response Profile 32512:FTT-T---FFF-F---

Expression	Statement	Occurs
(Bv = False) and ((Iv = Ic) or (Iv = Ic2) or (Iv = Ic3))	if	1

C.4.47 Four-Condition Expressions With the Response Profile 32767:FTTTTTTTTTTTTTT

C.4.47.1 Response Profile 32767:FTTTTTTTTTTTTTT

Expression	Statement	Occurs
(Av = Av2) or (Av3 = Av4) or (Av3 = Av5) or (Av3 = Av6)	if	4
(Bv or Bv2 or Bv3 or Bv4)	:=	1
	if	11
(Bv or Bv2) or (Bv3 or Bv4)	:=	1
(Ev = El or Ev2 = El or Ev3 = El or Ev4 = El)	if	6
(Ev = El) or (Ev2 = El) or (Ev3 = El) or (Ev4 = El)	if	2
(Iv /= Ic) or (Iv2 /= Ic2) or (Iv3 /= Ic3) or (Iv4 /= Ic4)	if	2
(Iv /= uil) or (Iv2 /= uil2) or (Iv3 /= uil) or (Iv4 /= uil2)	if	2
(Iv = Ic) or (Iv2 = Ic2) or (Iv3 = Ic3) or (Iv4 = Ic4)	if	1
(Iv > uil or Iv2 > uil2 or Iv3 > uil2 or Iv4 > uil2)	if	1
Bv = True or Bv2 = True or Bv3 = True or Bv4 = True	if	1
Bv or Bv2 or Bv3 or Bv4	:=	70
	if	7
Iv > uil or Iv2 > uil or Iv3 > uil or Iv4 > uil	if	8

C.4.47.2 Response Profile 32767:FTTTTTTTTTTT----

Expression	Statement	Occurs
(Ev = El or Ev = El2 or Bv or Bv2)	if	1

C.4.47.3 Response Profile 32767:FTTTTT--TT------

Expression	Statement	Occurs
(Ev = El) or (Ev = El2) or (Ev = El3) or Bv	:=	1

C.4.47.4 Response Profile 32767:FTTTT-T-TT--T---

Expression	Statement	Occurs
(Iv > Ic) or (Iv2 > Ic) or (Iv < Ic2) or (Iv2 < Ic2) {ic>ic2}	if	1

C.4.47.5 Response Profile 32767:FTT-TTT-TTT-----

Expression	Statement	Occurs
((Iv < Ic) or (Iv > Ic2)) or ((Iv2 < Ic) or (Iv2 > Ic2)) {ic<ic2}	if	3
(Iv < Ic) or (Iv > Ic2) or (Iv2 < Ic) or (Iv2 > Ic2) {ic<ic2}	if	1

C.4.47.6 Response Profile 32767:FTT-T---T-------

Expression	Statement	Occurs
((Ev = El) or (Ev = El2) or (Ev = El3) or (Ev = El4))	if	2
(Ev = El) or (Ev = El2) or (Ev = El3) or (Ev = El4)	if	4
(Iv = uil) or (Iv = uil2) or (Iv = uil3) or (Iv = uil4)	if	2
(Iv in uil..uil2) or (Iv in uil3..uil4) or (Iv in uil5..uil6) or (Iv in uil7..uil8) {uil<uil2<uil3<uil4<uil5<uil6}	if	6
Ev = El or Ev = El2 or Ev = El3 or Ev = El4	if	1

C.4.48 Four-Condition Expressions With the Response Profile 32768:TFFFFFFFFFFFFFFF

C.4.48.1 Response Profile 32768:TFFFFFFFFFFFFFFF

Expression	Statement	Occurs
((not Bv) and (not Bv2) and not Bv3 and not Bv4)	if	2
(not (Bv or Bv2)) and (not (Bv3 or Bv4))	:=	1
not Bv and not Bv2 and not Bv3 and not Bv4	:=	2
	if	1

C.4.48.2 Response Profile 32768:TFF-F---F-------

Expression	Statement	Occurs
not ((Iv = uil) or (Iv = uil2) or (Iv = uil3) or (Iv = uil4))	if	1

C.4.49 Four-Condition Expressions With the Response Profile 32769:TFFFFFFFFFFFFFFT

Expression	Statement	Occurs
(Bv = Bv2) and (Bv2 = Bv3) and (Bv3 = Bv4)	if	1
(Bv and Bv2 and Bv3 and Bv4) or (not Bv and not Bv2 and not Bv3 and not Bv4)	:=	1

C.4.50 Four-Condition Expressions With the Response Profile 53248:TTFTFFFFFFFFFFFF

Expression	Statement	Occurs
not ((Bv or Bv2) or (Bv3 and not Bv4))	:=	1

C.4.51 Four-Condition Expressions With the Response Profile 61999:TTTTFFTFFFTFTTTT

Expression	Statement	Occurs
(Bv = Bv2) or (Bv3 and not Bv4)	if	1
Bv = Bv2 or (Bv3 and not Bv4)	if	3

C.4.52 Four-Condition Expressions With the Response Profile 63624:TTTTTFFFTFFFTFFF

Expression	Statement	Occurs
(not Bv and not Bv2) or (not Bv3 and not Bv4)	:=	1

C.4.53 Four-Condition Expressions With the Response Profile 65456:TTTTTTTTTFTTFFFF

Expression	Statement	Occurs
not ((Ev = El and Bv) or (Ev = El and not Bv2 and Bv3))	if	1

C.4.54 Four-Condition Expressions With the Response Profile 65493:TTTTTTTTTTFTFTFT

Expression	Statement	Occurs
not Bv or (not Bv2 and not Bv3) or Bv4	:=	2

C.4.55 Four-Condition Expressions With the Response Profile 65524:TTTTTTTTTTTTFTFF

Expression	Statement	Occurs
(not Bv) or (not Bv2) or ((not Bv3) and Bv4)	if	1

C.4.56 Four-Condition Expressions With the Response Profile 65534:TTTTTTTTTTTTTTTF

Expression	Statement	Occurs
(Bv = False) or (Bv2 = False) or (Bv3 = False) or (Bv4 = False)	if	1

C.5 Expressions With Five Conditions

Expression	Statement	Occurs
((((Av = Av2) or (Av = Av3)) and (Av4 = null)) or (Av5 = Av4)) and (Av6 = null)	if	4
((((Flv >= url) and (Bv)) or ((Flv >= url) and (Flv <= url2))) and (Ev = El)) or Bv2 {url < url2}	:=	1
((Bv and not Bv2) or (Bv3 and not Bv4) or Bv5)	:=	2
((Bv or Bv2) and not Bv3) or Bv4 or Bv5	:=	1
((Bv) and (Bv2 = True) and (Bv3 = True)) or ((not Bv4) and (Bv5) and (Bv3 = True)) or ((Bv4) and (not Bv5) and (Bv2 = True))	if	2
((Ev <= El) or (Ev2 > El2)) and not Bv and Bv2 and Bv3	:=	2
((Ev = El and Bv) or (Bv2 and Ev2 = El2 and not Bv3))	if	1
((Iv = Ic and Iv2 /= Ic2 and Iv2 /= Ic3) or Iv = Ic4) and Bv	if	3
((not Bv or Bv2) and not (Bv3 or Bv4)) or Bv5	if	1
((not Bv) and (Bv2) and (Bv3)) and ((not Bv4) or (Ev = El))	if	1
((not Bv) and (not Bv2) and Bv3 and Bv4 and Bv5)	:=	1
	return	1
((not Bv) and Bv2 and ((Bv3 and (Bv4 or Bv5)) or (Bv4 and Bv5)))	return	1
((not Bv) and Bv2 and (not Bv3) and (not Bv4) and (not Bv5))	return	1

Expression	Statement	Occurs
((not Bv) and Bv2 and Bv3 and (Bv4 or Bv5))	=>	1
(Bv = Bv2) and (Bv3 = Bv2) and (Bv4 = Bv2) and (Bv5 = Bv2)	if	1
(Bv = False and Bv2 = False and Bv3 = False and Bv4 = False and Bv5)	=>	1
(Bv = False and Bv2 and Bv3 and Bv4 and Bv5)	=>	2
(Bv = True) or (Iv /= uil) or ((Iv2 /= uil) and (Iv2 /= uil2) and (Iv2 /= uil3))	if	1
(Bv and (Iv /= Ic)) or (Bv2 and (Iv /= Ic)) or Bv3 or Bv4	if	2
(Bv and (not Bv2) and Bv3 and Bv4 and Bv5)	return	1
(Bv and Bv2 and (not Bv3) and (not Bv4) and (not Bv5))	:=	1
	return	1
(Bv and Bv2 and Bv3 and Bv4 and Bv5)	=>	1
(Bv and Bv2 and Bv3 and Bv4) and (not Bv5)	:=	3
(Bv and Bv2 and Bv3) or Bv4 or Bv5	if	4
(Bv and Bv2) or (Bv3 and Bv4) or Bv5	:=	1
(Bv or Bv2 or Bv3 or Bv4) or Bv5	if	1
(Bv or Bv2) and (Bv3 or Bv4) and Bv5	:=	1
(Bv or Bv2) and (not Bv3 or not Bv4 or Bv5)	:=	1
(Bv or Bv2) and not (Bv3 or Bv4 or Bv5)	if	1
(Bv xor Bv2) or Bv3 or Bv4 or Bv5	:=	1
(Bv) and (not (Bv2)) and (not (Bv3)) and (not (Bv4)) and (not (Bv5))	:=	2
(Ev = El or Ev = El2 or Ev = El3) and Bv and Bv2	if	1
(Iv /= uil) and (Iv /= uil2) and (Iv /= uil3) and (Iv /= uil4) and (Iv /= uil5)	if	1
(Iv /= uil) or (Iv2 /= uil2) or (Iv3 /= uil) or (Iv4 /= uil2) or (Iv5 /= uil)	if	1
(Iv /= uil) or (Iv2 /= uil2) or (Iv3 /= uil2) or (Iv4 /= uil) or (Iv5 /= uil3)	if	1
(Iv /= uil) or (Iv2 /= uil2) or (Iv3 /= uil3) or (Iv4 /= uil2) or (Iv5 /= uil3)	if	1
(Iv = Ic and then Iv2 = Iv) or else (Iv = Ic and then Iv2 = Ic2) or else (Iv = Ic3 and then Iv2 = Ic2) or else (Iv = Ic4)	if	2
(Iv = Ic and then Iv2 = Iv3) or else (Iv = Ic and then Iv2 = Ic2) or else (Iv = Ic3 and then Iv2 = Ic2) or else (Iv = Ic4)	if	1
(Iv = uil) and (Iv2 = uil) and (Iv3 /= Iv4 or Iv5 /= Iv6 or Iv7 /= Iv8)	if	3
(Iv > Ic) or (Iv = uil) or (Iv > Iv2) or (Iv3 = uil) or (Bv = True)	if	2
(not Bv or not Bv2 or (Iv > uil)) or (not Bv3 or (Iv2 > uil))	if	8
(not Bv) and (not Bv2) and (Bv3 or Bv4 or Bv5)	:=	2
(not Bv) and (not Bv2) and (Bv3) and (Bv4 or Bv5)	:=	3
(not Bv) and (not Bv2) and (not Bv3) and Bv4 and Bv5	:=	1
(not Bv) and (not Bv2) and Bv3 and Bv4 and Bv5	:=	1
Bv and (Bv2 or Bv3 or (Bv4 and not Bv5))	if	1
Bv and (Iv = Iv2) and (Iv > Ic) and (Iv3 <= Ic2) and (Iv3 >= Ic3) {Ic2 > Ic3}	if	2
Bv and (not Bv2 and (not (Bv3 or Bv4 or Bv5)))	:=	2
Bv and (not Bv2) and (Fxv < url) and (Bv3 or Bv4)	:=	1
Bv and Bv2 and (not Bv3) and (not Bv4) and (not Bv5)	:=	1
Bv and Bv2 and Bv3 and (not Bv4) and (not Bv5)	:=	1

Expression	Statement	Occurs
Bv and Bv2 and Bv3 and Bv4 and Bv5	if	8
Bv and Bv2 and Bv3 and not Bv4 and not Bv5	:=	1
Bv and Ev = El and Ev2 = El2 and Ev3 = El and Ev4 = El2	:=	2
Bv and not Bv2 and not (Bv3 or Bv4) and not Bv5	if	2
Bv and not Bv2 and not Bv3 and not Bv4 and not Bv5	:=	1
Bv or ((Bv2 and not Bv3) and Bv4 and not Bv5)	:=	1
Bv or (Bv2 and ((Ev = El) or (Ev = El2) or (Ev = El3)))	:=	1
Bv or (Bv2 and not Bv3) or (Bv4 and not Bv5)	:=	2
Bv or (Bv2 or Bv3 or Bv4 or Bv5)	if	2
Bv or Bv2 or (Ev in Et and then Bv3) or (Ev in Et and then Bv4)	if	1
Bv or Bv2 or Bv3 or Bv4 or Bv5	:=	10
	if	1
Cv = Cl and then Cv2 = Cl and then Cv3 = Cl and then Cv4 = Cl and then Cv5 = Cl	if	2
Ev = El or Ev = El2 or (Iv = Iv2 and Iv3 = Iv4 and Ev = Ev2)	if	3
not ((Bv and Bv2) or (Bv3 and Bv4 and (Flv >= url)))	:=	1
not (Bv and Bv2) or Bv3 or (Fxv < url) or (Bv4)	if	1
not Bv and (Bv2 or Bv3 or Bv4 or Bv5)	:=	3
not Bv and (Ev = El and Ev2 = El and Ev3 = El and Ev4 = El)	:=	2
not Bv and Bv2 and not Bv3 and Bv4 and Bv5	:=	1
not Bv or (Bv2 and not Bv3) or not Bv4 or not Bv5	:=	1

C.6 Expressions With Six Conditions

Expression	Statement	Occurs
((Bv = False and Bv2 = False and Bv3 = False and Bv4) and (Bv5 or Bv6))	=>	1
((Bv = False) and (Bv2)) or ((Bv3 = False) and (Bv4)) or ((Bv5 = False) and (Bv6))	if	1
((Bv and not Bv2) or (Bv3 and not Bv4)) and (Bv5 or Bv6)	:=	1
((Bv and not Bv2) or Bv3 or (Bv4 and Ev = El)) and not (Ev2 = El2)	:=	1
((Bv or Bv2) and (Bv3 or Bv4)) or ((Bv5 or Bv3) and (Bv2 or Bv6))	:=	1
((Bv or Iv > uil or Bv2 or Iv2 > uil) and (Iv3 = uil) and (Iv4 = uil))	if	3
((Ev = El) or (Ev2 = El)) and ((Ev3 = El2) or (Ev4 = El2) or (Ev5 = El2) or (Ev6 = El2))	:=	1
((Fxc >= Fxv) and (Fxv >= Fxc2)) or ((Fxc3 >= Fxv) and (Fxv >= Fxc4)) or ((Fxc5 >= Fxv) and (Fxv >= Fxc6)) {Fxc2 < Fxc < Fxc4 < Fxc3 < Fxc6 < Fxc5}	:=	3
((Iv = uinn) and Bv) or ((Iv2 = uinn) and Bv2) or ((Iv3 = uinn) and Bv3)	if	1
(Bv = False and Bv2 = False and Bv3 = False and Bv4 = False and Bv5 = False and Bv6)	=>	1
(Bv = False and Bv2 = False and Bv3 = False and Bv4 and (Bv5 or Bv6))	=>	2
(Bv = False and Bv2 = False and Bv3 = False and Bv4 and Bv5 and Bv6)	=>	1

Expression	Statement	Occurs
(Bv = False) or (Bv2 = False) or (Bv3 = False) or (Bv4 = False) or (Bv5 = False) or (Bv6 = False)	if	1
(Bv = True) and (Iv = Ic) and (Bv2 = False) and (Bv3 = False) and (Bv4 = False) and (Bv5 = False)	if	1
(Bv = True) or (Ev = El) or (Bv2 = True) or (Bv3 = True) or (Ev2 = El) or (Ev3 = El)	if	1
(Bv and (not (Bv2 and Bv3))) or (Bv4 and (not (Bv5 and Bv6)))	:=	1
(Bv and Bv2 and Bv3 and Bv4 = False and Bv5 = False and Bv6 = False)	=>	1
(Bv and Bv2 and Bv3 and Bv4 and Bv5 = False and Bv6 = False)	=>	1
(Bv and Bv2) or (Bv3 and Bv4) or (Bv5 and Bv6)	:=	1
(Bv and then not Bv2 and then Ev = El) or else (Bv3 and then Bv4 and then Ev2 = El)	:=	1
(Bv or Bv2 or (Ev = El) or ((Bv3 or Bv4) and (Flv >= url)))	if	1
(Bv or Bv2) and not (Bv3 or Bv4 or Bv5 or Bv6)	:=	2
(Bv or not Bv2) or ((Bv3 or Bv4) and (Bv5 or Bv6))	:=	2
(Bv) and not (Bv2 or Bv3 or Bv4 or Bv5 or Bv6)	:=	2
(Ec = El) and (Iv /= Iv2 or Iv3 /= Iv4 or Ev /= Ev2) and not (Iv5 = uil and Iv /= uil)	if	3
(Ev /= El) and (Ev /= El2) and (Ev /= El3) and (Ev /= El4) and (Ev /= El5) and (Ev /= El6)	if	2
(Ev = El and Bv) or (Ev = El2 and Bv2) or (Ev = El3 and Bv3)	if	1
(Ev = El) and then (Iv < Ic) and then not ((Iv2 = uil2) and (Rv = Rl) and (Ev2 = El2) and (Iv3 = uil2))	while	3
(Ev = El) and then (Iv < Iv2) and then (Ev2 = El) and then not ((Iv2 = uil2) and (Ev3 = El2) and (Iv3 = uil2))	while	2
(Ev = El) or Bv or Bv2 or Bv3 or (Bv4 and Bv5)	:=	1
(Fxc > Fxv) or (Fxv > Fxc2) or ((Fxc3 >= Fxv) and (Fxv >= Fxc4)) or ((Fxc5 >= Fxv) and (Fxv >= Fxc6)) {Fxc < Fxc4 < Fxc3 < Fxc6 < Fxc5 < Fxc2}	:=	3
(Ic > Iv and (Ic2 < Iv or Iv2 = Ic3)) or (Ic < Iv and (Ic3 > Iv or Iv3 = Ic4)) {Ic3 < Ic2 < Ic < Ic4 < Ic5}	:=	2
(Iv = uil) and (Iv2 /= uil) and (Iv3 = uil) and (Iv2 /= Iv4 or Iv5 /= Iv6 or Iv7 /= Iv8)	if	3
(Iv in It or Iv in It2) and not (Bv or Bv2 or Bv3 or Bv4) {It'Last < It2'First}	if	1
(not Bv and not Bv2) and (not Bv3 and not Bv4) and (not Bv5 and not Bv6)	:=	1
Bv and Bv2 and Bv3 and Bv4 and Bv5 and Bv6	if	1
Bv and not (Bv2) and not (Bv3) and not (Bv4) and not (Bv5) and Bv6	:=	2
Bv and then (Ev = El or Ev = El2 or Ev = El3 or Ev = El4) and then Flv > Flv2	if	1
Bv or ((Ev >= El) and Bv2 and Bv3 and not Bv4 and (Ev2 < El2))	:=	2
Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6	:=	5
	if	1

Expression	Statement	Occurs
Bv or Bv2 or Bv3 or not (Bv4) or Bv5 or Bv6	:=	2
Cv = Cl and then Cv2 = Cl and then Cv3 = Cl and then Cv4 = Cl and then Cv5 = Cl and then Cv6 = Cl	if	2
Iv > uil or Iv2 > uil or Iv3 > uil or Iv4 > uil or Iv5 > uil or Iv6 > uil	if	1
not (Ev = El and Bv) and not (Ev = El2 and Bv2) and not (Ev = El3 and Bv3)	if	5
not Bv and (Bv2 or Bv3) and (Bv4 or Bv5) and not Bv6	:=	2
not Bv and not Bv2 and not Bv3 and not Bv4 and not Bv5 and not Bv6	if	1

C.7 Expressions With Seven Conditions

Expression	Statement	Occurs
((Bv and Ev = El and not Bv2) or (Bv3 and Ev2 = El2)) and not (Ev3 = El3) and not (Ev3 = El4)	:=	1
((Ev = El) or (Ev = El2) or (Ev = El3) or (Ev = El4)) and ((Ev2 = El5) or ((Bv) and (Ev2 = El6)))	if	2
((Fxv < Fxc) or (Bv or Bv2 or Bv3 or Bv4)) and ((Fxv2 < Fxc2) or Bv5)	if	1
((Iv = Ic) or (Iv = Ic2) or (Iv = Ic3) or (Iv = Ic4) or (Iv = Ic5) or (Iv = Ic6) or (Iv = Ic7))	if	2
((Iv = uil) and (Ev /= El)) or (Bv and not Bv2) or Bv3 or (Bv4 and not Bv5)	:=	1
((not Bv) and (Bv2 or (Bv3 and Bv4))) or (Bv5 and (not (Bv6 or Bv7)))	:=	1
(Bv and Bv2 and (Iv = uil) and (Iv2 > uil)) or (Bv3 and Bv2 and (Iv = uil) and (Iv2 > uil)) or (Bv4 and Bv2 and (Iv = uil) and (Iv2 > uil)) or (Bv5 and Bv2 and (Iv = uil) and (Iv2 > uil))	if	1
(Bv and Bv2) or (Fxv > url) or (Bv3 or Bv4 or Bv5 or Bv6)	:=	1
(Bv or (not Bv2 and Bv3 and Bv4 and Bv5)) and not Bv6 and not Bv7	:=	2
(Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7)	if	1
(Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6) and not Bv7	:=	2
(Bv or Bv2 or Bv3 or Bv4) and Bv5 and (not (Bv6 and not (Bv7)))	:=	1
(Bv) and ((Ev = El) or (Ev = El2) or (Ev = El3) or (Iv = Ic) or (Iv = Ic2) or (Iv = Ic3))	if	4
(Ec /= Ev) and not (Bv or Bv2 or Bv3) and Bv4 and Bv5 and Bv6	:=	2
(Ev = El or Ev2 = El or Ev3 = El or Ev4 = El or Ev5 = El or Ev6 = El or Ev7 = El)	if	2
(Ev = El) and (((Ev2 < El) and Bv and Bv2) or ((Ev2 > El) and Bv3 and Bv4))	:=	2
(Ev = El) and not ((Ev2 = El2) or (Ev2 = El3) or (Ev2 = El4) or (Ev2 = El5) or (Ev2 = El6) or (Bv))	if	2
(Ev = El) or (Ev = El2) or (Ev = El3) or (Ev = El4) or (Ev = El5) or (Ev = El6) or (Ev = El7)	if	1
(Ev = Ev2) or (Ev3 = Ev2) or (Ev4 = Ev2) or (Ev5 = Ev2) or (Ev6 = Ev2) or (Ev7 = Ev2) or (Ev8 = Ev2)	if	3

Expression	Statement	Occurs
(Iv /= uil) or (Iv2 /= uil) or (Iv3 /= uil) or (Iv4 /= uil) or (Iv5 /= uil) or (Iv6 /= uil2) or (Iv7 /= uil3)	if	1
(Iv /= uil) or (Iv2 /= uil2) or (Iv3 /= uil) or (Iv4 /= uil2) or (Iv5 /= uil) or (Iv6 /= uil3) or (Iv7 /= uil4)	if	1
(Iv /= uil) or (Iv2 /= uil2) or (Iv3 /= uil2) or (Iv4 /= uil2) or (Iv5 /= uil2) or (Iv6 /= uil3) or (Iv7 /= uil4)	if	1
(Iv /= uil) or (Iv2 /= uil2) or (Iv3 /= uil3) or (Iv4 /= uil2) or (Iv5 /= uil3) or (Iv6 /= uil4) or (Iv7 /= uil5)	if	1
Bv and (not (Ev = El and Bv2) and not (Ev = El2 and Bv3) and not (Ev = El3 and Bv4))	if	4
Bv or Bv2 or Bv3 or ((not Bv4) and (Bv5 or Bv6 or Bv7))	:=	2
Bv or Bv2 or Bv3 or Bv4 or Bv5 or (Bv6 and not Bv7)	:=	2
Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7	if	2
Cv = Cl and then Cv2 = Cl and then Cv3 = Cl and then Cv4 = Cl and then Cv5 = Cl and then Cv6 = Cl and then Cv7 = Cl	if	2
Ev = El or Ev = El2 or Ev = El3 or Ev = El4 or Ev = El5 or Ev = El6 or Ev = El7	if	2
not (Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7)	:=	1
	if	1

C.8 Expressions With Eight Conditions

Expression	Statement	Occurs
((Bv = False) and (Bv2 = False) and (Bv3 = False) and (Bv4 = False) and (Bv5 = False) and (Bv6 = False) and (Bv7 = False) and (Bv8 = False))	:=	1
((Bv and not Bv2) or (Bv3 and Ev = El)) and not (Ev2 = El2) and not (Ev2 = El3) and not (Ev2 = El4) and not (Ev2 = El5)	:=	1
((Iv = uil) and (Iv2 = uil2)) or ((Iv = uil3) and (Iv2 = uil4)) or ((Iv = uil5) and (Iv2 = uil6)) or ((Iv = uil7) and (Iv2 = uil8))	if	1
(Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7 or Bv8)	if	2
(Bv xor Bv2 xor Bv3 xor Bv4 xor Bv5 xor Bv6 xor Bv7 xor Bv8)	:=	1
(Ev = El) and not (Bv) and (((Ev2 < El) and Bv2 and Bv3) or ((Ev2 > El) and Bv4 and Bv5))	:=	2
(Ev = El) or (Ev = El2) or (Ev = El3) or (Ev = El4) or (Ev = El5) or (Ev = El6) or (Ev = El7) or (Ev = El8)	if	1
(Ev = Ev2) or (Ev3 = Ev2) or (Ev4 = Ev2) or (Ev5 = Ev2) or (Ev6 = Ev2) or (Ev7 = Ev2) or (Ev8 = Ev2) or (Ev9 = Ev2)	if	12
(Fxv > Fxc) and Bv and ((Bv2 and Bv3) or (not Bv3 and Bv4)) and not Bv5 and not Bv6 and not Bv7	:=	1
(Iv /= Iv2) or (Iv3 /= Iv4) or (Bv /= Bv2) or (Bv3 /= Bv4) or (Bv5 /= Bv6)	if	1
(not Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7) and not Bv8	:=	1
Bv and (Ev /= El) and Bv2 and Bv3 and not Bv4 and not Bv5 and not Bv6 and not Bv7	if	6
Bv and Bv2 and Bv3 and Bv4 and Bv5 and Bv6 and Bv7 and Bv8	if	2

Expression	Statement	Occurs
Bv or (not (Bv2) and Bv3) or (not (Bv4) and (Bv5 or Bv6 or Bv7 or Bv8))	:=	2
Bv or Bv2 or Bv3 or Bv4 or (Bv5 and not (Ev = El)) or Bv6 or Bv7	:=	2
Bv or Bv2 or Bv3 or Bv4 or (not (Ev = El) and Bv5) or (Bv6 and Bv7)	:=	1
Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7 or Bv8	:=	4
	if	3
Ev = El or Ev2 = El or Ev3 = El or Ev4 = El or Ev5 = El or Ev6 = El or Ev7 = El or Ev8 = El	if	1

C.9 Expressions With Nine Conditions

Expression	Statement	Occurs
(((Bv and (Bv2)) or (Bv3 and (Bv4)) or (Bv5 and (Bv6))) and (not (Bv7 or Bv8 or (Bv9))))	:=	2
((Bv and not Bv2) or (Bv3 and Ev = El)) and not (Ev2 = El2) and not (Ev2 = El3) and not (Ev2 = El4) and not (Ev2 = El5) and not (Ev2 = El6)	:=	1
((Bv or Bv2) and Bv3 and not Bv4) or (Bv5 and Bv and Bv6 and Bv7 and Bv8 and Iv < Ic)	:=	1
((Ev = El) or Bv or not Bv2) and Bv3 and not Bv4 and Bv5 and Bv6 and Bv7 and Bv8	:=	1
(Bv and (Bv2 or Bv3 or Bv4 or Bv5)) or Bv6 or (not Bv and Bv7) or (Bv and Bv7 and (Iv <= uil)) or ((Iv > uil) and Bv8 and Bv7)	:=	1
(Ev = El) and then ((Iv = uil) and then ((Ev2 = El) and (Iv2 = uil) and (Rv = Rl) and (Ev3 = El2) and (Iv3 = uil) and (Iv4 = uil) and (Cav = Cal)))	if	1
(Ev = El) and then ((Iv = uil) and then ((Ev2 = El2) and (Iv2 = uil) and (Rv = Rl) and (Ev3 = El3) and (Iv3 = uil) and (Iv4 = uil) and (Cav = Cal)))	if	2
(Ev = El) or ((Ev2 = El2) and Iv not in It) or ((Ev2 = El3) and Iv not in It2) or ((Ev2 = El3) and (Ev3 = El4) and (Iv = It2'last)) or ((Ev2 = El3) and (Ev3 = El5) and (Iv >= It2'last-uil)) {It'Last < It2'First}	if	2
(Ev = Ev2) or (Ev3 = Ev2) or (Ev4 = Ev2) or (Ev5 = Ev2) or (Ev6 = Ev2) or (Ev7 = Ev2) or (Ev8 = Ev2) or (Ev9 = Ev2) or (Ev10 = Ev2)	if	9
(Fxv < url) or (Bv or Bv2 or Bv3 or Bv4) or (Bv5 or Bv6 or Bv7 or Bv8)	:=	1
(Iv /= Ic) and (Iv /= Ic2) and (Iv /= Ic3) and (Iv2 /= Ic) and (Iv2 /= Ic2) and (Iv2 /= Ic3) and (Iv3 /= Ic) and (Iv3 /= Ic2) and (Iv3 /= Ic3)	if	1
(Iv /= Iv2) or (Iv /= Iv3) or (Bv /= Bv2) or (Bv3 /= Bv4) or (Bv5 /= Bv6) or (Iav /= Iav2)	if	1
(Iv = Ic and (Iv2 /= Ic2 or Iv3 /= Ic3)) or (Iv = Ic4 and (Iv4 /= Ic2 or Iv5 /= Ic3)) or (Iv = Ic5 and (Iv2 /= Ic6 or Iv3 /= Ic2))	if	2

Expression	Statement	Occurs
(Iv = Ic) and ((Bv = True) or (Bv2 = True) or (Bv3 = True)) and (Bv4 = True) and (Bv5 = False) and (Bv6 = False) and (Bv7 = False) and (Bv8 = True)	if	1
(not Bv) and (not Bv2) and (not Bv3) and (not Bv4) and (not Bv5) and (not Bv6) and (not Bv7) and (not Bv8) and (not Bv9)	:=	2
Bv and (((not Bv2) and (not Bv3) and not Bv4 and not Bv5) or ((not Bv6) and (not Bv7) and not Bv8 and not Bv9))	if	1
Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or ((Bv7 or Bv8) and not Bv9)	:=	2
Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7 or Bv8 or Bv9	if	1
Ev = El or else (Bv and Bv2 and Bv3 and Bv4 and Bv5 and Bv6 and Bv7 and Bv8)	if	1
not (Bv and Bv2 and Bv3 and Bv4 and Bv5 and Bv6 and Bv7 and Bv8 and Bv9)	:=	1
not Bv or not Bv2 or not Bv3 or not Bv4 or not Bv5 or not Bv6 or not Bv7 or not Bv8 or not Bv9	:=	1

C.10 Expressions With Ten Conditions

Expression	Statement	Occurs
((Bv and not Bv2) or (Bv3 and Ev = El)) and not (Ev2 = El2) and not (Ev2 = El3) and not (Ev2 = El4) and not (Ev2 = El5) and not (Ev2 = El6) and not (Ev2 = El7)	:=	1
((Bv or Bv2) and Bv3 and not Bv4) or (Bv5 and Bv6 and Bv7 and Bv8 and Bv9 and Iv < Ic)	:=	1
(Iv > Iv2) and Bv and (Bv2 or Bv3 or Bv4 or Bv5) and (Bv6 or Bv7 or Bv8 or Bv9)	:=	1
Bv and (Bv2 or Bv3 or Bv4 or Bv5) and (Bv6 or (Bv7 or Bv8 or Bv9 or Bv10))	:=	1
Bv and Bv2 and (not Bv3 or not Bv4 or Bv5 or Bv6 or Bv7 or Bv8 or Bv9 or Bv10)	:=	1
Bv or (Bv2 and Bv3) or (Bv4 and Bv5) or (Bv6 and Bv7) or (Bv8 and Bv9) or Bv10	:=	2
Bv or Bv2 or (Ev = El) or ((((Ev2 = El2) and (Ev = El3)) or ((Ev2 = El4) and (Ev = El5))) and (not (Bv3 or Bv4 or Bv5)))	if	1

C.11 Expressions With 11 Conditions

Expression	Statement	Occurs
((((Bv and (not Bv2)) or ((Fxv > Fxv2) and not (Bv3 or Bv4 or (Fxv2 < url) or Bv5))) and Bv6) or Bv7) and not (Bv8 or Bv9)	:=	1
((Bv and not Bv2) or (Bv3 and Ev = El)) and not (Ev2 = El2) and not (Ev2 = El3) and not (Ev2 = El4) and not (Ev2 = El5) and not (Ev2 = El6) and not (Ev2 = El7) and not (Ev2 = El8)	:=	1
(Bv and Ev = El) and not (Ev2 = El2) and not (Ev2 = El3) and not (Ev2 = El4) and not (Ev2 = El5) and not (Ev2 = El6) and not (Ev2 = El7) and not (Ev2 = El8) and not (Ev2 = El9) and not (Ev2 = El10)	:=	1

C.12 Expressions With 12 Conditions

Expression	Statement	Occurs
(((Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6) and not Bv7) or (Bv8 and Ev = El)) and not (Ev2 = El2) and not (Ev2 = El3) and not (Ev2 = El4)	:=	1
((Bv and not Bv2) or (Bv3 and Ev = El)) and not (Ev2 = El2) and not (Ev2 = El3) and not (Ev2 = El4) and not (Ev2 = El5) and not (Ev2 = El6) and not (Ev2 = El7) and not (Ev2 = El8) and not (Ev2 = El9)	:=	1
(not Bv) and (not Bv2) and (not Bv3) and (not Bv4) and (not Bv5) and (not Bv6) and (not Bv7) and (not Bv8) and (not Bv9) and (not Bv10) and (not Bv11) and (not Bv12)	:=	2

C.13 Expressions With 13 Conditions

Expression	Statement	Occurs
((Ev = El) and (Ev2 = El2)) or ((Ev = El3) and (Ev2 = El4)) or ((Ev = El5) and (Ev2 = El6)) or ((Ev = El7) and (Ev2 = El8)) or ((Ev = El9) and (Ev2 = El10)) or ((Ev = El11) and (Ev2 = El12)) or (Ev3 = Ev2)	if	2
(Bv and Bv2 and Bv3 and Bv4 and not Bc) or (Bv and Bv5 and Bv6 and Bv7 and not Bc) or (Bv8 and Bv9 and Bv10 and Bv4 and Bc) or (Bv8 and Bv11 and Bv12 and Bv7 and Bc)	:=	2
(Ev = Ev2) or (Ev3 = Ev2) or (Ev4 = Ev2) or (Ev5 = Ev2) or (Ev6 = Ev2) or (Ev7 = Ev2) or (Ev8 = Ev2) or (Ev9 = Ev2) or (Ev10 = Ev2) or (Ev11 = Ev2) or (Ev12 = Ev2) or (Ev13 = Ev2) or (Ev14 = Ev2)	if	3
Bv or Bv2 or (Bv3 or Bv4 or Bv5 or Bv6) or (Bv7 or Bv8 or Bv9 or Bv10) or Bv11 or not Bv12 or not Bv13	:=	1

Expression	Statement	Occurs
not (Bv) and (((Bv2 and Bv3) or (Bv2 and Bv4) or (Bv5 and Bv6) or (Bv5 and Bv7)) or ((Bv8 and Bv9) or (Bv2 and Bv10) or (Bv2 and Bv11) or (Bv5 and Bv12) or (Bv5 and Bv13)))	:=	1
not (Bv) and (((Bv2 and Bv3) or (Bv2 and Bv4) or (Bv5 and Bv6) or (Bv5 and Bv7)) or ((Bv8 and Bv9) or (Bv2 and Bv10) or (Bv2 and Bv11) or (Bv5 and Bv12) or (Bv5 and Bv13)))	:=	1
not (Bv) and ((Bv2 and (Iv /= Iv2)) or (Bv3 and (Iv3 /= Iv4)) or (Bv2 and (Iv5 /= Iv6)) or (Bv2 and (Iv7 /= Iv8)) or (Bv4 and (Iv9 /= Iv10)) or (Bv5 and (Iv11 /= Iv12)) or (Bv4 and (Iv13 /= Iv14)) or (Bv4 and (Iv15 /= Iv16)))	:=	1
not (Bv) and ((Bv2 and (Iv /= Iv2)) or (Bv3 and (Iv3 /= Iv4)) or (Bv2 and (Iv5 /= Iv6)) or (Bv2 and (Iv7 /= Iv8)) or (Bv4 and (Iv9 /= Iv10)) or (Bv5 and (Iv11 /= Iv12)) or (Bv4 and (Iv13 /= Iv14)) or (Bv4 and (Iv15 /= Iv16)))	:=	1

C.14 Expressions With 14 Conditions

Expression	Statement	Occurs
((Ev = El) and (not Bv)) or ((Ev = El2) and (not Bv2)) or ((Ev = El3) and (not Bv3)) or ((Ev = El4) and (not Bv4)) or ((Ev = El5) and (not Bv5)) or ((Ev = El6) and (not Bv6)) or ((Ev = El7) and (not Bv7))	:=	1
(Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7 or Bv8 or Bv9 or Bv10 or Bv11 or Bv12 or Bv13) and Bv14	:=	2
Bv and Iv > Iv2 and (Bv2 or Bv3 or Bv4 or Bv5) and ((Bv6 or Bv7 or Bv8 or Bv9) or (Bv10 or Bv11 or Bv12 or Bv13))	:=	1
Iv = Iv2 or Iv = Iv3 or Iv = Iv4 or Iv = Iv5 or Iv = Iv6 or Iv = Iv7 or Iv = Iv8 or Iv = Iv9 or Iv = Iv10 or Iv = Iv11 or Iv = Iv12 or Iv = Iv13 or Iv = Iv14 or Iv = Iv15	if	4
not (Bv or Bv2) or (Bv3 xor Bv4) or (Bv3 and ((Bv5 xor Bv6) or (Bv7 xor Bv8) or (Bv9 xor Bv10) or (Bv11 xor Bv12) or ((Iv = uil) and (Ev = El))))	:=	1

C.15 Expressions With 15 Conditions

Expression	Statement	Occurs
(((Bv and (Bv2)) or (Bv3 and (Bv4)) or (Bv5 and (Bv6)) or (Bv7 and (Bv8)) or (Bv9 and (Bv10)) or (Bv11 and (Bv12))) and (not (Bv13 or Bv14 or (Bv15))))	:=	2
(Ev = Ec) and (Ev2 = Ec) and (Ev3 = Ec) and (Ev4 = Ec) and (Ev5 = Ec) and (Ev6 = Ec) and (Ev7 = Ec) and (Ev8 = Ec) and (Ev9 = Ec) and (Ev10 = Ec) and (Ev11 = Ec) and (Ev12 = Ec) and (Ev13 = Ec) and (Ev14 = Ec) and (Ev15 = Ec)	if	1
(Ev = Ev2) or (Ev3 = Ev2) or (Ev4 = Ev2) or (Ev5 = Ev2) or (Ev6 = Ev2) or (Ev7 = Ev2) or (Ev8 = Ev2) or (Ev9 = Ev2) or (Ev10 = Ev2) or (Ev11 = Ev2) or (Ev12 = Ev2) or (Ev13 = Ev2) or (Ev14 = Ev2) or (Ev15 = Ev2) or (Ev16 = Ev2)	if	3
Iv = Iv2 or Iv = Iv3 or Iv = Iv4 or Iv = Iv5 or Iv = Iv6 or Iv = Iv7 or Iv = Iv8 or Iv = Iv9 or Iv = Iv10 or Iv = Iv11 or Iv = Iv12 or Iv = Iv13 or Iv = Iv14 or Iv = Iv15 or Iv = Iv16	if	1

C.16 Expressions With 16 Conditions

Expression	Statement	Occurs
((Ev = El) and (Ev2 = El2)) or ((Ev3 = El) and (Ev4 = El2)) or ((Ev5 = El) and (Ev6 = El2)) or ((Ev7 = El) and (Ev8 = El2)) or ((Ev9 = El) and (Ev10 = El2)) or ((Ev11 = El) and (Ev12 = El2)) or ((Ev13 = El) and (Ev14 = El2)) or ((Ev15 = El) and (Ev16 = El2))	:=	1
(Bv and Bv2 and (Iv = uil) and Iv2 > uil) or (Bv3 and Bv4 and (Iv3 = uil) and Iv4 > uil) or (Bv5 and Bv6 and (Iv5 = uil) and Iv6 > uil) or (Bv7 and Bv8 and (Iv7 = uil) and Iv8 > uil)	if	21
Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7 or Bv8 or Bv9 or Bv10 or Bv11 or Bv12 or Bv13 or Bv14 or Bv15 or Bv16	:=	2
	if	2
Bv or else Bav /= Bac or else Bv2 or else Bav2 /= Bac or else Bv3 or else Bav3 /= Bac or else Bv4 or else Bav4 /= Bac or else Bv5 or else Bav5 /= Bac or else Bv6 or else Bav6 /= Bac or else Bv7 or else Bav7 /= Bac or else Bv8 or else Bav8 /= Bac	if	1
not (Bv xor Bv2 xor Bv3 xor Bv4 xor Bv5 xor Bv6 xor Bv7 xor Bv8 xor Bv9 xor Bv10 xor Bv11 xor Bv12 xor Bv13 xor Bv14 xor Bv15 xor Bv16)	:=	2

C.17 Expressions With 17 Conditions

Expression	Statement	Occurs
(Bv or Bv2 or (Bv3 and Ev = El) or Bv4 or Bv5 or Bv6 or Bv7 or Bv8 or Bv9 or Bv10 or Bv11 or Bv12 or Bv13 or Bv14 or (Bv15 and Ev2 = El2))	:=	1
not (Bv and Bv2 and Bv3 and Bv4 and Bv5 and Bv6 and Bv7 and Bv8 and Bv9 and Bv10 and Bv11 and Bv12 and Bv13 and Bv14 and Bv15 and Bv16 and Bv17)	:=	2

C.18 Expressions With 18 Conditions

Expression	Statement	Occurs
(Ev = Ev2) or (Ev3 = Ev2) or (Ev4 = Ev2) or (Ev5 = Ev2) or (Ev6 = Ev2) or (Ev7 = Ev2) or (Ev8 = Ev2) or (Ev9 = Ev2) or (Ev10 = Ev2) or (Ev11 = Ev2) or (Ev12 = Ev2) or (Ev13 = Ev2) or (Ev14 = Ev2) or (Ev15 = Ev2) or (Ev16 = Ev2) or (Ev17 = Ev2) or (Ev18 = Ev2) or (Ev19 = Ev2)	if	3

C.19 Expressions With 19 Conditions

Expression	Statement	Occurs
(Bv = True) and (Bv2 = True) and (Bv3 = True) and (Bv4 = True) and (Iv /= Ic) and (Iv /= Ic2) and (Iv /= Ic3) and (Iv2 /= Ic) and (Iv2 /= Ic2) and (Iv2 /= Ic3) and (Iv3 /= Ic) and (Iv3 /= Ic2) and (Iv3 /= Ic3) and (Bv5 = True) and (Bv6 = True) and (Bv7 = True) and (Bv8 = False) and (Bv9 = False) and (Bv10 = False)	if	1

C.20 Expressions With 21 Conditions

Expression	Statement	Occurs
not (Bv) and Bv2 and Bv3 and not (Bv4 or Bv5 or Bv6 or Bv7 or Bv8 or Bv9 or Bv10 or Bv11 or Bv12 or Bv13 or Bv14 or Bv15 or Bv16 or Bv17 or Bv18 or Bv19 or Bv20 or Bv21)	:=	1

C.21 Expressions With 25 Conditions

Expression	Statement	Occurs
Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7 or Bv8 or Bv9 or Bv10 or Bv11 or Bv12 or Bv13 or Bv14 or Bv15 or Bv16 or Bv17 or Bv18 or Bv19 or Bv20 or Bv21 or Bv22 or Bv23 or Bv24 or Bv25	:=	1

C.22 Expressions With 28 Conditions

Expression	Statement	Occurs
Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7 or Bv8 or Bv9 or Bv10 or Bv11 or Bv12 or Bv13 or Bv14 or Bv15 or Bv16 or Bv17 or Bv18 or Bv19 or Bv20 or Bv21 or Bv22 or Bv23 or Bv24 or Bv25 or Bv26 or Bv27 or Bv28	:=	1

C.23 Expressions With 33 Conditions

Expression	Statement	Occurs
not Bv or not Bv2 or not Bv3 or not Bv4 or not Bv5 or not Bv6 or not Bv7 or not Bv8 or not Bv9 or (not Bv10 and not Bv11) or ((not Bv12 or not Bv13) and not Bv14) or not Bv15 or not Bv16 or not Bv17 or not Bv18 or not Bv19 or not Bv20 or not Bv21 or not Bv22 or not Bv23 or not Bv24 or not Bv25 or not Bv26 or not Bv27 or not Bv28 or not Bv29 or not Bv30 or not Bv31 or not Bv32 or not Bv33	:=	1

C.24 Expressions With 49 Conditions

Expression	Statement	Occurs
Bv or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7 or Bv8 or Bv9 or Bv10 or Bv11 or Bv12 or Bv13 or Bv14 or Bv15 or Bv16 or Bv17 or Bv18 or Bv19 or Bv20 or Bv21 or Bv22 or Bv23 or Bv24 or Bv25 or Bv26 or Bv27 or Bv28 or Bv29 or Bv30 or Bv31 or Bv32 or Bv33 or Bv34 or Bv35 or Bv36 or Bv37 or Bv38 or Bv39 or Bv40 or Bv41 or Bv42 or Bv43 or Bv44 or Bv45 or Bv46 or Ev = El or Ev2 = El or Ev3 = El	if	1

C.25 Expressions With 76 Conditions

Expression	Statement	Occurs
Bv or (Ev /= El) or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7 or Bv8 or Bv9 or Bv10 or Bv11 or Bv12 or Bv13 or Bv14 or Bv15 or Bv16 or Bv17 or Bv18 or Bv19 or Bv20 or Bv21 or Bv22 or Bv23 or Bv24 or Bv25 or Bv26 or Bv27 or Bv28 or Bv29 or Bv30 or Bv31 or Bv32 or Bv33 or Bv34 or Bv35 or Bv36 or Bv37 or Bv38 or Bv39 or Bv40 or Bv41 or Bv42 or Bv43 or Bv44 or Bv45 or Bv46 or Bv47 or Bv48 or Bv49 or Bv50 or Bv51 or (Ev2 = El2) or ((Ev3 = El2) and (Sav /= Sac)) or Bv52 or Bv53 or Bv54 or Bv55 or Bv56 or Bv57 or Bv58 or Bv59 or Bv60 or Bv61 or Bv62 or Bv63 or Bv64 or Bv65 or Ev4 /= El3 or Ev5 = El4 or Ev6 = El4 or Ev7 = El4 or Ev8 = El4 or Ev9 = El4 or Ev10 = El4	if	1

APPENDIX D—SOFTWARE FAULT INJECTION/MUTATION

This appendix documents the details of an investigation (substudy) conducted into Software Fault Injection/Mutation. Software Mutation (Fault Injection) is a structural coverage criterion used to assess test data set adequacy. It has been used to assess the adequacy of other structural coverage techniques, and is purported to subsume the other techniques (MCDC being one of them). This appendix provides an overview of Software Mutation (Fault Injection), including some of the fundamental assumptions made by the theory. A set of mutation operators is then defined. The performance of three forms of MCDC against the mutants generated by these operators is then studied, and compared against the Probability of Error Detection model developed earlier in this study. Some of the underlying assumptions of Software Mutation Theory are then addressed. Conclusions from the data are drawn, and limitations of the study methodology are identified.

D.1 Introduction

The purpose of this substudy effort is to compare and contrast the efficacy of three forms of MCDC under the rules of mutation [1], and compare this with the probability of error detection model developed earlier in this study. The intention is to show that the model is more accurate, and therefore represents a truer picture of how MCDC will behave in the discovery of errors in software logic. It is felt that the model is more accurate because mutation can only address a subset of what the model addresses. This will provide further data for the rational choice of which form of MCDC should be required for high-integrity or safety-critical software.

D.1.1 Mutation Overview

Software mutation is a test data adequacy technique which injects simple faults into a program, and then checks to see if the test data set for that program is sensitive to the error (if one exists) [1,2,3]. If the mutant program produces output that is different from the original program on at least one of the tests in the test data set, and the original program produces correct output, then the mutant is said to be killed. If all nonequivalent mutants (i.e., mutants that are not equivalent to the original program) are killed by the test data set, then the test data set is said to be mutation-adequate.

Software mutation is founded on two premises: the *competent programmer hypothesis* and the *coupling effect* [1]. The competent programmer hypothesis states that most faults in an implemented program are small discrepancies between it and the correct program (i.e., the correct and incorrect programs are very close to each other). The coupling effect asserts that test data that detects small faults in a program are sensitive enough to detect complex faults, since the complex faults are a combination of simple faults. Software mutation uses these premises as rationale to inject single syntactic changes into a program. These are known as single change mutants, and are formed according to a set of rules, known as mutation operators. Multiple change mutants are not injected as the coupling effect says they are not needed.

D.1.2 Study Parameters

The studies in this report were carried out exhaustively against two-, three-, and four-condition expressions. A partial analysis was conducted against five-condition expressions due to the lack of resources to complete this magnitude of work. There was no need to conduct any studies against one-condition expressions as MCDC requires exhaustive testing in this case, and therefore will always discover the errors that would be introduced by mutation. For the two-condition expression case an exhaustive analysis was conducted against all expression forms since the number of expressions (24) involved was small enough to handle.

Exhaustive analysis was not the case with the three-, four-, and five-condition expressions. For the higher-condition expression analysis a subset of abstract expressions was used that would reveal all the different performance results needed by this study. In essence, all that was necessary were the expressions from unique forms of expression trees. The details for this are discussed within the appropriate sections of the report. For the significant expression trees, all expressions were generated and analyzed.

The three-, four-, and five-condition expressions are too numerous to list, but the expression trees used to generate the expressions are defined. For the three-condition expressions, only a single tree is needed. For four-conditions, two trees are needed. For five-conditions, three trees are needed. Because of the number of expressions involved at the five-condition level, only those trees dominated by the (AND, ≤, ≥) operator were studied. These expressions were chosen because for two-, three-, and four-condition expressions, those dominated by the (AND, ≤, ≥) operators had results very close to the average.

The studies in this report were carried out against expressions which involved the classic Boolean operators (NOT, AND, OR, XOR) as well as the relational operators (=, ≠, >, ≥, <, ≤) operating on Booleans. There was insufficient time to include the relational operators operating on non-Booleans in this study. The methodology used in this study precludes the inclusion of non-Booleans. Study of this area should take place in another study.

D.1.3 Section Overview

In section D.2, the mutation operators are designed that were used in this study. The design of these operators is based on designs that have been done by others [2, 4, and 5]. The design strategy employed in this study attempted to address a number of problems with mutation that were identified by earlier designers of mutant operators. In particular, the number of equivalent mutants that would be generated was minimized. This was done in order to simplify the analyses conducted as part of the study.

Section D.2.3.4 addresses the issue of just how much of the Boolean function space is mutation of expressions able to cover or span. By this we mean how many Boolean functions are mutants able to represent. An answer to this is needed since the probability of error detection model developed earlier in this study addresses the entire Boolean function space at all condition levels. Just how small a subspace mutation is able to span will affect the results of the probability of mutation detection. The term span is used in this report so as to avoid overloading the terms cover and coverage as used in testing.

Section D.3 investigates the efficacy of MCDC against mutation, meaning just how good is MCDC at killing mutants. Based on the probability of error detection model it was expected that MCDC would kill the vast majority of mutants. However, if mutants congregate in those functions to which MCDC is insensitive (i.e., unable to discern) or never congregate there, then the results will be different. Analysis of this area is needed to better understand just where to place the mutation in importance in the study of MCDC itself, and the different forms of MCDC being considered.

Section D.4 addresses the issue of the coupling effect [1, 6, and 7]. As was pointed out earlier, this is one of the fundamental assumptions behind mutation theory. Though this study is rather limited, and addressing this issue was not part of the original plan, some data were generated from a brief visitation of this issue. This is an issue that requires further study.

Section D.5 looks at the claim made that mutation subsumes MCDC [8]. This is an issue that deserves careful consideration. If mutation is indeed able to subsume MCDC, and it is more cost-effective than MCDC, then MCDC should be dropped and mutation adopted. Again, this is a rather limited study, and this issue was not part of the original plan. However, some data were generated from the brief visitation of this issue. This is an issue that also requires further study.

In section D.6, some conclusions can be drawn from this study. Again, it should be remembered that this substudy is very limited, so these conclusions will not be the final word.

Section D.7 contains a brief example showing the complexity that would be introduced if one of the previously defined mutation operators was included into the set. The methodology of this study makes the analysis of this particular mutation extremely difficult, if not impossible. If it were decided to study this mutation, the methodology of this study would have to be expanded to include more than just the logic expressions from airborne software.

D.2 Design of Mutation Operators

In order to compare MCDC either with or against mutation, it was necessary to design a set of mutation operators applicable to logic expressions in the Ada programming language. A number of references were consulted [2, 4, and 5] to assist with this design. From these references, a number of goals for the mutation operators were derived. Unfortunately, not all of these goals were achieved (totally). The details are provided in the appropriate sections dealing with the goals, and summarized in the conclusions section (section D.6).

D.2.1 Goals

It was desired to design a set of mutation operators so that the same number of mutants would be generated for any expression with the same number of occurrences of conditions (e.g., *N* mutants for every expression with *M* conditions). This is desired because the probability of error detection model always considered all nonequivalent functions as possible erroneous implementations. If different numbers of mutants are generated for different expressions, then the comparison results might be skewed. Fortunately, this goal was achieved.

Perhaps more important than the same number of mutants, it was desired that the same number of Boolean functions would be represented by these mutant expressions. This is desired because the probability of error detection model always considers the same number of nonequivalent Boolean functions for all expressions of the same number of conditions (i.e., $_2 2^N{}_{-1}$ nonequivalent Boolean functions for all expressions of *N* conditions). If different numbers of mutant functions are generated for different expressions, this might skew the comparison results since not all expressions would be treated equally in the MCDC probability of mutation detection analysis. This is what happened as is documented in section D.2.3.4 (span) and section D.3 (skew).

It was also desired that the mutation operators would generate a minimum of redundant mutations (i.e., mutants that represent the same Boolean function). No redundant mutants would ensure that the mutants covered as much of the Boolean function space as possible. This is desired because the probability of error detection model always covers the entire Boolean function space. Minimal redundancy would help to put the MCDC probability of mutation detection analysis and MCDC probability of mutation function detection analysis on a more level footing. If different Boolean functions are spanned by differing numbers of mutants, then the analysis results will be skewed. This is what happened as is documented in section D.2.3.4 (spanning) and section D.3 (efficacy).

It was especially desirable to generate no mutants equivalent to the original expression as this has been identified in previous mutation studies as a major problem. This goal was not accomplished.

And finally, it would be nice if the mutation operators would generate mutants of the same Boolean functions for every equivalent expression (e.g., *A and B* versus *not (not A or not B)*). This again would put the comparison between the probability of error detection and the probability of mutation detection on a more equal footing. This goal was not accomplished.

D.2.2 Approach

Based on references 2, 4, and 5, and on our derived goals, the following approach was used to design the mutant operators. It was decided to use expression trees, and to introduce one change in the expression tree in order to form mutants. This allowed for a subset of mutation operators over what has been published in the literature. It also allowed for the minimization of the redundancy of mutants present in the other designs for mutant operators. This minimization of redundant mutants was more for efficiency reasons than for any other as the analyses conducted during this study were more against the corresponding Boolean functions represented by the mutants than against the mutants themselves.

Since expression trees were used and only expressions composed of Boolean operators and variables (operands) were looked at, only four types of nodes were used to construct the expression trees. Figure D-1 shows the expression trees used in this study for two and three-condition expressions. The tree on the left is for expressions with one binary Boolean operator and two conditions, each with a single occurrence. This form of tree turns out to be sufficient to describe all of the nondegenerate functions (i.e., those Boolean functions which are a function of

both conditions). The tree on the right is for expressions with two binary Boolean operators and three conditions. This form of tree turns out to be sufficient to describe all of the nondegenerate functions of three conditions with a single condition each. The legend in figure D-1 identifies the four types of nodes used in these trees. Details for each node type are discussed following figure D-1.

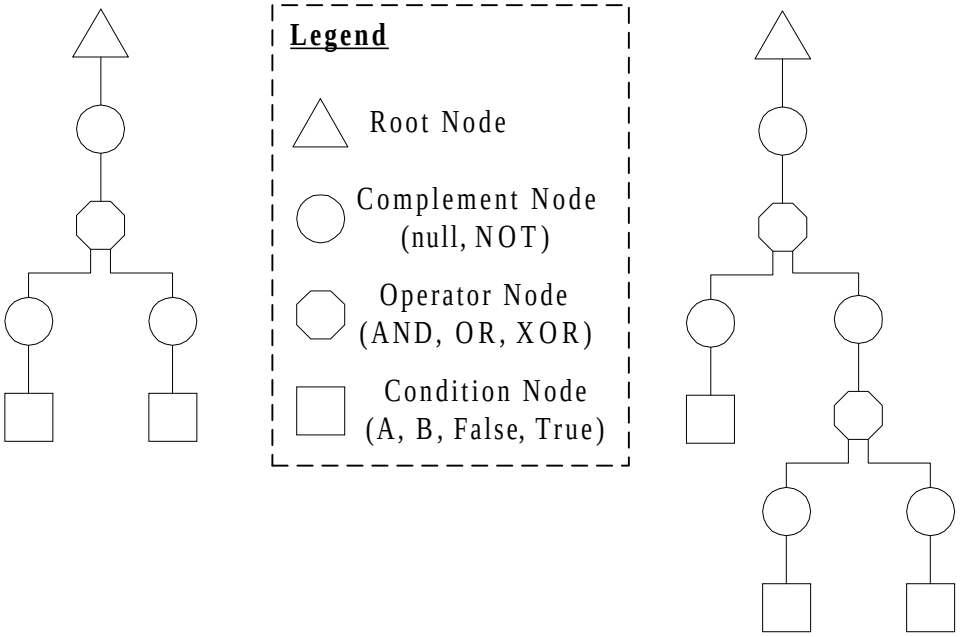


FIGURE D-1. TWO- AND THREE-CONDITION EXPRESSION TREES FOR MUTATION

The use of four nodes comes from Halstead's analysis of language. In his work, he proved that any language could be decomposed into two kinds of lexemes: operators and operands. Operands correspond to the conditions of a logic expression and have a corresponding node in the tree. Operators come in two forms: unary and binary. A separate node was used for each type. The complement node is for the unary Boolean operator (NOT). The operator node is for the binary Boolean operators (AND, OR, XOR) and the binary relational operators (=, ≠, <, ≤, >, ≥) operating on Booleans. The root node is there for implementation purposes only. Each of these nodes is discussed in the following subsections.

Note that there is another tree form that can be used with three conditions. That form is symmetrical to the tree previously used, where the symmetrical tree would have the subordinate binary operator on the left-hand side (LHS) of the upper binary operator. The two tree forms are given in figure D-2. It turns out that every expression that is expressible with three conditions with a single occurrence each, can be expressed with either tree.

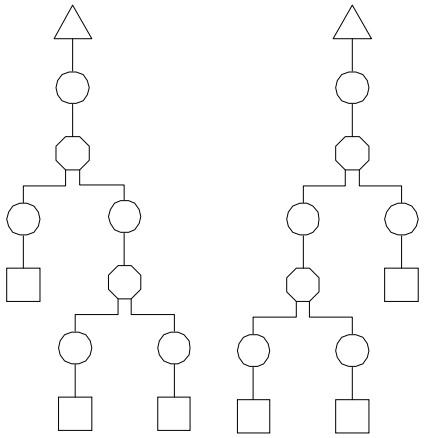


FIGURE D-2. TWO FORMS OF THREE-CONDITION EXPRESSION TREE

For four-condition expressions, there are 11 different trees that can be generated. These 11 trees group into two families of symmetrical trees. Since symmetrical trees do not add any significant features to the mutation analysis, only one of the trees from each family needs to be used. The two tree forms used in the four-condition analysis are given in figure D-3.

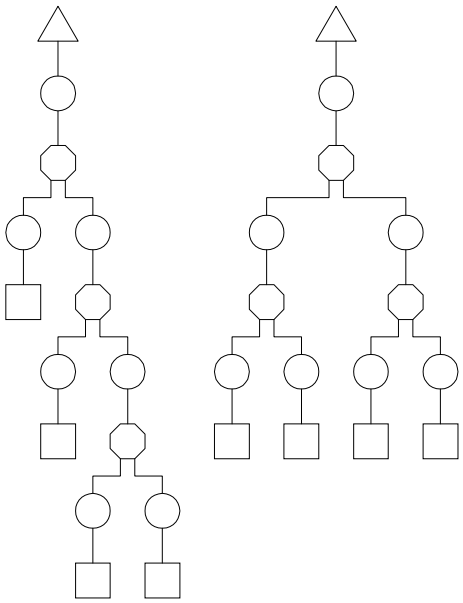


FIGURE D-3. TWO FORMS OF FOUR-CONDITION EXPRESSION TREE

For five-condition expressions, there are 45 different trees that can be generated. These 45 trees group into 3 families of symmetrical trees. Again, only one tree from each family needs to be used. The three tree forms used in the five-condition analysis are given in figure D-4.

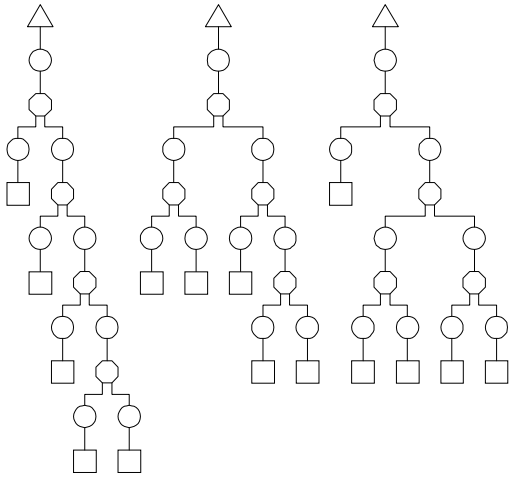


FIGURE D-4. THREE FORMS OF FIVE-CONDITION EXPRESSION TREE

D.2.2.1 Root Node

The first node type, which is always required in tree structures, is the root node. This node is denoted by a triangle and is there for the mathematics and the data structure of the programs written to assist with the analyses. This node has no printable component in any of the reports, and has no mutation operators associated with it.

D.2.2.2 Condition Node

The second node type that is needed is one for the conditions of the expression. This node is denoted by a square and corresponds to the leaves of the tree. This node can hold the names of any of the variables and constants that are being analyzed, and therefore has many different printable components. This node has one mutation operator associated with it.

In the published literature concerning the design of mutants, there are variable replacements and constant replacements of many varieties. The variety has to do with the underlying structure of the object (scalar, array, record, etc.) and whether they are constant or variable, with a separate mutation operator provided for each different structure and access class. In this study, the abstract expressions consist of only Boolean objects. If the previous mutation designers were followed, two mutation operators would have to be provided: one for variables and one for constants.

For the purposes of this study, these were combined into one mutation: condition replacement (or replace logical condition (RLC)). This mutation operator replaces the name of the current condition in the tree with each of the other names in the condition set for that expression. For two-condition expressions, the condition set consists of the names *(A, B, False, True)*. For three-condition expressions, the condition set consists of the names *(A, B, C, False, True)*. For four-condition expressions, the condition set consists of the names *(A, B, C, D, False, True)*. For five-condition expressions, the condition set consists of the names *(A, B, C, D, E, False, True)*.

For the variable replacements, only the conditions that should be present within the expression are used. The variable names are (*A, B*) for two-condition expressions, (*A, B, C*) for three-condition expressions, (*A, B, C, D*) for four-condition expressions, and (*A, B, C, D, E*) for five-condition expressions. For many reasons, the effect of including every variable visible to the expression was not studied. First, only the logic expressions themselves were used in this study. Their extraction from the airborne software did not include the context data necessary to look into this issue. Secondly, this would have increased the complexity of this study enormously. A brief look at the issues raised by including this into the study is presented in section D.7. Thirdly, in the search for "sufficient" mutation operators [9,10], it has been found that these mutations were not very productive, though they were numerous. Finally, the probability of error detection model developed in an earlier phase of this study, assumes that the expression is to be composed of the conditions (A, B) as opposed to (A, C) or (A, B, C) is generally understood. Perhaps a follow-on study should look at the performance of MCDC against expressions located in real source code.

For the constant replacements, only the two predefined Boolean constants (*False*, *True*) were used. This seemed a reasonable thing to do since any other constant visible to the expression would need to hold one of those values, so including them would have only generated redundant mutants.

As mentioned earlier, this node has four printable forms for two-condition expressions, five printable forms for three-condition expressions, six printable forms for four-condition expressions and seven printable forms for five-condition expressions. This mutation operator therefore generates three mutants for every condition node in a two-condition expression tree, which results in six total mutants. This mutation operator generates four mutants for every condition node in a three-condition expression tree, which results in twelve total mutants. This mutation operator generates five mutants for every condition node in a four-condition expression tree, which results in twenty total mutants. This mutation operator generates six mutants for every condition node in a five-condition expression tree, which results in thirty total mutants.

D.2.2.3 Operator (Halstead) Nodes

Finally, a node type is needed to hold the Boolean operators. Boolean operators come in two forms: unary (NOT) and binary (AND, OR, XOR, $=$, $\neq$, $<$, $\leq$, $>$, $\geq$). It was decided, for convenience, to have two separate node types, one for each form of Boolean operator. Each is discussed in the following subsections.

It was decided for the purposes of this study that the inclusion of necessary parenthesis for complement and operator mutations would be considered a part of the mutation (even though it adds three tokens). Consequently, there is no difference in the mutation of parenthesized versus nonparenthesized expressions. This means that the expressions *A and B and C* and *(A and B) and C* were mutated the same way. This was seen as desirable as it provides a uniform set of mutants over expressions, regardless of their initial form. This is consistent with the approach taken in the development of the probability of error detection model developed in a previous phase of this study.

D.2.2.3.1 Complement Node

The first of the operator node types is the complement node. This node holds whether or not a NOT operator is present in that place of the expression, and therefore has two printable forms: a blank, which was denoted as a *null* in figure D-1, and the NOT token itself. This node has one mutation operator associated with it.

The complementation node comes in two forms, as complementation applies to both operators and conditions (leaf operands). When the complement is added for an operator, then a corresponding set of parenthesis must be added to enclose the subexpression being complemented. For the purposes of this study, the addition of parenthesis to support operator complementation was considered part of the mutation, and therefore still counted as a single change (even though it adds three tokens to the source code).

This led to two different mutation operators: (un)complement operator (complement logical operator, (CLO)), and (un)complement condition (complement logical condition, (CLC)). These operators will either add the complement if it is not present, or remove it if it is. These operators will each generate one mutant per complement node in an expression tree. This results in three total mutants for two-condition expressions, five total mutants for three-condition expressions, seven total mutants for four-condition expressions, and nine total mutants for five-condition expressions.

D.2.2.3.2 Operator Node

The final node type is the operator node that holds the names of the binary Boolean operators. This node has nine printable components, one for each operator. This node has one mutation operator associated with it: logical operator replacement (replace logical operator (RLO)). This operator will replace the operator present with each of the other eight. This operator will generate eight mutants per operator node. This results in 8 mutants total for two-condition expressions, 16 mutants total for three-condition expressions, 24 mutants total for four-condition expressions, and 32 mutants total for five-condition expressions.

For the purposes of this study, the addition of parenthesis to support operator replacement was considered part of the mutation, analogous to the complement operator (CLO). These parenthesis are necessary in the Ada programming language to separate dissimilar binary Boolean operators (AND, OR, XOR). These parentheses are not necessary in other languages for which mutation operators have been defined.

Notice that in this substudy, the short-circuit forms of the binary Boolean operators (AND-THEN, OR-ELSE) were not included in the operator set. This is because there is no way for MCDC to distinguish between a short-circuit operator and its nonshort-circuited counterpart (i.e., (AND, AND-THEN) are indistinguishable as well as (OR, OR-ELSE)). To have included these operators into the set would have caused the generation of equivalent mutants to the original expression. As was mentioned previously, avoiding this was one of the goals for the design of the mutation operators.

D.2.3 Results

In this study, four mutant operators were used for logic expressions. Given the structure of the abstract expressions, these correlate well with the operators designed by others for Ada [2,5], C [4], and ForTran. Simplification was achieved by not requiring different operators for scalars, arrays, records, pointers, etc. The correlation between the operators designed for this study and those designed by others is given in table D-1.

In table D-1, the mutation operator names used by their designers are given along with their mnemonics in parenthesis. As can be seen from an examination of the table, all of the mutations defined by others have been covered. Note however, that certain mutation operators in this study cover multiple mutation operators from other studies. This occurred with the operators RLO (covering both binary Boolean operators and relational operators) and RLC (covering both variables and constants).

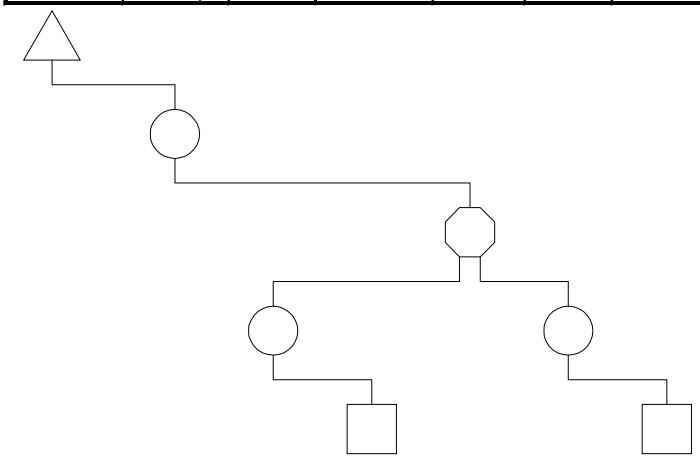
TABLE D-1. COMPARISON OF LOGIC MUTATION OPERATORS

Study	Ada	C	ForTran
(un)Complement Logical Operator (CLO)		Logical Negation (OLNG) Logical Context Negation (OCNG)	
(un)Complement Logical Condition (CLC)		Logical Negation (OLNG)	
Replace Logical Operator (RLO)	Logical Operator Replacement (ELR)	Logical to Logical Operator (OLNN)	Logical Connector Replacement (LCR)
		Logical to Relational Operator (OLRN)	
Replace Logical Condition (RLC)	Variable Replaced by a Variable (OVV)	Scalar Variable Reference Replacement (Vsrr)	Scalar Variable Replacement (SVR)
	Variable Replaced by a Constant (OVC)	Constant for Scalar Replacement (Ccsr)	Constant for Scalar Variable Replacement (CSR)

Table D-2 illustrates each of the mutations designed in this study for a two-condition expression. In the table, there is a column corresponding to each of the nodes in the expression tree, except for the root node. Below the table is a graphic showing how the columns correspond to the nodes of the expression tree. The first column identifies which mutation operator is applied to the base expression (*A and B*). The actual mutation is identified by the italicized text.

TABLE D-2. MUTATION OF TWO-CONDITION EXPRESSION

			A	and		B	
RLC			*B*	and		B	
RLC			*False*	and		B	
RLC			*True*	and		B	
CLC		*not*	A	and		B	
RLO			A	*or*		B	
RLO			A	*xor*		B	
RLO			A	=		B	
RLO			A	/=		B	
RLO			A	<		B	
RLO			A	<=		B	
RLO			A	>		B	
RLO			A	>=		B	
RLC			A	and		*A*	
RLC			A	and		*False*	
RLC			A	and		*True*	
CLC			A	and	*not*	B	
CLO	*not (*		A	and		B	*)*



D.2.3.1 Goal: Same Number of Mutants for Every *N*-Condition Expression

One of the goals in the design of our mutation operators was that the same number of mutants would be generated for every expression at the same (i.e., *N*) condition level. As was discussed in section D.2.2.3, each of the mutation operators generates a constant number of mutants for each kind of node; therefore, this goal was accomplished. Table D-3 presents the data for this analysis.

In table D-3, the mutation operator is identified in the first column. Data concerning that operator is presented in the same row as the name appears. The second column identifies the number of mutants that will be generated for each node that the mutation operator applies to.

TABLE D-3. COMPARISON OF LOGIC MUTATION OPERATORS

Mutation Operator	No. Mutants/Node	2-Condition Mutants	3-Condition Mutants	4-Condition Mutants	5-Condition Mutants
CLO	1	1	2	3	4
CLC	1	2	3	4	5
RLO	8	8	16	24	32
RLC	3 / 4 / 5 / 6	6	12	20	30
Total		17	33	51	71

Notice that the only operator that is dependent on the number of conditions is the Replace Logical Condition (RLC) operator. For two-condition expressions, this operator will generate three mutants per node (e.g., *B, False, True* for *A*). For three-condition expressions it will generate four mutants per node (e.g., *A, C, False, True* for *B*). For four-condition expressions, it will generate five mutants per node (e.g., *A, B, D, False, True* for *C*). For five-condition expressions, it will generate six mutants per node (e.g., *A, B, C, E, False, True* for *D*). The third through fifth columns identify how many total mutants will be generated by that mutation operator for two-condition expressions, three-condition expressions, four-condition expressions, and five-condition expressions respectively. The final row totals the total number of mutants for two-, three-, four- and five-condition expressions.

Table D-3 shows that because of the way in which the mutation operators were defined for this study, every expression with two conditions with a single occurrence each will have 17 mutants generated for it. This can be better understood by examining the expression tree in figure D-5 for the base expression: *A and B*. We will go through the tree and discuss each mutation. The mutations are shown in italics in the name sets by their corresponding nodes.

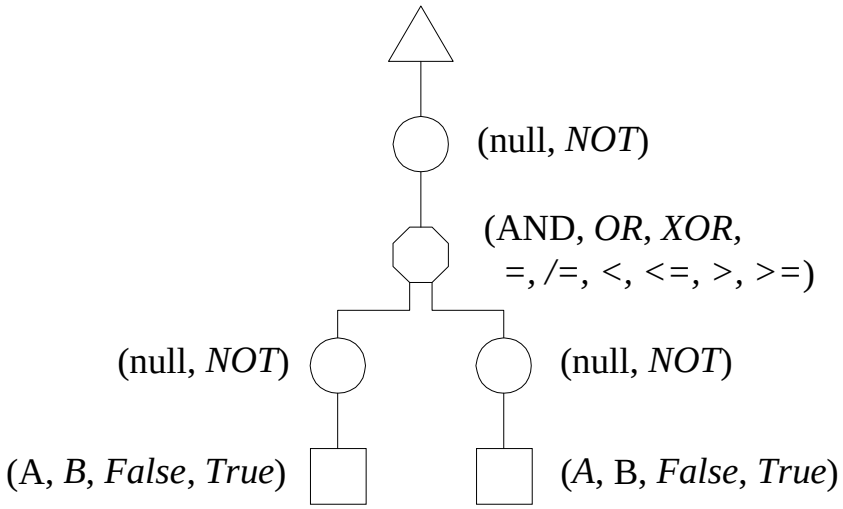


FIGURE D-5. MUTATIONS FOR TWO-CONDITION EXPRESSION TREE

In figure D-5, the three circles in the expression tree identify where complements can be present or not. This allows for the creation of one mutant each (either add or remove the complement). The octagon identifies where the binary Boolean operators (AND, OR, XOR, =, ≠, <, ≤, >, ≥) are present. This allows for the creation of an additional eight mutants by switching the operator to all of the other eight values (e.g., *OR* for *AND*, *XOR* for *AND*, = for *AND*, ≠ for *AND*, < for *AND*, ≤ for *AND*, > for *AND*, and ≥ for *AND*). The squares at the bottom of the tree (the leaves) identify where the conditions are present. This allows for the creation of an additional three mutants each by switching the condition value to each of the other three values (e.g., *B* for *A*, *False* for *A*, *True* for *A*). This is the same information as is presented in table D-2.

Figure D-6 presents the expression tree for three-condition expressions. The analysis for mutations is the same as was used for two-condition expressions in figure D-5. Notice that the only differences between figures D-5 and D-6 is the addition of more operators, and more operands in the condition name sets. The analysis for four- and five-condition expression trees is similar. The only differences will be in the number of operators and operands.

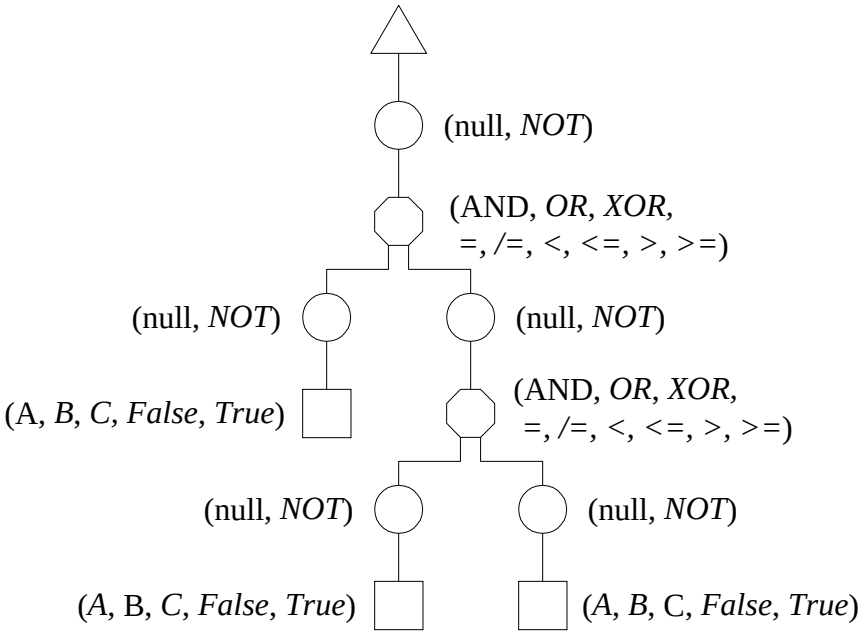


FIGURE D-6. MUTATIONS FOR THREE-CONDITION EXPRESSION TREE

D.2.3.2 Goal: No Expression Equivalent Mutants

One of the goals in the design of the mutation operators was that there would be no mutants generated which represented the same Boolean function as the mutated expression. Unfortunately, this goal could not be accomplished since the relational operators were included in the binary Boolean operator set. This is because the XOR and ≠ operators are equivalent Boolean functions.

If the relational operators had not been included in the binary Boolean operator set, or had eliminated one (XOR, ≠) from the operator set, this goal would have been accomplished by design. Given any expression conforming to the trees, there is no expression for which a single change in the tree will yield an equivalent expression except for (XOR, ≠). This would have not been the case if parenthesis had been considered as significant tokens and mutation operators made more sensitive to the presence or absence of parenthesis. It is not known if this parenthetic sensitivity would have changed the results of the study, or in which way those results would have changed.

D.2.3.3 Goal: Minimum Redundant Mutants

One of the goals in the design of the mutation operators was that the generation of mutants would result in a minimum of, preferably zero, equivalent mutant expressions. It turns out that this goal, at least the zero part, cannot be accomplished. The reason for this is that mutation of the conditions of a binary Boolean operator will lead to equivalency when the replacement is with constants. Recall from Boolean algebra that the expression: *False AND anything*, is always False, regardless of what the *anything* is.

This means that for those expressions involving AND operators, there will always be at least two equivalent mutants. As an example, consider the expression *A and B*. Two mutants of this expression are *False and B* and *A and False*. Both of these mutants represent the Boolean function 0:FFFF. Equivalent mutations exist for the OR and XOR operators also. An example for the expressions *A and B*, *A or B* and *A xor C* is presented in table D-4.

In table D-4, the first column identifies the Boolean function represented by the expressions in that row. The second column, which is the first data column, starts with the expression *A and B* and then presents its mutants according to the mutation operators. The actual mutation is shown in italicized text. Equivalent mutants, which represent the same Boolean function, are gathered in the same row. The second data column presents the same analysis for the expression *A or B*. The final data column presents the same analysis for the expression *A xor B*.

Notice in table D-4 that the expressions *A and B* and *A or B* have six sets of two equivalent mutant expressions each, while the expression *A xor B* has two sets of two equivalent mutants and one set of four equivalent mutants.

D.2.3.4 Goal: Same Number of Boolean Functions Represented by Mutant Expressions

One of the goals in the design of the mutation operators was that the same number of Boolean functions would be represented by the mutants generated for every expression at the same (i.e., N) condition level. Unfortunately, as is evidenced by the data in table D-4, this goal was not achieved. The results of the analyses pertaining to this goal are presented in the following subsections of this report.

TABLE D-4. EQUIVALENT EXPRESSIONS DUE TO MUTATION

Function	A and B		A or B		A xor B
0:FFFF	A and *False* *False* and B				A xor *A* *B* xor B
1:FFFT			A *and* B		A *and* B
2:FFTF	A and *not* B A > B		A > B		A > B
3:FFTT	A and *True* A and *A*		A or *False* A or *A*		A xor *False*
4:FTFF	*not* A and B A < B		A < B		A < B
5:FTFT	*True* and B *B* and B		*False* or B *B* or B		*False* xor B
6:FTTF	A *xor* B A ≠ B		A *xor* B A ≠ B		A ≠ B
7:FTTT	A *or* B				A *or* B
8:TFFF			*not (*A or B*)*		
9:TFFT	A = B		A = B		*not (*A xor B*)* *not* A xor B A xor *not* B A = B
10:TFTF					*True* xor B
11:TFTT	A ≥ B		A or *not* B A ≥ B		A ≥ B
12:TTFF					A xor *True*
13:TTFT	A ≤ B		*not* A or B A ≤ B		A ≤ B
14:TTTF	*not (*A and B*)*				
15:TTTT			A or *True* *True* or B		

To compare mutation against MCDC, one of the questions that must be answered is how well can mutation describe all nonequivalent Boolean functions. How many of the nonequivalent Boolean functions can be represented by expressions generated by the mutation operators designed for this study needed to be determined because the probability of error detection model developed for this study considered all nonequivalent Boolean functions as possible erroneous implementations. The model is independent of the number of changes to the source text that would be required for one of these erroneous implementations.

If mutation is only able to generate a subset of the other functions, then MCDC may not perform the same as the probability of error detection model would predict. Consider that if the mutation operators generate mutants that concentrate on the nonequivalent Boolean functions that MCDC is insensitive to, then MCDC will perform worse than the probability of error detection model predicts. If, on the other hand, mutation never generates an expression for a Boolean function

that MCDC is insensitive to, then MCDC will do better than the model predicts (i.e., 100% probability of error detection at all condition levels).

D.2.3.4.1 Two-Condition Expression Analysis

Since there are 16 total Boolean functions for two conditions, 15 of which are not equivalent to the original expression's function, the designed number of mutants (17) would be able to span the complete set of nonequivalent Boolean functions. However, as is shown in table D-4, not all of the mutants were nonequivalent expressions. Because of this, multiple expressions represent the same Boolean function. During this study, it was determined that between eleven and twelve nonequivalent Boolean functions were covered by the 17 mutants.

In table D-4, notice that there are 17 mutants for each of the three-base expressions (just as there should be). Notice however that only 11 nonequivalent Boolean functions are spanned by these mutants for the AND (0, 2, 3, 4, 5, 6, 7, 9, 11, 13, and 14) and OR (1, 2, 3, 4, 5, 6, 8, 9, 11, 13, and 15) operators. Also notice that for the XOR operator, 12 nonequivalent Boolean functions are spanned (0, 1, 2, 3, 4, 5, 7, 9, 10, 11, 12, and 13). Recall that (6) is an equivalent mutant, so it is not included in the spanning set.

For all the two-condition expressions, the equivalent mutants for one Boolean function ranged between two and five in four different patterns. The number of redundant mutants in expressions using the (AND, OR, $<$, $\leq$, $>$, $\geq$) operators ranged between six sets of two equivalent mutants for expressions with an even number of complemented conditions (i.e., 0 or 2). For those expressions with an odd number of complemented conditions (i.e., 1), there were three sets of two equivalent mutants and one set of four. For the expressions using the (XOR, $\neq$) operators, there was one set of two redundant mutants and one set of four redundant mutants, independent of the number of complemented conditions. For those expressions using the = operator, there was one set of two redundant mutants and one set of five redundant mutants, independent of the number of complemented conditions.

Table D-5 documents the different mutation patterns that were observed in the two-condition analysis. In the first column of table D-5, the number of nonequivalent mutants is given. The second column gives the number of spanned functions represented by those nonequivalent mutants. The next four columns identify the number of sets of equivalent mutants. The third column identifies the number of spanned functions that had a single mutant. The fourth column identifies the number of spanned functions that had two equivalent mutants. The fifth column identifies the number of spanned functions that had four equivalent mutants. The sixth column identifies the number of spanned functions that had five equivalent mutants. The final column identifies the binary Boolean operators involved in that pattern.

TABLE D-5. TWO-CONDITION EXPRESSION MUTATION PATTERNS

Nonequivalent Mutants	Spanned Functions	1	2	4	5	Operators
16	12	10	1	1		(XOR,≠)
17	11	5	6			(AND,OR,<,≤,>,≥)
17	11	7	3	1		(AND,OR,<,≤,>,≥)
17	12	10	1		1	(=)

D.2.3.4.2 Three-Condition Expression Analysis

For three-condition expressions, there are 256 total Boolean functions, 255 of which are not equivalent to the original expression's function. The number of designed mutants (33) is inadequate to span the complete set of Boolean functions. As with the mutants for two-condition expressions, not all of the mutants for three-condition expressions were nonequivalent expressions.

As with the two-condition expressions, there are a variable number of nonequivalent mutants and spanned functions for the mutants of three-condition expressions. The number of spanned functions ranged between a low of 19 and a high of 23. There were 27 different mutation patterns observed for three-condition expressions. A description of the three-condition expression mutation patterns is given in table D-6. Table D-6 follows the same organization as that of table D-5, except that there are more columns for the number of equivalent mutants.

A comparison between tables D-5 and D-6 shows some interesting trends when moving from two- to three-condition expressions. The first trend noted is the large increase in the number of profiles from 4 to 27. This trend shows that with respect to the goals of nonequivalent mutants and spanning functions, things will get much worse with more complex expressions. The second trend noted is the differences in the numbers of distinct operator sets. For two-condition expressions, there were three operator sets, the standard Boolean operators and relationals (AND, OR, <, ≤, >, ≥), the inequality operators (XOR, ≠), and the equality operator (=). With the three-condition expressions, there were a far larger number of operator sets (12). In addition, these operator sets are no longer so clearly distinguished. This trend also shows that with respect to the goals of nonequivalent mutants and spanning functions, things will get much worse with more complex expressions. Both of these trends point to software mutation results diverging greatly from the results obtained from the probability of error detection model developed during this study.

TABLE D-6. THREE-CONDITION EXPRESSION MUTATION PATTERNS

Nonequivalent Mutants	Spanned Functions	1	2	3	4	5	6	7	8	9	Operators
31	22	18	3					1			(XOR,≠)
32	22	14	7		1						(XOR,≠,<,≤)
32	22	15	5	1	1						(XOR,≠,<,≤)
32	22	15	6			1					(AND,OR,XOR,≠,>,≥)
32	22	16	4	1		1					(AND,OR,XOR,≠,>,≥)
32	22	18	3						1		(XOR,=,≠)
32	23	16	6		1						(AND,OR,XOR,≠)
32	23	17	5			1					(XOR,≠,<,≤,>,≥)
32	23	18	3		2						(AND,OR,XOR,≠)
32	23	19	2		1	1					(XOR,≠,<,≤,>,≥)
33	19	9	6	4							(AND,OR,<,≤,>,≥)
33	19	10	4	5							(AND,OR,<,≤,>,≥)
33	19	10	8					1			(AND,OR,<,≤,>,≥)
33	19	11	6	1				1			(AND,OR,<,≤,>,≥)
33	22	13	7	2							(AND,OR,<,≤,>,≥)
33	22	13	8		1						(AND,OR,<,≤,>,≥)
33	22	14	5	3							(AND,OR,<,≤,>,≥)
33	22	14	6	1	1						(AND,OR,<,≤,>,≥)
33	22	14	7			1					(AND,OR,=,<,≤,>,≥)
33	22	15	5	1		1					(AND,OR,=,<,≤,>,≥)
33	22	15	6				1				(AND,OR,=,>,≥)
33	22	16	4	1			1				(AND,OR,=,>,≥)
33	22	18	3							1	(=)
33	23	16	6			1					(AND,OR,=)
33	23	17	5				1				(=,<,≤,>,≥)
33	23	18	3		1	1					(AND,OR,=)
33	23	19	2		1		1				(=,<,≤,>,≥)

D.2.3.4.3 Four-Condition Expression Analysis

For four-condition expressions, there are 65,536 total Boolean functions, 65,535 of which are not equivalent to the original expression's function. The number of designed mutants (51) is inadequate to span the complete set of Boolean functions. This inadequacy is growing ever larger with more complex expressions. As with the mutants for two- and three-condition expressions, not all of the mutants for four-condition expressions were nonequivalent expressions.

As with the two- and three-condition expressions, there are a variable number of nonequivalent mutants and spanned functions for the mutants of four-condition expressions. It was not possible to finish the four-condition expression analysis for mutation patterns during this study.

However, the partial analysis yielded the following results. The number of spanned functions ranged between a low of 27 and a high of 37. The number of nonequivalent mutants ranged between a low of 48 and a high of 51. There were spanned functions represented by as many as 13 equivalent mutants. There were 92 different mutation patterns observed for four-condition expressions.

The four-condition expression mutation patterns exhibited a pattern that was not seen with three-condition expressions. In the three-condition case, every pattern was exhibited for both forms of expression trees. This was as expected, since every Boolean function which can be represented by one of the trees can be represented by the other with a suitable rearrangement of conditions. In the four-condition case, there were mutation patterns that applied to both forms of trees, and patterns that applied to only one form of tree. There were 28 patterns that applied to both forms of trees, 40 patterns that applied to only the tree on the left of figure D-3, and 23 patterns which applied to only the tree on the right of figure D-3. The results of this partial analysis showed that it was necessary to study different forms of expression trees. Why one form of expression tree is more stable than another is a topic for further study.

All of the results obtained for four-condition expressions point to a further divergence between mutation results and the probability of error detection model developed during this study.

D.2.3.4.4 On the Sufficiency of Two-Change Mutants

To put the comparison between MCDC and mutation on level ground, it would be necessary to come up with sufficient mutations to describe all the nonequivalent Boolean functions. It was not possible to do so with only mutants that had a single change within them. As was shown in table D-5, single-change mutation was only able to cover at most 12 of the 15 nonequivalent Boolean functions for two-condition expressions. It was possible for the two-condition expressions to describe the entire Boolean function space using both one- and two-change mutants. The results of this analysis are demonstrated in table D-7 for the base expression *A and B*.

In table D-7, the first column contains the response profile for the expression(s) within that row. The second column contains the expressions corresponding to the Boolean function. If the expression is a mutant, then the mutations are shown in italics. For the single-change mutants, all expressions are shown for those functions that had overlapped. For the two-change mutants, only one is listed when no single-change mutant corresponds to the function. The third column lists the number of one-change mutants that were generated for this function. The final column lists the number of two-change mutants that were generated for this function.

As the data in table D-7 shows, all Boolean functions can be spanned with the combination of one- and two-change mutants. Notice that only two-change mutants are sufficient to cover all functions. Also notice that the coverage of functions is not uniform (i.e., some functions have disproportionate numbers of expressions as compared to others).

TABLE D-7. ONE- AND TWO-CHANGE MUTATIONS AND SPANNED FUNCTIONS FOR *A and B*

Function	Expression	1 Change	2 Change
1:FFFT	A and B		23
0:FFFf	A and *False* *False* and B	2	54
2:FFtf	A and *not* B A > B	2	9
3:FFtT	A and *True* A and *A*	2	22
4:FtFf	*not* A and B A < B	2	9
5:FtFT	*True* and B *B* and B	2	22
6:Fttf	A *xor* B A ≠ B	2	20
7:FttT	A *or* B	1	11
8:tFFf	*not* A and *not* B		8
9:tFFT	A = B	1	19
10:tFtf	*True* and *not* B		16
11:tFtT	A ≥ B	1	13
12:ttFf	*not* A and *True*		16
13:ttFT	A ≤ B	1	13
14:tttf	*not (*A and B*)*	1	4
15:tttT	*not (False* and B*)*		30

Similar to the analysis carried out on two-condition expressions, the possibility that two-change mutants would allow for the spanning of all the nonequivalent Boolean functions was looked at. The possibility of this is based on the following reasoning. For three-condition expressions, the smallest number of spanned functions was 19. If each of these 19 mutants could themselves span a minimum of 19 functions, one of which would be the original expression and its function, then there would be the possibility of spanning 19*18=342 functions, more than is needed for the 256 functions possible. Unfortunately, this is not what happens as demonstrated in table D-8 for the base expression *A and (B and C)*. This table follows a similar format as table D-7 for the two-condition analysis. There are two sets of data columns. The first data column identifies the response profile for the expressions categorized in that row. The second column identifies the number of one-change mutants that correspond to the Boolean function while the final column identifies the number of two-change mutants for the function.

Examination of table D-8 shows that only 82 of the 255 nonequivalent Boolean functions are spanned by the one- and two-change mutants, despite the fact that 1122 mutants were generated. It turns out that this picture can be made slightly better with other expressions but cannot be

TABLE D-8. ONE- AND TWO-CHANGE MUTATIONS AND SPANNED FUNCTIONS FOR *A and (B and C)*

Function	1 Change	2 Change		Function	1 Change	2 Change
1:FFFFFFFT		47		119:FtttFttT		2
0:FFFFFFf	3	159		120:FttttFFf		4
2:FFFFFFtf	2	11		127:FttttttT		2
3:FFFFFFtT	3	37		135:tFFFFttT		2
4:FFFFFtFf	2	11		143:tFFFtttT		2
5:FFFFFtFT	3	37		144:tFFtFFFf		4
6:FFFFFttf	2	22		150:tFFtFttf		8
7:FFFFFttT	1	11		153:tFFttFFT		2
8:FFFFtFFf		10		159:tFFttttT		6
9:FFFFtFFT	1	23		165:tFtFFtFT		4
10:FFFFtFtf		36		175:tFtFtttT		4
11:FFFFtFtT	1	17		176:tFttFFFf		4
12:FFFFttFf		36		180:tFttFtFf		8
13:FFFFttFT	1	17		187:tFtttFtT		6
14:FFFFtttf	2	13		191:tFttttT		6
15:FFFFtttT		64		195:ttFFFFtT		4
16:FFFtFFFf	2	9		207:ttFFtttT		4
17:FFFtFFFT	3	39		208:ttFtFFFf		4
30:FFFttttf	2	20		210:ttFtFFtf		8
31:FFFtttT	1	11		221:ttFtttFT		6
32:FFtFFFFf		8		223:ttFttttT		6
34:FFtFFFtf		20		224:tttFFFFf		8
45:FFtFttFT		10		225:tttFFFFT	1	19
47:FFtFtttT		6		238:tttFtttf		18
48:FFttFFFf		8		239:tttFtttT	1	13
51:FFttFFtT		14		240:ttttFFFf		8
60:FFttttFf		8		241:ttttFFFT	1	13
63:FFttttT		4		242:ttttFFtf		6
64:FtFFFFFf		8		243:ttttFFtT		8
68:FtFFFtFf		20		244:ttttFtFf		6
75:FtFFtFtT		10		245:ttttFtFT		8
79:FtFFtFtT		6		246:ttttFttf		6
80:FtFtFFFf		8		247:ttttFttT		2
85:FtFtFtFT		14		248:tttttFFf		2
90:FtFttFtf		8		249:tttttFFT		6
95:FtFttttT		4		250:tttttFtf		6
96:FttFFFFf		8		251:tttttFtT		6
102:FttFFttf		4		252:ttttttFf		6
105:FttFtFFT		10		253:ttttttFT		6
111:FttFtttT		6		254:tttttttf	1	4
112:FtttFFFf		4		255:tttttttT		24

made perfect. The reason for this is that not all Boolean functions which are a function of three conditions can be expressed with the expression tree used in this study and the Boolean operators that were available. An example of this is the Boolean function 22:FFFTFTTF. Of the many different expressions that can be used to describe this function, none of them can use the form of tree that was used in this study as multiple occurrences of the conditions are required. This again shows, even more strongly than the two-condition expression analysis, that there will be some difficulty comparing mutation sensitivity of MCDC to the probability of error detection model.

D.2.3.5 Goal: Same Represented Boolean Functions for Equivalent Expressions

One of the goals in the design of the mutation operators was that the same represented (spanned) Boolean functions would result from the mutation of two equivalent expressions. It turns out that this goal cannot be accomplished. The reason is that the mutations of the binary Boolean operators (RLO) generate vastly different expressions. This resulted in there being at least two different Boolean functions between the mutations applied to different two-condition expressions representing the same Boolean function. An example of this analysis is given in table D-9 for the expressions (*A and B*), (*not (not A or not B)*), (*not A < B*), and (*not (A ≤ not B)*). All of these expressions represent the Boolean function (1:FFFT).

TABLE D-9. MUTATION DIFFERENCES BETWEEN EQUIVALENT EXPRESSIONS

1:FFFT	A and B	not (not A or not B)	not A < B	not (A ≤ not B)
0:FFFf	*False* and B A and *False*	not (not *False* or not B) not (not A or not *False*)	not *False* < B not A < *False*	not (*False* ≤ not B) not (A ≤ not *False*)
2:FFtf	A and *not* B A > B	not (not A or B) not (not A ≥ not B)	not A < *not* B	not (A ≤ B)
3:FFtT	A and *True* A and *A*	not (not A or not *True*) not (not A or not *A*)	not A < *True* not A < *A*	not (A ≤ not *True*) not (A ≤ not *A*)
4:FtFf	*not* A and B A < B	not (A or not B) not (not A ≤ not B)	A < B not A *and* B	not (A *or* not B) not (*not* A ≤ not B)
5:FtFT	*True* and B *B* and B	not (not *True* or not B) not (not *B* or not B)	not *True* < B not *B* < B	not (*True* ≤ not B) not (*B* ≤ not B)
6:Fttf	A *xor* B A ≠ B	not (not A = not B)	not A = B	not (A *xor* not B) not (A ≠ not B)
7:FttT	A *or* B	not (not A *and* not B)	not A ≤ B	not (A < not B)
8:tFFf			not A > B	not (A ≥ not B)
9:tFFT	A = B	not (not A *xor* not B) not (not A ≠ not B)	not A *xor* B not A ≠ B	not (A = not B)
11:tFtT	A ≥ B	not (not A > not B)		
13:ttFT	A ≤ B	not (not A < not B)	not A *or* B	not (A *and* not B)
14:tttf	*not* (A and B)	not A or not B	*not (*not A < B*)* not A ≥ B	A ≤ not B not (A > not B)

In table D-9, the first column identifies the response profile for the expressions within that row. Responses which differ from those given by the base expressions are given in lower case (e.g., "0:FFFf" differs from "1:FFFT" in the fourth-condition combination, hence the final "f" is lower case). The second column identifies the base expression *A and B*, and the mutants that are generated from it using the mutation rules developed for this study. The actual mutation is given in italicized text (e.g., substituting "False" for "A" in the expression "A and B" mutates to "*False* and B"). The third column identifies the base expression *not (not A or not B)*, and its mutants. The fourth column identifies the base expression *not A < B*, and its mutants. The fifth column identifies the base expression *not (A ≤ not B)*, and its mutants. Notice that there are some response profiles covered by mutants of all expressions, and others that are only covered by some.

The approach for three-condition expressions was equivalent to that for the two-condition expressions, with similar results to those given in table D-9. This was to be expected since the three-condition expressions only exacerbate this phenomenon. In addition, an additional analysis is necessary for three-condition expressions. Recall that in figure D-2 that there were two symmetrical tree forms that could be used for three-condition expressions. It was stated then that either tree form was sufficient for representing three-condition expressions with a single occurrence of each condition.

Note that even though every expression can be expressed with either form of tree, there is a slight difference in the mutants that will be generated between the two forms of trees. The difference is in the Boolean functions represented by the mutated expressions. This is similar to the case in table D-9 where multiple equivalent Boolean expressions generated mutants that covered different Boolean functions. Table D-10 presents two forms of the same expression (*A and B and C*) and the mutants that are generated from them. The sensitivity of MCDC was unaffected by these differences (i.e., the same number of mutants result in either case, and the same number of covered Boolean functions also result in either case).

In table D-10, just as in table D-9, the first column identifies the response profile for the expressions within that row. Responses that differ from those given by the base expressions are given in lower case. The second column identifies the base expression *A and (B and C)* and the mutants that are generated from it using the mutation rules developed for this study. The actual mutation is given in italicized text. The third column identifies the base expression *(A and B) and C* and its mutants. Notice that there are some response profiles covered by mutants of both expressions and others that are only covered by one.

TABLE D-10. MUTATION DIFFERENCES DUE TO DIFFERENT TREE FORMS

1:FFFFFFFT	A and (B and C)	(A and B) and C
0:FFFFFFFf	*False* and (B and C) A and (*False* and C) A and (B and *False*)	(*False* and B) and C (A and *False*) and C (A and B) and *False*
2:FFFFFFtf	A and (B and *not* C) A and (B > C)	(A and B) and *not* C (A and B) > C
3:FFFFFFtT	A and (B and *True*) A and (B and *A*) A and (B and *B*)	(A and B) and *True* (A and B) and *A* (A and B) and *B*
4:FFFFFtFf	A and (*not* B and C) A and (B < C)	(A and *not* B) and C (A > B) and C
5:FFFFFtFT	A and (*True* and C) A and (*A* and C) A and (*C* and C)	(A and *True*) and C (A and *A*) and C (A and *C*) and C
6:FFFFFttf	A and (B *xor* C) A and (B ≠ C)	
7:FFFFFttT	A and (B *or* C)	
9:FFFFtFFT	A and (B = C	
11:FFFFtFtT	A and (B ≥ C)	
13:FFFFttFT	A and (B ≤ C)	
14:FFFFtttf	A and *not* (B and C) A > (B and C)	
16:FFFtFFFf	*not* A and (B and C) A < (B and C)	(*not* A and B) and C (A < B) and C
17:FFFtFFFT	*True* and (B and C) *B* and (B and C) *C* and (B and C)	(*True* and B) and C (*B* and B) and C (*C* and B) and C
20:FFFtFtFf		(A *xor* B) and C (A ≠ B) and C
21:FFFtFtFT		(A *or* B) and C
30:FFFttttf	A *xor* (B and C) A ≠ (B and C)	
31:FFFttttT	A *or* (B and C)	
65:FtFFFFFT		(A = B) and C
69:FtFFFtFT		(A ≥ B) and C
81:FtFtFFFT		(A ≤ B) and C
84:FtFtFtFf		*not* (A and B) and C (A and B) < C
86:FtFtFttf		(A and B) *xor* C (A and B) ≠ C
87:FtFtFttT		(A and B) *or* C
169:tFtFtFFT		(A and B) = C
171:tFtFtFtT		(A and B) ≥ C
225:tttFFFFT	A = (B and C)	
239:tttFtttT	A ≥ (B and C)	
241:ttttFFFT	A ≤ (B and C)	
253:tttttFT		(A and B) ≤ C
254:ttttttf	*not (*A and (B and C)*)*	*not ((*A and B) and C*)*

D.3 Efficacy Analysis (Mutation Sensitivity)

The main part of this study was to investigate how MCDC test cases would perform against mutants. To perform this analysis, every expression was mutated according to the rules developed for this study. Then every minimal MCDC compliant test set was generated for each of the expressions. Finally, each of these test sets was compared against the mutants of the expression to see how many mutants were killed. The kill analysis was actually performed two ways. First, the kill analysis was performed against all mutants to see what the efficacy against mutants themselves was. Second, the mutants were grouped by spanned function, and the kill analysis performed against the functions of mutants. This second analysis is more comparable to what was done earlier with MCDC and the probability of error detection model.

An example kill analysis for mutant expressions is given in table D-11 for the expression *A and B*. The first three rows of the table give the condition codes for the condition combinations in (A,B). The first row gives the number, while the second row gives the values for A and the third row gives the values for B. The first column identifies which condition is given by the corresponding row. Columns three through six give the condition combinations, and the responses from the expressions contained lower in the table. When a response from an expression differs from that given by the base expression *A and B*, it is shown in lower case. If that response is one of the ones probed by the test set, then it is italicized. Column two gives the response profile function number for the expression given in that row. Column seven gives the expression under consideration. If the expression represents a mutant, the mutation is given in italicized text. The final column identifies which condition combinations (tests) will kill the mutant in the corresponding row. The final row identifies the MCDC test set for the base expression. Those condition combinations involved in the test have a lower case "*x*" in the corresponding column.

There are two ways to perform the kill analysis. One way uses the condition combination columns. Any mutant that has a lower case italicized entry in a column with a test set "*x*" is killed by that test. This can be done for every test in the test set. Any mutants that have not been killed are still alive. These are the mutants that have no italicized lower-case entry in the test set columns. The second way to perform the analysis uses the kill sets. Any expression that has an entry in its kill set that appears in the test set is killed by the test set. The remaining live mutants will have no entries in their kill sets in common with the test set. In this case, MCDC kills 16 of the 17 mutants for the expression *A and B*. The mutant *A = B* is still live because it can only be killed by the test (0), which is not in the MCDC test set (1,2,3).

The same analysis is performed against the spanned functions. For this analysis, all of the equivalent mutants are gathered into the Boolean function that they represent. Note that all mutants representing the same function will have the same function number and the same kill set. An example for the expression *A and B* is given in table D-12.

Examination of table D-12 shows that MCDC kills 10 of the 11 spanned functions for the expression *A and B*. Again, the mutant *A = B*, and thereby function 9:TFFT, is still live because it can only be killed by the test (0), which is not in the MCDC test set (1,2,3). This is as expected from the probability of error detection model as it shows that for two-condition expressions, MCDC is sensitive to 14 of the 15 nonequivalent functions.

TABLE D-11. MCDC MUTANT EXPRESSION KILL ANALYSIS FOR *A and B* (EXPANDED TRUTH TABLE)

		0	1	2	3		
A:		F	F	T	T		
B:		F	T	F	T		
	1:	F	F	F	T	A and B	
	4:	F	*t*	F	*f*	*not* A and B	(1,3)
	0:	F	F	F	*f*	*False* and B	(3)
	5:	F	*t*	F	T	*True* and B	(1)
	5:	F	*t*	F	T	*B* and B	(1)
	7:	F	*t*	*t*	T	A *or* B	(1,2)
	6:	F	*t*	*t*	*f*	A *xor* B	(1,2,3)
	9:	*t*	F	F	T	A = B	(0)
	6:	F	*t*	*t*	*f*	A ≠ B	(1,2,3)
	4:	F	*t*	F	*f*	A < B	(1,3)
	13:	*t*	*t*	F	T	A ≤ B	(0,1)
	2:	F	F	*t*	*F*	A > B	(2,3)
	11:	*t*	F	*t*	T	A ≥ B	(0,2)
	2:	F	F	*t*	*f*	A and *not* B	(2,3)
	0:	F	F	F	*f*	A and *False*	(3)
	3:	F	F	*t*	T	A and *True*	(2)
	3:	F	F	*t*	T	A and *A*	(2)
	14:	t	*t*	*t*	*f*	*not (*A and B*)*	(0,1,2,3)
			x	*x*	*x*		T:(1,2,3)

TABLE D-12. MCDC MUTANT FUNCTION KILL ANALYSIS FOR *A and B* (EXPANDED TRUTH TABLE)

		0	1	2	3		
A:		F	F	T	T		
B:		F	T	F	T		
	1:	F	F	F	T	A and B	
	0:	F	F	F	*f*	*False* and B, A and *False*	(3)
	2:	F	F	*t*	*f*	A and *not* B A > B	(2,3)
	3:	F	F	*t*	T	A and *True* A and *A*	(2)
	4:	F	*t*	F	*f*	*not* A and B A < B	(1,3)
	5:	F	*t*	F	T	*True* and B *B* and B	(1)
	6:	F	*t*	*t*	*f*	A *xor* B A ≠ B	(1,2,3)
	7:	F	*t*	*t*	T	A *or* B	(1,2)
	9:	*t*	F	F	T	A = B	(0)
	11:	*t*	F	*t*	T	A ≥ B	(0,2)
	13:	*t*	*t*	F	T	A ≤ B	(0,1)
	14:	t	*t*	*t*	*f*	*not (*A and B*)*	(0,1,2,3)
			x	*x*	*x*		T:(1,2,3)

The analyses behind tables D-11 and D-12 can be applied to all expressions/mutants/test sets for all condition levels. The mutation kill probability analysis results for one through five conditions is contained in table 39 for mutants and table 40 for spanned functions.

D.4 Coupling Effect

As mentioned at the beginning of this appendix, one of the assumptions of mutation testing is that of the coupling effect. The coupling effect asserts that tests which are sufficient for simple mutants (i.e., level 1 or single-change mutants) will detect the more complex mutants (i.e., level > 1 or multiple-change mutants) [1,6,7]. Though not one of the original intentions of this substudy, a brief analysis of the validity of this assumption was nevertheless conducted. This section presents the results of the analysis of this coupling effect which disproves it. Table D-13 presents the results of this investigation for the two-condition expression *A xor B* while table D-14 presents the results of this investigation for the three-condition expression *A and (B xor C).*

TABLE D-13. COUPLING EFFECT RESULTS FOR *A xor B*

Function	Expression	Test Set	Mutant Function	Mutant Expression
6:FTTF	A *xor* B	T:(1,2,3)	14:tTTF	*not* (A *and* B) A ≤ *not* B *not* A ≥ B

TABLE D-14. COUPLING EFFECT RESULTS FOR *A and (B xor C)*

Function	Expression	Test Set	Mutant Function	Mutant Expression
6:FFFFFTTF	A and (B xor C)	T:(0,1,5,6,7)	14:FFFFtTTF	A and *not* (B *and* C) A and (B ≤ *not* C) A and (not B ≥ C) A > (B *and* C)
			30:FFFttTTF	A *xor* (B *and* C) A ≠ (B *and* C)
		T:(0,2,5,6,7)	14:FFFFtTTF	A and *not* (B *and* C) A and (B ≤ *not* C) A and (not B ≥ C) A > (B *and* C)
			30:FFFttTTF	A *xor* (B *and* C) A ≠ (B *and* C)
		T:(1,3,5,6,7)	14:FFFFtTTF	A and *not* (B *and* C) A and (B ≤ *not* C) A and (not B ≥ C) A > (B *and* C)
		T:(2,3,5,6,7)	14:FFFFtTTF	A and *not* (B *and* C) A and (B ≤ *not* C) A and (not B ≥ C) A > (B *and* C)

The two-condition expression analysis first takes all of the mutants that were generated according to the mutation operators for each expression and found the minimal test set(s) which would kill all of those mutants. It turned out that for all the expressions using the (AND, OR, <, ≤, >, ≥) binary Boolean operators, all four tests were needed to kill all mutants. This of course meant that mutation sensitive test sets were sensitive to all mutants and spanned functions. The expressions using the (XOR, =, ≠) operators only needed three tests to kill all mutants, which allowed the analysis. The Boolean function that the test set was insensitive to was identified and two-change mutants for that function determined. In all cases, there were three mutant expressions that fit the function. The results of this analysis are presented in table D-13 for the expression *A xor B*.

In table D-13, the first column presents the response profile for the expression presented in the second column. The third column identifies the mutation-adequate test set for the corresponding expression. The fourth column presents the response profile that the mutation-adequate test set is unable to kill. The final column presents the expressions, generated by our mutation operators, that are two changes off of the expression in column two for the mutant function in column four (i.e., mutants of mutants).

Examination of table D-13 clearly shows that the coupling effect does not hold for something as simple as two-condition logic expressions. At this point, one could question whether the results of the study methodology are flawed as opposed to the coupling effect. After all, more extensive studies have tended to show that the coupling effect holds with a different set of mutation operators and randomly generated test data [6,7]. One of the major differences between earlier studies and this study is that the previous studies have used complete (sub)programs to introduce the mutants into and this analysis uses only logic expressions. This difference is not important since each expression could be embedded into a complete (sub)program and generate tests which would kill all the single-change mutants for the entire (sub)program. If this test set did not go beyond the single-change mutation adequate test set for the expression, it would not be able to kill the two-change mutants for that expression. Recall that the two-change mutants behave exactly as the “correct” expressions would under the single-change adequate test set.

To push this investigation further, the three-condition expressions were examined using the same methodology as was used on the two-condition expressions. As with the two-condition expressions, the single-change mutation adequate test sets for all three-condition expressions containing only the (AND, OR, <, ≤, >, ≥) binary Boolean operators killed all two-change mutants. Also, the three-condition expressions containing any of the (XOR, =, ≠) binary Boolean operators had two-change mutants which were not killed by the single-change mutation adequate test sets. The results of that examination for the expression *A and (B xor C)* are shown in table D-14.

Table D-14 shows that between one and two spanned functions are not killed by the single-change mutation adequate test set. This analysis shows that further investigation of the coupling effect is warranted. It has been demonstrated that the coupling effect does not always hold.

It should be noted that the performance of mutation adequate test sets only did as well as documented in this section because of the inclusion of the relational operators in the Boolean

operator set. If those operators are removed, and the analyses of tables D-13 and D-14 redone, an entirely different picture is obtained for the coupling effect. In the two-condition expression case, all expressions have mutation adequate test sets requiring less than all four tests, and all expressions have two-change mutants that are not killed by the single-change adequate test sets. The same holds for the three-condition expressions.

D.5 Mutation Subsumes MCDC

As mentioned at the beginning of this appendix, one of the assertions (and assumptions) of the mutation testing supporters is that software mutation subsumes all other structural coverage criteria, including MCDC [2, 8]. Though not one of the original intentions of this substudy, a brief analysis of the validity of this assertion was nevertheless conducted. This section presented the results of the initial analysis of this subsumption assertion which disproves it. It is acknowledged that this is a very preliminary assessment.

The first part of the analysis looks at the mutants that were supposed to provide MCDC defined in reference 8. In this study, eight MCDC adequate mutants were defined. Two of the mutants were supposed to show that both decision outcomes occur. Two of the mutants were supposed to show that all condition outcomes occur for each condition. Two of the mutants were supposed to show that each condition independently affects the outcome of an AND. Two of the mutants were supposed to show that each condition independently affects the outcome of an OR. As will be shown next, this did not quite occur.

It was unclear in the reference 8 whether the mutants should be treated as mutations or as traps. Traps are statements that look for a specific program state to be achieved, and then raise an exception to halt program execution and effectively kill the mutation. Since it was not clear which way to proceed, both methods were studied. The analysis for mutations is presented in tables D-15 and D-16 (for AND and OR respectively), while the analysis for traps is presented in tables D-17 and D-18. As these results show, treating the mutations as traps achieves more of the goals of MCDC, but still does not satisfy it.

In tables D-15 and D-16, columns one through five give the response profile for the expression in that row. The first column presents the response profile number while columns two through five present the responses to the condition combinations. Differences between the mutants and the base expressions are in lower-case text. The sixth column presents a mutant number for use in subsequent commentary. The seventh column presents the expression corresponding to the response profiles. The first row presents the base expression, and the remaining rows present the mutants. The eighth column presents commentary for the purpose of the mutants. The final two rows identify the tests necessary to kill all the mutants. The tests are identified in the first column. The fourth column contains justification for their presence. All lower-case responses of the mutants within the same columns as the tests are italicized.

Notice that in tables D-15 and D-16 that only two tests are necessary to satisfy the supposedly MCDC adequate mutants. This is an inadequate number of tests as MCDC requires a minimum of three tests for two conditions. Also notice that not all of the goals of MCDC were satisfied. First, the mutant to ensure that the decision was True does not do so. In fact, it requires no tests

TABLE D-15. MCDC ADEQUATE MUTATIONS FOR *A and B*

F					M No.	Expression	Comment
1:	F	F	F	T		A and B	base expression
1:	F	F	F	T	1	(A and B) = True	show decision True
14:	t	*t*	*t*	f	2	(A and B) = False	show decision False
3:	F	F	*t*	T	3	A = True	show condition A True
12:	t	*t*	F	f	4	A = False	show condition A False
5:	F	*t*	F	T	5	B = True	show condition B True
10:	t	F	*t*	f	6	B = False	show condition B False
13:	t	*t*	F	T	7	(A and B) = A	show condition A independent
11:	t	F	*t*	T	8	(A and B) = B	show condition B independent
		x				T:(1)	required to kill 5, also kills 2, 4, and 7
			x			T:(2)	required to kill 3, also kills 2, 6, and 8

TABLE D-16. MCDC ADEQUATE MUTATIONS FOR *A or B*

F					M No.	Expression	Comment
7:	F	T	T	T		A or B	base expression
7:	F	T	T	T	1	(A or B) = True	show decision True
8:	t	*f*	*f*	f	2	(A or B) = False	show decision False
3:	F	*f*	T	T	3	A = True	show condition A True
12:	t	T	*f*	f	4	A = False	show condition A False
5:	F	T	*f*	T	5	B = True	show condition B True
10:	t	*f*	T	f	6	B = False	show condition B False
13:	t	t	F	T	7	(A or B) = A	show condition A independent
11:	t	*f*	T	T	8	(A or B) = B	show condition B independent
		x				T:(1)	required to kill 3, also kills 2, 6, and 7
			x			T:(2)	required to kill 5, also kills 2, 4, and 8

since it is equivalent to the base expression. Second, the mutant to ensure that the conditions are independent also does not do so. What the mutations accomplished was to ensure that both conditions were True and False, which is merely Condition Coverage.

In tables D-17 and D-18, columns one through five again give the response profile for the function/mutation trap in the corresponding row. Note that for traps, the mutant is killed whenever the trap is fired. Therefore, in these two tables all True responses from the mutants are in lower case, whether or not they agree with the base expression response. The sixth column presents a mutant number for use in subsequent commentary. The seventh column presents the expression corresponding to the response profiles. The first row presents the base expression, and the remaining rows present the mutants. The eighth column presents commentary for the purpose of the mutants. The final two rows identify the tests necessary to kill all the mutants. The tests are identified in the first column. The fourth column contains justification for their presence. All lower-case responses of the mutants within the same columns as the tests are italicized.

In tables D-17 and D-18, as with tables D-15, and D-16 previously, the mutants achieve some but not all of the MCDC objectives. In the case of using the mutants as traps, these mutants now achieve Condition Decision Coverage.

TABLE D-17. MCDC ADEQUATE TRAPS FOR *A and B*

F					M No.	Expression	Comment
1:	F	F	F	T		A and B	base expression
1:	F	F	F	*t*	1	(A and B) = True	show decision True
14:	*t*	t	t	F	2	(A and B) = False	show decision False
3:	F	F	t	*t*	3	A = True	show condition A True
12:	*t*	t	F	F	4	A = False	show condition A False
5:	F	t	F	*t*	5	B = True	show condition B True
10:	*t*	F	t	F	6	B = False	show condition B False
13:	*t*	t	F	*t*	7	(A and B) = A	show condition A independent
11:	*t*	F	t	*t*	8	(A and B) = B	show condition B independent
	x					T:(0)	kills 2, 4, 6, 7, and 8 (only test that kills 2, 4, and 6, the remaining mutants)
				x		T:(3)	required to kill 1, also kills 3, 5, 7, and 8

TABLE D-18. MCDC ADEQUATE TRAPS FOR *A or B*

F					M No.	Expression	Comment
7:	F	T	T	T		A or B	base expression
7:	F	t	t	*t*	1	(A or B) = True	show decision True
8:	*t*	F	F	F	2	(A or B) = False	show decision False
3:	F	F	t	*t*	3	A = True	show condition A True
12:	*t*	t	F	F	4	A = False	show condition A False
5:	F	t	F	*t*	5	B = True	show condition B True
10:	*t*	F	t	F	6	B = False	show condition B False
11:	*t*	F	t	*t*	7	(A or B) = A	show condition A independent
13:	*t*	t	F	*t*	8	(A or B) = B	show condition B independent
	x					T:(1)	required to kill 2, also kills 4, 6, 7, and 8
				x		T:(2)	kills 1, 3, 5, 7, and 8 (only test that kills 1, 3, and 5, the remaining mutants)

The second part of the analysis is based on comparisons between test sets. All the minimal test sets for Mutation, Unique-Cause MCDC, Unique-Cause+Masking MCDC, and Masking MCDC were generated. Recall from section 8 that the form of expressions used in this study made Unique-Cause MCDC and Unique-Cause+Masking MCDC identical. Four comparisons were then performed.

1. The average number of tests necessary to satisfy the criteria. These results are presented in table D-19 and show that mutation requires on average more tests than either form of MCDC.

TABLE D-19. AVERAGE NUMBER OF TESTS IN A MINIMAL TEST SET

Number of Conditions	Mutation	Unique Cause	Unique Masking	Masking
1	2.0	2.0	2.0	2.0
2	3.6666	3.0	3.0	3.0
3	5.0882	4.0	4.0	4.0
4	6.6739	5.0	5.0	4.7990
5	*8.5358*	*6.0*	*6.0*	*5.5112*

2. The average number of minimal test sets for each criterion. Though not strictly related to the issue of mutation subsuming MCDC, these results do speak to one measure of the ease of satisfying the criteria. The results are presented in table D-20 and show that of the three forms of MCDC, Masking MCDC should be the easier one to satisfy. The issue of Mutation compared to MCDC requires further investigation. A larger number of larger test sets does not necessarily mean that Mutation is easier to satisfy than MCDC.

TABLE D-20. AVERAGE NUMBER OF MINIMAL TEST SETS

Number of Conditions	Mutation	Unique Cause	Unique Masking	Masking
1	1.0	1.0	1.0	1.0
2	1.0000	2.0000	2.0000	2.0000
3	3.7777	7.3333	7.3333	9.3333
4	41.9363	41.2222	41.2222	67.4444
5	*2064.67*	*73.1313*	*73.1313*	*460.139*

3. The probability that Mutation will satisfy either form of MCDC. These results are presented in table D-21 and clearly show that Mutation does not subsume any form of MCDC.

TABLE D-21. PROBABILITY OF MUTATION SATISFYING MCDC

Number of Conditions	Unique Cause	Unique Masking	Masking
1	1.0	1.0	1.0
2	1.0000	1.0000	1.0000
3	0.7947	0.7947	0.8750
4	0.5113	0.5113	0.8040
5	*0.2493*	*0.2493*	*0.8507*

4. The probability that either form of MCDC will satisfy mutation. These results are presented in table D-22, and show that all forms of MCDC do not subsume mutation. Note that this analysis differs from that in tables 39 and 40 in that table D-22 shows the probability that a MCDC test set is mutation adequate (i.e., kills *all* mutants).

TABLE D-22. PROBABILITY OF MCDC SATISFYING MUTATION

Number of Conditions	Unique Cause	Unique Masking	Masking
1	1.0	1.0	1.0
2	0.1667	0.1667	0.1667
3	0.0530	0.0530	0.0774
4	0.0167	0.0167	0.0511
5	*0.0000*	*0.0000*	*0.0000*

D.6 Mutation Conclusions

MCDC and Software Mutation (Fault Injection) are both software test data set adequacy criteria. In this substudy an initial comparison was conducted, and raised more questions than there were answers.

To conduct this study, it was necessary to design a set of mutation operators for logic expressions written in the Ada programming language. The design approach, though different from what has been documented in previous mutant design efforts, nevertheless led to an equivalent set of mutants to the previous efforts.

During the design of the mutation operators, it was found that a number of goals that were identified as desirable cannot be accomplished due to theoretical grounds. It turns out that the complete removal of redundant mutants cannot be accomplished in logic expressions because of some of the identities in Boolean algebra. It was discovered that the mutation operators would not provide a consistent spanning set for any level of multiple-condition expressions (i.e., two or more conditions). There was no time in the study to identify the underlying reasons why this was so. Study of that property requires further study.

In comparing MCDC against and with mutation, it was discovered that mutation does not provide as broad an exposure to error as was developed in the probability of error detection model. Because mutation only applied to a subset of what was covered by the model, the efficacy results were different than predicted by the model. However, it was shown that MCDC did perform very well against logic mutants. This study was limited to Boolean variable conditions in the expressions. Expanding the study to incorporate relational conditions requires further study.

Though not intended to be addressed in this study, two fundamental premises of mutation were considered. The first, the *coupling effect*, has been used to try and make mutation more tractable by limiting the number of mutants that need to be generated and killed. Data was provided showing that this effect does not hold with simple logic expressions. Because the "experimental protocol" differs from others who have looked into this issue, this needs further study. The second, that mutation subsumes MCDC, was also disproved with a simple example. The extent of this nonsubsumption was documented with further analyses. One of the things that came out of this further analysis is that Mutation and Masking MCDC performed against each other better than the other forms of MCDC. Why this is so requires further study.

It was also established that mutation requires more tests than any form of MCDC. This additional testing raises the probability of error detection according to the model developed as a part of this study. Whether that increase in testing is cost-effective is something requiring further study.

Finally, despite all the problems found with mutation, it is believed that it can be used as a yardstick with which to compare different forms of MCDC. This was done in section 8.

D.7 A Brief Investigation Into the Outside Variable Replacement Mutation

For the purposes of this study, variable (condition) replacements were limited to those that occurred in the expression, (e.g., for two-condition expressions, only A and B were considered). The major reason for this is that the probability of error detection model developed during a previous phase of this study made the assumption that it was known whether an expression was to represent a function of (A, B) or a function of (A, C). The model made no attempt to address the error of whether an expression was written in terms of (A, B) instead of (A, C) as intended. For one thing, this is an error that is probably best addressed by inspection as opposed to testing. For another thing, the mutation research into sufficient mutation operators has found that this mutation, Variable Replacement, is not very useful though it does produce an inordinate amount of mutants. Finally, this study only has logical expressions to work with outside the context of their systems, so there is no way to know how many outside variables to use in the mutation set of replacement Boolean variables.

As a bit of a reality check on whether this issue should be addressed in further studies, a brief investigation was conducted into the adequacy of MCDC against outside variable replacements. This investigation utilized all of the two-condition expressions and one outside variable, as well as a single two-condition expression and two outside variables. For the single outside variable analysis, the following methodology was used. First, the expression was mutated two ways, one with the substitution of C for A, and the second way was the substitution of C for B. Table D-23 shows the base expression *A and B* and the two mutants generated from it. The mutation is indicated in the table by italicized text (*C* in both cases).

TABLE D-23. TWO-CONDITION EXPRESSION, ONE OUTSIDE VARIABLE MUTATIONS

A	and	B
C	and	B
A	and	*C*

Now that the base expression and its mutants (2) are known, it can be determined if MCDC is sensitive to this error or not. In order to do this, the MCDC test set must be run against the mutants to see if it will give a different answer than the base expression for at least one of the tests. If it does, then the mutant is considered “killed,” and MCDC is adequate for this error. The complication comes about by the presence of the third variable (C). This variable can take on values that are independent of the assignments to A and B. This is accomplished by adding these values into the truth table. Table D-24 shows the expanded truth table for the base expression and the two mutants.

TABLE D-24. BASE AND MUTANT EXPRESSION RESPONSES
(EXPANDED TRUTH TABLE)

	0		1		2		3		
	0	1	2	3	4	5	6	7	
A:	F	F	F	F	T	T	T	T	
B:	F	F	T	T	F	F	T	T	
C:	F	T	F	T	F	T	F	T	
	F	F	F	F	F	F	T	T	A and B
	F	F	F	t	F	F	f	T	*C* and B
	F	F	F	F	F	t	f	T	A and *C*

In table D-24, the first row gives the condition code number for (A, B) while the second row gives the condition code numbers for (A, B, C). Notice that one-condition combination in (A, B) expands into two-condition combinations in (A, B, C). Rows three through five give the values for the conditions (A, B, C) in the condition combinations. Rows six through eight give the responses of the three expressions. Upper-case responses show where the mutants agree with the base expression, while lower-case responses show where the mutants disagree with the base expression. The first column is used to identify which conditions are represented in the corresponding row. The second through ninth columns represent the condition combinations and the responses from the base expression and its two mutants. The tenth column identifies which expression has the response profile of the corresponding row.

Examination of table D-24 shows that condition combinations (3, 6) in (A, B, C) will distinguish the first mutant from the base expression while condition combinations (5, 6) will distinguish the second mutant from the base expression. The MCDC test set for *A and B* is (1, 2, 3) in (A, B).

When a third condition (C) was added, each condition combination in (A,B) expands to two condition combinations in (A,B,C). This expansion results in eight test sets for MCDC of the base expression.

In table D-25, the identification of the MCDC test set has been added in the first column. The response profiles given by the two mutants are given in the corresponding row. Where one or more mutants disagree with the response from the base expression, an “x” is given in the corresponding column if both mutants are killed by the test set. If a mutant is not killed by an MCDC test set, it is listed in the final column.

Examination of table D-25 shows that of the eight test sets possible for the base expression, five of them (62.5%) are sensitive to the error of a single outside variable replacement. When this analysis is carried out against all of the two-condition expressions, there results a total of 384 test sets, 264 of which (68.75%) are sensitive to the error of a single outside variable replacement.

TABLE D-25. MCDC TEST SETS VS MUTANTS (EXPANDED TRUTH TABLE)

	0		1		2		3		
	0	1	2	3	4	5	6	7	
A:	F	F	F	F	T	T	T	T	
B:	F	F	T	T	F	F	T	T	
C:	F	T	F	T	F	T	F	T	
(2,4,6)			F		F		x		
(2,4,7)			F		F			T	C and B A and C
(2,5,6)			F			x	x		
(2,5,7)			F			F		T	C and B
(3,4,6)				x	F		x		
(3,4,7)				F	F			T	A and C
(3,5,6)				x		x	x		
(3,5,7)				x		x		T	

A similar analysis was carried out for two outside variable replacements (C, D) against the expression *A and B*. This expanded each of the condition codes in (A, B) to four-condition codes in (A, B, C, D). This expanded the single MCDC test set for (A, B) in two variables to 64 test sets in four variables. Of these, 25 (39.06%) are sensitive to the error of two outside variables replacement. This trend suggests that further investigation is warranted. In particular, this trend apparently goes against the findings of the "sufficient mutation operators" experiments.

D.8. References

1. DeMillo, R.A., Lipton, R.J., Sayward, F.G., "Hints on Test Data Selection: Help for the Practicing Programmer," IEEE Computer, Vol. 11, No. 4, April 1978, pp. 34-41.

2. Voas, J.M., McGraw, G., "Software Fault Injection," John Wiley & Sons, New York, 1998.

3. Friedman, M.A., Voas, J.M., "Software Assessment," John Wiley & Sons, New York, 1995.

4. Agrawal, H., DeMillo, R., Hathaway, R., Hsu, Wm., Hsu, W., Krauser, E., Martin, R.J., Mathur, A., Spafford, E., "Design of Mutant Operators for the C Programming Language," Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, 1989.

5. Offutt, A.J., Voas, J., Payne, J., "Mutation Operators for Ada," Technical Report ISSE-TR-96-09, Department of ISSE, George Mason University, 1996.

6. Offutt, A.J., "Investigations of the Software Testing Coupling Effect," ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 1, January 1992, pp. 5-20.

7. Offutt, A.J., "The Coupling Effect: Fact or Fiction?," Proceedings of the Third Symposium on Software Testing, Analysis and Verification (ACM SIGSOFT 89), 1989, pp. 131-140.

8. Offutt, A.J., Voas, J.M., "Subsumption of Condition Coverage Techniques by Mutation Testing," Technical Report ISSE-TR-96-01 Draft, Department of ISSE, George Mason University, 1996.

9. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C., "An Experimental Determination of Sufficient Mutant Operators," ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 2, April 1996, pp. 99-118.

10. Offutt, A.J., Rothermel, G., Zapf, C., "An Experimental Evaluation of Selective Mutation," Proceedings of the 15th International Conference on Software Engineering, 1993, pp. 100-107.