A PRACTICAL APPROACH TO
MODIFIED CONDITION/DECISION COVERAGE

Kelly J. Hayhurst, NASA Langley Research Center, Hampton, Virginia

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

Abstract

Testing of software intended for safety-critical applications in commercial transport aircraft must achieve modified condition/decision coverage (MC/DC) of the software structure. This requirement causes anxiety for many within the aviation software community. Results of a survey of the aviation software industry indicate that many developers believe that meeting the MC/DC requirement is difficult, and the cost is exorbitant. Some of the difficulties stem, no doubt, from the scant information available on the subject. This paper provides a practical 5-step approach for assessing MC/DC for aviation software products, and an analysis of some types of errors expected to be caught when MC/DC is achieved[1].

Introduction

Software has become the medium of choice for enabling advanced automation in aircraft, and also in ground and satellite-based systems that manage communication, navigation, and surveillance for air traffic control. As the capability and complexity of software-based systems increases, so does the challenge of verifying that these systems meet their requirements, including safety requirements. For systems that are safety and mission critical, extensive testing is required. However, the size and complexity of today's avionics products prohibit exhaustive testing.

The RTCA/DO-178B document *Software Considerations in Airborne Systems and Equipment Certification* [1] is the primary means used by aviation software developers to obtain Federal Aviation Administration (FAA) approval[2] of airborne computer software [2]. DO-178B describes software life cycle activities and design considerations, and enumerates sets of objectives for the software life cycle processes. For level A software (that is, software whose anomalous behavior could have catastrophic consequences), DO-178B requires that testing achieve modified condition/decision coverage (MC/DC) of the software structure. MC/DC is a structural coverage measure consisting of four criteria mostly concerned with exercising Boolean logic. The MC/DC criteria were developed to provide many of the benefits of exhaustive testing of Boolean expressions without requiring exhaustive testing [3].

Results of a 1999 survey of the aviation software industry showed that more than 75% of the survey respondents claimed meeting the MC/DC requirement in DO-178B was difficult, and 74% of the respondents said the cost was either substantial or nearly prohibitive [4]. Much of the cost of verifying level A software is often attributed to meeting the MC/DC objective. Additionally, many claim that the effectiveness of MC/DC with respect to finding errors is marginal at best. A recent case study by Dupuy and Leveson [5] found that testing augmented to satisfy MC/DC "while relatively expensive, was not significantly more expensive than achieving lower levels of code coverage. Important errors were found by the additional

[2] ED-12B 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 in Europe.

test cases required to achieve MC/DC coverage (i.e., in the software found not to be covered by blackbox functional testing)."

Definitions

Knowing the DO-178B glossary description of MC/DC plus the descriptions for *condition* and *decision* [1] is essential to building a working understanding of MC/DC.

> *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.*

Given the descriptions above, we note the following, often misunderstood, points:

- MC/DC applies to *every* Boolean expression. That is, MC/DC applies to assignment statements such as *Z:= A or B* and to statements such as *if A and B then ...*

- The number of inputs to a given decision may differ from the number of conditions. For example, the decision (**A** *and* **B**) *or* (**A** *and* **C**), where **A**, **B**, and **C** are Boolean variables, contains 3 inputs (**A**, **B**, and **C**) and 4 conditions (first **A**, **B**, **C**, and second **A**) because each occurrence of **A** is considered a unique condition.

Intent of MC/DC

The MC/DC criteria were developed by Chilenski and Miller to achieve a degree of confidence in the software comparable to that provided by exhaustive testing, while requiring fewer test cases [3]. That is, MC/DC is intended to assure, with a high degree of confidence, that requirements-based testing has demonstrated that each condition in each decision in the source code has the proper effect.

In the context of DO-178B, MC/DC serves as a measure of the adequacy of requirements-based testing—especially with respect to exercising logical expressions. In that regard, MC/DC is often used as an exit criterion (or one aspect of the exit criteria) for requirements-based testing. The RTCA/DO-248A document *Second Annual Report for Clarification of DO-178B "Software Considerations in Airborne Systems and Equipment Certification"* [6] explains the purpose of structural coverage analysis as follows:

> 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.
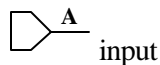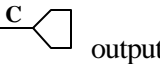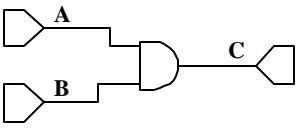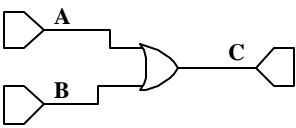>
> …
>
> The rationale is that if requirements-based testing proves that all intended functions are properly implemented, and if structural coverage analysis demonstrates that all existing code is reachable and adequately tested, these two together provide a greater level of confidence that there are no unintended functions.

MC/DC Fundamentals

The requirement to show the independent effect of each condition within a decision sets MC/DC apart from other structural coverage measures. According to Chilenski and Miller, showing that each logical condition independently affects a decision's outcome requires specific minimal test criteria for each logical operator [3]. Knowing and understanding the minimal test criteria for two logical operators is a sufficient basis, in most cases, for determining compliance with the MC/DC objective.

Understanding how to test a logical ***and*** operator and a logical ***or*** operator is essential to understanding MC/DC. For the analysis presented here, logical operators are shown schematically as logical gates; and, the terms "logical operator" and "gate" are used interchangeably. Table 1 shows schematic representations for the ***and*** and ***or*** operators. Note that Boolean operators are denoted by bolded italics: ***and***, and ***or***; Boolean conditions are denoted by bolded capital letters: **A**, **B**, **C**, …; and, Boolean outcomes are denoted *true* or *false* or *T* or *F*.

Table 1. Representations for Elementary Logical Gates

Schematic Representation	Truth Table
A input	
C output	
A B C := A *and* B;	A B C T T T T F F F T F F F F
A B C := A *or* B;	A B C T T T T F T F T T F F F

The following subsections describe the minimum test criteria for an ***and*** and an ***or*** gate.

Testing an and Gate

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

(1) A single test case where all inputs are set *true* with the output observed to be *true.*

(2) Test cases such that 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 test criteria make sense when considering how an ***and*** gate works. Any *false* input to an ***and*** gate will result in a *false* output. We show independent effect by complementing a test case consisting of all *true* inputs with test cases that set one and only one input *false* until each individual input has been shown to influence the output.

Hence, a specific set of *n*+1 test cases is needed to provide coverage 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 1) is given in Table 2. In this example, test case 1 in Table 2 provides the coverage for (1) above, and test cases 2-4 provide coverage for (2).
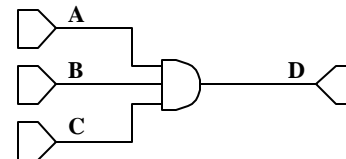
Figure 1. 3-input *and* gate

Table 2. Minimum Tests for a 3-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

With respect to showing independent effect, test cases 1 and 2 together show the independent effect of **A** because the value of **A** is the only input

value that changes along with the outcome value between those two test cases. Similarly, test cases 1 and 3 together show the independent effect of **B**; and test cases 1 and 4 together show the independent effect of **C**.

Testing an or Gate

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

(1) A single test case where all inputs are set *false* with the output observed to be *false*.

(2) A set of test cases where 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 2) is given in Table 3. In this example, test case 1 provides the coverage for (1) while test cases 2-4 provide the coverage for (2). The test pairs that show the independent effect of each input are similar to those for the *and* gate.
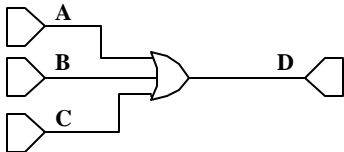

Figure 2. 3-input *or* gate

Table 3. Minimum Tests for a 3-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

A Note About the xor Gate

The *xor* gate is different with respect to MC/DC from the *and* and *or* gates. The *and* and *or* gates each have only one possible minimum test set. For the *xor*, Chilenski and Miller defined four possible minimum test sets that provide MC/DC for a 2-input *xor* gate: (*TT*, *TF*, *FT*), (*TF*, *FT*, *FF*), (*FT*, *TT*, *FF*), and (*TT*, *FF*, *TF*).

Note, however, that an *xor* operation may be viewed as a combination of *and* and *or* operations; and, test criteria for the *xor* can be derived from the minimum test criteria for the *and* and *or* operators. The expression **A** *xor* **B** can be rewritten (**A** *or* **B**) *and* *not* (**A** *and* **B**). This implementation of the *xor* requires four test cases, (that is, exhaustive testing), to provide MC/DC. In this case, analysis of this implementation of *xor* suggests that exhaustive testing of *xor* operations may be prudent.

The *xor* operation is used as an example to illustrate our approach to evaluating MC/DC and, incidentally, to demonstrate why exhaustive testing may be desirable for *xor* operations.

Evaluation Approach

This section presents a practical approach for evaluating whether a given set of requirements-based test cases conforms to three of the four requirements for MC/DC[3]:

- 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 evaluation approach builds on the minimum test cases for the *and* and *or* gates using two concepts taken from logic circuit theory: controllability and observability [7]. For software, controllability can be described loosely as the ability to set the values of an expression's inputs in order to test each logical operator (this corresponds

---

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

to meeting the minimum test criteria). Observability refers to the ability to propagate the output of a logical operator under test to an observable point.

To evaluate MC/DC using a gate-level approach, each logical operator in a decision in the source code is examined to determine whether the requirements-based tests have observably exercised the operator using the minimum test criteria. This approach 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 for a specific gate is one whose results are hidden from the observed outcome.

(4) Determine MC/DC based on the minimum test criteria for each operator.

(5) Finally, examine the outputs of the tests to confirm correct operation of the software. The point is not to repeat the analysis of the requirements-based test results, but rather to confirm that the schematic representation of the source code provides the same results. If an expected result in the test case does not match an 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.

Source Code Representation

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

Consider the following line of Ada source code:

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

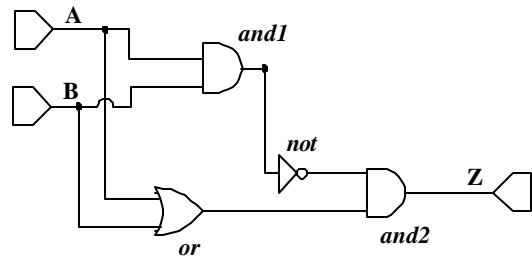This source code is shown schematically in Figure 3.

Figure 3. Schematic representation of source code

Although the example uses Ada code, the evaluation approach applies to all source code regardless of whether it is written in a high-level language such as Ada or in assembly language.

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 the example code are given in Table 4.

Table 4. Requirements-based Test Cases for Example

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

Recall that the source code in this example is implementing an *xor* operation. The test cases given in Table 4 provide MC/DC of an *xor* operator according to Chilenski and Miller; hence the test cases in Table 4 may be considered reasonable requirements-based tests. Figure 4 shows the test cases annotated on the schematic representation. Note that intermediate results have also been determined from the test inputs and shown on the schematic representation.
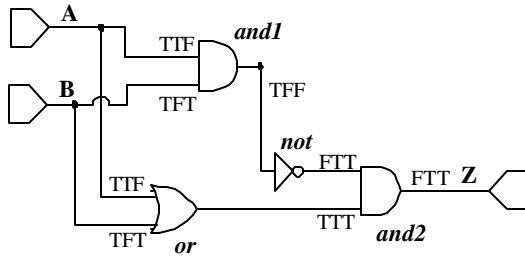
Figure 4. Schematic representation with test cases

Knowing the intermediate results is important because they provide the basis for determining which test cases do or do not contribute to valid MC/DC results. Test cases where the output is masked do not contribute to achieving MC/DC.

Elimination of Masked Tests

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 minimum test criteria to determine if they are sufficient to meet the MC/DC criteria.

This step is necessary to achieve observability. Only test cases whose outputs are observable (at **Z** in this example)[4] can be counted for credit towards MC/DC. An electrical analogy of "shorting" various "control inputs" such that they allow the "input of interest" to be transmitted through to the output is helpful in describing 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, the experimental input will be referred to as the input of interest and the other inputs as the control inputs. The truth table for an *and* gate in Table 5 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 at the output in the case where the control input is *false*.

---

[4] This assumes that no intermediate results were captured as part of the test results.

Table 5. 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 6 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 at the output in the case where the control input is *true*.

Table 6. 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*.

To determine which test cases are masked, it is easiest to work backwards through the schematic diagram. Consider again the expression **(A** *or* **B)** *and not* **(A** *and* **B)** shown in Figure 4. The *false* input to the gate labeled *and2* masks the corresponding input coming from the *or* gate. That is, the output of the *or* gate for test case 1 cannot be determined by looking at the results at **Z**. Hence test case 1 cannot be counted for credit towards MC/DC for the *or* gate. Figure 5 shows that test case 1 is eliminated for the *or* gate. Note that no test cases are masked for the *and1* gate.
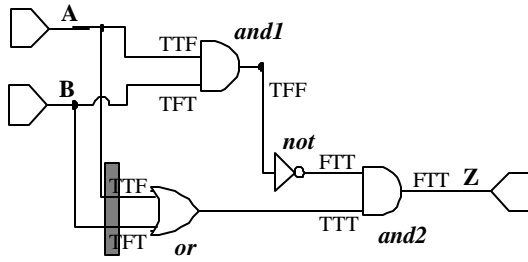
Figure 5. Schematic representation with masked test cases

Test cases that are not identified as masked test cases are considered valid for (or count for credit towards) MC/DC. In Figure 5, test cases 1, 2, and 3 are valid for the **and1** gate, **not** gate, and the **and2** gate. But, only test cases 2 and 3 are valid for the **or** gate.

Determination of MC/DC

The next step is to determine whether the valid test cases are sufficient to provide MC/DC. Here sufficiency is decided by examining the individual gates. Starting with the **and1** gate, the valid test cases are compared with the minimum test criteria defined for that gate. The test combinations *TT*, *TF,* and *FT* are needed here. In the example, test case 1 provides the *TT* test, test case 2 provides the *TF* test, and test case 3 provides the *FT* test case. Hence, test cases 1, 2, and 3 are sufficient to provide MC/DC for the **and1** gate.

Next, test combinations *FF*, *TF*, and *FT* are needed for the **or** gate. In this example, test case 2 provides the *TF* test, test case 3 provides the *FT* test. However, there is no test case for *FF*.

MC/DC is relevant to the **not** gate—but only with respect to showing that the input takes on all possible values. Showing independent effect does not apply because the **not** operator only works with a single operand. In this example, test case 1 provides a *true* input and both test cases 2 and 3 provide a *false* input to the **not** gate. This is sufficient for MC/DC for the **not** operator.

Finally, the inputs to the **and2** gate are checked against the minimum test requirements. In this case, test case 2 and 3 both provide a *TT* input, and test case 1 provides a *FT* input. However, there is

no test case that provides a *TF* input. Hence, the test cases in Table 4 are not sufficient to provide MC/DC of this implementation of the **xor** operation. These results are summarized in Table 6.

Table 6. Comparison of Minimum Tests with Valid Tests

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

The test suite in Table 4 should be supplemented with an additional test case, *FF*, to provide full MC/DC of the source code. A *FF* test case will provide a *FF* input to the **or** gate, and will also give a *TF* input to the **and2** gate. Adding the *FF* test case implies that exhaustive testing of the input combinations for this example is required to provide MC/DC—hence the previous recommendation that exhaustively testing an **xor** operation is justifiable.

Output Confirmation

The final step of the evaluation 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 with the MC/DC objective is done in conjunction with the determination of the proper requirements-based test results. In the example, the outputs determined by following the test inputs through the logic gates match the expected results.

The five steps of the evaluation method may be used as the MC/DC analysis method for any source code. However, if performed manually for an entire project, the method is labor intensive.

Instead, this approach is intended to give certification authorities or verification analysts a simple method to manually confirm that test cases or tools have given the proper results. The steps can also be used to help confirm that an automated tool properly assesses MC/DC. *A Practical Tutorial on Modified Condition/Decision Coverage* [8] provides further details and examples of the 5-step process. The tutorial also discusses 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.

Error Sensitivity

As noted by Dupuy and Leveson [5], the requirement to meet the MC/DC objective for level A software is considered controversial by many due, in part, to perceived ineffectiveness in detecting errors. This raises the issue of what types of errors will be detected by a test set that achieves MC/DC.

Structural coverage analysis using the evaluation approach presented above 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. According to section 6.4.4.3 of DO-178B [1], insufficient coverage 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 of DO-178B provides guidance for each of these.

The evaluation approach may also identify errors in the source code. Here we consider three classes of coding errors:

- Operator errors: where an incorrect operator is used; e.g., an *or* is used instead of an *and*
- Operand errors: where an incorrect operand is used; e.g., a **C** is used instead of a **B** (where **C** and **B** are both Boolean typed variables)
- Grouping errors: where operands and operators are incorrectly grouped.

Further, only single instances of each type of error are considered here.

Because MC/DC is a measure of the adequacy of requirements-based testing, the analysis of error sensitivity proceeds by examining whether test cases, designed to provide MC/DC of a logical requirement, will indicate if there is an error in the source code. Comparing truth tables for a correct and an incorrect expression can show whether a given test set is likely to catch an error in a logical expression. For this analysis, it does not matter whether the conditions in the truth tables are simple conditions or represent more complex subexpressions.

The following subsections consider each of the three error classes.

Operator Errors

A test set that provides MC/DC of a logical requirement is likely to catch single operator errors in the source code (that is, incorrectly coding one operator in a logical expression), assuming that the minimum test set to provide MC/DC for the *xor* case includes the *TT* test case. Having a requirement to evaluate **A** *and* **B**, while having corresponding source code that incorrectly evaluates **A** *or* **B**, is an example of a simple operator error.

Table 7 shows the minimum test requirements for three simple logical expressions.

Table 7. Truth Tables for Simple Expressions

A	B	A *and* B	A *or* B	A *xor* B
T	*T*	*T*		*F***
T	*F*	*F*	*T*	*T*
F	*T*	*F*	*T*	*T*
F	*F*		*F*	*F*

** For this analysis, any three test cases are considered valid for MC/DC as long as *TT* is included.

In the case where the correct code should contain **A** *and* **B**, the minimum test set expected to provide MC/DC for the *and* operator is (*TT*, *TF*, *FT*). That is, the requirements-based test cases are expected to contain tests that will provide the inputs (*TT*, *TF*, and *FT*) to the statement containing **A** *and* **B**. In this case, the test cases should detect when either an *or* or an *xor* is incorrectly coded for an *and* because the actual results and the expected results should not match for the *TF* and *FT* tests, as

shown in Table 7. The *TT* test case will also detect an implementation of *xor* for this example.

In the case where the correct code should contain **A** *or* **B**, the requirements-based tests are expected to contain (*FF*, *TF*, *FT*) to provide MC/DC. Such test cases should detect when an *and* is incorrectly coded for the *or* because the actual results and the expected results should not match for the *TF* and *FT* tests. An incorrect implementation using an *xor* is also detected when using the rule that the minimum test set for an *xor* must contain the *TT* test case. In absence of the *TT* test case requirement for the *xor*, the expected results and the actual results will match if the code incorrectly contains **A** *xor* **B**. If incorrectly coding *or* for *xor*, or vice versa, is a problem in development, requiring exhaustive testing of *xor* operations may be a reasonable step for detecting that error.

Operand Errors

The next error class is operand errors. This error occurs when one condition in an expression is incorrect; for example, when a requirement for evaluating **A** *and* **B** is incorrectly coded **A** *and* **C**, where **B** and **C** are two distinct conditions.

Table 8 shows all possible test sets with inputs **A**, **B**, and **C** that provide MC/DC for the expression **A** *and* **B**.

Table 8. Test Sets for A *and* B

Test Set Number	Test Cases (A, B, C)
1	*TTT*, *TFT*, *FTT*
2	*TTT*, *TFT*, *FTF*
3	*TTT*, *TFF*, *FTT*
4	*TTF*, *TFT*, *FTT*
5	*TTT*, *TFF*, *FTF*
6	*TTF*, *TFT*, *FTF*
7	*TTF*, *TFF*, *FTT*
8	*TTF*, *TFF*, *FTF*

Test cases in test sets 1, 2, 4, 6, 7, and 8 in Table 8 will fail to provide the expected results if the code incorrectly has **A** *and* **C**, thus catching the error. This leaves two test sets to examine, 3 and 5.

In test set 3, the values for **B** and **C** are the same in each test case; so, the error will not be detected by looking at the output. For test set 5, the coding error will be caught in the coverage analysis because test set 5 fails to meet the minimum test criteria for **A** *and* **C**.

Similarly, if **A** *or* **B** is incorrectly coded **A** *or* **C**, only one of eight possible test sets will fail to detect the error—the test set where **B** and **C** have exactly the same values. Hence, a test set that provides MC/DC at the requirements level should provide assurance that the source code does not have single operand errors—except for the case where the two conditions in question are tested with the same values in each test case.

Grouping Errors

The sensitivity analysis for single operator and operand errors clearly shows when those errors will be caught with a test set that provides MC/DC of the requirements. For grouping errors (that is, errors where parentheses are misplaced so as to change the functionality of an expression), there is no clear pattern of cases when the error will or will not be caught. Here we look at expressions with three operands and two distinct operators.

Consider the expression **A** *and* (**B** *or* **C**). A grouping error for this expression would be (**A** *and* **B**) *or* **C**—which results in a different Boolean function. The test set (*TTF*, *TFF*, *TFT*, *FTF*) provides MC/DC for the correct expression **A** *and* (**B** *or* **C**). This test set also provides MC/DC of the improperly coded expression, and produces the expected results when executed. That is, this particular test set will not identify the coding error.

Similarly, consider the expression **A** *or* (**B** *and* **C**), which can be incorrectly coded as (**A** *or* **B**) *and* **C**. The test set (*FTT*, *FFT*, *FTF*, *TFT*) gives the same test outputs for each expression and provides MC/DC for each expression. Again, the coding error will not be detected.

This analysis of error sensitivity is not intended to be comprehensive. However, this analysis does provide some insight into the types of errors one might expect to identify with a test set that provides MC/DC of a logical requirement. In general, such test sets appear to be more sensitive to operand and operator errors than grouping errors.

Additional discussion by Chilenski of theoretical aspects of MC/DC, including error sensitivity, can be found in [9, 10]. Experiments related to error sensitivity, such as comparing the frequency of logic errors in short versus long Boolean expressions, are proposed in *Comments on Modified Condition/Decision Coverage for Software Testing* [11].

Summary

This paper provided a brief introduction to a method for assessing whether requirements-based testing of a level A software product achieves the DO-178B objective for MC/DC. This approach enables a certification authority or verification analyst to effectively evaluate MC/DC claims on a level A software project without the aid of an automated tool, and can assist in selection, qualification, and approval of structural coverage analysis tools. Further, this paper provided a quick look at the ability of MC/DC-compliant test sets to detect three classes of simple coding errors.

References

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

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

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

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

[5] Dupuy, Arnaud, Nancy Leveson, October 2000, *An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software*, Digital Avionics Systems Conference.

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

[7] Abramovici, Miron, Melvin A. Breuer, Arthur D. Friedman, 1990, *Digital Systems Testing and Testable Design,* Computer Science Press.

[8] Hayhurst, Kelly J., Dan S. Veerhusen, John J. Chilenski, Leanna K. Rierson, May 2001, *A Practical Tutorial on Modified Condition/Decision Coverage*, NASA/TM-2001-210876.

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

[10] Chilenski, John Joseph, January 2001, *MCDC Forms (Unique-Cause, Masking) versus Error Sensitivity*, White paper submitted to NASA Langley Research Center under contract NAS1-20341.

[11] White, Allan L., March 2001, *Comments on Modified Condition/Decision Coverage for Software Testing*, 2001 IEEE Aerospace Conference proceedings.