COMPARISON OF DC AND MC/DC CODE COVERAGES

Zalán SZŰGYI, Zoltán PORKOLÁB
Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University,
Pázmány Péter sétány 1/C., Hungary, e-mail: lupin,gsd@elte.hu

ABSTRACT

In software development testing plays the most important role to discover bugs and to verify that the product meets its requirements. This paper focuses on programs written in C++ programming language and analyses two important testing methods, the Decision Coverage and the more strict Modified Condition / Decision Coverage. We examine how the program characteristics – lines of code, McCabe metric, the number of the arguments in the decisions, deepness of the nested block – affect the number of necessary test cases of these testing methods. Choosing test methods is always a compromise between the code correctness and the available resources. Less strict ones require fewer test cases and consume less resources, however, they may discover fewer errors. Our results may help to chose between Decision Coverage and Modified Condition / Decision coverage.

Keywords: *DC, MC/DC, test, coverage, C++*

1. INTRODUCTION

Accurate software testing is mandatory in order to find the bugs in a software, and to prove that the code is working properly [16] . Software testing consists of four level of test processes: unit testing, integration testing, system testing, and acceptance testing [18]. Our paper focuses on unit testing which verifies the functionality of the source code in subprogram level. Unit test usually contains a suite of test cases; one test case verifies one property of a subprogram. However, in real projects the subprograms can be complex and usually they need several test cases to fully verify them.

When a test case executes a code snippet of subprogram, we can say that the test case covers it. The code coverage is a measure of the test quality: the higher code coverage provides better tested code. A good test suite aims to achieve the highest possible code coverage on the target source code. The exact definition of the *code coverage* depends on the testing methods. Different security standards define different requirements on tests. The simple ones require only that each statement of the program has to be executed at once. However, more strict requirements specify additional requisites such as all the arguments of the logical expression need to be tested against true and against false.

During the design of the system architecture, it is essential to choose a proper testing method. More strict methods can discover more bugs in a software, however, they require more test cases too, which, after all, increases the cost of the project.

This paper focuses on two important and widely used test methods. The first one is the Decision Coverage (DC), which is a generally used testing method, and requires that every decision in a subprogram must be evaluated both as true and false. However, more secure softwares such as softwares used in aviation require more strict method to test them. For this purpose the Modified Condition / Decision Coverage (MC/DC) method can be a good choice. MC/DC method assesses the requirements of DC, and additionally specifies that every argument of the decision must be shown to independently affect the outcome. Section 2 details these test methods.

The goal of this paper is to support the system designer to choose from the previously mentioned two testing methods. We have compared them by the quantity of the necessary test cases to reach hundred per cent coverage. We have examined how the number of test cases are affected by program characteristics such as McCabe metric, maximum deepness of nested blocks, arity of the most complex decision in a subprogram, etc. The source of our analysis is several middle size open source project, written in C++ programming language.

We found that the difference of the number of required DC and MC/DC test cases is mainly depend on the arity of the decisions. We may need twice more test cases to cover MC/DC if most of the subprograms in a project contain complex decision. In general case, however, we need about ten per cent more test cases if we apply MC/DC method to test our program.

The paper is organized as follows: Section 2 details the most common testing methods including the DC and the MC/DC. We describes the algorithm to compute the number of necessary test cases for DC in Section 3 and for MC/DC in Section 4. The result is detailed in Section 5, while we discuss the related work in Section 6, and Section 7 concludes our results.

2. TESTING METHODS

A variety of methods exist to test the subprograms of a project. In this section we detail the most commonly used ones.

Statement Coverage (SC) requires that each statement of a subprogram must be invoked at least once. The main advantage of this method is that it can be applied directly on object code. However, this method is insensible to some control structures. Consider the code snippet below:

```
T* t = 0;
if (condition)
  t = new T();
t->method();
```

One test case – where the variable `condition` is true – may provide 100% statement coverage, because all the

statements are invoked. In that case the program works properly and we may recognize it is faultless. However, in real applications the `condition` can be false, which might cause non-deterministic behaviour or segmentation fault.

Decision Coverage (DC) enhances statement coverage by requiring that every decision must be evaluated both as true and as false. Thus the previous problem will be discovered at testing time. However, this method ignores branches within boolean expressions, which occur due to short-circuit operators. Let us consider the boolean expression $A||B$. Two test cases (where $A == true$, $B == false$, and $A == false$, $B == false$) can satisfy the requirement of DC, however, the effect of B is not tested. Thus these test cases cannot distinguish between the decision $A||B$ and the decision A.

Condition / Decision Coverage (C/DC) requires that all the arguments in a logical expression must be evaluated both as true and as false. This method obviously solves the problem of DC, however, it takes for a huge overhead due to the increase of arguments in the logical expression increases the number of required test cases exponentially.

Modified Condition / Decision Coverage (MC/DC) is derived from the (C/DC) testing method, however, it needs less test cases to achieve 100 % coverage. This testing method has three requirements:

1. every statement must be invoked at least once,
2. every decision must be evaluated both as true and as false,
3. each condition must be shown to independently affect the evaluation of the decision. In the example in Table 1, the test cases #1 and #2 provide the same input ($false$) for decision B and C. However, decision A is $fals$ in test case #1 and it is $true$ in test case #2. The value of the decision in test case #1 is $false$, and it is $true$ in test case #2. Hence this two test cases prove that the condition A is independently affect the result of the decision.

The independence requirement ensures that the effect of each condition is tested relative to the other conditions.

	#1	#2	#3	#4
A	F	T	F	F
B	F	F	T	F
C	F	F	F	T
output	F	T	T	T

Table 1 Minimal coverage set of $A||B||C$

More information on these coverage methods and others can be found in [12, 13].

3. ANALYSIS OF DC

Our analysis is based on the Clang C++ compiler [1,15]. This tool parses the C++ source code, and build the Abstract Syntax Tree (AST), which is the input of our analyzer algorithms.

The Decision Coverage requires that every decision must be evaluated as true and as false at least once. Thus two test cases are needed. However, one test case can cover several decisions in a function (e.g.: they are in sequence), and more than two test cases are needed if a decision contains nested decisions. See the example below:

```
if(c1)
{
  if(c2)
    stmt1;
  else
    stmt2;
}
else
  stmt3;

//...

if(c3)
  stmt4;
```

If the condition expressions in the example are independent, a test case, which tests the `c1` can test the `c3` at the same time. However, the true branch of `c1`, has a nested condition `c2`. It means `c1` must be evaluated as true twice, because two test cases are needed for `c2`. Thus altogether we need three test cases to reach 100% decision coverage. The test cases, in which `c1`, `c2`, `c3` are evaluated as $(true, true, true)$, $(true, false, false)$ and $(false, any, any)$ are suitable for this.

In our analysis, we consider all the decisions are independent. After a human analysis of source code we consider it does not influence our results in merit.

The algorithm in Table 2 describes the way we compute the number of necessary test cases for DC.

Require: Start with the topmost block of a function

```
1:  if no decision in block then
2:    return 1
3:  else
4:    ReqTC ← 0
5:    for all d in decisions in same level do
6:      if d is in if statement then
7:        trueTC ← ComputeDC(truebranch)
8:        if false branch exist then
9:          falseTC ← ComputeDC(falsebranch)
10:       else
11:         falseTC ← 1
12:       end if
13:       ReqTC ← MAX(ReqTC, trueTC + falseTC)
14:     else if d is in loop statement then
15:       TC ← ComputeDC(loopbody)
16:       ReqTC ← MAX(ReqTC, TC)
17:     else if d is in switch statement then
18:       tmp ← 0
```

```
19:         for all c in case branches do
20:           tmp ← tmp + ComputeDC(c)
21:         end for
22:         if there was not default branch then
23:           tmp ← tmp + 1
24:         end if
25:         ReqTC ← MAX(ReqTC, tmp)
26:       end if
27:     end for
28:     return ReqTC
29:   end if
```

Table 2 ComputeDC

4. ANALYSIS OF MC/DC

The method to analyze MC / DC coverage has two main steps. The first one counts how many test cases are needed to cover the decisions separately. The second step than checks how these decisions affect each other. For efficiency reasons we use approximate value for a decisions with more than 15 conditions. The approximate value, which we use is the number of arguments plus one. This value comes from the technical report [12]. Since these huge decisions are very rare in source codes, the influence of approximate values are not significant.

4.1. Processing the decisions separately

The way to process the decisions, depends on their complexity. The one argument decisions are the simplest: we need two test cases to cover them and we can deal with them in the same way as we did in DC case.

Decisions with two arguments requires three test cases. If the operator is the *logical and* the test cases need to evaluate the arguments as $(true, true)$, $(true, false)$ and $(false, any)$. In C++ the `logical and` operator is a shortcut operator, thus if the first argument is *false*, the second one does not count. If the operator is the *logical or* the test cases should evaluate the arguments as $(false, false)$, $(false, true)$ and $(true, any)$.

If the decision contains more than two arguments we do the four steps below:

1. 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.

2. Eliminate the masked test cases. For example let us consider $A\&\&B$, and there is a test case 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 a logical subexpression and in this case the outcome value of A does not affect the whole logical expression. It means this test case cannot be used to show a part of A is independently affect the result. Therefore, we can say this test case is masked for A. Thus we need to find other test cases to prove that the parts of A are independently affect the result of the decision. You can find a more detailed description and examples in [12] about this step.

3. 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. (The operator *logical and* requires three test cases: $(true, true), (true, false), (false, true)$, thus it implies three sets. The first set contains the indices of those non-masked test cases which evaluate both argument $true$.) The *logical and* and *logical or* operators imply three sets and the *logical not* implies two. 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.

4. 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 index them from 0 to $m-1$. (Most of them will be masked.) Sets $s_0, s_1, ..., s_k$ are the ones mentioned in previous item. We calculate the minimal covering set by Integer Programming (IP), where for every s_i set we have a disparity which is:

   $$\sum_{j=0}^{m-1} f(j) X_j \geq 1$$

   where

   $$f(j) = \begin{cases} 1 & \text{if } j \in s_i \\ 0 & \text{otherwise} \end{cases}$$

   The target function is:

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

   In which 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 [2, 9] with glpk linear programming kit [3]

4.2. Put them together

Like in DC, one test case can test several decisions when they are in the 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.

```
if(a || b) // first decision
{
  if(c && d) // second (nested) decision
  {
    //...
  }
}
```

There are three test cases that are needed for both decisions: $(true, false), (false, true), (false, false)$ for the first and $(true, true), (true, false), (false, true)$ for the second. 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.

Algorithm in Table 3 details how we compute the necessary MD/CD test cases in general.

Require: Start with the topmost block of a function
1: **if** no decision in block **then**
2: **return** (1,0)
3: **else**
4: $ReqTC \leftarrow 0$
5: **for all** d in decisions in same level **do**
6: **if** d is in if statement **then**
7: $(trueN, falseN) \leftarrow ProcessDecision(d)$
8: $TNum \leftarrow ComputeMCDC(truebranch)$
9: $FNum \leftarrow ComputeMCDC(falsebranch)$
10: $trueN \leftarrow MAX(trueN, TNum)$
11: $falseN \leftarrow MAX(falseN, FNum)$
12: **return** $trueN + falseN$
13: **else if** d is in loop statement **then**
14: $(trueN, falseN) \leftarrow ProcessDecision(d)$
15: $BNum \leftarrow ComputeMCDC(loopbody)$
16: $trueN \leftarrow MAX(trueN, BNum)$
17: **return** $trueN + falseN$
18: **else if** d is in switch statement **then**
19: $tmp \leftarrow 0$
20: **for all** c in case branches **do**
21: $tmp \leftarrow tmp + ComputeMCDC(c)$
22: **end for**
23: **if** there was not default branch **then**
24: $tmp \leftarrow tmp + 1$
25: **end if**
26: **return** tmp
27: **end if**
28: **end for**
29: **end if**

Table 3 ComputeMCDC

5. RESULTS

We analyzed five open source projects written in C++ programming language. These projects were

1. BiblioteQ [4], which is a cataloging and library management program;
2. DOSBox [5] a multi platform DOS simulator;
3. FileZilla [6] an FTP client;
4. GParted [7] partition manager;
5. Xerces [8] which is an XML parser library.

First we present some basic characteristics of the projects and then we detail how these characteristics affect the number of required test cases in DC and MC/DC methods. Table 4 shows the effective lines of code. The GParted and BiblioteQ are small and the others are medium size projects.

BiblioteQ	$\sim$ 41000
DOSBox	$\sim$ 119000
FileZilla	$\sim$ 78000
GParted	$\sim$ 16000
Xerces	$\sim$ 142000

Table 4 Effective lines of code

5.1. Cyclomatic complexity

Cyclomatic complexity (or McCabe metric) [11, 17] is widely used metric to measure the complexity of a program. It measures the linearly independent path of the source code. To see how this metric affect the test methods we categorized the functions by their McCabe values. Most of the functions has few McCabe value, because there are several getter, setter and simple helper functions in a projects. Table 5 shows the categories (rows) and the number of functions per projects in a given category (columns). We joint those categories which contain only few functions. In the table we refer the projects by the number as they were enumerated in the beginning of this section.

McCabe cat.	**#1**	**#2**	**#3**	**#4**	**#5**
1	250	1503	1385	320	6785
2	89	483	481	97	1155
3	57	343	332	66	524
4	67	373	236	67	298
5	36	316	180	51	229
6	32	125	152	19	127
7–8	32	113	180	34	156
9–12	25	202	201	26	162
13–20	33	111	144	14	147
21+	50	110	81	8	119

Table 5 Distribution of functions by McCabe metric

Table 6 presents the ratio of the number of required test cases in MC/DC and DC. Every cell represent a ratio in a given category mentioned above, in the corresponding project. For example functions in GParted project (#4) which McCabe value more than 21 require 40% more test cases to test MC/DC than DC. We get very interesting results. In some project the greater McCabe value implies the greater ratio of the two testing methods such as FileZilla (#3) or GParted (#4), however, in other projects it does not affect the difference. This may happen, because the McCabe value can high either the presence of many simple

decisions in sequence or the complex nested decision hierarchy. The first case does not affect the difference, while the second case highly influences it.

McCabe cat.	#1	#2	#3	#4	#5
1	1.00	1.00	1.00	1.00	1.00
2	1.05	1.02	1.07	1.06	1.08
3	1.12	1.09	1.12	1.10	1.12
4	1.03	1.07	1.19	1.11	1.15
5	1.07	1.06	1.15	1.19	1.16
6	1.06	1.18	1.21	1.21	1.12
7–8	1.06	1.15	1.22	1.39	1.14
9–12	1.03	1.12	1.31	1.27	1.15
13–20	1.02	1.25	1.36	1.46	1.13
21+	1.05	1.19	1.52	1.40	1.12

Table 6 Ratio of required test cases for MC/DC and DC of functions by McCabe metric

Figure 1 shows an overview of the ratio. In the first group (marked with "all") we dealt with all the functions in project to compute the ratio, while in the second group (marked with "7+") we kept only those functions, which McCabe value is seven or greater.

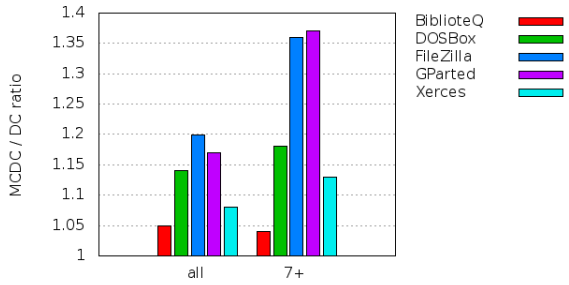

Fig. 1 Ratio of required test cases for MC/DC and DC of functions by McCabe metric

5.2. Maximum deepness of nested blocks

Maximum deepness of nested blocks is another metric to see the complexity of a function. While a functions have a great McCabe value if they contain lot of decision point in a sequence, this metric results a great value only if the decisions are nested. Table 7 shows the nesting characteristics of the projects. The structure of the table is similar to Table 5.

Deepness cat.	#1	#2	#3	#4	#5
1	250	1503	1385	320	6865
2	147	888	919	159	1504
3	110	556	550	136	651
4	58	539	272	53	314
5–6	43	147	190	25	230
7+	63	46	56	9	138

Table 7 Distribution of functions by maximum deepness of nested blocks

Table 8 details the ratio of the number require test cases for functions grouped by their nesting value. The structure of the table is similar to Table 6. We have similar consequences than in previous subsection. FileZilla and GParted has a great increase on the ratio when the deepness is growing, the others gain a slight increase. While we can nest simple decision which does not require additional MC/DC test cases the result is acceptable.

Deepness cat.	#1	#2	#3	#4	#5
1	1.00	1.00	1.00	1.00	1.00
2	1.08	1.04	1.11	1.03	1.09
3	1.02	1.05	1.17	1.23	1.11
4	1.08	1.10	1.22	1.21	1.15
5–6	1.08	1.20	1.39	1.32	1.14
7+	1.05	1.30	1.75	2.16	1.15

Table 8 Ratio of required test cases for MC/DC and DC of functions by maximum deepness of nested blocks

Figure 2 shows an overview of the ratio grouped by nesting value. Its structure is similar to Fig. 1. The first group (marked with "all") contains all the functions, while we select only those functions in the second group which contain at least four level nested blocks.

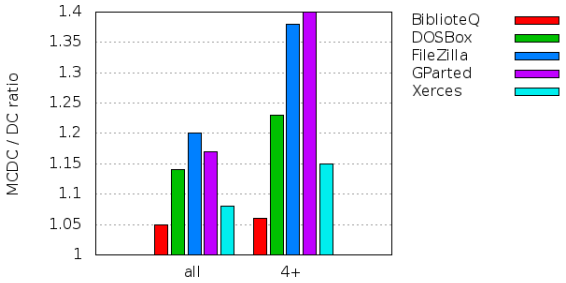

Fig. 2 Ratio of required test cases for MC/DC and DC of functions by nesting

5.3. Maximum arity of the decisions

Maximum arity of the decisions is a metric that measures the complexity of decisions not a functions. A function with a single quite complex decision has a low McCabe or nesting value, however, this metric provide a high value. Distribution of the functions in projects by maximum arity of their decisions is shown in Table 9. The structure of the table is similar to previous ones.

Max arity cat.	#1	#2	#3	#4	#5
0–1	563	2883	2528	570	8748
2	93	565	541	67	732
3	6	54	114	23	172
4+	9	24	51	13	50

Table 9 Distribution of functions by maximum arity of decisions

Table 10 shows the ratio of functions grouped by the maximum arity of their decisions. The structure of the table is similar to Table 6. As we expected the grow of the decisions' arity significantly increase the difference of the number of required test cases. GParted (#4) has the greatest ratio in functions that has at least four argument decisions. In this case more than twice test cases are needed for MC/DC method than DC. However, BiblioteQ (#1) requires more than 50% test cases to cover MC/DC for the same characteristics functions.

Max arity cat.	#1	#2	#3	#4	#5
0–1	1.00	1.00	1.00	1.00	1.00
2	1.09	1.20	1.36	1.36	1.21
3	1.21	1.29	1.62	1.69	1.24
4+	1.56	1.72	2.16	2.36	1.79

Table 10 Ratio of required test cases for MC/DC and DC of functions by McCabe metric by maximum arity of decisions

Figure 3 shows an overview of the ratio grouped by the maximum arity of decisions in a functions. Its structure is similar to Fig. 1. The first group (marked with "all") contains all the functions, while we select only those functions in the second group which has a decision with least four argument.

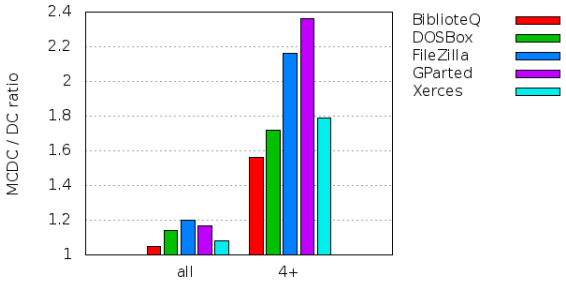

Fig. 3 Ratio of required test cases for MC/DC and DC of functions by nesting

5.4. Overall

Fig 4 shows the ratio of the number of required test cases for all functions in the project, and in the last column, the overall ratio of the functions of all the projects together.

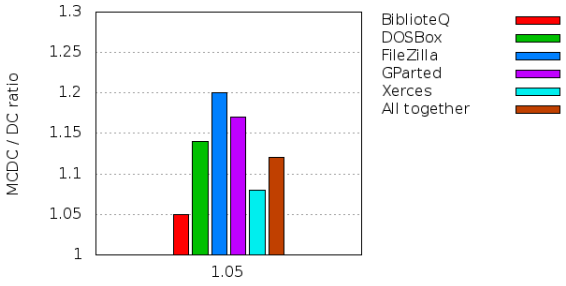

Fig. 4 Ratio of required test cases for MC/DC and DC

We can say about ten per cent more test cases are needed to covert MC/DC than DC in general. However, for functions with high number of arguments in their decisions the difference can be significantly higher: we need more than twice test cases to fulfil the requirements of MC/DC.

6. RELATED WORK

Dupuy et. al. made an empirical evaluation of MC/DC test criteria on the software of HETE-2 (High Energy Transient Explorer) scientific satellite. They found errors that are not detectable with functional testing [10].

Rajan et. al. focused on the connection between the structure of program or model and MC/DC test adequacy coverage [19]. They made their analysis on six realistic systems from the civil avionics domain, and found the MC/DC metric is highly sensitive on the project's structure.

The strict rules of MC/DC test coverage require to high amount and complex test cases. Rayadurgam et. al, made an investigation to the method for generating test cases according to structural coverage criteria [20]. Nevertheless, Jones et. al. focused on the algorithm that is able to effectively reduce the test suit of MC/DC [14].

Studies pointed out that some bugs are not detectable with MC/DC testing. Several research is done to improve this testing mechanism. Woodward et. al compared MC/DC with JJ-paths [24]. Vilkomir et. al introduce their Reinforced Condition / Decision Coverage criteria and compared it against MC/DC [22, 23]. Yu et. al, in turn, compared MC/DC with MUMCUT coverage criteria [25].

We made the similar study for programs written in Ada programming language [21], where we made the analysis on mission critical industrial softwares.

In our future work, we plan to refine our analysis method. We need to handle the same subexpressions in the different decisions. This require heavy data flow analysis on source code, but it makes more precise results. We plan to extend the analysis on more projects, and we can add more aspects on analysis as well.

7. CONCLUSION

We analyzed several well-known open source projects written in C++ 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 right testing methods based on the characteristics of the source could be a useful method.

We found that the difference between DC and MC/DC is about two to seven per cent because the decisions in most functions have only one argument and there are several functions which do not contain decisions at all. If we exclude this special cases we will get a difference that is three or four times larger. The most important aspect that affects the difference is the maximum arity of decisions. For those

functions where there are decisions with more than four arguments, almost forty per cent more MC/DC test cases are needed than DC. But such subprograms are only less than one per cent of the whole project.

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

REFERENCES

[1] Clang C++ compiler: http://clang.llvm.org/.

[2] The homepage of LEMON library: http://lemon.cs.elte.hu/trac/lemon/.

[3] The homepage of glpk LP solver library: http://www.gnu.org/software/glpk/.

[4] The homepage of BiblioteQ the library manager: http://biblioteq.sourceforge.net/.

[5] The homepage of DOSBox the DOS simulator: http://www.dosbox.com/.

[6] The homepage of FileZilla ftp client: https://filezilla-project.org/.

[7] The homepage of GParted partition manager: http://gparted.sourceforge.net/.

[8] The homepage of Xerces XML parser library: http://xerces.apache.org/xerces-c/.

[9] Balázs Dezső, Alpár Jüttner, and Péter Kovács. Lemon–an open source c++ graph template library. *Electronic Notes in Theoretical Computer Science*, 264(5):23–45, 2011.

[10] Arnaud Dupuy and Nancy Leveson. An empirical evaluation of the mc/dc coverage criterion on the hete-2 satellite software. In *Digital Avionics Systems Conference, 2000. Proceedings. DASC. The 19th*, volume 1, pages 1B6–1. IEEE, 2000.

[11] Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *Software Engineering, IEEE Transactions on*, 17(12):1284–1288, 1991.

[12] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. A practical tutorial on modified condition/decision coverage. Technical Report TM-2001-210876, NASA.

[13] K.J. Hayhurst and D.S. Veerhusen. A practical approach to modified condition/decision coverage. In *Digital Avionics Systems, 2001. DASC. 20th Conference*, volume 1, pages 1B2/1–1B2/10 vol.1, 2001.

[14] J.A. Jones and M.J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *Software Engineering, IEEE Transactions on*, 29(3):195–209, 2003.

[15] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[16] Aditya P Mathur. *Foundations of software testing*. China Machine Press, 2008.

[17] T.J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, 1976.

[18] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[19] A. Rajan, M.W. Whalen, and M.P.E. Heimdahl. The effect of program and model structure on mc/dc test adequacy coverage. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 161–170, 2008.

[20] S. Rayadurgam and M.P.E. Heimdahl. Coverage based test-case generation using model checkers. In *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*, pages 83–91, 2001.

[21] Zalán Szűgyi and Zoltán Porkoláb. Necessary test cases for decision coverage and modified condition / decision coverage. *Periodica Polytechnica Electrical Engineering and Computer Science*, 52(3–4):187 – 195, 2008.

[22] Sergiy A. Vilkomir and Jonathan P. Bowen. From MC/DC to RC/DC: formalization and analysis of control-flow testing criteria. *Formal Aspects of Computing*, 18(1):42–62, 2006.

[23] SergiyA. Vilkomir and JonathanP. Bowen. Reinforced condition/decision coverage (RC/DC): A new criterion for software testing. In Didier Bert, JonathanP. Bowen, MartinC. Henson, and Ken Robinson, editors, *ZB 2002:Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 291–308. Springer Berlin Heidelberg, 2002.

[24] Martin R. Woodward and Michael A. Hennell. On the relationship between two control-flow coverage criteria: all JJ-paths and {MCDC}. *Information and Software Technology*, 48(7):433 – 440, 2006.

[25] Yuen Tak Yu and Man Fai Lau. A comparison of mc/dc, {MUMCUT} and several other coverage criteria for logical decisions. *Journal of Systems and Software*, 79(5):577 – 590, 2006.

Received November 2, 2013, accepted December 21, 2013

BIOGRAPHY

Zalán Szűgyi was born on 11. 02. 1980. In 2007 he graduated (MSc) at the department of Programming Languages and Compilers of the Faculty of Informatics at Eötvös Loránd University in Budapest. He is working on his PhD in the field of programming languages, C++. Since 2010 he is working as assistant lecturer at the Department of Programming Languages and Compilers.

Zoltán Porkoláb was born on 11. 11. 1963. He received his MSc. in 1987 and defended his PhD. in 2003 at Eötvös Loránd University (ELTE) in Budapest. He is associate professor at the Department of Programming Languages and Compilers of the Faculty of Informatics at ELTE, where he leads the C++ and Generative Programming research group.