Reinforced Condition/Decision Coverage (RC/DC): A New Criterion for Software Testing

Sergiy A. Vilkomir and Jonathan P. Bowen

South Bank University, Centre for Applied Formal Methods
School of Computing, Information Systems and Mathematics
103 Borough Road, London SE1 0AA, UK
Email: {*vilkoms, bowenjp*}*@sbu.ac.uk*
URL: *http://www.cafm.sbu.ac.uk/*

Abstract. A new Reinforced Condition/Decision Coverage (RC/DC) criterion for software testing is proposed. This criterion provides further development of the well-known Modified Condition/Decision Coverage (MC/DC) criterion and is more suitable for testing of safety-critical software. Formal definitions in the Z notation for RC/DC, as well as MC/DC, are presented. Specific examples of using of these criteria are considered and some features are formally proved.

1 Introduction

Software testing criteria determine requirements for the scope and the volume of software testing. The requirements for testing logical structure of programs are specified using so-called control-flow criteria. The aim of these criteria is testing $decisions$ (the program points at which the control flow can divide into various paths) and $conditions$ (atomic predicates which form component parts of decisions) in a program.

One of the simplest control-flow criteria is the decision coverage criterion which states that every decision in the program has taken all possible outcomes at least once [13]. This criterion requires only two test cases for each binary decision. The greatest number of test cases is required by the multiple condition coverage criterion which states that all possible combinations of condition outcomes in each decision have been invoked a minimum of once [13].

The decision coverage criterion is weak and not sufficient, especially for the testing of safety-critical software. The multiple condition coverage criterion requires 2^n test cases for a decision made up of n conditions and this is often not possible in practice. So it is necessary to use an intermediate criterion that combines sufficient scope of testing with a relatively small number of test cases. One of such criteria is the Modified Condition/Decision Coverage (MC/DC), which requires testing of the independent influence of every condition on the decision. This criterion has been introduced in the RTCA/DO-178B standard [16], which provides regulatory requirements for avionics software.

We considered the formal definitions of the main control-flow criteria in [17, 19]. For those requiring more background information, a review of control-flow criteria with a fuller list of relevant references is available [19]. In this paper we propose the further development of this approach. The paper is structured as follows. Section 2 presents a detailed analysis of MC/DC. A new version of the definition in the Z notation [1] is proposed and the explanation how this formal approach can eliminate the ambiguity of informal definitions is given. A specific example using MC/DC is considered, illustrating the interdependence of the conditions and decisions. We analyze a major shortcoming of the MC/DC criterion, namely the deficiency of requirements for the testing of the "false operation" type of failures. Examples of situations when failures of this type are present are considered. These have especially vital importance for safety-critical applications in particular.

To eliminate the shortcoming of MC/DC, we propose a new Reinforced Condition/Decision Coverage (RC/DC) criterion, which is considered in Section 3. Z schemas for the formal definition of RC/DC and examples of its application are provided.

2 MC/DC

2.1 General definition

The definition of the MC/DC criterion, according to [16], is the following:

> *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 on 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 the 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.*

The maximum number of required tests for a decision with n conditions is $2n$. The place of MC/DC in the hierarchy of control-flow criteria is given in Figure 1. The definitions and analysis of these criteria are considered in [13–15, 19, 22].

The first part of the MC/DC definition (*every point of entry and exit in the program has been invoked at least once*) is just the standard statement coverage criterion. This part is traditionally added to all control-flow criteria and is not directly connect with the main point of MC/DC.

The second and the third parts of the definition are just the condition and decision coverage criteria. The inclusion of these parts in the definition of MC/DC could be considered excessive because satisfiability of the condition and decision coverage results from the main part of the MC/DC definition: *each condition has been shown to independently affect the decision's outcome*.

The key word in this definition is "independently"; i.e., the aim of MC/DC is the elimination during testing of the mutual influence of the individual conditions and the testing of the correctness of each condition separately.

Investigation of MC/DC has initially been considered in [4, 5, 10]. Detailed consideration of the different aspects of this criterion was carried out more recently (1999–2001) in [2, 6, 7, 9]. The successful practical application of MC/DC for satellite control software has been evaluated [8] though the difficulties during the analysis of this type

$$\begin{tikzcd}
 & \text{multiple condition coverage} \arrow[d] & \\
 & \text{modified condition/decision coverage} \arrow[d] & \\
 & \text{full predicate coverage} \arrow[d] & \\
 & \text{decision/condition coverage} \arrow[dl] \arrow[dr] & \\
\text{decision coverage} \arrow[dr] & & \text{condition coverage} \arrow[dl] \\
 & \text{statement coverage} &
\end{tikzcd}$$

Fig. 1. The hierarchy of control-flow criteria.

of coverage (e.g., it is extremely expensive to carry out and can affect staff morale and time) were also addressed [3]. The application of this criterion in the testing of digital circuits was considered in [12]. A number of software tools (LDRA Testbed, McCabe, ATTOL, CodeTEST, Cantata++, etc.) support the MD/DC criterion.

However, it should be noted that the original definition of the MC/DC criterion allows different interpretations and understanding during the application of the criterion. The informal definition gives no precise answer to some practical questions. For example:

– How to handle the situation when it is impossible to vary a condition and a decision while holding fixed all other conditions (see Section 2.3 for an example of such a situation). To assume that such a condition does not satisfy the MC/DC criterion is probably not the best way of dealing with this situation. In any case, such conditions should be checked during testing.
– How to understand multiple occurrences of a condition in a decision. For example, for a decision of the form $(A \wedge B) \vee (\neg A \wedge C)$, should we assume three conditions (A, B, and C) or four (the first A, B, C, and the second A) conditions? Both approaches have been used [4, 9] but the last one seems unnatural for many situations.
– How to treat degenerate conditions and decisions, which are either all 0 or all 1. Of course, the appearance of such conditions should attract the attention of the tester and be justified. But if it is valid for some reason, does it mean that in this case MC/DC is not satisfied because such conditions cannot be varied?

– How to consider coupled conditions, i.e., conditions that cannot be varied independently. According to [4], two or more conditions are strongly coupled if varying one always varies the other, and weakly coupled if varying one sometimes, but not always, varies the others. However, it is questionable whether strongly coupled conditions really exist.

For a more precise definition of the MC/DC criterion, eliminating inaccuracies and answering the above questions, we propose a formal definition of MC/DC using the Z notation.

2.2 Formal definition in the Z notation

We considered the formal definitions in the Z notation for all main control-flow criteria, including MC/DC, in [19]. In this paper, a slightly different approach is proposed. Within the framework of this approach, we distinguish the difference between input variables for a whole program and input variables for an individual decision. It allows us to consider possible changes of values of input variables and the multiple execution a decision in a loop. A more precise definition of the notion of a decision is introduced gradually and used in the next section for the formulation of a formal definition for RC/DC.

For defining the criteria, the given sets $STATEMENT$ and $INPUT$ are used:

$$[STATEMENT, INPUT]$$

The $STATEMENT$ set contains all possible program statements. The second set contains all possible values of all input, output and internal variables during the running of the program. In practice, the elements of $INPUT$ are values associated with variables under consideration at any particular part of the program, but this level of detail is not required at this level of abstraction. We use the name $INPUT$ to emphasize the fact that the members of this set are the input data relative to decisions and conditions in a program. The set $startinput$ is a set of the values of the input variables only and is a subset of $INPUT$. Some of the values of the input variables (namely, the $starttest$ set) are used as testing data.

$$\begin{array}{|l}
startinput, starttest : \mathbb{P}\, INPUT \\
statementinput : STATEMENT \times INPUT \rightarrowtail\!\!\!\!\!\!\!\!\;\;\; \mathbb{P}\, INPUT \\
\hline
starttest \subseteq startinput \\
\operatorname{dom} statementinput = STATEMENT \times startinput
\end{array}$$

The $statementinput$ function returns (for each program statement and each fixed values of input variables) all values of all program variables which are possible when this statement is executed. It can be several different sets of values (several different elements of $INPUT$) for one decision because of potential multiple execution of this decision in a loop.

Now we can create a Z schema[1] for the statement coverage criterion, which is a component of all other control-flow criteria including MC/DC.

[1] All schemas in this paper have been checked using the ZTC type-checker package [11].

StatementCoverage

$st : STATEMENT$

$\bigcup\{i : starttest \bullet statementinput(st, i)\} \neq \varnothing$

If for some statement st the value of $statementinput$ is the empty set for all testing data, this means that this statement is never executed during test runs of a program; i.e., the testing data does not satisfy the statement coverage criterion.

We now introduce some definitions. The $Bool$ set contains values for logical variables: 1 ($TRUE$) and 0 ($FALSE$):

$$Bool == \{0, 1\}$$

We encode this as numbers since this is a standard nomenclature by many testers. We use the elements of this set as values of $cond$, a set of partial "logical" functions on $INPUT$:

$$cond == INPUT \rightarrow\!\!\!\!\!+ \; Bool$$

The following schema describes a decision. We consider a decision as a program statement ($decst$) and an associated logical function ($value$).

Dec

$decst : STATEMENT$
$value : cond$
$decinput, decinput0, decinput1, testset : \mathbb{P}_1\ INPUT$
$argdec : \mathbb{P}_1\ cond$

$decinput = \mathrm{dom}\ value = \bigcup\{i : startinput \bullet statementinput(decst, i)\}$
$decinput0 = \{i : decinput \mid value\ i = 0\}$
$decinput1 = \{i : decinput \mid value\ i = 1\}$
$\langle decinput0, decinput1\rangle$ partitions $decinput$
$testset = \bigcup\{i : starttest \bullet statementinput(decst, i)\}$
$argdec \subseteq \{c : cond \mid \mathrm{dom}\ c = decinput \wedge \mathrm{ran}\ c = Bool\}$

The $decinput$ set contains data (values of input, output, and internal variables), which activates the given decision. The $decinput0$ and $decinput1$ sets contain data, for which the decision equals 0 and 1 respectively. These sets partition $decinput$; i.e., $decinput0 \cup decinput1 = decinput$ and $decinput0 \cap decinput1 = \varnothing$.

The $argdec$ set contains all conditions (atomic predicates), which make a decision. For example, if a decision is of the form $A_1 \vee B_1$, then the A_1 and B_1 conditions form part of the decision and $argdec$ is $\{A_1, B_1\}$ (see also an example in section 2.3). These conditions are the arguments of the logical formula, which determines the decision $value$ function uniquely.

The $testset$ set contains data from $INPUT$, which activating the given decision during test runs of a program, i.e., when the $starttest$ set is used as input data. The members of $testset$ are testing data for a given decision. If a decision statement is executed multiple times inside a loop, one test case for the whole program from $starttest$ (one test run) generates several test cases for the decision from $testset$.

The relation $InputPairs$ describes pairs of data from $INPUT$. It is convenient to use this type because we always apply a pair of test sets for testing when varying each condition or decision.

$$InputPairs == INPUT \leftrightarrow INPUT$$

Consider $DecModified$, a modified version of the Dec schema. Two sets of pairs of data are considered for each condition.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DecModified}} \\
Dec \\
changedec, changedecfix : cond \leftrightarrow\!\!\!\!\!\!\!+ \;\; InputPairs \\
\hline
\operatorname{dom} changedec = \operatorname{dom} changedecfix = argdec \\
\operatorname{ran} changedec \cup \operatorname{ran} changedecfix \subseteq decinput \leftrightarrow decinput \\
\forall\, c : cond \mid c \in argdec \bullet \\
\quad changedec\ c = \{i_0, i_1 : decinput \mid value\ i_0 \neq value\ i_1 \land c\ i_0 \neq c\ i_1\} \land \\
\quad changedecfix\ c = \{i_0, i_1 : decinput \mid (i_0, i_1) \in changedec\ c \land \\
\qquad (\forall\, othercond : argdec \mid othercond \neq c \bullet othercond\ i_0 = othercond\ i_1)\} \\
\hline
\end{array}$$

The set $changedec\ c$ is a set of pairs of data which simultaneously varying the decision and condition c, i.e., condition c equals 0 for one element of the pair and equals 1 for another element. The $changedecfix\ c$ set contains pairs of data which varying the decision and given and only given condition c, i.e., for all other conditions from the decision, the condition value for the first element of the pair coincides with the condition value for the second element. Obviously, $changedecfix\ c \subseteq changedec\ c$.

For definition of MC/DC (and, later, RC/DC), the $choice$ function is used.

$$\begin{array}{|l}
choice : InputPairs \times InputPairs \rightarrow InputPairs \\
\hline
\forall\, a, b : InputPairs \bullet \\
\quad (a \neq \varnothing \Rightarrow choice(a, b) = a) \land (a = \varnothing \Rightarrow choice(a, b) = b)
\end{array}$$

The arguments are two sets. If the first one is not empty the function just returns it; otherwise, the second set is returned.

Now we can create a formal definition of MC/DC. For each condition in each decision, the aim of this criterion is to have, as a part of the testing data, pairs of input data that vary this condition simultaneously with the decision while, if it is possible, fixing all other conditions. The following Z schema captures MC/DC:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{MC\_DC}} \\
StatementCoverage \\
\hline
\forall\, DecModified;\ c : cond \mid c \in argdec \bullet \\
\quad (testset \times testset) \cap choice(changedecfix\ c, changedec\ c) \neq \varnothing \\
\hline
\end{array}$$

Let us prove that it is always possible to choose the testing data, which satisfy MC/DC, i.e., that $choice(changedecfix\ c, changedec\ c) \neq \varnothing$, using the method of the proof by contradiction.

Lemma 1

$MC\_DC;\ c : cond \vdash choice(changedecfix\ c, changedec\ c) \neq \varnothing$

Proof

$choice(changedecfix\ c, changedec\ c) = \varnothing$ [assumption]

$\Leftrightarrow changedec\ c = \varnothing$ [definition of $choice$]

$\Leftrightarrow \neg\ (\exists\, i_0, i_1 : decinput \mid c\ i_0 \neq c\ i_1 \bullet$ [definition of $changedec$]
$\quad value\ i_0 \neq value\ i_1)$

$\Leftrightarrow \forall\, i_0, i_1 : decinput \mid c\ i_0 = 0 \wedge c\ i_1 = 1 \bullet$ [logic]
$\quad value\ i_0 = value\ i_1$

$\Rightarrow \forall\, i_0, i_1 : decinput \mid$ $[decinput0, decinput1 : \mathbb{P}_1\ INPUT]$
$\quad c\ i_0 = 0 \wedge c\ i_1 = 1 \bullet \exists\, i_2 : decinput \bullet$
$\quad value\ i_2 \neq value\ i_0 \wedge value\ i_2 \neq value\ i_1$

$\Rightarrow \forall\, i_0, i_1 : decinput \mid c\ i_0 = 0 \wedge c\ i_1 = 1 \bullet$ $[c\ i_2 = 0 \vee c\ i_2 = 1]$
$\quad \exists\, i_2 : decinput \bullet (c\ i_2 = 0 \wedge value\ i_2 \neq value\ i_1) \vee$
$\quad (c\ i_2 = 1 \wedge value\ i_2 \neq value\ i_0)$

$\Rightarrow \exists\, i_0, i_1, i_2 : decinput \bullet$ $[c\ i_1 = 1 \wedge c\ i_0 = 0]$
$\quad (c\ i_2 \neq c\ i_1 \wedge value\ i_2 \neq value\ i_1) \vee$
$\quad (c\ i_2 \neq c\ i_0 \wedge value\ i_2 \neq value\ i_0)$

$\Rightarrow \exists\, n, m : decinput \bullet$ $[n = i_2 \wedge (m = i_0 \vee m = i_1)]$
$\quad c\ n \neq c\ m \wedge value\ n \neq value\ m$

$\Leftrightarrow changedec\ c \neq \varnothing$ [definition of $changedec$]

$\Leftrightarrow choice(changedecfix\ c, changedec\ c) \neq \varnothing$ [definition of $choice$]

$\Rightarrow$ false [contradiction]

Let us consider how the proposed formal definition of MC/DC answers the questions formulated in Section 2.1:

– *How to handle the situation when it is impossible to vary a condition and a decision while holding fixed all other conditions.* According the formal definition of MC/DC, if it is impossible to find such testing data (i.e., $changedecfix\ c = \varnothing$) we can vary the condition and the decision without fixing other conditions (i.e., take testing data from $changedec\ c$).
– *How to understand multiple occurrences of a condition in a decision.* According to the definition of a decision (in the Dec schema), we consider a set ($argdec$) of conditions that make a decision. This means that each condition is considered only once. This approach is more mathematically valid and corresponds with understanding a decision as a function of conditions.

– *How to treat degenerate conditions and decisions.* According to the definition of a decision (again in the Dec schema), both of the sets $decinput0$ and $decinput1$ are non-empty. This means that every decision should take the value of both 0 and 1 and the degenerate decisions are excluded from consideration. The Dec schema also ensures that the range of every condition is equal to $Bool$; i.e., every condition should take both 0 and 1 values and thus degenerate conditions are excluded from consideration. The reason for this approach is that degenerate conditions and decisions are always covered by any testing data. So, we consider them as satisfying MC/DC because it does not make demands on such decisions and conditions.
– *How to consider the coupled conditions.* The coupled conditions [4] make problems for selecting the testing data satisfying the MC/DC criterion. However, these problems exist only for weakly coupled conditions. As we show below (see Lemma 2), if one condition A always varies the other condition B then $A = B \lor A = \neg B$, where $\neg$ is formally defined as follows:

$$\begin{array}{|l}
\neg\ : cond \rightarrowtail\!\!\!\!\twoheadrightarrow cond \\
\hline
\forall\, c : cond \bullet \neg\ c = c \,\text{⨾}\, \{0 \mapsto 1, 1 \mapsto 0\}
\end{array}$$

So we can consider A and B as entering the same condition into a decision. In other words, strongly coupled conditions as mentioned in [4] do not exist.

Lemma 2
$MC\_DC;\ A, B : cond \vdash$
$(\forall\, i_0, i_1 : decinput \bullet A\ i_0 \neq A\ i_1 \Rightarrow B\ i_0 \neq B\ i_1) \Rightarrow (A = B \lor A = \neg\ B)$
Proof
$\neg\ ((\forall\, i_0, i_1 : decinput \bullet$ [assumption]
$\quad A\ i_0 \neq A\ i_1 \Rightarrow B\ i_0 \neq B\ i_1) \Rightarrow (A = B \lor A = \neg\ B))$
$\Leftrightarrow (\forall\, i_0, i_1 : decinput \bullet$ [logic]
$\quad A\ i_0 \neq A\ i_1 \Rightarrow B\ i_0 \neq B\ i_1) \land (A \neq B \land A \neq \neg\ B)$
$\Rightarrow (\exists\, i_0, i_1 : decinput \bullet$
$\quad A\ i_0 = 0 \land A\ i_1 = 1 \land B\ i_0 \neq B\ i_1) \land$ [ran $A = Bool$]
$\quad (A \neq B \land A \neq \neg\ B)$
$[\textbf{CASE 1}:\ B\ i_0 = 1,\ B\ i_1 = 0]$
$\Rightarrow A \neq \neg\ B$ [logic]
$\Leftrightarrow \exists\, i_2 : decinput \bullet A\ i_2 = B\ i_2$ [logic]
$[\textbf{CASE 1.1}:\ A\ i_2 = 0,\ B\ i_2 = 0]$
$\Rightarrow A\ i_2 \neq A\ i_1$ $[A\ i_1 = 1]$
$\Rightarrow B\ i_2 \neq B\ i_1$ $[A\ i_2 \neq A\ i_1 \Rightarrow B\ i_2 \neq B\ i_1]$
$\Rightarrow B\ i_2 = 1$ $[B\ i_1 = 0]$
$\Rightarrow$ false [contradiction with **CASE 1.1**]
$[\textbf{CASE 1.2}:\ A\ i_2 = 1,\ B\ i_2 = 1]$

$$\begin{array}{lr}
\Rightarrow A\ i_2 \neq A\ i_0 & [A\ i_0 = 0] \\
\Rightarrow B\ i_2 \neq B\ i_0 & [A\ i_2 \neq A\ i_0 \Rightarrow B\ i_2 \neq B\ i_0] \\
\Rightarrow B\ i_2 = 0 & [B\ i_0 = 1] \\
\Rightarrow \text{false} & [\text{contradiction with } \mathbf{CASE\ 1.2}] \\
[\mathbf{CASE\ 2}:\ \ B\ i_0 = 0,\ B\ i_1 = 1] & \\
\Rightarrow A \neq B & [\text{logic}] \\
\Leftrightarrow \exists\, i_2 : \mathit{decinput} \bullet A\ i_2 \neq B\ i_2 & [\text{logic}] \\
[\mathbf{CASE\ 2.1}:\ \ A\ i_2 = 0,\ B\ i_2 = 1] & \\
\Rightarrow A\ i_2 \neq A\ i_1 & [A\ i_1 = 1] \\
\Rightarrow B\ i_2 \neq B\ i_1 & [A\ i_2 \neq A\ i_1 \Rightarrow B\ i_2 \neq B\ i_1] \\
\Rightarrow B\ i_2 = 0 & [B\ i_1 = 1] \\
\Rightarrow \text{false} & [\text{contradiction with } \mathbf{CASE\ 2.1}] \\
[\mathbf{CASE\ 2.2}:\ \ A\ i_2 = 1,\ B\ i_2 = 0] & \\
\Rightarrow A\ i_2 \neq A\ i_0 & [A\ i_0 = 0] \\
\Rightarrow B\ i_2 \neq B\ i_0 & [A\ i_2 \neq A\ i_0 \Rightarrow B\ i_2 \neq B\ i_0] \\
\Rightarrow B\ i_2 = 1 & [B\ i_0 = 0] \\
\Rightarrow \text{false} & [\text{contradiction with } \mathbf{CASE\ 2.2}]
\end{array}$$

2.3 A case study

The contents of the proposed formal definitions are considered below. Different examples of MC/DC use that have been presented previously (for example, see [4]) have often considered only simple decisions containing two or three conditions. Using MC/DC for such decisions has no great practical use because full searching of all test cases is easy achieved. Furthermore, such examples do not reflect complicated situations, which are typical in realistic practical examples of use of this criterion. We consider a more complex example (but one that is still far from a real practical problem because of space considerations), which takes into account the following factors:

- dependence of the values of the conditions and decisions on input data;
- dependence of the specific decision on its place in the computer program, i.e., on the values of other decisions in the program;
- dependence of the conditions in the specific decision on each other, i.e., the possibility that one condition takes a value depending on the value of other conditions in this decision.

This example uses a computer program fragment, whose graph is given in Figure 2.

In this fragment, the input data x and y are read; let the value of both x and y be between 0 and 100. For this simple example, we could consider $INPUT$ as just a pair of values:

$$INPUT == (0 \,..\, 100) \times (0 \,..\, 100)$$

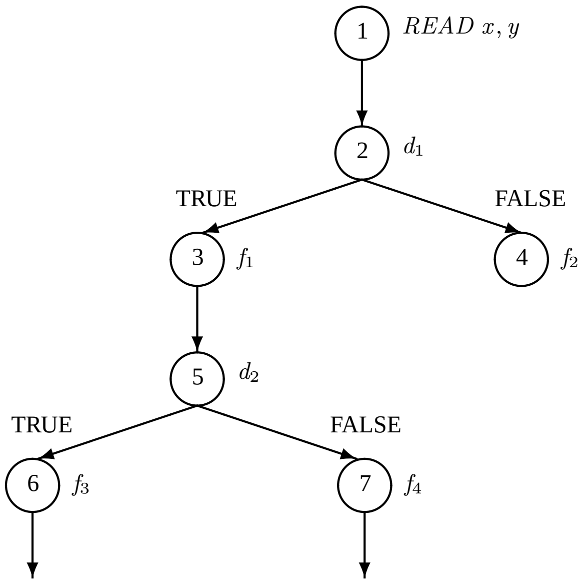

Fig. 2. Flow graph of a program fragment.

Then, depending on the values of x and y, the computation by one from four formulae $f_1 - f_4$ is implemented.

The control flow of this program is determined by two decisions: d_1 and d_2. Let d_1 depends on conditions A_1 and B_1 and d_2 depends on conditions A, B, C, and D, as it is shown below:

$$d_1, d_2 : \mathit{DecModified}$$
$$A, B, C, D, A_1, B_1 : \mathit{cond}$$
$$x, y : 0 \,..\, 100$$

$$A\ (x, y) = 1 \Leftrightarrow x > 20$$
$$B\ (x, y) = 1 \Leftrightarrow y < 60$$
$$C\ (x, y) = 1 \Leftrightarrow x > 40$$
$$D\ (x, y) = 1 \Leftrightarrow y < 80$$
$$A_1\ (x, y) = 1 \Leftrightarrow x > 20$$
$$B_1\ (x, y) = 1 \Leftrightarrow y > 60$$
$$d_1.\mathit{decinput} = \mathit{INPUT}$$
$$d_2.\mathit{decinput} = \{x, y : 0 \,..\, 100 \mid x > 20 \vee y > 60\}$$
$$d_1.\mathit{argdec} = \{A_1, B_1\}$$
$$d_2.\mathit{argdec} = \{A, B, C, D\}$$
$$d_1.\mathit{value}\ (x, y) = 1 \Leftrightarrow A_1\ (x, y) = 1 \vee B_1\ (x, y) = 1$$
$$d_2.\mathit{value}\ (x, y) = 1 \Leftrightarrow$$
$$((A\ (x, y) = 1 \wedge B\ (x, y) = 1) \vee (C\ (x, y) = 1 \wedge D\ (x, y) = 1))$$

For the d_1 decision, $d_1.decinput$ is all possible $INPUT$s and both conditions A_1 and B_1 are independent. The examples of the testing data satisfy the MC/DC criterion for d_1 are given in Table 1.

num	values			testing data	variations		MC/DC
	A_1	B_1	$d_1.value$	(x, y)	A_1	B_1	
1	1	1	1	$(50, 70)$			
2	1	0	1	$(50, 50)$	$*$		+
3	0	1	1	$(10, 70)$		$*$	+
4	0	0	0	$(10, 50)$	$*$	$*$	+

Table 1. Testing data satisfied the MC/DC criterion for d_1.

For the d_2 decision, the set $d_2.decinput$ is more restricted than the full set of possible $INPUT$s because of the interdependency of $d1$ and $d2$. The members of $d_2.decinput$ are only the input data for which d_1 equals 1, i.e., is TRUE.

The conditions in d_2 are interdependent in pairs. For conditions A and C the situation $(A = 0 \wedge C = 1)$ is impossible because $(C = 1) \Rightarrow (A = 1)$. For conditions B and D the situation $(B = 1 \wedge D = 0)$ is impossible because $(B = 1) \Rightarrow (D = 1)$.

As it is shown in Table 2, only 8 of the 16 combinations of condition values are possible.

num	values					testing data	variations				MC/DC
	A	B	C	D	$d_2.value$	(x, y)	A	B	C	D	
1	1	1	1	1	1	$(50, 50)$					
2	1	1	1	0	-	impossible					
3	1	1	0	1	1	$(30, 50)$	$\bullet$	$*$			+
4	1	0	1	1	1	$(50, 70)$			$*$	$*$	+
5	0	1	1	1	-	impossible					
6	1	1	0	0	-	impossible					
7	1	0	1	0	0	$(50, 90)$				$*$	+
8	1	0	0	1	0	$(30, 70)$		$*$	$*$		+
9	0	1	1	0	-	impossible					
10	0	1	0	1	-	impossible					
11	0	0	1	1	-	impossible					
12	1	0	0	0	0	$(30, 90)$					
13	0	1	0	0	-	impossible					
14	0	0	1	0	-	impossible					
15	0	0	0	1	0	$(10, 70)$	$\bullet$				+
16	0	0	0	0	0	$(10, 90)$					

Table 2. Testing data satisfied the MC/DC criterion for d_2.

The combinations $(0,1,1,1)$, $(0,1,1,0)$, $(0,0,1,1)$, and $(0,0,1,0)$ are impossible because of the value of the condition A. Combinations $(1,1,1,0)$, $(1,1,0,0)$, and $(0,1,0,0)$ are impossible because of the value of the condition D. The combination $(0,1,0,1)$ is impossible because of the value of the decision d_1.

Testing data satisfying the MC/DC criterion for the conditions B, C, and D are shown in Table 2 marked as '$*$'.

For the condition A it is impossible to choose similar combinations, i.e., combinations for which the values of A and d_2 are changed and the values of B, C, and D are fixed. So, following the formal definition of MC/DC, in this case for the condition A it is sufficient to take any combinations which vary A and d_2 without fixing other conditions. For example, it is possible to take combinations $(1,1,0,1)$ and $(0,0,0,1)$, for which the values of A and d_2 vary simultaneously (marked '$\bullet$' in Table 2). The testing set satisfying the MC/DC criterion for the decision d_2 consists of five pairs of the input data (marked '+' in Table 2).

2.4 The main shortcoming of MC/DC

As already mentioned in this paper, the MC/DC criterion is used mainly for testing of safety-critical avionics software [16]. The main aim of MC/DC is testing situations when changing a condition implies a change in a decision. Often a decision can be associated with some safety-critical operation of a system. In such cases, MC/DC requires the testing of situations when changing one condition has some consequence on the operation of the system. A software error in such situations could involve "*non-operation*" (inability to operate on demand) type of failures. Such situations are extremely important and the MC/DC requirements are entirely reasonable.

But, as we show below, this criterion has one substantial shortcoming, not previously mentioned in the literature, namely deficiency of requirements for testing of the "*false actuation*" (operation without demand) type of failures. This could make this criterion insufficient for many safety-critical applications.

The false actuation of a system could be invoked by a software error in situations when changing a condition should not imply changing a decision. We now consider several examples.

Railway points: Consider a railway computer control system and a decision that is responsible for switching over the points by which trains can be routed in one direction to another.

Let there be two tracks (main and reserved); the condition determines track states (which may be either occupied or clear) and the decision determines changing the route from the main track to the reserved track and vice versa. Consider two situations for the non-operation and false actuation types of failures.

The first situation is when the main track becomes occupied (varying the condition) and, therefore, it is necessary to switch over the points to the reserve track (varying the decision). The failure in this situation involves keeping the value of the decision instead of varying it; this means non-operation of the system and could result in a possible crash.

The second situation is when the reserved track becomes occupied (varying the condition) and, therefore, it is necessary to keep the main track as a route (keeping the decision). The failure in this situation involves varying the value of the decision instead of keeping it fixed that means false operation of the system and a possible crash.

Thus, from the safety point of view, these situations are symmetrical and can lead to a crash. Therefore, both types of failures should be considered and both situations should be tested with the same accuracy.

Nuclear reactor protection system: Consider a decision that is responsible for actuating a reactor protection system at a nuclear power plant (i.e., the reactor shutdown) and a condition that describes some criterion for the actuation (e.g., excessive pressure over some specified level). Varying this decision because of variation of the condition should be tested since failure in this situation means the non-operation of the system in case of emergency conditions and can lead to the nuclear accident.

Nevertheless, keeping the value of the decision is also important. The failure in this situation means the false actuation of the system during normal operating and can lead to non-forced reactor shutdown, the deterioration of the physical equipment, and the underproduction of electricity.

The typical architecture of nuclear reactor protection systems (three channels with "2 from 3" logical voting) takes into account this particular problem. The use of three identical channels decreases the probability of the system not operating correctly. However, if it is only required to consider this factor, the "1 from 3" logic is more reliable. The aim of using "2 from 3" voting is to provide protection against false actuation of a system as in this case the false signal from one channel does not lead to system actuation.

Thus, during the reactor protection system software testing, it is necessary to include test cases both for varying and keeping a decision's outcomes.

Planned halt of a computer control system: Sometimes specific situations are possible, when keeping a decision is much more important for safety than varying a decision. Consider a decision that is responsible for a planned (non-emergency) halt of a continuous process control system, e.g., for planned maintenance. Conditions describe when this process is in a safe state allowing switching off of the control system. Again consider two situations.

The first situation is when the state of the process becomes safe (varying the condition) and it is possible to switch system off (varying the decision). The failure in this situation does not have grave consequences and means only a delay of the system halt.

The second situation is when the state of the process becomes unsafe (varying the condition) and the control system should continue in operation (keeping the decision). The failure in this situation means that the system is erroneously switched off. Such a fault leads to loss of control and this is important with respect to safety. So it is important to test the keeping of the value of this decision.

The examples considered above demonstrate that for many cases testing only varying a decision when varying a condition (i.e., using MC/DC) is insufficient from the safety point of view. To eliminate this shortcoming, we propose the use of a new RC/DC criterion in critical applications.

3 RC/DC

3.1 General definition

The main idea of RC/DC is for future development of MC/DC with the purpose of making it even stronger. In that way, all requirements of MC/DC are valid and a new requirement for keeping the value of a decision when varying a condition is added to the testing regime.

With the objective of ensuring compatibility and continuity with the MC/DC definition, we define RC/DC as follows:

> 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 on all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, each condition in a decision has been shown to independently affect the decision's outcome, *and each condition in a decision has been shown to independently keep the decision's outcome*. A condition is shown to independently affect *and keep* a decision's outcome by varying just that condition while holding fixed *(if it is possible)* all other conditions.

The reservation "if it is possible" is used because it is far from always being possible to independently affect or keep the value of a decision. For more accurate consideration and analysis of all possible situations we propose the formal definition of RC/DC using the Z notation in the next section.

3.2 Formal definition in the Z notation

For elaboration of the formal definition of RC/DC, we carry out further development of the Z schema, describing the notion of the decision. The $DecReinforced$ schema below differs from the $DecModified$ schema by adding four new functions: $keep0$, $keep1$, $keep0fix$ and $keep1fix$. Each of these functions connects conditions with pairs of input data.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DecReinforced}} \\
\textit{DecModified} \\
\textit{keep}0, \textit{keep}1, \textit{keep}0\textit{fix}, \textit{keep}1\textit{fix} : \textit{cond} \pfun \textit{InputPairs} \\
\hline
\mathrm{dom}\, \textit{keep}0 = \mathrm{dom}\, \textit{keep}1 = \mathrm{dom}\, \textit{keep}0\textit{fix} = \mathrm{dom}\, \textit{keep}1\textit{fix} = \textit{argdec} \\
\mathrm{ran}\, \textit{keep}0 \cup \mathrm{ran}\, \textit{keep}1 \cup \mathrm{ran}\, \textit{keep}0\textit{fix} \cup \mathrm{ran}\, \textit{keep}1\textit{fix} \subseteq \textit{decinput} \leftrightarrow \textit{decinput} \\
\forall\, c : \textit{cond} \mid c \in \textit{argdec} \bullet \\
\quad \textit{keep}0\ c = \{i_0, i_1 : \textit{decinput} \mid \textit{value}\ i_0 = \textit{value}\ i_1 = 0 \land c\ i_0 \neq c\ i_1\} \land \\
\quad \textit{keep}1\ c = \{i_0, i_1 : \textit{decinput} \mid \textit{value}\ i_0 = \textit{value}\ i_1 = 1 \land c\ i_0 \neq c\ i_1\} \land \\
\quad \textit{keep}0\textit{fix}\ c = \{i_0, i_1 : \textit{decinput} \mid (i_0, i_1) \in \textit{keep}0\ c \land \\
\qquad (\forall\, \textit{othercond} : \textit{argdec} \mid \textit{othercond} \neq c \bullet \textit{othercond}\ i_0 = \textit{othercond}\ i_1)\} \land \\
\quad \textit{keep}1\textit{fix}\ c = \{i_0, i_1 : \textit{decinput} \mid (i_0, i_1) \in \textit{keep}1\ c \land \\
\qquad (\forall\, \textit{othercond} : \textit{argdec} \mid \textit{othercond} \neq c \bullet \textit{othercond}\ i_0 = \textit{othercond}\ i_1)\}
\end{array}$$

The functions $\textit{keep}0$ and $\textit{keep}1$ assign for each condition the subset of pairs of input data that vary the condition but keep the value of the decision equal to 0 (for $\textit{keep}0$) or 1 (for $\textit{keep}1$). The difference between the $\textit{keep}0\textit{fix}/\textit{keep}1\textit{fix}$ functions and the $\textit{keep}0/\textit{keep}1$ functions is that, for the $\textit{keep}0\textit{fix}/\textit{keep}1\textit{fix}$ functions, all the other conditions are kept fixed. This is similar to the difference between $\textit{changedecfix}$ and $\textit{changedec}$ in the $\textit{DecModified}$ schema.

The introduced functions allow formulation of the formal definition of the RC/DC criterion:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RC\_DC}} \\
\textit{MC\_DC} \\
\hline
\forall\, \textit{DecReinforced};\ c : \textit{cond} \mid c \in \textit{argdec} \bullet \\
\quad (\textbf{let}\ \textit{target}0 == \textit{choice}(\textit{keep}0\textit{fix}\ c, \textit{keep}0\ c); \\
\quad \textit{target}1 == \textit{choice}(\textit{keep}1\textit{fix}\ c, \textit{keep}1\ c) \bullet \\
\qquad (\textit{target}0 \neq \varnothing \Rightarrow (\textit{testset} \times \textit{testset}) \cap \textit{target}0 \neq \varnothing) \land \\
\qquad (\textit{target}1 \neq \varnothing \Rightarrow (\textit{testset} \times \textit{testset}) \cap \textit{target}1 \neq \varnothing))
\end{array}$$

This criterion contains MC/DC as a component and also includes the requirements for testing of invariability of the decision when a condition varies. In this way, the set of test cases should contain pairs of input data from two sets $\textit{target}0$ and $\textit{target}1$, which keep (if it is possible) the value of the decision and fix (if it is possible) all other conditions. If holding other conditions is not possible, the test cases that keep the value of the decision without fixing other conditions should be used.

Let us prove that if the decision does not coincide with the condition or the condition's negation, it is always possible to choose testing data that satisfies RC/DC; i.e., $(\textit{target}0 \neq \varnothing) \lor (\textit{target}1 \neq \varnothing)$.

Lemma 3

$\textit{RC\_DC};\ c, \textit{value} : \textit{cond} \vdash$

$(\textit{value} \neq c \land \textit{value} \neq \lnot\, c) \Rightarrow (\textit{target}0 \neq \varnothing \lor \textit{target}1 \neq \varnothing)$

Proof

$\neg\ ((value \neq c \wedge value \neq \neg\ c) \Rightarrow$ [assumption]
$\quad (target0 \neq \varnothing \vee target1 \neq \varnothing))$
$\Leftrightarrow value \neq c \wedge value \neq \neg\ c \wedge target0 = \varnothing \wedge target1 = \varnothing$ [logic]
$\Rightarrow choice(keep0fix\ c, keep0\ c) = \varnothing\ \wedge$ [definition of $target0$ and $target1$]
$\quad choice(keep1fix\ c, keep1\ c) = \varnothing$
$\Leftrightarrow keep0\ c = \varnothing \wedge keep1\ c = \varnothing$ [definition of $choice$]
$\Leftrightarrow \neg\ (\exists\, i_0, i_1 : decinput \bullet$ [definition of $keep0$ and $keep1$]
$\quad c\ i_0 \neq c\ i_0 \wedge value\ i_0 = value\ i_1)$
$\Leftrightarrow \forall\, i_0, i_1 : decinput \bullet c\ i_0 \neq c\ i_1 \Rightarrow value\ i_0 \neq value\ i_1$ [logic]
$\Rightarrow c = value \vee c = \neg\ value$ [**Lemma 2**]
$\Rightarrow$ false [contradiction]

It should be noted that there are decisions that do not keep one of the values (0 or 1) when varying a condition. For example, $A \vee B$ does not keep 0 and $A \wedge B$ does not keep 1.

3.3 A case study

Consider the following criterion of the protection system actuation for the VVER-1000 type [21] nuclear reactor: the system should shut down a reactor in the case of decrease of the pressure in the first circuit to less than $150\ kg/cm^2$ under the coolant temperature of more than 260° centigrade and reactor capacity equal or more than 75% of the rated capacity or decrease of the pressure in the first circuit to less than $140\ kg/cm^2$ under the coolant temperature more than 280° centigrade.

For measurement of each input parameter (pressure p and temperature t), three sensors are used (inputs p_1, p_2, p_3 and t_1, t_2, t_3) with majority voting of inputs.

Hence, 13 conditions are used for determination of necessity of the system actuation:

$$\begin{array}{lll}
P_{11} = p_1 < 150 & P_{21} = p_2 < 150 & P_{31} = p_3 < 150 \\
P_{12} = p_1 < 140 & P_{22} = p_2 < 140 & P_{32} = p_3 < 140 \\
T_{11} = t_1 > 260 & T_{21} = t_2 > 260 & T_{31} = t_3 > 260 \\
T_{12} = t_1 > 280 & T_{22} = t_2 > 280 & T_{32} = t_3 > 280 \\
NR = N \geq 0.75 N_r & &
\end{array}$$

The decision that is responsible for this actuation criterion is:

$$(((P_{11} \wedge P_{21}) \vee (P_{11} \wedge P_{31}) \vee (P_{21} \wedge P_{31})) \wedge ((T_{11} \wedge T_{21}) \vee (T_{11} \wedge T_{31}) \vee (T_{21} \wedge T_{31})) \wedge NR) \vee (((P_{12} \wedge P_{22}) \vee (P_{12} \wedge P_{32}) \vee (P_{22} \wedge P_{32})) \wedge ((T_{12} \wedge T_{22}) \vee (T_{12} \wedge T_{32}) \vee (T_{22} \wedge T_{32})))$$

A general number of all possible combinations of values of the conditions equals $2^{13} = 8192$. Not all combinations are possible because of coupled conditions. Thus, it is impossible to have $P_{i1} = 0 \wedge P_{i2} = 1$ and also $T_{i1} = 1 \wedge T_{i2} = 0$. However, the number of possible combinations is still too large to be completely checked during practical testing.

The RC/DC criterion requires a maximum of 6 test cases for each condition (two for varying the decision, two for keeping it 0, and two for keeping it 1). So, not more than 78 combinations are required for this decision. However this number is overestimated since the same combinations can be used for testing different conditions. The minimization of the number of test cases for RC/DC could demand special analysis and be a hard task. Costs of this analysis could exceed the obtained benefit from the minimization. But if further reduction of test cases is not very important, the selection of test cases presents no difficulty.

For testing maintaining the value 0 for the decision during variation of the condition P_{11}, it is sufficient to select combinations of input data, for which $T_{11} = 0$ and $T_{21} = 0$. It ensures that the decision equals 0; therefore any possible values of the other conditions could be fixed. For testing maintaining the value 1 for the decision during variation of the condition P_{11}, it is sufficient to fix, for example, $P_{12} = P_{22} = T_{12} = T_{22} = 1$ and any allowed values of the other conditions.

4 Conclusion and Future Work

In this paper we have proposed and formalized a new Reinforced Condition/Decision Coverage (RC/DC) criterion for software testing, which strengthens the requirements of the well-known Modified Condition/Decision Coverage (MC/DC) criterion [16].

Z schemas have been formulated to provide the formal definition of MC/DC (see also [19]) in the Z notation [1], which accurately capture the particular features of this criterion. However, the MC/DC criterion does not include requirements for testing of “false operation” type failures. Such failures, as we have shown in several examples, can be highly important for safety-critical computer systems.

The proposed RC/DC criterion aims to eliminate this shortcoming and requires the consideration of situations when varying a condition keeps the value of a decision constant. Though the number of required test cases rises, this growth remains linear compared to the number of conditions in a decision, making the approach practicable. We have illustrated application of the RC/DC criterion in the testing of nuclear reactor protection system software. An important area of application of the RC/DC criterion could be using it as a regulatory requirement in standards [18, 20].

One direction for future work could be using RC/DC not only for software testing but also for integration testing of a whole computer system, assuming the system specification as a basis. Another important aim is automated generation of test inputs in line with the RC/DC criterion.

References

1. Bowen, J. P. Z: A formal specification notation. In M. Frappier and H. Habrias (eds.), *Software Specification Methods: An Overview Using a Case Study*, Chapter 1. Springer-Verlag, FACIT series, 2001, pp. 3–19.

2. Burton, S. *Towards Automated Unit Testing of Statechart Implementations*. Technical Report YCS319, Department of Computer Science, University of York, UK. September 1999.
3. Chapman, R. Industrial Experience with SPARK. *Proceedings of ACM SIGAda Annual International Conference (SIGAda 2000)*, November 12–16, 2000, Johns Hopkins University/Applied Physics Laboratory, Laurel, MD, USA.
4. Chilenski, J. and Miller, S. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, September 1994, pp. 193–200.
5. Chilenski, J. and Newcomb, P. H. Formal specification tool for test coverage analysis. *Proceedings of the Ninth Knowledge-Based Software Engineering Conference*, September 20–23, 1994, pp. 59–68.
6. DeWalt, M. MCDC. A blistering love/hate relationship. *FAA National Software Conference*, Long Beach, CA, USA, April 6–9, 1999.
7. Dolman, B. *Definition of Statement Coverage, Decision Coverage and Modified Condition Decision Coverage*. WG-52/SC-190 Discussion paper. Paper reference: D004, revision 1. Draft, September 25, 2000.
8. Dupuy, A. and Leveson, N. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. *Proceedings of the Digital Aviation Systems Conference (DASC)*, Philadelphia, USA, October 2000.
9. Hayhurst, K. J., Veerhusen, D. S., Chilenski, J. J., and Rierson, L. K. *A Practical Tutorial on Modified Condition/Decision Coverage*, Report NASA/TM-2001-210876, NASA, USA, May 2001.
10. Jasper, R., Brennan, M., Williamson, K., Currier, B., and Zimmerman, D. Test data generation and feasible path analysis. *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, Seattle, WA, USA, August 17–19, 1994, pp. 95–107.
11. Jia, X. *ZTC: A Type Checker for Z Notation. User's Guide. Version 2.03, August 1998.* Division of Software Engineering, School of Computer Science, Telecommunication, and Information Systems, DePaul University, USA, 1998.
12. Li, Y. Y. Structural test cases analysis and implementation. *42nd Midwest Symposium on Circuits and Systems*, 8–11 August, 1999, Volume 2, pp. 882–885.
13. Myers, G. *The Art of Software Testing.* Wiley-Interscience, 1979.
14. Offutt, A. J., Xiong, Y., and Liu, S. Criteria for generating specification-based tests. *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)*, Las Vegas, Nevada, USA, October 18–21, 1999, pp. 119–129.
15. Roper, M. *Software Testing.* McGraw-Hill, 1994.
16. RTCA/DO-178B. *Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Washington DC, USA, 1992.
17. Vilkomir, S. A. and Bowen, J. P. *Formalization of Control-flow Criteria of Software Testing*. Technical Report SBU-CISM-01-01, SCISM, South Bank University, London, UK, January 2001.
18. Vilkomir, S. A. and Bowen, J. P. *Application of Formal Methods for Establishing Regulatory Requirements for Safety-Critical Software of Real-Time Control Systems.* Technical Report SBU-CISM-01-03, SCISM, South Bank University, London, UK, 2001.
19. Vilkomir, S. A. and Bowen, J. P. Formalization of software testing criteria using the Z notation, *Proceedings of COMPSAC 2001: 25th IEEE Annual International Computer Software and Applications Conference*, Chicago, Illinois, USA, 8–12 October 2001. IEEE Computer Society Press, 2001, pp. 351–356.
20. Vilkomir, S. A. and Kharchenko, V. S. Methodology of the Review of Software for Safety Important Systems. *Safety and Reliability. Proceedings of ESREL'99 – The Tenth European Conference on Safety and Reliability*, Munich-Garching, Germany, 13–17 September 1999, Vol. 1, pp. 593–596.

21. Voznessensky, V. and Berkovich, V. VVER 440 and VVER-1000. Design Features in Comparison with Western PWRS. *International Conference on Design and Safety of Advanced Nuclear Power Plants*, Tokyo, October 1992, Vol. 4.
22. Zhu, H., Hall P. A., and May, H. R. Software unit test coverage and adequacy. *ACM Computing Surveys*, Vol. 29, No. 4, December 1997, pp. 336–427.