MC/DC Supplementary v5

Section Supplement

Software Testing
TTA - Methodologies
Guide for ISTQB Exam CTAL-TTA

ISTQB® TTA
2021 Syllabus v4

September 2025

Catherine Newbould BSc. (Hons.), AKC

More Information for Modified Condition/decision coverage

Due to time constraints, the exam is unlikely to require MC/DC test cases to be created from a given scenario; rather, it will assess the suggested test cases to determine if they provide adequate MC/DC coverage. Thus, the primary focus of the book is on approaches for analyzing test cases provided, not creating them. However, this supplementary information is provided for those who want to understand MC/DC testing in greater depth.

Methodologies for determining the minimum number of test cases for full MC/DC coverage.
For full consideration of the minimum number of MC/DC, there are two primary formal methods: unique-cause and masking. The truth tables are typically the starting point for these methods. For simple, relatively uncomplicated code, it may not be necessary to use a formal approach with a truth table. However, using the truth table can help reduce the risk of errors. A combination of both unique-cause and masking can be used.

The general principles are:

- The condition must independently affect the decision outcome
- Each condition must be shown to take both true and false values
- Conditions may or may not change, depending on the method used, and provided they are not responsible for changing the outcome

There can be multiple valid sets of test cases that satisfy MC/DC requirement.

Informal MC/DC
Instead of analyzing every possible combination of conditions as in formal methods, such as unique-cause, an informal Modified Condition/Decision Coverage MC/DC method has a less rigid approach to satisfying the MC/DC requirements. It can be used when strict formal methods are not feasible or necessary.

Characteristics of the informal method are:

- Highlights how each condition contributes to the decision through selected tests
- Allows more flexibility, such as using manual reasoning or tools to justify independence
- May use Masking MC/DC**,** where other conditions can change, provided the tested condition's effect is still observable
- Relies on testers' understanding of the logic

Unique-cause MC/DC
Unique-cause MC/DC is a formal testing method that examines all input combinations of conditions. It checks if a condition independently affects a decision. The condition must change the decision's outcome by itself. All other conditions must stay the same during the test.

Characteristics of unique-cause are:
- Ensures no masking by other conditions.
- Each test pair demonstrates the cause-effect relationship resulting from a single condition change.

Masking MC/DC
Masking MC/DC allows more than one condition to change between test pairs, as long as the effect of one specific condition can still be isolated. This may be necessary due to constraints such as unreachable combinations and where one condition cannot be isolated.

Masking MC/DC accepts more flexible test pairs, provided:
- The effect of the tested condition is provable, and
- The effect of other changing conditions is logically irrelevant, masked by logic.

Although not as strict as unique-cause, masking still meets the avionics DO-178C Level A MC/DC coverage criteria and those of many other regulations.

Examples:
Consider the following scenario and the examples of using informal, unique-cause and masking methods.

Scenario: You are tasked with testing software that determines if the aircraft should divert to a different destination airport. Boolean conditions handled by the code are:

high_ fuel_consumption:	true if the aircraft's fuel consumption rate is higher than expected, leaving insufficient fuel to reach the destination.
deteriorating_weather:	true if weather conditions at the destination are deteriorating rapidly.
low_fuel:	true if the aircraft is unable to hold for an extended period due to limited fuel reserves.

The logic in the code is:

IF (higher_fuel_consumption) OR (deteriorating_weather AND low_fuel) THEN
    divert to the nearest suitable alternate airport.

T, true and F, false are used below. In some sources, 0 is used to represent false, and 1 to represent true.

Example 1: Using Informal MC/DC method to determine the minimum test cases required for full MC/DC coverage for the scenario above. This is the least formal of the methods.

Step 1: Choose two test cases where changing deteriorating_weather and keeping low_fuel and higher-fuel-consumption unchanged:
Test case 1. F (T, T) outcome divert
Test case 2. F (F, T) outcome no divert

Step 2: Choose two test cases where changing low_fuel and keeping deteriorating_weather and higher-fuel-consumption unchanged changes the outcome:
Test case 3: F (T, T) outcome divert
Test case 4: F (T, F) outcome no divert

Step 3: Choose two test cases where changing higher-fuel-consumption and keeping low_fuel and keeping deteriorating_weather and low_fuel unchanged changes the outcome:
Test case 5. T (F T) outcome divert
Test case 6. F (T F) outcome no divert

Step 4: Inspect test cases:
Test case 1. F (T, T) outcome divert
Test case 2. F (F, T) outcome no divert
Test case 3. F (T, T) outcome divert
Test case 4. F (T, F) outcome no divert
Test case 5. T (F, T) outcome divert
Test case 6. F (T, F) outcome no divert

Remove duplicate test cases and in the remaining test cases check outcomes change:
Test case 4 and test case 6 are the same, remove one of these
Test case 1 and test case 3 are the same, remove one of these

This leaves:
Test case 1. F (T, T) outcome divert
Test case 2. F (F, T) outcome no divert
Test case 5. T (F, T) outcome divert
Test case 4. F (T, F) outcome no divert

Test cases 1 and 2 show the effect of changing deteriorating_weather
Test cases 1 and 4 show the effect of changing low_fuel changing
Test cases 2 and 5 show the effect of changing higher_fuel_consumption

Example: 2 Using Unique-Cause MC/DC to determine the minimum test cases required for full MC/DC coverage for the scenario above.

Step 1: Create a table showing all options. For an overview of how to do this, see 2.5 Multiple condition testing. For more details, see Decision Tables in Advanced Test Analyst by C. Newbould.

Test Case	High_fuel_ consumption	Deteriorating_weather	Low_fuel	Divert_other_airport
1	true	true	true	true / divert
2	true	true	false	true / divert
3	true	false	true	true / divert
4	true	false	false	true / divert
5	false	true	true	true / divert
6	false	true	false	false / no divert
7	false	false	true	false / no divert
8	false	false	false	false / no divert

Step 2: Identify a pair of test cases where the outcome results are different with only one condition changing.
For example: Test case 3 T, (F, T) outcome divert and Test case 7 F, (F, T) outcome no divert.

Step 3: Next use test case 5 F, (T, T), outcome divert). It changes one condition compared to test case 7.

Step 4: Then use test case 6 (F, T, F) outcome no divert.

The outcomes alternate:
Test case 3: T, (F, T) divert
Test case 7: F, (F, T) no divert , higher_fuel_consumption changed from test case 3
Test case 5: F, (T, T) divert, deteriorating weather changed from test case 7
Test case 6: F, (T, F) no divert, low_fuel changed from test case 5

Alternative possible for Step 2: If the two test cases selected with different divert condition outcomes are: Test case 6 F, (T, F) outcome no divert and Test case 5 F, (T, T) outcome divert.

Alternative possible for Step 3: Use test case 7 F, (F, T) outcome no divert.
followed by test case 3 T, (F, T) outcome divert.

Test case 6: F, (T, F) no divert
Test case 5: F, (T, T) divert, low-fuel changes
Test case 7: F, (F, T) no divert, deteriorating weather changes
Test case 3: T, (F, T) divert high fuel consumption changes

Example: 3 Using Masking MC/DC, to determine the minimum test cases required for full MC/DC coverage for the scenario above.

Step 1: Create a table. For an overview of how to do this, see 2.5 Multiple condition testing. For more detail see Decision Tables in Advanced Test Analyst by C. Newbould.

Test case	High_fuel_consumption	Deteriorating_weather	Low_fuel	Divert_other_airport
1	true	true	true	true / divert
2	true	true	false	true / divert
3	true	false	true	true / divert
4	true	false	false	true / divert
5	false	true	true	true / divert
6	false	true	false	false / no divert
7	false	false	true	false / no divert
8	false	false	false	false / no divert

Step 2: Empty cells are where value has no bearing on outcome.

Test case	High_fuel_consumption	Deteriorating_weather	Low_fuel	Divert_other_airport
1	true			true / divert
2	true			true / divert
3	true			true / divert
4	true			true / divert
5		true	true	true / divert
6	false	true	false	false / no divert
7	false	false		false / no divert
8	false	false		false / no divert

Cells above that are now blank indicate the condition has no bearing on the outcome.

Step 3: Test case 6 should be chosen as there are values in all cells. The outcome is false.
Test case 6 F (T, F) changes low_fuel from test case 5, outcome no divert.

Step 4 A test case that changes one of the values inside the brackets and the outcome should be chosen:
Test case 5 F (T, T) outcome divert.

Step 4 A test case that changes the value of the other condition inside the brackets and the outcome should be chosen:
Ttest case 7 F (F, T) changes deteriorating_weather from test case, 5 outcome no divert.

Step 5 A test case changing the value of the last condition to be evaluated and changes the outcome should be chosen:
Test case 3 T (F, T) changes higher_fuel consumption, test case 6, outcome divert.

Short-circuiting
Modern languages, such as Python, JavaScript, Java, C#, Go, Swift, support short-circuiting. By default. Older languages might not use short-circuiting.

Many sources mix short-circuiting principles, particularly with the masking method, as there is some commonality in the idea that some conditions need not be evaluated. This can be confusing for determining test cases.

The number of MC/DC test cases can only be reduced if it is specified that masked conditions are excluded from needing proof of their independent effect. When using short-circuit logic, not all conditions may be evaluated, leading to the possibility that certain MC/DC test cases could be seen as duplicates.

Example, MC/DC and Short-circuiting
With the scenario above and the M/C/DC method examples, four test cases are required: With default short-circuiting, this reduces to three distinct test cases, as evaluation of
(deteriorating_weather AND low-fuel) false is made as soon as deteriorating_weather is determined to be false, and low-fuel is not then checked.

Test Case	MC/DC Test case no short-circuiting	MC/DC Test case with short-circuiting
1	F (T, T)	F (T)
2	F (T, F)	F (F)
3	T (T, F)	T (F
4	F (F, T)	F (F)

(deteriorating_weather and low_fuel) is evaluated first, according to BEMDAS; Brackets, Exponents, Multiplication, Division, Addition and Subtraction, from left to right.
With short-circuiting, test cases 2 and 4 are the same.

Note: An exam question may assume a modern programming language is being used or only give a correct option that assumes short-circuiting. The other options in the question do not achieve MC/DC proper or reduced coverage, or a defined coverage requirement.

Proper MC/DC testing must be used to show that every condition is independently evaluated as both true and false and the outcome changes. If MC/DC test cases are reduced, the quality of coverage may be compromised.

Best practices for proper MC/DC testing include:
- Understand the software language's evaluation rules.
- Use coverage tools that track condition evaluation, not just decision outcomes.
- Consider code reviews and manual analysis for safety-critical software.

Test cases can be removed in practice. For example, during test optimization or runtime testing providing:

Testing is only concerned with runtime behavior**,** not strict coverage metrics such as MC/DC.
Testing is using black-box testing, and the skipped conditions cannot affect the outcome or behavior.
The software under test includes logic that is not safety-critical, and basic coverage, such as decision or condition coverage, is considered adequate.
Testing is validating optimization of code paths such as performance), not logic correctness.

See **Pages 78 and 79** in the Software Testing Technical Test Analyst guide book.