Annotating Control-Flow Graphs for Formalized Test Coverage Criteria

Sean Kauffman
Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
Email: sean.k@queensu.ca

Carlos Moreno
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
Email: cmoreno@uwaterloo.ca

Sebastian Fischmeister
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
Email: sfischme@uwaterloo.ca

Abstract—Control flow coverage criteria are an important part of the process of qualifying embedded software for safety-critical systems. Criteria such as modified condition/decision coverage (MC/DC) as defined by DO-178B are used by regulators to judge the adequacy of testing and by QA engineers to design tests when full path coverage is impossible.

Despite their importance, these coverage criteria are often misunderstood. One problem is that their definitions are typically written in natural language specification documents, making them imprecise. Other works have proposed formal definitions using binary predicate logic, but these definitions are difficult to apply to the analysis of real programs. Control-Flow Graphs (CFGs) are the most common model for analyzing program logic in compilers, and seem to be a good fit for defining and analyzing coverage criteria. However, CFGs discard the explicit concept of a decision, making their use for this task seem impossible.

In this paper, we show how to annotate a CFG with decision information inferred from the graph itself. We call this annotated model a Control-Flow Decision Graph (CFDG) and we use it to formally define several common coverage criteria. We have implemented our algorithms in a tool which we show can be applied to automatically annotate CFGs output from popular compilers.

I. INTRODUCTION

Control flow coverage criteria are an important part in the process of qualifying embedded software for safety-critical systems. Standards such as DO-178B/C in aerospace systems and ISO-26262 in the automotive domain prescribe modified condition/decision coverage (MC/DC) as a prerequisite for software test design. Coverage criteria are used during software qualification to guide test design and demonstrate that no part of the code base is overlooked.

Despite their importance, coverage criteria are often misunderstood. One problem is that their definitions are typically written in natural language specification documents, making them imprecise. Other works have proposed formal definitions using binary predicate logic or Z notation, but these definitions can be difficult to apply to the analysis of real programs. One of the most common representations of program control flow is the Control-Flow Graph (CFG), which is used by most compilers for optimization and analysis, and is easy to represent visually. As such, modeling coverage criteria using CFGs seems like a natural fit to improve user comprehension and simplify automated measurement.

Unfortunately, CFGs do not explicitly retain a concept that is crucial to measuring control-flow coverage criteria: *decisions*. Decisions represent potentially complex branch conditions in programs, and they may be represented as multiple vertices in a CFG. To define coverage criteria using a CFG model and represent it for easier human and machine comprehension, decisions need to be made explicit.

In this paper, we propose an intermediate representation of program logic that extends CFGs to represent decisions explicitly called Control-Flow Decision Graphs (CFDGs). We show how to automatically transform a CFG into a CFDG and we use this structure to formalize common control flow test coverage criteria. Our method is implemented as a tool that can be used to annotate CFGs output by the popular GCC and Clang compilers in *dot* format, allowing them to be visually compared.

The remainder of the paper is structured as follows: Section II discusses related work. Section III defines terms and notation used in the paper. Section IV presents the context of the problem and why it is important. Section V defines a Control-Flow Decision Graph. Section VI uses the model to formally define test coverage criteria. Section VII introduces a tool to annotate CFGs output by popular compilers. Section VIII concludes the paper.

II. RELATED WORK

Previous works have formalized coverage criteria requirements for comparison purposes. Comar et al. formalized MC/DC coverage using Branch Decision Diagrams to compare with Object Branch Coverage [1]. Vilkomir and Bowen used Z notation to formalize and compare traditional test coverage criteria including MC/DC in [2], [3], and added RC/DC in [4]. They applied the same techniques to formalize other regulatory requirements in the Z notation in [5]. Woodward and Hennell compared JJ-Path, or LCSAJ, coverage to MC/DC and defined when JJ-Path subsumes MC/DC [6]. Kirner proposed a formal model for MC/DC to be used to show that transformation and compilation do not result in coverage violations [7]. Ammann et al. formalized a comprehensive set of criteria using boolean predicate logic [8]. Kapoor and Bowen used boolean logic to compare the fault detection effectiveness of MC/DC and

RC/DC [9]. Kosmatov et al. formalized data-oriented boundary testing heuristics for a new family of model-based coverage criteria [10].

Other works have formalized coverage criteria for automatic test case generation. In 1994, Jasper et al. proposed using their Ada framework (TSDT) to generate test cases that met MC/DC requirements [11]. They modeled the criterion using two functions on tuples of boolean expressions and vectors. Ghani and Clark proposed using search-based test generation to meet MC/DC requirements [12]. They used an AST-based emitter to export decisions in disjunctive normal form (DNF), then used them in cost functions for simulated annealing. Offutt, Xiong and Liu introduced a method of test code generation coverage criteria such as FPC from SCR condition tables or UML state charts [13].

III. DEFINITIONS AND NOTATION

In this section, we present definitions of terms and notation used throughout the paper.

A. Conditions and Decisions

A *condition* is an expression defined on a Boolean ($\mathbb{B}$) algebra with elements *true* and *false*. A *decision* is a function from two or more conditions and binary operators to a two-valued boolean algebra $\mathbb{B}$. The operators in a decision may be any of the following: normal AND ($\wedge$), short-circuit AND (&&), normal OR ($\vee$), short-circuit OR ($\|$), and XOR ($\oplus$). For example, the decision ($a \oplus b$) combines two conditions (reading the values of symbols a and b) with an XOR.

A decision is equivalent to an `if` statement in imperative programming, or any equivalent structures that make decisions about the control flow of the program such as a `while` statement.

B. Test Coverage

An *input* is a mapping from a symbol in a program to a concrete, true or false, value. One input can map to one or more conditions. For example, the decision ($a \oplus b \vee a$) combines three conditions, two of which a are mapped to the same input and one which b is mapped to a second input.

A *test* $t \in \mathbb{B}^*$ is a finite sequence of Boolean values mapping the inputs of a program to concrete values. Note that we assume a fixed order for program symbols, permitting the omission of explicit symbol mapping. A *test suite* $T \in 2^{\mathbb{B}^*}$ is a set of tests.

C. Control Flow Graph

A *control flow graph* (CFG) of a program is a connected, directed graph $G = (V, E)$ where each vertex $v \in V$ has outdegree ≤ 2 and represents a program point and each edge $(t, h) \in E$ represents the possibility that execution after the end of the tail (t) may continue with the beginning of the head (h) [14]. The domain of vertices is given by $\mathcal{V}$, so the domain of CFGs is $\mathcal{G} = \mathcal{V} \times (\mathcal{V} \times \mathcal{V})$.

Each edge represents possible program flow and each vertex (a program point) is equivalent to a single statement. A CFG has a single *entry vertex*, given by the function *entry* : $\mathcal{G} \to \mathcal{V}$, which is a special vertex with indegree 0 from which the program flow begins. A CFG has a set of one-or-more *exit vertices*, given by the function *exit* : $\mathcal{G} \to \mathcal{V}$, which are special vertices with outdegree 0 at which the program flow ends. A vertex in a CFG *dominates* another vertex if all paths from the entry vertex to the dominated vertex must pass through the dominating vertex. The entry vertex dominates every other vertex in the CFG. A vertex has successor vertices given by the function *successors* : $\mathcal{V} \to 2^{\mathcal{V}}$ defined as $successors(v) = \{s \in V : (v, s) \in E\}$.

D. Program Runs

We define *Run* : $\mathcal{G} \times \mathbb{B}^* \to E^* \cup E^\omega$ to be a function from a CFG and a test to the sequence of edges in the CFG $(t, h) \in E$ that are traversed. The sequence, called a (lowercase) *run*, can be finite or infinite. Runs returned by *Run* must begin with the entry node and, if finite, end with an exit node.

A run may walk each edge zero or more times. Walking an edge zero times means that the condition represented by the vertex before the edge never resulted in a choice of that edge when executing the program with the given input. Edges may be walked more than once due to loops in the program. A run may be a finite sequence in the case of a terminating program or an infinite sequence in the case of a non-terminating program.

Listing 1 shows a program with a single decision containing a short-circuit AND operator, and two conditions, each with a separate input. Figure 1 shows three copies of the CFG for this program. Each of the graphs contains a run of the program, with the executed edges marked as dashed lines and highlighted in red. Below each graph, the state of both inputs is given.

Listing 1: A program with one decision and two conditions

```
x = 0;

if (a && b) {
  x = 1;
}

return x;
```

For example, in Figure 1b, a is true, so the vertex with b is reached. Since b is false, the vertex with $x = 1$ is not reached and the program instead continues to the final statement.

IV. MOTIVATION

The most widely used and accepted model for program control flow is the CFG. CFGs are relatively simple but contain all of the necessary information to analyze program control flow properties. However, CFGs do not explicitly define the
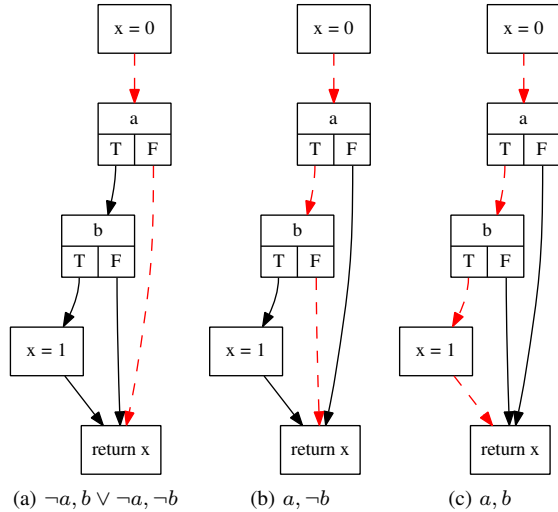
Fig. 1: Possible runs of the program in Listing 1

concept of a decision, making them difficult to use to model coverage criteria.

Control flow test coverage criteria specify the control flow properties of the tests in a coverage. They are typically defined in natural language documents which are open to interpretation by engineers and regulators and are often misunderstood and misapplied. A natural fit for describing and analyzing these coverage criteria is a CFG, but many such criteria rely on the definition of decisions, which CFGs lack. For example, a commonly used criteria is *decision* coverage, which specifies that executing a test suite must result in the program taking all possible outcomes from every decision.

Prior methods of formalizing control flow test coverages have typically used boolean predicate logic to construct solvers to either test that a coverage meets a criterion or to automatically generate test coverages that do so. These efforts are useful in that they have formalized the criteria so that meeting them is unambiguous. However, these efforts do not facilitate finding other properties of test coverage criteria, or give users other methods to reason about them. We believe that modifying CFGs to carry decision coverage information provides more natural way to express formal coverage criteria and can help users to reason about those criteria in their test development.

One idea that has been proposed is to use object code coverage in place of other criteria when information about decisions is lacking. CFGs can even be inferred from dynamic analysis of compiled binaries [15]. While it is true that object code (i.e., the assembly-level output of a compiler) is closer to what the system will execute, object code discards information about the intent of the program that may be important and that is present in the source code. In particular, object code coverage does not change the MC/DC requirements [1]. The Federal Aviation Administration (FAA) allows the use of object code coverage only when it is shown to be equivalent to source code coverage [16]. This means that the coverage criteria must use source code, not object code, to define the notion of a decision.

V. CONTROL-FLOW DECISION GRAPHS

In this section, we construct a model that can be used to formally describe the control flow of a program and many control-flow test coverage criteria. We show how to infer decision information from a CFG to construct such a model. We later use this model to formalize the test coverage criteria in Section VI. Note that, throughout this section, we assume that conditions are represented by single vertices and that decisions contain no extra, interstitial vertices. We define an interstitial vertex, in this case, as a non-condition vertex that appears between condition vertices within a decision. In Section VII, we show that these assumptions do not always hold when using CFGs output by certain tools.

To model control flow through a decision in the DFG we can use the common practice from interprocedural analysis [17] of adding a subgraph to a CFG which usually represents a called function. The subgraph of the decision only has one entry edge, leading to one entry vertex which dominates the subgraph. That there is only one entry vertex is easy to see from the definition of a CFG and a decision. Incoming edges to a CFG vertex represent the possibility of control flow transitioning to the beginning of that vertex.

We define a *Control-Flow Decision Graph* (CFDG) as a CFG with an additional set of decision subgraphs that group vertices of the CFG. We create a CFDG from a CFG by subgraphs of conditions in the *shape* of decisions. The CFG is then represented by the CFDG = (G, D_s), where $G = (V, E)$ is the original CFG, and D_s is the set of decision subgraphs. Once these subgraphs are created we can reason about them independently, and understand their relations.

The idea of inferring decisions from CFGs comes from an observation of their shape. First, consider that any vertex in a CFG with outdegree 2 is a condition. All conditions are part of a decision (that may contain only one condition). Of the five Boolean operators we consider for decisions ($\wedge$, &&, $\vee$, ||, and $\oplus$), $\wedge$, $\vee$ and $\oplus$ only require one CFG vertex. This makes sense if one considers the effect that no short-circuiting has on evaluation - both sides of the operator must always be evaluated so there is no alternative control flow that would split the vertex (basic block). As such, we need only to consider multiple vertices for the short-circuit operators: && and ||.

Lemma 1 (Outdegree of Decision Subgraphs). *A decision can be represented as a control flow subgraph that has two successor vertices.*

Proof. The statement is proved by construction, observing that a decision involves a conditional expression that evaluates to a binary result, and control continues at one of two possible vertices, depending on this binary result. Thus, we can always construct a subgraph for a decision with exactly two successor vertices. □

Lemma 2 (Indegree of Decision Subgraphs). *A decision can be represented as a control flow subgraph which is dominated by a single entry vertex.*

Proof. We will prove the statement by induction on the number of conditions. The base case is a decision with one condition. The statement is trivially true in this case since the graph contains a single vertex representing the condition.

As induction hypothesis, we assume that the statement is true for decisions consisting of n conditions.

We will show the induction step by contradiction. Consider a decision with $n + 1$ conditions and assume, for the purpose of a contradiction, that it is not dominated by a single vertex.

The lexical rules we have established require us to express the decision as a condition combined via a binary operator with a decision with n conditions. If the graph for a decision with $n+1$ conditions is not dominated by a single vertex, then there must be an incoming edge to a vertex in the decision subgraph. The resulting structure cannot happen with natural conditional constructs, and would require a label right after the first condition (to be targeted by a `goto` statement), which is not allowed. $\square$

Theorem 1 (Shared Successors of Conditions in a Decision). *Assuming no interstitial vertices within a decision in a CFG, the subgraph that corresponds to a decision has one predecessor and two successors, and all conditions in that decision must share a successor with the subgraph.*

Proof. The first part of the statement follows directly from lemmas 1 and 2.

The second part can be shown by induction on the number of conditions. The base case is a decision with one condition. The statement is trivially true in this case since the decision subgraph contains a single vertex representing the condition, so they must share all successors.

As induction hypothesis, we assume that the statement is true for decisions consisting of n conditions.

We will show the induction step by contradiction. Assume a decision with $n + 1$ conditions such that the first condition shares no successors with the decision subgraph. Since the decision can only have two successors and the final condition's successors are those two, the first condition's successors must be vertices within the decision subgraph. Since, by definition, a condition has two successors, the two vertices within the decision must be different. This contradicts Lemma 2 that says the decision with n conditions must be dominated by a single vertex, as it would add an incoming edge to a vertex in the decision that does not pass through the entry vertex. $\square$

A. Algorithm

Figure 1 shows a CFG with one decision including an && operator. In the CFG, there are two decisions, a and b, that must be true to reach the vertex with $x = 1$. If either is *false*, the outgoing edge leads to the *same* vertex

containing return x. This same shape is true for any && decision; any condition that is *false* leads to the same successor vertex. Crucially, || decisions are shaped identically, except that the equal successor vertex is reached on a *true* condition. This leads to the algorithms below. Note that we overload the *successor* : $2^{\mathcal{V}} \rightarrow 2^{\mathcal{V}}$ function for sets of vertices that represent decision subgraphs, defined as $successor(D) = \{s \in V : (c, s) \in E \wedge c \in D\}$ for a CFDG $(G, V), D_s$.

The procedure for annotating a CFG to create a CFDG is given in Algorithms 1 and 2. The initial procedure is given in Algorithm 1, which takes a CFG as its input and returns the CFDG. The algorithm first sets up a data structure to ensure that vertices are only visited once, which is necessary in case of loops. It then sets up a data structure D_{map} that maps condition vertices to a set of vertices that represents a decision. Then, the algorithm iterates over these sets, calling *merge* on each one, so long as the vertex that maps to it has not yet been visited. Note that the parameters, especially *visited* and D_{map}, should be considered passed-by-reference.

Algorithm 2 works by recursing over subsequent decisions, merging each one reverse order and returning the successor vertices of the decision. On Line 2, the successors of the decision D_1 are iterated over. This should be understood as the union of all the successors to the vertices in D_1 (a set). If a successor has not been visited, it is checked to see if it is a condition. If it is, then its current decision is looked up in the map and is recursed on, getting the set of its successors from the return. Then, on Line 8, if the successors returned by the recursive call contain a shared successor with D_1 (not the current successor we iterated over), then the decisions are merged, and the map is updated for all contained vertices.

Algorithm 1 CFDG Creation Algorithm

1: **procedure** CREATECFDG((V, E))
2:      $visited \leftarrow \{v \mapsto false : v \in V\}$
3:      $D_{\mathrm{map}} \leftarrow \{v \mapsto \{v\} : v \in V \text{ . } outdegree(v) = 2\}$
4:      **for** $v \mapsto D_1 \in V$ **do**
5:          **if not** $visited(v)$ **then**
6:              $merge(\ D_1, D_{\mathrm{map}}, visited\ )$
7: **return** $(V, E), \{D_s : v \mapsto D_s \in D_{\mathrm{map}}\}$

B. Example of constructing a CFDG

Listing 2: One decision with three conditions

```
x = ((a && b) || c);
```

Consider the program in Listing 2, which contains one decision with a short-circuit AND as well as a short-circuit OR, and three conditions. Figure 2a shows the CFG for this program where the vertices representing the conditions are shown with rounded corners.

At the beginning of the algorithm, Algorithm 1 iterates over the conditions, forming the decision subgraphs D_i for $i \in \{1, 2, 3\}$, which are represented by black, solid-line boxes

Algorithm 2 Decision Merging Algorithm

```
 1: procedure MERGE( D₁, D_MAP, visited )
 2:     for s ∈ successors(D₁) do            ▷ s is a vertex
 3:         if not visited(s) then
 4:             visited(s) ← true
 5:             if outdegree(s) = 2 then     ▷ s is a condition
 6:                 D₂ ← D_map(s)
 7:                 S ← merge( D₂, D_map, visited )
 8:                 if successors(D₁) \ s ∩ S ≠ ∅ then
 9:                     D₁ ← D₁ ∪ D₂             ▷ merge!
10:                     for v ↦ _ ∈ D₁ do
11:                         D_map(v) ← D₁
12: return successors(D₁)
```
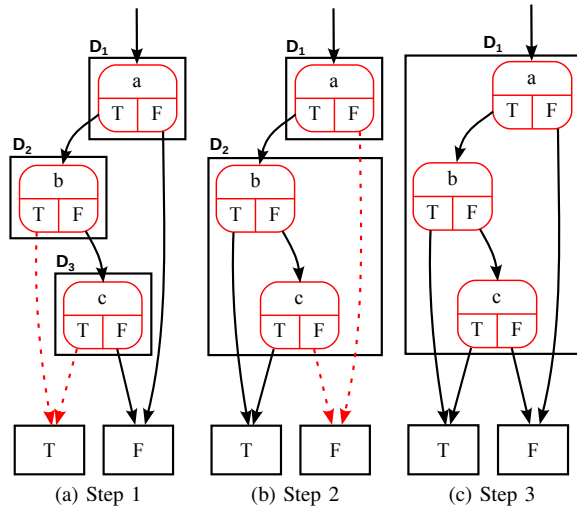

(a) Step 1   (b) Step 2   (c) Step 3

Fig. 2: Steps of Algorithms 1 and 2 on the CFG for Listing 2

around the condition vertices. The algorithm then calls *merge* on Decision D_1.

Algorithm 2 then recurses forward in the CFG, until it reaches the last decision D_3 containing the condition vertex, c. The recursive call to *merge* on D_3 (Line 7) returns the set of its successor vertices, $\{T, F\}$. This set is then compared to the successors of D_2, subtracting the entry vertex of D_3, c. The intersection of those two successor sets is non-empty, since the T vertex is shared by both sets. This is shown in Figure 2a, where the two edges pointing at the shared successor are highlighted in red and dashed. As a result of this shared successor, D_2 and D_3 are merged, and the recursive call to *merge* returns the successors of that merged decision, $\{T, F\}$. Like before, and as shown in Figure 2b, the successors of D_1 minus b are compared to $\{T, F\}$ with the intersection found non-empty. As such, D_1 and D_2 are merged, with the final decision shown in Figure 2c.

C. Correctness and Complexity

Algorithm 1 on its own is trivially correct, *assuming that all conditions are represented as single vertices in a CFG*, but we must argue for the correctness of Algorithm 2.

Theorem 2 (Soundness). *Algorithm 2 is sound.*

Proof. The proof of soundness for Algorithm is based on Theorem 1, which says that all conditions in a decision must share a successor with the decision itself. Algorithm 2 merges two decisions D_1 and D_2 when D_1's successors are the entry vertex of D_2 (there is only one, by Lemma 2) and one of the successors of D_2. This forms a new decision which is clearly sound, as it maintains the same successors, and still has only one entry vertex (that of D_1).   □

Theorem 3 (Completeness). *Assuming no interstitial vertices within a decision, Algorithm 2 is complete.*

Proof. By our assumption that every condition is represented as a single vertex, clearly Algorithm 1 will create decisions for every condition. So, need only show that every pair of decisions will be merged if they are really in one decision. It is clear that every decision is compared with its successor conditions to see if they belong in the same decision. For a decision to not be merged, it must not share a successor with its predecessor decision, in which case it is not in the same decision by Theorem 1.   □

Theorem 4 (Correctness). *Algorithm 2 is correct.*

Proof. By Theorem 2 Algorithm 2 is sound, and by Theorem 3 it is complete.   □

Theorem 5 (Complexity). *Algorithms 1 and 2 run in linear time complexity in the size of the CFG.*

Proof. Because of the *visited* data structure, clearly vertices in a CFG are visited at most twice. This can happen if a vertex is the last condition in a decision and is visited from a call to *merge* from Algorithm 1 before being visited in a recursive call in Algorithm 2.

Edges are only visited when they are outgoing edges from a vertex being visited (in calls to *successors*, with a maximum of two edges visited per call and a maximum of two calls.   □

VI. TEST COVERAGE CRITERIA DEFINITIONS

This section uses the CFDG model described in Section V to formally define common test coverage criteria.

Given a program $\mathcal{P}$, its CFDG $(V, E), D_s$, a function that defines the set of entry vertices of a graph $entry(V, E) = \{v \in V : \nexists(\cdot, v) \in E\}$, and a function that defines a set of exit vertices of a graph $exit(V, E) = \{v \in V : \nexists(v, \cdot) \in E\}$, a test suite $\mathcal{T}$, and the set of test runs $R = \{\text{Run}(G, t) : t \in \mathcal{T}\}$, we define the following test coverage criteria.

A. Statement Coverage (SC)

Statement coverage (SC) is the simplest criterion, and only requires that every statement in a program has been executed at least once [18].

$\mathcal{T}$ meets 100% SC requirements for $\mathcal{P}$ if it meets the following condition:

Condition 1 ($\mathcal{T}$ contains test cases which visit all of the vertices).

$$\forall v \in V. \ \exists r \in R. \ (\cdot, v) \in \boldsymbol{r} \vee (v, \cdot) \in \boldsymbol{r}$$

B. Decision Coverage (DC)

Decision coverage (DC) expands on SC by stipulating that every statement in a program has executed at least once, and every decision in the program has taken all possible outcomes at least once [18]. Recall that, by Lemma 1, decision subgraphs in CFDGs have only two successors.

$\mathcal{T}$ meets 100% DC requirements for $\mathcal{P}$ if it meets Condition 1, and the following condition:

Condition 2 ($\mathcal{T}$ contains test cases which visit both of the outgoing edges from every decision).

$$\forall D \in D_{\mathrm{s}}. \ \exists r \in R. \ \exists v_1, s_1, v_2, s_2 \in V.$$
$$(v_1, s_1), (v_2, s_2) \in \boldsymbol{r} \wedge s_1 \neq s_2 \wedge v_1, v_2 \in D \wedge s_1, s_2 \notin D$$

C. Condition Coverage (CC)

Condition coverage (CC), also called branch coverage, is a stronger expansion on SC, and specifies that every statement in a program has been executed at least once, and every condition in every decision has taken all possible outcomes at least once [18].

$\mathcal{T}$ meets 100% CC requirements for $\mathcal{P}$ if it meets Condition 1, and the following condition:

Condition 3 ($\mathcal{T}$ contains test cases which take both outgoing edges from every condition vertex in every decision subgraph).

$$\forall D \in D_{\mathrm{s}}. \ \forall v \in D. \ \exists r \in R. \ \exists s_1, s_2 \in V.$$
$$(v, s_1), (v, s_2) \in \boldsymbol{r} \wedge s_1 \neq s_2$$

D. Decision/Condition Coverage (D/CC)

Decision/condition coverage (D/CC) combines the requirements of DC and CC. D/CC specifies that every statement in a program has been executed at least once, every decision in the program has taken all possible outcomes at least once, and every condition in every decision has taken all possible outcomes at least once [18]. Note that CC does not imply DC because changing condition outcomes may not imply changing decision outcomes. To see this, suppose a decision (c_1 && c_2). If c_1 is varied while c_2 is held false, then the decision outcome does not change.

$\mathcal{T}$ meets 100% D/CC coverage requirements for $\mathcal{P}$ if it meets conditions 1, 2, and 3.

E. Multiple Condition Coverage (MCC)

Multiple condition coverage (MCC) is a still stronger criterion. MCC requires that every statement in a program has been executed at least once, and all possible combinations of the conditions in every decision have been taken at least once [18].

$\mathcal{T}$ meets 100% MCC requirements for $\mathcal{P}$ if it meets conditions 1, and the following condition:

Condition 4 ($\mathcal{T}$ contains test cases which, for every condition in every decision, vary only that condition).

$$\forall D \in D_{\mathrm{s}}. \ \forall v \in D. \ \exists \boldsymbol{r}_1, \boldsymbol{r}_2 \in R. \ \exists v, s_1, s_2, c, x \in V.$$
$$(v, s_1) \in \boldsymbol{r}_1 \wedge (v, s_2) \in \boldsymbol{r}_2 \wedge s_1 \neq s_2 \wedge$$
$$\{(c, x) \in \boldsymbol{r}_1 : c \in D \wedge c \neq v\} = \{(c, x) \in \boldsymbol{r}_2 : c \in D \wedge c \neq v\}$$

F. Full Predicate Coverage (FPC)

Full predicate coverage (FPC) requires that every statement in a program has been executed at least once, and every condition in every decision has taken all possible outcomes where the outcome is directly correlated to the outcome of the decision. This is a weaker criterion than MC/DC (introduced below) in that it does not require other conditions to be held fixed when varying a condition [13], [2], [3].

$\mathcal{T}$ meets 100% FPC requirements for $\mathcal{P}$ if it meets conditions 1, and the following condition:

Condition 5 ($\mathcal{T}$ contains test cases which, for every condition in every decision, vary that condition and result in a change to the outcome of the decision).

$$\forall D \in D_{\mathrm{s}}. \ \forall v \in D. \ \exists \boldsymbol{r}_1, \boldsymbol{r}_2 \in R. \ \exists s_1, s_2, c, x_1, x_2 \in V.$$
$$(v, s_1) \in \boldsymbol{r}_1 \wedge (v, s_2) \in \boldsymbol{r}_2 \wedge s_1 \neq s_2 \wedge$$
$$(c, x_1) \in \boldsymbol{r}_1 \wedge (c, x_2) \in \boldsymbol{r}_2 \wedge c \in D \wedge \{x_1, x2\} = successors(D)$$

G. Modified Condition/Decision Coverage (MC/DC)

Modified condition/decision coverage (MC/DC) is a criterion for qualifying the completeness of tests used in development standards in the avionics and automotive industries, notably in DO-178B and ISO-26262. It specifies that every entry and exit point to a program is tested, that every possible outcome of a decision is tested, that every possible result of every condition in a decision is tested, and that every condition in a decision independently affects the decision's outcome [19], [1], [20].

$\mathcal{T}$ meets 100% MC/DC requirements for $\mathcal{P}$ if it meets conditions 2, 3, and the following conditions:

Condition 6 ($\mathcal{T}$ contains a test case which visits the entry vertex).

$$\forall v \in entry(V, E). \ \exists r \in R. \ (v, \cdot) \in \boldsymbol{r}$$

Condition 7 ($\mathcal{T}$ contains test cases which visit all of the exit vertices).

$$\forall v \in exit(V, E). \ \exists \boldsymbol{r} \in R. \ (\cdot, v) \in \boldsymbol{r}$$

Condition 8 ($\mathcal{T}$ contains test cases which, for every condition in every decision, vary *only* that condition *and* result in a change to the outcome of the decision).

$$\forall D \in D_{s}. \ \forall v \in D. \ \exists \boldsymbol{r}_1, \boldsymbol{r}_2 \in R. \ \exists s_1, s_2, c, x, o, x_1, x_2 \in V.$$

$$(v, s_1) \in \boldsymbol{r}_1 \wedge (v, s_2) \in \boldsymbol{r}_2 \wedge s_1 \neq s_2 \wedge$$

$$\{(c, x) \in \boldsymbol{r}_1 : c \in D \wedge c \neq v\} = \{(c, x) \in \boldsymbol{r}_2 : c \in D \wedge c \neq v\}$$

$$\wedge (o, x_1) \in \boldsymbol{r}_1 \wedge (o, x_2) \in \boldsymbol{r}_2 \wedge o \in D \wedge \{x_1, x_2\} = successors(D)$$

VII. A TOOL TO ANNOTATE CFGS

We implemented the algorithms from Section V-A in an open-source tool available at [21] called *cfg2cfdg*. The tool is written in Python and uses the PyGraphViz [22] library to read and write GraphViz files in *dot* format. GraphViz dot files are a common format for representing graphs for analysis and visualization, including for CFGs. The popular compilers
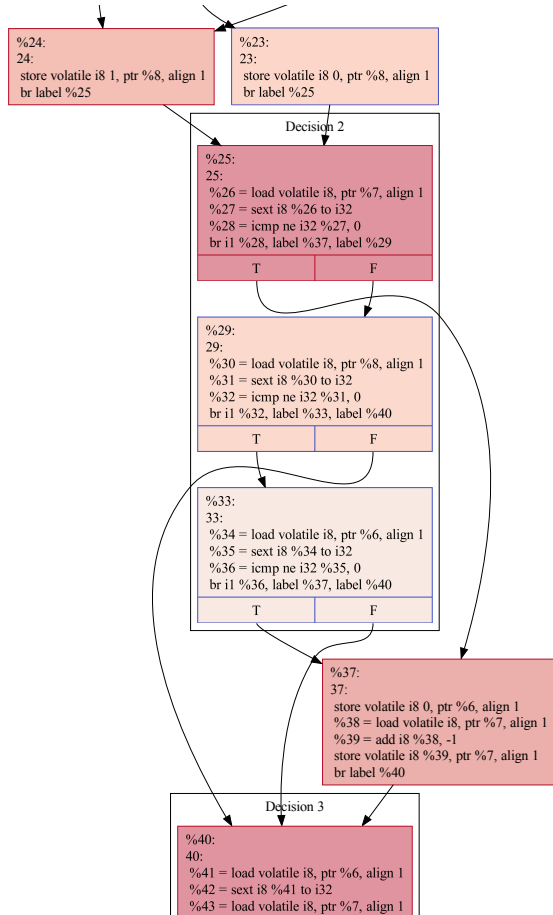

Fig. 3: Parts of a visualized CFG output from Clang annotated using cfg2cfdg

GNU Compiler Collection (GCC) and Clang are able to write GraphViz dot files containing CFGs for functions written in their supported languages and cfg2cfdg can annotate these files with decision information.

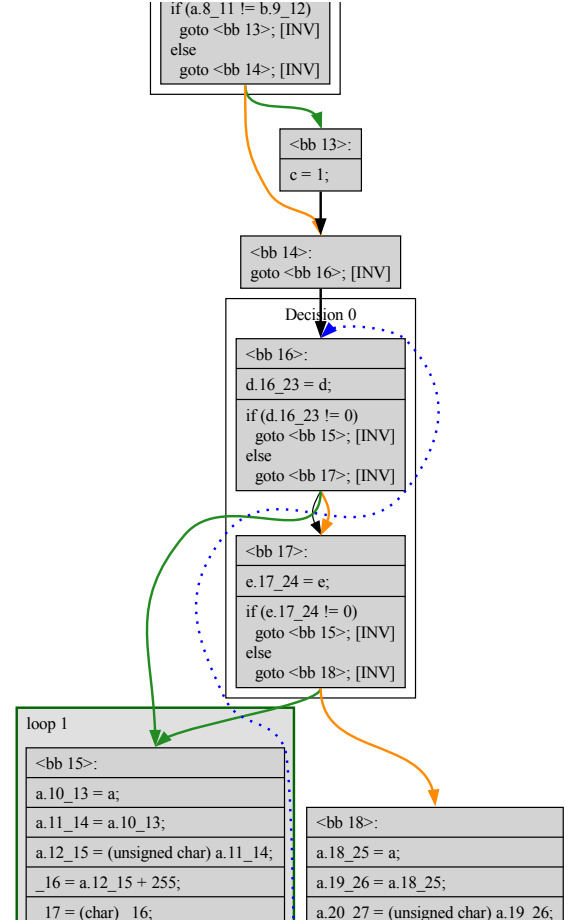Fig. 4: Parts of a visualized CFG output from GCC annotated using cfg2cfdg

Visualizations of annotated outputs from both compilers are shown in Figures 3 and 4. In the figure, vertices in the CFDG are shown as boxes, with arrows showing the edges. The decision annotations are visible in both graphs as boxes around the vertices they encapsulate, with the labels **Decision #**.

Figure 3 shows part of a CFDG from Clang with a complex decision (Decision 2) with three conditions combined with an && and an || operator similar to Listing 2. Clang names each basic block in the CFG with a comment on the first line in the form %n before the label n:. In the figure, Decision 2 has two entry vertices, %24 and %23, and two successor vertices, %37 and %40. Decision 2 can be understood as a conditional on %25 || (%29 && %33), where basic-block %37 is reached if the decision is true, with execution continuing to %40 (no **else** block).

Figure 4 shows part of a CFDG from GCC with a loop where Decision 0 has two conditions, where the back edge is a dotted blue line. GCC names each basic block by its label:

`<bb n>:`. In the figure, Decision 0 has two entry vertices, `<bb 14>` and `<bb 15>` (also labeled `loop 1`), which has a back-edge creating the loop, and two successor vertices, `<bb 15>` and `<bb 18>`. Decision 0 can be understood as a while loop on `<bb 16> || <bb 17>`. When the loop terminates, it continues with `<bb 18>`.

A. Challenges

Working with the output from these tools means relying on their specific interpretations of how CFGs should behave, which does not always align with expectations from the theory. One challenge we encountered was that Clang adds interstitial vertices to loops, separating *guards* and *latches*. This means that decisions do not meet the assumptions stated in Section V and, as such, the algorithm is not complete and decisions are not properly merged. GCC, on the other hand, treats loops as we expected, and our tool has no trouble adding decisions around loop decisions as in Figure 4.

Using the tool is generally simple, requiring only the dot filename as input and immediately writing out a new dot file with decision annotations. Visualization of the new CFDG using open-source GraphViz software is generally easy, but we did find that in some circumstances it was necessary to edit the files before visualization. For example, GCC can use subgraphs in the file to annotate function boundaries, and this can interfere with visualizing decisions. We found that editing the function subgraph names to remove the keyword *cluster* solved the problem.

VIII. Conclusion

We have proposed a model, called a Control-Flow Decision Graph (CFDG), for formalizing and reasoning about control flow coverage criteria. We introduced an algorithm to construct a CFDG from a CFG and proved its correctness and complexity, and then implemented that algorithm in an open-source tool for annotating CFGs from GCC and Clang [21]. We also formalize common test coverage criteria using the CFDG model, showing how explicit information about decisions in a CFG can help with program understanding and analysis.

It is our hope that this work will provide the basis for further insight into the relationship of decisions to program logic and control-flow properties.

References

[1] C. Comar, J. Guitton, O. Hainque, and T. Quinot, "Formalization and comparison of MCDC and object branch coverage criteria," in *ERTS (Embedded Real Time Software and Systems Conference)*, 2012.

[4] 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, no. 1, pp. 42–62, 2006. [Online]. Available: http://dx.doi.org/10.1007/s00165-005-0084-7

[2] S. A. Vilkomir and J. P. Bowen, "Formalization of control-flow criteria of software testing," in *Proceedings of COMPSAC*. Citeseer, 2001.

[3] S. A. Vilkomir and J. P. Bowen "Formalization of software testing criteria using the z notation," in *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*. IEEE, 2001, pp. 351–356.

[5] S. A. Vilkomir and J. P. Bowen "Application of formal methods for establishing regulatory requirements for safety-critical software of real-time control systems," Technical Report SBU-CISM-01-03, SCISM, South Bank University, London, UK, Tech. Rep., 2001.

[6] M. R. Woodward and M. A. Hennell, "On the relationship between two control-flow coverage criteria: all JJ-paths and MCDC," *Information and Software Technology*, vol. 48, no. 7, pp. 433–440, 2006.

[7] R. Kirner, "Towards preserving model coverage and structural code coverage," *EURASIP J. Embedded Syst.*, vol. 2009, pp. 6:1–6:16, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1155/2009/127945

[8] P. Ammann, J. Offutt, and H. Huang, "Coverage criteria for logical expressions," in *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. IEEE, 2003, pp. 99–107.

[9] K. Kapoor and J. P. Bowen, "A formal analysis of MCDC and RCDC test criteria," *Software Testing, Verification and Reliability*, vol. 15, no. 1, pp. 21–40, 2005.

[10] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting, "Boundary coverage criteria for test generation from formal models," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*. IEEE, 2004, pp. 139–150.

[11] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman, "Test data generation and feasible path analysis," in *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. ACM, 1994, pp. 95–107.

[12] K. Ghani and J. A. Clark, "Automatic Test Data Generation for Multiple Condition and MCDC Coverage," in *Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on*. IEEE, 2009, pp. 152–157.

[13] A. J. Offutt, Y. Xiong, and S. Liu, "Criteria for generating specification-based tests," in *Engineering of Complex Computer Systems, 1999. ICECCS'99. Fifth IEEE International Conference on*. IEEE, 1999, pp. 119–129.

[14] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: ACM, 1970, pp. 1–19. [Online]. Available: http://doi.acm.org/10.1145/800028.808479

[15] A. Rimsa, J. Nelson Amaral, and F. M. Q. Pereira, "Practical dynamic reconstruction of control flow graphs," *Software: Practice and Experience*, vol. 51, no. 2, pp. 353–384, 2021.

[16] CAST Certification Authorities Software Team, "Structural coverage of object code," Tech. Rep., June 2003.

[17] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, pp. 49–61.

[18] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[19] H. Kelly J., V. Dan S., C. John J., and R. Leanna K., "A practical tutorial on modified condition/decision coverage," Tech. Rep., 2001.

[20] J. J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," DTIC Document, Tech. Rep., 2001.

[21] S. Kauffman, "Control-flow graph to control-flow decision graph (cfg2cfdg) tool repository," https://bitbucket.org/seanmk/cfg2cfdg, 2024, accessed: Feb 2024.

[22] P. Developers, "Pygraphviz python module website," https://pygraphviz.github.io/, 2024, accessed: Feb 2024.