SC-MCC: A Stronger Code Coverage Criterion

Sangharatna Godboley[1,*], Monika Rani Golla[2], and P Radha Krishna[3]
[1,2,3] NITMiner Technologies, Department of Computer Science and Engineering,
National Institute of Technology Warangal, Warangal, India
[1]sanghu@nitw.ac.in, [2]gm720080@student.nitw.ac.in, [3]prkrishna@nitw.ac.in
*corresponding author

Abstract—White-box testing typically involves structural code coverage criteria that measure the extent to which a program's source code has been tested. To efficiently generate test cases while achieving high code coverage, it is important to consider the Short-Circuit evaluation, a feature present in high-level programming languages. In this paper, we propose a new coverage technique called Short-Circuit Multiple Condition Coverage (SC-MCC) and compare it with the state-of-the-art Modified Condition/Decision Coverage (MC/DC). We analyze the test suite size and the time required for the final test suite generation by varying the number of clauses in the conditions. Our experimental results demonstrate that SC-MCC provides sufficient confidence and proves to be more efficient than MC/DC for higher-level languages.

Keywords–White-box testing; Short-Circuit Multiple Condition; Modified Condition/Decision Coverage; Short-Circuit Evaluation;

1. Introduction

White-box testing typically includes structural code coverage criteria such as *Statement Coverage, Decision Coverage, Condition Coverage, Condition/Decision Coverage, Modified Condition/Decision Coverage (MC/DC)* [1, 2, 3, 4], and *Multiple Condition Coverage (MCC)* [5, 6, 7, 8]. In this work, we focus on MCC and MC/DC approaches. MCC generates test cases of the order of 2^N, where N is the total number of Atomic Conditions in a Boolean expression, while MC/DC generates a linear number of test cases ranging from $N+1$ to $2N$. Practitioners recommend the use of MC/DC to test safety-critical applications [9, 10]. Moreover, standards such as DO178B(C) [11] mandate the computation of MC/DC for the *Level A* category software.

When testing a program using Short-Circuit (SC) evaluation, the compiler skips the evaluation of certain sub-expressions within a logical expression. That is, the compiler stops evaluating further sub-expressions as soon as the value of the expression is determined. Since most safety-critical applications are developed using high-level languages, the SC evaluation property can be adopted during testing. The range of test cases for MC/DC with SC (SC-MC/DC) remains the same, that is, from $N+1$ to $2N$ test cases, whereas MCC with SC (SC-MCC) drastically drops the number of test cases. However, this cannot be generalized because *don't-care* situations are completely dynamic, and the number of *generated* test cases depends on them. Still, the number of test cases required for SC-MCC is significantly less than 2^N. In this work, we argue that SC-MCC incurs less overhead compared to SC-MC/DC, and achieves confident code coverage. Here, confident code coverage can be termed as a complete criterion that subsumes all other criteria, including SC-MC/DC. We study the problems with SC-MC/DC and show how SC-MCC can effectively overcome these issues.

To define some of the terms used in MC/DC coverage, we consider an example of a function (see Listing 1) that calculates the area of a triangle based on the lengths of its sides.

```
1 float calculate_area(float s1, float s2, float s3):
2     float p, area;
3     if( !(s1 <= 0 or s2 <= 0 or s3 <= 0) ){
4         p = (s1 + s2 + s3) / 2;
5         area = (p * (p - s1) * (p - s2) * (p - s3))
  ** 0.5;
6         return area;
7     }else{
8         return 0;
9     }
```

Listing 1. A code snippet of a sample original program.

Independent Pair: Two conditions are considered independent if the value of one does not affect the outcome of the other. For example, in the `calculate_area` function, the conditions $s1 \leq 0$ and $s2 \leq 0$ are independent, but $s1 \leq 0$ and $p \leq 0$ are not independent, since the value of p depends on $s1$.

Minimal Set: A set of conditions is considered minimal if removing any condition from the makes it no longer possible to satisfy the MC/DC coverage criterion. For example, in the `calculate_area` function, the set of conditions $\{s1 \leq 0, s2 \leq 0, s3 \leq 0\}$ is a minimal set.

Test Sequence: A test sequence is a set of inputs that executes a specific path through the program. For example, a test sequence for the `calculate_area` function that satisfies the $s1 \leq 0$ condition could be $\{s1 = -1, s2 = 2, s3 = 2\}$.

Test Set: A test set is a collection of test sequences that satisfies the MC/DC coverage criterion for a program. For example, a test set for the `calculate_area` function might include test sequences like $\{s1 = 3, s2 = 4, s3 = 5\}$, $\{s1 = 0, s2 = 1, s3 = 1\}$, and $\{s1 = -1, s2 = 2, s3 = 2\}$.

Test Set Size: The size of a test set is the number of test sequences it contains. A smaller test set size is generally preferable, as it reduces the time and effort required for testing.

Feasible Solution: A solution to a problem is considered feasible if it meets all the constraints of the problem. For exam-

2693-9371/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS-C60940.2023.00083

ple, a feasible solution for the `calculate_area` function might be $\{s1 = 3, s2 = 4, s3 = 5\}$, since it satisfies all the conditions of the program.

Infeasible Solution: A solution to a problem is considered infeasible if it violates one or more of the constraints of the problem. For example, $\{s1 = 0, s2 = 0, s3 = 0\}$ is an infeasible solution for the `calculate_area` function, since it violates the condition that all sides must be positive.

In the case of MC/DC, the generation of test cases is very expensive [12, 13]. Moreover, the set of independent pairs may have multiple choices (minimal sets). It is challenging to choose the best choice/solution for the independent pairs. Here, there are two possible consequences: (i) the selected minimal set is a bad choice, resulting in a low MC/DC score, and (ii) checking all possible minimal sets and considering the best one, which is a time-consuming step. Suppose, for a predicate (6 atomic conditions), we have 5 minimal sets, and each set takes 1 minute to execute. To check all the sets, it takes 5 minutes. On the other hand, for a predicate with 15 to 20 atomic conditions, the number of minimal sets is reasonably large, say 200 sets (instead of 10 sets). To select the best 10 choices/sets, it may need an extra 190 minutes. So, we need to estimate a good number of predicates for a complete program by addressing both consequences.

To prove satisfiable/unsatisfiable [14] for the independent pair test scenarios, it is required to check for each minimal set's test sequence whether it is feasible or infeasible. Because the feasible sequences contribute to the overall MC/DC score, the generation of MC/DC test cases might suffer from the two mentioned consequences. There are several commercial tools (e.g., [15, 16]) that report MC/DC, which results in different versions of MC/DC. However, their algorithms are Black-Box, and thus it is unknown how these tools deal with the above problems. Hence, to achieve better results in software testing, it is recommended to use test cases based on the SC-MCC coverage criterion rather than any version of the MC/DC criterion.

In this work, we propose an economical solution to address the above issues. In general, the test cases generated for SC-MCC are comparatively very few compared to those generated for MCC. Also, unlike MC/DC where there are multiple sets, SC-MCC has only one solution. It is a fact that the SC-MCC solution size is larger than the solution size of any version of MC/DC. However, the difference between the number of test scenarios does not vary much. Secondly, SC-MCC has only one solution, and the test scenarios are executed only once, hence less time consumption. These advantages of SC-MCC make it preferable to generate optimal test cases rather than MC/DC. Further, there is a good chance of achieving high mutation scores [17] for the two faulty types, namely Logical Operator Replacement (LOR) and Relational Operator Replacement (ROR) with SC-MCC when compared to SC-MC/DC. Because different algorithms use different styles, one should choose an algorithm that results in a low MC/DC score to obtain the best choice of a minimal set. On the other hand, any algorithm that computes SC-MCC reports a unique score only.

2. Related Work

Multiple Condition Coverage (MCC) is a challenging metric to meet due to its requirement that every possible combination of conditions must be tested. However, various approaches have been proposed to address this challenge, including risk analysis [13], model-based testing techniques [18], automated testing techniques [19], and combining model checking with symbolic execution [20]. These approaches can increase efficiency and effectiveness in testing while fulfilling coverage criteria, leading to more reliable and secure safety-critical systems. Heimdahl et al. [13] discuss various approaches for calculating and measuring code coverage, which refers to the extent to which a software codebase has been tested, and highlight the importance of achieving high coverage to ensure the reliability and functionality of software systems.

Several automated testing techniques, such as Unified Combinatorial Interaction Testing (U-CIT) [21] and coverage-guided fuzzing [20], are proposed to generate test cases more efficiently and effectively. Model-Based Testing (MBT) [22, 23] is an approach to automatically generate executable test cases according to system specification models. Safety SysML State Machine (S2MSM) [24] is a modeling language specifically designed to address safety requirements in Safety-Critical Systems (SCSs) during the requirement modeling stage. Bounded model checking [25] is a technique used to verify the correctness of the SCSs by exploring all possible states within a certain bound. SCSs such as avionics prefer MC/DC criterion satisfaction due to their effectiveness in detecting errors.

Sanjai et al. [26] proposed a new approach that combines model checking with symbolic execution to generate test sequences that satisfy MC/DC-like coverage criteria for state-based formalisms. This involves encoding the software behavioral model and the coverage criterion as a temporal logic formula, which is then checked against the model using a model checker. If the formula is not satisfied, the model checker generates a counterexample, which is used to guide symbolic execution to generate a new test sequence that satisfies the coverage criterion.

Chilenski [27] explored three variations of the Modified Condition Decision Coverage (MCDC) criterion, a popular testing technique used in software engineering. The results of the study show that the stronger version of the criterion is the most effective in detecting faults, followed by the standard MCDC criterion and then the MCC criterion. However, it requires significantly more test cases to achieve the same level of coverage as the other two criteria, making it more time-consuming and expensive to implement. Literature shows that, the use of short-circuit evaluation can make MC/DC testing more practical for safety-critical software implemented in languages that support it, such as C and C++. This can help reduce the time and cost associated with testing such software while still ensuring a high level of safety.

Dupuy et al. [6] concluded that although the cost of testing to meet MC/DC criteria was relatively high, it was not substantially more expensive than obtaining lower levels of code coverage. They discovered that essential errors were identified through the additional test cases necessary to achieve MC/DC coverage, meaning in the software that was not covered by black-box functional testing. Hence, their study highlights the importance of augmenting testing to achieve MC/DC coverage. Sergiy et al. [28] proposed the RC/DC criterion that addresses the limitation of the MC/DC criterion, which does not encompass the testing of "false operation" type failures that can be critical in safety-critical computer systems. The RC/DC criterion aims to overcome this limitation by mandating the assessment of scenarios in which modifying a condition retains the value of a decision.

Similarly, Kandl et al. [29] proposed a new criterion, MC/DC with Short-Circuit evaluation. The study indicates that the overhead associated with generating a test suite for MCC only increases by an average of approximately 35% when compared to MC/DC. Furthermore, the maximum overhead observed was approximately 100%. Here, in this paper, we formalize the notation of SC-MCC in the form of a Meta Program. Also, we evaluated the efficiency of the SC-MCC coverage criterion against MC/DC by analyzing the number of test cases generated when the number of conditions varies. Our findings indicated that the adoption of SC-MCC resulted in a reasonable increase in computation time compared to MC/DC. Therefore, our study supports the viability of using SC-MCC as an alternative coverage criterion.

The main contributions of this paper are as follows:

- We measured the efficiency of the proposed SC-MCC coverage criterion over MC/DC in terms of condition count versus the number of test cases generated. Thus, we proved that there is an acceptable overhead when SC-MCC has opted instead of MC/DC.
- We compared the time taken to prove n number of conditions for each coverage criterion say MC/DC, SC-MCC, and MCC. Thus, demonstrating the efficiency of SC-MCC over MCC.

3. Proposed Approach

The proposed framework comprises two modes: (a) Mode 1 corresponds to MCDC, and (b) Mode 2 corresponds to SC-MCC. The original C program serves as input for both of these modes. There are two versions of the program: one follows the C Bounded Model Checker (CBMC) [25], and the other follows GCOV [30]. In the CBMC version, the `scanf` function is replaced with the `_CPROVER_input` function.

In [31], authors developed a Meta Program Generator (MPG) that instruments the test sequences (MC/DC or SC-MCC) of each predicate into the program. Fig. 1 depicts the framework of Mode 1, MC/DC. First, MC/DC sequences are obtained from the CBMC version program using the *"–cover mcdc –show-properties"* option of the CBMC tool. These sequences are instrumented into the given CBMC version program, called MC/DC Meta Program, using MPG. Next, with the help of the *"–cover cover"* option of CBMC, the MC/DC criterion-based test suites are generated from the MC/DC Meta program. The framework of Mode 2, SC-MCC, is shown in Fig. 2. The SC-MCC sequences are generated by our Sequence Generator. These sequences are instrumented into the given CBMC version program, called SC-MCC Meta Program, using Meta Program Generator. Next, with the help of the *"–cover cover"* option of CBMC, the SC-MCC criteria-based test suites are generated from the SC-MCC Meta program.

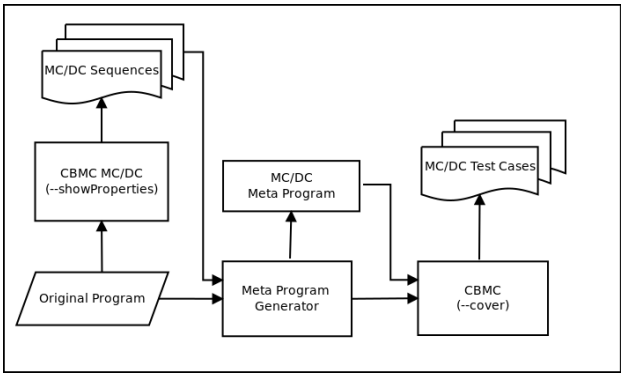

Figure 1. Framework of the Mode1-MC/DC.

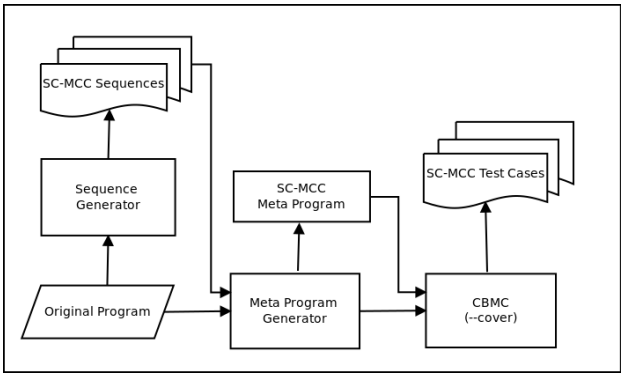

Figure 2. Framework of the Mode2-SC-MCC.

Example of MPG: We explain the working of the MPG with an example. Consider the code snippet of the sample original program shown in Listing 2 [31]. After identifying the predicates, we generated SC-MCC sequences, and the MPG outputs a Meta program that contains each predicate's SC-MCC sequences just above the predicate line, in the following format: *__CPROVER_cover(...);*. The corresponding meta program to Listing 2 is shown in Listing 3 [32]. Here, the lines from 2 to 9 are the SC-MCC sequences of the predicate located at line 10. Thus, this meta program, when executed with the help of CBMC's *–cover* option, produces the SC-MCC tests.

```
1 int symb = nondet_int();
2 __CPROVER_input("symb",symb);
3 if((symb != 10) && (symb != 6) && (symb != 1) && (
      symb != 8) && (symb != 2) && (symb != 5)){printf
      ("Problem with return -2 \n");}
```

```
calculate_output(symb);
```

Listing 2. A code snippet of a sample original program.

```
int symb = nondet_int();
__CPROVER_input("symb",symb);
__CPROVER_cover((!(!(symb!=10)&&symb!=6&&symb!=1&&
    symb!=8&&symb!=2&&symb!=5)));
__CPROVER_cover((!(symb!=10&&!(symb!=6)&&symb!=1&&
    symb!=8&&symb!=2&&symb!=5)));
__CPROVER_cover((!(symb!=10&&symb!=6&&!(symb!=1)&&
    symb!=8&&symb!=2&&symb!=5)));
__CPROVER_cover((!(symb!=10&&symb!=6&&symb!=1&&!(
    symb!=8)&&symb!=2&&symb!=5)));
__CPROVER_cover((!(symb!=10&&symb!=6&&symb!=1&&symb
    !=8&&!(symb!=2)&&symb!=5)));
__CPROVER_cover((!(symb!=10&&symb!=6&&symb!=1&&symb
    !=8&&symb!=2&&!(symb!=5))));
__CPROVER_cover((!(symb!=10&&symb!=6&&symb!=1&&symb
    !=8&&symb!=2&&symb!=5)));
if((symb != 10) && (symb != 6) && (symb != 1) && (
    symb != 8) && (symb != 2) && (symb != 5))
{
    printf("Problem with return -2 \n");
}
calculate_output(symb);
```

Listing 3. A code snippet of SC-MCC Meta program(generated by MPG).

TABLE I
TOTAL NUMBER OF TRUTH COMBINATIONS OR TEST SEQUENCES FOR DIFFERENT CRITERIA (NOTE: N IS THE TOTAL NUMBER OF ATOMIC CONDITIONS IN A BOOLEAN EXPRESSION; #C IS THE CONDITION COUNT; #MODE1 AND #MODE2 ARE THE NUMBER OF MC/DC AND SC-MCC COMBINATIONS GENERATED USING CBMC, RESPECTIVELY.).

#C	MC/DC		#Mode1	#Mode2	MCC
N	N+1	2N	-	$<<2^N$	2^N
1	2	2	2	2	2
2	3	4	3	3	4
3	4	6	4	5	8
4	5	8	5	7	16
5	6	10	6	11	32
6	7	12	7	15	64
7	8	14	8	23	128
8	9	16	9	31	256
9	10	18	10	50	512
10	11	20	11	69	1024
11	12	22	12	107	2048
12	13	24	13	145	4096
13	14	26	TO(1 hr)	233	8192
14	15	28	TO(1 hr)	321	16384
15	16	30	TO(1 hr)	497	32768
16	17	32	TO(1 hr)	673	65536
17	18	34	TO(1 hr)	937	131072
18	19	36	TO(1 hr)	1201	262144
19	20	38	TO(1 hr)	1729	524288
20	21	40	TO(1 hr)	2257	1048576

4. RESULTS AND DISCUSSION

For experiments, we considered one sample C program as shown in Appendix A. Listing 4 with 20 predicates, and it's annotated version in Listing 5[1] [32]. The predicates are in ascending order, with the total number of atomic conditions present being the same as the predicate serial number. For example, Predicate 2 has two atomic conditions, and Predicate 20 has twenty atomic conditions.

To demonstrate our view, we considered the document presented in [27], which shows the most recent advancements in MC/DC versions along with their detailed analysis. The empirical data for the average coverage test set size for common solvable expressions and the number of minimal test sets were presented in this document. In this work, we are interested in showing the size of the test set and the total number of possible minimal test sets required for any expression with a certain number of atomic conditions. Specifically, there are 10 unique Boolean expressions for predicates with two conditions, 52 for predicates with three conditions, 56 for predicates with four conditions, 70 for predicates with five conditions, and 46 for predicates with six conditions.

TABLE II
TOTAL CHECKS AND TIME TAKEN BY AVERAGE MC/DC TEST SETS FOR DIFFERENT CONDITIONS COUNT. (NOTE: #C IS THE CONDITION COUNT; MSIZE IS THE AVG. MC/DC TEST SET SIZE; #MSETS IS THE AVG. MC/DC TEST SETS; 1 CHECK = 1 TIME UNIT; TTIME IS THE TOTAL TIME.)

#C	MSize	#MSets	#Checks	TTime
1	2	1	2	2
2	3	1	3	3
3	4	1	4	4
4	5	2	10	10
5	6	4	24	24
6	7	10	70	70

We extracted data from [27] to display the size of test sets and the total number of test sets (choices required for MC/DC). We considered the basic version of MC/DC Unique Cause with Context-Dependent that shows the minimum choices required (whole numbers, instead of floating-point values, are considered here). Columns 2 and 3 in Table II show the useful data that was extracted. Note that the truth combinations for any Boolean expression are computed statically. To generate a test input for a particular truth combination/test sequence, we need to check its model availability. The test input for the test sequence can only be generated if the constraint solver proves the test sequence as SATISFIABLE. Additionally, to prove that the test sequence is UNSATISFIABLE and that a test input cannot be generated, we need to check again with the constraint solver.

To achieve maximum coverage and discover all the bugs in the code, one needs to check all the Test Sets/Choices. This is necessary because it cannot be validated that any selected set/choice has achieved optimal MC/DC unless it has achieved 100% coverage with the first set itself. Certainly, there is a

[1]For space issues we have shown part of the program so that readers can look at the paper and understand the annotations. For the full version of the programs readers are recommended to refer [32].

need to check with another minimal test set as well. Existing work, like CBMC (a state-of-the-art for BMC) with coverage mode (see Table I), selects a random N+1 test sequence and shows the satisfiability of the sequence.

TABLE III
TIME EFFORT COMPARISON FOR MC/DC, SC-MCC, AND MCC. (NOTE: #C IS THE CONDITION COUNT; MT, ST AND MCT ARE THE TOTAL TIME TAKEN TO PROVE MC/DC, SC-MCC AND MCC TEST SEQUENCES, RESPECTIVELY.)

#C	MT (s)	ST (s)	MCT (s)
1	2	2	2
2	3	3	4
3	4	5	8
4	10	7	16
5	24	11	32
6	70	15	64

TABLE IV
THE ESTIMATED TIME DATA MC/DC TEST SEQUENCES WITH AVERAGE OF 15 MINIMAL MC/DC TEST SETS. SINCE ACTUAL DATA FOR MC/DC AND CBMC-MC/DC IS MISSING WE HAVE NOT COMPUTED THE ESTIMATED DATA BEYOND 12 ATOMIC CONDITIONS IN A BOOLEAN EXPRESSION. (NOTE: #C IS THE CONDITION COUNT; MT, ST AND MCT ARE THE TOTAL TIME TAKEN TO PROVE MC/DC, SC-MCC AND MCC TEST SEQUENCES, RESPECTIVELY.)

#C	MT (s) ((N+1) X 15 = ?)	ST (s)	MCT (s)
7	8 X 15 = 120	23	128
8	9 X 15 = 135	31	256
9	10 X 15 = 150	50	512
10	11 X 15 = 165	69	1024
11	12 X 15 = 180	107	2048
12	13 X 15 = 195	145	4096

To calculate the total number of checks needed for all MC/DC test inputs, we require Column 4 in Table II, which represents the product of Columns 2 and 3. Additionally, we assume that each sequence takes an equal amount of time units (refer to Column 5 in Table II). By analyzing the data from Tables I and II, we can derive the information regarding the potential time consumption for MC/DC, SC-MCC, and MCC, as illustrated in Table III.

In Table III, Column 2 displays the time efforts required for MC/DC sequences. To enhance the confidence in code coverage, we need to verify all the minimal sets. At a certain point, these checks rise to a peak, making the generation of MC/DC test cases very expensive, as illustrated in Fig. 3. For instance, consider the last row, which indicates a Boolean expression with 6 atomic conditions. It requires 70 checks/test sequences (some of which may be redundant), surpassing even that of MCC. However, SC-MCC requires only 15 checks/test sequences to achieve optimal code coverage and provide high confidence that all the bugs have been detected. It's worth noting that existing MC/DC and MCC methods are very ex-

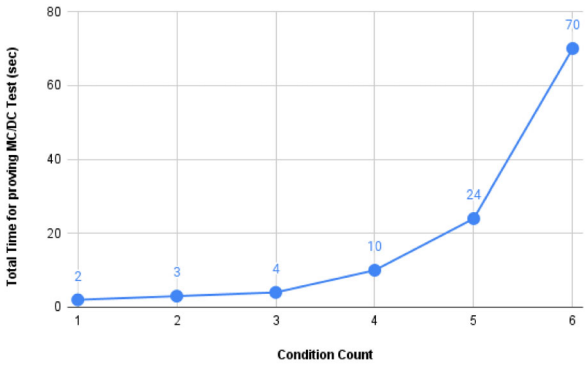

Figure 3. Time efforts required for MC/DC sequences.

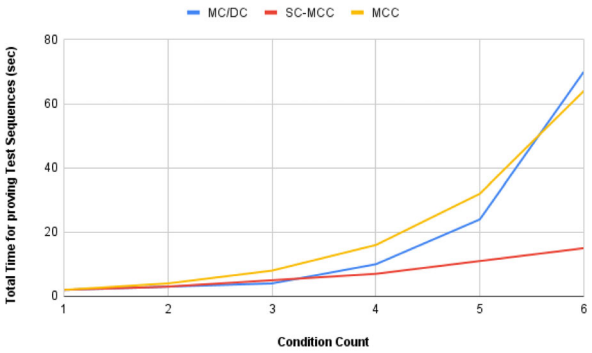

Figure 4. Time efforts required for MC/DC, SC-MCC and MCC Test sequences.

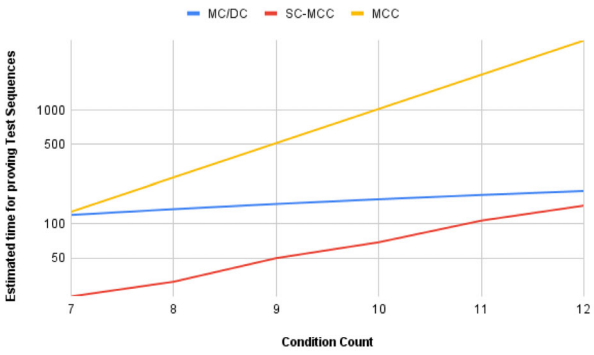

Figure 5. Estimated Time efforts required for MC/DC, SC-MCC and MCC Test sequences.

pensive and impractical. The information on the total number of required test sequences for MC/DC and CBMC-MC/DC (see Table I) can ensure optimal code coverage is achieved. To increase code coverage and error detection probability, one needs to check all possible choices [27] and follow Table II for Optimal MC/DC test cases (which can be very expensive, as shown in Table III).

To estimate the trend of computing the average minimum number of test sets for Boolean expressions with different atomic conditions in increasing order, we assume that for a predicate with 6 atomic conditions, average minimal MC/DC test sets can be computed in 15 units. Note that, in real-time, computation time units change dynamically and could be very high. Table IV presents the time efforts required for different checks for MC/DC, SC-MCC, and MCC. Here, MCC exhibits an exponential increase and thus cannot be considered a stronger solution. The corresponding graphs are illustrated in Figures 4 and 5. From Table IV, it is evident that MC/DC is very expensive and may become impractical at a certain point. Therefore, the optimal solution is to adopt the SC-MCC technique, which assures optimal code coverage and a high bug detection rate. Furthermore, SC-MCC is linear in nature compared to other methods.

5. Conclusion

It is reasonable to argue that MCC-based test cases have greater bug-finding capabilities than MC/DC. Since MCC generates all possible test cases, it provides more comprehensive coverage of the conditions in a predicate, making it more effective in identifying faults that may be missed by MC/DC. Moreover, because MCC generates all possible test cases, it can identify faults that may not be related to independent pairs of conditions but rather arise from the interaction of multiple conditions. This makes MCC more effective in detecting faults that are not covered by MC/DC. Studies have shown that SC-MCC techniques can enhance the fault-finding capabilities of traditional MC/DC. For example, in [29], SC-MCC detects 100% of faults, compared to the 95.23% fault detection rate achieved by traditional MC/DC. In our future work, we plan to experiment with certain benchmark programs to validate the proposed framework implemented with well-known automated test case generation techniques such as Model Checking, Fuzzing, and Dynamic Symbolic Execution.

Acknowledgement

This work is sponsored by IBITF, Indian Institute of Technology (IIT) Bhilai, under the grant of PRAYAS scheme, DST, Government of India.

References

[1] S. K. Barisal, S. P. S. Chauhan, A. Dutta, S. Godboley, B. Sahoo, and D. P. Mohapatra, "Boompizer: Minimization and prioritization of concolic based boosted mc/dc test cases," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 10, Part B, pp. 9757–9776, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157821003414

[2] S. Godboley, A. Dutta, D. P. Mohapatra, and R. Mall, "Scaling modified condition/decision coverage using distributed concolic testing for java programs," *Computer Standards & Interfaces*, vol. 59, pp. 61–86, 2018.

[3] M. R. Golla and S. Godboley, "Gmutant: A gcov based mutation testing analyser," in *Proceedings of the 16th Innovations in Software Engineering Conference*, ser. ISEC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3578527.3578546

[4] G. M. Rani and S. Godboley, "Poster: A gcov based new profiler, gmcov, for mc/dc and sc-mcc," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 469–472.

[5] Z. Awedikian, K. Ayari, and G. Antoniol, "Mc/dc automatic test input data generation," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 1657–1664.

[6] A. Dupuy and N. Leveson, "An empirical evaluation of the mc/dc coverage criterion on the hete-2 satellite software," in *19th DASC. 19th Digital Avionics Systems Conference. Proceedings (Cat. No.00CH37126)*, vol. 1, 2000, pp. 1B6/1–1B6/7 vol.1.

[7] S. Godboley and A. Dutta, "Dy-copeca: A dynamic version of mc/dc analyzer for c program," 01 2021, pp. 197–204.

[8] S. Godboley, A. Dutta, D. Mohapatra, and R. Mall, "Gecojap: A novel source-code preprocessing technique to improve code coverage," *Computer Standards & Interfaces*, vol. 55, 04 2017.

[9] M. A. Almeida, J. de Melo Bezerra, and C. M. Hirata, "Automatic generation of test cases for critical systems based on mc/dc criteria," in *2013 IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC)*, 2013, pp. 7C5–1–7C5–10.

[10] J. Guan, J. Offutt, and P. Ammann, "An industrial case study of structural testing applied to safety-critical embedded software," in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ser. ISESE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 272–277.

[11] H. Lougee, "Software considerations in airborne systems and equipment certification," 2001. [Online]. Available: https://api.semanticscholar.org/CorpusID:5959053

[12] A. Dutta, S. S. Srivastava, S. Godboley, and D. P. Mohapatra, "Combi-fl: Neural network and sbfl based fault localization using mutation analysis," *Journal of Computer Languages*, vol. 66, p. 101064, 2021.

[13] M. P. Heimdahl, M. W. Whalen, A. Rajan, and M. Staats, "On mc/dc and implementation structure: An empirical study," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008, pp. 5.B.3–1–5.B.3–13.

[14] J. Jaffar, R. Maghareh, S. Godboley, and X.-L. Ha, "Tracerx: Dynamic symbolic execution with interpolation (competition contribution)," in *Fundamental Approaches to Software Engineering*, H. Wehrheim and J. Cabot, Eds. Cham: Springer International Publishing, 2020, pp. 530–534.
[15] V. T. GmbH, "Testwell ctc++ test coverage analyser."
[16] R. Systems, "Qualification of rapicover for mc/dc coverage of do-178b level-a software."
[17] D. Holling, S. Banescu, M. Probst, A. Petrovska, and A. Pretschner, "Nequivack: Assessing mutation score confidence," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2016, pp. 152–161.
[18] M. N. Zafar, W. Afzal, and E. Enoiu, "Evaluating system-level test generation for industrial software: A comparison between manual, combinatorial and model-based testing," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, ser. AST '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 148–159.
[19] K. Ghani and J. A. Clark, "Automatic test data generation for multiple condition and mcdc coverage," in *2009 Fourth International Conference on Software Engineering Advances*, 2009, pp. 152–157.
[20] Y. Yang, "Improve model testing by integrating bounded model checking and coverage guided fuzzing," *Electronics*, vol. 12, no. 7, 2023.
[21] G. Coskun, C. Coskun, H. Mercan, and C. Yilmaz, "Using unified combinatorial interaction testing for mc/dc coverage," in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2022, pp. 57–62.
[22] L. C. Jaw, H. T. Van, D. Homan, V. Crum, W. Chou, K. Keller, K. Swearingen, and T. Smith, "Model-based approach to validation and verification of flight critical software," in *2008 IEEE Aerospace Conference*, 2008, pp. 1–8.
[23] C. Singh, J. Shivamurthy, and A. Garg, "Model based test framework for verification of flight control software," in *2023 International Conference on Computer, Electrical & Communication Engineering (ICCECE)*, 2023, pp. 1–5.
[24] C. Wang, H. Sun, H. Dou, H. Chen, and J. Liu, "Mc/dc test case automatic generation for safety-critical systems," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, 2022, pp. 732–743.
[25] D. Kroening and M. Tautschnig, "Cbmc – c bounded model checker," vol. 8413, 04 2014, pp. 389–391.
[26] S. Rayadurgam and M. Heimdahl, "Generating mc/dc adequate test sequences through model checking." 01 2003, p. 91.
[27] J. Chilenski, "An investigation of three forms of the modified condition decision coverage (mcdc) criterion," 01 2001.
[28] S. A. Vilkomir and J. P. Bowen, "From mc/dc to rc/dc: formalization and analysis of control-flow testing criteria," *Formal Aspects of Computing*, vol. 18, pp. 42–62, 2006.
[29] S. Kandl and S. Chandrashekar, "Reasonability of mc/dc for safety-relevant software implemented in programming languages with short-circuit evaluation," in *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*, 2013, pp. 1–6.
[30] F. S. Foundation, "gcov-a test coverage program in: Using the gnu compiler collection (gcc)," 02 2010.
[31] M. R. Golla, "Automated sc-mcc test case generation," in *15th Innovations in Software Engineering Conference*, ser. ISEC 2022. New York, NY, USA: ACM, 2022.
[32] "Figshare Data: Listings," 2021, https://figshare.com/s/42bfeb5db3728c829501.

Appendix A: Case study

```
#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <stdlib.h>
int main()
{
int a,b,c,d;
scanf("%d",&a);
scanf("%d",&b);
scanf("%d",&c);
scanf("%d",&d);

if (( a > 10))
   print("Predicate 1");


if (( a > 10) || (b < 10))
   print("Predicate 2");


if ((( a > 10) && (b < 10)) || (c == 10))
   print("Predicate 3");


if ((( a > 10) && (b < 10)) || ((c == 10) && (d !=
    10)))
   print("Predicate 4");


if ((( a > 10) && (b < 10)) || ((c == 10) && (d !=
    10))) || (a == 10))
   print("Predicate 5");


if ((( a > 10) && (b < 10)) || ((c == 10) && (d !=
    10))) || ((a == 10) || (b != 10)))
   print("Predicate 6");


if ((( a > 10) && (b < 10)) || ((c == 10) && (d !=
    10))) || (((a == 10) || (b != 10)) && (c > 10)))
   print("Predicate 7");


if ((( a > 10) && (b < 10)) || ((c == 10) && (d !=
    10))) || (((a == 10) || (b != 10)) && ((c > 10)
    || ( d < 10))))
```

```
   print("Predicate 8");


if ((((( a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10)))) && (a <= 10))
   print("Predicate 9");


if ((((( a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10)))) && ((a <= 10) && (b >= 10)))
   print("Predicate 10");


if ((((( a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10)))) && (((a <= 10) && (b >= 10)) || (c <= 10)))
   print("Predicate 11");


if ((((( a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10)))) && (((a <= 10) && (b >= 10)) || ((c <= 10) && (d >= 10))))
   print("Predicate 12");


if (((((( a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10)))) && (((a <= 10) && (b >= 10)) || ((c <= 10) && (d >= 10))))) || (a != 10))
   print("Predicate 13");


if (((((( a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10)))) && (((a <= 10) && (b >= 10)) || ((c <= 10) && (d >= 10))))) || ((a != 10) && (b != 10)))
   print("Predicate 14");


if (((((( a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10)))) && (((a <= 10) && (b >= 10)) || ((c <= 10) && (d >= 10))))) || (((a != 10) && (b != 10)) || (c != 10)))
   print("Predicate 15");


if (((((( a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10)))) && (((a <= 10) && (b >= 10)) || ((c <= 10) && (d >= 10))))) || (((a != 10) && (b != 10)) || ((c != 10) && (d != 10))))
   print("Predicate 16");


if (((((( a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10)))) && (((a <= 10) && (b >= 10)) || ((c <= 10) && (d >= 10))))) || ((((a != 10) && (b != 10)) || ((c != 10) && (d != 10))) && (a > 10)))
   print("Predicate 17");


if (((((( a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10)))) && (((a <= 10) && (b >= 10)) || ((c <= 10) && (d >= 10))))) || ((((a != 10) && (b != 10)) || ((c != 10) && (d != 10))) && (((( a > 10) && (b < 10)))))
   print("Predicate 18");


if (((((( a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10)))) && (((a <= 10) && (b >= 10)) || ((c <= 10) && (d >= 10))))) || ((((a != 10) && (b != 10)) || ((c != 10) && (d != 10))) && (((( a > 10) && (b < 10)) || (c == 10))))
   print("Predicate 19");


if (((((( a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10)))) && (((a <= 10) && (b >= 10)) || ((c <= 10) && (d >= 10))))) || ((((a != 10) && (b != 10)) || ((c != 10) && (d != 10))) && (((( a > 10) && (b < 10)) || ((c == 10) && (d != 10))))))
   print("Predicate 20");

return 0;
}
```

Listing 4. This program is a sample C program with 20 predicates.

```

#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <stdlib.h>
int main()
{
int a,b,c,d;
int a= nondet_int(); __CPROVER_input("a", a);
int b= nondet_int(); __CPROVER_input("b", b);
int c= nondet_int(); __CPROVER_input("c", c);
int d= nondet_int(); __CPROVER_input("d", d);

if (( a > 10))
   print("Predicate 1");

__CPROVER_cover(! (!(a>10)&&!(b<10)) );
__CPROVER_cover(! (!(a>10)&&b<10) );
__CPROVER_cover(! (a>10) );
if (( a > 10) || (b < 10))
   print("Predicate 2");

__CPROVER_cover(! (!(a>10)&&!(c==10)) );
__CPROVER_cover(! (!(a>10)&&c==10) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10) );
__CPROVER_cover(! (a>10&&b<10) );
if ((( a > 10) && (b < 10)) || (c == 10))
   print("Predicate 3");

__CPROVER_cover(! (!(a>10)&&!(c==10)) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)) );
__CPROVER_cover(! (!(a>10)&&c==10&&d!=10) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)) )
    ;
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&d!=10) );
__CPROVER_cover(! (a>10&&b<10) );
if ((( a > 10) && (b < 10)) || ((c == 10) && (d != 10)))
   print("Predicate 4");

__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&a==10) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&!(a
    ==10)) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&a==10)
```

```
    );
__CPROVER_cover(! (!(a>10)&&c==10&&d!=10) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&!(a==10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&a==10) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&!(a==10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&a==10) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&d!=10) );
__CPROVER_cover(! (a>10&&b<10) );
if ((((  a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (a == 10))
   print("Predicate 5");

__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&b!=10) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&a==10) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&!(a==10)&&b!=10) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&a==10) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&b!=10) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&a==10) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&!(a==10)&&b!=10) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&!(a==10)&&b!=10) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&a==10) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&d!=10) );
__CPROVER_cover(! (a>10&&b<10) );
if ((((  a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || ((a == 10) || (b != 10)))
   print("Predicate 6");

__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&b!=10&&!(c>10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&b!=10&&c>10) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&a==10&&!(c>10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&a==10&&c>10) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&!(a==10)&&b!=10&&!(c>10)) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&!(a==10)&&b!=10&&c>10) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&a==10&&!(c>10)) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&a==10&&c>10) );
__CPROVER_cover(! (!(a>10)&&c==10&&d!=10) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&!(a==10)&&b!=10&&!(c>10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&!(a==10)&&b!=10&&c>10) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&a==10) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&!(a==10)&&b!=10&&!(c>10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&!(a==10)&&b!=10&&c>10) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&a==10&&!(c>10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&a==10&&c>10) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&!(a==10)&&b!=10&&!(c>10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&!(a==10)&&b!=10&&c>10) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&a==10&&!(c>10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&a==10&&c>10) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&d!=10) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&!(a==10)&&b!=10&&!(c>10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&!(a==10)&&b!=10&&c>10) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&a==10&&!(c>10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&!(c==10)&&a==10&&c>10) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&!(a==10)&&b!=10&&!(c>10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&!(a==10)&&b!=10&&c>10) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&a==10&&!(c>10)) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&!(d!=10)&&a==10&&c>10) );
__CPROVER_cover(! (a>10&&!(b<10)&&c==10&&d!=10) );
__CPROVER_cover(! (a>10&&b<10) );
if ((((  a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && (c > 10)))
   print("Predicate 7");

...
if ((((  a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10))))
   print("Predicate 8");

...
if (((((  a > 10) && (b < 10)) || ((c == 10) && (d != 10))) || (((a == 10) || (b != 10)) && ((c > 10) || ( d < 10)))) && (a <= 10))
   print("Predicate 9");

__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&b!=10&&!(c>10)&&!(d<10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&b!=10&&!(c>10)&&d<10&&!(a<=10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&b!=10&&!(c>10)&&d<10&&a<=10&&!(b>=10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&b!=10&&!(c>10)&&d<10&&a<=10&&b>=10) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&b!=10&&c>10&&!(a<=10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&b!=10&&c>10&&a<=10&&!(b>=10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&!(a==10)&&b!=10&&c>10&&a<=10&&b>=10) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&a==10&&!(c>10)&&!(d<10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&a==10&&!(c>10)&&d<10&&!(a<=10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&a==10&&!(c>10)&&d<10&&a<=10&&!(b>=10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&a==10&&!(c>10)&&d<10&&a<=10&&b>=10) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&a==10&&c>10&&!(a<=10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&a==10&&c>10&&a<=10&&!(b>=10)) );
__CPROVER_cover(! (!(a>10)&&!(c==10)&&a==10&&c>10&&a<=10&&b>=10) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&!(a==10)&&!(b!=10)) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&!(a==10)&&b!=10&&!(c>10)&&!(d<10)) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&!(a==10)&&b!=10&&!(c>10)&&d<10&&!(a<=10)) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&!(a==10)&&b!=10&&!(c>10)&&d<10&&a<=10&&!(b>=10)) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&!(a==10)&&b!=10&&!(c>10)&&d<10&&a<=10&&b>=10) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&!(a==10)&&b!=10&&c>10&&!(a<=10)) );
__CPROVER_cover(! (!(a>10)&&c==10&&!(d!=10)&&!(a
```

```
    ==10)&&b!=10&&c>10&&a<=10&&!(b>=10)) );
__CPROVER_cover( !(!(a>10)&&c==10&&!(d!=10)&&!(a
    ==10)&&b!=10&&c>10&&a<=10&&b>=10) );
__CPROVER_cover( !(!(a>10)&&c==10&&!(d!=10)&&a
    ==10&&!(c>10)&&!(d<10)) );
__CPROVER_cover( !(!(a>10)&&c==10&&!(d!=10)&&a
    ==10&&!(c>10)&&d<10&&!(a<=10)) );
__CPROVER_cover( !(!(a>10)&&c==10&&!(d!=10)&&a
    ==10&&!(c>10)&&d<10&&a<=10&&!(b>=10)) );
__CPROVER_cover( !(!(a>10)&&c==10&&!(d!=10)&&a
    ==10&&!(c>10)&&d<10&&a<=10&&b>=10) );
__CPROVER_cover( !(!(a>10)&&c==10&&!(d!=10)&&a==10&&
    c>10&&!(a<=10)) );
__CPROVER_cover( !(!(a>10)&&c==10&&!(d!=10)&&a==10&&
    c>10&&a<=10&&!(b>=10)) );
__CPROVER_cover( !(!(a>10)&&c==10&&!(d!=10)&&a==10&&
    c>10&&a<=10&&b>=10) );
__CPROVER_cover( !(!(a>10)&&c==10&&d!=10&&!(a<=10))
    );
__CPROVER_cover( !(!(a>10)&&c==10&&d!=10&&a<=10&&!(b
    >=10)) );
__CPROVER_cover( !(!(a>10)&&c==10&&d!=10&&a<=10&&b
    >=10) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&!(a==10)
    &&!(b!=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&!(a==10)
    &&b!=10&&!(c>10)&&!(d<10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&!(a==10)
    &&b!=10&&!(c>10)&&d<10&&!(a<=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&!(a==10)
    &&b!=10&&!(c>10)&&d<10&&a<=10&&!(b>=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&!(a==10)
    &&b!=10&&!(c>10)&&d<10&&a<=10&&b>=10) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&!(a==10)
    &&b!=10&&c>10&&!(a<=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&!(a==10)
    &&b!=10&&c>10&&a<=10&&!(b>=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&!(a==10)
    &&b!=10&&c>10&&a<=10&&b>=10) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&a
    ==10&&!(c>10)&&!(d<10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&a
    ==10&&!(c>10)&&d<10&&!(a<=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&a
    ==10&&!(c>10)&&d<10&&a<=10&&!(b>=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&a
    ==10&&!(c>10)&&d<10&&a<=10&&b>=10) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&a==10&&c
    >10&&!(a<=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&a==10&&c
    >10&&a<=10&&!(b>=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&!(c==10)&&a==10&&c
    >10&&a<=10&&b>=10) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&!(d!=10)
    &&!(a==10)&&!(b!=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&!(d!=10)
    &&!(a==10)&&b!=10&&!(c>10)&&!(d<10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&!(d!=10)
    &&!(a==10)&&b!=10&&!(c>10)&&d<10&&!(a<=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&!(d!=10)
    &&!(a==10)&&b!=10&&!(c>10)&&d<10&&a<=10&&!(b
    >=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&!(d!=10)
    &&!(a==10)&&b!=10&&!(c>10)&&d<10&&a<=10&&b>=10)
    );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&!(d!=10)
    &&!(a==10)&&b!=10&&c>10&&!(a<=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&!(d!=10)
    &&!(a==10)&&b!=10&&c>10&&a<=10&&!(b>=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&!(d!=10)
    &&!(a==10)&&b!=10&&c>10&&a<=10&&b>=10) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&!(d!=10)&&a
    ==10&&!(c>10)&&!(d<10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&!(d!=10)&&a
    ==10&&!(c>10)&&d<10&&!(a<=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&!(d!=10)&&a
    ==10&&!(c>10)&&d<10&&a<=10&&!(b>=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&!(d!=10)&&a
    ==10&&!(c>10)&&d<10&&a<=10&&b>=10) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&d!=10&&!(a
    <=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&d!=10&&a
    <=10&&!(b>=10)) );
__CPROVER_cover( !(a>10&&!(b<10)&&c==10&&d!=10&&a
    <=10&&b>=10) );
__CPROVER_cover( !(a>10&&b<10&&!(a<=10)) );
__CPROVER_cover( !(a>10&&b<10&&a<=10&&!(b>=10)) );
__CPROVER_cover( !(a>10&&b<10&&a<=10&&b>=10) );
if (((((  a > 10) && (b < 10)) || ((c == 10) && (d !=
    10))) || (((a == 10) || (b != 10)) && ((c > 10)
    || ( d < 10)))) && ((a <= 10) && (b >= 10)))
  print("Predicate 10");

...
if (((((  a > 10) && (b < 10)) || ((c == 10) && (d !=
    10))) || (((a == 10) || (b != 10)) && ((c > 10)
    || ( d < 10)))) && (((a <= 10) && (b >= 10)) ||
    (c <= 10)))
  print("Predicate 11");

...
if (((((  a > 10) && (b < 10)) || ((c == 10) && (d !=
    10))) || (((a == 10) || (b != 10)) && ((c > 10)
    || ( d < 10)))) && (((a <= 10) && (b >= 10)) ||
    ((c <= 10) && (d >= 10))))
  print("Predicate 12");

...
if (((((((  a > 10) && (b < 10)) || ((c == 10) && (d
    != 10))) || (((a == 10) || (b != 10)) && ((c >
    10) || ( d < 10)))) && (((a <= 10) && (b >= 10))
    || ((c <= 10) && (d >= 10))))) || (a != 10))
  print("Predicate 13");

...
...
...
...

if (((((((  a > 10) && (b < 10)) || ((c == 10) && (d
    != 10))) || (((a == 10) || (b != 10)) && ((c >
    10) || ( d < 10)))) && (((a <= 10) && (b >= 10))
    || ((c <= 10) && (d >= 10))))) || ((((a != 10)
    && (b != 10)) || ((c != 10) && (d != 10))) &&
    (((  a > 10) && (b < 10)) || (c == 10))))
  print("Predicate 19");

...
if (((((((  a > 10) && (b < 10)) || ((c == 10) && (d
    != 10))) || (((a == 10) || (b != 10)) && ((c >
    10) || ( d < 10)))) && (((a <= 10) && (b >= 10))
    || ((c <= 10) && (d >= 10))))) || ((((a != 10)
    && (b != 10)) || ((c != 10) && (d != 10))) &&
    (((  a > 10) && (b < 10)) || ((c == 10) && (d !=
    10)))))
  print("Predicate 20");
return 0;
}
```

Listing 5. This program is a sample program to show SC-MCC sequences for each predicate of size 1 to 6 as presented in Table III; and of size 7 and 10 as shown in Table IV.