Modified Condition/Decision Coverage in the GNU Compiler Collection

Jørgen Kvalsvik
j@lambda.is

January 2, 2025

Abstract

We describe the implementation of the masking Modified Condition/Decision Coverage (MC/DC) support in GCC 14, a powerful structural coverage metric with wide industry adoption for safety critical applications. By analyzing the structure of Boolean expressions with Binary Decision Diagrams we can observe the key property of MC/DC, the power to independently affect the outcome, and map to the edges of the Control Flow Graph. This mapping can be translated to a few bitwise instructions and enables GCC to instrument programs to efficiently observe and record when conditions have been taken and have an independent effect on the outcome of a decision. By analyzing the BDD rather than the program syntax, GCC can measure MC/DC for almost all of its languages with a single language-agnostic implementation, including support for C, C++, D, and Rust.

1 Introduction

In this paper we describe the algorithms designed and techniques used in the masking MC/DC support written by the author for the GNU Compiler Collection (GCC). Modified Condition/Decision Coverage (MC/DC) [7] is a code coverage metric that aims to improve program quality by requiring that all basic conditions are shown to have an *independent effect* on the outcome. It has long been understood that MC/DC is a useful criterion for ensuring high quality software, notably being mandated by standards such as DO-178 and ISO26262 for software in safety critical systems where malfunction would put lives at risk. Development in this space has been slow for a few decades; the topic of structural coverage and MC/DC received considerable attention around the turn of the century [8, 7, 4, 6], but at the time the analysis was largely manual, a process which is slow and error prone. The next decade saw development of automated instrumentation for analysis [13, 14, 5] and the rise of tooling like VectorCAST and LDRA which relies on source code instrumentation. GCC has supported statement and branch coverage since the late 1990s, with GCC the 2.95 manual [12] describing the gcov tool and coverage. 2019 – 2024 saw a dramatic shift in the tooling space, with multiple compiler vendors supporting MC/DC natively; The Green Hills compiler got native support for MC/DC in 2019 [11], and Clang got support for up-to 6 conditions MC/DC for C and C++ in 2023 [10], a restriction which was relaxed in 2024 [9]. Support for MC/DC was introduced in GCC 14, released in 2024. GCC has always implemented coverage as object code coverage [2, 5] (as opposed to source code coverage), and the MC/DC supports builds on the object coverage framework by analyzing the Control Flow Graph (CFG). This is different from the abstract syntax tree driven approaches like the Whalen et al., Green Hills, and Clang implementations. There is also interest in measuring MC/DC of Rust programs [16], which is supported by the approach described in this paper as the CFG analysis is unaware of the programming language.

MC/DC is satisfied if:

- every entry and exit point has been invoked at least once
- every decision in the program has taken all possible outcomes at least once
- every basic condition has taken on all possible outcomes at least once, and
- each basic condition has been shown to independently affect the decision's outcome

There are a few variations on the metric; the most significant ones are *unique cause* MC/DC and *masking* MC/DC, both described in detail by Hayhurst et al. [7]. Unspecified, it usually refers to unique cause MC/DC, where only one condition can vary between two test input vectors to demonstrate independent decisive power. Masking MC/DC relaxes this requirement and permits more than one condition to change between test inputs if they cannot influence the outcome. Chilenski [4] demonstrated they are generally equally effective at finding defects, and that *masking* MC/DC accepts the largest set of test inputs that achieves coverage; for N conditions, unique cause MC/DC requires at least $N+1$ test cases, while masking MC/DC requires $\lceil 2\sqrt{N} \rceil$ test cases. In this paper, unless otherwise specified, MC/DC means *masking* MC/DC.

2 Binary Decision Diagrams

Boolean functions have a canonical representation as a reduced ordered binary decision diagram (BDD) [3] where a path from the root to a *terminal* vertex corresponds to an input vector $(x_1, \ldots, x_n)$, and the vertices correspond to the evaluation of a basic condition. Reduced, ordered binary decision diagrams is what is usually meant by BDD, and all uses of BDD in this paper means reduced ordered binary decision diagram. Andersen [1] is a good introduction. A BDD is *ordered* when all paths through the variables respect the linear order $x_1 < x_2 < \ldots < x_n$ and *reduced* when there are no redundant tests of variables, and no two distinct vertices have the same variable name and successors. Short circuiting logic has a natural expression in BDDs as shortcut edges to deeper levels, e.g. the edge $(x_1, 1)$ in Figure 2a. The leaves of a BDD, called the *terminals* or *decisions*, are denoted with the literals 0 and 1. The internal vertices, called the *non-terminals*, correspond to a basic condition in the Boolean expression and have exactly 2 successors, the then and else branches. For any BDD there is a path from every x_i to every x_k where $i < k$, and all paths end in one of the two terminals.

Condition coverage is defined as every condition in a Boolean function (decision) having taken all possible outcomes at least once [7]. Since vertices in the BDD correspond to the evaluation of the basic conditions, and edges the outcomes of the basic conditions, we can determine condition coverage by recording the paths taken through the BDD during execution; coverage is achieved when every edge has been taken at least once. In the context of object code coverage, condition coverage is sometimes called *edge coverage*.

The key property of MC/DC is the *independence criterion*. For MC/DC, recording the vertices as they are visited is not sufficient for determining coverage as some vertices may be visited without independently affecting the decision due to the *masking effect*. A condition is *masked* if changing its value while keeping the other inputs fixed does not change the decision. This is intuitive for short circuiting - since a short circuited condition is not even evaluated, it cannot affect the decision. Short circuiting is determined by the left operand and operator; $(1 \vee x)$ and $(0 \wedge x)$. Boolean operators are commutative, $(x_1 \vee x_2) \Leftrightarrow (x_2 \vee x_1)$, so the inability to affect on the decision must also apply for $(x \vee 1)$ and $(x \wedge 0)$, i.e. the right operand *masks* the left operand. This can be seen in the truth table in Figure 1a; for $(x_1 \vee x_2)$ the decision can be fully determined by the left operand $x_1 = 1$ (rows 3, 4) and the x_2 does not affect the decision, and likewise for $(x_1 \wedge x_2)$ where $x_1 = 0$ (rows 1, 2). For $(x_1 \vee 1)$ (rows 2, 4) and $(x_1 \wedge 0)$ (rows 1, 3), the left operand has no effect on the decision. Figure 2 shows the same function as a BDD. Simply put, conditions are masked if they would be short circuited in the reverse-order evaluation of the Boolean function; if (x_i, x) short circuits x_k, then (x_k, x) masks x_i.

x_1	x_2	$\vee$	$\wedge$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

(a)

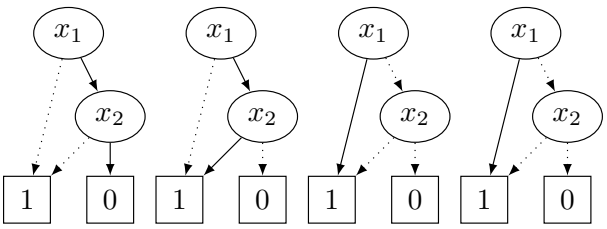
x_1	x_2	$\vee$
0	0	0
0	1	1
1	*	1
1	*	1

(b)

x_2	x_1	$\vee$
0	0	0
1	*	1
0	1	1
1	*	1

(c)

Figure 1: a is the truth table for the Boolean operators. b and c are the truth tables for $\vee$ in left-to-right and right-to-left evaluation order with conditions short circuited (*).



(a) 0 0 (b) 0 1 (c) 1 0 (d) 1 1

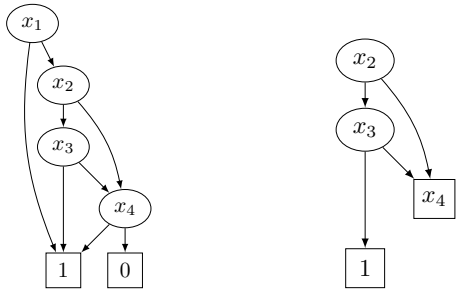
Figure 2: $(x_1 \vee x_2)$ as a BDD. The regular edges make up the path for the given input vector. b demonstrates masking; changing x_1 does not change the decision (1).

3 Finding the masking table

In this section we describe an algorithm for computing the masking table from the structure of a Boolean function by analyzing the corresponding BDD. BDDs were used by Comar et al. [5] to show that when the BDD is a tree then edge coverage implies MC/DC, which GTD GmbH used to implement an MC/DC tool [15] that checks and requires that all Boolean expressions are tree-like BDDs. This algorithm only relies on the BDD being reduced and ordered, and works for arbitrary Boolean functions.

Short circuiting and masking can be understood in the BDD as different paths to the same terminal. Subexpressions preserve local short circuiting and masking, seen as multiple paths to a *pseudo-terminal*, a vertex that would be a terminal if the expression was independent. Multiple paths to the same pseudo-terminal means it will have multiple incoming edges (in-degree ≥ 2); the short circuiting edges and the one non-short circuiting edge. Consider the Boolean function $(x_1 \vee (x_2 \wedge x_3) \vee x_4)$, and the BDD in Figure 3a. $(x_2 \wedge x_3)$ (Figure 3b) is *embedded* in the BDD, and what would be the terminals in $(x_2 \wedge x_3)$ become the pseudo-terminals x_4 and 1, both with an in-degree ≥ 2. The goal is to determine which terms are masked upon taking an edge, which corresponds to a specific outcome of evaluating a condition. The specific Boolean operator is generally of little significance for the analysis, in contrast to the abstract syntax tree approach of Whalen et al. [13], as the BDDs for the Boolean functions with inverted operands are isomorphic. This is explained by De Morgan's laws, $\neg(P \vee Q) \Leftrightarrow (\neg P) \wedge (\neg Q)$ and $\neg(P \wedge Q) \Leftrightarrow (\neg P) \vee (\neg Q)$, and in the BDD by inverting all the comparisons and swapping the terminals. Negation is built into the evaluation of a basic condition (contrast `if (v != 0)` to `if (v == 0)`). As a consequence, we only need to be concerned with the *shape* of the BDD.

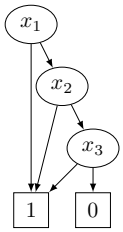
Consider the Boolean expression B on a normalized form $(A_1 \vee A_2 \vee \cdots \vee A_n)$ or $(A_1 \wedge A_2 \wedge \cdots \wedge A_n)$ where A_k is either a basic condition or a possibly nested complex Boolean function combined with a *different* operator. Let B_T be the short circuit terminal of B, 1 if $\vee$, or 0 if $\wedge$, which will have an in-degree



(a) $(x_1 \vee (x_2 \wedge x_3) \vee x_4)$ (b) $(x_2 \wedge x_3)$

Figure 3: A Boolean function (a) and its subexpression (b). The terminals x_4 and 1 in (b) become pseudo-terminals in (a). The masking effects in subexpressions will be preserved in the superexpression.

≥ 2. We want to find the *masking table*, where the notation $(u, v) \sim \{ x_k, x_{k+1}, \ldots, x_n \}$ means that taking the edge (u, v) mask the conditions $x_k, x_{k+1}, \ldots, x_n$. A_{k+1} can only be entered through A_k, which follows directly the ordering property. It follows that all paths through A_k must go through A_{k+1} or to B_T. For example, let A'_k be the result of evaluating A_k on its own. Precomputing would not change the truth table of B, and since A'_k is a basic condition it has exactly two successors, one being the short circuiting edge to B_T, and the other the evaluation of the right operand A_{k+1}. An example can be seen in Figure 5 where the substitution $x_1x_2 = (x_1 \wedge x_2)$ is succeeded by 1 and x_3, which corresponds to the edges $(x_1, x_3), (x_2, x_3), (x_2, 1)$. Finally, the last condition of A_k *must* decide the outcome. By using these observations we can identify the vertices of A_k from B; given the edges (A_i, B_T) and (A_k, B_T) where $i < k$, (A_k, B_T) masks A_i. The problem now becomes finding the subset of vertices where all paths go through either (1) the edge (A_i, B_T), where the source vertex is last term of A_i, or (2) an edge to the term of A_{i+1}. The boundaries of A_k can be found with the function $P(x) = \{(x_e, x_n, x_m) \mid x_n \in \text{preds}(x), x_m \in \text{preds}(x), x_e \in \text{succs}(x) - \{ x_n \}, n < m\}$; for each vertex x with an in-degree ≥ 2, consider the Cartesian product of the predecessors $\text{preds}(x)^2$ where the pairs (x_n, x_m) are ordered so that $n < m$ and duplicates are removed, and x_e is the non-x successor of x_n. Note that $\text{succs}(A_k) = \{ x, x_e \}$. Masked conditions can found from the triple (x_e, x_n, x_m): if all paths from a vertex x_i where $i <= n$ go through either x_e or x it is masked when the edge (x_m, x) is taken.

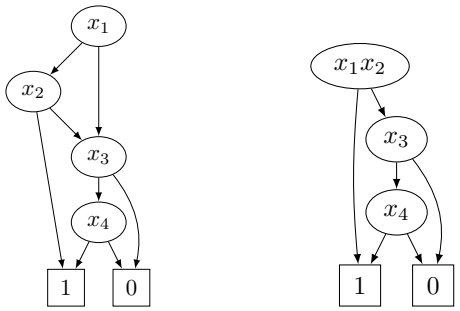


(a)

	x_1	x_2	x_3
$x_1 = 1$		*	*
$x_2 = 1$	-		*
$x_3 = 1$	-	-	

(b)

Figure 4: $(x_1 \vee x_2 \vee x_3)$ and the short circuited (*) and masked (-) terms when a basic condition takes on 1.



(a) $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$ (b) $x_1x_2 \vee (x_3 \wedge x_4)$

Figure 5: Two equivalent Boolean functions where a subexpression is computed outside the function. All edges from the subgraph $(x_2 \wedge x_2)$ go to x_3 and 1, which are the successors x_1x_2.

We now have a an algorithm for computing the masking table from the BDD:

1. For each vertex x with in-degree ≥ 2, find the triples $\{ (x_e, x_n, x_m), \ldots \} = P(x)$.
2. For each triple, remove $\text{succs}(x_n) = \{ x, x_e \}$ from the BDD; these are the pseudo-terminals of A_k.
3. Remove and collect all vertices made into leaves until no more vertices can be removed; the collected vertices are A_k.

4. Add the collected vertices to the masking table; $M(x_m, x) + A_k$.

An example run of the algorithm can be seen in Figure 6. A_k may be an arbitrarily complex BDD, but since all paths must go through either x or x_e and removing leaves is equivalent to inverting the edges and collecting all paths from x and x_e to the first condition in A_k, e.g. with a breadth-first search. The search will stop when it reaches the root of the BDD or the vertex x_p where there is a path from x_p to the other pseudo-terminal of A_k. Note that algorithm may collect multiple A_i where $i < k$ that short circuit to the same pseudo-terminal x. The simplest example is $(x_1 \vee x_2 \vee x_3)$, as seen in Figure 4. For this function, $P(1)$ contains $(x_e, x_n, x_m) = (x_3, x_2, x_3)$, which means 1 and x_3 should be removed from the graph. This makes x_2 a leaf, which when removed makes x_1 a leaf and yields the masking table entry $(x_3, 1) \sim \{ x_1, x_2 \}$. The correctness is not affected as the masking table maps edges to sets with duplicates removed, and the inefficiency can be addressed with memoization. It is possible that there is an even faster approach for this, either by limiting the search by going through the other pseudo-terminal, or by removing all x_e at once, but these approaches were not explored.

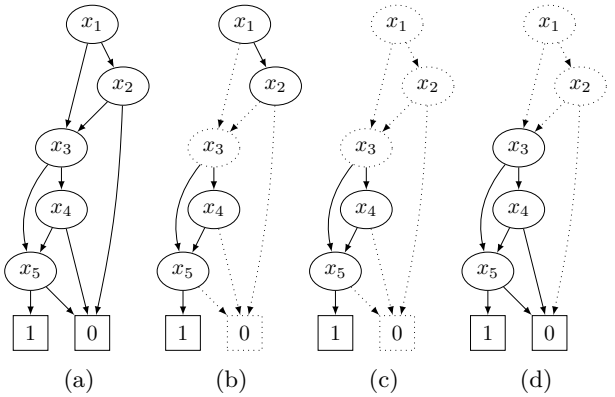


Figure 6: $((x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge x_5)$ for the edge $(x_4, 0)$ and $(x_e, x_n, x_m) = (x_3, x_2, x_4)$. In the first step (b) $\text{succs}(x_n) = \{ x_3, 0 \}$ are removed (dotted). Repeatedly removing leaves gives (c). (d) shows the collected conditions; $(x_4, 0) \sim \{ x_1, x_2 \}$.

4 Instrumenting programs for measuring MC/DC

If the control flow graph (CFG) is carefully constructed to directly model the evaluation of conditional expressions as BDDs this algorithm can be run directly on the CFG. This is a departure from the approaches of Whalen et al. [13] and Sagnik [11] who derive the masking table from the abstract syntax tree (AST). By doing the analysis directly on the CFG the analysis becomes language agnostic, or requires minimal information from the front-end, and can be used with several languages. For this work the compilers for C, C++, D, and Rust (which is experimental in GCC 14) were all shown capable to instrument for MC/DC. Note that the Go front-end does not construct a CFG isomorphic to the canonical BDD for Boolean expressions and consequently cannot measure MC/DC. Other languages supported by GCC (Ada, Fortran, Objective-C, Modula-2) were not tested, but might work. The only modification other than the CFG analysis was in a lowering pass, where complex Boolean expressions were transformed to If-then-else normal form (INF), which was extended with an identifier that maps a basic condition to its Boolean expression. INF is a Boolean expression built entirely from the if-then-else operator and the constants 0 and 1 such that all tests are performed on variables. Any Boolean function is expressible in INF [1] which can be represented in graph form as a *decision tree* and in a refined form a BDD. The BDD is a natural way for compilers to implement Boolean expression evaluation as it encodes both evaluation order and short circuiting with no redundant tests.

The approach is similar to measuring condition coverage as described in Section 2 by recording the paths through the BDD during execution. However, in MC/DC an edge may be taken without having an independent effect on the decision for that input vector. The masking table is a function $m : E \to 2^{\mathcal{C}}$ where $\mathcal{C}$ is the set of basic conditions. Then $m : e \mapsto C \subseteq \mathcal{C}$ maps e to a possibly empty set of basic conditions that do not have an effect on the decision.

Edge	Masked conditions	Bitmask
(x_2, x_3)	x_1	10000
(x_4, x_5)	x_3	00100
$(x_4, 0)$	x_1, x_2	11000
$(x_5, 0)$	x_1, x_2, x_3, x_4	11110

Figure 7: Masking table m for $((x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge x_5)$ (Figure 6). The third column is the bitmask representation of the masked conditions.

When the program reaches a terminal through a path E, the covered condition outcomes will be the edges $E - \{ x : m(E) \}$, that is, with the contribution of the masked conditions voided and the remaining condition outcomes (edges) shown to have an independent effect. Coverage is achieved when all edges have been marked at least once. For example, given $((x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge x_5)$ in Figure 6, the masking table m in Figure 7, and the input vector (0 1 0 0 1). The edges taken are (x_1, x_2) (x_2, x_3) (x_3, x_4) $(x_4, 0)$. Applying m to the edges gives the $\{ x_1, x_2 \}$ which means the covered condition outcomes are $x_3 = 0, x_4 = 0$. For condition coverage the covered outcomes would be $x_1 = 0,\ x_2 = 1,\ x_3 = 0,\ x_4 = 0$.

As seen in Section 3 the masking table m only depends on the structure of the Boolean function, and can be computed offline, i.e. during compilation. There is a bijection between the position (index) of the basic conditions in a Boolean function and the natural numbers. By limiting the number of conditions, paths can be represented and recorded using fixed-size bitsets and bitwise operations. For example, the input vector (0 1 0 0 1) and path (x_1, x_2) (x_2, x_3) (x_3, x_4) $(x_4, 0)$ can be represented as the two bitsets $f = 10110, t = 01000$ where bit $t[n] = 1$ if the nth condition was true, $f[n] = 1$ for false. Note that $t[5] = f[5] = 0$ because x_5 was short circuited. Edges are recorded and masked by with a few bitwise instructions before performing the conditional jump, as seen in Figure 8. Upon taking an edge to a terminal both bitsets are flushed to global bitsets, what Whalen et al. [13] call *independence arrays*. Coverage is achieved when all the bits are set in both counters. Sagnik [11] uses a similar approach with a single bitset, using odd/even indices for the true/false

```
if (x_n != 0)
  then: goto _then_n
  else: goto _else_n

if (x_n != 0)
  then:
    _t &= ~m[n, 1]
    _f &= ~m[n, 1]
    _t |= (1 << n)
    goto _then_n
  else:
    _t &= ~m[n, 0]
    _f &= ~m[n, 0]
    _f |= (1 << n)
    goto _else_n
```

Figure 8: The evaluation of a basic condition in INF with and without instrumentation. `m[n, 0]` looks up the bitmask for the false outcome of the nth condition in m. Operators use the semantics from C: `&=` is bitwise-and, `|=` is bitwise-or, `~` inverts all bits, and `1 << n` is a bitmask where only the nth bit is set.

outcomes. Limiting the number of conditions and using fixed-size bitsets is a purely practical choice, and the approach would work just as well with variable sized bitsets. In GCC 14, the bitset is 32 or 64 depending on the target platform and configuration. Boolean functions with more than 32 conditions are very uncommon, 64 even more so, so this limitation rarely becomes an issue.

5 Summary and future work

Interpreting the Control Flow Graph (CFG) directly as a Binary Decision Diagram (BDD) is an effective way of inferring the masking- and independence properties of Boolean expressions. The novel implementation of Modified Condition/Decision Coverage (MC/DC) in the GNU Compiler Collection (GCC) does not build on syntax, but rather analyzes the BDD and builds a table that maps decisions to efficient operations on bitsets. These operations are

inserted into the instrumented program and records when a condition is evaluated and shown to have an independent effect on the outcome, per the masking MC/DC criterion. The BDD analysis is language independent unlike the syntax based approach taken by Green Hills [11], Clang [10], and Whalen et al. [13], and the implementation can be shared between C, C++, D, Rust, and any language where the compiler front-end generates a BDD-like CFG for Boolean expressions.

The GCC implementation is limited by the size of the bitset, typically 64 bits (one bit per basic condition). This constraint could be relaxed, either by tuning the algorithm or by targeting variable-sized bitsets. The approach also relies on the compiler front-end encoding Boolean expressions as BDDs, which the Go front-end notably does not, and future work may either detect whether the CFG is a BDD. Future work may also extend GCC to support the other forms of MC/DC, in particular unique cause MC/DC.

References

[1] Henrik Reif Andersen. *An Introduction to Binary Decision Diagrams*. https://www.inf.ed.ac.uk/teaching/courses/fv/files/andersen-bdd.pdf. 1999.

[2] Matteo Bordin et al. "Object and Source Coverage for Critical Applications with the COUVERTURE Open Analysis Framework". In: (Jan. 2010).

[3] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* C-35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[4] John Chilenski. "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion". In: (Apr. 2001). URL: https://rosap.ntl.bts.gov/view/dot/42764/dot_42764_DS1.pdf.

[5] Cyrille Comar et al. "Formalization and Comparison of MCDC and Object Branch Coverage Criteria". In: Feb. 2012.

[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, 1B6/1–1B6/7 vol.1. DOI: 10.1109/DASC.2000.886883.

[7] Kelly J. Hayhurst et al. *A Practical Tutorial on Modified Condition/Decision Coverage*. Tech. rep. 20010057789. May 2001. URL: https://ntrs.nasa.gov/citations/20010057789.

[8] Michael Dale Herring. *Testing Safety Critical Software*. Florida Institute of Technology, 1997.

[9] Takumi Nakamura. *New algorithm and file format for MC/DC*. https://discourse.llvm.org/t/rfc-coverage-new-algorithm-and-file-format-for-mc-dc/76798/1. 2024.

[10] Alan Phipps. *MC/DC in LLVM Source-Based Code Coverage: clang*. https://reviews.llvm.org/D138849. 2022.

[11] Saha Sagnik. "Adding Support for MC/DC instrumentation in the Green Hills C/C++ compiler". MA thesis. Department of Electrical Engineering and Computer Science: Massachusetts Institute of Technology, 2019.

[12] Richard M. Stallman et al. *GNU Compiler Collection 2.95 Manual*. https://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.html. 1999.

[13] Michael W. Whalen, Mats P.E. Heimdahl, and Ian J. De Silva. *Efficient Test Coverage Measurement for MC/DC*. Tech. rep. 2013. URL: https://hdl.handle.net/11299/217382.

[14] Michael W. Whalen et al. "A Flexible and Non-intrusive Approach for Computing Complex Structural Coverage Metrics". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 506–516. DOI: 10.1109/ICSE.2015.68.

[15] Thomas Wucher and Andoni Arregui. *MC/DC for Space - A new Approach to Ensure MC/DC Structural Coverage with Exclusively Open Source Tools*. https://gtd-gmbh.gitlab.io/mcdc-checker/mcdc-checker/MCDC_for_

Space_ESA_Software_PA_Workshop_2021.pdf. Accessed 2023-08-06. 2021.

[16] Wanja Zaeske et al. *Towards Modified Condition/Decision Coverage of Rust*. 2024. arXiv: 2409.08708 [cs.SE]. URL: https://arxiv.org/abs/2409.08708.