periodica polytechnica

Electrical Engineering
52/3-4 (2008) 187–195
doi: 10.3311/pp.ee.2008-3-4.07
web: http://www.pp.bme.hu/ee

RESEARCH ARTICLE

Necessary test cases for Decision Coverage and Modified Condition / Decision Coverage

Zalán Szűgyi / *Zoltán* Porkoláb

Received 2008-07-22

Abstract

Test coverage refers to the extent to which a given software verification activity satisfies its objectives. Several types of coverage analysis exist to check code correctness. Less strict analysis methods require fewer test cases to satisfy their requirements and consume less resources. Choosing test methods is a compromise between the code correctness and the available resources. However this selection should be based on quantitative consideration. In this paper we concern 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: Mc- Cabe metric, nesting and maximal argument number in decisions. We discuss how these aspects are affected by difference of the necessary test cases for these testing methods.

Keywords

Testing · DC · MC/DC · Ada

Acknowledgement

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

Zalán Szűgyi

Department of Programming Languages and Compilers, ELTE,, Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
e-mail: lupin, gsd@elte.hu

Zoltán Porkoláb

Department of Programming Languages and Compilers, ELTE, Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
e-mail: lupin, gsd@elte.hu

1 Introduction

Coverage refers to the extent to which a given verification activity satisfies its objectives. Coverage measures can be applied to any verification activities, 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 [18] of the verification accomplished [13].

The code coverage analysis contains three main steps [19], 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.

Code coverage analysis is a structural testing technique (white box testing), where the test program behavior is compared against the apparent intention of the source code. Different types of analysis require different sets of test cases. We concern Decision Coverage (DC), and Modified Condition / Decision Coverage (MC/DC) testing methods. DC only requires that every line of code in a subprogram must be executed and every decisions must be evaluated both to true and false. MC/DC is more strict as it should fulfil the requirements of DC and additionally, it demands to show that every condition in a decision independently affects the outcome. It is clear that more test cases are needed to satisfy the requirements of MC/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 wre written in Ada programming language and we analyzed them in several aspects: McCabe metrics [14], nesting, and maximal argument number in decisions. We examined how these aspects affected the difference of the necessary test cases.

In Chapter 2 we describe the most frequently used coverage metrics. In Chapter 3 we give a detailed description about how we analyzed the source codes of projects. Then we discuss the results of our analysis in Chapter 4. We give an overview of related works in Chapter 5. Summary and conclusion are given in chapter 6.

2 Coverage metrics

In this chapter we describe some commonly used coverage metrics.

2.1 Statement coverage

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

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

In this example a single 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, which might cause non-deterministic behavior or segmentation fault.

2.2 Decision coverage

This method requires that each statement must be invoked at least once and each decision must be evaluated both to true and false. In this case the error mentioned in the previous example is discovered in testing time. This metric has the advantage of simplicity without the problems of statement coverage. A disadvantage is that it ignores branches within Boolean expressions which occur due to short-circuit operators. Let us see to following example:

```
if A or else 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 to true and 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 = false, B = false and A = true, B = false and A = false, B = true) provide MC/DC. The MC/DC is derived from by Condition / Decision Coverage testing method. More information on these coverage methods can be found in [13], [12], [19]. MC/DC is used for various environments, from test generation [16] 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 [20] parser generator with [21] grammar file to create the Abstract Syntax Tree (AST) of the source code. Our analysis uses this AST.

3.1 Counting test cases for decision coverage

The Decision Coverage requires that every decision must be evaluated to true and false at least once. So we need at least two test cases for every decision to satisfy these requirements. But one test case can cover several decisions and more than 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). Condition_1 must be evaluated to true twice because it has a nested decision in its true part. 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 $max(A_i + B_i)$ where A_i, B_i belongs to the ith decision ($i = 1 \ldots$ number of decisions).

If there are identical decisions on the same level we classify them. The identical decisions will be placed into the same class. Then we consider $max(A_j + B_j)$ 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 consider as number of arguments plus one test cases are needed. This approximate value comes from [13].

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, generated by [20] 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 case 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 [13] 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 fully covered 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 $m = 2^n$ and we number them $0..m-1$. Of course almost all will be masked. Let us suppose that all the logical operators have two arguments (neither of them are `not`), so we have $s = 3 \cdot (n-1)$ sets. We calculate the minimal covering set by Integer Programming, where for every s_i set we have a disparity which is:

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

  Our target function is:

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

  In wich the value of each X_k is either 0 or 1. When the result is calculated we get the minimal covering set. Each 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;
...
```

Three test cases are 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 arguments: $a_1, ..., a_n$
Decision 2 has m arguments: $b_1, ..., b_m$
The first s arguments are the common arguments where $s \leq \min(n, m)$
Our algorithm works with k arguments $c_1, ..., c_k$ where $k = n + m - s$
c_i.true means the number of test cases where the argument c_i evaluated to $true$.
c_i.false means the number of test cases where the argument c_i evaluated to false.
Let us consider:

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

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

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

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

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 arguments 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 requirement 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. Values d_{true}, d_{false} denote the number of test cases, which have been evaluated to true and false, respectively. Then the number of necessary test cases are:

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

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

4 Measurement and results

We analyzed twelve projects written in Ada programming language. The sources contained both Ada 83 and Ada 95 language versions. Six of the projects originated in an industrial company and related to control systems and device drivers. The rest of the projects were open source applications in various fields [24–29], and were downloaded from `sourceforge.net`. In every project about fifty per cent 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

Table 1 shows the distribution of decisions by their argument numbers.

Tab. 1.

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

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 table 2 shows those subprograms where the McCabe values are between 0 and 10, the table 3 shows those where the McCabe values are between 11 and 20 etc. Each

row of the tables refers 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.

Tab. 2. McCabe values are between 0 and 10

	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 [24].	91	171	189	18	1.11
8 [25].	7	8	8	0	1.00
9 [26].	299	478	505	27	1.06
10 [27].	451	701	751	50	1.07
11 [28].	112	174	177	3	1.02
12 [29].	61	86	95	9	1.10
$\sum$	22327	39140	41750	2610	1.07

Tab. 3. McCabe values are between 11 and 20

	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 [24]	4	18	19	1	1.05
8 [25].	2	15	15	0	1.00
9 [26].	10	61	66	5	1.08
10 [27].	1	2	2	0	1.00
11 [28].	5	20	21	1	1.05
12 [29].	0	-	-	-	-
$\sum$	945	7047	7664	617	1.09

Since the McCabe metric 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 does 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 the 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 that way there can be a big difference between the DC and MC/DC values.

The Fig. 1 shows the ratio of the number of necessary test cases for MC/DC and DC. The colums correspond to the proper table of this chapter.

Tab. 4. McCabe values are between 21 and 30

	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 [24]	0	-	-	-	-
8 [25].	0	-	-	-	-
9 [26].	3	46	46	0	1.00
10 [27].	1	4	4	0	1.00
11 [28].	0	-	-	-	-
12 [29].	0	-	-	-	-
$\sum$	298	3260	3482	222	1.07

Tab. 5. McCabe values are between 31 and 40

	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 [24]	0	-	-	-	-
8 [25].	0	-	-	-	-
9 [26].	-	-	-	-	-
10 [27].	1	20	20	0	1.00
11 [28].	0	-	-	-	-
12 [29].	0	-	-	-	-
$\sum$	142	1923	2134	211	1.11

Tab. 6. McCabe values are above 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 [24]	0	-	-	-	-
8 [25].	1	47	47	0	1.00
9 [26].	1	5	5	0	1.00
10 [27].	0	-	-	-	-
11 [28].	0	-	-	-	-
12 [29].	0	-	-	-	-
$\sum$	180	5481	5963	482	1.09

4.2 Differences and the nesting

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

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.

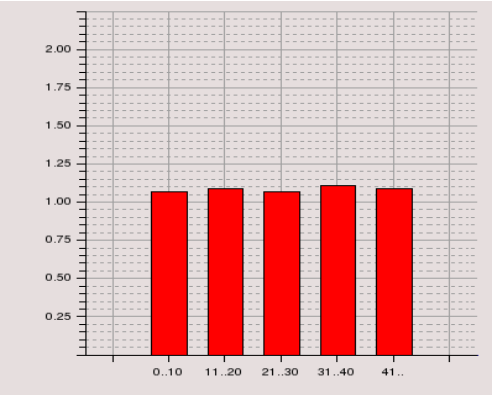

Fig. 1. McCabe

Tab. 7. 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 [24].	63	93	97	4	1.04
8 [25].	7	18	18	0	1.00
9 [26].	240	317	331	14	1.04
10 [27].	353	429	456	27	1.06
11 [28].	95	130	132	2	1.02
12 [29].	51	65	71	6	1.09
∑	17841	27091	28263	1171	1.04

Tab. 8. The maximum nesting is between 2 and 3

	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 [24].	29	84	99	15	1.18
8 [25].	2	5	5	0	1.00
9 [26].	58	161	175	14	1.09
10 [27].	85	230	251	21	1.09
11 [28].	18	50	51	1	1.02
12 [29].	9	19	22	3	1.16
∑	4250	15716	17002	1286	1.08

Tab. 9. The maximum nesting is between 4 and 6

	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 [24].	3	12	12	0	1.00
8 [25].	1	47	47	0	1.00
9 [26].	12	87	91	4	1.05
10 [27].	16	68	69	1	1.01
11 [28].	4	14	15	1	1.07
12 [29].	1	2	2	0	1.00
∑	1325	9382	10447	1065	1.11

Tab. 10. The maximum nesting is above 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 [24]	0	-	-	-	-
8 [25].	0	-	-	-	-
9 [26].	3	25	25	0	1.00
10 [27].	0	-	-	-	-
11 [28].	0	-	-	-	-
12 [29].	0	-	-	-	-
∑	214	2410	2726	316	1.13

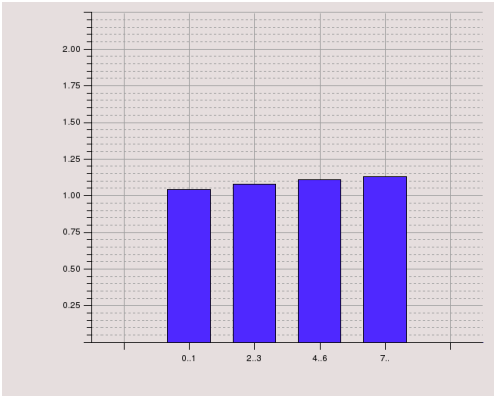

Fig. 2. Nesting

The orientation of Fig. 2 is the same as in previous chapters.

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. 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 for DC.

The orientation of Fig. 3is the same as in previous chapters.

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

	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 [24].	31	31	31	0	1.00
8 [25].	6	6	6	0	1.00
9 [26].	160	160	160	0	1.00
10 [27].	291	291	291	0	1.00
11 [28].	82	82	82	0	1.00
12 [29].	42	42	42	0	1.00
$\sum$	13439	13439	13439	0	1.00

Tab. 12. The maximum argument number is 1

	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 [24].	53	122	122	0	1.00
8 [25].	3	17	17	0	1.00
9 [26].	127	309	309	7	1.00
10 [27].	131	322	322	0	1.00
11 [28].	27	83	83	0	1.00
12 [29].	10	25	25	0	1.00
$\sum$	8190	29229	29229	0	1.00

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

	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 [24].	11	36	54	18	1.50
8 [25].	1	47	47	0	1.00
9 [26].	23	94	113	19	1.20
10 [27].	27	85	113	28	1.33
11 [28].	8	29	33	4	1.14
12 [29].	9	19	28	9	1.47
$\sum$	1857	11288	13377	2089	1.19

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

	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 [24].	0	-	-	-	-
8 [25].	0	-	-	-	-
9 [26].	3	34	40	6	1.17
10 [27].	4	29	43	14	1.48
11 [28].	0	-	-	-	-
12 [29].	0	-	-	-	-
$\sum$	236	1696	2437	741	1.44

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

	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 [24].	0	-	-	-	-
8 [25].	0	-	-	-	-
9 [26].	0	-	-	-	-
10 [27].	1	2	7	5	3.50
11 [28].	0	-	-	-	-
12 [29].	0	-	-	-	-
$\sum$	122	877	1602	725	1.83

Tab. 16. The maximum argument number is above 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 [24].	0	-	-	-	-
8 [25].	0	-	-	-	-
9 [26].	0	-	-	-	-
10 [27].	0	-	-	-	-
11 [28].	0	-	-	-	-
12 [29].	0	-	-	-	-
$\sum$	48	459	909	450	1.98

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. Columns Diff and Rat contain the difference and the ratio of the values in columns DC and MC/DC respectively. There are three tables: In Table 17 all subprograms of the projects appear. In Table 18, we excluded those subprograms which have no decisions at all. And in Table 19, we excluded those subprograms that either have no decisions or there are no decisions with more than one argument.

The Fig. 4 shows the ratio of compulsory test cases of DC and MC/DC testing method. The column A represents the sum-

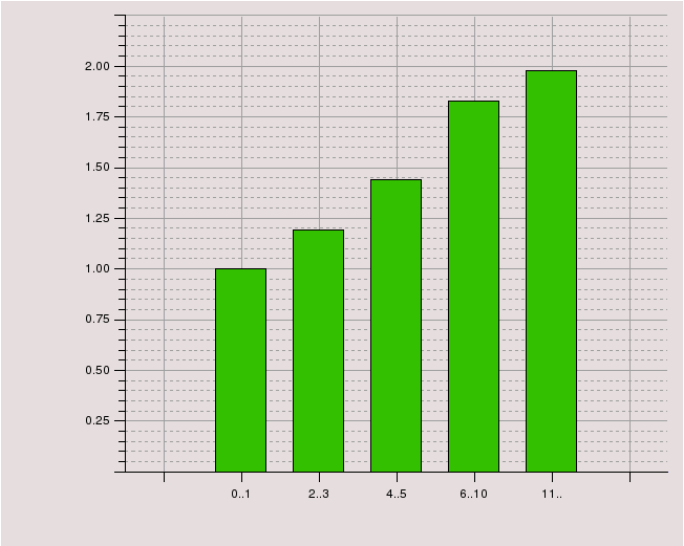

Fig. 3. Maximu argument numbers

Tab. 17. All subprograms of the projects

	DC	MC/DC	Diff	Rat
1.	4449	4682	233	1.05
2.	3694	4031	337	1.09
3.	12611	13292	681	1.05
4.	4678	4908	230	1.05
5.	14391	15685	1294	1.08
6.	15172	16426	1254	1.08
7 [24].	189	208	19	1.10
8 [25].	70	70	0	1.00
9 [26].	590	622	32	1.05
10 [27].	727	776	49	1.07
11 [28].	194	198	4	1.02
12 [29].	86	95	9	1.10
$\sum$	56851	60993	4142	1.07

Tab. 18. Subprograms containing decisions

	DC	MC/DC	Diff	Rat
1.	3315	3548	233	1.07
2.	3108	3445	337	1.11
3.	9312	9993	681	1.07
4.	3682	3912	230	1.06
5.	10922	12216	1294	1.12
6.	11829	13083	1254	1.11
7 [24].	158	177	19	1.12
8 [25].	64	64	0	1.00
9 [26].	430	462	32	1.07
10 [27].	436	485	49	1.11
11 [28].	112	116	4	1.04
12 [29].	44	53	9	1.20
$\sum$	43412	47554	4142	1.10

mary of the Table 17, the column B belongs to Table 18 and the column C to Table 19.

Tab. 19. All subprograms of the projects

	DC	MC/DC	Diff	Rat
1.	1007	1240	233	1.23
2.	1151	1488	337	1.29
3.	2837	3518	681	1.24
4.	1216	1446	230	1.19
5.	3082	4376	1294	1.42
6.	4554	5779	1254	1.27
7 [24].	36	55	19	1.53
8 [25].	47	47	0	1.00
9 [26].	128	160	32	1.25
10 [27].	116	165	49	1.42
11 [28].	29	33	4	1.14
12 [29].	19	28	9	1.47
$\sum$	14320	18432	4142	1.29

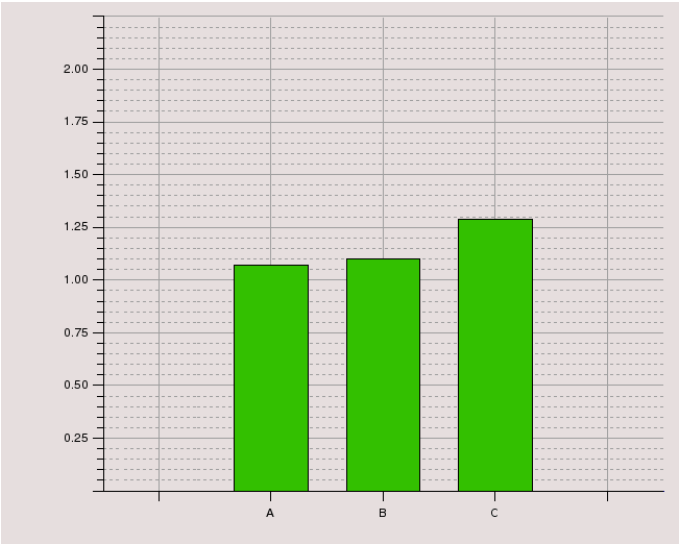

Fig. 4. Overall

5 Related works

Chilenski at al. [6] gave a comprehensive analysis of analysing structural coverage criteria. They provided formal techniques for axiomatizing Ada programs and translating the path expressions of subprogram bodies into conjunctive normal form. Coverage specifications were combined with feasible path expressions to construct a minimal set of test specifications. However, these methods proved effective only for small programs written in a restricted subset of the Ada language (cca 5000 effective lines of code).

Goldberg et al. [10] describes the design and prototype implementation of a structural testing system that uses a theorem prover to determine feasibility of testing requirements and to optimize the number of test cases required to achieve test coverage. Using this approach, the authors were able to determine path feasibility for moderately-sized program units of production code written in a subset of Ada. On these problems, the authors argued that computer solutions were obtained much faster and with greater accuracy than manual analysis.

Gotlieb et al. [11] introduced constraint solving based tech-

niques. After statically transform a procedure into a constraint system by using Static Single Assignment form and control-dependencies they solved the system to check whether at least one feasible control flow path going through the selected point exists and to generate test data that correspond to one of these paths. The key point of their approach is to take advantage of advances in constraint techniques when solving the generated constraint system. The authors developed a prototype implementation on a restricted subset of the C language.

As the examples above show, most of the related works are able to handle only small or medium sized projects, and implemented over resticted language subsets. In our research, however, real-life projects in size up to 220.000 effective lines of code have been measured.

6 Conclusions and future work

We analyzed several projects written in Ada programming language and measured the difference of the required test cases of Decision Coverage and the more strict Modified Condition / Decision Coverage. To reduce development efforts it is essential to find a good balance between minimizing testing costs and find the most of the possible bugs in the code. Choosing the rigth testing methods based on the character of the source could be a useful method. 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 more precise measurement on programs containing special syntactical and semantical features, like exception handling. We also plan to analyse a wider set of projects from different industrial areas.

References

1 **Adrion W R, Branstad M A, Cherniavsky J C**, *Validation, Verification, and Testing of Computer Software*, ACM Computing Surveys (CSUR) **14** (June 1982), no. 2, 159-192, DOI 10.1145/356876.356879.

2 **Amman P, Offutt J, Huang H**, *Coverage Criteria for Logical Expressions*, Proc. of 14th International Symposium on Software Reliability Engineering, pp. 99, DOI 10.1109/ISSRE.2003.1251034, (to appear in print).

3 **Beizer B**, *Software testing techniques*, Van Nostrand Reinhold Co., New York, NY, 1990. 2nd ed.

4 **Cherniavsky J C**, *On finding test data sets for loop free programs*, Inform. Process. Lett **8** (1979), no. 2, DOI 10.1016/0020-0190(79)90155-8.

5 **Chilenski J J, Miller S P**, *Applicability of Modified Condition/Decision Coverage to Software Testing*, Software Engineering Journal **9** (September 1994), 193-200, DOI 10.1049/sej.1994.0025.

6 **Chilenski J J, Newcomb P H**, *Formal Specification Tools for Test Coverage Analysis*, Proceedings of the Ninth Knowledge-Based Software Engineering Conference (KBSE'94), Monterey, CA, USA, 1994, pp. 59-68, DOI 10.1109/KBSE.1994.342677, (to appear in print).

7 **DeMillo R A, Lipton R J, Sayward F G**, *Hints on test data selection: Help for the practicing programmer*, Computer **11** (1978), no. 4, 34–43, DOI 10.1109/C-M.1978.218136.

8 **Dupuy A, Leveson N**, *An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software*, Proc. of 19th Digital Avionics Systems Conferences, 2000, pp. 1B6/1-1B6/7, DOI 10.1109/DASC.2000.886883.

9 **White A L**, *Comments on Modified Condition/Decision Coverage for Software Testing*, 2001 IEEE Aerospace Conference Proceedings, 10-17 March 2001, Big Sky, Montana, USA, pp. 2821-2828.

10 **Goldberg A, Wang T C, Zimmerman D**, *Applications of feasible path analysis to program testing, International Symposium on Software Testing and Analysis*, Proc of 1994 ACM SIGSOFT international symposium on Software testing and analysis, Seattle, Washington, United States, 1994, pp. 80-94, DOI 10.1145/186258.186523, (to appear in print).

11 **Gotlieb A, Botella B, Rueher M**, *Automatic test data generation using constraint solving techniques*, ACM SIGSOFT Software Engineering Notes **23** (1998), 53-62, DOI 10.1145/271775.271790.

12 **Hayhurst K J, Veerhusen D S**, *A Practical Approach to Modified Condition/Decision Coverage*, 20th Digital Avionics Systems Conference (DASC), Daytona Beach, Florida, USA, October 14, 2001, pp. 1B2/1-1B2/10, DOI 10.1109/DASC.2001.963305.

13 **Hayhurst K J, Veerhusen D S, Chilenski J J, Rierson L K**, *A Practical Tutorial on Modified Condition/Decision Coverage (NASA / TM-2001-210876, technical report)*.

14 **McCabe T J**, *A Complexity Measure*, IEEE Trans. Software Engineering, SE-2(4), posted on 1976, 308-320, DOI 10.1109/TSE.1976.233837, (to appear in print).

15 **Myers G J**, *The Art of Software Testing*, John Wiley, 1986.

16 **Rayadurgam S, Heimdahl M P E**, *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, DOI 10.1109/ECBS.2001.922409, (to appear in print).

17 **Weyucker E J, Ostrand T J**, *Theories of program testing and the application of revealing subdomains*, IEEE Trans. Softw. Eng. SE- 6 (May, 1980), pp. 236–246, DOI 10.1109/TSE.1980.234485, (to appear in print).

18 **Zhu H, Hall P A V, May J H R**, *Software unit test coverage and adequacy*, ACM Computing Surveys **29**, 366–427, DOI 10.1145/267580.267590.

19 **Cornett S**, *Code Coverage Analysis*, available at `http://www.bullseye.com/coverage.html`.

20 available at `http://www.antlr.org/`.

21 **O. Kellogg**, available at `http://www.antlr.org/grammar/ada`.

22 available at `https://lemon.cs.elte.hu/site/`.

23 available at `http://www.gnu.org/software/glpk/`.

24 available at `http://sourceforge.net/projects/aptesting`.

25 available at `http://sourceforge.net/projects/gnat-asis/`.

26 available at `http://sourceforge.net/projects/p2ada/`.

27 available at `http://sourceforge.net/projects/libadacrypt-dev/`.

28 available at `http://sourceforge.net/projects/gnat-asis/`.

29 available at `http://sourceforge.net/projects/zlib-ada/`.