Necessary test cases for Decision Coverage and Modied Condition / Decision Coverage*

Zalán Szűgyi and Zoltán Porkoláb

Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
{lupin, gsd}@elte.hu

Abstract. Test coverage refers to the extent to which a given software verification activity has satisfied its objectives. There are several type of coverage analysis exists to check the code correctness. Usually the less strict analysis methods require fewer test cases to satisfy their requirements and it consumes less resources. Chosing test methods is a compromise between the code correctness and the available resources. However this selection should based on quantitative consideration. In this paper we concern to the Decision Coverage and the more strict Modified Condition / Decision Coverage. We examined several projects written in Ada programming language. Some of them are developed in the industry and the others are open source. We analyzed them in several aspects: McCabe metric, nesting and maximal argument number in decisions. We discuss how these aspects are affected the difference of the necessary test cases for these testing methods.

1 Introduction

Coverage refers to the extent to which a given verification activity has satisfied its objectives. Coverage measures can be applied to any verification activity, although they are most frequently applied to testing activities. Appropriate coverage measures give the people doing, managing, and auditing verification activities a sense of the adequacy [17] of the verification accomplished. [11]

The code coverage analysis contains three main steps [18], such as: finding areas of a program not exercised by a set of test cases, creating additional test cases to increase coverage and determining a quantitative measure of code coverage, which is an indirect measure of quality. Optionally it contains a fourth step: identifying redundant test cases that do not increase coverage.

The code coverage analysis is a structural testing technique (white box testing), where it compares test program behavior against the apparent intention of the source code [9] . Different types of analysis requires different set of test cases. We concern to Decision Coverage (DC), and Modified Condition / Decision Coverage (MC/DC) testing methods. The DC only requires that every lines of code in a subprogram must be executed and every decisions must be evaluated

* Supported by the Hungarian Ministry of Education under Grant FKFP0018/2002

both to true and false. The MC/DC is more strict. It contains the requirements of DC and it demands to show that every condition in a decision independently affects the outcome. It is clear, there are more test cases are needed to satisfy the requirements of MC/DC than DC. But it is not so trivial *how much* can be spared when testing by DC instead of MC/DC. In this paper we answer that question by analyzing several projects used in the industry. These projects were written in Ada programming language and we analyzed them in several aspects: McCabe metrics [12], nesting, and maximal argument number in decisions. We examined how these aspects affected the difference of the necessary test cases.

In the second chapter we describe the most frequently used coverage metrics. In the third chapter we give a detailed description about how we analyzed the source codes of projects. Then we discuss the results of our analysis in the fourth chapter. And the summary and the conclusion comes in the fifth chapter.

2 Coverage Metrics

In this chapter we describe some commonly used coverage metrics.

2.1 Statement Coverage

To achieve statement coverage, every executable statement in the program is invoked at least once during software testing. The main advantage of this method is that it can applied directly in object code and is not necessary to process source code. But this method is insensible to some control structure. Let us see the following example:

```
T* t = NULL;
if (condition)
    t = new T();
t->method();
```

In this example only one test case (where the `condition` is true) is enough to achieve 100% statement coverage because every statement is invoked once. In that case our program works fine, and we recognize it is faultless. But in the real usage, the condition can be false, and it causes in-deterministic behavior or segmentation fault.

2.2 Decision Coverage

This method requires that every statement must be invoked at last once and every decision must be evaluated as true and as false. In this case the error from the previous example turns out in testing time. This metric has the advantage of simplicity without the problems of statement coverage. A disadvantage is ignoring branches within boolean expressions which occur due to short-circuit operators. Let us see to following example:

```
if A or B then
```

Two test cases where (A = true, B = false and A = false, B = false) can satisfy the requirements of DC, but the effect of B is not tested. Thus these test cases cannot distinguish between the decision (A or B) and the decision A.

2.3 Modified Condition / Decision Coverage

The MC/DC criterion requires that every statement must be invoked at least once, every decision must be evaluated as true and as false, and each condition must be shown to independently affect the outcome of the decision. The independence requirement ensures that the effect of each condition is tested relative to the other conditions. In our example three test cases (where A = true, B = true and A = true, B = false and A = false, B = true) provide MC/DC. The MC/DC is refined by Condition / Decision Coverage. You can read more information of these coverage methods in [11], [10], [18]. MC/DC is used for various environments, from test generation [14] to measure complexity [8].

3 Analysis Method

In this chapter we describe our method to analyze the source codes written in Ada programming language. We used Antlr [19] parser generator with [20] grammar file to create the Abstract Syntax Tree (AST) of the source code. Our analysis is used on this AST.

3.1 Counting Test Cases for Decision Coverage

The Decision Coverage requires that every decision must be evaluated as true and as false at least once. So we need at least two test cases for every decision to satisfy these requirements. But one test case can test several decisions and more then two test cases are needed if a decision contains nested decisions. Let us see the following example:

```
if Condition_1 then
    if Condition_2 then
        true_statement_2;
    else
        false_statement_2;
    end if;
else
    false_statement_1
end if;
--...
if Condition_3 then
    true_statement_3;
end if;
```

Three test cases are needed to cover DC where the Condition_1, Condition_2, Condition_3 are for example: (true, true, true), (true, false, false), (false, any, any). The Condition_1 must be evaluated as true twice because it has a nested decision in its true part. The Condition_3 can be tested the same time as Condition_1 because they are in the same level.

In summary we can say, $A + B$ test cases are needed to cover a decision. A is either the number of necessary test cases that are nested within a true consequence or 1 if there is no nested decision there. B means the same but in false consequence. A subprogram may contain more decisions in a same level. If all of these decisions are unique then we calculate the $max(A_i + B_i)$ where A_i, B_i belongs to the i^{th} decision ($i = 1..n$) where n is the number of decisions.

If there are identical decisions in same level we classify them. The identical decisions will be placed into the same class. Then we consider the $max(A_j + B_j)$ where A_j, B_j belongs to the j^{th} class. We calculate A_j and B_j in the following way: A_j `= max (` $A_{j_1}..A_{j_k}$ `),` B_j `= max (` $B_{j_1}..B_{j_k}$ `)` where `k` is the number of the decisions in class `j`. A_{j_l} and B_{j_l} are the number of necessary test cases for true and false consequences of the corresponding decision. ($l = 1..k$)

3.2 Counting Test Cases for Modified Condition / Decision Coverage

In this case we have two main steps. First we count how many test cases are needed to cover the decisions separately [2] and then we check how these decisions affect each other. If a decision contains more than 15 arguments, then we calculate with argument number plus one test cases, which comes from [11].

Analyzing decisions separately

- If the decision contains only one argument or the negation of that argument we need exactly two test cases. Dealing with this case is same as we do in Decision Coverage.
- If the decision contains two arguments with logical operator `and, and then, or, or else,` or `xor` we need exactly three test cases:
  - TT, TF, FT for `and`
  - TT, TF, one of FT, FF for `and then`
  - FF, FT, TF for `or`
  - FF, FT, one of TF, TT for `or else`
  - three of TT, TF, FT, FF for `xor`

  where T means true and F means false.
- If the decision contains more arguments, then we use the following algorithm:
  1. Transform the AST that belongs to the decision to contain information about the precedence of logical operators. (The AST, which generated by [19] is a bit different.)
  2. Generate all the possible combinations of values for the arguments. (2^n combinations, where `n` is the number of arguments.) These are the potential test cases.

3. Eliminate the masked test cases. For example let us consider `A and B`, where `B` is `false`. In this occasion the whole logical expression is `false` and independent of `A`. But `A` is not necessarily a logical variable. It can be another logical expression too and in this case the outcome value of `A` does not affect the whole logical expression. Therefore this test case is masked for `A` and it can be eliminated (for `A`). You can find a more detailed description and examples in [11] about this step.
4. For every logical operator in the decision: we collect the non-masked test cases which satisfy one of its requirements. So we get a set of test cases for every requirement of every logical operator. If one of these sets is empty the decision cannot be covered 100% by MC/DC. If this happens we try to achieve the highest possible coverage.
5. Calculate the minimal covering set of these sets. We do it in the following way: let us suppose we have n arguments in a decision. The maximum number of test cases is $m = 2n$ and we number them $0..m-1$. Of course almost all will be masked. Let us suppose all of the logical operators has two arguments (neither of them are `not`), so we have $s = 3 * (n-1)$ sets. We calculate the minimal covering set by Integer Programming, where for every s_i set we have a disparity which is:

$$\sum_{k=0}^{m-1} \chi_{k \in S_i} X_k > 1$$

Where χ_α is 1 if α is true and 0 otherwise.
Our target function is:

$$min \sum_{k=0}^{m-1} X_k$$

With constraint: the value of every x_k can be only 0 or 1. When the result is calculated we get the minimal covering set. Every test case indexed with k is a member of the minimal covering set if x_k is 1.
To do that calculation we used Lemon graph library [5] with glpk linear programming kit [6].

`Analyzing decisions together`
Like in DC one test case can test several decisions when they are in same level, and one decision may require more test cases when it has nested decisions. But the way to calculate this is a bit more difficult because we have to deal with conditions in a decision. Here is an example about the problem of decisions in same level:

```
if a and b then
    --...
end if;
--...
if c or d then
```

```
    --...
end if;
```

There are three test cases necessary to satisfy the requirements of both of these decisions. The test cases for the first decision are: TT, TF, FT, and for the second decision are: FF, TF, FT. So three test cases can exercise both of the decisions simultaneously, because their conditions are independent. But let us see what happens if we change the `c` to `a` in the second decision:

```
if a and b then
    --...
end if;
--...
if a or d then
    --...
end if;
```

Now three test cases are not enough because in the first decision, `a` has to be `true` twice and `false` once. And in the second decision it must be `true` once and `false` twice. So we need four test cases and two of them have to evaluate `a` as `true` and two others as `false`.

The method to calculate how many test cases are needed for decisions standing in same level:
Decision 1 has n variables: $a_1, ..., a_n$
Decision 2 has m variables: $b_1, ..., b_m$
The first s variables are the common variables where $s \leq min(n, m)$
Our algorithm works with k variables $c_1, ..., c_k$ where $k = n + m - s$
$c_i.true$ means the number of test cases where the variable c_i evaluated as $true$.
$c_i.false$ means the number of test cases where the variable c_i evaluated as $false$.
Let us consider:

$$c_i.true = \begin{cases} max(a_i.true, b_i.true) & \text{if } i = 1..s \\ a_i.true & \text{if } i = s+1..n \\ b_{i-n}.true & \text{if } i = n+1..n+m-s \end{cases}$$

$$c_i.false = \begin{cases} max(a_i.false, b_i.false) & \text{if } i = 1..s \\ a_i.false & \text{if } i = s+1..n \\ b_{i-n}.false & \text{if } i = n+1..n+m-s \end{cases}$$

If there are more than two decisions, we start the algorithm again with $c_1, ..., c_k$, and the variables of the next decision, and repeat it until all the decisions are processed.
Number of test cases:

$$max_{i=1..k}(c_i.true + c_i.false)$$

We deal with the nested decisions in the following way. Let us see an example:

```
if a or b then //first decision
    if c and d then  //second (nested) decision
        -- ...
    end if
end if
```

There are three test cases that are needed for both decisions: TF, FT, FF for the first and TT, TF, FT for the second. The variables are independent, so we can test them simultaneously. But in the third case the first decision is false, therefore the second decision cannot be executed. So we need an extra test case – where the first decision is true – to exercise the third requirements of the nested decision.

In general we calculate the maximum number of test cases that are needed to exercise the requirement of true and false consequences of decisions (m_{true}, m_{false} are the corresponding values). Then we get the set of test cases which cover the decision. We calculate how many of them are evaluated as `true` and how many are `false`. (The corresponding values are: d_{true}, d_{false}.) Then the number of necessary test cases are:

$$max(m_{true}, d_{true}) + max(m_{false}, d_{false})$$

We always consider the variables in nested decisions independent from the variables of outer decisions. Our future work is to refine this method to deal with the same variables.

4 Measurement and results

We analyzed twelve projects written in Ada programming language. Six of them were provided by an industrial company and the rest were open source projects downloaded from `sourceforge.net`. In every project about half of the subprograms have no decisions. These are the initialiser, getter and setter subprograms. About twenty five per cent of the subprograms have only one argument in their decisions. We used only those files which contain at least one subprogram definition, not only declarations. Let us see the overall details:

Number of files:	3549
Effective lines of code:	888432
Number of subprograms:	23892
Nr. of subprog. without decision:	13439
Nr. of subprog. with exactly 1 argument in their decisions:	8190
Nr. of subprog. with more arguments in their decisions:	2263

The table 1 shows the distribution of decisions by their argument numbers.

Number of arguments:	1	2	3	4	5	6	7	8	9
Number of decisions:	51542	3466	626	289	111	99	32	37	20

Number of arguments:	10	11	12	13	14	15	16	18	22	23	34
Number of decisions:	18	14	13	9	4	4	1	1	1	4	1

Table 1.

4.1 Differences and the McCabe metric

In this chapter we can see how the McCabe metric values affect the difference between the necessary test cases for DC and MC/DC. We grouped the subprograms of the projects by their McCabe values. The first table shows those subprograms where the McCabe values are between 0 and 10, the second shows those where the McCabe values are between 11 and 20 etc. Every rows of the tables refer to an individual project and the last row contains the summary. The `Nr.` column holds the number of subprograms in the group. The `DC` and `MC/DC` columns mean how many test cases are needed to cover all the subprograms in the group for DC and MC/DC. The `difference` column contains the difference of `DC` and `MC/DC` columns and the `Ratio` column means how many times more test cases are needed to cover MC/DC than DC.

	Nr.	DC	MC/DC	Difference	Ratio
1.	1533	2361	2517	156	1.07
2.	1212	2845	3113	259	1.09
3.	5498	9220	9730	510	1.06
4.	1746	3314	3505	191	1.06
5.	5792	9801	10523	722	1.07
6.	5690	9972	10636	664	1.07
7.	91	171	189	18	1.11
8.	7	8	8	0	1.00
9.	299	478	505	27	1.06
10.	451	701	751	50	1.07
11.	112	174	177	3	1.02
12.	61	86	95	9	1.10
$\sum$	22327	39140	41750	2610	1.07

Table 2. McCabe values are between 0 and 10

	Nr.	DC	MC/DC	Difference	Ratio
1.	85	791	825	34	1.04
2.	56	475	537	62	1.13
3.	177	1542	1634	92	1.06
4.	72	705	729	21	1.03
5.	244	1499	1678	179	1.12
6.	289	1919	2138	219	1.11
7.	4	18	19	1	1.05
8.	2	15	15	0	1.00
9.	10	61	66	5	1.08
10.	1	2	2	0	1.00
11.	5	20	21	1	1.05
12.	0	-	-	-	-
$\sum$	945	7047	7664	617	1.09

Table 3. McCabe values are between 11 and 20

	Nr.	DC	MC/DC	Difference	Ratio
1.	34	579	587	8	1.01
2.	13	215	220	5	1.02
3.	55	729	749	20	1.03
4.	12	166	178	12	1.07
5.	106	820	907	87	1.11
6.	74	701	791	90	1.13
7.	0	-	-	-	-
8.	0	-	-	-	-
9.	3	46	46	0	1.00
10.	1	4	4	0	1.00
11.	0	-	-	-	-
12.	0	-	-	-	-
$\sum$	298	3260	3482	222	1.07

Table 4. McCabe values are between 21 and 30

	Nr.	DC	MC/DC	Difference	Ratio
1.	13	295	300	5	1.02
2.	4	113	117	4	1.03
3.	27	394	409	15	1.04
4.	11	207	210	3	1.01
5.	31	256	302	46	1.18
6.	55	638	778	138	1.22
7.	0	-	-	-	-
8.	0	-	-	-	-
9.	-	-	-	-	-
10.	1	20	20	0	1.00
11.	0	-	-	-	-
12.	0	-	-	-	-
$\sum$	142	1923	2134	211	1.11

Table 5. McCabe values are between 31 and 40

	Nr.	DC	MC/DC	Difference	Ratio
1.	13	423	453	30	1.07
2.	1	37	44	7	1.19
3.	20	726	770	44	1.06
4.	6	286	286	0	1.00
5.	70	2015	2275	260	1.13
6.	68	1942	2083	141	1.07
7.	0	-	-	-	-
8.	1	47	47	0	1.00
9.	1	5	5	0	1.00
10.	0	-	-	-	-
11.	0	-	-	-	-
12.	0	-	-	-	-
$\sum$	180	5481	5963	482	1.09

Table 6. McCabe values are above than 40

Since the McCabe deals with the number of decisions and not their structure or their argument numbers, we can say the difference between the necessary test cases for DC and MC/DC do not depend on the McCabe metric. We accept this result, because the McCabe value of a subprogram can be high even if there is only one argument in decisions. In that way the DC and MC/DC values are same. And on the other hand the McCabe value is low when there are few decisions in a subprogram even if they have many arguments. In such way the DC and MC/DC values can highly diverge.

4.2 Differences and the Nesting

In this chapter we grouped the subprograms by the deepness of the nested structures. The orientation of the tables are the same as in the previous chapter.

The maximum nesting is between 0 and 1

	Nr.	DC	MC/DC	Difference	Ratio
1.	1360	2498	2564	66	1.03
2.	912	1696	1831	135	1.08
3.	4331	6304	6552	248	1.04
4.	1341	2055	2133	78	1.04
5.	4885	8231	8748	517	1.06
6.	4430	7507	7883	376	1.05
7.	63	93	97	4	1.04
8.	7	18	18	0	1.00
9.	240	317	331	14	1.04
10.	353	429	456	27	1.06
11.	95	130	132	2	1.02
12.	51	65	71	6	1.09
$\sum$	17841	27091	28263	1171	1.04

Table 7. The maximum nesting is between 0 and 1

	Nr.	DC	MC/DC	Difference	Ratio
1.	234	1058	1125	67	1.06
2.	298	1436	1589	153	1.11
3.	948	3613	3840	227	1.06
4.	366	1358	1449	91	1.06
5.	1058	3782	4173	391	1.10
6.	1145	3903	4223	320	1.08
7.	29	84	99	15	1.18
8.	2	5	5	0	1.00
9.	58	161	175	14	1.09
10.	85	230	251	21	1.09
11.	18	50	51	1	1.02
12.	9	19	22	3	1.16
$\sum$	4250	15716	17002	1286	1.08

Table 8. The maximum nesting is between 2 and 3

	Nr.	DC	MC/DC	Difference	Ratio
1.	76	804	895	91	1.11
2.	74	555	604	49	1.09
3.	262	2027	2189	162	1.08
4.	121	910	971	61	1.07
5.	262	1896	2165	269	1.14
6.	493	2960	3387	427	1.14
7.	3	12	12	0	1.00
8.	1	47	47	0	1.00
9.	12	87	91	4	1.05
10.	16	68	69	1	1.01
11.	4	14	15	1	1.07
12.	1	2	2	0	1.00
$\sum$	1325	9382	10447	1065	1.11

Table 9. The maximum nesting is between 4 and 6

	Nr.	DC	MC/DC	Difference	Ratio
1.	8	89	98	7	1.10
2.	2	7	7	0	1.00
3.	36	667	711	44	1.07
4.	19	338	355	17	1.05
5.	38	482	599	117	1.24
6.	108	802	933	131	1.16
7.	0	-	-	-	-
8.	0	-	-	-	-
9.	3	25	25	0	1.00
10.	0	-	-	-	-
11.	0	-	-	-	-
12.	0	-	-	-	-
$\sum$	214	2410	2726	316	1.13

Table 10. The maximum nesting is above than 6

An increase in the maximum nesting value causes the increase of the ratio very slightly, thus it does not affect the difference of necessary test cases significantly.

4.3 Differences and the Maximum Argument Numbers

Here we can see how the largest decision (which contains the most arguments) affects the difference in the number of necessary test cases for DC and MC/DC. We grouped the subprograms by the number of arguments of the largest decisions. The orientation of the tables are the same as in the previous chapters.

	Nr.	DC	MC/DC	Difference	Ratio
1.	1134	1134	1134	0	1.00
2.	586	586	586	0	1.00
3.	3299	3299	3299	0	1.00
4.	996	996	996	0	1.00
5.	3469	3469	3469	0	1.00
6.	3343	3343	3343	0	1.00
7.	31	31	31	0	1.00
8.	6	6	6	0	1.00
9.	160	160	160	0	1.00
10.	291	291	291	0	1.00
11.	82	82	82	0	1.00
12.	42	42	42	0	1.00
$\sum$	13439	13439	13439	0	1.00

Table 11. The maximum argument number is 0 (no decisions)

	Nr.	DC	MC/DC	Difference	Ratio
1.	389	2308	2308	0	1.00
2.	493	1957	1957	0	1.00
3.	1817	6475	6475	0	1.00
4.	656	2466	2466	0	1.00
5.	2324	7840	7840	0	1.00
6.	2130	7304	7304	0	1.00
7.	53	122	122	0	1.00
8.	3	17	17	0	1.00
9.	127	309	309	0	1.00
10.	131	322	322	0	1.00
11.	27	83	83	0	1.00
12.	10	25	25	0	1.00
$\sum$	8190	29229	29229	0	1.00

Table 12. The maximum argument number is 1

	Nr.	DC	MC/DC	Difference	Ratio
1.	134	873	1031	158	1.18
2.	166	900	1081	181	1.20
3.	416	2396	2866	470	1.19
4.	179	1072	1249	177	1.17
5.	307	2106	2539	433	1.21
6.	571	3504	4223	719	1.21
7.	11	36	54	18	1.50
8.	1	47	47	0	1.00
9.	23	94	113	19	1.20
10.	27	85	113	28	1.33
11.	8	29	33	4	1.14
12.	9	19	28	9	1.47
$\sum$	1857	11288	13377	2089	1.19

Table 13. The maximum argument number is between 2 and 3

	Nr.	DC	MC/DC	Difference	Ratio
1.	3	8	25	17	3.125
2.	25	156	236	80	1.51
3.	33	362	484	122	1.34
4.	10	117	135	18	1.15
5.	69	401	642	241	1.60
6.	74	471	673	202	1.43
7.	0	-	-	-	-
8.	0	-	-	-	-
9.	3	34	40	6	1.17
10.	4	29	43	14	1.48
11.	0	-	-	-	-
12.	0	-	-	-	-
$\sum$	236	1696	2437	741	1.44

Table 14. The maximum argument number is between 4 and 5

	Nr.	DC	MC/DC	Difference	Ratio
1.	0	-	-	-	-
2.	11	65	110	45	1.69
3.	6	26	66	40	2.54
4.	5	18	46	28	2.56
5.	52	359	718	359	2.00
6.	44	399	630	231	1.58
7.	0	-	-	-	-
8.	0	-	-	-	-
9.	0	-	-	-	-
10.	1	2	7	5	3.50
11.	0	-	-	-	-
12.	0	-	-	-	-
$\sum$	122	877	1602	725	1.83

Table 15. The maximum argument number is between 6 and 10

	Nr.	DC	MC/DC	Difference	Ratio
1.	0	-	-	-	-
2.	5	30	61	31	2.03
3.	6	53	102	49	1.92
4.	1	9	16	7	1.78
5.	22	216	477	216	2.21
6.	14	151	253	102	1.68
7.	0	-	-	-	-
8.	0	-	-	-	-
9.	0	-	-	-	-
10.	0	-	-	-	-
11.	0	-	-	-	-
12.	0	-	-	-	-
$\sum$	48	459	909	450	1.98

Table 16. The maximum argument number is above than 10

In the first two cases there is no difference between DC and MC/DC. It comes from the definition of these testing methods. As the number of arguments increases in decisions, the difference is increasing as well. Decisions with more than ten arguments in subprograms require almost twice as many test cases for MC/DC than DC.

4.4 Differences Overall

In this chapter the differences can be seen for the whole projects separately and in the last row together. The `DC` and `MC/DC` columns mean how many test cases are needed to cover all the subprograms in the projects for DC and MC/DC. The `Diff` and the `Rat` columns contain the difference and the quotient of `DC` and `MC/DC` columns. `A` means the all subprograms of the projects. In `B`, we excluded those subprograms which have no decisions at all. And in `C`, we excluded those subprograms that either have no decisions or there are no decisions with more than one argument.

	A				B				C			
	DC	MC/DC	Diff	Rat	DC	MC/DC	Diff	Rat	DC	MC/DC	Diff	Rat
1.	4449	4682	233	1.05	3315	3548	233	1.07	1007	1240	233	1.23
2.	3694	4031	337	1.09	3108	3445	337	1.11	1151	1488	337	1.29
3.	12611	13292	681	1.05	9312	9993	681	1.07	2837	3518	681	1.24
4.	4678	4908	230	1.05	3682	3912	230	1.06	1216	1446	230	1.19
5.	14391	15685	1294	1.08	10922	12216	1294	1.12	3082	4376	1294	1.42
6.	15172	16426	1254	1.08	11829	13083	1254	1.11	4554	5779	1254	1.27
7.	189	208	19	1.10	158	177	19	1.12	36	55	19	1.53
8.	70	70	0	1.00	64	64	0	1.00	47	47	0	1.00
9.	590	622	32	1.05	430	462	32	1.07	128	160	32	1.25
10.	727	776	49	1.07	436	485	49	1.11	116	165	49	1.42
11.	194	198	4	1.02	112	116	4	1.04	29	33	4	1.14
12.	86	95	9	1.10	44	53	9	1.20	19	28	9	1.47
$\sum$	56851	60993	4142	1.07	43412	47554	4142	1.10	14320	18432	4142	1.29

Table 17. Summary

5 Conclusion and future work

We analyzed several projects written in Ada programming language and estimated the difference of the required test cases of Decision Coverage and the more strict Modified Condition / Decision Coverage. We found that the difference is about five to ten per cent because the decisions in most subprograms have only one argument and there are several subprograms which do not contain decisions at all. If we exclude these subprograms we get a difference that is four times larger. Most importantly, the maximum number of arguments in decisions affects the difference. For those subprograms where there are decisions with more than six arguments, almost twice as many MC/DC test cases are needed as DC. But these subprograms are only less than one per cent of the whole project.

In general we can say five to ten per cent more test cases are needed to satisfy the requirements of MC/DC then DC.

Our future work is to refine our analyzer program to do a better estimation in some exceptional cases, and we plan to do this analysis more other projects. We

would also consider exceptions and exception handling as a branching statement and extend our survey to manage these cases.

References

1. W. R. Adrion, Ma. A. Branstad, J. C. Cherniavsky: Validation, Verification, and Testing of Computer Software, ACM Computing Surveys (CSUR), v.14 n.2, pp. 159-192, June 1982.
2. P. Amman, J. Offutt, H. Huang: Coverage Criteria for Logical Expressions, In proc. of 14th International Symposium on Software Reliability Engineering, pp. 99.
3. B. Beizer: Software testing techniques (2nd ed.), Van Nostrand Reinhold Co., New York, NY, 1990
4. J. C. Cherniavsky: On finding test data sets for loop free programs, Inform. Process. Lett 8, 2 (1979).
5. J. J. Chilenski and S. P. Miller, Applicability of Modified Condition/Decision Coverage to Software Testing, Software Engineering Journal, Volume 9, Issue 5, September 1994, pp. 193-200.
6. J. J. Chilenski and P. H. Newcomb, Formal Specification Tools for Test Coverage Analysis, Proceedings of the Ninth Knowledge-Based Software Engineering Conference (KBSE'94), Monterey, CA, USA, September 20-23 1994, pp. 59-68.
7. R. A. Demillo, R. J. Lipton, F. G. Sayward: Hints on test data selection: Help for the practicing programmer, Computer 11, 4 (1978), pp. 34–43.
8. A. Dupuy, N. Leveson: An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software, In Proc. of 19th Digital Avionics Systems Conferences, 2000, pp. 1B6/1-1B6/7 vol.1.
9. T. Gyimóthy, R. Ferenc and I. Siket: Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. In IEEE Transactions on Software Engineering
10. K. J. Hayhurst, D. S. Veerhusen: A Practical Approach to Modified Condition/Decision Coverage, 20th Digital Avionics Systems Conference (DASC), Daytona Beach, Florida, USA, October 14-18, 2001, Vol. 1, pp. 1B2/1-1B2/10.
11. K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, L. K. Rierson: A Practical Tutorial on Modified Condition/Decision Coverage (NASA / TM-2001-210876, technical report).
12. T. J. McCabe: A Complexity Measure, IEEE Trans. Software Engineering, SE-2(4), pp. 308–320, 1976
13. G. J. Myers: The Art of Software Testing. John Wiley, 1986.
14. S. Rayadurgam, M.P.E. Heimdahl: Coverage based test-case generation using model checkers, Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop, pp. 83-91.
15. E. J. Weyucker, T. J. Ostrand: Theories of program testing and the application of revealing subdomains, IEEE Trans. Sofiw. Eng. SE- 6 (May, 1980), pp. 236–246.
16. A. L. White: Comments on Modified Condition/Decision Coverage for Software Testing, 2001 IEEE Aerospace Conference Proceedings, 10-17 March 2001, Big Sky, Montana, USA, volume 6, pp. 2821-2828.
17. H. Zhu, P. A. V. Hall, J. H. R. May: Software unit test coverage and adequacy, ACM Computing Surveys, vol. 29, issue 4, pp. 366–427
18. S. Cornett: Code Coverage Analysis `http://www.bullseye.com/coverage.html`
19. `http://www.antlr.org/`

20. O. Kellogg: http://www.antlr.org/grammar/ada
21. https://lemon.cs.elte.hu/site/
22. http://www.gnu.org/software/glpk/