Computing (2015) 97:261–279
DOI 10.1007/s00607-014-0418-5

Reasonability of MC/DC for safety-relevant software implemented in programming languages with short-circuit evaluation

Susanne Kandl · Sandeep Chandrashekar

Received: 28 June 2013 / Accepted: 18 June 2014 / Published online: 3 August 2014
© The Author(s) 2014. This article is published with open access at Springerlink.com

Abstract Modified condition/decision coverage (MC/DC) is a structural code coverage metric, originally defined in the standard DO-178B, intended to be an efficient coverage metric for the evaluation of the testing process of software incorporating decisions with complex Boolean expressions. The upcoming standard ISO 26262 for safety-relevant automotive systems prescribes MC/DC for ASIL D as a highly recommended coverage metric. One assumed benefit of MC/DC is that it requires a much smaller number of test cases in comparison to multiple condition coverage (MCC), while sustaining a quite high error-detection probability. Programming languages like C, commonly used for implementing software for the automotive domain, are using short-circuit evaluation. For short-circuit evaluation the number of test cases for MCC is much smaller than in a non-short-circuit environment because many redundant test cases occur. We evaluated the trade-off between the number of test cases for MCC and MC/DC for a case study from the automotive domain and observed a very low overhead (only 5 %) for the number of test cases necessary for MCC compared to MC/DC. This motivated an analysis of programs containing decisions where the number and structure of the referring Boolean expressions vary. Our results show that the overhead for a test suite for MCC is on the average only about 35 % compared to MC/DC and the *maximum* overhead is approximately 100 % (for decisions with up to 5 conditions). This means that a test set for MCC is in the worst case around twice as big as a test set for MC/DC for a program with short-circuit evaluation with maximum 5 conditions.

S. Kandl (✉)
Institute of Computer Engineering, Vienna University of Technology, Treitlstrasse 3,
1040 Vienna, Austria
e-mail: susanne@vmars.tuwien.ac.at

S. Chandrashekar
Infineon Technologies India Pvt. Ltd., Bangalore, India
e-mail: Sandeep.Chandrashekar@infineon.com

Considering the lower error-detection effectiveness of MC/DC compared to MCC, we conclude with the strong recommendation to use MCC as a coverage metric for testing safety-relevant software (with a limited number of conditions) implemented in programming languages with short-circuit evaluation.

Keywords Testing · Safety-relevant systems · Coverage · MC/DC · Error-detection rate

Mathematics Subject Classification 68M15 · 68N30 · 94C12

1 Introduction

Safety-relevant software, causing crucial damage to people or the environment when malfunctioning, has to be tested exhaustively to guarantee a high reliability. Different methods for evaluating the maturity of the testing process are applied: review processes, assessment processes, coverage metrics.

Modified condition/decision coverage (MC/DC) is a structural code coverage metric, originally defined in the standard DO-178B Software Considerations in Airborne Systems and Equipment Certification [1]. Besides using MC/DC as a qualitative measure to ensure the compliance of the implementation with the specification, MC/DC is intended to be an efficient coverage metric for the evaluation of the testing process of software incorporating decisions with complex Boolean expressions containing multiple conditions, like `if`$(A\&\&(B||C))$ `statement_1 else statement_2`.

The original definition of MC/DC addressed mainly programming languages used at that time for safety-critical applications in the avionics domain, like Ada. The Boolean operators AND and OR of Ada are using *eager evaluation* (aka: *greedy evaluation*), i.e. the whole Boolean expression is evaluated to determine the resulting outcome. Complete testing of such an expression is given by the metric MCC (multiple condition coverage). This coverage requires all possible Boolean assignments to the input variables of the decision. For a decision containing N Boolean conditions ($N = 3$ for the example above) we would need to generate 2^N inputs to test all possible combinations. This means the testing effort grows *exponentially* with an increasing complexity of the decision. MC/DC requires a much smaller number of test cases in comparison to MCC. For a Boolean expression with N conditions MC/DC requires only $N + 1$ test cases, so the number of test cases grows only *linearly* with the number of conditions. Although the number of test cases is much smaller than for complete testing, MC/DC sustains a quite high error detection probability [2].

MC/DC is also defined as a highly recommended metric for ASIL D systems in the upcoming standard ISO 26262 Road Vehicles—Functional Safety [3, Part 6, Table 14, p. 22] for safety-relevant automotive systems, as given by the Table 1: The table shows the recommended coverage criteria for the different ASIL's. ASIL means *automotive safety integrity level*, ASIL A is the ASIL with lowest safety-relevance and ASIL D is the ASIL with highest safety-relevance. A method is either recommended (+) or highly recommended (++).

Table 1 Structural coverage metrics at the software unit level [3]

Method	ASIL			
	A	B	C	D
1a Statement coverage	++	++	+	+
1b Branch coverage	+	++	++	++
1c MC/DC (modified condition/decision coverage)	+	+	+	++

Commonly used programming languages for software applications in the automotive domain are Assembler and C. In contrast to programming languages like Ada, C uses *short-circuit evaluation*. That means if the result of a Boolean expression is already determined by the first part of an expression, the remaining part of this expression is not executed anymore. For short-circuit evaluation the number of test cases for MCC is much smaller than 2^N because many redundant test cases occur.

Based on a study of the code structure of typical code for programs of the MCAL (microcontroller abstraction layer, part of the AUTOSAR concept) to gain an overview of the attributes for an eligible case study, we selected a representative case study for our experiment and evaluated the testing effort for both, MCC and MC/DC. The initial aim was to show the benefit of using MC/DC instead of MCC for the evaluation of the testing process, expecting a significant decrease of the testing effort (the number of required test cases, respectively). Indeed the number of test cases for MCC was only about 5 % higher than the number of test cases for MC/DC. This result motivated us to systematically examine programs with decisions depending on different complex Boolean expressions up to the number of 5 conditions. Even for 5 conditions we observed only an *average overhead* of around 35 % compared to the number of test cases for MC/DC, which is still feasible for testing. As the MC/DC-test set is only a subset of the MCC-test set, it may happen that errors are not detected by the MC/DC-test set although covered by the MCC-test set. For artificially introduced mutations into the original program we observed 4 out of 100 errors that are detected by the MCC-test set, but not by the MC/DC-test set. Regarding this enhanced confidence in the testing process using the MCC-criterion instead of the MC/DC-criterion, and considering the acceptable overhead for the increased number of test cases to achieve full MCC, we question the use of the MC/DC-criterion for safety-relevant software implemented in a programming language with short-circuit evaluation.

The paper is organized as follows: First we explain the different coverage criteria considered for our test environment (addressing also the expected and the real error-detection probability of MC/DC). Then we give a short overview of the test environment for executing the test runs and a rough description of the use case for our experiment. In an example we compare an MCC-test set with an MC/DC-test set for a programming language with short-circuit evaluation to motivate *how* the overhead for MCC is determined. In Sect. 4 we first describe the set-up in detail to determine the estimated overhead for programs containing multiple decisions of different structures, then we show our experimental results for the estimated overhead. Subsequently, we compare the error-detection effectiveness of MCC to MC/DC. Then we discuss what these results mean for practice. After presenting some related work, we conclude with a summary.

2 Coverage criteria

2.1 Basic terminology

A **test set** is a set of test cases. A **test case** is a vector containing the values for the input data (for all the input variables). A test set for a specific coverage criterion (e.g., an MCC-test set) consists of a *minimal* number of test cases to achieve full coverage (regarding the coverage criterion) on the program under test.

2.2 Decision coverage (DC)

Decision coverage requires test cases to cover both branches of a decision (the if- and the else-branch). For each decision the DC-criterion requires two test cases. For a decision depending on a complex Boolean expression (like `if (A && B)` also only two test cases are necessary for DC, e.g. the test cases (A = true, B = true) and (A = false, B = false).

2.3 Unique-cause MC/DC (MC/DC)

Unique-Cause MC/DC is introduced in DO-178B [1], discussed in detail in [4], and expanded with variations of the metric (e.g., Masking MC/DC) in [5], respectively. For simplicity we use the term MC/DC for Unique-Cause MC/DC, as we only consider this version of MC/DC. The metric is a structural coverage metric defined on the source code and is designed to test programs with decisions that depend on one or more conditions, like `if` $(A\&\&(B||C))$ `statement_1 else statement_2`.

For MC/DC we need a set of test cases to show that changing the value for each particular condition changes the outcome of the total decision independently from the values of the other conditions. (This works as long there is no coupling between different instances of conditions.)

A test set conforming to MC/DC consists of test cases that guarantee that [1,4]:

- every point of entry and exit in the model has been invoked at least once,
- every basic condition in a decision in the model has been taken on all possible outcomes at least once, and
- each basic condition has been shown to *independently* affect the decision's outcome.

The independence of each condition has to be shown. If a variable occurs several times within a formula each instance of this variable has to be treated separately. Independence is defined via *Independence Pairs*. An independence pair for a variable A is defined by a pair of input values for the Boolean expression in a decision that shows: By changing the value of A (A = true $\rightarrow$ A = false, or vice versa) while fixing the values for the other input variables (B = true $\rightarrow$ B = true, or B = false $\rightarrow$ B = false) also the outcome changes (true $\rightarrow$ false, or vice versa).

Consider the example $A\&\&(B||C)$: The truth table is given in Table 2 (third column). In the following $\bar{1}$ represents the test case (F, F, F), $\bar{2}$ represents the test case

Table 2 Example MC/DC-Test Set

Test case	A	B	C	$A\&\&(B\|\|C)$
$\bar{1}$	F	F	F	F
$\bar{2}$	F	F	T	F
$\bar{3}$	F	T	F	F
$\bar{4}$	F	T	T	F
$\bar{5}$	T	F	F	F
$\bar{6}$	T	F	T	T
$\bar{7}$	T	T	F	T
$\bar{8}$	T	T	T	T

(F, F, T), and so on. F is the abbreviation for *false*, and T is the abbreviation for *true*. The independence pairs for the variable A are ($\bar{2}$, $\bar{6}$) (A changes from F to T, the other variables remain the same, the outcome changes from F to T), ($\bar{3}$, $\bar{7}$) (A changes from F to T, the outcome changes from F to T), and ($\bar{4}$, $\bar{8}$) (A changes from F to T, the outcome changes from F to T). The independence pair for the variable B is ($\bar{5}$, $\bar{7}$) and the independence pair for the variable C is ($\bar{5}$, $\bar{6}$). The test set for MC/DC is a subset of the test cases of the different independence pairs. This subset has to be chosen in such a way that for each variable (at least) one independence pair is covered (in general, there are multiple valid subsets). For the example we have the test set for MC/DC consisting of {$\bar{5}$, $\bar{6}$, $\bar{7}$} plus one test case of {$\bar{2}$, $\bar{3}$}. The number of test cases for N conditions is $N + 1$. This can be shown by an analysis of the independence graph (a graph-based representation of the independence pairs). For details please refer to [5].

2.3.1 Error-detection probability of MC/DC

In [2] different code coverage metrics are compared and a subsumption hierarchy for the most relevant code coverage metrics is given. It is stated that *"the modified condition/decision coverage criterion is more sensitive to errors in the encoding or compilation of a single operand than decision or condition/decision coverage"*, but MCC is still the strongest coverage metric. The probability of detecting an error is given as a function of tests executed. For a given set of M distinct tests, the probability $P_{(N,M)}$ of detecting an error in an incorrect implementation of a Boolean expression with N conditions is given by [2]

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

This correlation is shown in Figure 1 for $N = 4$. The lower part of the fraction 2^{2^N} represents the number of Boolean functions (for N Boolean operands there are 2^N possible combinations of operand values, thus 2^{2^N} Boolean functions). The term $2^{(2^N-M)} - 1$ in the upper part of the fraction refers to the number of other functions that are indistinguishable, i.e. produce the same outcome for the M distinct tests. The term $1 - fraction$ shows the probability of detecting an error in an incorrect implementation of a Boolean expression.

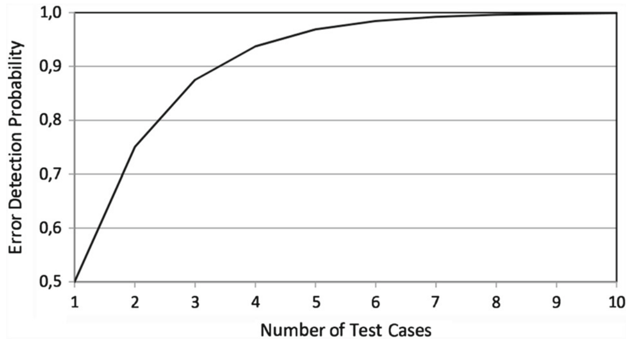

Fig. 1 Error-detection probability of MC/DC [2]

For $N = 4$ and $M = 5$ (the minimal number of test cases) the resulting error detection probability is 0.96. For $N = 5$ and $M = 6$ the error detection probability is 0.98.

As M increases there is a rapid increase in the error detection probability. As N grows, $P_{(N,M)}$ rapidly converges to $1 - 1/2^M$ and the sensitivity changes only marginally with N. That means for N increasing, the likelihood of detecting an error in an expression of N conditions with $N+1$ test cases increases, too. This non-intuitive result occurs because the dominant factor (the number of tests) increases with N while the sensitivity to errors remains relatively stable [2]. Indeed the *real* error-detection probability of MC/DC is less than expected, as we showed in previous works [6]: For a case study 22 % of the mutated variable names and 8 % of the mutated operators are not detected by the MC/DC-test set. In contrast to that, MCC covers all detectable errors. The error-detection rate for the case study will be analyzed in Sect. 4.

2.4 Multiple condition coverage (MCC)

Multiple condition coverage requires that every point of entry and exit in the program is invoked at least once, and all possible combinations of the outcomes of the conditions within each decision are taken at least once. This metric requires for N conditions 2^N test cases for programming languages with greedy evaluation (i.e., the Boolean expression is evaluated for all possible value assignments). As there is no formula to calculate the number of test cases for MCC for programs with short-circuit evaluation, we will analyze this number in an empirical approach.

3 Test environment

3.1 Validation platform

The recent test environment within Infineon is realized in the UVP (universal validation platform). The UVP provides a platform where it is possible to automate the test runs for hardware-dependent software. This type of software can not be tested in an

emulation environment as the functionality is only given in the specific hardware environment (e.g., the compilation process is hardware-specific). The test set-up requires a detailed configuration of the test environment. The UVP was designed to automate as much as possible of this configuration.

The main idea is to generate for each test the application running on the target and then test it with the appropriate set of stimuli and testing algorithms. This approach is feasible by automating the code generation path and including in the test application the considered code for testing under the control of a master test plan. Each version of generated code has also a corresponding test set-up, test stimuli sets, and testing algorithms which are automatically loaded into the test bench. The generation of the hardware-dependent software in the current configuration is done automatically with a small parametric application on the top. Also the compilation, download, launching, and execution of the test application are automated. Result logging and determination of different coverage criteria are supported [7,8].

3.2 Use case

We defined some attributes for the case study on which we want to apply our analysis. The case study should provide a complexity in the control flow justifying the use of MC/DC. We studied the code structure of typical code for programs part of the MCAL (microcontroller abstraction layer) to gain some information on the nature of the programs to choose an appropriate case study. Based on an analysis of the code structure of the programs we decided for a case study with following properties: The case study is a program for pulse width modulation, part of a set of hardware drivers for the Tricore processor. It has about 6,100 lines of code and it includes 24 decisions, with a different number of conditions. Moreover it contains further decisions in the preprocessor directives. This program was chosen because it showed the highest complexity regarding the control flow graph. It is remarkable that the considered case studies have a limited number of conditions in the decisions (usually $\leq$5).

3.3 Test runs

The case study cannot be executed as a standalone application, but only as part of a driver package consisting of multiple files. So in the overall a system is executed with approximately 15,000 lines of code, including the observed case study with 6,100 LOC. The first test run was executed by a test set provided by Infineon. This test set contains test cases generated manually based on the requirements from the system specification. The initial test run for the case study yields the following coverage results (Table 3):

Table 3 Results of the initial test run for the case study

Decision coverage	MCC	MC/DC
63 %	63 %	62 %
795/1253	937/1499	887/1424

Table 4 Example_A

Expression	MCC	MC/DC
A&&B&&C	F- -	F- -
	TF-	TF-
	TTF	TTF
	TTT	TTT

The %-value gives the achieved percentage of achieved coverage, calculated by the quotient: number of executed test cases/number of required test cases to achieve full coverage (given in the second row).

Observation A: The value for the achieved MCC-coverage is similar to the value of the achieved decision coverage (DC). For MCC in comparison to DC only 246 (=1499−1253) additional test cases are needed. This means that many of the decisions occurring in the tested system are simple decisions (depending on a singular condition like `if(A)`), i.e., for these decisions the number of test cases needed for DC equals the number of test cases needed for MCC [and equals the number of test cases needed for MC/DC]. Only a small part of decisions contain a complex Boolean expression with more than one condition (additional test cases to achieve maximum MCC are required).

Observation B: The achieved coverage value for MCC is nearly the same as for MC/DC (63 % vs. 62 %). To achieve full MCC only 75 (=1499−1424) additional test cases are needed in comparison to the MC/DC test set. These additional 75 test cases denote an overhead of approximately 5 % compared to the number of necessary test cases for MC/DC.

Although we learned from Observation A that only a few decisions are of a complex structure, the value of 5 % is surprisingly low. Usually MCC needs a significantly higher number of test cases than MC/DC, and thus we would have expected a higher overhead of the MCC test set compared to the MC/DC test set. This insight motivated us to compare the test sets for MCC and MC/DC for programs with short-circuit evaluation in more detail (see Sect. 4). Based on the results from the initial test run, we identified the missing test cases to achieve full MCC and MC/DC. After adding the missing test cases that can be executed we achieved full coverage.

3.4 MCC-test set versus MC/DC-test set for short-circuit evaluation

The surprisingly low overhead for our case study motivated an analysis in which extend short-circuit evaluation influences the number of test cases required for MCC. The required number of test cases depends on the structure of the Boolean expression, demonstrated by following examples, see Tables 4 and 5: The symbol F is false, the symbol T is true, and the "-" can be any kind of value. The test case marked in bold represents the difference between the two test sets.

Both examples show an expression with 3 conditions, so full MCC without short-circuit evaluation would need $2^3 = 8$ test cases. In the column MCC the required test cases for MCC with short-circuit evaluation are given: For the expression in

Table 5 Example_B

Expression	MCC	MC/DC
$A\&\&B\|\|C$	F-F	F-F
	F-T	F-T
	TFF	TFF
	TFT	
	TT-	TT-

Example_A 4 test cases are required for MCC, so the number of test cases for MCC equals the number of test cases for MC/DC. For the expression in Example_B 5 test cases for MCC are required, whereas 4 test cases are required for MC/DC. For this Boolean expression the overhead is 25 %.

4 Analysis and results

4.1 Overhead for MCC compared to MC/DC

We executed an analysis of the number of test cases needed for MCC, and MC/DC, respectively, for all possible Boolean expressions up to 5 conditions: For a Boolean expression with 2 conditions, 2 expressions are possible: `A && B`, and `A || B`. For a Boolean expression with 3 conditions, 4 expressions are possible: `A && B && C`, `A && B || C`, `A || B && C`, and `A || B || C`. For a Boolean expression with 4 conditions 8 different expressions are possible, and for a Boolean expression with 5 conditions 16 different expressions are possible.

Besides that we can integrate parentheses, this increases the number of possible Boolean expressions (OP...operator): For a Boolean expression with 4 conditions 6 different ways to apply parentheses are possible (for each of the 8 expressions): `(A OP B) OP C OP D; A OP (B OP C) OP D; A OP B OP (C OP D); (A OP B) OP (C OP D); (A OP B OP C) OP D; A OP (B OP C OP D)`. For a Boolean expression with 5 conditions 14 different ways to apply parentheses are possible (for each of the 16 expressions).

For the evaluation we only considered variants of the Boolean expressions for which the parentheses have an impact on the result of the expression: `A && B && C` is the same as `(A && B) && C`, whereas `A || B && C` differs from `(A || B) && C`. The idea of these permutations is to cover as many different Boolean expressions as possible.

Then we determine the number of required test cases for each Boolean expression by enumeration.

Analyzing the number of test cases required for MCC with short-circuit evaluation for all possible Boolean expressions with up to 5 conditions (without parentheses *wp* and including parentheses *ip*) shows following results, see Table 6.

The columns are described in the following.

Cond	Number of conditions.
MCDC	Number of test cases for MC/DC.

Table 6 Summary of the results

Cond	MCDC	MCC_maxwp	MCC_maxip	OH_max (%)	OH_avwp (%)	OH_avip (%)
2	3	3	3	0.00	0.00	0.00
3	4	5	5	25.00	6.25	9.38
4	5	7	8	60.00	17.50	20.42
5	6	11	13	116.00	33.33	35.82

MCC_maxwp Maximum number of test cases for MCC for all Boolean expressions without parentheses.

MCC_maxip Maximum number of test cases for MCC for all Boolean expressions including parentheses.

OH_max Considering that a system under test has decisions with Boolean expressions that require the maximum number of observed test cases for MCC, this value would be the maximum overhead for the number of test cases for MCC (compared to the number of required test cases for MC/DC).

OH_avwp Considering that a system under test contains all kinds of possible Boolean expressions with N conditions without parentheses (uniformly distributed), this value describes the average overhead for the required test cases for MCC.

OH_avip Considering that a system under test contains all kinds of possible Boolean expressions with N conditions also including parentheses (uniformly distributed), this value describes the average overhead for the required test cases for MCC.

A value of 60 % overhead means that a test set for MCC requires 60 % more test cases than the test set for MC/DC.Example: The number of test cases for MC/DC is 100, then the number of test cases for MCC is 160.

The maximum overhead of 116 % means that a test set for MCC is approximately twice as big as the test set for MC/DC (e.g., 100 test cases for MC/DC means 216 test cases for MCC).

Observations from this survey:

- For 2 conditions the number of test cases for MCC is always equal to the number of test cases for MC/DC.
- For 3, or 4 conditions, respectively, the maximum overhead for MCC is 25 %, or 60 %, respectively. So in the worst case (many decisions with Boolean expressions containing 4 conditions), MCC testing means an overhead of 60 % for the test cases (in comparison to MC/DC). This overhead is acceptable.
- For 3, or 4 conditions, respectively, the average overhead (including also expressions with parentheses) for MCC is approx. 9 %, or 20 %, respectively, so almost negligible.
- Even for 5 conditions the average overhead is around 35 % (compared to the number of test cases for MC/DC), which is still feasible for testing. Based on the

experiences from our case studies, the number of conditions within software for the automotive domain is often limited by 5, so the resulting overhead is acceptable.

4.2 Error-detection effectiveness MCC versus MC/DC

The main attribute of a test set we are interested in is the error-detection effectiveness, i.e. how many errors are detected, or how many errors are not detected, respectively. For a comparison between the error-detection effectiveness of the MCC-test set compared to the MC/DC-test set, we are interested in errors in the program that are detected by the MCC-test set, but reveal undetected by the MC/DC-test set.

Fault versus Error: Although the terms fault, error, and failure are well-defined they are sometimes used in a confusing way in literature. Referring to [9] we have to distinguish three different terms for erroneous system behavior.

A *fault* is the cause of an error, and thus the indirect cause of a failure.
An *error* is an unintended, resp. incorrect, internal state of a computer.
A *transient* error exists only for a short interval of time and disappears without an explicit repair action. If the error persists permanently until an explicit repair action removes it, we call it a *permanent* error.
A *failure* is an event that denotes a deviation between the actual service and the specified or intended service, occurring at a particular point in real time.
Most computer-system failures can be traced to an incorrect internal state of the computer, e.g., a wrong data element in the memory or a register.

We consider a programming mistake as a fault, the consequence is an error in the software upon activation, the error becomes effective when this error produces erroneous data which affect the delivered service, then a failure occurs. In the literature about testing and the effectiveness of a test set, both terms occur:
Fault-detection effectiveness and *error*-detection effectiveness. Indeed we are interested in determining *both*: the errors and the underlying faults, but within our testing process we can only observe the errors. Therefore in this work we stick to the term error-detection effectiveness.

Minimization of the test set: The initial test set includes redundant test cases that are not necessary to achieve the intended coverage. In a first step we have to reduce the test set to a *minimal* test set, i.e. removing the redundant test cases. This minimal test set is used for the test runs to determine the error-detection effectiveness.

Program mutations: To determine the real error-detection effectiveness we introduce faults into the original programs. These faults can relate to a mutation for an operator, the name of a variable, or concrete values.

Consider the example given in Listing 1. Let us assume the programmer omits the brackets in the Boolean expression in line 4, resulting in the program given in Listing 2. The required test sets for MCC and MC/DC are given in Table 7. The test case marked in bold depicts the test case capable to detect the introduced fault.

Table 7 Test sets for MCC and MC/DC for the listing 1

Test case	MCC	MC/DC
$\bar{1}$	F-F-	F-F-
$\bar{2}$	F-TF	F-TF
$\bar{3}$	F-TT	F-TT
$\bar{4}$	TFF-	TFF-
$\bar{5}$	TFTF	
$\bar{6}$	TFTT	
$\bar{7}$	**TT-F**	
$\bar{8}$	TT-T	TT-T

```
int erg;

int test(_Bool a, _Bool b, _Bool c, _Bool d) {
  if ((a && b || c) && d) {
    erg = 42;
  }
  else {
    erg = 24;
  }
  return erg;
}
```

Listing 1 Original Program

```
int erg;

int test(_Bool a, _Bool b, _Bool c, _Bool d) {
  if (a && b || c && d) {
    erg = 42;
  }
  else {
    erg = 24;
  }
  return erg;
}
```

Listing 2 Erroneous Program

Running these test sets on the original program results in full MCC, and full MC/DC, respectively. For executing the mutated program with the MC/DC-test set all the test cases pass the testing, no erroneous behavior is observed. Executing the mutated program with the MCC-test set, shows an error for test case $\bar{7}$. So for this example the MCC-test set is capable to detect the introduced mutation, whereas the MC/DC-test does not detect the error.

Latent faults/errors: We always have to be aware of that there may occur mutations in the program that have no effect on the observable values. This kind of faults/errors are called *latent*, see for instance the example given in Listing 3: The mutation occurs

in line 3 where `switch2` is written instead of `switch1`. This mutation of the name of a variable has no impact on the control flow of the program, thus no effect on the resulting value for the variable `erg`. This kind of mutation can not be detected by *any* test case, neither by the MC/DC-test set, nor by the MCC-test set.

```
int test(_Bool a, _Bool b, _Bool c, _Bool d) {
  if ((a || b) && c) {
    switch1 = 1; —> MUTATION: switch2 =1;
  }
  else {
    switch1 = 0;
  }
  if (c && d) {
    switch2 = 1;
  }
  else {
    switch2 = 0;
  }
  if(switch1 == switch2) {
    erg=42;
  }
  else {
    erg=24;
  }
  return erg;
}
```

Listing 3 Latent Error

Error-detection effectiveness for the case study: For our case study we considered 100 different mutations, with the result that 4 out of them were errors that were not detected by the MC/DC-test set, but were detected by the MCC-test set. This may seem a low rate of undetected errors, but keeping in mind that we deal with safety-relevant software, for which we aim at high reliability, 4 % undetected errors is a very high rate.

4.3 Discussion of the results

Based on the observations from the case study, we question the reasonability of MC/DC instead of MCC for software from the automotive domain (with a manageable complexity, i.e. a limited number of conditions) realized in a programming language with short-circuit evaluation. We learned that the overhead for the MCC-test set is almost negligible (regarding the number of test cases) in comparison to an MC/DC-test set. As we showed in the analysis the number of test cases required for MCC (for a system implemented in a language with short-circuit evaluation) causes only a small overhead (5 % for our case study) for testing in comparison to MC/DC. This can be explained in following way: Many of the decisions with a complex Boolean expression contain only 2 conditions. For 2 conditions the number of test cases required for MCC is equal to the number of test cases required for MC/DC (for both 3 test cases), so the MCC-test set is the same as the MC/DC-test set for these decisions. Some decisions contain more

conditions, even for these decisions the additional test cases for MCC are only a few (e.g., 5 vs. 4 test cases for `A && B ||C`, see Example_B in Sect. 3.4). Furthermore, we showed in the comparison of the error-detection effectiveness of MCC vs. MC/DC that some errors are only detected by the MCC-test set. In contrast to an MC/DC-test set, the MCC-test set covers the whole possible input-data space, so it guarantees that all detectable errors are identified. With the restricted MC/DC-test set, not all errors may be identified.

The usage of MC/DC makes sense as a *qualitative* means to assess the maturity of the software development process. The metric can be used to prove whether the requirements defined in the system specification map the implemented code (a poor value for MC/DC for a test set generated requirement-based indicates a lack in the specification, or unspecified functionality in the implemented code). This kind of deviations indicate a gap between the specification and the implementation. The use of MC/DC used as a *quantitative* measure is reasonable when it is used as an alternative coverage metric to stronger coverage metrics, like MCC, because it is not feasible to realize full testing (*stronger* in this context means that the test set of MCC is a superset of the test set of MC/DC, i.e. the test cases of MCC cover a larger part of the input data space, thus the ability to detect errors in the program is higher). But as far as the overhead for MCC is so low, MCC is better suitable as a quantitative measure for the evaluation of the testing process for safety-relevant programs implemented in a programming language with short-circuit evaluation.

Regarding the guidelines of the standard ISO 26262 [3] and addressing the aim of high reliability required for safety-relevant programs it would be desirable to combine the benefits of both metrics: As deriving the MC/DC-test set is a non-trivial issue this process assumes a detailed analysis of the structure (the control flow) of the program. So building a test set to achieve maximum MC/DC for a system under test forces the test engineers to study both, the specification and the implementation, in a very precise way. This activity by itself enforces the quality of the testing process. On the other side, by achieving full MCC it is guaranteed that all *detectable* errors are identified (not detectable errors are *latent* faults or errors, deviations in the program that have no impact on the resulting output; these errors are not detectable, no matter which test cases are applied).

Besides that, the test engineer should always be aware of that a *structural* code coverage metric is only evaluated based on the implementation. Achieving a specific coverage goal by incremental test case-generation until x % coverage is achieved may increase the part of the tested code. But in the sense of a structured verification process, i.e., checking whether the system is conform with the specification, or not, this is by far not sufficient, see also [10]. In this white paper Büchner defines some commonly used code coverage measures and discusses their strength and weakness. Small examples are used to illustrate some measures to indicate common traps and pitfalls. Two main weaknesses of code coverage are identified: (1) Code coverage measurements cannot detect omissions, e.g. missing or incomplete code. (2) Code coverage measurement is insensitive to calculations (Example: Given a complex calculation as part of the control flow, a single input may cover this calculation regarding a structural code coverage metric, thus achieving 100 % coverage with only one test input. But this single test input does *not* verify the correctness of the complex calculation.). An increasing value

for code coverage indicates a progress in the testing process, nevertheless achieving 100 % code coverage is not sufficient to rely on the proper functioning of a system.

Structural code coverage metrics should only be a supplement to approaches like *requirement-based* testing, in which the *requirements* guide the testing process (and not the test data), see [11] and [12].

[13] mentions some *misunderstandings of the MC/DC Objective*:

- Not understanding the intent of structural coverage.
- Trying to meet the MC/DC objective apart from requirement-based testing (that is, using the source code to derive inputs for all test cases).
- Using MC/DC as a testing method (that is, expecting MC/DC to find errors instead of assuring that requirements-based testing is adequate).
- Etc.

A coverage criterion is only a means to define a set of test cases and providing a quantitative measure which parts of the control flow and which subset of the input-data space is covered by this test set. Test sets for the DC-, MC/DC-, and MCC-criterion guarantee that all the branches of the control flow graph are covered by running the test set. But the MCC-test set contains more test cases than the MC/DC-test set, i.e. it covers more values of the input-data space. With MCC *all* the possible inputs for a decision are considered for testing, thus it covers the complete input-data space and assures that all *detectable* faults (i.e., all faults, except the latent faults) are detected by testing. The MC/DC-test set, as a subset of the MCC-test set, contains less test cases, thus it covers a smaller subset of the input-data space as the MCC-test set. This causes a decreased fault-detection sensitivity. As long as the overhead for an MCC-test set is reasonable, it is always better to use MCC instead of MC/DC.

The problem we see with MC/DC in the context of ISO 26262 is, that it is only mentioned as a metric to be fulfilled for the testing of software. The standard does not give any guidelines about *requirement-based testing*, nor does it give any advice how to use MC/DC as a *technique* as part of the software testing process. The danger is that achieving full MC/DC may be used as an argument for a sufficient testing process. But it has following fundamental restrictions:

(a) MC/DC used as a quantitative measure is only reasonable if the test cases are derived *directly* from the requirements (and not by any other means, like static analysis of the source code).
(b) MC/DC cannot be used to argue for reliability of a system regarding the confidence in error-freeness.

To cover as much as possible of the input-data space to maximize the probability to detect faults in the program, we recommend to use MCC instead of MC/DC as a code coverage metric (as long as the overhead is acceptable).

5 Related work

The most important discussion on the *applicability of MC/DC* for testing (safety-critical) software is [2].

The *Practical Tutorial on Modified Condition/Decision Coverage* [13, pp. 42ff.] addresses the topic short-circuit evaluation. Besides the default logical operators of ADA **and** and **or**, the short-circuit control forms **and then** and **or else** are mentioned. Short circuit logic can be caused by a language construct or by a compiler option. It is stated that *For MC/DC, short circuit expressions can be treated in the same manner as conventional* **and** *and* **or** *gates*, as demonstrated by following example.

For the Boolean expression `A && B && C && D` the requirement-based test cases for MC/DC are given by {*TTTT***T**, *TTTF***F**, *TTFT***F**, *TFTT***F**, *FTTT***F**} (the bold values depict the outcome of the decision).

An alternative test set is given by {*TTTT***T**, *TTTF***F**, *TTFF***F**, *TFFF***F**, *FFFF***F**}.

The question is whether this test set also provides MC/DC for the source code, and the answer is: yes. For short-circuit logic once an operand evaluates to *false*, the outcome of the whole expression is set to *false* without evaluation of the remaining operands. In principle the value of the remaining operands does not matter.

Thus the minimum test set for a four-input **and**-gate with short-circuit logic is given by the test set {*TTTT***T**, *TTTF***F**, *TTF* - **F**, *TF* - - **F**, *F* - - - **F**}.

Similar the minimum test set for a four-input **or**-gate with short-circuit logic is given by the test set {*FFFF***F**, *T* - - - **T**, *FT* - - **T**, *FFT* - **T**, *FFFT***T**}.

A list of all statements that are covered by MC/DC for the programming languages C, C++, and Ada83 is given in [14]. General comparisons of code coverage metrics are [15] for structural based metrics and [16] for data-flow based metrics. Jones and Harrold [17] introduce a test-suite reduction with the focus on the MC/DC-criterion. Staas et al. [18] discuss general considerations for the usage of code coverage metrics only as a quantitative measure. The authors conclude with two simple statements: (1) Coverage criteria satisfaction alone is a poor indication of test suite effectiveness. (2) The use of structural coverage as a supplement -not a target- for test generation can have a positive impact. Our results emphasize especially the second point. Also Büchner [19] presents in his collection of eight misapprehensions about coverage that in general *code coverage is* not *a sufficient criterion to assess the quality of code*.

Another aspect for using structural code coverage metrics for the evaluation of the testing effort is the fact that the test set depends on the structure of the code, as shown in [20] for MC/DC. In this work two program versions are compared with the same functionality, one with an in-lined Boolean expression, the other one with the Boolean expression in a non-in-lined version. In-lined in this context means: Consider a program with a decision `if(A)` and the Boolean variable `A = C || D`. The in-lined version of the program looks like this `if (C || D)`, whereas the non-in-lined version looks like this `A = C || D...if(A)`. MC/DC for the non-in-lined version is different from MC/DC for the in-lined version. The examples show that MC/DC is highly sensitive to the structure of the implementation. In general, it is recommended to use MC/DC on the in-lined version of the implementation.

In [21] different code coverage metrics (decision coverage, full predicate coverage, and MC/DC) are compared regarding their effectiveness in finding errors. One result of this work is that, in general, the MC/DC criterion was found to be effective [for detecting faults] independent of the number of conditions in the Boolean decision. Yu and Lau compare several structural coverage criteria, including MC/DC, with the result that MC/DC is cost effective in relation to other criteria [22]. In [23] the testing

effort between decision coverage and MC/DC is compared. A case study for structural testing applied to safety-critical embedded software is [24]. An empirical evaluation of MC/DC for satellite software is [25].

To our knowledge there are no empirical works on the comparison of MCC and MC/DC, especially not focusing on short-circuit evaluation.

6 Summary and conclusion

A satisfying test process should realize the ideal trade-off between effort and confidence in the test result. Structural code coverage metrics are one means to determine the maturity of the testing process, both in a qualitative and in a quantitative way. A structural code coverage metric used in a quantitative way only determines the part of the program executed during testing. MCC covers all possible input values of a decision depending on a complex Boolean expression. Considering *all* input values is, in general, not possible as the number of test cases increases exponentially (assuming a non-short circuit evaluation). MC/DC requires only a subset of the MCC-test set, the number of required test cases grows linearly with the number of conditions in the Boolean expression of the decision. MC/DC used as a qualitative measure can help to identify deviations of the implemented system from the original specification. The usage of MC/DC as a quantitative measure instead of a stronger coverage metric, like MCC, would only be reasonable if full testing is not feasible.

For the use case we showed that the overhead of the number of test cases for MCC is only approximately 5 % compared to the number of test cases for MC/DC. This is caused, on the one hand, by the restricted complexity of the underlying system, on the other hand, by the given property of the programming language C using short-circuit evaluation. In our detailed analysis we considered C-programs with different Boolean expressions with up to 5 conditions, without parentheses and including parentheses. As we showed in our analysis the overhead of test cases for MCC in comparison to MC/DC for short-circuit evaluation is in the worst case (only complex Boolean expressions of a type with maximum overhead for the number of test cases) 116 % for 5 conditions. Considering different possible variants of Boolean expressions the expected average overhead for the number of test cases for MCC in comparison to MC/DC is around 35 % for 5 conditions (and even less for a smaller number of conditions). So the overhead for testing is reasonable. Comparing the error-detection effectiveness of MCC compared to the MC/DC we showed for the case study that 4 out of 100 errors are not detected by the MC/DC-test set although covered by test cases part of the MCC-test set. Regarding the increased confidence in an MCC-test set (covering all detectable errors) and the observed overhead we conclude with the strong recommendation to use the MCC-criterion for safety-relevant programs with short-circuit evaluation (with a limited number of conditions).

Acknowledgments This work has been partially funded by the ARTEMIS Joint Undertaking and the National Funding Agency of Austria for the project VeTeSS under the funding ID ARTEMIS-2011-1-295311 and was supported by Infineon Technologies AG (Munich, Germany).

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

References

1. RTCA Inc. (1992) DO-178B: Software Considerations in Airborne Systems and Equipment Certification, Requirements and Technical Concepts for Aviation, Washington, DC
2. Chilenski J, Miller SP (1994) Applicability of modified condition/decision coverage to software testing. Softw Eng J 9(5):193–200
3. ISO: International Organization for Standardization (2009) ISO 26262: Functional safety—road vehicles, draft
4. RTCA Inc. (2001) DO-248B: Final Report for Clarification of DO-178B: Software Considerations in Airborne Systems and Equipment Certification, Requirements and Technical Concepts for Aviation, Washington, DC
5. Chilenski JJ (2001) An investigation of three forms of the modified condition decision coverage (MCDC) criterion, US Department of Transportation, Federal Aviation Administration, DOT/FAA/AR-01/18
6. Kandl S, Kirner R (2010) Error detection rate of MC/DC for a case study from the automotive domain. In: Min SL et al (eds) Proceedings of the 8th IFIP workshop on software technologies for future embedded and ubiquitous systems (SEUS 2010), LNCS, vol 6399, pp 131–142
7. Infineon: Automotive & Industrial System & Software Engineering SCE5 (April 2004) Universal validation platform—common validation platform for safety-related projects, Internal document of IFX
8. Infineon: AIM MC D SCE5 System & Software Engineering (2004) UVPMM—concept, Internal document of IFX
9. Kopetz H (1997) Real-time systems: design principles for distributed embedded applications, 1st edn. Kluwer Academic Publishers, Massachusetts, USA
10. Büchner F (2012) White paper: Is 100 % code coverage enough? Tessy, Hitex
11. Ammann P, Black PE (1999) A specification-based coverage metric to evaluate test sets. In: HASE, pp 239–248
12. Abdurazik A, Ammann P, Ding W, Offutt J (2000) Evaluation of three specification-based testing criteria. In: Proceedings of the sixth IEEE international conference on engineering of complex computer systems, ICECCS 2000, pp 179–187
13. Hayhurst KJ, Veerhusen DS, Chilenski JJ, Rierson LK (2001) A practical tutorial on modified condition/decision coverage. NASA Langley Technical Report Server, NASA
14. Chilenski J, Richey LA (1997) Definition for a masking form of modified condition decision coverage (MCDC). Technical report, Boeing, Seattle, WA
15. Ntafos SC (1988) A comparison of some structural testing strategies. Softw Eng IEEE Trans 14(6):868–874
16. Clarke LA, Podgurski A, Richardson DJ, Zeil SJ (1985) A comparison of data flow path selection criteria. In: ICSE ’85: Proceedings of the 8th international conference on software engineering. Los Alamitos, CA, USA, IEEE Computer Society Press, pp 244–251
17. Jones JA, Harrold MJ (2001) Test-suite reduction and prioritization for modified condition/decision coverage. In: Proceedings of the IEEE international conference on software maintenance, pp 92–101. doi:10.1109/ICSM.2001.972715
18. Staats M, Gay G, Whalen M, Heimdahl M (2012) On the danger of coverage directed test case generation. In: Proceedings of the 15th international conference on fundamental approaches to software engineering, FASE’12. Berlin, Springer, pp 409–424. doi:10.1007/978-3-642-28872-2_28
19. Büchner F (2010) Acht Irrtümer über Code Coverage. Elektronik Praxis
20. Rajan A, Whalen MW, Heimdahl MPE (2008) The effect of program and model structure on MC/DC test adequacy coverage. In: ICSE ’08: Proceedings of the 30th international conference on Software engineering. ACM, New York, pp 161–170
21. Kapoor K, Bowen J (2003) Experimental evaluation of the variation in effectiveness for DC, FPC and MC/DC test criteria. In: Proceedings of the international symposium on empirical software engineering, ISESE 2003, pp 185–194. doi:10.1109/ISESE.2003.1237977

22. Yu YT, Laub ML (2006) A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. J Syst Softw 79(5):577–590
23. Szűgyi Z, Porkoláb Z (2008) Necessary test cases for decision coverage and modified condition/decision coverage, vol 52. Periodica Polytechnica, Electrical Engineering
24. Guan J, Offutt J, Ammann P (2006) An industrial case study of structural testing applied to safety-critical embedded software. In: Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering, ISESE '06. ACM, New York, pp 272–277. doi:10.1145/1159733.1159774
25. Dupuy A, Leveson N (2000) An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In: Proceedings of the 19th digital aviation systems conference, vol 1, pp 1B6/1–1B6/7. doi:10.1109/DASC.2000.886883

Copyright of Computing is the property of Springer Science & Business Media B.V. and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.