An introduction to MC/DC coverage

Marco Di Natale
Scuola Superiore S. Anna- Pisa, Italy

Taken from
Model-based testing

Purpose of this Lesson

- Learn about structural coverage
- Learn about MC/DC coverage and how to apply it
More on MCC

- Catastrophic
- Hazardous/Severe-major
- Major
- Minor
- No Effect

**Level A**: funzioni hardware/software il cui malfunzionamento potrebbe causare condizioni d'avaria catastrofiche per l'aeromobile.

**Level B**: funzioni hardware/software il cui malfunzionamento potrebbe causare condizioni d'avaria Hazardous/Severe-major per l'aeromobile.

**Level C**: funzioni hardware/software il cui malfunzionamento potrebbe causare condizioni d'avaria Major per l'aeromobile.

**Level D**: funzioni hardware/software il cui malfunzionamento potrebbe causare condizioni d'avaria Minor per l'aeromobile.

**Level E**: funzioni hardware/software il cui malfunzionamento non ha conseguenze.
Testing

"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?"

Wiener, Ruth: Digital Woes, Why We Should Not Depend on Software.

- The purpose of the verification process is to detect and report errors that have been introduced in the development process.
- The verification process must ensure that the produced software implements intended function completely and correctly, while avoiding unintended function.
- Verification is an integral process, which is coupled with every development step. Testing quality at the end of the life cycle is impractical.
Model-based testing

DO-178B
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.”

**Coverage**

- Coverage refers to the extent to which a given verification activity has satisfied its objectives: 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. As a measure, coverage is usually expressed as the percentage of an activity that is accomplished.
Coverage

Two specific measures of test coverage are

- **Requirements coverage analysis**: how well the requirements based testing verified the implementation of the software requirements
  - Requires traceability between the software requirements and the test cases.
- **Software structure coverage (structural coverage)** determines how much of the code structure was executed by the requirements-based tests
  - establishes traceability between the code structure and the test cases
- Requirements coverage analysis precedes structural coverage analysis.
Requirements coverage analysis

• 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.
Requirements coverage analysis

• 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.
Requirements coverage analysis

“During the development of any non-trivial program, software structure is almost always created that cannot be determined from top-level software specifications”.

- Derived requirements 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.
Requirements 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 implemented code structure has been
adequately tested and does not contain any unintended functionality.
Structural Coverage Analysis

- Structural coverage analysis provides a means to confirm that “the requirements-based test procedures exercised the code structure”.
- 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 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.
Structural Coverage Analysis

• 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.
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.

• Note 1: 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.
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.
## Model-based testing

<table>
<thead>
<tr>
<th>Coverage Criteria</th>
<th>Statement Coverage</th>
<th>Decision Coverage</th>
<th>Condition Coverage</th>
<th>Condition/Decision Coverage</th>
<th>MC/DC</th>
<th>Multiple Condition Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Every point of entry and exit in the program has been invoked at least once</td>
<td></td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Every statement in the program has been invoked at least once</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Every decision in the program has taken all possible outcomes at least once</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Every condition in a decision in the program has taken all possible outcomes at least once</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Every condition in a decision has been shown to independently affect that decision’s outcome</td>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
<td>• 8</td>
<td></td>
</tr>
<tr>
<td>Every combination of condition outcomes within a decision has been invoked at least once</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>•</td>
</tr>
</tbody>
</table>

Weakest to strongest
Statement coverage

• To achieve statement coverage, every executable statement in the program is invoked at least once during software testing.
  – shows that all code statements are reachable (reachable based on test cases developed from the requirements).
• Statement coverage is considered a weak criterion because it is insensitive to some control structures.
• Example:
  
  ```
  if (x > 1) && (y = 0)
      z = z / x;
  if (z = 2) || (y > 1)
      z := z + 1;
  ```
• 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.
• Analysis of logic expressions is not part of the statement coverage criterion.
Statement coverage

– 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 (\(A \text{ or } B\), where \(A \text{ and } B\) are both conditions.)
Decision coverage (DC)

- Decision coverage requires two test cases for each decision: 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.
Condition coverage (CC)

- 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 possible outcomes at least once.

- In this case, for the decision \((A \text{ or } B)\) 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.
Condition/Decision coverage (C/DC)

- 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 (A and B).
Modified Condition/Decision coverage (MC/DC)

• 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.
Multiple Condition coverage (MCC)

- 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 $2n$ tests.10
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”

• Structural testing is the process of exercising software with test scenarios written from the source code, not from the requirements.
  – It does not meet the DO-178B objective that all code structure is exercised by requirements-based procedures.

• The correct approach when structural coverage analysis identifies untested code is to consider the possible causes
  – If any additional testing is required, it should be requirements-based testing, using high-level, low-level, or derived requirements, as appropriate.
Structural coverage analysis and structural testing

- Structured 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 not implemented in the code.
- Structural testing provides no information about whether the code is doing what it is supposed to be doing as specified in the requirements.
  - Since the code itself is used as the basis of the test cases, structural testing may fail to find simple coding errors.
  - It is a natural tendency to consider outputs of the actual code as the expected results.
- 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.
- 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**
More on MCC

• According to legend, there were once folks who advocated requiring 100% multiple condition coverage (that is, exhaustive testing) for level A software.
  – The motivation is 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.

• The table shows the number of Boolean expressions with $n$ conditions for all of the logic expressions taken from the airborne software of five different Line Replaceable Units (LRUs) from level A systems. (from two airplane models in 1995). As the table shows, actual code has been written with more than 36 conditions.

<table>
<thead>
<tr>
<th>Number of Conditions, $n$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6-10</th>
<th>11-15</th>
<th>16-20</th>
<th>21-35</th>
<th>36-76</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Boolean expressions with $n$ conditions</td>
<td>16491</td>
<td>2262</td>
<td>685</td>
<td>391</td>
<td>131</td>
<td>219</td>
<td>35</td>
<td>36</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

• Clearly, multiple condition coverage is impractical for systems such as these
More on MCC

- 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.”
- 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.
- Q1: Consider an expression with 36 inputs. How much time would it take to execute all of the test cases required for multiple condition coverage of this expression if you could run 100 test cases per second?
- A1:
- Q2: 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.)
- A2:
More on MCC

• The requirement to show the \textit{independent effect of each condition within a decision makes MC/DC} unique among coverage criteria.

• Determining whether a condition has independent effect might seem rather simple: a condition has independent effect when that condition \textit{alone determines the outcome of the decision}.

• \textit{Condition}–A Boolean expression containing no Boolean operators.

• \textit{Decision}–A Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. \textit{If a condition appears more than once in a decision, each occurrence is a distinct condition}.

• \textit{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.
More on MCC

• 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 \((A \text{ and } B) \text{ or } (A \text{ and } C)\), 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.
More on MCC

• 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.
More on MCC

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

• E := (B or C) and D; (statement 3)

• 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.
More on MCC

• 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., \((A \text{ and } B) \text{ or } (A \text{ and } C)\).
More on MCC

• The unique-cause approach commonly is taught by presenting a truth table for an expression; for example, the decision \( Z := (A \text{ or } B) \text{ and } (C \text{ or } D) \)

• 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. The columns in gray indicate the independence pairs for each condition. For example, test 2 coupled with test 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 \( Z \).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Z</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>10</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>11</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>12</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td>7</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>2</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td></td>
<td></td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More on MCC

- **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.
Gate-level representation for MC/DC

- 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:
  - 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.
Gate-level representation for MC/DC

- 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 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 sections.
  - basic building blocks that are fundamental to the MC/DC approach.
  - steps of the approach.
  - how to apply the approach to different scenarios, such as grouped functionality, short-circuit control forms, and bit-wise operations.
  - resolving errors or shortcomings identified through the structural coverage analysis.
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. Representation of the elementary logical operators: and, or, xor, and not.

<table>
<thead>
<tr>
<th>Name</th>
<th>Schematic Representation</th>
<th>Code example</th>
<th>Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td><img src="image" alt="Input Diagram" /></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td><img src="image" alt="Output Diagram" /></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| and Gate | ![And Gate Diagram](image) | $C := A \text{ and } B$; | $\begin{array}{c|c|c|c}
A & B & C \\
T & T & T \\
T & F & F \\
F & T & F \\
F & F & F \\
\end{array}$ |
| or Gate | ![Or Gate Diagram](image) | $C := A \text{ or } B$; | $\begin{array}{c|c|c|c}
A & B & C \\
T & T & T \\
T & F & T \\
F & T & T \\
F & F & F \\
\end{array}$ |
| xor Gate | ![Xor Gate Diagram](image) | $C := A \text{ xor } B$; | $\begin{array}{c|c|c|c}
A & B & C \\
T & T & F \\
T & F & T \\
F & T & T \\
F & F & F \\
\end{array}$ |
| not Gate | ![Not Gate Diagram](image) | $B := \text{ not } A$; | $\begin{array}{c|c|c}
A & B \\
T & F \\
F & T \\
\end{array}$ |
MC/DC Building Blocks ("how do I test a …?")

• 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.**
MC/DC Building Blocks ("how do I test a …?")

- 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).

<table>
<thead>
<tr>
<th>Test Case Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input A</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Input B</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>Input C</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>Output D</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
MC/DC Building Blocks (“how do I test a …?”)

- 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).

<table>
<thead>
<tr>
<th>Test Case Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input A</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Input B</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>Input C</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>Output D</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
MC/DC Building Blocks ("how do I test a …?")

- A comparator evaluates two numerical inputs and returns a Boolean based on the comparison criteria. 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).
MC/DC Building Blocks (“how do I test a …?”)

- 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-equalto 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 \( x = 5000 \).
MC/DC Building Blocks ("how do I test a …?")

- 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:
  - Input x set at a value slightly above the comparison point
  - Input x set at a value slightly below the comparison point
  - 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).

<table>
<thead>
<tr>
<th>Test Case Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input altitude</td>
<td>25</td>
<td>32000</td>
<td>2500</td>
<td>2499</td>
<td>2501</td>
</tr>
<tr>
<td>Output A</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

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

- 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;

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

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

A minimal test set for the statement if $Z$ then $a := x$ else $a := y$ is shown in Table 9.

<table>
<thead>
<tr>
<th>Test Case Number</th>
<th>1 Traverse the then path</th>
<th>2 Traverse the else path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input $x$</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td>Input $y$</td>
<td>50</td>
<td>34</td>
</tr>
<tr>
<td>Input $Z$</td>
<td>$T$</td>
<td>$F$</td>
</tr>
<tr>
<td>Output $a$</td>
<td>12</td>
<td>34</td>
</tr>
</tbody>
</table>
Bibliography

Taken from