

Coverage Criteria for Logical Expressions

Paul Ammann, Jeff Offutt and Hong Huang
Information and Software Engineering Department
George Mason University
Fairfax, VA 22030, USA
(+1) 703-993-1660 / 1654
{pammann,ofut,hhuang2}@ise.gmu.edu

Abstract

A large number of coverage criteria to generate tests from logical expressions have been proposed. Although there have been large variations in the terminology, the articulation of the criteria and the original source of the expressions, many of these criteria are fundamentally the same. The most commonly known and widely used criterion is that of Modified Condition Decision Coverage (MCDC), but some articulations of MCDC have had some ambiguities. This has led to confusion on the part of testers, students, and tool developers on how best to implement these test criteria. This paper presents a complete comprehensive set of criteria that incorporate all the existing criteria, and eliminates the ambiguities by introducing precise definitions of the various possibilities.

1. Introduction

Logical expressions are common to almost every type of software artifact, including program source code, finite state machines, and formal specifications. Because they are so common, easily formalizable, and easy to process automatically, several test criteria have been defined that are based on logical expressions. While logic coverage criteria have been known for a long time, their use has been steadily growing in recent years. One major cause for their use in practice has been the requirement by the US Federal Aviation Administration (FAA) that one of the logic coverage criteria, Modified Condition Decision Coverage (MCDC) [3], be used for safety critical parts of the avionics software in commercial aircraft [14]. A number of other criteria exist, some of which are equivalent to MCDC and others of which are very similar. In fact, a careful study of the published literature on the subject reveals

that the same ideas have been repeated several times in different contexts [3, 4, 8, 13, 15, 16]. The contribution of this paper is to create a comprehensive unifying organization of logic expression criteria, so as to simplify and clarify them.

We start with a sound theoretical foundation for logic predicates and clauses with the goal of making the subsequent testing criteria simpler. We take a generic view of the structures and criteria, independent of whether the logic expressions are derived from source code, specifications, and finite state machines.

We formalize logical expressions in a common mathematical way. A *predicate* is an expression that evaluates to a boolean value, and is our topmost structure. A simple example is: $((a > b) \vee C) \wedge p(x)$. Predicates may contain boolean variables, non-boolean variables that are compared with relational operators, and calls to function that return a boolean value, all three of which may be joined with logical operators. The internal structure is created by the logical operators:

- \neg – the *negation* operator
- \wedge – the *and* operator
- \vee – the *or* operator
- \rightarrow – the *implication* operator
- \oplus – the *exclusive or* operator
- \leftrightarrow – the *equivalence* operator

Some of these operators (\rightarrow , \oplus , \leftrightarrow) may seem unusual for readers with a bias toward source code, but they are common in specification languages and convenient in our presentation. Also note that the equivalence operator shows up in programming languages as an equality comparator, that is, if **A** and **B** are boolean variables, “if (**A** == **B**)” is equivalence. Precedence is as typical and matches the order listed above from highest to lowest. When there is doubt, we use parentheses for clarity.

A *clause* is a predicate that does not contain any of the logical operators. The predicate $((a > b) \vee C) \wedge p(x)$ contains three clauses; a relational expression $(a > b)$, a boolean variable C and a boolean function call $p(x)$.

A predicate may be written in a variety of logically equivalent ways. For example, the predicate $((a > b) \wedge p(x)) \vee (C \wedge p(x))$ is logically equivalent to the predicate given in the previous paragraph, but $((a > b) \vee p(x)) \wedge (C \vee p(x))$ is not. The usual rules of boolean algebra may be used to convert boolean expressions into equivalent forms.

We treat logical expressions according to their meanings, not their syntax. As a consequence, a given logical expression yields the same test requirements for a given coverage criteria no matter which of the equivalent (syntactic) forms of the logic expression is used. We note that other researchers have taken the opposite tack – usually in an effort to equate the detection power of a test set with syntactic mutations of an expression [16].

2. Logic Expression Criteria

We assume that test coverage is evaluated in terms of test criteria, as articulated by test requirements. *Test requirements (TR)* are specific elements of software artifacts that must be satisfied or covered. Test requirements can be described in terms of a variety of software artifacts, including the source code, design components, specification modeling elements, or even descriptions of the input space. *Test specifications* are specific descriptions of test cases, often associated with or derived from test requirements. If test requirements describe “what” is to be covered, test specifications define in general terms “how” they will be covered. In the case of branch coverage, for example, test specifications are descriptions of the inputs that are needed to reach a particular branch (usually in predicate form).

These two terms allow a straightforward definition for a test coverage criterion. A *test criterion* is a rule or collection of rules that impose test requirements on a set of test cases. That is, the criterion describes the test requirements in a complete and unambiguous manner.

Clauses and predicates are used to introduce the simplest logic expression coverage criteria. Let P be a set of predicates and C be the set of clauses in the predicates in P . For each predicate $p \in P$, let C_p be the set of clauses in p , that is $C_p = \{c | c \in p\}$. Typically, C is the union of the clauses in each predicate in P , that is $C = \bigcup_{p \in P} C_p$.

Definition 1 Predicate Coverage (PC): *For each*

$p \in P$, *TR contains two requirements: p evaluates to true, and p evaluates to false.*

Predicate Coverage is equivalent to the common branch coverage criterion for source code, where each branch in a program (or edge in a graph) is covered, and has also been called Decision Coverage [12]. For the predicate given above, $((a > b) \vee C) \wedge p(x)$, two tests that satisfy Predicate Coverage are $(a = 5, b = 4, C = \text{false}, p(x) = \text{true})$ and $(a = 5, b = 6, C = \text{false}, p(x) = \text{true})$.

A failing of this criterion is that the individual clauses are not always exercised. Predicate coverage for the example above is satisfied without varying either C or $p(x)$. To rectify this problem, we move to the clause level.

Definition 2 Clause Coverage (CC): *For each $c \in C$, TR contains two requirements: c evaluates to true, and c evaluates to false.*

Our predicate $((a > b) \vee C) \wedge p(x)$ requires different values to satisfy CC. Clause Coverage requires that $(a > b) = \text{true}$ and false , $C = \text{true}$ and false , and $p(x) = \text{true}$ and false . This can be satisfied with two tests: $((a = 5, b = 4), (C = \text{true}), p(x) = \text{true})$ and $((a = 5, b = 6), (C = \text{false}), p(x) = \text{false})$.

Clause Coverage has also been called Condition Coverage [12]. A common way to compare test criteria is in terms of subsumption. A test criterion C1 *subsumes* C2 if and only if every test set that satisfies C1 will also satisfy C2. For example, if every branch is executed (branch coverage) and the program contains no dead code, then we are ensured that every statement will be executed (statement coverage). That is, branch coverage subsumes statement coverage.

Clause Coverage does not subsume Predicate Coverage, and Predicate Coverage does not subsume Clause Coverage, as shown by the predicate $p = a \vee b$. Formally, the clauses C are those in p : $C = C_p = \{a, b\}$. Consider the four test inputs that enumerate the combinations of logical values for the clauses:

$$\begin{aligned} p &= a \vee b \\ t_1 &= (a = \text{true}, b = \text{true}) \rightarrow p = \text{true} \\ t_2 &= (a = \text{true}, b = \text{false}) \rightarrow p = \text{true} \\ t_3 &= (a = \text{false}, b = \text{true}) \rightarrow p = \text{true} \\ t_4 &= (a = \text{false}, b = \text{false}) \rightarrow p = \text{false} \end{aligned}$$

If we choose the pair of test cases $T_1 = \{t_1, t_2\}$, it satisfies neither Clause Coverage (because a is never false) nor Predicate Coverage (because p is never false). The pair of tests $T_2 = \{t_2, t_3\}$ satisfies Clause Coverage, but not Predicate Coverage (because p is never

false). The pair of tests $T_3 = \{t_2, t_4\}$ satisfies Predicate Coverage, but not Clause Coverage (because b is never true). The pair of tests $T_4 = \{t_1, t_4\}$ is the only pair that satisfies both Clause and Predicate Coverage. These test sets demonstrate that neither Predicate Coverage nor Clause Coverage subsume the other.

From the testing perspective, we would certainly like a coverage criterion that tests individual clauses and that also tests the predicate. The most direct approach to rectify this problem is to try all combinations of clauses:

Definition 3 Combinatorial Coverage (CoC): For each $p \in P$, TR has test requirements for the clauses in C_p to evaluate to each possible combination of truth values.

Combinatorial Coverage has also been called Multiple Condition Coverage [12]. For the predicate $((A \vee B) \wedge C)$, the complete truth table contains eight elements:

	A	B	C	$(A \vee B) \wedge C$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

For a predicate p with n independent clauses, there are 2^n possible assignments of truth values, thus Combinatorial Coverage is unwieldy at best, and impractical for predicates with more than a few clauses. What is needed is a criterion that captures the effect of each clause, but does so in a reasonable number of tests. These requirements have led to a powerful collection of test criteria that are based on the notion of making individual clauses “active” as defined in the next subsection.

2.1. Active Clause Criteria

The lack of subsumption between Clause and Predicate Coverage is unfortunate, but there are deeper problems. Specifically, when we introduce tests at the clause level, we also want to have an effect on the predicate. When debugging, we say that one fault *masks* another if the second fault cannot be observed until the first fault is corrected. There is a similar notion of masking in logical expressions. In the predicate

$p = a \wedge b$, if $b = false$, b can be said to *mask* a because no matter what value a has, p will still be *false*. Furthermore, if the language uses short circuit evaluation, if a is false, b will not even be evaluated! To avoid masking when we construct tests, we want to construct tests such that the value of the predicate is directly dependent on the value of the clause that we want to test. For convenience of expression, we call “the clause that we want to test” the *major clause*.

Definition 4 Determination: Given a clause c_i in predicate p , called the major clause, we say that c_i determines p if the remaining minor clauses $c_j \in p$, $j \neq i$, have values so that changing the truth value of c_i changes the truth value of p .

Note that this definition explicitly does **not** require that the major clause and the predicate have the same value ($c_i = p$). This issue has been left ambiguous by previous definitions, and ACC-like criteria have sometimes been taught as requiring that the predicate and the major clause must have the same value. This interpretation is not practical. When the negation operator is used, for example, if the predicate is $p = \neg a$, it becomes impossible for the major clause and the predicate to have the same value.

Consider the example above, where $p = a \vee b$. If b is false, then clause a determines p , because the value of p is exactly the value of a . However if b is true, then a does not determine p , since the value of p is true regardless of the value of a . A general method for computing determination is given in Section 2.4.

Definition 5 Active Clause Coverage (ACC): For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j , $j \neq i$ so that c_i determines p . TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false.

For example, for $p = a \vee b$, we end up with a total of four requirements in TR , two for clause a and two for clause b . Clause a determines p if and only if b is false. So we have the two test requirements $\{(a = true, b = false), (a = false, b = false)\}$. Clause b determines p if and only if a is false, yielding the two test requirements $\{(a = false, b = true), (a = false, b = false)\}$. This is summarized in the partial truth table below (the values for the major clauses are in bold face). Note that for the remainder of the paper, we ignore the structure of the clause and only focus on its truth assignment, that is, we do not use test case values but only the test specifications.

	a	b
$c_i = a$	T	f
	F	f
$c_i = b$	f	T
	f	F

Two of these requirements are identical, so we end up with three distinct test requirements for ACC for the predicate $a \vee b$, $\{(a = \text{true}, b = \text{false}), (a = \text{false}, b = \text{true}), (a = \text{false}, b = \text{false})\}$. Such overlap always happens, and it turns out that for a predicate with n independent clauses, at most $n+1$ distinct test requirements, rather than the $2n$ one might expect, are sufficient to satisfy ACC.

ACC is almost identical to the way MCDC was initially described in original papers on MCDC and in other papers and textbooks. It turns out that this criterion has some ambiguity, which has led to confusion in how to interpret MCDC. The most important question is whether the minor clauses c_j need to have the same values when the major clause c_i is true as when c_i is false. Resolving this ambiguity leads to three distinct and interesting flavors of ACC. For a simple predicate such as $p = a \vee b$, the three flavors turn out to be identical, but differences appear for more complex predicates. The most general flavor allows the minor clauses to have different values.

Definition 5(i) General Active Clause Coverage (GACC): For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$ so that c_i determines p . TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j do not need to be the same when c_i is true as when c_i is false, that is, $c_j(c_i = \text{true}) = c_j(c_i = \text{false}) \vee c_j$ OR $c_j(c_i = \text{true}) \neq c_j(c_i = \text{false}) \vee c_j$.

Unfortunately, GACC does not subsume Predicate Coverage in all cases, as the following example shows. Consider the predicate $p = a \leftrightarrow b$. Clause a determines p for any assignment of truth values to b . So when a is true, we can choose b to also be true, and when a is false, we can choose b to also be false. We can make the same selections for clause b . Thus we end up with only two test requirements, $TR = \{(a = \text{true}, b = \text{true}), (a = \text{false}, b = \text{false})\}$. Predicate p evaluates to true for both of these cases, so Predicate Coverage is not achieved.

Most testing researchers have a strong feeling that ACC should subsume PC, thus the second flavor of ACC insists that p evaluates to true for one assignment of values to the major clause c_i , and false for the other. Note that c_i and p do not have to have the same values,

as discussed with the definition for determination.

Definition 5(ii) Correlated Active Clause Coverage (CACC): For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$ so that c_i determines p . TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j must cause p to be true for one value of the major clause c_i and false for the other, that is, it is required that $p(c_i = \text{true}) \neq p(c_i = \text{false})$.

Consider the example $p = a \wedge (b \vee c)$. For a to determine the value of p , the expression $b \vee c$ must be true. There are three ways to achieve this: b true and c false, b false and c true, and both b and c true. So, it would be possible to satisfy Correlated Active Clause Coverage with respect to clause a with the two test requirements: $\{(a = \text{true}, b = \text{true}, c = \text{false}), (a = \text{false}, b = \text{false}, c = \text{true})\}$. There are other possible sets of test requirements with respect to a , as enumerated in the following partial truth table. The row numbers are taken from the complete truth table for the predicate given previously. Specifically, CACC can be satisfied for a by choosing one test requirement from rows 1, 2 and 3, and another from rows 5, 6 and 7. There are, of course, nine possible ways to do this.

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F

The version of MCDC commonly called “masking MCDC” [4] is equivalent to CACC. That is, masking MCDC allows the values for the minor clauses to be different for the two values of the major clauses. The original definition of MCDC [3] was often interpreted to mean that the values for the minor clauses c_j must be the same for the two values of the major clauses c_i . This version, sometimes called “unique cause MCDC”, is captured in the final flavor of ACC.

Definition 5(iii) Restricted Active Clause Coverage (RACC): For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$ so that c_i determines p . TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j must be the same when c_i is true as when c_i is false, that is, it is required that $c_j(c_i = \text{true}) = c_j(c_i = \text{false}) \vee c_j$.

For the example $p = a \wedge (b \vee c)$, only three of the nine sets of test requirements that satisfy Correlated Active Clause Coverage with respect to clause a will satisfy Restricted Active Clause Coverage with respect to clause a . In terms of the previously given complete truth table, row 2 can be paired with row 6, row 3 with row 7, or row 1 with row 5. Thus, instead of the nine ways to satisfy CACC, only three can satisfy RACC. This is illustrated in the following table.

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
5	F	T	T	F
2	T	T	F	T
6	F	T	F	F
3	T	F	T	T
7	F	F	T	F

2.2. CACC versus RACC

Because of the more restrictive nature of RACC, there are some logical expressions that can be completely satisfied under CACC, but that have infeasible test requirements under RACC. These expressions are a little subtle and only exist if there are dependency relationships among that clauses, that is, some combinations of values for the clauses are prohibited. This often happens in real programs where program variables frequently depend upon one another, as illustrated in the following example.

Consider a system with a valve that might be either open or closed, and that has several modes, two of which are “Standby” and “Operational.” Suppose that there are two constraints:

1. The valve must be open in “Operational” and closed in all other modes.
2. The mode cannot be both “Standby” and “Operational” at the same time.

These constraints lead to the following clause definitions:

$$\begin{aligned} a &= \text{“The valve is closed”} \\ b &= \text{“The system status is Operational”} \\ c &= \text{“The system status is Standby”} \end{aligned}$$

The two constraints can now be formalized as:

1. $\neg a \leftrightarrow b$
2. $\neg(b \wedge c)$

Suppose that a certain action can only be taken if the valve is closed and the system status is either in *Operational* or *Standby*. That is, the system has the following predicate:

$$\begin{aligned} p &= \text{valve is open AND} \\ &\quad (\text{system status is Operational OR} \\ &\quad \text{system status is Standby}) \\ &= a \wedge (b \vee c) \end{aligned}$$

This is exactly the predicate that was analyzed above. The two constraints, however, limit the feasible values in the truth table.

Recall that for a to determine the value of p , either b or c or both must be true. Constraint 1 rules out the rows where a and b have the same values, that is, rows 1, 2, and 7. Constraint 2 rules out the rows where b and c are both true, that is, rows 1 and 5. Thus the only feasible rows are 3 and 6. Recall that CACC can be satisfied by choosing one from rows 1, 2 or 3 and one from rows 5, 6 or 7. But RACC requires one of the pairs 2 and 6, 3 and 7, or 1 and 5. Thus RACC is infeasible for a in this predicate.

2.3. InActive Clause Criteria

The Active Clause Coverage Criteria focus on making sure the major clauses do affect their predicates. A complementary criterion to Active Clause Coverage was defined by Vilkomir and Bowen [15]. It ensures that changing a major clause that should *not* affect the predicate does not, in fact, affect the predicate.

Definition 6 Inactive Clause Coverage: For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$ so that c_i does not determine p . TR has four requirements for c_i under these circumstances: c_i evaluates to true with p true, c_i evaluates to false with p true, c_i evaluates to true with p false, and c_i evaluates to false with p false.

Although Inactive Clause Coverage (ICC) has some of the same ambiguity as ACC, there are only two distinct flavors. Major clause c_i cannot correlate with p since c_i does not determine p , so the notion of correlation is not relevant for Inactive Clause Coverage. This leaves the two flavors *General Inactive Clause Coverage (GICC)* and *Restricted Inactive Clause Coverage (RICC)*. Also, Predicate Coverage is guaranteed in all flavors due to the structure of the definition. The formal versions of GICC and RICC are as follows.

Definition 6(i) General Inactive Clause Coverage (GICC): For each $p \in P$ and each major clause

$c_i \in C_p$, choose minor clauses c_j , $j \neq i$ so that c_i does not determine p . The values chosen for the minor clauses c_j do not need to be the same when c_i is true as when c_i is false, that is, $c_j(c_i = true) = c_j(c_i = false) \forall c_j$ OR $c_j(c_i = true) \neq c_j(c_i = false) \forall c_j$.

Definition 6(ii) Restricted Inactive Clause Coverage (RICC): For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j , $j \neq i$ so that c_i does not determine p . The values chosen for the minor clauses c_j must be the same when c_i is true as when c_i is false, that is, it is required that $c_j(c_i = true) = c_j(c_i = false) \forall c_j$.

Figure 1 shows the subsumption relationships among the logic expression criteria. Note that the Inactive Clause Coverage Criteria do not subsume any of the Active Criteria, and vice versa.

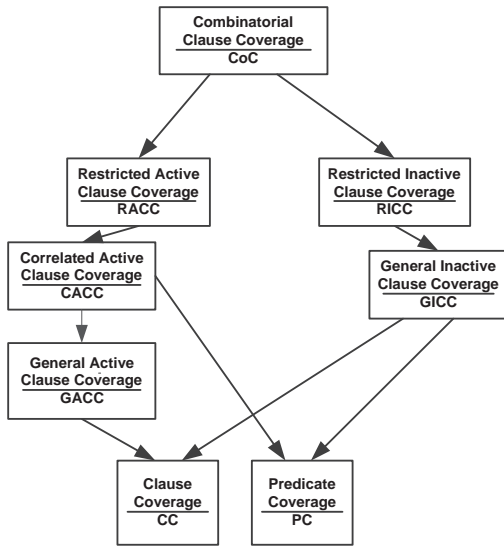


Figure 1. Subsumption relations among logic coverage criteria

2.4. Making a Clause Determine a Predicate

A key problem with applying logical coverage criteria is solving determination and three methods are known. The method of determining p_c given here simply uses the boolean derivative developed by Akers [1]. Both Chilenski [4] and Kuhn [11] applied Akers's derivative to the problem of determination. Other methods are the pairs table method of Chilenski and Miller [3] and the tree method, which was independently discovered by Chilenski [4] and Offutt [13]. The

tree method implements the boolean derivative method in a procedural way. One advantage of the boolean derivative method is the issue of multiple occurrences of the same clause is handled cleanly, that is, the fact that the clause appears more than once is explicitly represented.

For a predicate p with clause (or boolean variable) c , let $p_{c=true}$ represent the predicate p with every occurrence of c replaced by *true* and $p_{c=false}$ be the predicate p with every occurrence of c replaced by *false*. Note that neither $p_{c=true}$ nor $p_{c=false}$ contains any occurrences of the clause c . Now we connect the two expressions with an exclusive or:

$$p_c = p_{c=true} \oplus p_{c=false}$$

It turns out that p_c describes the exact conditions under which the value of c determines that of p . That is, if values for the clauses in p_c are chosen so that p_c is true, then the truth value of c determines the truth value of p . If the clauses in p_c are chosen so that p_c evaluates to false, then the truth value of p is independent of the truth value of c . This is exactly what we need to implement the various flavors of Active and Inactive Clause Coverage.

This method is best explained with an example. Consider the expression used before, $p = a \wedge (b \vee c)$. If the major clause is a , then the boolean derivative finds truth assignments for b and c as follows:

$$\begin{aligned}
 p_a &= p_{a=true} \oplus p_{a=false} \\
 &= (true \wedge (b \vee c)) \oplus (false \wedge (b \vee c)) \\
 &= (b \vee c) \oplus false \\
 &= b \vee c.
 \end{aligned}$$

This example ends with a nondeterministic answer, which points out the key difference between CACC and RACC. Three choices of values make $b \vee c$ true, ($b = c = true$), ($b = true, c = false$), and ($b = false, c = true$). For CACC, we could pick one pair of values when a is true and another when a is false. For RACC, we must choose the same pair for both values of a .

The derivation for b and equivalently for c is slightly more complicated, but the result has only one solution:

$$\begin{aligned}
 p_b &= p_{b=true} \oplus p_{b=false} \\
 &= (a \wedge (true \vee c)) \oplus (a \wedge (false \vee c)) \\
 &= (a \wedge true) \oplus (a \wedge c) \\
 &= a \oplus (a \wedge c) \\
 &= a \wedge \neg c
 \end{aligned}$$

The computation for p_c is equivalent and yields the solution $a \wedge \neg b$. If the result of the boolean derivative is a tautology (for example, $B \vee \neg B$ or *TRUE*),

that means the minor clauses can have any value and the major clause will still determine the value of the predicate.

3. Predicate Transformation Issues

ACC criteria are considered to be expensive for testers, and attempts have been made to reduce the cost. One approach is to rewrite the program to eliminate multi-clause predicates, thus reducing the problem to branch testing. A conjecture is that the resulting tests will be equivalent to ACC. However, we explicitly advise against this approach for two reasons. One, the resulting rewritten program may have substantially more complicated control structure than the original (including repeated statements), thus endangering both reliability and maintainability. Second, as the following examples demonstrate, the transformed program will not require tests that are equivalent to the tests for ACC on the original program.

Consider the following program segment, where *a* and *b* are arbitrary boolean clauses and *S1* and *S2* are arbitrary statements. *S1* and *S2* could be single statements, block statements, or function calls.

```

if (a && b)
    S1;
else
    S2;

```

The Correlated Active Clause Coverage criterion requires the test specifications (t, t) , (t, f) , and (f, t) for the predicate $a \wedge b$. However, if the program segment is transformed into the following functionally equivalent structure:

```

if (a)
{
    if (b)
        S1;
    else
        S2;
}
else
    S2;

```

the Predicate Coverage criterion requires three tests: (t, t) to reach statement *S1*, (t, f) to reach the first occurrence of statement *S2*, and either (f, f) or (f, t) to reach the second occurrence of statement *S2*. Choosing (t, t) , (t, f) , and (f, f) means that our tests do **not** satisfy CACC in that they do not allow *a* to fully determine the predicate's value. Moreover, the duplication of *S2* in the above example has been taught to be

poor programming for years, because of the potential for mistakes when duplicating code.

A larger example reveals the flaw even more clearly. Consider the simple program segment:

```

if ((a && b) || c)
    S1;
else
    S2;

```

A straightforward rewrite of this program fragment to remove the multi-clause predicate results in:

<pre> if (a) if (b) if (c) S1; else S1; else if (c) S1; else S2; </pre>		<pre> else if (b) if (c) S1; else S2; else if (c) S1; else S2; </pre>
---	--	---

This fragment is cumbersome in the extreme, and likely to be error-prone. Applying the predicate coverage criterion to this would be equivalent to applying combinatorial coverage to the original predicate. A reasonably clever programmer (or good optimizing compiler) would simplify it as follows:

```

if (a)
    if (b)
        S1;
    else
        if (c)
            S1;
        else
            S2;
else
    if (c)
        S1;
    else
        S2;

```

The following table illustrates truth assignments that can be used to satisfy CACC for the original program segment and predicate testing for the modified version. An 'X' under CACC or Predicate indicates that truth assignment is used to satisfy the criterion for the appropriate program fragment. Clearly, predicate coverage on an equivalent program is not the same as CACC testing on the original.

	a	b	c	$(a \wedge b) \vee c$	CACC	Predicate
1	t	t	t	T		X
2	t	t	f	T	X	
3	t	f	t	T	X	X
4	t	f	f	F	X	X
5	f	t	t	T		X
6	f	t	f	F	X	
7	f	f	t	T		
8	f	f	f	F		X

4. Related Work

The active clause criteria had its beginning in Myers’ 1979 book [12]. He defined decision and condition coverage, which Chilenski and Miller later used as a conceptual basis for MCDC [14, 3]. The definitions as originally given correspond to GACC in this paper and did not address whether minor clauses had to have the same value for both values of the major clause. Most members of the aviation community interpreted MCDC to mean that the values of the minor clauses had to be the same, an interpretation that is called “unique-cause MCDC” [2]. Unique-cause MCDC corresponds to our RACC. More recently, the FAA has accepted the view that the minor clauses can differ, which is called “masking MCDC” [4]. Masking MCDC corresponds to our CACC.

The inactive clause criteria are adapted from the RC/DC method of Vilkomir and Bowen [15]. Their stated goal was to improve on MCDC, and the inactive clause criteria are clearly complementary to the active clause criteria.

Jasper et al. presented techniques for generating tests to satisfy MCDC [9]. They took the definition of MCDC from Chilenski and Miller’s paper with the “default” interpretation that the minor clauses must be the same for both values of the major clauses. They went on to modify the interpretation so that if two clauses are coupled, which implies it is impossible to satisfy determination for both, the two clauses are allowed to have different values for the minor clauses. The fact that different values are allowed only when clauses are coupled puts their interpretation of MCDC between our RACC and CACC.

Weyuker, Goradia and Singh presented techniques for generating test data for software specifications that are limited to boolean variables [16]. The techniques were compared in terms of the ability of the resulting test cases to kill mutants [5, 6]. Results showed that their technique that was equivalent to MCDC performed better than any of the other techniques. Unlike our work, Weyuker et al. incorporated syntax as well

as meaning into their criteria. They presented a notion called *meaningful impact* which is related to our notion of determination, but which has a syntactic basis rather than a semantic one.

Kuhn investigated methods for generating tests to satisfy various decision-based criteria, including MCDC tests [11]. He used the definition from Chilenski and Miller [14, 3], and proposed the boolean derivative to satisfy MCDC. In effect, this interpreted MCDC as our CACC.

Dupuy and Leveson’s 2000 paper evaluated MCDC experimentally [7]. They presented results from an empirical study that compared pure functional testing with functional testing augmented by MCDC. The experiment was performed during the testing of the attitude control software for the HETE-2 (High Energy Transient Explorer) scientific satellite. The definition of MCDC from their paper is the traditional definition given in the FAA report and Chilenski and Miller’s paper: “Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to affect that decision outcome independently. A condition is shown to affect a decision’s outcome independently by varying just that decision while holding fixed all other possible conditions.”

Note the typo in last line: “varying just that decision” should be “varying just that condition”. This does not say that the decision has a different value when the condition’s value changes. “Holding fixed” can be assumed to imply that the minor clauses cannot change with different values for the major clause (that is, RACC, not CACC).

Jones and Harrold have developed a method for reducing the regression tests that were developed to satisfy MCDC [10]. They defined MCDC as follows: “MC/DC is a stricter form of decision (or branch) coverage. ... MC/DC requires that each condition in a decision be shown by execution to independently affect the outcome of the decision”. This is taken directly from Chilenski and Miller’s original paper, and their interpretation of the definition is our CACC.

5. Conclusions

This paper has presented a set of test criteria for logical expressions by integrating existing ideas into a common unifying framework. Most of the concepts in this paper already existed, but were articulated in very different ways. Some of these diverse descriptions were ambiguous, and the diversity in terminology and presentations has led to confusion on the part of testers,

educators, teachers, and tool vendors. It also has led to researchers re-inventing existing ideas by applying them to slightly different contexts and using different terminology, as demonstrated by the literature review in Section 4. This paper introduces a new way to express the requirement that tests “independently affect the outcome of a decision” by defining the term determination, and separating minor and major clauses. This allows the differences between general, restricted, and correlated active clause coverage to be cleanly formalized, and also allows the differences between active and inactive clause coverage to be formalized.

The definitions presented here resolve the ambiguities by elegantly putting all the criteria into a common family and structure. With the definitions as expressed here, it does not matter whether the predicates and clauses are formed from boolean variables or logical expressions, and whether they are derived from program source code, specifications, or design notations.

6. Acknowledgment

We would like to thank John Chilenski of Boeing, the inventor of MCDC, for extensive discussions on the material in this paper as well as numerous suggestions.

References

- [1] S. B. Akers. On a theory of boolean functions. *Journal Society Industrial Applied Mathematics*, 7(4):487–498, December 1959.
- [2] J. J. Chilenski. Personal communication, March 2003.
- [3] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, September 1994.
- [4] John Chilenski and L. A. Richey. Definition for a masking form of modified condition decision coverage (MCDC). Technical report, Boeing, Seattle WA, 1997. <http://www.boeing.com/nosearch/mcdc/>.
- [5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [6] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
- [7] Arnaud Dupuy and Nancy Leveson. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In *Proceedings of the Digital Aviations Systems Conference (DASC)*, October 2000.
- [8] K. Foster. Error sensitive test case analysis. *IEEE Transactions on Software Engineering*, 6(3):258–264, May 1980.
- [9] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *Proceedings of the 1994 International Symposium on Software Testing, and Analysis*, pages 95–107, Seattle WA, August 1994.
- [10] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition / decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, March 2003.
- [11] D. Richard Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering Methodology*, 8(4):411–424, October 1999.
- [12] G. Myers. *The Art of Software Testing*. John Wiley and Sons, New York NY, 1979.
- [13] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *The Journal of Software Testing, Verification, and Reliability*, 13(1):25–53, March 2003.
- [14] RTCA-DO-178B. Software considerations in airborne systems and equipment certification, December 1992.
- [15] S. A. Vilkomir and J. P. Bowen. Reinforced condition/decision coverage (RC/DC): A new criterion for software testing. In *Proceedings of ZB2002: 2nd International Conference of Z and B Users*, pages 295–313, Grenoble, France, January 2002. Springer-Verlag, LNCS 2272.
- [16] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.