NASA/TM-2001-210876

A Practical Tutorial on Modified Condition/ Decision Coverage

Kelly J. Hayhurst
Langley Research Center, Hampton, Virginia

Dan S. Veerhusen
Rockwell Collins, Inc., Cedar Rapids, Iowa

John J. Chilenski
The Boeing Company, Seattle, Washington

Leanna K. Rierson
Federal Aviation Administration, Washington, D.C.

May 2001

The NASA STI Program Office … in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at (301) 621-0134

- Phone the NASA STI Help Desk at (301) 621-0390

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076-1320

NASA/TM-2001-210876

A Practical Tutorial on Modified Condition/ Decision Coverage

Kelly J. Hayhurst
Langley Research Center, Hampton, Virginia

Dan S. Veerhusen
Rockwell Collins, Inc., Cedar Rapids, Iowa

John J. Chilenski
The Boeing Company, Seattle, Washington

Leanna K. Rierson
Federal Aviation Administration, Washington, D. C.

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

May 2001

Available from:

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 605-6000

Table of Contents

1 BACKGROUND AND PURPOSE 1

1.1 SCOPE OF THE TUTORIAL 1
1.2 NOTES ON DEFINITIONS AND NOTATION 2

2 MC/DC APOLOGIA 2

2.1 THE VERIFICATION PROCESS 3
2.2 DEFINITION AND ROLE OF COVERAGE 4
2.2.1 Requirements Coverage Analysis 5
2.2.2 Structural Coverage Analysis 6
2.3 TYPES OF STRUCTURAL COVERAGE 7
2.3.1 Statement Coverage 8
2.3.2 Decision Coverage 8
2.3.3 Condition Coverage 8
2.3.4 Condition/Decision Coverage 9
2.3.5 Modified Condition/Decision Coverage 9
2.3.6 Multiple Condition Coverage 9
2.4 STRUCTURAL COVERAGE VERSUS STRUCTURAL TESTING 9
2.5 MORE DETAILS ABOUT MODIFIED CONDITION/DECISION COVERAGE 10
2.5.1 A Note on Source versus Object Code Coverage 11
2.6 CONFOUNDING ISSUES 12

3 MC/DC APPROACH 14

3.1 MC/DC BUILDING BLOCKS ("HOW DO I TEST A …?") 15
3.1.1 and Gate 16
3.1.2 or Gate 16
3.1.3 xor Gate 17
3.1.4 not Gate 17
3.1.5 Comparator 18
3.1.6 If-then-else Statement 19
3.1.7 Loop Statements 20
3.2 EVALUATION METHOD 22
3.2.1 Source Code Representation 23
3.2.2 Identification of Test Inputs 25
3.2.3 Elimination of Masked Tests 25
3.2.4 Determination of MC/DC 28
3.2.5 Output Confirmation 29
3.3 FOLLOWING THE FIVE-STEP EVALUATION PROCESS 29
3.4 GROUPED FUNCTIONALITY 36
3.5 DEVELOPING COMPLEX CONSTRUCTS 42
3.6 MC/DC WITH SHORT CIRCUIT LOGIC 42
3.7 MC/DC WITH BIT-WISE OPERATIONS 45
3.7.1 Examples with Bit-Wise Operations 45
3.7.2 Alternate Treatments of Bit-wise Operations 47
3.8 ANALYSIS RESOLUTION 47

4 MC/DC ADMONITIONS 49

4.1 AUTOMATION PROBLEMS 49
4.1.1 How Coverage Analysis Tools Work 50
4.1.2 Factors to Consider in Selecting or Qualifying a Tool 50
4.2 PROCESS PROBLEMS 54
4.2.1 Inadequate Planning for MC/DC 54
4.2.2 Misunderstanding the MC/DC Objective 55
4.2.3 Inefficient Testing Strategies 55
4.2.4 Poor Management of Verification Resources 55

5 ASSESSMENT PROCESS **56**
STEP 1—REVIEW VERIFICATION PLANS 57
STEP 2—DETERMINE NEED FOR TOOL QUALIFICATION 58
STEP 3—REVIEW DATA RELATED TO QUALIFICATION OF MC/DC TOOLS 58
STEP 4—REVIEW TEST CASES AND PROCEDURES 59
STEP 5—REVIEW CHECKLISTS FOR TEST CASES, PROCEDURES, AND RESULTS 60
STEP 6—DETERMINE EFFECTIVENESS OF TEST PROGRAM 60
Task 6.1—Assess results of requirements-based tests *61*
Task 6.2—Assess failure explanations and rework *61*
Task 6.3—Assess coverage achievement *61*

6 SUMMARY **62**

7 REFERENCES **63**

APPENDIX A **65**

SOLUTIONS TO EXERCISES **65**
Solution 2.5a *65*
Solution 2.5b *65*
Solution 3.3a, OR/XOR Exercise *65*
Solution 3.3b, Ground Test Exercise *67*
Solution 3.3c, Weight on Wheels Exercise *68*
Solution 3.3d, Gain Exercise *69*
Solution 3.5, Reset-Overides-Set Latch Exercise *70*

APPENDIX B **72**

CERTIFICATION AUTHORITIES SOFTWARE TEAM POSITION PAPER ON MASKING MC/DC **72**

APPENDIX C **78**

BACKGROUND ON TUTORIAL AUTHORS **78**

Abstract

This tutorial provides a practical approach to assessing modified condition/decision coverage (MC/DC) for aviation software products that must comply with regulatory guidance for DO-178B level A software. The tutorial's approach to MC/DC is a 5-step process that allows a certification authority or verification analyst to evaluate MC/DC claims without the aid of a coverage tool. In addition to the MC/DC approach, the tutorial addresses factors to consider in selecting and qualifying a structural coverage analysis tool, tips for reviewing life cycle data related to MC/DC, and pitfalls common to structural coverage analysis.

1 Background and Purpose

The RTCA/DO-178B document *Software Considerations in Airborne Systems and Equipment Certification* is the primary means used by aviation software developers to obtain Federal Aviation Administration (FAA) approval[1] of airborne computer software (ref. 1, 2). DO-178B describes software life cycle activities and design considerations, and enumerates sets of objectives for the software life cycle processes. The objectives applicable to a given piece of software are based on the software level determined by a system safety assessment. The objectives serve as a focal point for approval of the software.

This tutorial concerns one particular objective in DO-178B: objective 5 in Table A-7 of Annex A. This objective, which is applicable to level A software only, requires that tests achieve modified condition/decision coverage (MC/DC) of the software structure. The purpose of the tutorial is to provide sufficient information upon which a diligent person may build a strong working knowledge of how to meet the MC/DC objective, and provide a means to assess whether the objective has been met[2].

1.1 Scope of the Tutorial

This tutorial provides a broad view of MC/DC, concentrating on practical information for software engineers. Topics include the role of MC/DC within the verification process described in DO-178B, the rationale for the MC/DC objective, a pragmatic approach for manually evaluating MC/DC, and an aid for assessing an applicant's MC/DC program. Although understanding the rationale for MC/DC is not strictly necessary for developing a working knowledge of its use, information is included to help reduce current well-documented misunderstandings about the topic (ref. 3).

The tutorial is a self-study course designed for individuals who either develop and verify aviation software products that must comply with the DO-178B objectives for level A, or who provide oversight and assurance of such products. Readers are assumed to have a basic knowledge of Boolean algebra and DO-178B. Specific references to DO-178B and other supporting materials are cited throughout. Also included throughout are exercises designed to give readers structured opportunities to assess their understanding of the concepts presented; solutions for the exercises are given in Appendix A. Readers

---

[1] ED-12B, the European equivalent of DO-178B, is recognized by the Joint Aviation Authorities (JAA) via JAA temporary guidance leaflet #4 as the primary means for obtaining approval of airborne computer software.

[2] This work was supported by the FAA William J. Hughes Technical Center, Atlantic City International Airport, New Jersey.

are encouraged to complete the exercises as they are encountered in the text to obtain maximum benefit from the tutorial.

After successful completion of this tutorial, readers should be able to:

- Explain the rationale for MC/DC
- Assess whether specific test cases meet the MC/DC criteria
- Ask informed questions about structural coverage analysis tools
- Determine if the integral processes support compliance with the MC/DC objective
- Avoid common problems associated with MC/DC

Lest anyone think otherwise, please recognize that this tutorial does not constitute regulatory software policy or guidance. Furthermore, the approach to MC/DC presented in this tutorial is just one possible approach to assessing compliance with the MC/DC objective and, as such, should not be considered the *only* means of complying with the objective. We have tried to clearly identify all places where the provided information goes beyond what is explicitly stated in DO-178B.

Chapter 2 provides the rationale for MC/DC. A manual approach to evaluating MC/DC is presented in chapter 3. Chapter 4 provides a collection of ways to help mitigate problems associated with MC/DC, including problems in selecting and qualifying a structural coverage analysis tool and process problems. Finally, chapter 5 provides information on determining an applicant's compliance with the MC/DC objective. For further information on relevant research and theoretical aspects of MC/DC see *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion* (ref. 4).

1.2 Notes on definitions and notation

The definitions and descriptions of terms used in this tutorial are consistent with those given in the Glossary of DO-178B, unless noted otherwise. The following notational conventions are used throughout the tutorial:

Boolean operators are denoted by bolded italics: ***and***, ***or***, ***xor, not***

Boolean conditions are denoted by bolded capital letters: **A**, **B**, **C**, …

Non-Boolean variables are denoted in plain lower case letters: x, y, z, …

Boolean outcomes are written as either *false* or *true*, or *F* or *T*

A test case for a Boolean function with n inputs is denoted by $C = (C_1C_2\ldots C_n)$, where $C_i=F$ or T

Where graphical representations are introduced in the text, the relevant symbols are defined when they are introduced. Code segments used as examples are written in Ada unless noted otherwise.

2 MC/DC Apologia

This chapter explains the context in which the MC/DC objective exists, and presents a rationale for the objective. Readers whose sole interest is in learning how to achieve or assess compliance may wish to skim this chapter.

2.1 The Verification Process

> *"No product of human intellect comes out right the first time. We rewrite sentences, rip out knitting stitches, replant gardens, remodel houses, and repair bridges. Why should software be any different?" (ref. 5)*

According to DO-178B[3], the purpose of the verification process is to detect and report errors that have been introduced in the development processes. The verification process does not produce software; its responsibility is to ensure that the produced software implements intended function completely and correctly, while avoiding unintended function. Because each development process may introduce errors, verification is an integral process (see Figure 1), which is coupled with every development process. Including verification activities with each development activity is intended to help "build in" quality at each step, because "testing or analyzing in" quality at the end of the life cycle is impractical.

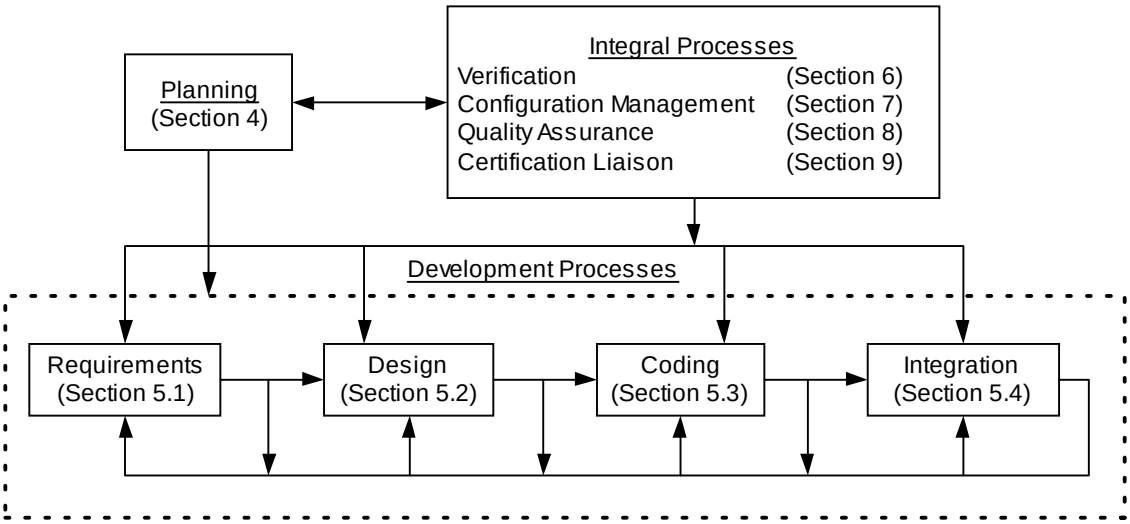


Figure 1. DO-178B software life cycle processes.[4]

Verification examines the relationship between the software product and the requirements. At an abstract level, the software verification process exists to constantly ask the question: are we building the system right?

Verification can be expensive and time consuming, even for software that is not safety critical. To date, no one has been able to define objective test measures of software quality. That is, typical statistical approaches to quality assurance, which work well for physical devices, do not apply to software. Consequently, drawing conclusions about software quality short of testing every possible input to the program is fraught with danger. This fact contributes to the current reliance on structural coverage as one measure of the completeness of testing.

---

[3] This is the last time the phrase "According to DO-178B" or its variants will appear in this document. The reader should simply assume its presence everywhere.

[4] This figure is based on a similar Life Cycle Diagram in RTCA/DO-254 *Design Assurance Guidance for Airborne Electronic Hardware* (ref. 6).

2.2 Definition and Role of Coverage

> *"Our goal, then, should be to provide enough testing to ensure that the probability of failure due to hibernating bugs is low enough to accept. 'Enough' implies judgement." (ref. 7)*

Coverage refers to the extent to which a given verification activity has satisfied its objectives. Coverage measures can be applied to any verification activity, although they are most frequently applied to testing activities. Appropriate coverage measures give the people doing, managing, and auditing verification activities a sense of the adequacy of the verification accomplished; in essence, providing an exit criteria for when to stop. That is, what is "enough" is defined in terms of coverage.

Coverage is a measure, not a method or a test. Thus, phrases such as "MC/DC testing" can do more harm than good[5]. As a measure, coverage is usually expressed as the percentage of an activity that is accomplished. Two specific measures of test coverage are shown in Figure 2 (ref. 2): requirements coverage and software structure coverage (to be consistent with common usage, we will use the phrase *structural coverage* hereafter). Requirements coverage analysis determines how well the requirements-based testing verified the implementation of the software requirements (DO-178B, section 6.4.4.1), and establishes traceability between the software requirements and the test cases (DO-178B, section 6.2). Structural coverage analysis determines how much of the code structure was executed by the requirements-based tests (DO-178B, section 6.4.4.2), and establishes traceability between the code structure and the test cases (DO-178B, section 6.2). Please note that requirements coverage analysis precedes structural coverage analysis.

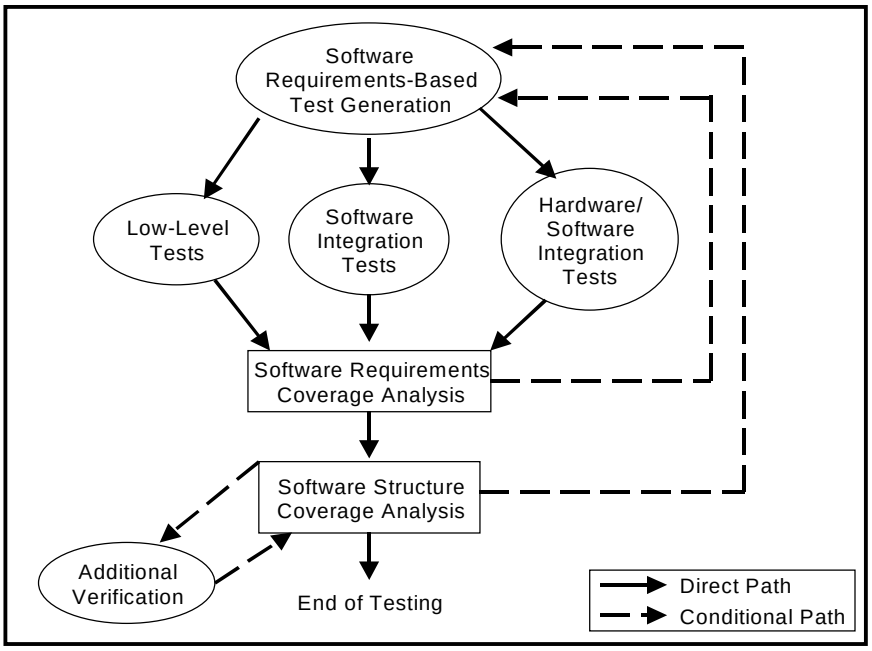


Figure 2. DO-178B software testing activities.

[5] The terms "black-box testing" and "white-box testing", which are used extensively in software engineering literature, also may be misleading. These terms (especially "white-box testing") tend to obscure the necessary connection to requirements. These terms do not appear in DO-178B; neither do they appear in this tutorial. The distinction between *coverage* and *testing* will be further discussed in section 2.4.

2.2.1 Requirements Coverage Analysis

According to software engineering theory, software requirements should contain a finite list of behaviors and features, and each requirement should be written to be verifiable. Testing based on requirements is appealing because it is done from the perspective of the user (thus providing a demonstration of intended function), and allows for development of test plans and cases concurrently with development of the requirements. Given a finite list of requirements and a set of completion criteria, requirements-based testing becomes a feasible process, unlike exhaustive testing (ref. 5).

The Annex A objectives for requirements-based test coverage are stated with respect to both high- and low-level requirements. Objective 3 in Table A-7 requires test coverage of high-level requirements for software levels A-D; and objective 4 in Table A-7 requires test coverage of low-level requirements for software levels A-C. The corresponding guidance in section 6.4.4.1 states that the test coverage analysis should show that test cases exist for each software requirement, and that the test cases satisfy the criteria for normal and robustness testing.

Unfortunately, a test set that meets requirements coverage is not necessarily a thorough test of the software, for several reasons:

- the software requirements and the design description (used as the basis for the test set) may not contain a complete and accurate specification of all the behavior represented in the executable code;
- the software requirements may not be written with sufficient granularity to assure that all the functional behaviors implemented in the source code are tested; and,
- requirements-based testing *alone* cannot confirm that the code does not include unintended functionality.

"[D]uring the development of any non-trivial program, software structure is almost always created that cannot be determined from top-level software specifications" (ref. 8). Derived requirements, as described in DO-178B, were instituted for this reason, and should be tested as part of requirements-based testing. If the derived requirements are not documented appropriately, however, there will likely be no requirements-based tests for them; and, consequently, requirements coverage analysis has no documentation basis from which to say that the requirements-based tests are insufficient. The software structure or implementation detail, which is ideally documented as derived requirements, demands structural coverage analysis.

Different engineers may generate different, yet functionally equivalent, low-level requirements from the same set of high-level requirements. Likewise, different engineers may generate different, yet functionally equivalent, source code from the same set of low-level requirements. For example, a low-level requirement to assign to x twice the input value y may be coded as x := 2 * y; x := y + y; or x := y / 0.5. Logical low-level requirements may likewise be implemented in a different yet equivalent manner. For example, a low-level requirement to monitor a stop light could be implemented as **Red_On** := **Red_Light**; or as **Red_On** := ***not*** **Green_Light** ***and not*** **Yellow_Light**. The designer of the low-level requirements and the person testing the low-level requirements do not necessarily know the source code implementation generated. Thus structural coverage analysis is required to assure that the as-implemented code structure has been adequately tested and does not contain any unintended functionality. For a detailed example of the impact of code structure on coverage, see Chilenski's analysis of various

implementations of the type-of-triangle problem[6] in *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion* (ref. 4).

2.2.2 Structural Coverage Analysis

Structural coverage analysis provides a means to confirm that "the requirements-based test procedures exercised the code structure" (DO-178B, section 6.4.4). Recall that in the flow of testing activities, requirements coverage will have been accomplished and reviewed before structural coverage analysis begins. The subsequent structural coverage analysis reveals what source code structure has been executed with the requirements-based test cases. The RTCA/DO-248A[7] document *Second Annual Report for Clarification of DO-178B "Software Considerations in Airborne Systems and Equipment Certification"* (ref. 9) explains the intent of structural coverage analysis in the response to the Frequently Asked Question (FAQ #43) "What is the intent of structural coverage analysis?":

> Sections 6.4.4.2 and 6.4.4.3 of DO-178B/ED-12B define the purpose of structural coverage analysis and the possible resolution for code structure that was not exercised during requirements-based testing.
>
> The purpose of structural coverage analysis with the associated structural coverage analysis resolution is to complement requirements-based testing as follows:
>
> 1. Provide evidence that the code structure was verified to the degree required for the applicable software level;
> 2. Provide a means to support demonstration of absence of unintended functions;
> 3. Establish the thoroughness of requirements-based testing.
>
> With respect to intended function, evidence that testing was rigorous and complete is provided by the combination of requirements-based testing (both normal range testing and robustness testing) and requirements-based test coverage analysis.
>
> When drafting DO-178B/ED-12B, it was realized that requirements-based testing cannot completely provide this kind of evidence with respect to unintended functions. Code that is implemented without being linked to requirements may not be exercised by requirements-based tests. Such code could result in unintended functions. Therefore, something additional should be done since unintended functions could affect safety. A technically feasible solution was found in structural coverage analysis.
>
> The rationale is that if requirements-based testing proves that all intended functions are properly implemented, and if structural coverage analysis demonstrates that all existing code is reachable and adequately tested, these two together provide a greater level of confidence that there are no unintended functions. Structural coverage analysis will:
>
> - Indicate to what extent the requirements-based test procedures exercise the code structure; and
> - Reveal code structure that was not exercised during testing.
>
> <u>Note 1</u>: In the above text, the term "exercised during requirements-based testing" does not only mean that the specific code was exercised. It also means that the behavior of the code has been compared with the requirements to which it traces.

---

[6] The type-of triangle problem is taken from Glenford Myers' (ref. 10) classic example of requirements for determining whether a triangle is scalene, equilateral, or isosceles based on the lengths of the sides of the triangle.

[7] Information from RTCA/DO-248A is quoted throughout the tutorial with permission from the RTCA.

Note 2: Other papers on structural coverage analysis and the link between DO-178B/ED-12B and the FAR/JAR's may be found by using keyword index in Appendix C [of DO-248A].

The amount of code structure that has been exercised can be measured by different criteria. Several of these structural coverage criteria are discussed briefly in the next section.

2.3 Types of Structural Coverage

Typically structural coverage criteria are divided into two types: data flow and control flow. Data flow criteria measure the flow of data between variable assignments and references to the variables. Data flow metrics, such as all-definitions and all-uses (ref. 7), involve analysis of the paths (or subpaths) between the definition of a variable and its subsequent use. Because the DO-178B objectives for test coverage of software structure do not include explicit data flow criteria, the following discussion focuses on control flow.

Control flow criteria measure the flow of control between statements and sequences of statements. The structural coverage criteria in many standards, including DO-178B, are often control flow criteria. For control flow criteria, the degree of structural coverage achieved is measured in terms of statement invocations, Boolean expressions evaluated, and control constructs exercised. Table 1 gives the definitions of some common structural coverage measures based on control flow. A dot (•) indicates the criteria that applies to each type of coverage.

Table 1. Types of Structural Coverage

Coverage Criteria	Statement Coverage	Decision Coverage	Condition Coverage	Condition/ Decision Coverage	MC/DC	Multiple Condition Coverage
Every point of entry and exit in the program has been invoked at least once		•	•	•	•	•
Every statement in the program has been invoked at least once	•					
Every decision in the program has taken all possible outcomes at least once		•		•	•	•
Every condition in a decision in the program has taken all possible outcomes at least once			•	•	•	•
Every condition in a decision has been shown to independently affect that decision's outcome					•	•[8]
Every combination of condition outcomes within a decision has been invoked at least once						•

[8] Multiple condition coverage does not explicitly require showing the independent effect of each condition. This will be done, in most cases, by showing that every combination of decision inputs has been invoked. Note, however, that logical expressions exist wherein every condition cannot have an independent effect.

Three of the measures in Table 1 are found in objectives for test coverage given in DO-178B Table A-7 of Annex A[9]:

- objective 7 requires statement coverage for software levels A-C
- objective 6 requires decision coverage for software levels A-B
- objective 5 requires MC/DC for software level A

The structural coverage measures in Table 1 range in order from the weakest, statement coverage, to the strongest, multiple condition coverage. A brief description of each of the structural coverage measures in Table 1 is given in the following sections. For a more detailed description of each of these measures, see *The Art of Software Testing* (ref. 10).

2.3.1 Statement Coverage

To achieve statement coverage, every executable statement in the program is invoked at least once during software testing. Achieving statement coverage shows that all code statements are reachable (in the context of DO-178B, reachable based on test cases developed from the requirements). Statement coverage is considered a weak criterion because it is insensitive to some control structures (ref. 11). Consider the following code segment (ref. 12):

```
if  (x > 1) and (y = 0) then z := z / x; end if;
if  (z = 2) or (y > 1) then z := z + 1; end if;
```

By choosing x = 2, y = 0, and z = 4 as input to this code segment, every statement is executed at least once. However, if an ***or*** is coded by mistake in the first statement instead of an ***and***, the test case will not detect a problem. This makes sense because analysis of logic expressions is not part of the statement coverage criterion. According to Myers (ref. 10), "statement-coverage criterion is so weak that it is generally considered useless." At best, statement coverage should be considered a minimal requirement.

The remaining measures in Table 1 consider various aspects of decision logic as part of their criteria. To highlight differences between these measures, we will refer to the decision (**A *or* B**), where **A** and **B** are both conditions.

2.3.2 Decision Coverage

Decision coverage requires two test cases: one for a *true* outcome and another for a *false* outcome. For simple decisions (i.e., decisions with a single condition), decision coverage ensures complete testing of control constructs. But, not all decisions are simple. For the decision (**A *or* B**), test cases (*TF*) and (*FF*) will toggle the decision outcome between *true* and *false*. However, the effect of **B** is not tested; that is, those test cases cannot distinguish between the decision (**A *or* B**) and the decision **A**.

2.3.3 Condition Coverage

Condition coverage requires that each condition in a decision take on all possible outcomes at least once (to overcome the problem in the previous example), but does not require that the decision take on all

[9] There are actually four objectives in Table A-7 for test coverage of software structure. Objective 8, requiring data coupling and control coupling for software levels A-C, is not addressed in this tutorial; but, is mentioned here for completeness.

possible outcomes at least once. In this case, for the decision (**A *or* B**) test cases (*TF*) and (*FT*) meet the coverage criterion, but do not cause the decision to take on all possible outcomes. As with decision coverage, a minimum of two tests cases is required for each decision.

2.3.4 Condition/Decision Coverage

Condition/decision coverage combines the requirements for decision coverage with those for condition coverage. That is, there must be sufficient test cases to toggle the decision outcome between *true* and *false* and to toggle each condition value between *true* and *false*. Hence, a minimum of two test cases are necessary for each decision. Using the example (**A *or* B**), test cases (*TT*) and (*FF*) would meet the coverage requirement. However, these two tests do not distinguish the correct expression (**A *or* B**) from the expression **A** or from the expression **B** or from the expression (**A *and* B**).

2.3.5 Modified Condition/Decision Coverage

The MC/DC criterion enhances the condition/decision coverage criterion by requiring that each condition be shown to independently affect the outcome of the decision. The independence requirement ensures that the effect of each condition is tested relative to the other conditions. However, achieving MC/DC requires more thoughtful selection of the test cases, as will be discussed further in chapter 3, and, in general, a minimum of n+1 test cases for a decision with n inputs. For the example (**A *or* B**), test cases (*TF*), (*FT*), and (*FF*) provide MC/DC. For decisions with a large number of inputs, MC/DC requires considerably more test cases than any of the coverage measures discussed above.

2.3.6 Multiple Condition Coverage

Finally, multiple condition coverage requires test cases that ensure each possible combination of inputs to a decision is executed at least once; that is, multiple condition coverage requires exhaustive testing of the input combinations to a decision. In theory, multiple condition coverage is the most desirable structural coverage measure; but, it is impractical for many cases. For a decision with n inputs, multiple condition coverage requires 2^n tests.[10]

2.4 Structural Coverage versus Structural Testing

The distinction between structural coverage analysis and structural (or structure-based) testing is often misunderstood. Some of the confusion stems from the misguided notion that coverage is a testing method. But, the confusion is undoubtedly fueled by language in DO-178A[11] specifically referring to structure-based testing. According to DO-248A Discussion Paper #3 (ref. 9), "Basically, DO-178A/ED-12A requires that structural testing is carried out but does not define explicitly what type of structural testing is acceptable, nor does it define the scope of structural testing required for the different levels of software." Neither structural testing nor structure-based testing is mentioned in DO-178B.

To clarify the difference between structural coverage analysis and structural testing, DO-248A contains the following FAQ in response to the question "Why is structural testing not a DO-178B/ED-12B requirement?" (ref. 9, FAQ #44):

---

[10] In the context of DO-178B, the number of inputs and the number of conditions in an expression can be different. For example, the expression (**A *and* B**) ***or*** (**A *and* C**) has three inputs, but four conditions, because each occurrence of **A** is considered a unique condition. The maximum number of possible test cases is always 2^n, where n is the number of inputs, not the number of conditions.

[11] DO-178A/ED-12A was the predecessor to DO-178B/ED-12B.

There is a distinction between structural coverage analysis and structural testing. The purpose of structural coverage analysis is to "*determine which code structure was not exercised by the requirements-based test procedures*"(reference DO-178B/ED-12B Section 6.4.4.2). Structural testing is the process of exercising software with test scenarios written from the source code, not from the requirements. Structural testing does not meet the DO-178B/ED-12B objective that all code structure is exercised by the requirements-based test procedures. The correct approach when structural coverage analysis identifies untested code is to consider the possible causes in accordance with DO-178B/ED-12B Section 6.4.4.3. If any additional testing is required, it should be requirements-based testing, using high-level, low-level, or derived requirements, as appropriate.

Structured[12] testing cannot find errors such as the non-implementation of some of the requirements. Since the starting point for developing structural test cases is the code itself, there is no way of finding requirements (high-level, low-level, or derived) not implemented in the code through structural tests. It is a natural tendency to consider outputs of the actual code (which is de facto the reference for structural testing) as the expected results. This bias cannot occur when expected outputs of a tested piece of code are determined by analysis of the requirements.

Since the code itself is used as the basis of the test cases, structural testing may fail to find simple coding errors.

Structural testing provides no information about whether the code is doing what it is *supposed* to be doing as specified in the requirements. With respect to control flow, structural testing does not provide any information as to whether the right decisions are being made for the right reasons. Finally, structural testing fails to assure that there are no unintended functions. In the best case, structural testing confirms that the object code and processor properly implement the source code.

2.5 More Details about Modified Condition/Decision Coverage

According to legend, there were once folks who advocated requiring 100% multiple condition coverage (that is, exhaustive testing) for level A software. The motivation was simple: testing all possible combinations of inputs for each decision ensures that the correct decision outcome is reached in all cases. The problem with such testing, however, is that for a decision with n inputs, 2^n tests are required. In cases where n is small, running 2^n tests may be reasonable; running 2^n tests for large n is impracticable.

In avionics systems, complex Boolean expressions are common. Table 2 shows the number of Boolean expressions with n conditions for all of the logic expressions taken from the airborne software (written in Ada) of five different Line Replaceable Units (LRUs) from level A systems (ref. 4). The five LRUs came from five different airborne systems from two different airplane models in 1995 (two from one model and three from the other). As Chilenski's data shows, actual code has been written with more than 36 conditions.

Table 2. Boolean Expression Profile for 5 Line Replaceable Units

	Number of Conditions, *n*									
	1	**2**	**3**	**4**	**5**	**6-10**	**11-15**	**16-20**	**21-35**	**36-76**
Number of Boolean expressions with *n* conditions	16491	2262	685	391	131	219	35	36	4	2

[12] Although the text of FAQ #44 that appears in DO-248A uses "Structured", the correct word for the FAQ (and the word intended by the subgroup that wrote the FAQ) is "Structural".

Clearly, multiple condition coverage is impractical for systems such as these. MC/DC attempts to provide a practical alternative. "The modified condition/decision coverage criterion was developed to achieve many of the benefits of multiple-condition testing while retaining the linear growth in required test cases of condition/decision testing. The essence of the modified condition/decision coverage criterion is that each condition must be shown to independently affect the outcome of this decision, i.e., one must demonstrate that the outcome of a decision changes as a result of changing a single condition." (ref. 13) MC/DC is intended to assure, with a high degree of confidence, that the verification process has shown that each condition in each decision in the source code has the proper effect.

Exercise 2.5a: Consider an expression with 36 inputs. How much time would it take to execute all of the test cases required for multiple condition coverage (exhaustive testing) of this expression if you could run 100 test cases per second?

Exercise 2.5b: If your test artifacts include a single line for the test results of each test case, how tall would the report be for test results for achieving multiple condition coverage for an expression with 36 inputs? (Assume 64 lines per sheet of paper, and 250 sheets of paper per inch height.)

2.5.1 A Note on Source versus Object Code Coverage

Structural coverage achieved at the source code level can differ from that achieved at the object code level. Depending on language and compiler features used, multiple object code statements can be generated from a single source code statement (ref. 8). According to Beizer, a test suite that provides 100% statement coverage at the source code level for a "good piece of logic-intensive modern code" might cover 75% or less of the statements at the object code level (ref. 7). Consequently, achieving MC/DC at the source code level does not guarantee MC/DC at the object code level, and vice versa.

For software levels A-C, structural coverage analysis may be performed on the source code (DO-178B, section 6.4.4.2b). For level A software, however, additional verification should be performed if the compiler generates object code not directly traceable to the source code statements. A common misconception exists that MC/DC must be performed on the object code if the compiler generates code that is not directly traceable to the source code. The additional verification should establish the correctness of the code sequences that are not directly traceable; that is, the requirement for additional analysis applies only to those object code segments that are not traceable. Issues related to source code to object code traceability are addressed in FAQ #41 and FAQ #42 in DO-248A and are being documented in FAA policy.

There has been debate as to whether structural coverage, MC/DC in particular, can be demonstrated by analyzing the object code in lieu of the source code. According to FAQ #42 in DO-248A (ref. 9), structural coverage, including MC/DC, can be demonstrated at the object code level…

> as long as analysis can be provided which demonstrates that the coverage analysis conducted at the object code will be equivalent to the same coverage analysis at the source code level. In fact, for Level A software coverage, DO-178B/ED-12B Section 6.4.4.2b states that if "*…the compiler generates object code that is not directly traceable to Source Code statements. Then, additional verification should be performed on the object code...*" This is often satisfied by analyzing the object code to ensure that it is directly traceable to the source code. Hence, DO-178B/ED-12B determines the conditions for analysis of the source code for structural coverage, and it does not prevent one from performing analysis directly on the object code.

The analysis necessary to establish that coverage achieved at the object code level is equivalent to achieving the same level of coverage at the source code is ***not*** trivial in the general case. In some cases, however, showing equivalence may be simplified by using short-circuit control forms. According to FAQ #42 (ref. 9), compiler features such as short-circuit evaluation of Boolean expressions can be used to simplify the analysis.

> "When utilizing compiler features to simplify analysis, one relies on the compiler to behave as expected. Therefore, one may need to qualify the compiler features being used as a verification tool. (See Section 12.2.2)."

Further information on source to object traceability is not included in this tutorial due to forthcoming policy from the FAA. However, further information on short-circuit control forms is presented in section 3.6.

2.6 Confounding Issues

The requirement to show the *independent* effect of each condition within a decision makes MC/DC unique among coverage criteria. Without any constraining definitions, determining whether a condition has independent effect might seem rather simple: a condition has independent effect when that condition *alone* determines the outcome of the decision. At first glance, this simple definition seems to be consistent with the intent of MC/DC. Whether the simple notion is truly consistent with DO-178B, however, requires knowing the full DO-178B definition[13] of MC/DC plus the definitions for *condition* and *decision* (ref. 2).

> *Condition–A Boolean expression containing no Boolean operators.*
>
> *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.*
>
> *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.*

These definitions raise a number of confounding issues when determining whether a set of test cases provides MC/DC.

The first issue involves the meaning of "condition". Without the last sentence in the definition of decision, most people would probably say that the decision (**A** ***and*** **B**) ***or*** (**A** ***and*** **C**), where **A, B**, and **C** are conditions set by the software, contains three conditions—**A, B,** and **C**. According to the last sentence of the definition, however, this decision contains four conditions: the first **A**, **B**, **C**, and the second **A**. The first occurrence of **A** is said to be *coupled* with the second occurrence of **A** because a change to one condition affects the other. According to the definition of MC/DC above, showing independent effect in this example requires, among other things, showing what happens when the value of the first **A** is held constant, while the value of the second **A** is toggled between *false* and *true*. This typically cannot be accomplished in any meaningful way.

[13] We are using the word "definition" loosely. A strong case can be made that the Glossary entries provide, at best, descriptions rather than definitions. Because the distinction between "definition" and "description" is probably not important to most readers of this tutorial, we ignore it everywhere except this footnote.

The next issue involves the scope of "within a decision". For example, consider the following code statements:

A:= B or C; (statement 1)
E:= A and D; (statement 2)

These two statements are logically equivalent to:

E:= (B or C) and D; (statement 3)

Statements 1, 2, and 3 all contain decisions, even though none of the statements are branch points such as an *if* statement. That is, a decision is not synonymous with a branch point. MC/DC applies to all decisions—not just those within a branch point.

Further, a test set that provides MC/DC for statements 1 and 2 individually will not necessarily provide MC/DC for statement 3. That is, if a complex decision statement is decomposed into a set of less complex (but logically equivalent) decision statements, providing MC/DC for the parts is not always equivalent to providing MC/DC for the whole. For the example above, tests (*TFT*), (*FTF*), and (*FFT*) for (**B**,**C**,**D**) provide MC/DC for statements 1 and 2 individually, but do not provide MC/DC for statement 3.

The final issue involves the concept of independent effect. Showing that a condition independently affects a decision's outcome by varying just that condition while holding all others fixed is commonly referred to as the unique-cause approach to MC/DC. This approach ensures that the effect of each condition is tested relative to the other conditions without requiring analysis of the logic of each decision (that is, if changing the value of a single condition causes the value of the decision outcome to change, then the single condition is assumed to be the cause for the change—no further analysis is needed). Historically, the unique-cause approach has often been the only acceptable means of showing the independent effect of a condition. The unique-cause approach cannot be applied, however, to decisions where there are repeated or strongly coupled conditions; e.g., (**A** ***and*** **B**) ***or*** (**A** ***and*** **C**).

The unique-cause approach commonly is taught by presenting a truth table for an expression; for example, the decision **Z**:= (**A** ***or*** **B**) ***and*** (**C** ***or*** **D**) shown in Table 3. In the truth table approach, test cases that provide MC/DC are selected by identifying pairs of rows where only one condition and the decision outcome change values between the two rows. In Table 3, the columns shaded in gray indicate the independence pairs for each condition. For example, test case 2 coupled with test case 10 together demonstrate the independent effect of **A**, because **A** is the only condition that has changed value along with the change in value of the outcome **Z**. Although the truth table is a simple approach to showing the independent effect of a condition, the truth table approach suffers from a number of limitations: (a) the truth table is unwieldy for large logical expressions; and, for a logical expression with n inputs, only n+1 of the 2^n rows are useful; (b) the truth table addresses only one logical expression at a time; and, (c) the truth table does not connect the inputs and outputs from the requirements-based tests with the source code structure.

The approach to MC/DC given in this tutorial differs from the traditional approach and mitigates many of the difficulties described above. The approach, presented in the next chapter, requires analysis of the logic of a decision to confirm independent effect of the conditions. This analysis (which goes beyond that required for the unique-cause approach) has the advantages of (a) allowing more test cases to meet the MC/DC criteria than unique cause (which may make confirming MC/DC easier), (b) applying to decisions with coupled conditions that frequently occur in avionics applications, and (c) having capability equivalent to the unique-cause approach to detect errors (see Appendix B).

Table 3. Truth Table Approach to MC/DC

	A	B	C	D		Z		A	B	C	D
1	*F*	*F*	*F*	*F*		*F*					
2	*F*	*F*	*F*	*T*		*F*		10	6		
3	*F*	*F*	*T*	*F*		*F*		11	7		
4	*F*	*F*	*T*	*T*		*F*		12	8		
5	*F*	*T*	*F*	*F*		*F*				7	6
6	*F*	*T*	*F*	*T*		*T*			2		5
7	*F*	*T*	*T*	*F*		*T*			3	5	
8	*F*	*T*	*T*	*T*		*T*			4		
9	*T*	*F*	*F*	*F*		*F*				11	10
10	*T*	*F*	*F*	*T*		*T*		2			9
11	*T*	*F*	*T*	*F*		*T*		3		9	
12	*T*	*F*	*T*	*T*		*T*		4			
13	*T*	*T*	*F*	*F*		*F*				15	14
14	*T*	*T*	*F*	*T*		*T*					13
15	*T*	*T*	*T*	*F*		*T*				13	
16	*T*	*T*	*T*	*T*		*T*					

3 MC/DC Approach

This chapter presents a practical approach based on gate-level representations of logic constructs for evaluating whether a given set of requirements-based test cases conforms with three of the four requirements for MC/DC[14]:

- every decision in the program has taken all possible outcomes at least once
- every condition in a decision in the program has taken all possible outcomes at least once
- every condition in a decision has been shown to independently affect that decision's outcome

The MC/DC approach for the tutorial was selected because it requires an explicit mapping of the requirements-based tests to the source code structure. This approach applies to any source code written regardless of whether it is in a high-level language such as Ada or in assembly language. Mapping requirements-based test cases to source code structure reinforces the notion that structural coverage analysis is a check on the adequacy of requirements-based tests for a given source code implementation. The approach also capitalizes on several concepts familiar to engineers, including a schematic representation of the source code (which allows you to see everything needed for assessing MC/DC on one page), and the hardware testing concepts of controllability and observability.

The MC/DC approach provides simple steps that allow a certification authority or verification analyst to evaluate MC/DC claims without the aid of a coverage tool. The steps can also be used to help confirm that a tool properly assesses MC/DC. The MC/DC approach in this chapter does not evaluate the correctness of existing requirements-based tests—it is assumed that these tests have been reviewed adequately for correctness and coverage of the requirements. The purpose of the approach, however, is to determine if existing requirements-based test cases provide the level of rigor required to achieve MC/DC

[14] The fourth requirement for meeting MC/DC, testing of entry and exit points, is common to all structural coverage measures, except for statement coverage, and, as such, is not critical to a discussion of MC/DC.

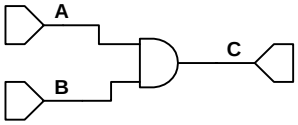
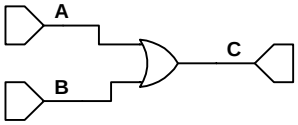
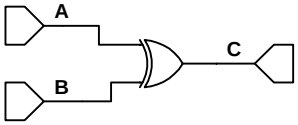
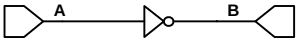
of the source code. **This approach is *not* intended to be the only method for determining compliance with the MC/DC objective, but rather a method to be added to your array of review techniques.**

This chapter is divided into eight sections. Section 3.1 discusses basic building blocks that are fundamental to the MC/DC approach. Section 3.2 presents the steps of the approach. Sections 3.3-3.7 address how to apply the approach to different scenarios, such as grouped functionality, short-circuit control forms, and bit-wise operations. Finally, resolving errors or shortcomings identified through the structural coverage analysis is discussed in section 3.8.

3.1 MC/DC Building Blocks ("how do I test a …?")

Understanding how to test individual logical operators, such as a logical expression with a single ***and*** operator, is essential to understanding MC/DC. In this tutorial, logical operators are shown schematically as logical gates; and, the terms "logical operator" and "gate" are used interchangeably. Table 4 shows the schematic representation of the elementary logical operators: ***and***, ***or***, ***xor***, and ***not***.

Table 4. Representations for Elementary Logical Expressions

Name	Schematic Representation	Code example	Truth Table
Input			
Output			
and Gate		**C** := **A *and* B**;	**A B C** *T T T* *T F F* *F T F* *F F F*
or Gate		**C** := **A *or* B**;	**A B C** *T T T* *T F T* *F T T* *F F F*
xor Gate		**C** := **A *xor* B**;	**A B C** *T T F* *T F T* *F T T* *F F F*
not Gate		**B** := ***not*** **A**;	**A B** *T F* *F T*

According to Chilenski and Miller, showing that each logical condition independently affects a decision's outcome requires specific minimal test sets for each logical operator (ref. 13). Knowing the minimal test sets for each logical operator provides the basis for determining compliance with the MC/DC objective. Here, the ***and*** gate, ***or*** gate, ***xor*** gate, and the ***not*** gate are considered to be basic constructs. Given the test requirements for these basic constructs, more complex constructs containing Boolean expressions can be examined, including a comparator, *if* statement, and *loop* statements. Minimum testing requirements and a tabular example for each of these constructs are described below.

3.1.1 and Gate

Minimum testing to achieve MC/DC for an ***and*** gate requires the following:

(1) All inputs are set *true* with the output observed to be *true*. This requires one test case for each *n*-input ***and*** gate.

(2) Each and every input is set exclusively *false* with the output observed to be *false*. This requires *n* test cases for each *n*-input ***and*** gate.

The requirements make sense when considering how an ***and*** gate works. Changing a single condition starting from a state where all inputs are *true* will change the outcome; that is, an ***and*** gate is sensitive to any *false* input. Hence, a specific set of *n*+1 test cases is needed for an *n*-input ***and*** gate. These specific *n*+1 test cases meet the intent of MC/DC by demonstrating that the ***and*** gate is correctly implemented.

An example of the minimum testing required for a three-input ***and*** gate (shown in Figure 3) is given in Table 5. In this example, test case 1 in Table 5 provides the coverage for (1) above, and test cases 2-4 provide coverage for (2).

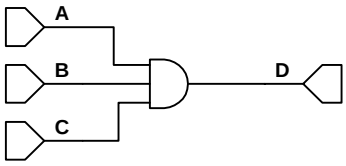


Figure 3. Three-input ***and*** gate.

Table 5. Minimum Tests for Three-input ***and*** Gate

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

3.1.2 or Gate

Minimum testing to achieve MC/DC for an ***or*** gate requires the following:

(1) All inputs are set *false* with the output observed to be *false*. This requires one test case for each *n*-input ***or*** gate.

(2) Each and every input is set exclusively *true* with the output observed to be *true*. This requires *n* test cases for each *n*-input ***or*** gate.

These requirements are based on an ***or*** gate's sensitivity to a *true* input. Here again, *n*+1 specific test cases are needed to test an *n*-input ***or*** gate. These specific *n*+1 test cases meet the intent of MC/DC by demonstrating that the ***or*** gate is correctly implemented.

An example of the minimum testing required for a three-input ***or*** gate (shown in Figure 4) is given in Table 6. In this example, test case 1 provides the coverage for (1) while test cases 2-4 provide the coverage for (2).

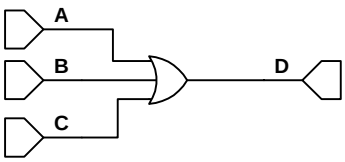


Figure 4. Three-input ***or*** gate.

Table 6. Minimum Tests for a Three-input ***or*** Gate

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

3.1.3 xor Gate

The ***xor*** gate differs from both the ***and*** and the ***or*** gates with respect to MC/DC in that there are multiple minimum test sets for an ***xor***. Consider the two-input ***xor*** gate shown in Figure 5. All of the possible test cases for this ***xor*** gate are shown in Table 7. For a two-input ***xor*** gate, any combination of three test cases will provide MC/DC.

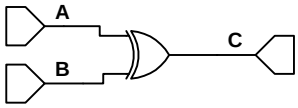


Figure 5. Two-input ***xor*** gate.

Table 7. Test Cases for a Two-input ***xor*** Gate

Test Case Number	1	2	3	4
Input **A**	*T*	*T*	*F*	*F*
Input **B**	*T*	*F*	*T*	*F*
Output **C**	*F*	*T*	*T*	*F*

Hence, minimum testing to meet the definition of MC/DC for a two-input ***xor*** gate requires one of the following sets of test cases from Table 7:

(1) test cases 1, 2, and 3
(2) test cases 1, 2, and 4
(3) test cases 1, 3, and 4
(4) test cases 2, 3, and 4

Note that for a test set to distinguish between an ***or*** and an ***xor*** gate it must contain test case 1 in Table 7. Hence, test sets 1, 2, and 3 above can detect when an ***or*** is coded incorrectly for an ***xor***, and vice versa. While not explicitly required by MC/DC, elimination of test set 4 as a valid test set is worth considering.

Note also that minimum tests to achieve MC/DC for an ***xor*** gate with more than two inputs are implementation dependent. Hence, no single set of rules applies universally to an ***xor*** gate with more than two inputs.

3.1.4 not Gate

The logical ***not*** works differently from the previous gates: the ***not*** works only on a single operand. That operand may be a single condition or a logical expression. But, with respect to a gate level representation, there is a single input to the ***not*** gate as shown in Figure 6.

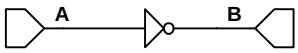


Figure 6. ***not*** gate.

Minimum testing to achieve MC/DC for a logical ***not*** requires the following:

(1) The input is set *false* with the output observed to be *true*.
(2) The input is set *true* with the output observed to be *false*.

3.1.5 Comparator

A comparator evaluates two numerical inputs and returns a Boolean based on the comparison criteria. Within the context of DO-178B, a comparator is a condition and also a simple decision. The following comparison criteria are considered in this tutorial:

- less than
- greater than
- less than or equal to
- greater than or equal to
- equal to
- not equal to

In general, the comparison point can be a constant or another variable (see Figure 7).

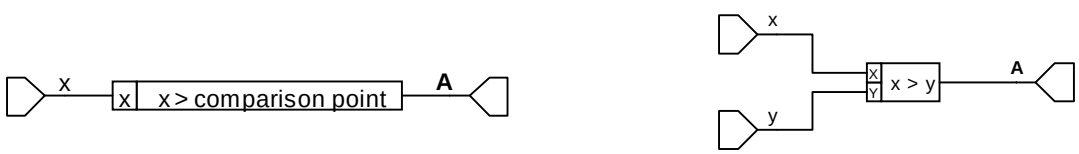


Figure 7. Two types of comparators.

In either case, two test cases will confirm MC/DC for a comparator—one test case with a *true* outcome, and one test case with a *false* outcome. Hence, minimum testing for a comparator requires the following:

(1) Input x set at a value above the comparison point (or y)
(2) Input x set at a value below the comparison point (or y)

However, the numerical aspects of a comparator must also be considered in determining reasonable tests. For example, given a software requirement to test ($x \leq 5000$), one test case with a *true* outcome (e.g., $x = -30000$), and one test case with a *false* outcome (e.g., $x = 30000$) provide MC/DC. However, these cases do not confirm that the design is accurately implemented in the source code. Specifically in this example, the test cases do not confirm that 5000 is the correct comparison point or less-than-or-equal-to is the appropriate relational operator. The source code could have been implemented as $x < -5000$ or as $x \leq 500$ and still pass the test cases.

Selecting two test cases closer to the comparison point is better, but does not cover certain coding errors. For example, test cases with $x = 5000$ and $x = 5001$ are better, but they will not detect a coding error of $x = 5000$.

In general, three test cases are needed to assure that simple coding errors have not been made; that is,

that the correct relational operator and comparison point are used in the code. So, while MC/DC only requires two tests, minimum *good* requirements-based testing for a comparator requires:

(1) Input x set at a value slightly above the comparison point

(2) Input x set at a value slightly below the comparison point

(3) Input x set at a value equal to the comparison point

The definition of "slightly" is determined by engineering judgement based on the numerical resolution of the target computer, the test equipment driving the inputs, and the resolution of the output device. Consider for example, the following set of test cases for a design that sets the output **A** *true* when altitude is greater than 2500 (see Figure 8 and Table 8).

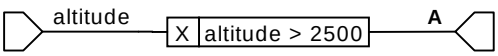


Figure 8. Comparator for altitude >2500.

Table 8. Test Cases for Comparator Example

Test Case Number	**1**	**2**	**3**	**4**	**5**
Input altitude	25	32000	2500	2499	2501
Output **A**	*F*	*T*	*F*	*F*	*T*

Test cases 1 and 2 give the desired MC/DC output. However, those test cases do not confirm that the toggle occurred at 2500, and not elsewhere. Even adding test case 3 does not improve the test suite much. The design could have been implemented with a comparison point anywhere between 2501 and 32000, and give the same result for test cases 1, 2, and 3. Test cases 3, 4, and 5 are a better set, because this set confirms that the transition occurs at 2500.

3.1.6 If-then-else Statement

The *if-then-else* statement is a switch that controls the execution of the software. Consider the following example where x, y, and z are integers and **C** is a Boolean:

if C then z := x else z := y;

Two different schematic representations of this code are shown in Figure 9.

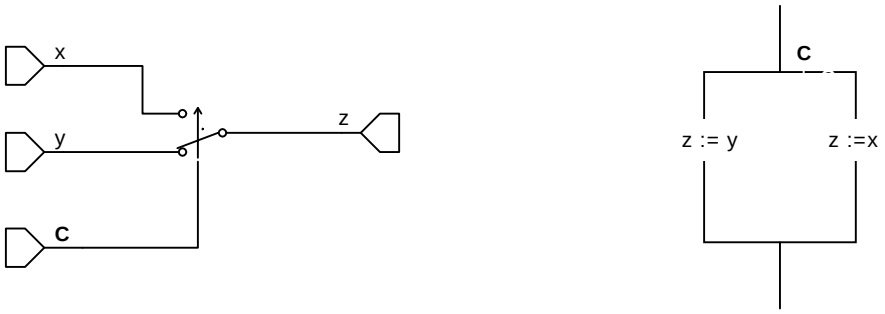


Figure 9. Two *if-then-else* constructs.

Minimum testing for the *if-then-else* statement requires the following:

(1) Inputs that force the execution of the *then* path (that is, the decision evaluates to *true*)

(2) Inputs that force the execution of the *else* path (that is, the decision evaluates to *false*). Note that the decision must evaluate to *false* with confirmation that the *then* path did not execute, even if there is no *else* path.

(3) Inputs to exercise any logical gates in the decision as required by sections 3.1.1-3.1.5

For example, for a single condition **Z**, the statement *if* ***Z*** *then…else…* requires only two test cases to achieve MC/DC. The decision in *if* ***X or Y or Z*** *then… else…* requires four test cases to achieve MC/DC. A minimal test set for the statement *if* ***Z*** *then a := x else a := y* is shown in Table 9.

Table 9. Minimum tests for *if Z then a := x else a := y*

Test Case Number	**1** **Traverse the *then* path**	**2** **Traverse the *else* path**
Input x	12	18
Input y	50	34
Input **Z**	*T*	*F*
Output a	12	34

Note that a *case* statement may be handled similarly to the *if-then-else* statement.

3.1.7 Loop Statements

Loop statements are constructs that cause a sequence of statements to be executed zero or more times. Constructs such as the *while loop* and the *for loop* are switches to control the execution of the software similar to an *if-then-else* construct. In the context of MC/DC, the challenge is to confirm that loops have been traversed appropriately.

While Loop

Consider the following example where **Weight_On_Wheels** is a Boolean:

```
while Weight_On_Wheels loop
    radar_mode := Off;
end loop;
```

A schematic representation of this code is shown in Figure 10. In this case, **Weight_On_Wheels** is the decision for the *while loop* construct.

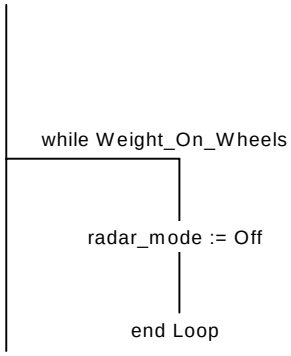


Figure 10. *while loop* construct.

Minimum testing for the *while loop* requires the following:

(1) Inputs to force the execution of the statements in the loop (that is, the decision evaluates to *true*)

(2) Inputs to force the exit of the loop (that is, the decision evaluates to *false*)

(3) Inputs to exercise any logical gates in the decision as required by sections 3.1.1-3.1.5

In Figure 10, two test cases may be used to achieve MC/DC. One test case confirms that radar_mode remains off as long as **Weight_On_Wheels** is *true*. The second test case confirms that radar_mode could be set to something other than off when **Weight_On_Wheels** is *false*. In the case where **Weight_On_Wheels** is replaced by a Boolean expression, the Boolean expression would also need to be evaluated, and the setting of radar_mode to off confirmed, by the methods previously described in sections 3.1.1-3.1.5.

Exit when

Not all decisions must appear at the start of a loop. The *exit when* statement can be used anywhere within a *loop* construct or a *for loop* to terminate the execution of the loop.[15] Consider the following example of a *loop* statement with an *exit when* condition. In this example, **Current_Signal** = *false* is the decision (see Figure 11).

```
loop
    get (Current_Signal);
    exit when Current_Signal = false;
end loop;
```

Minimum testing for the *exit when* statement requires the following:

(1) Inputs to force the repeated execution of the statements in the loop when the decision for the *exit when* evaluates to *false*

[15] The Ada language permits multiple *exit when* statements within a loop. Because this may be interpreted as violating the single exit principle in software engineering, some developers may restrict the use of *exit when* statements.

(2) Inputs to force the immediate exit of the loop when the decision for the *exit when* evaluates to *true*

(3) Inputs to exercise any logical gates in the decision as required by sections 3.1.1-3.1.5

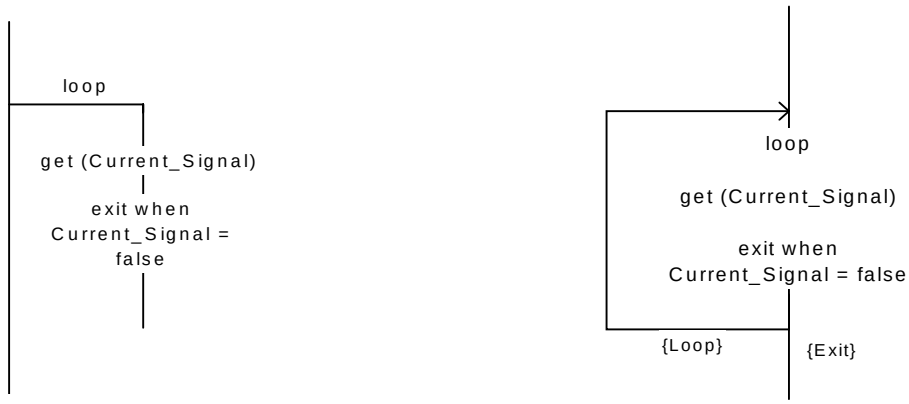


Figure 11. Two types of *exit when* statements.

3.2 Evaluation Method

The minimum test requirements in section 3.1 establish the inputs and expected output for individual logical operators necessary to confirm that:

- the decision has taken all possible outcomes at least once,
- every condition in the decision has taken all possible outcomes at least once, and
- every condition in the decision independently affects the decision's outcome.

Only one minimum test set will provide MC/DC when the only operator in the decision is an ***and,*** and only one minimum test set will provide MC/DC when the only operator in the decision is an ***or***; however, a number of different test sets will meet the minimum test requirements when the operator is an ***xor***.

Confirming MC/DC for a decision with mixed logical operators, such as (**A *or* B) *and*** (**C *or* D**), is complicated by the fact that the output from one logical operator may mask the output of another logical operator. For example, any *false* input to an ***and*** gate masks all other inputs; that is, the output of the ***and*** gate will be *false* regardless of the other inputs. Similarly, any *true* input to an ***or*** gate masks all other inputs; a *true* input will cause the output to be *true* regardless of the other inputs.

Two concepts taken from testing logic circuits are helpful in understanding MC/DC for complex logical expressions: controllability and observability (ref. 14). For software, controllability can be described loosely as the ability to test each logical operator of an expression by setting the values of the expression's inputs (this corresponds to meeting the minimum test requirements in section 3.1). Observability refers to the ability to propagate the output of a logical operator under test to an observable point.

To evaluate MC/DC using the gate-level approach, each logical operator in a decision in the source code is examined to determine whether the requirements-based tests have observably exercised the operator using the minimum tests from section 3.1. This approach, which applies to both decisions with

common logical operators and decisions with mixed logical operators, involves the following five steps:

(1) Create a schematic representation of the source code.

(2) Identify the test inputs used. Test inputs are obtained from the requirements-based tests of the software product.

(3) Eliminate masked test cases. A masked test case is one whose results for a specific gate are hidden from the observed outcome.

(4) Determine MC/DC based on the building blocks discussed in section 3.1.

(5) Finally, examine the outputs of the tests to confirm correct operation of the software. This step may seem redundant because this confirmation is accomplished in the comparison of the requirements-based test cases and results with the requirements. However, if the expected result in the test case does not match the output expected based on the gate representation of the code, an error is indicated, either in the source code or in its schematic representation.

Each of these steps is described below.

3.2.1 Source Code Representation

In the first step of the process, a schematic representation of the software is generated. The symbols used to represent the source code are not important, so long as they are consistent. For this tutorial, the symbols shown in Table 4 and Table 10 are used.

The following example is used to illustrate the steps of the evaluation method, starting with the source code representation.

Example 1. Consider the following line of source code:

A := (B and C) or D;

This source code is shown schematically in Figure 12.

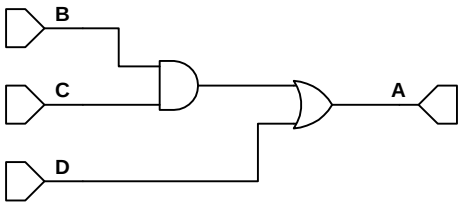
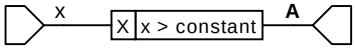
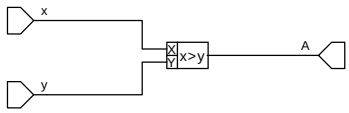
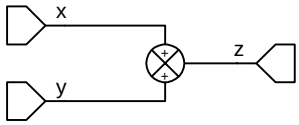
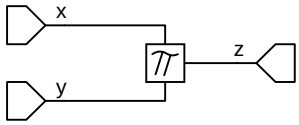
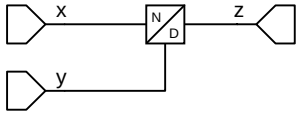
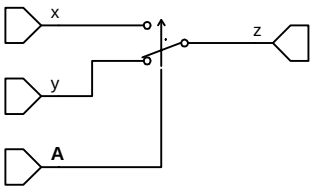
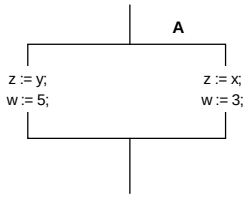


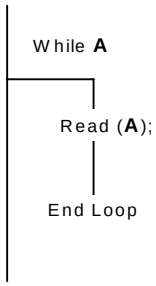
Figure 12. Schematic representation of example 1 source code.

Table 10. Symbols for Source Code Representation[16]

Name	Schematic Representation	Code example
Comparator (x with constant)		**A** := x > constant;
Comparator (x with y)		**A** := x > y;
Summer (addition or subtraction may be shown)		z := x + y;
Multiplier		z := x * y;
Divider		z := x / y;

If-then-else		If **A** then
		z := x;
		Else
		z := y;
		End if;

If-then-else		If **A** then
		z := x;
		w := 3;
		Else
		z := y;
		w := 5;
		End if;

While Loop		While **A** Loop Read (**A**); End loop;

[16] Note that some of the symbols in Table 10 are not building block constructs (for example, the arithmetic symbols), but are needed to represent functionality typical in avionics applications.

3.2.2 Identification of Test Inputs

The next step of the process takes the inputs from the requirements-based test cases and maps them to the schematic representation. This provides a view of the test cases and the source code in a convenient format. Inputs and expected observable outputs for the requirements-based test cases for example 1 are given in Table 11.

Table 11. Requirements-based Test Cases for Example 1

Test Case Number	**1**	**2**	**3**	**4**	**5**
Input **B**	*T*	*F*	*F*	*T*	*T*
Input **C**	*T*	*T*	*T*	*T*	*F*
Input **D**	*F*	*F*	*T*	*T*	*F*
Output **A**	*T*	*F*	*T*	*T*	*F*

Figure 13 shows the test cases annotated on the schematic representation. Note that intermediate results are also determined from the test inputs and shown on the schematic representation.

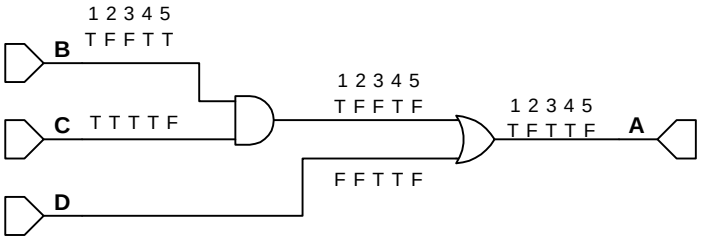


Figure 13. Schematic representation with test cases for example 1.

Knowing the intermediate results is important because some inputs may mask the effect of other inputs when two or more logic constructs are evaluated together. Test cases where the output is masked do not contribute to achieving MC/DC.

Using the annotated figure, the requirements-based tests cases that do not contribute (or count for credit) towards achieving MC/DC can be identified. Once those test cases are eliminated from consideration, the remaining test cases can be compared to the building blocks to determine if they are sufficient to meet the MC/DC criteria.

3.2.3 Elimination of Masked Tests

This step is necessary to achieve observability. Only test cases whose outputs are observable can be counted for credit towards MC/DC. In the following discussion, the electrical analogy of "shorting" various "control inputs" such that they do not impact the "input of interest" being transmitted through them is used to describe several key principles of observability.

To introduce the first principle, consider an ***and*** gate. Since we will concentrate on only one input at a time, we will refer to the experimental input as the input of interest and the other inputs as the control inputs. The truth table for an ***and*** gate in Table 12 shows that the output of the ***and*** gate will always be the input of interest if the control input to the ***and*** gate is *true*. The state of the input of interest is indeterminate in the case where the control input is *false*.

Table 12. Control Input to an ***and*** Gate

Input of Interest	Control Input	Output
T	*T*	*T* (input of interest)
F	*T*	*F* (input of interest)
T or *F* (don't care)	*F*	*F*

This leads to Principle 1: **W *and*** *true* = **W**

Thus any ***and*** gate may be viewed as a direct path from the input of interest to the output whenever the other input(s) to the ***and*** gate are *true*.

Taking a similar approach with the ***or*** gate yields the second principle. The truth table for an ***or*** gate in Table 13 shows that the output of the ***or*** gate will always be the input of interest if the control input to the ***or*** gate is *false*. The state of the input of interest is indeterminate in the case where the control input is *true*.

Table 13. Control Input to an ***or*** Gate

Input of Interest	Control Input	Output
T	*F*	*T* (input of interest)
F	*F*	*F* (input of interest)
T or *F* (don't care)	*T*	*T*

Hence, Principle 2: **W *or*** *false* = **W**

That is, any ***or*** gate may be viewed as a direct path from the input of interest to the output whenever the other input(s) to the ***or*** gate are *false*.

Finally, consider the ***xor*** gate. The truth table for an ***xor*** gate in Table 14 shows that the output of the ***xor*** gate will always be the input of interest if the control input is *false*. The truth table for an ***xor*** gate also shows that the output of the ***xor*** gate will always be the logical ***not*** of the input of interest if the control input is *true*. Thus, the input of interest is always determinate if the control input and output are known.

Table 14. Control Input to an ***xor*** Gate

Input of Interest	Control Input	Output
F	*F*	*F* (input of interest)
F	*T*	*T* (Not input of interest)
T	*F*	*T* (input of interest)
T	*T*	*F* (Not input of interest)

This establishes the final two principles:

Principle 3: **W *xor*** *false* = **W**

Principle 4: **W *xor*** *true* = ***not*** **W**

The judicious selection of a control input for the ***and*** gate, ***or*** gate, or ***xor*** gate opens a direct path from the input of interest to the output. This selection can be used repeatedly to allow visibility of an input of interest across multiple gates as shown in Figures 14 and 15. Figure 14 shows that the output will always be the input of interest when the control inputs are as shown. Any changes to control inputs 1 or 2 will make the input of interest become indeterminate when examining the output. A change to control input 3 will result in the output being the logical ***not*** of the input of interest.

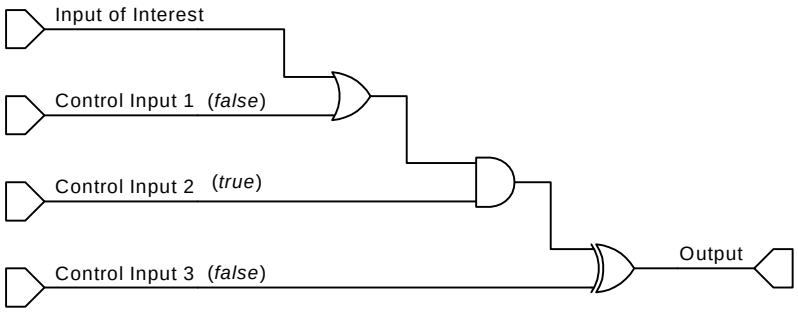


Figure 14. Simple example of directly observable output.

Figure 15 shows another example that allows the input of interest to be directly visible at the output. This is accomplished by the use of three control inputs as shown. Two additional inputs are "don't care" inputs, as their state has no impact on the output for this example. These examples show that the proper selection of control inputs allows an input of interest to be directly observable at the output.

The converse of each of the above principles is used to identify whether a test case is masked. For example, a *false* input to an ***and*** gate will mask all other inputs to that gate. Similarly, a *true* input to an ***or*** gate will mask all other inputs to that gate.

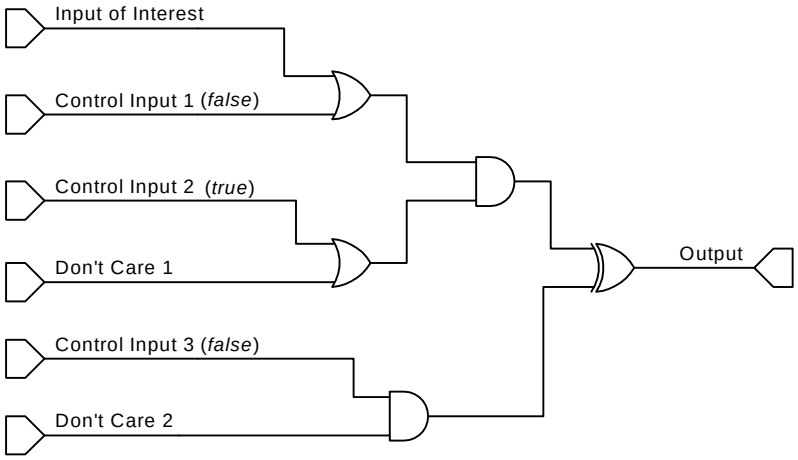


Figure 15. Directly observable output with complex gate structure.

To determine which test cases are masked, it is easiest to work backwards through the logic diagram. Consider again the expression in example 1: **A**:= (**B** ***and*** **C**) ***or*** **D**. Test cases where **D** is *true* cannot be used to determine if the ***and*** gate is implemented correctly. Any time **D** is *true*, the output **A** will be *true*. By setting **D** *true*, the correct output of the ***and*** gate cannot be determined by looking at the results at **A**. Figure 16 shows that test cases 3 and 4 are eliminated from consideration for the ***and*** gate.

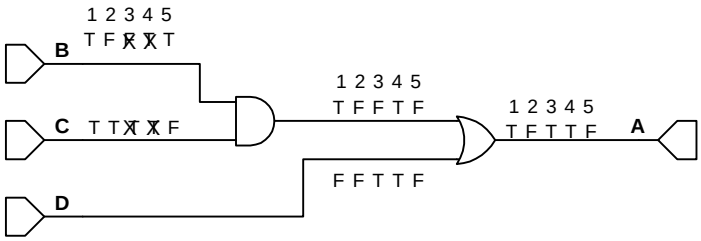


Figure 16. Schematic representation with masked test cases for example 1.

In Figure 16, only test cases 1, 2, and 5 are valid for testing the ***and*** gate, because **D** is set to *false* (**W** ***or*** *false* = **W**) only in these cases. The ***and*** gate is masked in test cases 3 and 4 because **D** is *true* (**W** ***or*** *true* = *true*). Note, masking is not an issue for the ***or*** gate because the output of the ***or*** gate does not feed another gate. Hence, all of the test cases for the ***or*** gate are valid. Table 15 lists the masked test cases and the valid test cases for each gate in the example code.

Table 15. Masked and Valid Test Cases for Example 1

Gate	Masked Test Cases	Rationale for Rejection	Valid Test Cases
and	3, 4	***and*** is masked by *T* input to ***or*** gate for test cases 3 and 4	1, 2, 5
or	None		1, 2, 3, 4, 5

3.2.4 Determination of MC/DC

Having established that test cases 1, 2, and 5 are valid for showing MC/DC for the ***and*** gate in example 1, and that all of the test cases are valid for the ***or*** gate, the next step is to determine whether the valid test cases are sufficient to provide MC/DC.

Starting with the ***and*** gate, the valid test cases are compared with the test case building blocks defined in section 3.1.1. The test combinations *TT*, *TF*, and *FT* are needed. In the example, test case 1 provides the *TT* test, test case 2 provides the *FT* test, and test case 5 provides the *TF* test case. Hence, test cases 1, 2, and 5 are sufficient to provide MC/DC for the ***and*** gate.

Next, the ***or*** gate tests are compared with the test case building blocks defined in section 3.1.2. Test combinations *FF*, *TF*, and *FT* are needed. For the example, test case 2 provides the *FF* test, test case 1 provides the *TF* test, and test case 3 provides the *FT* test. Test case 4, a *TT* input to the ***or*** gate, is not needed for MC/DC; and test case 5 duplicates test case 2 for the ***or*** gate. Table 16 summarizes these results.

Table 16. Comparison of Building Blocks with Valid Tests for Example 1

Gate	Valid Test Inputs	Missing Test Cases
and	*TT* Case 1 *TF* Case 5 *FT* Case 2	None
or	*TF* Case 1 *FT* Case 3 *FF* Case 2 or 5	None

Hence, test cases 1, 2, 3, and 5 satisfy MC/DC for example 1. Note that test cases 1, 2, and 5 contribute to demonstrating coverage for both the ***and*** gate and the ***or*** gate. Test case 4 does not contribute to MC/DC.

3.2.5 Output Confirmation

After confirming that the requirements-based tests provide MC/DC, the final step of the process is to confirm that the expected results are actually obtained by the tests. The output confirmation step is included as a reminder that showing compliance to MC/DC requirements is done in conjunction with the determination of the proper requirements-based test results. In example 1, the outputs determined by following the test inputs through the logic gates match the expected results.

3.3 Following the Five-step Evaluation Process

This section contains three examples to further illustrate the five-step process of evaluating test cases for MC/DC.

Example 2. Suppose you have examined the test cases in Table 17 and determined that they are adequate requirements-based tests. Note that **Z** is the only observable output for these test cases. Determine if the test cases provide MC/DC for the source code provided.

Table 17. Requirements-based Test Cases for Example 2

Test Case Number	1	2	3	4	5
Input **A**	*F*	*F*	*T*	*F*	*T*
Input **B**	*F*	*T*	*F*	*T*	*F*
Input **C**	*T*	*F*	*F*	*T*	*T*
Input **D**	*F*	*T*	*F*	*F*	*F*
Output **Z**	*F*	*T*	*F*	*T*	*T*

Source Code:

```
Z := (A or B) and (C or D);
```

Step 1: Show the source code schematically (see Figure 17).

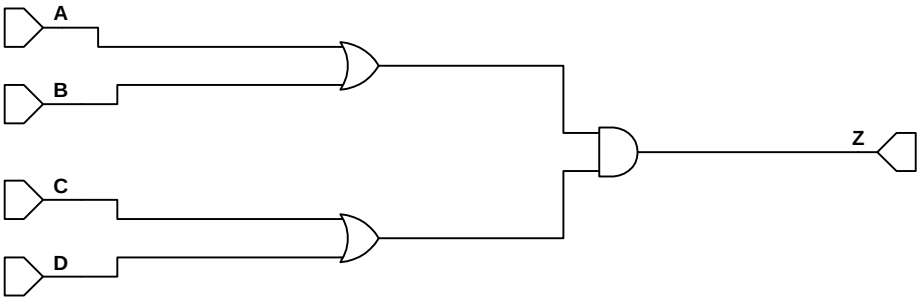


Figure 17. Example 2, step 1—schematic representation of source code.

Step 2: Map test cases to the source code picture (see Figure 18).

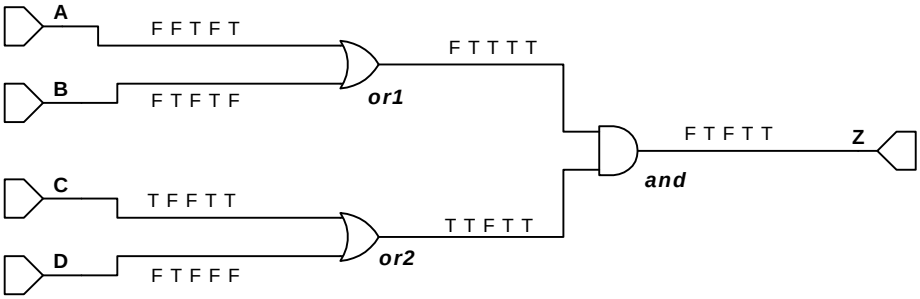


Figure 18. Example 2, step 2—schematic representation with test cases.

Step 3: Eliminate masked tests (see Figure 19). In this case, any *false* input to the ***and*** gate will mask the other input. Hence, the *false* outcome of ***or1*** will mask test case 1 for the ***or2*** gate. Similarly, the *false* outcome of ***or2*** will mask test case 3 for the ***or1*** gate.

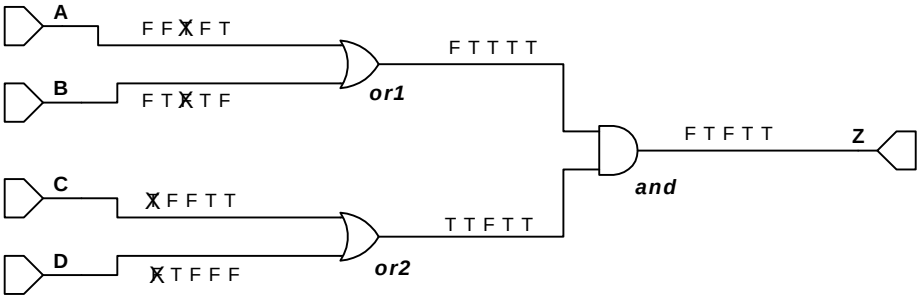


Figure 19. Example 2, step 3—masked test cases.

Step 4: Determine MC/DC. As shown in Table 18, the test set in Table 17 provides MC/DC for this example.

Table 18. Comparison of Building Blocks with Valid Tests for Example 2

Gate	Valid Test Inputs	Missing Test Cases
or1	*FF* Case 1 *FT* Case 2 or 4 *TF* Case 5	None
or2	*FF* Case 3 *FT* Case 2 *TF* Case 4 or 5	None
and	*TT* Case 2, 4, or 5 *TF* Case 3 *FT* Case 1	None

Step 5: Confirm output. The outputs computed match those provided.

<u>Example 2A.</u> Consider a change to test case 2 in example 2 from *FTFT* to *TFFT*. Does this change result in a test suite that still provides MC/DC? The change, shown in Figure 20, amounts to changing the inputs to ***or1*** in test case 2 from *FT* to *TF*. There are no other changes.

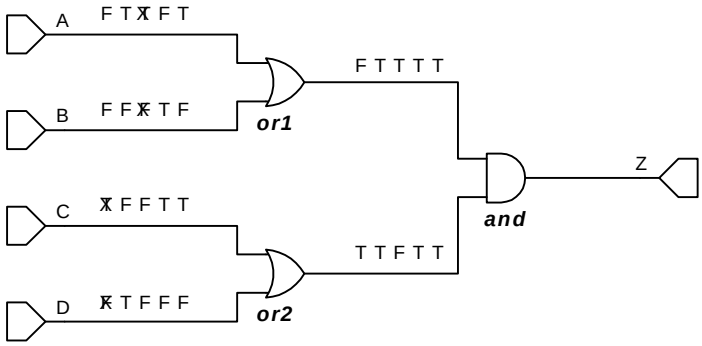


Figure 20. Example 2A, step 3—schematic representation with test cases.

The comparison with the building blocks results in the identical results for ***or2*** and ***and***. The results for ***or1*** are slightly different as shown in Table 19 below.

Table 19. Comparison of Results for ***or1*** Gate for Example 2 and 2A

Example 2		Example 2A	
Valid Test Inputs for *or1*	**Missing Test Cases**	**Valid Test Inputs for *or1***	**Missing Test Cases**
FF Case 1 *FT* Case 2 or 4 *TF* Case 5	None	*FF* Case 1 *FT* Case 4 *TF* Case 2 or 5	None

This example shows the difference between unique-cause MC/DC and masking MC/DC. Although the test suites in both example 2 and example 2A provide MC/DC, only the test suite in example 2A complies with the unique-cause definition. The difference in the test suites can be seen by looking at the independence pairs for each condition, as shown in Table 20.

Table 20. Independence Pairs for Example 2 and 2A

Independence Pairs for Example 2				Independence Pairs for Example 2A			
A	**B**	**C**	**D**	**A**	**B**	**C**	**D**
FFTF	FFTF	TFFF	**FTFT**	FFTF	FFTF	TFFF	**TFFT**
TFTF	FTTF	TFTF	TFFF	TFTF	FTTF	TFTF	TFFF

The difference between example 2 and example 2A lies in how the independent effect of input **D** is shown. In example 2, tests *FTFT* and *TFFF* together show the independent effect of **D**; however, more than one input changes between the two test cases—violating unique cause. The fact that **A** and **B** change values between the two test cases does not impact the decision outcome, because the value of the ***or1*** subterm remains the same. That is, input **D** is changed while all other subterms remain fixed as shown by test cases 2 and 3.

As explained in Appendix B, either the unique-cause or masking approach to MC/DC will detect the same types of errors.

Example 3. Suppose you have a design that calls for the evaluation of (**A *and not* B**) ***or*** (**C *xor* D**). Further, suppose you have examined the test cases in Table 21 and determined that they are adequate requirements-based tests. Determine if the test cases provide MC/DC of the source code provided.

Table 21. Requirements-based Test Cases for Example 3

Test Case Number	1	2	3	4	5
Input **A**	*T*	*T*	*F*	*F*	*F*
Input **B**	*T*	*F*	*T*	*T*	*T*
Input **C**	*F*	*F*	*F*	*F*	*T*
Input **D**	*F*	*F*	*F*	*T*	*F*
Output **Z**	*F*	*T*	*F*	*T*	*T*

Source Code:

```
Z :=  (A and not B) or (C xor D);
```

Step 1: Show the source code schematically (Figure 21).

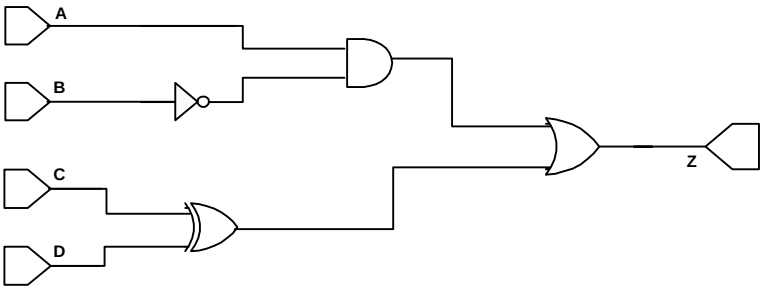


Figure 21. Example 3, step 1—schematic representation of source code.

Step 2: Map test cases to the source code picture (see Figure 22).

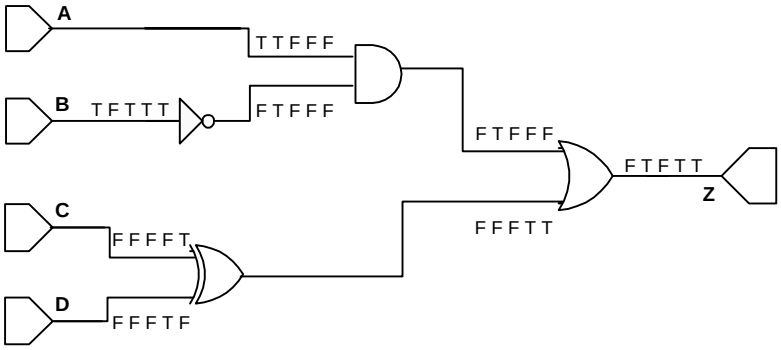


Figure 22. Example 3, step 2—schematic representation with test cases.

Step 3: Eliminate masked tests (Figure 23).

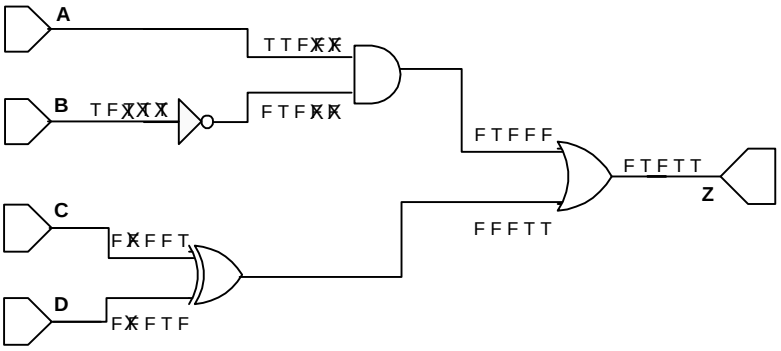


Figure 23. Example 3, step 3—eliminating masked test cases.

Step 4: Determine MC/DC. As shown in Table 22, a test case is needed where the ***and*** gate has **A** *false* and ***not*** **B** *true*. To ensure visibility at **Z**, the output of **C** ***xor*** **D** must be *false* also. One possible test case for (**ABCD**) is (*FFTT*).

Table 22. Comparison of Building Blocks with Valid Tests for Example 3

Gate	Valid Test Inputs	Missing Test Cases
and	*TF* Case 1 *TT* Case 2	*FT*
not	*T* Case 1 *F* Case 2	None
xor	*FF* Case 1 or 3 *FT* Case 4 *TF* Case 5	None
or	*FF* Case 1 or 3 *TF* Case 2 *FT* Case 4 or 5	None

Step 5: Confirm output. The outputs computed match those provided. Hence, test cases 1, 2, 4, 5, and (*FFTT*) provide MC/DC for example 3.

Exercises: For each of the following, complete the five steps of the evaluation process to determine whether the given requirements-based test set provides MC/DC.

Exercise 3.3a: Suppose that the source code in example 3 is actually

Z:= (A and not B) or (C or D);

Would the test cases in Table 21 catch the coding error?

Exercise 3.3b: Suppose you are evaluating a design that controls the start of a ground test function. The requirements for the design are as follows:

1. The ground test function is initiated when the discrete **Start_GT** is set *true*.
2. The user has requested the initiation of ground test when the discrete **Maint_Rqst** is set *true*.
3. The ground test function is to be initiated only if the maintenance system is valid (**Maint_Valid** = *true*).
4. The ground test function is to be disabled (**Start_GT** = *false*) if the weight on wheel discrete (**WOW**) is *false*.
5. The ground test function is to be disabled (**Start_GT** = *false*) if either engine 1 is running (**Engine_1_On**) or engine 2 is running (**Engine_2_On**).

The following tests are provided as verification of the design.

Test Case Number	1	2	3	4	5	6
Maint_Rqst	*T*	*F*	*T*	*T*	*T*	*T*
Maint_Valid	*T*	*T*	*F*	*T*	*T*	*T*
WOW	*T*	*T*	*T*	*T*	*T*	*F*
Engine_1_On	*F*	*F*	*F*	*T*	*T*	*F*
Engine_2_On	*F*	*F*	*F*	*T*	*F*	*F*
Start_GT	*T*	*F*	*F*	*F*	*F*	*F*

The source code used to implement the design is as follows:

Start_GT := Maint_Rqst and Maint_Valid and WOW and not (Engine_1_On or Engine_2_On);

Do the test cases given provide MC/DC of the above source code?

Exercise 3.3c: Suppose you are evaluating a design that determines when the airplane is on the ground with weight on the wheels (**WOW**). The **WOW** requirements state that the **WOW** discrete is to be set when 1) both squat switches (**Squat_L**, **Squat_R**) are set or 2) airspeed is valid and less than 40 knots.
The following tests are provided as verification of the design.

Test Case Number	**1**	**2**	**3**	**4**	**5**
Squat_L	*T*	*F*	*T*	*T*	*T*
Squat_R	*T*	*T*	*F*	*F*	*T*
Airspeed	35	35	45	35	45
Airspeed_Valid	*T*	*T*	*T*	*F*	*F*
WOW	*T*	*T*	*F*	*F*	*T*

The source code used to implement the design is as follows:

WOW := (Squat_L and Squat_R) or ((Airspeed < 40.0) and Airspeed_Valid);

Do the test cases given provide MC/DC based on the above source code?

Exercise 3.3d: Requirements for a set of gains are as follows:

Gain	**Air Data Valid and In Air**	**Not(Air Data Valid and In Air)**
Gain_1	0.34 * True Air Speed	100
Gain_2	0.0012 * Ground Speed	0.5
Gain_3	0.056 * Vertical Speed	0

For this example the following data names and units are used:

Parameter	**Variable Name**	**Units**
True Air Speed	TAS	Miles per Hour (MPH)
Ground Speed	Gnd_Spd	Miles per Hour (MPH)
Vertical Speed	VS	Feet per Minute (FPM)
In Air	**In_Air**	Boolean
Air Data Valid	**Air_Data_Valid**	Boolean

The following test cases are provided for the requirements:

Test 1 Conditions: Set up the simulation while In Air with Air Data Valid and a True Air Speed of 500 MPH, a ground speed of 550 MPH and a vertical speed of +100 FPM. Observe that Gain_1 = 170, Gain_2 = 0.66 and Gain_3 = 5.6.

Test 2 Conditions: Set up the simulation while In Air with Air Data not Valid and a True Air Speed of 500 MPH, a ground speed of 550 MPH and a vertical speed of +100 FPM. Observe that Gain_1 = 100, Gain_2 = 0.66 and Gain_3 = 0.0.

Test 3 Conditions: Set up the simulation while not In Air with Air Data Valid and a True Air Speed of 50 MPH, a ground speed of 55 MPH and a vertical speed of +10 FPM. Observe that Gain_1 = 100, Gain_2 = 0.66 and Gain_3 = 0.0.

The requirements are implemented as:

```
If Air_Data_Valid and In_Air then
      Gain_1 := TAS * 0.34;
      Gain_2 := Gnd_Spd * 0.0012;
      Gain_3 := VS * 0.056;
  Else
      Gain_1 := 100.0;
      Gain_2 := 0.5;
      Gain_3 := 0.0;
End If;
```

Do the test cases above meet the requirements of MC/DC for the source code implemented?

3.4 Grouped Functionality

The discussion in section 3.3 focused on determining compliance with the MC/DC criteria for a single line of code. Because the five-step approach considers the observability of each decision outcome with respect to the expected results, the approach can be used for assessing coverage of multiple lines of source code.

Example 4. Consider the following source code and the requirements-based test cases in Table 23:

Source Code:

```
A := (B or C) and D;
E := (X and Y) or C;
Z := A and E;
```

Table 23. Requirements-based Test Cases for Example 4

Test Case Number	1	2	3	4	5
Input **B**	*T*	*F*	*F*	*T*	*T*
Input **C**	*F*	*T*	*F*	*F*	*F*
Input **D**	*T*	*T*	*T*	*F*	*T*
Input **X**	*T*	*F*	*T*	*T*	*T*
Input **Y**	*T*	*T*	*T*	*T*	*F*
Output **Z**	*T*	*T*	*F*	*F*	*F*

Step 1: Show the source code schematically (Figure 24).

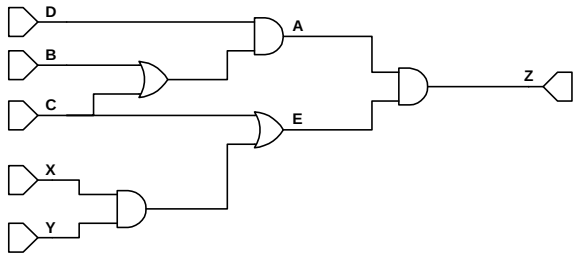


Figure 24. Example 4, step 1—schematic representation of source code.

Step 2: Map test cases to the source code picture (see Figure 25). Note that the values obtained for variables **A** and **E** are not included in the test results given in Table 23. Because there are no expected results for **A** and **E**, we need to make sure that the results of **A** and **E** are visible at **Z**.

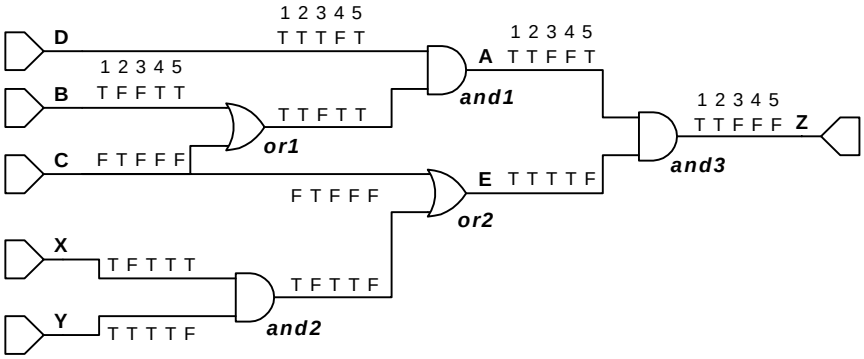


Figure 25. Example 4, step 2— schematic representation with test cases.

Step 3: Eliminate masked tests (see Figure 26). Table 24 gives the rationale for test cases that are masked for each gate.

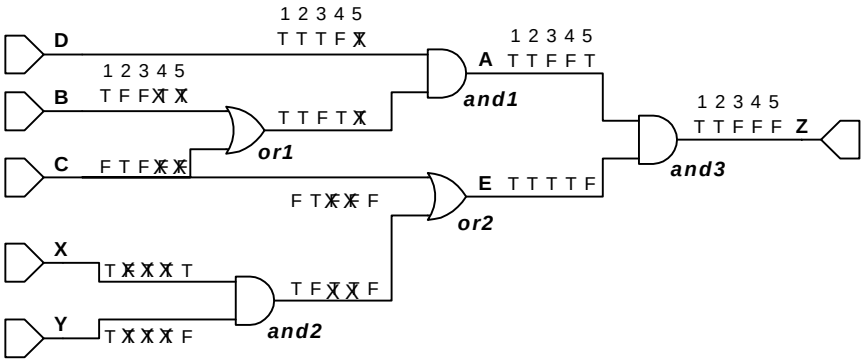


Figure 26. Example 4, step 3—eliminating masked test cases.

Table 24. Masked and Valid Test Cases for Example 4

Gate	Masked Test Cases	Rationale for Rejection	Valid Test Cases
and1	5	***and1*** is masked by *F* output of ***or2*** for case 5	1, 2, 3, 4
and2	2, 3, 4	***and2*** is masked by *F* output of ***and1*** for cases 3 and 4 ***and2*** is masked by *T* input to ***or2*** for case 2	1, 5
and3	None		1, 2, 3, 4, 5
or1	4, 5	***or1*** is masked by *F* input to ***and1*** for case 4 ***or1*** is masked by *F* output of ***or2*** for case 5	1, 2, 3
or2	3, 4	***or2*** is masked by *F* output of ***and1*** for cases 3 and 4	1, 2, 5

Step 4: Determine MC/DC by evaluating the valid test cases for each gate against the building blocks in section 3.1. Table 25 summarizes the valid test cases for each gate and shows which tests defined in section 3.1 are missing.

Table 25. Comparison of Building Blocks with Valid Tests for Example 4

Gate	Valid Test Inputs	Missing Test Cases
and1	*TT* Case 1 or 2 *TF* Case 3 *FT* Case 4	None
and2	*TT* Case 1 *TF* Case 5	*FT*
and3	*TT* Case 1 or 2 *FT* Case 3 or 4 *TF* Case 5	None
or1	*TF* Case 1 *FT* Case 2 *FF* Case 3	None
or2	*FT* Case 1 *TF* Case 2 *FF* Case 5	None

Table 25 shows that MC/DC has been obtained for all of the gates with the exception of ***and2***. The ***and2*** gate does not have a valid test for the case where **X** is *false* and **Y** is *true*. It is interesting to note that there is a test case for ***and2*** which provides the **X** *false*, **Y** *true* inputs (namely, case 2). This shows that although test cases are present, they may not contribute towards MC/DC if their outcome is not observable. One possible valid test case with an observable output is **B** *true*, **C** *false*, **D** *true*, **X** *false*, and **Y** *true*.

Exhaustive testing of the design would require 32 test cases (2^5). If each line of source code is examined individually, the truth table approach would require four test cases for **A**, four test cases for **B**, and three test cases for **E**. Some of those test cases may overlap, but comparison of the entries in the different truth tables would be required to know if they do. Example 4 has shown, in a single analysis, that 6 test cases (5 inputs + 1) can provide complete MC/DC.

Step 5: Confirm output. The outputs computed match those provided.

Example 5. Suppose you are evaluating a power management design for a system that has two generators. Failure flags (**Generator_1_Fail**, **Generator_2_Fail**) are set when the respective generator fails. There is a requirement to set a flag (**Shed_Partial_Load**) to start electrical load shedding in the event of a single generator failure. Another flag (**Shed_Full_Load**) is to be set if both generators fail. Does the following set of tests provide MC/DC based on the source code provided?

Test 1: Both generators are failed and observe **Shed_Partial_Load** is *false* and **Shed_Full_Load** is *true*.

Test 2: Fail #1 generator with #2 generator valid and observe **Shed_Partial_Load** is *true* and **Shed_Full_Load** is *false*.

Test 3: Fail #2 generator with #1 generator valid and observe **Shed_Partial_Load** is *true* and **Shed_Full_Load** is *false*.

Source Code:

```
Shed_Partial_Load := Generator_1_Fail xor Generator_2_Fail;
Shed_Full_Load := Generator_1_Fail and Generator_2_Fail;
```

Step 1: Show the source code schematically (see Figure 27).

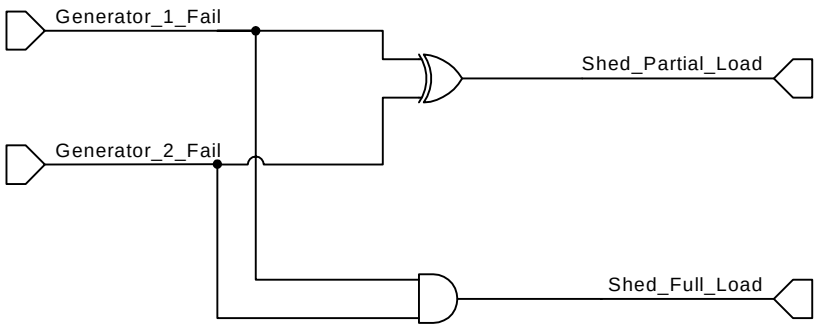


Figure 27. Example 5, step 1—schematic representation of source code.

Step 2: Map test cases to the source code picture as in Figure 28.

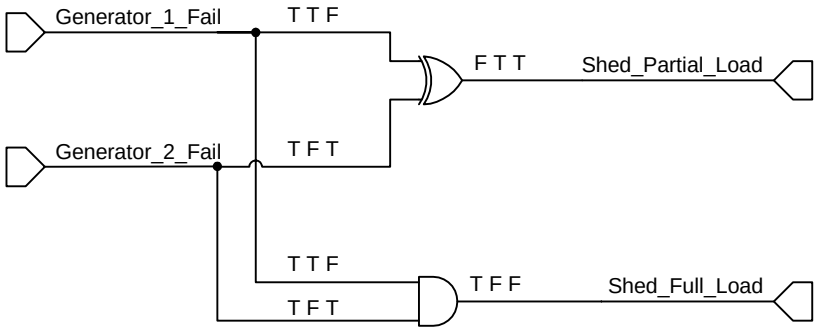


Figure 28. Example 5, step 2—schematic representation with test cases.

Step 3: Eliminate masked tests (see Figure 29). In this example, no test cases are masked, because no gate outputs are input to other gates.

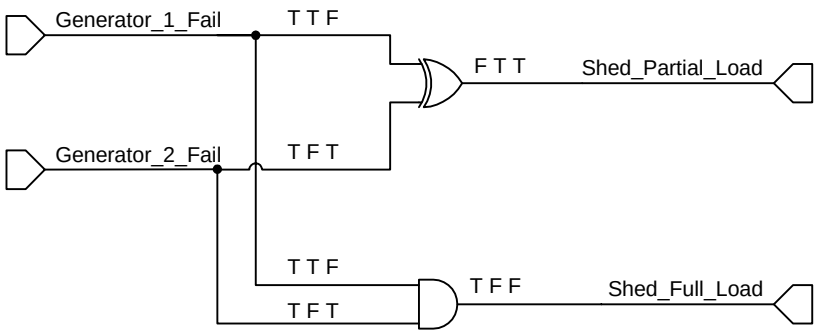


Figure 29. Example 5, step 3—masked test cases.

Step 4: Determine MC/DC (see Table 26).

Table 26. Comparison of Building Blocks with Valid Tests for Example 5

Gate	Valid Test Inputs	Missing Test Cases
xor	*TT* Case 1 *TF* Case 2 *FT* Case 3	None
and	*TT* Case 1 *TF* Case 2 *FT* Case 3	None

Step 5: Confirm output. Outputs computed match those provided.

<u>Example 5A.</u> Does the test set in example 5 provide MC/DC if the design is implemented in the source code as:

Shed_Full_Load := Generator_1_Fail and Generator_2_Fail;
Shed_Partial_Load := (Generator_1_Fail or Generator_2_Fail) and not Shed_Full_Load;

Step 1: Show the source code schematically (see Figure 30).

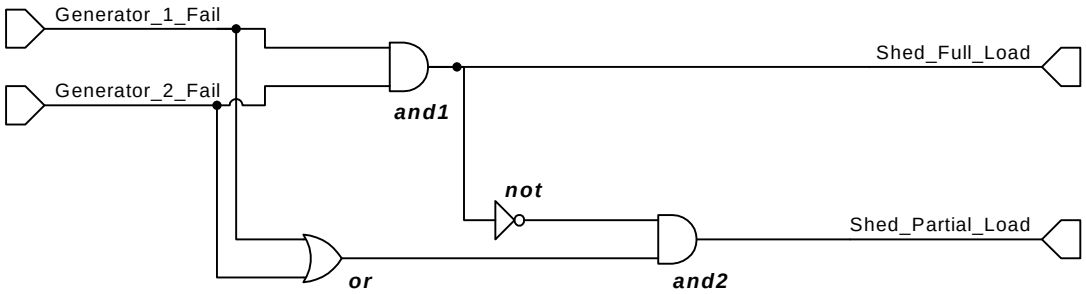


Figure 30. Example 5 Example 5A, step 1—schematic representation of source code.

Step 2: Map test cases to the source code picture (see Figure 31).

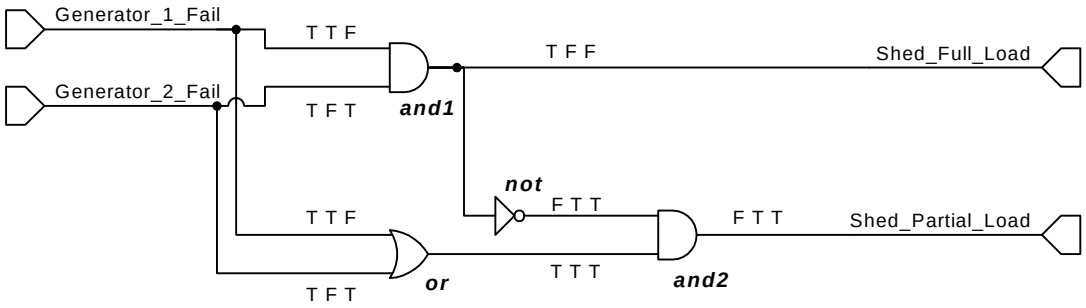


Figure 31. Example 5, step 2—schematic representation with test cases.

Step 3: Eliminate masked tests (see Figure 32).

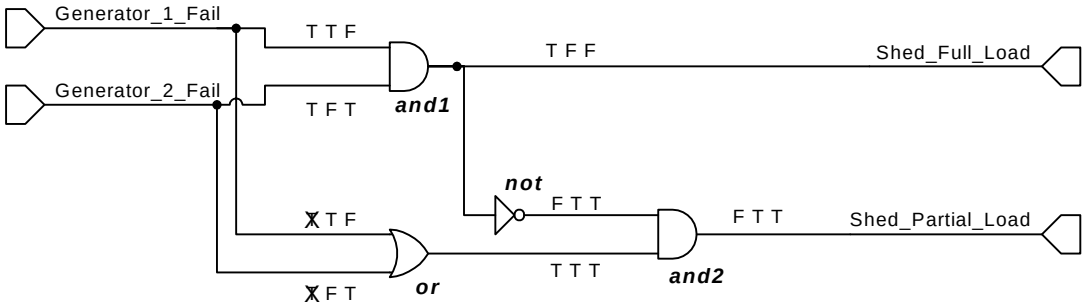


Figure 32. Example 5, step 3—masked test cases.

Step 4: Determine MC/DC. While Table 27 seems to indicate that two additional test cases are required, only one other test case is possible (i.e., **Generator_1_Fail** and **Generator_2_Fail** both set *false*). This additional test provides complete coverage of the inputs and completes the test cases needed for ***and2*** and ***or***.

Table 27. Comparison of Building Blocks with Valid Tests for Example 5

Gate	Valid Test Inputs	Missing Test Cases
and1	*TT* Case 1 *TF* Case 2 *FT* Case 3	None
and2	*TT* Case 1 *FT* Case 3	*TF*
or	*TF* Case 2 *FT* Case 3	*FF*
not	*T* Case 1 *F* Case 2 or 3	None

Step 5: Confirm output. Outputs computed match those provided.

Example 5a is an excellent case of showing that the determination of MC/DC is directly related to the source code implementation.

3.5 Developing Complex Constructs

The building blocks provided in section 3.1 provide the capability to perform MC/DC analysis on a wide variety of source code. These building blocks may be expanded into more complex constructs as needed. This is valuable when common logic strings are repeated throughout the software design.

Example 6. Consider a Reset-Overides-Set latch as shown in Figure 33. This latch could be expressed in the following source code where **Output**, **Set**, and **Reset** are all Booleans:

Output := (Output or Set) and not Reset;

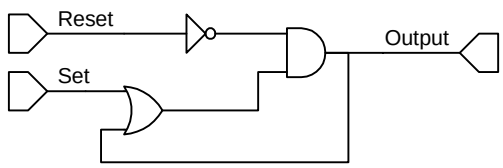


Figure 33. Reset-Overides-Set Latch—schematic representation of source code.

While evaluating the latch every time it is used is valid, it may make sense to create a building block applicable to the specific function of the latch. This is similar to the reuse principal in software engineering. Once a valid set of test cases are determined for the function, they may be reused elsewhere.

The minimum test cases for the latch are shown in Table 28.

Table 28. Minimum Test Cases for the Reset-Overides-Set Latch

Test Case Number	**1**	**2**	**3**	**4**	**5**
Input **Set**	*F*	*F*	*T*	*F*	*T*
Input **Reset**	*T*	*F*	*F*	*F*	*T*
Output	*F*	*F*	*T*	*T*	*F*

Exercise 3.5: Confirm that the minimum test cases in Table 28 provide MC/DC for the Reset-Overides-Set Latch. Also, consider why test case 1 is included but does not directly contribute to MC/DC.

3.6 MC/DC with Short Circuit Logic

When using a standard ***and*** or ***or*** operator, both of the operands in the expression are typically evaluated. For some programming languages, the order of evaluation is defined by the language, while for others, it is left as a compiler-dependent decision. Some programming languages also provide short-circuit control forms.

In Ada, the short-circuit control forms, ***and then*** and ***or else***, produce the same results as the logical operators ***and*** and ***or*** for Boolean types, except that the left operand is *always* evaluated first (ref. 15). The right operand is only evaluated if its value is needed to determine the result of the expression. The && and the || operators in C and C++ are similar (ref. 16, 17).

Short circuit logic also occurs when compiler options are selected which do not require all of the operands within an expression to be evaluated once the output has been determined. Short circuit logic, whether by language construct or compiler option, is similar to the masking discussion of section 3.2.3. That is, the compiler takes advantage of the converse of the principles discussed in section 3.2.3. For example, once any operand used by an ***and*** gate has been evaluated to *false*, the outcome of that expression is known to be *false*. No further evaluation is required, nor will it change the outcome expected. An ***or*** gate likewise will always result in a *true* outcome once any of the inputs evaluates to *true*.

For MC/DC, short circuit expressions can be treated in the same manner as conventional ***and*** and ***or*** gates, as shown in example 7. The follow-on to the example shows how testing can take advantage of the deterministic evaluation order provided by short circuit logic.

Example 7. You have examined the test cases in Table 29 and determined that they are adequate requirements-based test cases. Determine if the test cases provide MC/DC using the source code provided.

Table 29. Requirements-based Test Cases for Example 7

Test Case Number	**1**	**2**	**3**	**4**	**5**
Input **A**	*T*	*T*	*T*	*T*	*F*
Input **B**	*T*	*T*	*T*	*F*	*T*
Input **C**	*T*	*T*	*F*	*T*	*T*
Input **D**	*T*	*F*	*T*	*T*	*T*
Output **Z**	*T*	*F*	*F*	*F*	*F*

Source Code:

```
Z := A and then B and then C and then D;
```

Step 1: Show the source code schematically (see Figure 34). Note that the ***and then*** is shown as the conventional ***and***.

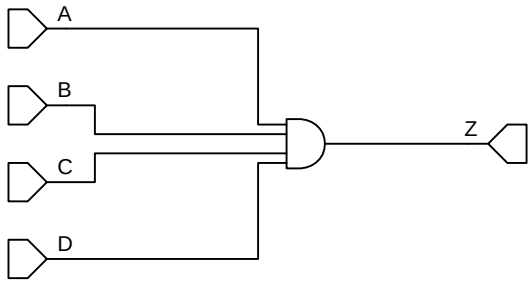


Figure 34. Example 7, step 1—schematic representation of source code.

Step 2: Map test cases to the source code picture (see Figure 35).

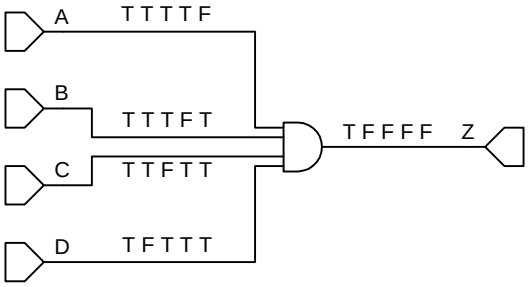


Figure 35. Example 7, step 2—schematic representation with test cases.

Step 3: Eliminate masked tests. Because there are no intermediate gates in this example, there are no masked test cases.

Step 4: Determine MC/DC (see Table 30). In the gate-level approach to determining MC/DC, there is no difference in the determination of MC/DC between the ***and*** gate or the ***and then*** gate.

Table 30. Comparison of Building Blocks with Valid Tests for Example 7

Gate	Valid Test Inputs	Missing Test Cases
and	*TTTT* Case 1 *TTTF* Case 2 *TTFT* Case 3 *TFTT* Case 4 *FTTT* Case 5	None

Step 5: Confirm output. Outputs computed match those provided.

<u>Example 7 follow-on.</u> Now consider example 7 again with the test cases in Table 31.

Table 31. Test Set 2 for Example 7

Test Case Number	1	2	3	4	5
Input **A**	*T*	*T*	*T*	*T*	*F*
Input **B**	*T*	*T*	*T*	*F*	*F*
Input **C**	*T*	*T*	*F*	*F*	*F*
Input **D**	*T*	*F*	*F*	*F*	*F*
Output **Z**	*T*	*F*	*F*	*F*	*F*

Would these test cases provide MC/DC for the source code? Yes. In this example, the operands are evaluated as short circuit logic. Once an operand evaluates to *false*, **Z** is set to *false* without evaluation of the remaining operands. In essence, the value of the remaining operands does not matter. So, for a 4-input ***and*** gate implemented in short circuit logic, the minimum tests are given in Table 32, where the value X represents “don’t care”.

Table 32. Minimum Tests for a Four-input ***and*** Gate with Short Circuit Logic

Test Case Number	1	2	3	4	5
Input **A**	*T*	*F*	*T*	*T*	*T*
Input **B**	*T*	*X*	*F*	*T*	*T*
Input **C**	*T*	*X*	*X*	*F*	*T*
Input **D**	*T*	*X*	*X*	*X*	*F*
Output **Z**	*T*	*F*	*F*	*F*	*F*

Similarly, the minimum tests for a 4-input ***or*** gate implemented in short circuit logic (e.g., **A *or else* B *or else* C *or else* D**) are given in Table 33. Keep in mind that using the minimum tests in Tables 30 and 31 is valid only if the compiler works correctly, even when optimizing.

Table 33. Minimum Tests for a Four-input ***or*** Gate with Short Circuit Logic

Test Case Number	1	2	3	4	5
Input **A**	*F*	*T*	*F*	*F*	*F*
Input **B**	*F*	*X*	*T*	*F*	*F*
Input **C**	*F*	*X*	*X*	*T*	*F*
Input **D**	*F*	*X*	*X*	*X*	*T*
Output **Z**	*F*	*T*	*T*	*T*	*T*

3.7 MC/DC with Bit-wise Operations

Some real time embedded applications require bit-wise operations to support hardware interfaces, memory and throughput constraints, testability issues, or contiguous data transfer constraints. In such cases, Boolean expressions may be represented by individual bits within a word (and some bits in the word may not be used at all). For MC/DC for these cases, bit-wise operations require testing of each individual bit that represents a condition. Testing the code gets more interesting as more bits are packed into a word to represent multiple conditions within a single word. In the case of a word where each of the 16 bits represent separate conditions, all 16 bits are treated as separate conditions.

3.7.1 Examples with Bit-Wise Operations

Example 8. Consider the ***bit-wise and*** of packed 16-bit words x and y with an assignment to z. The source code may be expressed in the C language where **&** represents a ***bit-wise and*** operator as:

```
z = x & y;
```

The test set in Table 34 provides MC/DC of the source code *if* z is examined as an integer value (not as a Boolean).

Table 34. Test Set 1 for Example 8

Test Case Number	1	2	3
Input x	2#1111111111111111#	2#0000000000000000#	2#1111111111111111#
Input y	2#0000000000000000#	2#1111111111111111#	2#1111111111111111#
Observed z	2#0000000000000000#	2#0000000000000000#	2#1111111111111111#

The test set in Table 35 also provides MC/DC for the source code *if* z is examined as an integer value (not as a Boolean).

Table 35. Test Set 2 for Example 8

Test Case Number	1	2	3
Input x	2#1111000011110000#	2#0000111100001111#	2#1111111111111111#
Input y	2#0000111100001111#	2#1111111111110000#	2#1111000011111111#
Observed z	2#0000000000000000#	2#0000111100000000#	2#1111000011111111#

Both test sets show that each individual bit location is tested with the *TF*, *FT*, and *TT* combinations required for an ***and*** gate.

Example 9. As the complexity of a source code line increases, so does the effort required to show MC/DC. For example, consider a design where the first two bits of a variable 'status' are used to represent Boolean conditions. Here, status_bit0 represents **Weight_On_Wheels**, and status_bit1 represents **Engine_Off**. A Boolean condition, **On_Ground**, is determined when either **Weight_On_Wheels** or **Engine_Off** is *true* as shown by the following test (in the C language):

```
On_Ground =  (status & (3)) != 0;
```

Due to the combination of a ***bit-wise and*** involving a constant along with a comparison in a single statement, the output of the ***bit-wise and*** is not observable as an integer. This forces us to change status one bit at a time to observe that the output changes state as a result of only that bit. Thus the test cases required for example 9 are given in Table 36. Note that there is an advantage to setting the 'X' values in Table 36 to one because that will also detect a coding error where the mask inadvertently picks up data in bits 2 through 15.

Table 36. Test Set for Example 9

Test Case Number	1	2	3
Status	#2XXXXXXXXXXXXXX00#	#2XXXXXXXXXXXXXX01#	2#XXXXXXXXXXXXXX10#
On_Ground	*False*	*True*	*True*

Note that the observability of the output in example 8 allows testing of the two-input ***bit-wise and*** gate with three test cases and that example 9 requires n+1 test cases where n is the number of bits being evaluated in the comparison.

Example 10. It is often necessary to strip certain bits out of data before the data can be used. Consider the case where bits 12 to 15 of a word (input_word) contain status information regarding the data contained in bits 0 to 11. Before the data can be used, the status information must be removed. One method for removing the status information is to define a constant for a mask and then ***bit-wise and*** the mask with the word. For example,

```
output_word = input_word & 0FFF;
```

Testing the requirement to mask out the upper four bits of input_word requires showing that each of the upper four bits of output_word are set to zero and that the lower 12 bits remain as they are in

input_word. Because the mask is a fixed value that cannot be changed, only two of the three test cases defined for an ***and*** gate are possible. Tests that show that the four upper bits of output_word are always set to zero for inputs of zero or one in the upper four bits of input_word are reasonable.

3.7.2 Alternate Treatments of Bit-wise Operations

Not everyone treats bit-wise operations in the way described in the previous section. At least three different assertions supporting alternate treatments have been made. We will address each of these briefly.

Some people assert that using bit-wise operations *at all* is poor programming practice. However, as mentioned previously, some embedded applications require bit-wise operations.

Some people assert that bit-wise operations are merely arithmetic operations and thus do not need to be considered for MC/DC. Following the logical implication of this assertion to its natural conclusion would allow coding standards to be developed requiring bit-wise representations of all Booleans. This effective elimination of the need to achieve MC/DC violates the intent of the MC/DC objective.

Some people assert that far more test cases than shown in the above examples are needed to demonstrate independence. This assertion is based on the belief that independent effect of the individual bits *within* a word with respect to the other bits *within* the word must be shown. However, what must be shown is independent effect of the conditions for each bit-wise operation with respect to each bit location. That is, independence does not apply within a *single* word, but between relevant bits of two *separate* words.

3.8 Analysis Resolution

Structural coverage analysis using the MC/DC approach in this tutorial can identify errors or shortcomings in two ways. First, the analysis may show that the code structure was not exercised sufficiently by the requirements-based tests to meet the MC/DC criteria; as shown in example 3. According to section 6.4.4.3 of DO-178B, the unexecuted code may result from shortcomings in requirements-based test cases or procedures, inadequacies in software requirements, dead code, or deactivated code. Section 6.4.4.3 provides guidance for each of these. The MC/DC approach may also identify errors in the source code, as shown in example 11.

Example 11. Suppose you have a requirement to evaluate the expression **A *and*** (**B *xor* C**), and the requirements-based test cases, shown in Table 37, have been accepted as adequate.

Table 37. Requirements-based Test Cases for Example 11

Test Case Number	1	2	3	4
Input **A**	*F*	*T*	*T*	*T*
Input **B**	*F*	*F*	*T*	*T*
Input **C**	*T*	*F*	*F*	*T*
Output **Z**	*F*	*F*	*T*	*F*

Consider the source code:

Z:= B and (B xor C);

In this case, the coder has incorrectly coded a **B** for the **A**.

Step1: Show the source code schematically (see Figure 36).

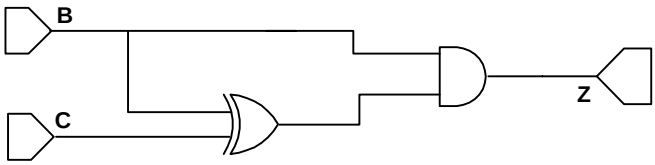


Figure 36. Example 11, step 1—schematic representation of source code.

Step 2: Map test cases to the source code picture (see Figure 37).

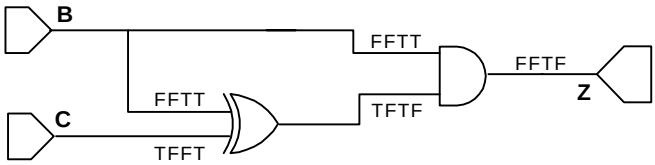


Figure 37. Example 11, step 2—schematic representation with test cases.

Step 3: Eliminate masked tests (see Figure 38).

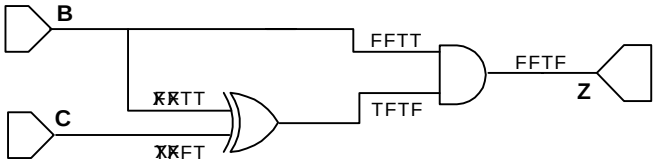


Figure 38. Example 11, step 3—eliminating masked test cases.

Step 4: Determine MC/DC. As shown in Table 38, there are not sufficient tests to meet the minimum tests required for the ***xor*** gate. In this example, the requirements-based test cases do not sufficiently exercise the code because there is an error in the code. This error should be identified when the verification engineer examines the requirements to create the missing test case. When the structural coverage analysis identifies incorrect code, the code should be corrected and test procedures executed in accord with the program's procedures for code changes. Note that if the source code did not have an error, then the test cases in Table 37 would provide MC/DC.

Table 38. Comparison of Building Blocks with Valid Tests for Example 11

Gate	Valid Test Inputs	Missing Test Cases
xor	*TF* Case 3 *TT* Case 4	either *FF* or *FT*
and	*TT* Case 3 *FT* Case 1 *TF* Case 4	None

Step 5: Confirm output. Note that the output computed in step 2 matches the expected output. That is, running the test cases and checking expected results is not sufficient in this example to catch the error.

Note that step 5 may identify an error if the outcome computed from step 2 differs from the expected outcome in the test cases. In this instance, the error may be in the requirements or the source code, or the error may be in the schematic representation. If the error is in the software requirements, the requirements should be corrected and additional test cases developed and test procedures executed, as discussed in section 6.4.4.3b in DO-178B. If the error is in the source code, the code should be corrected and test procedures executed in accord with the program's procedures for code changes.

4 MC/DC Admonitions

"We learn wisdom from failure much more than from success." -Samuel Smiles (ref. 18)

Knowing how to assess requirements-based test sets for MC/DC is necessary, but not sufficient, for complying with the MC/DC objective. Making mistakes in areas other than test set assessment is still a real possibility. This chapter discusses some of the common mistakes that can be made when trying to comply with the MC/DC objective. These problems fall into two broad categories: automation and process.

4.1 Automation Problems

Using a structural coverage analysis tool is particularly appealing for replacing the manual tedium of MC/DC analysis. Using tools to perform tasks that are repetitive, complex, and time consuming can help eliminate the possibility of errors resulting from mental fatigue and release humans for tasks that cannot be automated, such as technical reviewing. While the potential benefit from using tools to automate structural coverage analysis is obvious, the potential harm from using tools without properly confirming operation of the tool may not be.

> *"Automation is a great idea. To make it a good investment, as well, the secret is to think about testing first and automation second." (ref. 19)*

A tool should never be used to replace knowledge about structural coverage analysis, in general, or about MC/DC, in particular. Whenever the decision is made to automate coverage analysis, candidate tools should be carefully assessed to determine the functionality and limitations of each. Appropriate verification procedures should be implemented to account for all tool limitations. The approach given in chapter 3 can be used to help evaluate whether a tool performs properly.

To help prevent problems related to automation of coverage analysis, the following sections give a general overview of how structural coverage analysis tools work and some important factors to consider in selecting and qualifying a structural coverage tool. Please note that the factors outlined in this section do ***not*** cover all the factors that should be considered in using automated tools for structural coverage analysis. Also, specific tool and tool vendors are not discussed.

4.1.1 How Coverage Analysis Tools Work

"[I]t's dangerous to automate something that we don't understand" (ref. 19)

Structural coverage analysis tools typically provide increased visibility into testing by either instrumenting code, or providing other intervention techniques to gain visibility. In general, code is instrumented by adding a series of probes (hardware or software), flags, or other monitoring mechanisms to the original source code or object code. This enables the analysis tool to determine exactly what parts of the code are exercised. Once the code is instrumented, test cases are executed and the coverage analysis tool is expected to track which parts of the code are exercised by the test cases and, where complex analysis is required, how they are exercised. If pass/fail criteria for structural coverage are specified, the tool should analyze the code against these criteria. If pass/fail criteria are not specified, the tool should report the level of structural coverage the test cases achieve.

Because instrumentation changes the code, demonstrating that the instrumentation does not conceal or introduce an error is essential. This is usually achieved by running the same test cases on both the original source code and the instrumented code and comparing results. This implies that sufficient visibility exists in the uninstrumented code, as well as the device being tested and the test equipment, to observe the same results as in the instrumented version. Other methods of demonstrating that the instrumentation did not affect the expected results are also possible. Keep in mind that the coverage tool confirms that the instrumented code meets the MC/DC criteria. The assumption is that the uninstrumented code does, too. Whether this is the case depends on the instrumentation method and compiler combination used.

Different coverage analysis tools use different instrumentation schemes. Knowing which scheme is used by a particular tool is important.

4.1.2 Factors to Consider in Selecting or Qualifying a Tool

The instrumentation scheme is not the only thing that must be understood for tool selection and qualification. The following factors are also important:

- types of monitored statements
- where statements are monitored (source versus object code)
- maximum number of conditions and decisions that can be monitored
- algorithms used for determining independent effect
- handling of relational operators
- instrumentation effects

For each of these, several key questions, along with the rationale for those questions, are listed below.

4.1.2.1 Types of Monitored Statements

Does the tool monitor all of the coverage points?

Can the tool properly handle coverage points for all of the coverage requirements a single statement generates?

To demonstrate MC/DC, a structural coverage analysis tool should monitor statements, entry and exit points, decision and branching statements, and Boolean conditions. Some tools do not support all of the coverage points required for MC/DC. For example, not all structural coverage tools support coverage of entry and exit points. Such a tool can support part of the structural coverage analysis if other means are used to cover entry and exit points. Programming language can also impact the type of statements that are monitored by the tool. For example, certain programming languages[17] do not have a Boolean or logical type. For these languages, the tool may have to infer which expressions are Boolean and which are not by the use of Boolean operators. In these cases, the coverage analysis tool may not be able to monitor all Boolean expressions.

A structural coverage analysis tool should also be able to monitor a statement for multiple coverage points. Consider the following return statement:

```
return (A and B) or C;
```

This statement should be monitored for the following coverage points:

- Statement–must be invoked at least once
- Exit Point–must be invoked at least once
- Decision–must take all possible outcomes (*false*, *true*) at least once
- Condition–each condition (**A**, **B**, **C**) must take all possible outcomes (*false*, *true*) at least once, and each condition (**A**, **B**, **C**) must demonstrate its independent effect

A different return statement (for example, return (x + y) / z;) requires monitoring for different coverage points.

4.1.2.2 Source versus Object Monitoring

Does analysis show that coverage at the object code level is equivalent to coverage at the source code level?

Some structural coverage analysis tools monitor coverage at the source code level; others monitor at the object code level. Achieving MC/DC at the object code level is not necessarily equivalent to achieving MC/DC at the source code level.

As discussed in section 2.5.1, MC/DC may be demonstrated at the object code level "as long as analysis can be provided which demonstrates that the coverage analysis conducted at the object code will be equivalent to the same coverage analysis at the source code level" (ref. 9, FAQ #42). Consequently, using a tool that monitors coverage at the object code level requires additional analysis to confirm the equivalence between coverage at the object code level for the tool and coverage at the source code level.

[17] C is a notorious example of such a language.

This analysis is not typically trivial and should be reviewed by certification authorities early in the project.

4.1.2.3 Maximum number of conditions and decisions that can be monitored

Does the tool have limits on the number of conditions monitored in a given Boolean expression?

Does the tool have limits on the total number of conditions, decisions, or statements that can be monitored?

Some tools limit the number of conditions that can be monitored in a Boolean expression, and some also limit the total number of conditions, decisions, or statements that can be monitored. For a single decision, the monitoring scheme may depend on the number of conditions in the decision. For example, a tool may use one scheme for decisions of eight or less conditions, another for decisions of nine through sixteen conditions, yet another for seventeen through thirty-two conditions, and may not handle decisions with more than thirty-two conditions. In some cases, large expressions may be completely ignored, or the tool may monitor a portion of the expression. Limits on monitored coverage points should be clearly identified and understood. In cases where these limits affect the coverage analysis, mitigation strategies and procedures should be defined and documented. For example, if the total number of elements to be monitored exceeds the tool's limit, one approach would be to monitor different subsets of the system, run the tests once for each monitored subset, and combine the multiple analyses into one analysis for the system. If an expression is too big for the tool, then manual analysis will likely be needed.

4.1.2.4 Algorithms Used for Determining Independent Effect

What is the basis for determining independent effect?

Different structural coverage analysis tools use different algorithms to determine independent effect of a condition. This tutorial covers one method: the MC/DC approach given in chapter 3. Other approaches use expression trees, Boolean difference functions, KV-maps, or function trees. There may be other alternatives in addition to these.

4.1.2.5 Handling Relational Operators

Does the tool properly monitor conditions for independent effect in the presence of relational operators?

When relational operators are used in a Boolean expression (for example, (x<y) ***and*** (x>z)), the independent effect of the conditions in the expression must be demonstrated. Because relational operators are not Boolean operators, the tool should be examined to make sure that it properly monitors conditions in the presence of relational operators.

4.1.2.6 Instrumentation Effects

Does the instrumentation affect the structural coverage analysis?

The effect of probes used for structural coverage analysis fall into the following categories and are discussed below:

- Increased resources, memory and throughput, necessary to support the probes
- Compiler operation due to the presence of software probes
- Factors due to different environments, compiler and/or target, between uninstrumented and instrumented code

4.1.2.6.1 Impact on Resources

What are the effects of instrumentation on resources, memory, and throughput?

Do hardware monitors need to be disabled to allow the instrumented code to execute?

Do real-time computations have different outcomes based on timing differences?

Do the additional memory requirements of the instrumented software cause different memory page boundaries to be crossed?

Ideally, instrumentation would only add the overhead necessary to support the probes themselves, leaving all other aspects of the airborne software and system unchanged. For hardware probes, the ideal can be realized to a very high degree. It can be met fully if the hardware running the airborne software has been designed to provide run-time execution data directly from the hardware at hardware speeds. The major question about hardware probes is: are they catching what is actually going on within the software? Unfortunately, hardware probes generally monitor instruction fetches and can be fooled by modern hardware architectures with cache memories. Just because an instruction is fetched into cache does not mean that it is actually executed. Because an instruction is executed, does not mean it is executed properly. Just as with software probes, hardware probes need to be analyzed to determine what information they are actually producing, under what circumstances, with what assumptions.

Software probes add overhead for memory and CPU cycles. Consequently, the instrumented airborne software may not perform properly in memory or throughput critical components. Throughput critical components may not be able to handle the extra CPU cycles necessary to execute the probes. Memory critical components may not be able to handle the extra memory necessary for the probe instructions.

Other resources may be necessary to support the probes besides memory and throughput. For example, a communication mechanism may be needed that allows for the capture or transfer of the execution data provided by the probes. These additional resources need to be investigated to determine their effects on the development, verification, and structural coverage analysis processes.

4.1.2.6.2 Compiler Operation

Is the behavior of the compiler affected by software probes?

The behavior of the compiler and the resulting executable may be changed by software probes inserted into the source code. Ideally, the only change in the executable would be the support of the probes, and all other functional aspects of the airborne software would still perform correctly (other than throughput and memory usage). However, the presence of the probes may cause the compiler to generate code that will not perform properly.

For example, an expression may require the calling of two functions, say *A* and *B*, where *B* is dependent on a side-effect of *A*. The uninstrumented airborne software executes properly because the compiler always generates code that calls *A* before *B*. In the instrumented code, the compiler now generates code that calls *B* before *A* in the expression. Now the instrumented code no longer performs

correctly in the system context, and this incorrect behavior has nothing to do with the memory and timing effects of the probes.

4.1.2.6.3 Environment Factors

What elements of the software environment are subject to change due to requirements for the structural coverage analysis tool?

Using an automated structural coverage analysis tool may require different tools or tool settings for the instrumented software from those required for the airborne software. The differences may include:

- Compiler–either a different compiler, or different compiler switches
- Linker–either a different linker, or a different linker switches
- Target–a different target environment (such as, an emulator or simulator)

Each of these can affect the fidelity of the execution results between uninstrumented and instrumented source code. The degree of fidelity should be assessed to determine the proper role of the structural coverage analysis tool. Multiple tools may be required (just as in normal verification testing, multiple test tools may be required).

4.2 Process Problems

Even when automated tools are not used, or when the potential pitfalls of such tools are avoided, problems in the development and verification processes can occur. Some of the common ones related to compliance with the MC/DC objective are discussed below.

4.2.1 Inadequate Planning for MC/DC

Planning in advance for both the technical and administrative aspects of verification is essential for project success; however, planning for MC/DC is often inadequate. The following planning problems are frequently seen:

- planning for the wrong software level
- inadequate detail in the *Software Verification Plan* (and other plans) to allow team members to follow
- underestimating resources (including time, cost, and personnel) needed to meet the MC/DC objective
- inadequate planning for the change process
- inadequate planning for use and qualification of structural coverage analysis tools
- poor compliance with plans by the development and verification teams
- inadequate updates for plans
- inadequate planning for design and code standards; that is, underestimating the impact of complex designs and tightly coupled conditions and decisions on achieving MC/DC

4.2.2 Misunderstanding the MC/DC Objective

As discussed in chapter 2, the rationale for the MC/DC objective is often misunderstood—especially the connection to requirements-based testing and unintended function. On one hand, achieving coverage helps confirm that the code does not include unintended function; on the other hand, achieving coverage does not confirm that all of the requirements have been tested. Hence, some applicants do more testing than required; while others do less. The following are some examples of misunderstanding the MC/DC objective:

- not understanding the intent of structural coverage
- trying to meet the MC/DC objective apart from requirements-based testing; that is, using the source code to derive inputs for all test cases
- trying to achieve MC/DC before having a stable implementation
- using MC/DC as a testing method; that is, expecting MC/DC to find errors instead of assuring that requirements-based testing is adequate
- having erroneous expectations about the capability of structural coverage analysis tools (see section 4.1)
- not knowing when MC/DC has been met
- not recognizing the potential impact of obtaining MC/DC on coding standards and compiler settings and options
- relying on MC/DC to find problems that should have been addressed earlier in the software life cycle (such as complex or erroneous logic)
- not knowing whether to meet the MC/DC objective at the source code or object code level. This is often caused by misunderstanding source to object code traceability.

4.2.3 Inefficient Testing Strategies

Another problem is failing to take advantage of the coverage hierarchy, as shown in Table 1. Statement coverage and decision coverage are often monitored independently of MC/DC. While DO-178B does not prohibit this approach; it may not be the most efficient use of resources. Treating coverage criteria separately may result in the following inefficiencies:

- complicated software change tracking, because the software could change before each type of coverage analysis has been completed
- redundant activities
- conflicting results if different tools are used for each analysis

4.2.4 Poor Management of Verification Resources

Verification resources (time, funding, and personnel) are rarely abundant. Consequently, mismanagement of those resources can lead to a number of problems, including the following:

- test cases being developed by inexperienced or unqualified engineers
- inadequate training of the verification team, especially training for new verification tools
- inadequate documentation of test cases and procedures in order to support future changes and regression analysis and testing

- extensive and detailed work being performed late in the project life cycle within a compressed schedule
- testing critical functions late in the project life cycle
- inadequate change control process, including inadequate regression testing
- reliance on structural coverage analysis tools to compensate for inadequate personnel or experience

5 Assessment Process

Thus far, this tutorial has focused exclusively on MC/DC. This chapter does not. Instead, this chapter provides hints to certification authorities about what to look for in the assessment process. Although the chapter is intended primarily for authorities responsible for review and approval, software developers may find this section useful to better understand the evidence sought in the assessment process or to perform internal reviews in preparation for scrutiny by certification authorities or designated engineering representatives (DERs)[18].

General steps in the review of the verification process are shown in Figure 40. For each assessment step, questions are given to guide the evaluation process with respect to MC/DC. Because the review of MC/DC data typically takes place within the context of the overall verification review, most of the questions apply regardless of the structural coverage level. However, the focus is on assessment of the MC/DC approach and evidence. The questions are *not* intended to be used strictly as a checklist and are *not* inclusive of all possible situations that need to be reviewed.

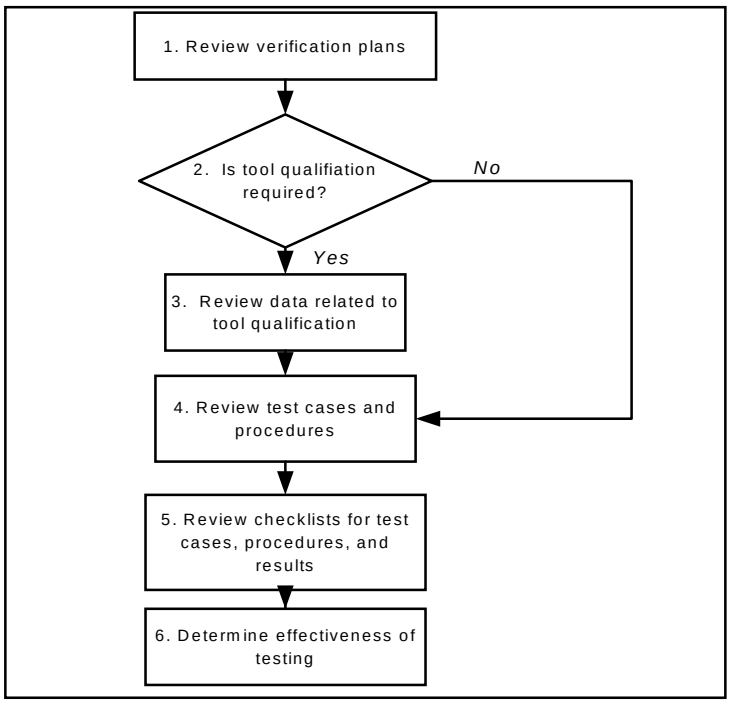


Figure 40. Steps in the review of an applicant's testing program.

[18] DERs are used by the FAA to make findings of compliance and conduct oversight on behalf of the FAA. Transport Canada has a similar delegation system. The JAA does not have a designee system.

The steps in Figure 40 complement the guidance in the FAA's Job Aid "Conducting Software Reviews Prior to Certification" (ref. 20), providing specific guidance for reviewing data relevant to structural coverage. The Job Aid partitions the assessment process into four stages of involvement:

- Stage of Involvement #1—The Planning Review
- Stage of Involvement #2—The Development Review
- Stage of Involvement #3—The Verification/Test Review
- Stage of Involvement #4—The Final Review

Typically, Steps 1-2 in Figure 40 would occur during Stage of Involvement #1. Step 3 may be evaluated initially in Stage of Involvement #1 and completed during Stage of Involvement #3. Steps 4-6 would occur during Stage of Involvement #3.

Step 1—Review Verification Plans

Early in the review process, the *Software Verification Plan* should be reviewed to ensure that activities planned for achieving MC/DC, if followed, will satisfy the DO-178B objective. Other plans including the *Plan for Software Aspects of Certification*, *Software Configuration Management Plan*, *Software Quality Assurance Plan*, and tool plans may contain additional information related to MC/DC.

The following questions might be considered when reviewing the plans:

- Are the plans sufficiently clear and detailed to allow the development and verification engineers to follow them consistently?
- Do the plans (or supporting documents) specify who is allowed to perform verification tasks?
- Do the plans specify how each requirement will be tested (e.g., module test, software integration, etc.)?
- Do the plans address all aspects of MC/DC analysis? For example, are the following addressed:
  - tools and tool qualification, if tools are used for MC/DC
  - the relationship between requirements-based testing and measuring MC/DC
  - a process for determining when additional requirements-based tests should be added, if coverage is not achieved as expected
  - a procedure for regression analysis and testing, if necessary
  - the transition criteria to start and end MC/DC
- Do the plans address the software change process for the airborne software and tools (if tools are used)?
- Do the plans address regression analysis and testing with respect to the unique requirements for MC/DC?
- Do the plans address possible reuse of verification tools? For example, is credit being claimed from previous tool qualifications or will the tool qualification data be used in a future program?
- Is there evidence that the plans are being followed (such as, progress against timeframes, staffing, verification records, and SQA records)?

Step 2—Determine need for tool qualification

Applicants may use tools to perform some verification activities, including MC/DC analysis. Tool usage (that is, which tools are used for what purposes) should be clearly documented during the applicant's planning process along with additional verification processes to cover for limitations of the tools (for example, additional questions in checklists).

Tool qualification should be addressed in the applicant's planning documents when structural coverage tools are used. Tool qualification is required if the tool reduces, eliminates, or automates an objective of DO-178B and if the output of the tool is not verified as required by section 6 of DO-178B. FAA Notice 8110.91, *Guidelines for the Qualification of Software Tools Using RTCA/DO-178B* (ref. 21), provides guidelines to determine if a tool should be qualified.

The following questions might be considered when determining whether a tool needs to be qualified:

- Can the tool allow an existing error to remain undetected?
- Will there be little or no verification of the output of the tool?
- Will the tool be used to assist or replace a process that has a significant effect on the integrity of the product being developed?

Step 3—Review data related to qualification of MC/DC tools

If tools are used for MC/DC analysis, the following questions should be considered when reviewing the qualification data:

- Do the plans state which MC/DC tools are being qualified and the rationale for qualification? (Note: this might be in the *Plan for Software Aspects of Certification* or a separate tool qualification plan.)
- Are the specific tool requirements documented? DO-178B, section 12.2.3.2 lists the typical information that should be included in the Tool Operational Requirements document.
- Does the Tool Operational Requirements document identify all of the tool's functions?
- Is the effect of various coding considerations (e.g. structures and naming conventions) addressed?
- Does the tool qualification data address whether the tool needs to instrument the code to perform MC/DC analysis?
- If the tool needs to instrument the code, has the effect of the instrumentation on the code been assessed?
- If the tool measures coverage at the object code level, is additional analysis available to support the equivalence of coverage at the object and source code levels?
- Is the tool qualification process sufficient to discover errors in the tool and limitations of the tool's functions?
- Does the tool qualification data address how tool deficiencies that are found while the tools are being used in a certification project should be handled?
- Does the tool qualification data detail how changes to the tool will be evaluated and controlled?
- Are procedures for using each tool documented?

- Are limitations of the tool that may affect assessment of coverage clearly documented and addressed (e.g., the limitations discussed in chapter 4)?
- Is the tool configuration controlled and documented in the plans and Software Life Cycle Environment Configuration Index?
- Are the verification engineers using the tool configuration identified in the plans and the Software Life Cycle Environment Configuration Index?

Step 4—Review test cases and procedures

From an assessment perspective, evidence is needed to determine whether the test cases and procedures were developed according to the documented plans, whether the test cases cover all the software requirements, and whether the test cases are sufficient to meet the MC/DC objective. The test cases may be spread across multiple test procedures or multiple test levels (i.e., module tests, integration tests, etc.). Regardless, the connection between the test cases and requirements should be identified clearly in traceability matrices.

In most projects, the applicant reviews their requirements-based test cases to assure that all requirements are adequately covered prior to review by a certification authority. If these requirements-based tests are not adequate to achieve MC/DC, then additional requirements-based tests or analysis may be needed.

The following questions may be used to evaluate the test cases and procedures:

- Do the test cases and procedures adhere to the relevant plans and standards? For example, have coding standards, especially those relevant to limitations of structural coverage tools, been followed?
- If plans or standards have not been followed, is there documented rationale for deviations from stated plans and standards?
- Is the rationale for each test case clearly explained?
- Are the test cases and procedures appropriately commented to allow future updates?
- Have the test cases and procedures been subjected to appropriate change and configuration control?
- Is the separation between test cases clear? For example, are test start and stop identified? This assists tracing the source of unexpected drops in coverage.
- Do the test cases and procedures specify required input data and expected output data?
- Were the inputs for each test case derived from the requirements (as opposed to being derived from the source code)?
- Have the appropriate memory locations and variables been preset?
- Are the test cases and procedures sufficient to cover all the relevant requirements? That is, do the traceability matrices provide clear association between test cases and requirements?
- Are the test cases and procedures sufficient to achieve MC/DC?
- Are sufficient tests to provide MC/DC identified for each logic construct?
- Are there sufficient robustness test cases and procedures?

- Are only test inputs that are unmasked (that is, whose outcomes are directly observable) identified or counted for credit for MC/DC?
- Are requirements where analysis is required in addition to (or in lieu of) requirements-based testing clearly documented (e.g., requirements for hardware polling)?
- Are test cases and procedures correct?

Step 5—Review checklists for test cases, procedures, and results

For most level A projects, the applicant has checklists for reviewing test cases, procedures, and results. During a review, these checklists should be assessed considering the following questions for MC/DC:

- Are the checklists sufficient to determine that the requirements-based test cases, procedures, and results meet the MC/DC objective?
- Have the checklists been reviewed?
- Do the checklists specify:
  - who performed the review?
  - what data was reviewed (with revision)?
  - when it was reviewed?
  - what was found?
  - what corrective actions were taken, if necessary?
- Do the checklists require evaluation of tolerances specified in the requirements?
- Do the checklists ensure that results of the test cases can be visually verified? (e.g., can the verification engineer visually determine when requirements-based tests have passed or failed?)
- Will the checklists reveal whether the results of the test cases that are counted for credit towards MC/DC are observable?
- Will the checklists address limitations of the structural coverage analysis tool as documented in the tool qualification?
- Will the checklists reveal test cases that violate project standards?
- Will the checklists reveal test cases that are not expected to achieve 100% structural coverage (e.g., hardware polling)?

Step 6—Determine effectiveness of test program

In general, three tasks are associated with determining the effectiveness of the overall test program (see Figure 41). The final task deals with assessing whether the appropriate level of structural coverage is achieved.

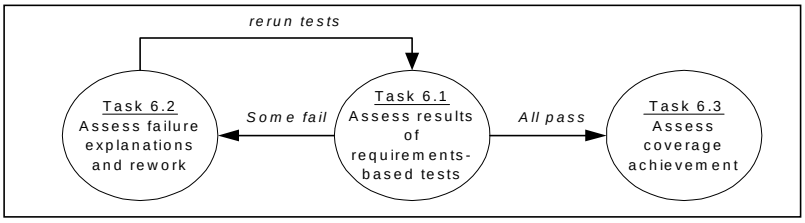


Figure 41. Tasks for assessing test program effectiveness.

Task 6.1—Assess results of requirements-based tests

Because MC/DC is a measure of the adequacy of requirements-based testing, the first logical step after test execution is to determine whether all requirements-based tests pass. In addition to checking the final pass/fail results, the test cases and results for some randomly selected requirements should be examined to ensure that the results reflect the given inputs for those cases. Test results also should be checked carefully with respect to any specified tolerances.

Questions to assess the requirements-based test results are listed below:

- Are the test result files clearly linked to the test procedures and code? (i.e., does configuration control and traceability exist?)
- Are failed test cases obvious from the test results?
- Do the test results indicate whether each procedure passed or failed and the final pass/fail results?
- Do the test results adhere to the relevant plans, standards, and procedures?
- Have the test results been subjected to appropriate configuration control?

Task 6.2—Assess failure explanations and rework

Understanding the failed cases is equally as important as verifying those test cases that pass. Each failed test case should have a suitable explanation for why it failed including references to all applicable problem reports. In some cases, rework of some life cycle data will be required; in other cases, only an explanation for the failed test cases is needed. If rework is required, the impact of changes should be carefully evaluated and the changed items should be subjected to the appropriate change and configuration control. According to section 7.2.4 of DO-178B, "[S]oftware changes should be traced to their origin and the software life cycle processes repeated from that point at which the change affects their outputs." Once all rework is complete, test cases should be rerun in compliance with plans for regression testing. Note: there may be cases where failed requirements-based tests are acceptable; however, it is typical for them to be fixed and rerun.

The following questions might be considered to assess failures and rework:

- Is there an acceptable rationale for deviations from expected results, standards, or plans?
- Are explanations for the failed test cases technically sound and accurate?
- Do explanations for failed test cases contain accurate references to relevant problem reports?
- Are explanations for code or test rework suitable to address the failure?
- Have test cases been re-executed in compliance with plans for regression testing?
- Have the test results from regression testing been documented appropriately?

Task 6.3—Assess coverage achievement

Some test sets will be expected to achieve 100% MC/DC; others may not. Test documentation such as that described in chapter 3 should be examined to verify compliance. Where full coverage is not expected, the supporting analysis should be well documented.

Unanticipated levels of coverage from test execution also should be explained. The explanation should cover exactly what parts of the code have not been exercised and why. If all the requirements

have been covered by tests without achieving MC/DC, dead code, unintended function, or incorrectly documented de-activated code may be indicated. Dead code should be removed, and unintended function and de-activated code should be explained and addressed, as discussed in section 6.4.4.3 of DO-178B. Source code errors may also be indicated if the requirements-based tests do not provide MC/DC, as shown in example 11. The source code errors should be corrected and test procedures executed in accord with the program's procedures for code changes. In all cases, supplemental test cases added to achieve MC/DC should be consistent with the requirements.

The following questions may be considered when assessing coverage achievement:

- Has the applicant correctly applied the MC/DC criteria?
- If statement coverage and decision coverage are assumed in the implementation of MC/DC, have they truly been achieved?
- Is 100% MC/DC achieved through requirements-based testing?
- If 100% MC/DC is not achieved through requirements-based testing, is there an explanation detailing which parts of the code were not executed and why?
  - Have additional test cases been added?
- Are explanations for drops in coverage sufficiently detailed and acceptable?
- Are there problem reports associated with dead code?
- Has dead code been analyzed or removed?

6 Summary

The subject of MC/DC has been a source of consternation for many within the aviation software community. This tutorial attempted to relieve anxiety and confusion by providing practical information regarding the intent of the MC/DC objective in DO-178B, and an approach to assess whether the objective has been met. In addition to presenting an analysis approach, the tutorial also reviewed important factors to consider in selecting and qualifying a structural coverage tool and tips for appraising an applicant's life cycle data relevant to MC/DC. Mastery of the topics presented in this tutorial will enable a certification authority or verification analyst to effectively evaluate MC/DC claims on a level A software project, and will aid in selection, qualification, and approval of structural coverage analysis tools.

7 References

1. *Advisory Circular #20-115B*. U. S. Department of Transportation, Federal Aviation Administration, issued January 11, 1993.

2. *RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification.* RTCA, Inc., Washington, D. C., December 1992.

3. Hayhurst, Kelly J.; Dorsey, Cheryl A.; Knight, John C.; Leveson, Nancy G.; McCormick, G. Frank*: Streamlining Software Aspects of Certification: Report on the SSAC Survey*. NASA/TM-1999-209519, August 1999.

4. Chilenski, John Joseph: *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*. FAA Tech Center Report DOT/FAA/AR-01/18, April 2001.

5. Wiener, Ruth: *Digital Woes, Why We Should Not Depend on Software*. Addison-Wesley Publishing Company, 1993.

6. *RTCA/DO-254, Design Assurance Guidance for Airborne Electronic Hardware.* RTCA, Inc., Washington, D. C., April 2000.

7. Beizer, Boris: *Software Testing Techniques.* Second ed., Van Nostrand Reinhold Company, Inc., 1983.

8. Herring, Michael Dale: Testing Safety Critical Software. M. S. Thesis, Florida Institute of Technology, May 1997.

9. *RTCA/DO-248A, Second Annual Report for Clarification of DO-178B "Software Considerations in Airborne Systems and Equipment Certification".* RTCA, Inc., Washington, D. C., September 13, 2000.

10. Myers, Glenford J.: *The Art of Software Testing*. John Wiley & Sons, 1979.

11. Cornett, Steve: Code Coverage Analysis, Bullseye Testing Technology, http://www.bullseye.com/webCoverage.html Accessed March 2001.

12. DeWalt, Michael: MCDC, A blistering love/hate relationship. *Proceedings of the Federal Aviation Administration Software Standardization Seminar*, Long Beach, CA, April 1999.

13. Chilenski, John Joseph; and Miller, Steven. P.: Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, Vol. 7, No. 5, September 1994, pp. 193-200.

14. Abramovici, Miron; Breuer, Melvin A.; and Friedman, Arthur D.: *Digital Systems Testing and Testable Design.* Computer Science Press, 1990.

15. *International Standard ANSI/ISO/IEC-8652:1995, Ada 95 Reference Manual.* Intermetrics, Inc., January 1995.

16. Ellis, Margaret A; and Stroustrup, Bjarne: *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1990.

17. Kernighan, Brian W.; and Ritchie, Dennis M.: *The C Programming Language*. Second ed., Prentice Hall, 1988.

18. Bartlett, John; Kaplan, Justin, ed.: *Bartlett's Familiar Quotations : A Collection of Passages, Phrases, and Proverbs Traced to Their Sources in Ancient and Modern Literature*. Little Brown & Company, 1992.

19. Bach, James: Testing Automation Snake Oil, http://www.satisfice.com/articles/test_automation_snake_oil.pdf Accessed March 2001.

20. FAA Job Aid: Conducting Software Reviews Prior to Certification. Aircraft Certification Service, June 1998, http://av-info.faa.gov/software/jobaid.htm. Accessed January 2001.

21. *Notice 8110.91. Guidelines for the Qualification of Software Tools using RTCA/DO-178B.* U. S. Department of Transportation, Federal Aviation Administration, Cancellation Date 1/15/02. (Note: 8110.83 was renumbered to 8110.91 in January 2001)

Appendix A

Solutions to Exercises

Solution 2.5a

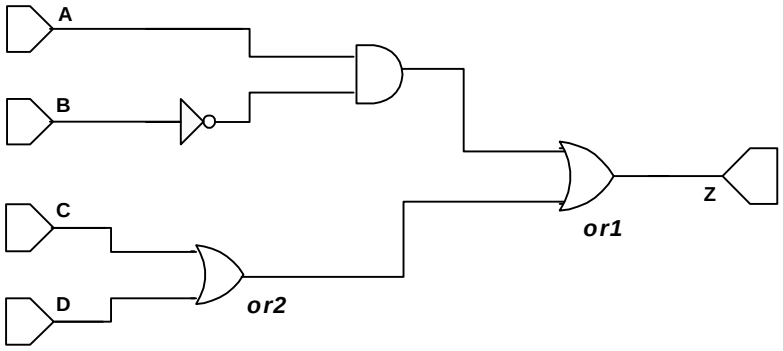
(2^{36} tests)*(1 sec/100 tests)*(1 minute/60 sec)*(1hour/60 min)*(1 day/24 hour)*(1 year/365 day) = Approximately 21.79 years

Solution 2.5b

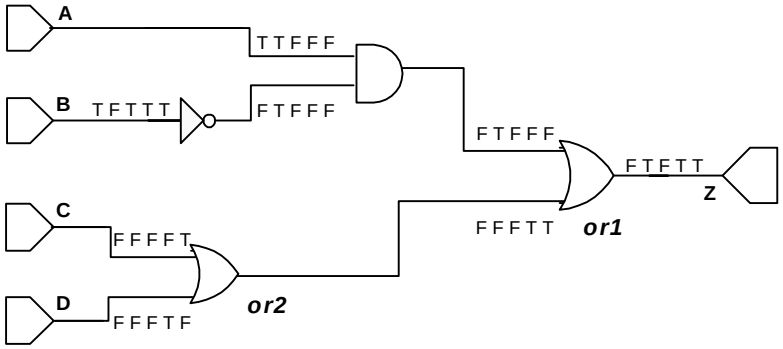
(2^{36} tests)*(1 page/64 lines)*(1 inch/250 pages)*(1 yard/36 inch)*(1 mile/1759.65 yards) = Approximately 67.8 miles

Solution 3.3a, OR/XOR Exercise

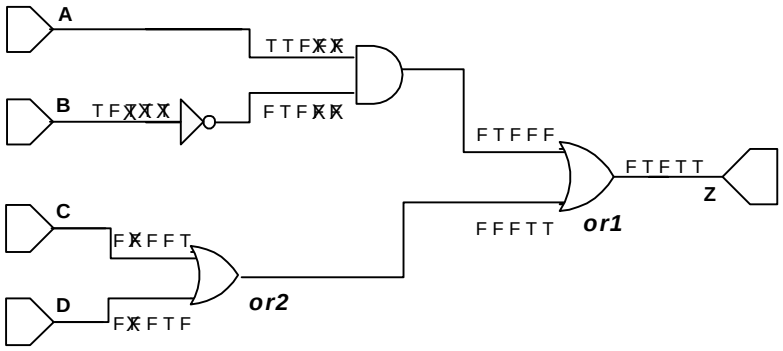
Step 1: Show the source code schematically.



Step 2: Map test cases to the source code picture.



Step 3: Eliminate masked tests.



Step 4: Determine MC/DC. As in example 3, a test case is still needed where the ***and*** gate has **A** *false* and **not B** *true*.

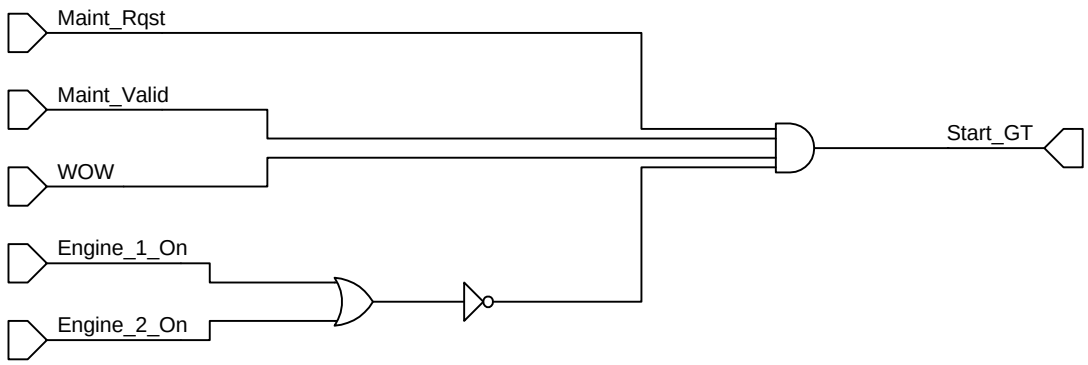
Gate	Valid Test Inputs	Missing Test Cases
and	*TF* Case 1 *TT* Case 2	*FT*
not	*T* Case 1 *F* Case 2	None
or1	*FF* Case 1 or 3 *FT* Case 4 or 5 *TF* Case 2	None
or2	*FF* Case 1 or 3 *TF* Case 5 *FT* Case 4	None

Step 5: Confirm output. The outputs computed match those provided. Hence, test cases 1, 2, 4, and 5 plus (*FFTT*) provide MC/DC for example 3.

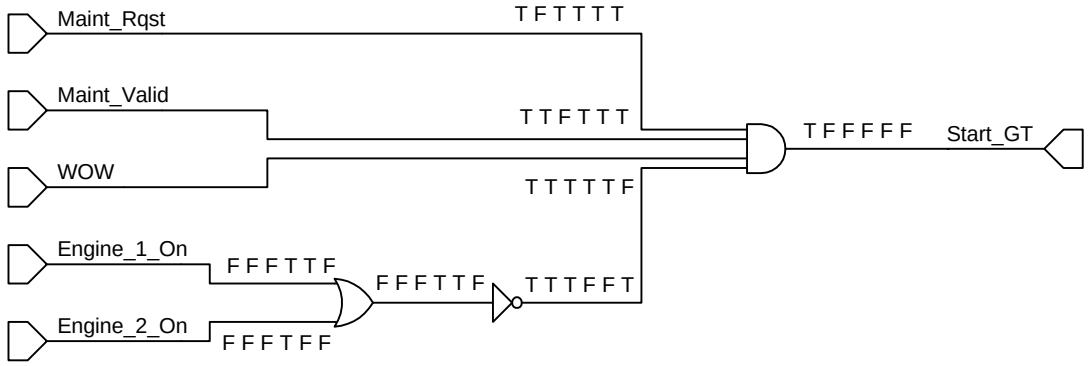
The requirements-based tests in this case will not detect that the ***or2*** gate should have been an ***xor*** gate.

Solution 3.3b, Ground Test Exercise

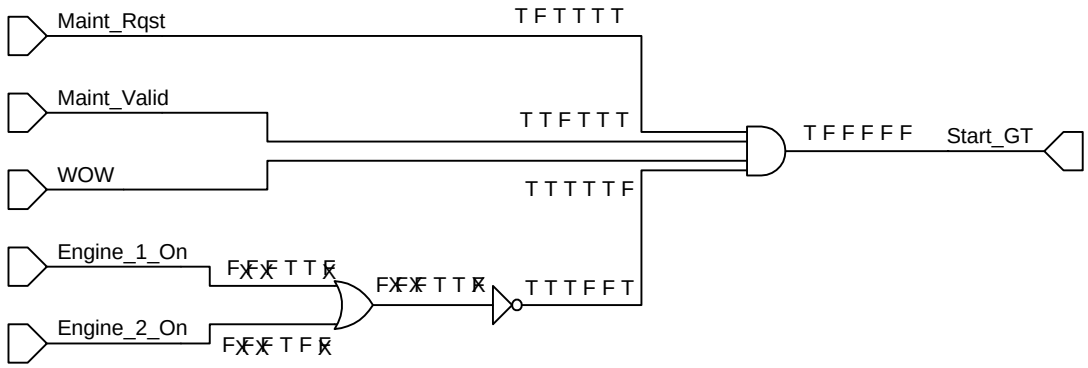
Step 1: Show the source code schematically.



Step 2: Map test cases to the source code picture.



Step 3: Eliminate masked tests.



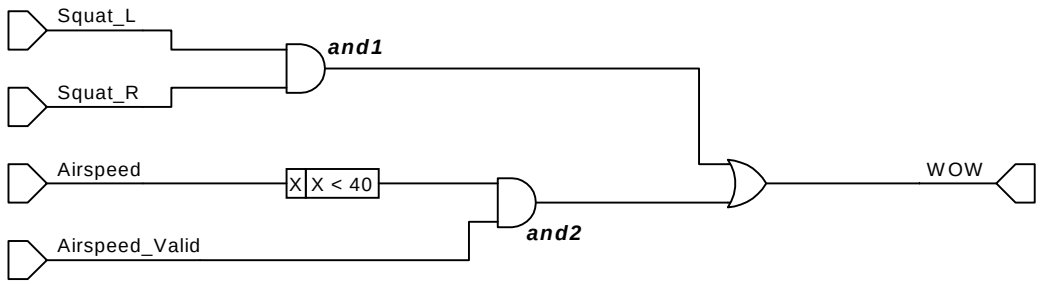
Step 4: Determine MC/DC. Note that the testing for the ***or***-gate is incomplete.

Gate	Valid Test Inputs	Missing Test Cases
and	*TTTT* Case 1 *FTTT* Case 2 *TFTT* Case 3 *TTFT* Case 6 *TTTF* Case 4 or 5	None
or	*FF* Case 1 *TF* Case 5	*FT*
not	*T* Case 4 or 5 *F* Case 1	None

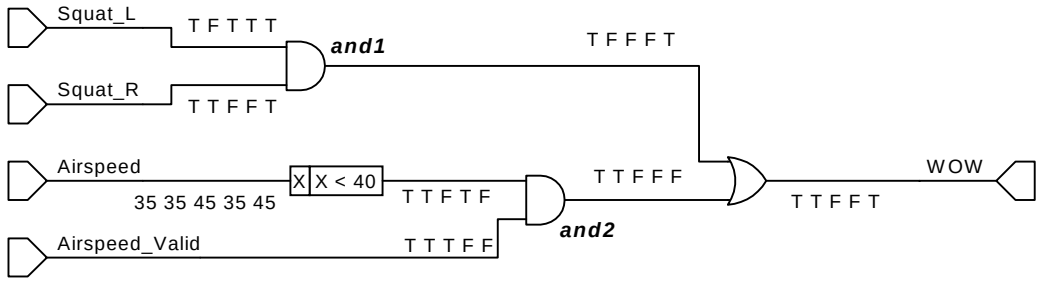
Step 5: Confirm output. Outputs computed match those provided.

Solution 3.3c, Weight on Wheels Exercise

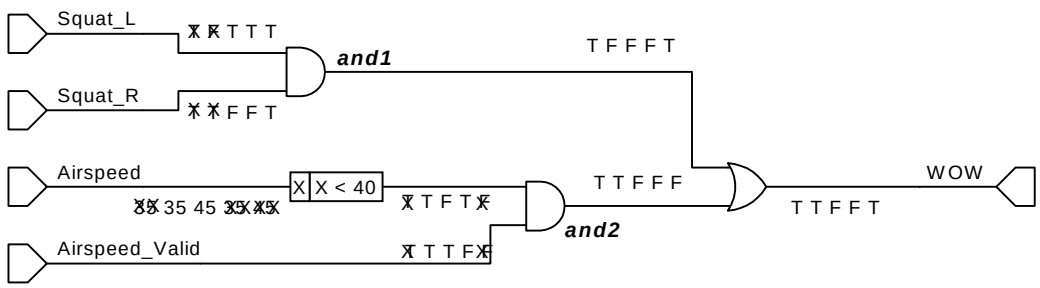
Step 1: Show the source code schematically.



Step 2: Map test cases to the source code picture.



Step 3: Eliminate masked tests.



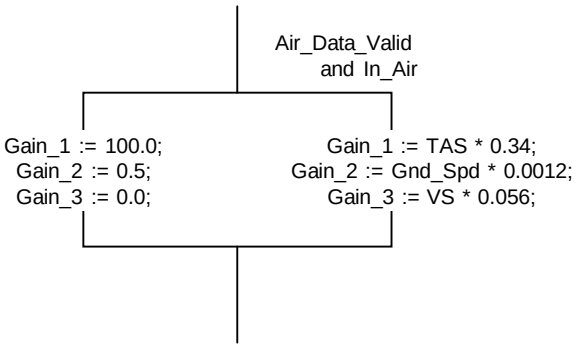
Step 4: Determine MC/DC.

Gate	Valid Test Inputs	Missing Test Cases
and1	*TT* Case 5 *TF* Case 3 or 4	*FT*
and2	*TT* Case 2 *FT* Case 3 *TF* Case 4	None
or	*FF* Case 3 or 4 *TF* Case 5 *FT* Case 2	None
Comparator	*T* Case 2 *F* Case 3	None

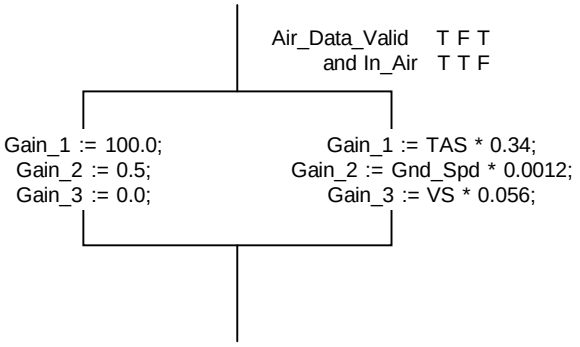
Step 5: Confirm output. Outputs match those provided.

Solution 3.3d, Gain Exercise

Step 1: Show the source code schematically.



Step 2: Map test case to the source code picture.



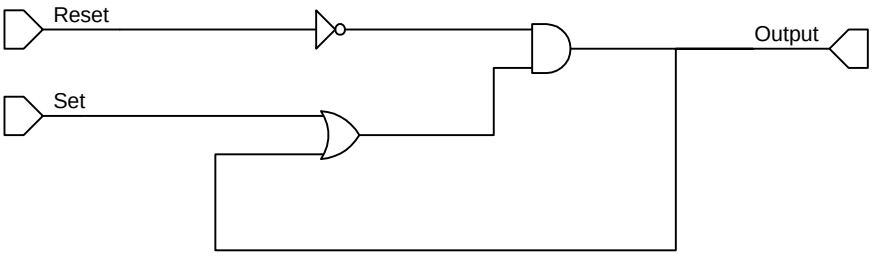
Step 3: Eliminate masked tests. There are no masked tests to remove for this example

Step 4: Determine MC/DC. For this example, testing the design with three test cases is sufficient to show MC/DC. Note that while the first two test conditions exercise both branches in the software, they are not sufficient to show MC/DC. The third test case must be used to provide the MC/DC assurance for the ***and*** gate. Note also that for complex decisions it may be advantageous to show the decision logic using the gate level schematic representation.

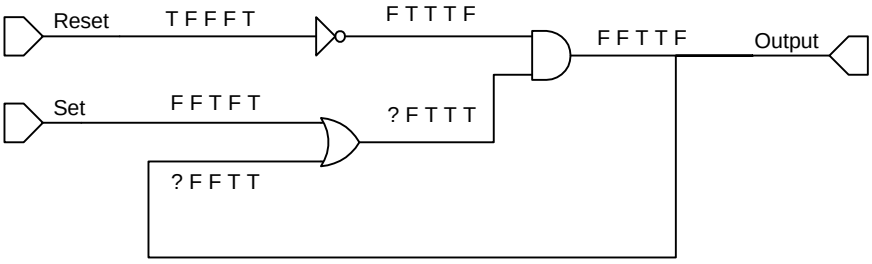
Step 5: Confirm output. Outputs match expected results.

Solution 3.5, Reset-Overides-Set Latch Exercise

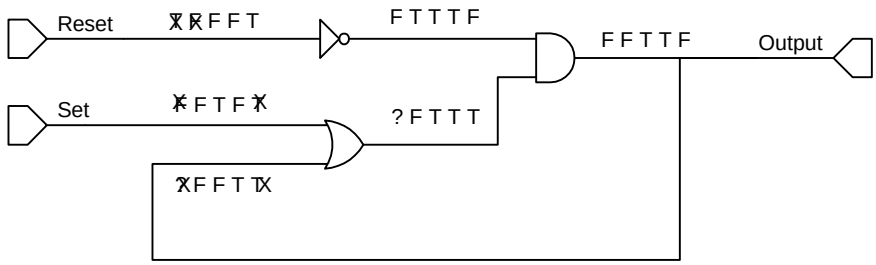
Step 1: Show the source code schematically.



Step 2: Map test cases to the source code picture. Note that the lower input to the ***or*** gate is delayed by one because it uses the past value of Output to compute the current value of Output.



Step 3: Eliminate masked tests.



Step 4: Determine MC/DC. The test set meets the MC/DC objectives.

Gate	Valid Test Inputs	Missing Test Cases
not	*F* Case 3 or 4 *T* Case 5	None
and	*TT* Case 3 or 4 *TF* Case 2 *FT* Case 5	None
or	*TF* Case 3 *FT* Case 4 *FF* Case 2	None

Step 5: Confirm output. Outputs computed match those provided.

This example is interesting because it shows the impact of using an output in the computation of itself. In this case, the value obtained in the last computation of the output feeds back into the computation. This results in the observation that the initial input to the ***or*** gate is indeterminate based on the test data provided. Test case 1 thus does not contribute to the MC/DC testing, but is required to provide a baseline state for the subsequent tests.

Appendix B

Certification Authorities Software Team Position Paper on Masking MC/DC

On February 6-7, 2001, the Certification Authorities Software Team (CAST) was given a presentation titled "Rationale for Accepting Masking MC/DC in Certification Projects", based on a white paper of the same title submitted to CAST. The CAST members in attendance concurred that masking MC/DC should be considered an acceptable method for meeting objective 5 of Table A-7 in DO-178B. However, the CAST requested revisions to the white paper. The revised version of the white paper, resulting from the CAST comments, appears below. This version of the paper has been re-submitted to CAST for review and approval at their next meeting. **Lest anyone think otherwise, please recognize that this white paper does not constitute regulatory software policy or guidance.**

Title: Rationale for Accepting Masking MC/DC in Certification Projects

Background

Structural coverage analysis in DO-178B (ref. B1) asks the question: Do the requirements-based test cases adequately exercise the structure of the source code? Two factors in exercising any structural element of the source code are: (a) the ability to test that element by setting the values of the element's inputs (this is the concept of controllability), and (b) the ability to propagate the output of that element to some observable point (this is the concept of observability). Controllability and observability are fundamental concepts used in testing logic circuits, and also apply well to testing software.

Different coverage measures found in Table A-7 of DO-178B address different structural elements of the code.

- For statement coverage, the structural elements to be exercised are the statements, and the adequacy requirement is that each statement must be executed at least once.
- For decision coverage, the structural elements to be exercised are the decisions, and the adequacy requirement is that each decision must take on each possible value at least once.
- For modified condition/decision coverage (MC/DC), the structural elements to be exercised are the logical conditions within a decision, and the adequacy requirement is that each logical condition must be shown to independently affect the decision's outcome.

Showing that each logical condition within a decision independently affects the decision's outcome requires a minimum test set for each logical operator as given in the original paper on MC/DC by Chilenski and Miller (ref. B2) and repeated here as follows:

- For a 2-input ***and*** operator, there is one test set: (TT, TF, FT).*
- For a 2-input ***or*** operator, there is one test set: (FF, TF, FT).
- For a 2-input ***xor*** operator, there are 4 possible test sets: (TT, TF, FT); (TF, FT, FF); (FT, FF, TT), and (FF, TT, TF).

These minimum test sets establish the inputs needed at a logical operator to show independent effect

* For convenience, the inputs to a test case are written as T for *true* and F for *false*. The notation TT, for example, represents a 2-input test case where both inputs are *true*.

of each input to that operator. Note that the minimum test sets are not exhaustive test sets and, hence, will not detect all possible errors. For example, a test set that contains the minimum tests for an ***or*** operator will not detect an error if an ***xor*** is incorrectly coded in place of an ***or***, and vice versa. However, the minimum test cases are sufficient to show independent effect required to meet the MC/DC criteria.

Two different approaches to confirming that the minimum tests are achieved are the unique-cause approach and the masking approach.

For unique-cause MC/DC, a condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.

For masking MC/DC, a condition is shown to independently affect a decision's outcome by applying principles of Boolean logic to assure that no other condition influences the outcome (even though more than one condition in the decision may change value).

Purpose

The purpose of this white paper is to establish that masking MC/DC:

- meets the definition of independent effect by guaranteeing the same minimum test cases at each logical operator as unique cause, and
- is acceptable for meeting the MC/DC objective of DO-178B (objective 5 in Table A-7).

Showing Independent Effect

A condition independently affects a decision's outcome if that condition *alone* can determine the value of the decision's outcome. Two test cases that show the independent effect of a condition within a decision are referred to as an independence pair.

Unique-Cause MC/DC

Unique cause is the original approach to showing the independent effect of a condition mentioned in the description of MC/DC in the DO-178B Glossary. In the unique-cause approach, only the values of the condition of interest and the decision's outcome can change between the two test cases in an independence pair–everything else must remain the same. Holding the value of every other condition fixed ensures that the one condition that changed value is the only condition that influences the value of the decision's outcome. The logic of the decision does not need to be examined to determine that the condition of interest is solely responsible for the change in the value of the decision's outcome.

A truth table is often used to illustrate the unique-cause approach. The left-hand columns of the truth table list all possible input combinations for the decision, while the shaded columns on the right indicate the possible independence pairs for each condition. The truth table for the decision **Z** = (**A** ***or*** **B**) ***and*** (**C** ***or*** **D**), where **A**, **B**, **C**, **D**, and **Z** are Boolean conditions, is shown in Table 1.

With unique cause, each test case can pair with at most one other test case to show independent effect of a condition. In Table 1, for example, test case 2 can only be paired with test case 10 to show the independent effect of **A**. The following are the possible independence pairs for each condition as shown in Table 1: test pairs (2, 10), (3, 11), and (4, 12) show the independent effect of **A**; (2, 6), (3, 7), (4, 8) show the independent effect of **B**; (5, 7), (9, 11), and (13, 15) show the independent effect of **C**; and (5, 6), (9, 10), and (13, 14) show the independent effect of **D**.

Table 1. Unique-Cause Approach to Independence Pairs for **Z** = (**A** ***or*** **B**) ***and*** (**C** ***or*** **D**)

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

Any combination of the independence pairs (with a minimum of one pair for each condition) will yield the minimum tests described by Chilenski and Miller for each logical operator. To see this, consider test cases 2, 5, 6, 7, and 10 from Table 1 for **Z** = (**A** ***or*** **B**) ***and*** (**C** ***or*** **D**). This set of test cases contains an independence pair for each condition. These test cases are mapped to a schematic representation of the code in Figure 1.

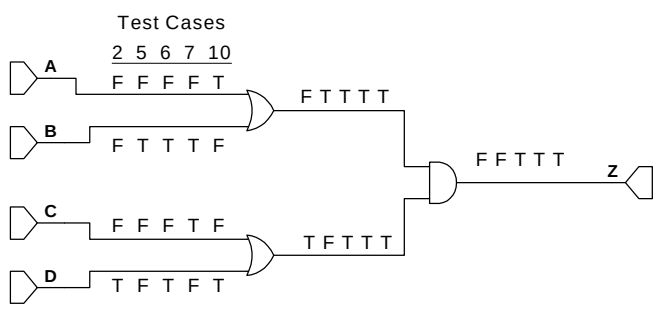


Figure 1. Schematic View of **Z** = (**A** ***or*** **B**) ***and*** (**C** ***or*** **D**) with test cases 2, 5, 6, 7, and 10

For a test case to count for credit towards MC/DC at a particular logical operator, the output of the test case at that operator must be observable. In Figure 1, the output of (**A** ***or*** **B**) is unobservable at **Z** for test case 5 because of the *false* output of (**C** ***or*** **D**) for that test. Similarly, the output of (**C** ***or*** **D**) is unobservable at **Z** for test case 2 because of the *false* output of (**A or B**). Discounting those test cases, there are FF, FT, and TF tests for each ***or*** operator, and TT, TF, and FT tests for the ***and*** operator.

The unique-cause approach guarantees the minimum tests for each logical operator for decisions *without* strongly coupled conditions (such as repeated conditions). The difficulty with coupled conditions stems from the DO-178B Glossary definition of *decision* that states that "If a condition appears more than once in a decision, each occurrence is a distinct condition". Hence, for the decision (**A** ***and*** **B**) ***or*** (**A** ***and*** **C**), showing independent effect by unique cause requires, among other things, showing what happens when the value of the first **A** is held constant, while the value of the second **A** is toggled between *false* and *true*. This typically cannot be accomplished in any meaningful way. Note that some developers may

establish coding standards that do not allow decisions with coupled conditions to avoid the coupling problem with unique-cause MC/DC.

Masking

Masking refers to the concept that specific inputs to a logic construct can hide the effect of other inputs to the construct. For example, a *false* input to an ***and*** operator masks all other inputs, and a *true* input to an ***or*** operator masks all other inputs. The masking approach to MC/DC allows more than one input to change in an independence pair, as long as the condition of interest is shown to be the ***only*** condition that affects the value of the decision outcome. However, analysis of the internal logic of the decision is needed to show that the condition of interest is the only condition causing the value of the decision's outcome to change.

To illustrate masking MC/DC, consider again the decision **Z** = (**A** ***or*** **B**) ***and*** (**C** ***or*** **D**). To show the independent effect of **A**, the subterm (**C** ***or*** **D**) must be *true* because the value of the decision's outcome will always be *false* if (**C** ***or*** **D**) is *false*. For unique-cause, the values of **C** and **D** must be fixed in any independence pair for **A**. However, masking allows **C** and **D** to change values in the independence pair for **A** *as long as* the outcome of (**C** **or** **D**) is *true*. In this way, the masking approach allows for more independence pairs for each condition than unique cause. For example, test case 2 could be paired with either test case 10, 11, or 12 to show the independent effect of **A** using masking. Table 2 shows the possible independence pairs for the example decision using the masking approach.

Table 2. Masking Approach to Independence Pairs for (**A** ***or*** **B**) ***and*** (**C** ***or*** **D**)

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

* Note that ***X*** and ***Y*** represent intermediate subterm (**A** ***or*** **B**) and subterm (**C** ***or*** **D**), respectively

Another way of examining the issue is to substitute proxy variables for all but the subterm of interest. For example, when looking for the independence of **A** or **B** in the above example, substitute the value of ***Y*** for the subterm (**C** ***or*** **D)** and then the same rules as unique-cause apply.

To verify that the minimum test cases exist for each logical operator with masking, consider the test cases 2, 5, 6, 7, and 12 shown with the schematic representation in Figure 2. This test set is the same as

the test set in Figure 1, except test case 12 has replaced test case 10. In terms of independence pairs, test cases 2 and 12 together form an independence pair for **A** under the definition of masking.

Note that the input values in Figure 2 are the same as in Figure 1 except for the inputs in test case 12 to (**C** ***or*** **D**). That is, the difference that masking makes for the independence pair for **A** is localized to a single logical operator. Note also, that the minimum tests at each operator are still observed in this example. Minimum tests for each logical operator are guaranteed in all cases with the masking approach, for decisions with non-coupled as well as coupled conditions.

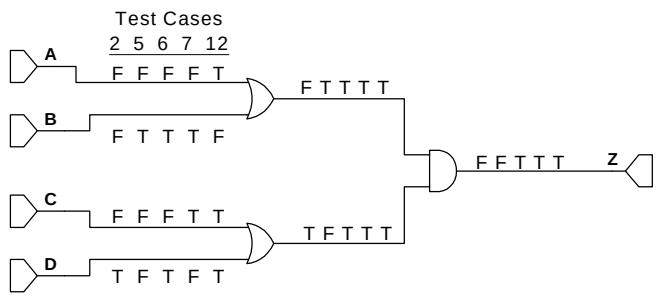


Figure 2. Schematic View of (**A** ***or*** **B**) ***and*** (**C** ***or*** **D**) with test cases 2, 5, 6, 7, and 12

Comparing Unique Cause and Masking

Both unique-cause MC/DC and masking MC/DC guarantee the minimum tests discussed by Chilenski and Miller at each logical operator of a decision—thus, showing the independent effect of each condition in the decision. Unique-Cause MC/DC and Masking MC/DC are identical for decisions with a common logical operator; that is, for decisions such as (**A** ***and*** **B** ***and*** **C**) or (**A** ***or*** **B** ***or*** **C**). When comparing unique cause and masking for decisions with mixed logical operators (e.g., (**A** ***or*** **B**) ***and*** **C**), the following points should be considered:

- In most cases, masking MC/DC allows more independence pairs per condition than unique-cause MC/DC. Any test set that satisfies unique cause will also satisfy masking; that is, masking independence pairs are a superset of unique-cause independence pairs. The advantage of having more independence pairs is the potential to reduce time for both humans and tools to determine whether the requirements-based test cases provide MC/DC.
- In general, the same number of test cases are needed to satisfy unique cause as masking.
- Masking MC/DC requires analysis of the decision logic to confirm either the minimum tests for each logical operator, or the independence pair for each condition in the decision. This analysis is not needed for unique-cause MC/DC.
- Masking MC/DC can be applied to decisions with coupled conditions. Hence, masking can be applied to decisions where unique-cause cannot be applied.

Note on Error Sensitivity

It is tempting to compare the unique-cause approach to masking with respect to the ability to detect errors. In a research project funded by the FAA, John Chilenski (of the Boeing Company) carried out a comparison of unique-cause MC/DC and masking MC/DC. The full report (ref. B3) will be available on the FAA web-site in the very near future (http://av-info.faa.gov/software). In a recent white paper,

Chilenski concluded that his analysis of error sensitivity between unique-cause and masking "has not shown that there is any significant difference" (ref. B4). However, reliance on such a comparison is unwise because: (a) the purpose of MC/DC is to determine the adequacy of the requirements-based tests to exercise the structure of the code—not to detect errors, (b) analysis of the ability of different test sets to detect different types of errors is extremely complex and subjective, and (c) Chilenski's work has not been reviewed by the engineering or academic community.

Summary

According to SC-190/WG-52 Frequently Asked Question #43 (ref. B5), structural coverage analysis complements requirements-based tests by:

1. Providing *"evidence that the code structure was verified to the degree required for the applicable software level";*
2. Providing "*a means to support demonstration of absence of unintended functions*"; and
3. Establishing "*the thoroughness of requirements-based testing*".

Masking MC/DC, as well as unique-cause MC/DC, satisfies all three of these "intents".

Both the unique-cause and masking approaches to MC/DC provide the same minimum tests of a logical operator in a decision. These minimum tests confirm that each condition independently affects the decision's outcome. The significant difference between the two approaches is that masking requires analysis of the logic of each decision, whereas unique-cause does not. Note that the masking MC/DC artifacts should be subject to the same planning, configuration management, and quality assurance requirements as any other artifact of the verification process.

When DO-178B was written, the research on masking MC/DC was still being carried out; therefore, unique-cause MC/DC was the technique documented. Since that time, research has shown that masking MC/DC also meets the intent of the MC/DC objective. Therefore, it is proposed that masking MC/DC be considered an acceptable method for meeting MC/DC by applicants striving to meet the objectives of DO-178B, level A.

References [for the white paper]

B1. *RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Inc., Washington, D. C., December 1992.

B2. Chilenski, John Joseph; and Miller, Steven. P.: Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, Vol. 7, No. 5, September 1994, pp. 193-200.

B3. Chilenski, John Joseph: *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*. FAA Tech Center Report DOT/FAA/AR-01/18, March 2001.

B4. Chilenski, John Joseph: *MCDC Forms (Unique-Cause, Masking) versus Error Sensitivity*. a white paper submitted to NASA Langley Research Center under contract NAS1-20341, January 2001.

B5. *RTCA/DO-248A, Second Annual Report for Clarification of DO-178B "Software Considerations in Airborne Systems and Equipment Certification"*. RTCA, Inc., Washington, D. C., September 13, 2000.

Appendix C

Background on Tutorial Authors

Ms. Kelly Hayhurst is a senior research scientist in the area of design correctness and certification at NASA Langley Research Center and has supported research on MC/DC with the FAA since 1996. Mr. Dan Veerhusen is a Principal Software Engineer and software DER in the Air Transport Systems Division of Rockwell Collins, Inc. Mr. Veerhusen is involved in software process development and verification techniques for Flight Control products. Mr. John Chilenski, an Associate Technical Fellow of The Boeing Company, primarily works in the area of verification and validation of software and systems. Mr. Chilenski is the co-author of "Applicability of modified condition/decision coverage to software testing", one of the first papers published on the topic. Ms. Leanna Rierson is the Federal Aviation Administration's Chief Scientific and Technical Advisor (also known as, National Resource Specialist) for Aircraft Computer Software.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
May 2001

3. REPORT TYPE AND DATES COVERED
Technical Memorandum

4. TITLE AND SUBTITLE
A Practical Tutorial on Modified Condition/Decision Coverage

5. FUNDING NUMBERS
WU 728-30-10-03

6. AUTHOR(S)
Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L-18088

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA/TM-2001-210876

11. SUPPLEMENTARY NOTES
Hayhurst: Langley Research Center, Hampton, VA; Veerhusen: Rockwell Collins, Inc., Cerdar Rapids, IA; Chilenski: The Boeing Company, Seattle, WA; Rierson: Federal Aviation Administration, Washington, D.C.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 61 Distribution: Nonstandard
Availability: NASA CASI (301) 621-0390

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
This tutorial provides a practical approach to assessing modified condition/decision coverage (MC/DC) for aviation software products that must comply with regulatory guidance for DO-178B level A software. The tutorial's approach to MC/DC is a 5-step process that allows a certification authority or verification analyst to evaluate MC/DC claims without the aid of a coverage tool. In addition to the MC/DC approach, the tutorial addresses factors to consider in selecting and qualifying a structural coverage analysis tool, tips for reviewing life cycle data related to MC/DC, and pitfalls common to structural coverage analysis.

14. SUBJECT TERMS
Modified condition/decision coverage; MC/DC; Software; Certification; DO-178B; Structural coverage

15. NUMBER OF PAGES
85

16. PRICE CODE
A05

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL