Error Detection Rate of MC/DC for a Case Study from the Automotive Domain *

Susanne Kandl and Raimund Kirner

Institute of Computer Engineering
Vienna University of Technology, Austria
{susanne,raimund}@vmars.tuwien.ac.at

Abstract. Chilenski and Miller [1] claim that the error detection probability of a test set with full modified condition/decision coverage (MC/DC) on the system under test converges to 100% for an increasing number of test cases, but there are also examples where the error detection probability of an MC/DC adequate test set is indeed zero. In this work we analyze the effective error detection rate of a test set that achieves maximum possible MC/DC on the code for a case study from the automotive domain. First we generate the test cases automatically with a model checker. Then we mutate the original program to generate three different error scenarios: the first error scenario focuses on errors in the value domain, the second error scenario focuses on errors in the domain of the variable names and the third error scenario focuses on errors within the operators of the boolean expressions in the decisions of the case study. Applying the test set to these mutated program versions shows that all errors of the values are detected, but the error detection rate for mutated variable names or mutated operators is quite disappointing (for our case study 22% of the mutated variable names, resp. 8% of the mutated operators are not detected by the original MC/DC test set). With this work we show that testing a system with a test set that achieves maximum possible MC/DC on the code detects less errors than expected.

1 Introduction

Safety-critical systems are systems where a malfunction causes crucial damage to people or the environment. Examples are applications from the avionics domain or control systems for nuclear power plants. Nowadays also applications from the automotive domain become more and more safety-critical, for instance advanced driver assistance systems, like steer-by-wire or drive-by-wire. Safety-critical embedded systems have to be tested exhaustively to ensure that there are no errors in the system. The evaluation of the quality of the testing process is usually done with some code coverage metrics that determine the proportion of the program that has been executed within the testing process. For instance, a value of 60% for decision coverage means that 6 of 10 branches of all if-else decisions have been tested. Apart from very simple and small programs it is in general not possible to test all execution paths. This is especially true for programs with

* This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project “Sustaining Entire Code-Coverage on Code Optimization (SECCO)” under contract P20944-N13.

complex boolean expressions within the decisions. For a decision depending on a condition that consists of *n* boolean subconditions we would need to generate 2^n inputs to test all possible combinations. That means the testing effort would grow *exponentially* with increasing complexity of the condition within the decision.

Modified condition/decision coverage (MC/DC) is a metric originally defined in the standard DO-178B [2], a standard for safety-critical systems in the avionics domain. In principle MC/DC defines that the set of test data has to show that each condition within a decision can independently, i.e., while the outcome of all other conditions in the decision remain constant, influence the outcome of the decision. It can be shown that MC/DC needs only *n+1* test cases to test a decision that contains *n* conditions. Thus the testing effort grows only *linearly* with the number of conditions per decision. Until now MC/DC was mainly an important coverage metric for applications from the avionics domain. Due to the fact that more and more applications for cars are also high-safety critical and because of the fact that these applications become even more complex, the coverage metric MC/DC is now also an issue for the testing process for components from the automotive domain. Existing standards for safety-critical systems for cars are IEC 61508 [3] or ISO 26262 (Road vehicles - Functional safety) [4] which is available as a draft, the final version is expected for 2011.

An ideal testing process is capable of finding any error within the system. Our aim in practice is a test set consisting of the smallest possible number of test cases that is able to detect as many errors as possible. MC/DC is seen as a suitable metric to evaluate the testing process of safety-critical systems with a manageable number of test cases. But what about the effective error detection rate of MC/DC for a real case study? In [1] it is stated that the error detection probability of MC/DC is nearly 100% for an increasing number of test cases. On the other hand side there are examples containing coding errors for which the probability of detecting an error with an MC/DC adequate test set is actually zero, see [5]. This contradiction motivated us to evaluate the error detection rate of an MC/DC adequate test set for a real case study from the automotive domain. Our goal was to find out how the coverage correlates with the error detection rate and to prove whether a test set with full MC/DC on the code is capable to find the errors.

For our experiments we define three error scenarios: In the first scenario only concrete *values* for output variables are changed. The second error scenario focuses on errors in the *names* of output variables. The third error scenario considers errors of the *operators* in the boolean expressions in the decisions. The test cases are generated automatically with a model checker to confirm a suitable MC/DC test set. The test runs are executed with the mutated program versions and the MC/DC adequate test set. The results show that a MC/DC adequate test set is capable to reveal all errors of concrete values, but it fails in detecting errors for variable names or operators.

The paper is organized as follows: In the following section we recapitulate the coverage metric MC/DC. Subsequently we describe our test case generation method. In Section 5 we describe in detail how the program versions for our test runs are mutated. Section 6 shows our experimental results. In the concluding section we discuss what these results mean for practice, i.e. how applicable MC/DC is for the evaluation of the testing process for safety-critical systems in the automotive domain.

2 Unique-Cause MC/DC

MC/DC is a code coverage metric introduced in DO-178B [2], discussed in detail in [6], resp. expanded with variations of the metric in [7]. 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` $((A \wedge (B \vee C))$ `statement_1 else statement_2`.

In 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 suite conforming to MC/DC consists of test cases that guarantee that [2], [6]:

- 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, e.g. for $(A \wedge B) \vee (A \wedge C)$ beside the independence of B and C the independence of A has to be shown for the first occurrence and the second occurrence of A. Independence is defined via *Independence Pairs*. For details please refer to [7].

Consider the example $A \wedge (B \vee C)$: The truth table is given in Table 1 (third column). In the following $\bar{0}$ represents the test case $(0, 0, 0)$, $\bar{1}$ represents the test case $(0, 0, 1)$, and so on. The independence pairs for the variable A are $(\bar{1}, \bar{5})$, $(\bar{2}, \bar{6})$ and $(\bar{3}, \bar{7})$, the independence pair for the variable B is $(\bar{4}, \bar{6})$ and the independence pair for the variable C is $(\bar{4}, \bar{5})$. Thus we have the test set for MC/DC consisting of $\{\bar{4}, \bar{5}, \bar{6}\}$ plus one test case of $\{\bar{1}, \bar{2}\}$ (remember that for n conditions we need $n + 1$ test cases).

Testcase	*A B C*	$A \wedge (B \vee C)$	$(A \wedge B) \vee C$	$(A \wedge B) \oplus C$
$\bar{0}$	0 0 0	0	0	0
$\bar{1}$	0 0 1	0	1	1
$\bar{2}$	0 1 0	0	0	0
$\bar{3}$	0 1 1	0	1	1
$\bar{4}$	1 0 0	0	0	0
$\bar{5}$	1 0 1	1	1	1
$\bar{6}$	1 1 0	1	1	1
$\bar{7}$	1 1 1	1	1	**0**

Table 1. Truth Table for Different Boolean Expressions

3 Error Detection Probability - Theoretical Assumption and Counterexample

In [1] 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", as we can see in Figure 1.

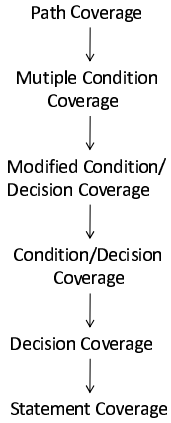

Fig. 1. Subsumption Hierarchy for Control Flow Metrics [1]

Moreover 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 [1]

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

This correlation is shown in Figure 2 for $N = 4$.

One important result of this relation is the relatively low probability of detecting errors with only two test cases, as normally required in decision or condition decision testing. 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 also. This non-intuitive result occurs because the dominant factor (the number of tests) increases with N while the sensitivity to errors remains relatively stable.[1]

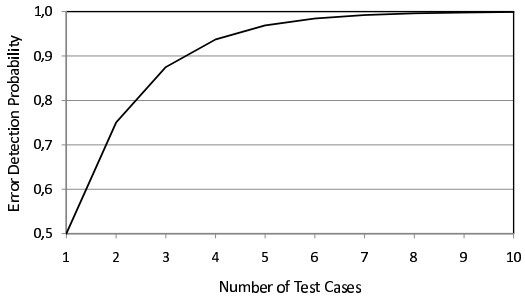

Fig. 2. Error Detection Probability of MC/DC [1]

3.1 Counterexample for Theoretical Assumption

In contrast to the assumption above one can easily construct examples similar to [5] that show that the error detection probability is actually zero for a given decision with a complex boolean expression. The example shows that for $(A \wedge B) \vee C$ the test set for achieving full MC/DC consists of the test cases $\{\bar{2}, \bar{4}, \bar{6}\}$ plus one test case out of $\{\bar{1}, \bar{3}, \bar{5}\}$. Mutating the logical operator *OR* to a logical *XOR* changes the output only for the test case $\bar{7}$, i.e. for the combination True, True, True for the variables A, B and C, thus the occurring error is not detected with a given minimal MC/DC test set. (See also the last two columns in the Table 1.) Furthermore it can also be shown that a test set for decision coverage would detect the error with a probability of 20%, so this is also a counterexample for the statement that MC/DC subsumes decision coverage.

We were interested if such counterexamples are only artificial outliers that have no influence on the overall error detection probability of MC/DC test sets or if such counterexamples really decrease the error detection probability of MC/DC. First we constructed a few examples manually for small programs that showed that in the worst case even about 30% of operator errors were not detected with a MC/DC adequate test set. This result was quite alarming for testing safety-critical systems. Then we executed our test runs on a real case study from the automotive domain. The results are given in Section 6. In the following we describe our test case generation method and the error scenarios.

4 Test Case Generation

The generation of test cases that result in full MC/DC is a non-trivial issue. As we have seen in Section 2 the test cases have to be determined by deriving the independence pairs for each sub-condition. We generate the test cases automatically with a model checker [8], in our case NuSMV[1].

[1] http://nusmv.irst.itc.it

4.1 Principle

A model checker takes a model of the system under test and proves whether a given property is valid within the model or not. In the case of a violation of a given property the model checkers produces a *counterexample*, that means a trace where the property is violated. This trace is a concrete execution path for the program we want to test and can therefore be used as a test case. The big challenge for generating suitable test cases consists mainly in the method *how* the trap properties (the properties that produce counterexamples that can be used as test cases) are formulated.

4.2 Method

Our test case generation method is motivated by works from Whalen et al [9]. In this paper a metric called *Unique First Cause Coverage (UFC)* is introduced. This metric is adapted from the MC/DC criterion and is defined on LTL (linear temporal logic) formulas. For a given set of requirements for a program to test we have to translate the requirements into a formal language (in our case LTL). Then the trap properties for deriving an MC/DC adequate test set are directly derived from these formal properties by mutation with rules that are similar to the definition of MC/DC on the code. That means if we have a sufficient set of trap properties, the model checker produces a complete test set for full MC/DC.

4.3 Example

We want to demonstrate the test case generation method on the following example. Listing 1.1 shows a small C program, for which the corresponding NuSMV-model is given in Listing 1.2. In the program we have a decision which evaluates to True (`res=42`) or False (`res=24`) depending on the boolean expression $A \wedge (B \vee C)$ similar to the example in Section 2. The NuSMV-model represents the behavior of the program in the automaton language of NuSMV. After the declaration of the variables within the ASSIGN block the model calculates `res` depending on the validity of the boolean expression. The specification for this small program consists only of two requirements, from these we can derive the properties we need to generate the test cases (MC/DC-trap properties in Listing 1.2). With the given trap properties we gain the test set for MC/DC consisting of $\{\bar{2}, \bar{4}, \bar{5}, \bar{6}\}$. Applying these test cases to the original implementation results in full MC/DC.

4.4 Unreachable Code

Applying this test case generation method to our case study from the automotive domain showed that some of the formulated trap properties are true within the model, that means that no counterexample is produced. Analyzing these cases showed that there are a few infeasible paths in the program caused by a program structure depicted in Listing 1.3. In this program structure there is no possible combination for the input variables *a* and *b* to reach the else-branch of the decision in line 7. (with 0,0 statement_2 is executed, with 0,1 also statement_2, with 1,0 - statement_3 and with 1,1 statement_1 is executed).

```c
#include <stdio.h>
typedef int bool;
int erg;

int test(bool a, bool b, bool c)
{
  if (a && (b || c))
    res = 42;
  else
    res = 24;
}

int main()
{
  test(0,0,1);
  printf("Result: %d \n", res);
}
```

Listing 1.1. C Source Code

```
MODULE main
VAR  -- Variables
 a: boolean;
 b: boolean;
 c: boolean;
 res: {0, 42, 24};
ASSIGN
  init(res) := 0;
  next(res) := case
    a & (b | c): 42;
    !(a & (b | c)): 24;
    1: res;
  esac;

-- REQUIREMENTS - original
PSLSPEC AG(a&(b|c)->AX(res=42));
PSLSPEC AG(!(a&(b|c))->AX(res=24));

-- MC/DC - trap properties
PSLSPEC AG(a&(!b|!c)->AX!(res=24));
PSLSPEC AG(a&(!b|c)->AX!(res=42));
PSLSPEC AG(a&(b|!c)->AX!(res=42));
PSLSPEC AG(!a&(b|c)->AX!(res=24));
```

Listing 1.2. NuSMV Model

4.5 Test Traces vs. Test Steps

The described test case generation method produces complete traces within the program. A test *trace* consists of multiple test *steps*. A test step is a mapping between input data and the expected output for a decision. See the example given in Listing 1.3. If we want to test the decision in line 4 we need a trace to statement_2 (line 5) and a trace to the else-branch of this decision (line 6). For testing the if-decision in line 7 again we need a trace to the else-branch of the previous decision (line 6) to reach the if-decision in line 7 and to execute the corresponding statement_3 (line 8). This yields the side effect that the generated test set is redundant in that way that a) the same statement (e.g., statement_2) maybe executed several times and b) also the corresponding values are checked multiple. For a minimal MC/DC test set we only need the test data for the different decisions, so we can reduce the test traces to singular test steps. So we reduced the generated traces to the necessary test steps to gain a *minimal* MC/DC test set (without redundant test data).

For our testing process it is important to mention that we test not only the input-output mappings of the program, but also the values of the internal variables. Consider the following case: For given input data the output in the testing process conforms to the expected output, but nevertheless there is an erroneous value during the calculation of the output values. This case would be also detected within the testing process. The

evaluation of the coverage was done with Tessy[2]. In the overall with the generated test set we achieve the maximal possible MC/DC coverage of 92,77%.

```
1 if (a && b)
2   statement_1
3 else
4    if (!a)
5      statement_2
6    else
7       if (!b)
8         statement_3
```

Listing 1.3. Unreachable Code

5 Case Study and Error Scenarios

For our experiment we define three different error scenarios:

- **Value Domain:** The first error scenario investigates how many errors are detected by the produced test set for a erroneous value, i.e. for a given variable `variable_1` with a specified value of 2, we change the specified value to the value, for instance, 3.
- **Variable Domain:** The second error scenario checks how many errors are detected by the produced test set if there are erroneous variable names for the output variables within the implementation, for instance, in the program there exist 2 variables with the names `variable_1` and `variable_2` we change some occurrences of the variable `variable_1` to the name of the other variable `variable_2` and vice versa. (In that case we just have to take care that these changes are compatible with the referring data types of the different variables.)
- **Operator Domain:** The third error scenario focuses on coding errors for the operators of the boolean expressions within the decisions. The given test set is executed on the erroneous program versions to find out if the test cases succeed or fail. For the example from Listing 1.1 we may change the first operator in $A \wedge (B \vee C)$ from the logical *AND* to the logical *OR* and vice versa for the second occurring operator.

We use the term *error* for some program property that differs from the correct implementation due to the system specification. Our case study is a low safety-critical control system from the automotive domain. It regulates a steering mechanism controlled by various input values through multiple output values, these output values are dependent on the input values and the program's internal values. To give a draft of the complexity of the program the control flow graph is given in Figure 3.

[2] http://www.hitex.com/index.php?id=module-unit-test

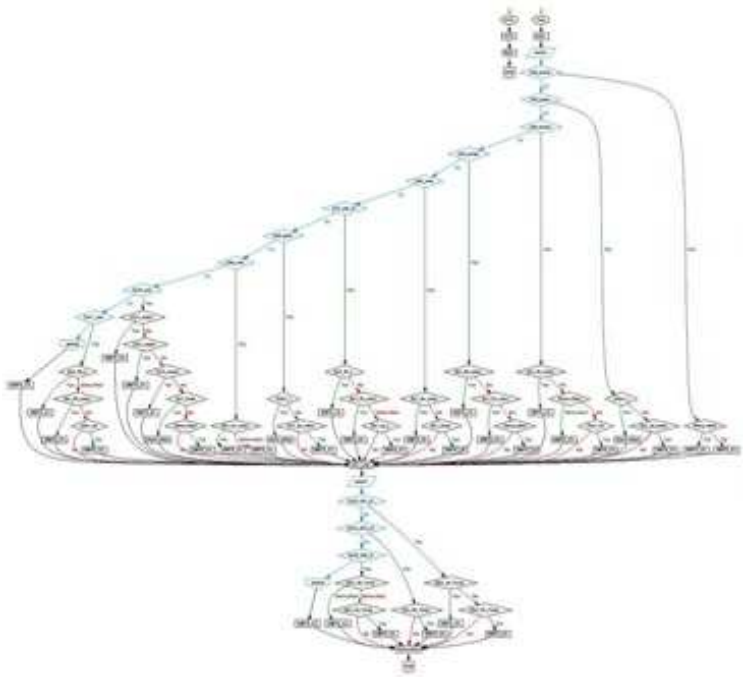

Fig. 3. Case Study - Control Flow Graph

For the test runs the original program is mutated systematically on basis of the predefined error scenarios and the resulting program versions were executed with the given MC/DC test set. If at least one test case fails in the test run, we know that the coding error has been detected, otherwise if all test cases run successfully we register that the error has not been detected in the test run.

6 Experimental Results of the Testing Process

Remember that with our test case generation method we gained a minimal test set that achieves 92,77% MC/DC on the code. This is the maximum value for the achievable coverage due to some parts of unreachable code. We mutated the original program referring to the three error scenarios (Value Domain *Err_Val*, Variable Domain *Err_Var* and Operator Domain *Err_Op*) and executed it with the test sets for different values of MC/DC, namely a test set with 20%, 40%, 60%, 80% and the maximum of 92,77% coverage. The following table shows the percentage of detected errors, i.e. a value of 28% means that 28 out of 100 error were detected. The results are also given in the diagram Figure 4.

Coverage	*Err_Val*	*Err_Var*	*Err_Op*
20	16	14	40
40	36	28	58
60	46	40	80
80	70	56	90
max	100	78	92

Table 2. Error Detection Rate (in %) for Test Sets with Different MC/DC Coverage

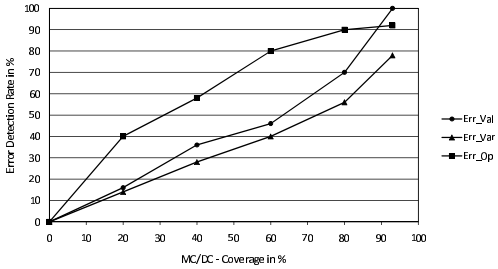

Fig. 4. Experimental Results

7 Discussion of the Results

As we can see the error detection rate for errors in the value domain increases like expected with the increasing coverage, resp. with a larger test set. For the maximum possible coverage the error detection rate is indeed 100% for our case study. This means full MC/DC coverage guarantees that an error will be detected with a given test set.

The results for errors in the variable and the operator domain differ significantly from that. The error detection rate for errors in the variable domain also increases with the coverage, resp. with a bigger set of test cases, but with the complete test set there are still 22% of errors undetected. Looking at the original program we see that there are several statement blocks where different variables are assigned. See Listing 1.4. If the name of the variable_1 in line 2 is changed to the name variable_2, the assignment of value_1 to the variable_1 gets lost, that means the value for variable_2 is still correct, the value for variable_1 may be correct (depending on the initial value before the assignment in line 2). Vice versa if we mutate the name of variable_2 in line 3 to variable_1, the value of variable_1 is overwritten with value_2. This may also be undetected for the case value_1 equals value_2, which is an improbable coincidence for integer values but quite thinkable for boolean values. This may be an explanation for such a high amount of undetected errors for variable names.

```
1 if (a && b)
2     variable_1 = value_1
3     variable_2 = value_2
```

Listing 1.4. Mutation Variable Name

For the third error scenario with mutated operators for the boolean expressions within decisions we see that already a test set with low coverage is capable to identify many errors, for instance the test set with 40% coverage already finds more than the half of errors. But still with a complete MC/DC test 8% of erroneous operators remain undetected. This value demonstrates the empirical evaluation of the example given in Section 3.1. We think that 22, resp. 8 of 100 undetected errors for a safety-critical system is quite risky.

8 Related Work

Besides the original definition of the code coverage metric MC/DC in the standard DO-178B [2] and the corresponding document *Final Clarification of DO-178B* [6] the metric is discussed and extended in [7]. The applicability is studied in Chilensky and Miller [1], in this document an assumption for the error detection probability is given but not proved with empirical data. An empirical evaluation of the MC/DC criterion can be found in [10]. Although the most important issue for the quality of a code coverage metric for safety-critical systems is indeed the capability of detecting an error, it is surprising that there are hardly any empirical studies of the error detection probability of MC/DC for real case studies. A principal comparison of structural testing strategies is [11], whereas an empirical evaluation of the effectiveness of different code coverage metrics can be found in [12]. Also in [13] MC/DC is compared to other coverage criteria for logical decisions. Rajan et al. show in [14] that even the structure of the program (for instance, if a boolean expression is inlined or not) has an effect on the MC/DC coverage. The introduced test case generation method using model checkers is described in Gargantini et al. [15], Hamon et al. [16] and Okun et al. [17] and [18]. For the test case generation of MC/DC adequate tests the trap properties have to be formulated to enforce the model checker to produce the appropriate paths we need to achieve MC/DC. This is discussed in Raydurgan and Heimdahl [19] or Whalen et al. [9]. The last work gives results that show that for a given complete test set for MC/DC (derived from the implementation or model of the SUT) does not achieve full MC/DC caused by special structure in the code, for instance, macros, that where not considered in the model used for test case generation.

9 Summary and Conclusion

A testing process is only as good as it is capable to reveal errors within the system under test. Coverage metrics like MC/DC are a means of evaluating the testing process, i.e. to measure which parts of the program have been executed within the testing process. In the standard DO-178B a high-safety critical system has to be tested with a test set that achieves full MC/DC coverage on the code. Recently upcoming standards like ISO 26262 will also prescribe this metric for safety-critical applications from the automotive domain. In this work we have shown that by achieving full MC/DC during testing it is not guaranteed that the probability of undetected errors is sufficiently low concerning reliability requirements for safety-critical system. An MC/DC adequate test set seems to be capable to reveal all errors in the value domain, but many errors concerning erroneous variable names or erroneous operators are not detected with this test set: for our case study 22%, resp. 8% of the errors were not detected which is really precarious for a safety-critical system. Similar works (e.g., [14]) also show that MC/DC is not robust to structural changes in the implementation.

Overall it is important to be aware of that so far MC/DC is the best metric for testing safety-critical system with complex boolean expressions within decisions referring to the tradeoff between testing effort (number of test cases to achieve full coverage) and efficiency in the error detection rate, *but* although achieving full MC/DC coverage there may be still a high amount of errors undetected.

References

1. Chilenski, J., Miller, S.: Applicability of modified condition/decision coverage to software testing. Software Engineering Journal **9**(5) (Sep 1994) 193–200
2. RTCA Inc.: DO-178B: Software Considerations in Airborne Systems and Equipment Certification. Requirements and Technical Concepts for Aviation, Washington, DC (December 1992)
3. International Electrotechnical Commission: IEC 61508: Functional Safety of Electrical/ Electronic/ Programmable Safety-Related Systems (1999)
4. ISO: International Organization for Standardization: ISO 26262: Functional safety road vehicles, draft (2009)
5. Bhansali, P.V.: The MCDC paradoxon. SIGSOFT Softw. Eng. Notes **32**(3) (2007) 1–4
6. RTCA Inc.: 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 (October 2001)
7. John Joseph Chilenski: An investigation of three forms of the modified condition decision coverage (MCDC) criterion. U.S.Department of Transportation, Federal Aviation Administration, DOT/FAA/AR-01/18 (April 2001)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000)
9. Whalen, M.W., Rajan, A., Heimdahl, M.P., Miller, S.P.: Coverage metrics for requirements-based testing. In: ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis, New York, NY, USA, ACM (2006) 25–36
10. Dupuy, A., Leveson, A.: An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. Digital Aviation Systems Conference (October 2000)
11. Ntafos, S.: A comparison of some structural testing strategies. Software Engineering, IEEE Transactions on **14**(6) (June 1988) 868 –874
12. Kapoor, K., Bowen, J.: Experimental evaluation of the variation in effectiveness for DC, FPC and MC/DC test criteria. In: Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on. (Sept.-1 Oct. 2003) 185–194
13. Yu, Y.T., Laub, M.L.: A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. Journal of Systems and Software **79**(Issue 5) (May 2006) 577–590
14. Rajan, A., Whalen, M.W., Heimdahl, M.P.: 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, New York, NY, USA, ACM (2008) 161–170
15. Gargantini, A., Heitmeyer, C.: Using Model Checking to Generate Tests From Requirements Specifications. In: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering. (1999) 146–162
16. Hamon, G., de Moura, L., Rushby, J.: Generating Efficient Test Sets with a Model Checker. In: Proceedings of the 2nd International Conference on Software Engineering and Formal Methods. (2004) 261–270
17. Okun, V., Black, P., Yesha, Y.: Testing with model checkers: Insuring fault visibility (2003)
18. Okun, V., Black, P.E.: Issues in software testing with model checkers (2003)
19. Rayadurgam, S., Heimdahl, M.P.: Generating MC/DC adequate test sequences through model checking. In: Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop – SEW-03, Greenbelt, Maryland (December 2003)