

# Improving Logic-Based Testing

Gary Kaminski<sup>a</sup>, Paul Ammann<sup>a</sup>, Jeff Offutt<sup>a,\*</sup>

<sup>a</sup>*Software Engineering, George Mason University, Fairfax VA, USA*

---

## Abstract

Logic-based testers design tests from logical expressions that appear in software artifacts such as source code, design models, and requirements specifications. This paper presents three improvements to logic-based test design. First, in the context of mutation testing, we present fault hierarchies for the six relational operators. Applying the ROR mutation operator causes each relational operator to generate seven mutants per clause. The fault hierarchies show that only three of these seven mutants are needed. Second, we show how to bring the power of the ROR operator to logic-based test criteria such as the widely used Multiple Condition-Decision Coverage (MCDC) test criterion. Third, we present theoretical results supported by empirical data that show that the more recent coverage criterion of minimal-MUMCUT can find significantly more faults than MCDC. The paper has three specific recommendations: (1) Change the way the ROR mutation operator is defined in existing and future mutation systems; (2) Augment logic-based test criteria to incorporate relational operator replacement from mutation; (3) Replace the use of MCDC with minimal-MUMCUT, both in practice and in standards documents like FAA-DO178B.

*Keywords:* Software testing, Logic-based testing, Mutation analysis, MCDC

---

## 1. Introduction

Logical predicates are pervasive among virtually all software artifacts. They are used in decision statements in program source (*if, while, for, switch,*

---

\*Corresponding author

*Email addresses:* [gkaminsk@gmu.edu](mailto:gkaminsk@gmu.edu) (Gary Kaminski), [pammann@gmu.edu](mailto:pammann@gmu.edu) (Paul Ammann), [offutt@gmu.edu](mailto:offutt@gmu.edu) (Jeff Offutt)

...), in UML diagrams, architectural languages, finite state machines, natural language requirements, and formal specifications. Indeed, logical predicates are fundamental to the execution of software. They are also notoriously hard to get correct, and get harder as the decisions become more complicated [1].

Not surprisingly, it is quite common to design tests from logical predicates, called *logic-based testing* in this paper. Logic-based testing is used widely both in academic and industry, and dozens of test design criteria have been developed. This research describes three specific ways to improve logic-based testing. They are related and apply to several logic-based testing criteria.

The first improvement is to the logic-based testing capabilities of mutation analysis. Mutation analysis [9, 10] creates alternate versions of programs (*mutants*) by applying syntax-modifying mutation operators. Testers design tests to cause the mutants to result in incorrect output. Mutation incorporates logic-based testing through specific mutation operators. This research improves on mutation logic-based testing by eliminating redundancy, therefore saving cost.

The second improvement increases the fault-detection strength of logic-based criteria. The most widely known logic-based criterion is MCDC, which is required by the US Federal Aviation Administration to test safety critical systems [29]. This research uses ideas from mutation to strengthen logic-based criteria such as MCDC and its equivalent, ACC [2], by testing at a lower level of abstraction, inside the clauses. Preliminary versions of these two results were published in a prior workshop paper [20].

The third improvement is a demonstration, both theoretical and empirical, of how minimal-MUMCUT [21] is significantly stronger than MCDC. Minimal-MUMCUT is proved to detect more faults than MCDC, with little additional cost. In addition, the improvements based on mutation apply equally well to minimal-MUMCUT. A preliminary version of this result was published in a prior workshop paper [18].

The organization of this paper is as follows. Section 2 provides background in logic-based testing, mutation operators, logic mutation operators, and recent results in logic-based testing. Section 3 explores the relationship between the ROR mutation operator and logic-based criteria, presents a new ROR fault hierarchy, and then a new version of the ROR operator that is just as effective but that produces less than half the number of mutants. Section 4 presents versions of logic-based criteria that are modified with ideas from mutation testing to be more effective and only slightly more expensive in

terms of tests required. Section 5 provides theoretical arguments and empirical evidence that shows that minimal-MUMCUT is significantly stronger than MCDC with only a slight increase in cost. Section 6 offers conclusions and recommendations for future automation.

## 2. Background

This paper uses the following definitions for logical expressions. A *predicate* is an expression that has the value *true* or *false*. Predicates can be assembled from boolean valued *clauses* via boolean operators such as **not** ( $\neg$ ), **and** ( $\wedge$ ), and **or** ( $\vee$ )<sup>1</sup>. For example, if  $B$  is a boolean variable, it is a predicate (as well as a clause).  $a > b$  is a predicate with one clause and  $a > b \wedge b \leq c$  is a predicate with two clauses. Significantly, from the perspective of this paper, the internal structure of the clauses is ignored. That is, in the predicate  $x + y \geq z * 2$ , the arithmetic is ignored and it is treated no differently from  $A \geq B$ .

Mutation analysis [9, 10] creates *mutated* versions of a program, each of which differs from the original by a small syntactic change, then asks the tester to design inputs to *kill* the mutants by causing each mutant to have a different result from the original version. The ability for mutation testing to help testers design high quality tests has always depended directly on the mutation operators. In program-based mutation testing, a *mutation operator* is a rule that specifies changes to syntactic elements in a program. Well designed mutations operators can result in very powerful testing, but poorly designed operators can result in ineffective tests.

Logic-based testing is included in mutation through the *relational operator replacement* (ROR) operator. The classic definition of the *ROR* operator is from the 1991 detailed description of the Mothra mutation system [23]: Each occurrence of a relational operator ( $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $=$ ,  $\neq$ ) is replaced by each other operator and the expression is replaced by *true* and *false*. Most mutation systems (including muJava [26]) since Mothra have implemented these operators following these definitions. The following example shows a simple Java method with the seven mutants generated from the *ROR* operator. The  $\Delta$  notation means the previous statement is replaced with the new statement to create the mutant. For example, the first mutant replaces the original *if* ( $x == y$ ) with *if* ( $x != y$ ).

---

<sup>1</sup>Some papers use the term *decision* for predicates and *condition* for clauses.

---

```

public boolean equal (int x, int y)
{
    if (x == y)
        Δ1 if (x != y)
        Δ2 if (x >= y)
        Δ3 if (x <= y)
        Δ4 if (x > y)
        Δ5 if (x < y)
        Δ6 if (true)
        Δ7 if (false)
    {
        return true;
    }
    return false;
}

```

---

Logic-based test criteria explicitly require tests that satisfy certain truth values on the predicates. The most powerful logic coverage criterion is *combinatorial coverage*, which requires every possible assignment of truth values to clauses. In a predicate with  $n$  clauses, combinatorial coverage requires  $2^n$  tests. While this is practical for small predicates (3 or 4 clauses), cost makes combinatorial coverage impractical for larger predicates.

Other logic coverage criteria ask for a specific subset of the possible  $2^n$  tests defined by combinatorial coverage. *Clause coverage* (sometimes called *condition coverage*) requires each clause to take on the values *true* and *false*, and can be satisfied with just two tests. *Predicate coverage* (sometimes called *decision coverage*) requires the entire predicate to take on the values *true* and *false*, and can be also be satisfied with just two tests. Clause coverage and predicate coverage are both fairly weak, and neither subsumes the other [1].

Modified Condition Decision Coverage (MCDC) [5] and the equivalent Active Clause Coverage (ACC) [1], are widely perceived as powerful test criteria, and MCDC is required for certification of safety critical systems [29]. The idea behind MCDC is that each clause should be tested to be both *true* and *false* under circumstances where the clause “matters,” that is, changing the value of the clause necessarily changes the value of the predicate. Note that MCDC test requirements come in pairs—one requirement for the clause

*true* and one for the clause *false*. MCDC comes in various forms depending on whether each test pair faces additional constraints beyond what is mentioned above, but the details of these forms are independent of the contributions of this paper.

A different approach to logic-based testing is based on fault detection power with respect to the Lau and Yu fault hierarchy [24], as shown in figure 1. Each three letter acronym represents a type of fault that can occur in logical predicates. At the top of the hierarchy is the *literal insertion fault* (LIF), where a literal is inserted into the predicate. For example,  $ad \vee \neg ac \vee be$  might be written as  $adb \vee \neg ac \vee be$ . If a test can detect LIFs, it is guaranteed to detect all the other fault types in the hierarchy except LOF and ORF[.]. The other fault types are *term omission fault* (TOF), *literal reference fault* (LRF), *literal omission fault* (LOF), *operator reference fault-disjunctive* (ORF[+]), *literal negation fault* (LNF), *operator reference fault-conjunctive* (ORF[.]), *term negation fault* (TNF), and *expression negation fault* (ENF). Their full descriptions are given in Lau and Yu’s paper [24].

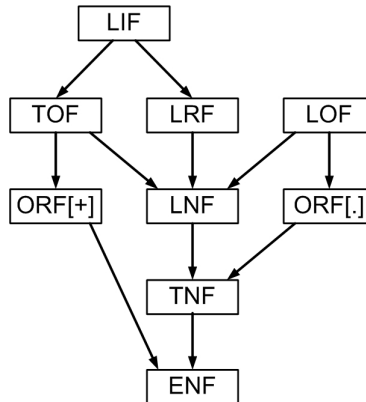


Figure 1: Lau and Yu’s Fault Hierarchy

Despite the widespread use and confidence in MCDC, MCDC tests are only guaranteed to detect TNF and ENF faults [21]! This fault hierarchy has been used to develop alternative logic-based test criteria, the most powerful being MUMCUT [4]. MUMCUT is guaranteed to detect the entire hierarchy of faults. MUMCUT generally only applies to predicates written in Disjunctive Normal Form (DNF), although extensions to more general forms have been explored [30]. As originally defined, MUMCUT generates many unnecessary tests; a problem that has been addressed by several refinements to

MUMCUT [11, 16, 17, 19], resulting in minimal-MUMCUT.

### 3. Improving Logic-Based Testing with Mutation

Mutation operators have been designed for several programming languages, including Fortran 77 [10, 23], C [8], and Java [22, 26]. Jia and Harman recently surveyed mutation analysis for programs and other software engineering artifacts [13]. The statement-level operators have been fairly stable since the Mothra project [23], with the only major change being from the *selective* operator study [27], where it was found that using five Mothra mutation operators would yield tests that killed most other mutants.

However, users of mutation have observed that mutation creates many test requirements (that is, mutants) relative to the number of tests needed by other test criteria. For example, Li *et al.* found that mutation had far more test requirements than the edge-pair, all-uses and prime path criteria, even though it ultimately yielded **fewer** tests [25]. The conclusion from this result is that the mutants somehow “overlap” in the tests needed to kill them, and probably in their ability to find faults. In other words, some mutants appear to be redundant.

The development of fault detection hierarchies, most notably the Lau and Yu’s fault hierarchy in figure 1, offers a way to deal with the problem of redundant mutants. A fault hierarchy describes a *dominance* relation among fault classes. If a test set detects faults in a given class in the hierarchy, those tests are also guaranteed to detect faults in classes that class dominates. From the mutation analysis perspective, this means that if a fault hierarchy is built for a set of mutants, there is no need to generate mutants that are dominated by other mutants.

A closely related approach to increasing the effectiveness of each mutant is the notion of a *subsuming Higher-Order-Mutant* (HOM) [12]. A subsuming HOM is built by combining several mutations in such a way that the combined mutant (the HOM) *subsumes* all of its constituent mutants. A key property of a subsuming HOM is that a test that kills the HOM is also guaranteed to kill the subsumed mutants, and hence these mutants do not need to be generated. Kaminski and Ammann put both of these ideas together in the context of mutation analysis of logical expressions in disjunctive normal form (DNF) [14]. They showed that not only are many redundant mutants generated for logical expressions, worse, these mutants miss many faults in the Lau and Yu fault hierarchy. They further showed how to automatically

construct a small number of subsuming HOMs that guarantee detection of the entire Lau and Yu fault hierarchy and, by subsumption, all mutants generated by a typical mutation analysis tool on the boolean structure of logical expressions.

This section considers the relational operators that commonly appear in boolean expressions. In terms of mutation analysis, this means considering the *relational operator replacement* (ROR) operator. Our first contribution is to develop fault hierarchies for the mutants generated by the ROR operator. These hierarchies show that only three of the seven mutants generated by a single application of an ROR operator are necessary.

### 3.1. ROR and Logic Criteria

This section proves that logic-based testing criteria do **not** subsume ROR mutation testing. For convenience, we use the term *ROR mutation* to mean mutation using just the ROR operator. The strongest logic coverage criterion is *combinatorial coverage*, which requires that a predicate be tested with all combinations of truth values. Consider the two-clause predicate  $a < b \vee c < d$ . Combinatorial coverage requires the tests TT, TF, FT, FF. The following assignments to  $a$ ,  $b$ ,  $c$ , and  $d$  satisfy combinatorial coverage:

a=1, b=2, c=1, d=2 (TT)

a=1, b=2, c=2, d=1 (TF)

a=2, b=1, c=1, d=2 (FT)

a=2, b=1, c=2, d=1 (FF)

However, none of these four tests detects either of the two ROR mutants where  $<$  is replaced by  $\leq$ . To detect the mutant where  $a < b$  is changed to  $a \leq b$ ,  $a$  and  $b$  must have the same value. Thus, combinatorial coverage does not subsume ROR mutation. Combinatorial coverage is the strongest logic criterion and subsumes all others, thus no logic coverage criterion can subsume ROR mutation.

Intuitively, the logic coverage criteria treat each clause as a unit, as if it were a boolean variable, ignoring any structure inside the clause such as the relational operators. The ROR operator, on the other hand, explicitly requires tests to evaluate whether the correct relational operator was used. That is, logic criteria test the *clause* level, whereas the ROR operator tests the *relational* level, a more detailed level of abstraction.

### 3.2. Fault Hierarchies for ROR Mutation

Mutation is widely considered to be expensive, but “expense” can be measured in several ways. Li *et al.* [25] found that although mutation creates more test requirements (that is, mutants) than other test criteria, it does **not** need more tests. The obvious implication from that result is that many mutants are unnecessary or redundant. The selective operator study [27] reduced the number of mutants by an order of magnitude, but mutation systems such as muJava [26] still generate more mutants than are necessary. (Li *et al.*’s study used the muJava selective set of mutation operators.)

This section analyzes the ROR mutation operator on a case by case basis. For each relational operator, the conditions under which mutants created by that operator will be killed are derived. These are called *detection conditions*, which are used to form a hierarchy of mutants.

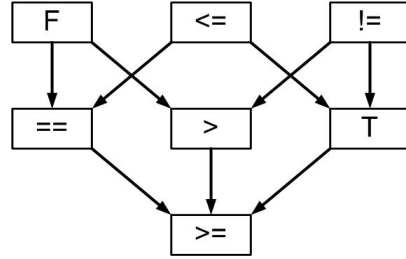
The table in figure 2(A) shows the detection conditions for each mutant of the  $<$  operator. When  $a < b$  is mutated to *false*, the detection condition is  $a < b$  and the value for  $a < b$  must be *true*. Likewise, if the mutant is  $a \leq b$ , the detection condition is  $a == b$  and the value must be *false*.

This leads to the mutant class hierarchy for  $<$  in figure 2(A). The arrows imply a dominance in the sense of Lau and Yu [24]. If a test kills the mutant where  $<$  is replaced by *false*, that test is guaranteed to also kill the mutants where  $<$  is replaced by  $==$  and  $>$ , and by transitivity, the mutant where  $<$  is replaced by  $>=$ . This result follows immediately from the detection conditions. If  $a < b$  is satisfied (the detection condition for the *false* mutant), then  $a \leq b$  and  $a \neq b$  are necessarily satisfied (the detection conditions for  $==$  and  $>$ ). So the false mutant subsumes the  $==$  and  $>$  mutants.

Figures 2(B), 2(C), and 3 show the detection conditions and mutant hierarchies for the other relational operators. These results hold for all programming and specification languages that use relational operators as defined in standard mathematics.

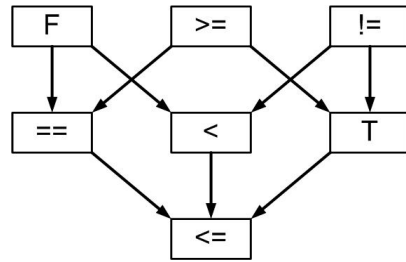
For all six relational operators, we can immediately see that tests that detect three of the ROR mutants are guaranteed to detect all seven ROR mutants. Conversely, if there is no arrow in a hierarchy from one mutant to another, a test that detects the first mutant is guaranteed **not** to detect the other. For example, consider figure 2(A). A test that detects the mutant where  $<$  is replaced with  $\leq$  will never detect the mutant where  $<$  is replaced with  $>$ . Also any test that detects a mutant at the top level of a hierarchy is guaranteed **not** to detect either of the other two mutants at the top level of

Mutant	Detection condition	Value of $a < b$
< replaced by F	$a < b$	T
< replaced by <=	$a == b$	F
< replaced by !=	$a > b$	F
< replaced by ==	$a <= b$	T or F
< replaced by >	$a != b$	T or F
< replaced by T	$a >= b$	F
< replaced by >=	T	T or F



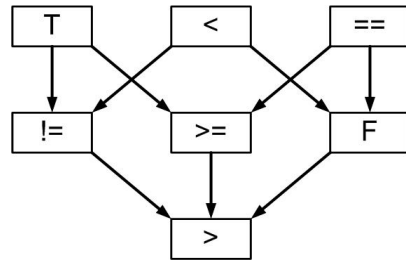
Part A, the < operator

Mutant	Detection condition	Value of $a > b$
> replaced by F	$a > b$	T
> replaced by >=	$a == b$	F
> replaced by !=	$a < b$	F
> replaced by ==	$a >= b$	T or F
> replaced by <	$a != b$	T or F
> replaced by T	$a <= b$	F
> replaced by <=	T	T or F



Part B, the > operator

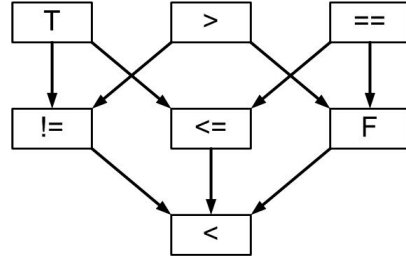
Mutant	Detection condition	Value of $a <= b$
<= replaced by T	$a > b$	F
<= replaced by <	$a == b$	T
<= replaced by ==	$a < b$	T
<= replaced by !=	$a >= b$	T or F
<= replaced by >=	$a != b$	T or F
<= replaced by F	$a <= b$	T
<= replaced by >	T	T or F



Part C, the <= operator

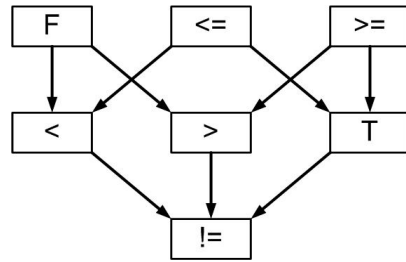
Figure 2: Mutants, Detection Conditions, and Class Hierarchy for <, >, and <=

Mutant	Detection condition	Value of $a \geq b$
$\geq$ replaced by T	$a < b$	F
$\geq$ replaced by $>$	$a == b$	T
$\geq$ replaced by $==$	$a > b$	T
$\geq$ replaced by $!=$	$a \leq b$	T or F
$\geq$ replaced by $\leq$	$a != b$	T or F
$\geq$ replaced by F	$a >= b$	T
$\geq$ replaced by $<$	T	T or F



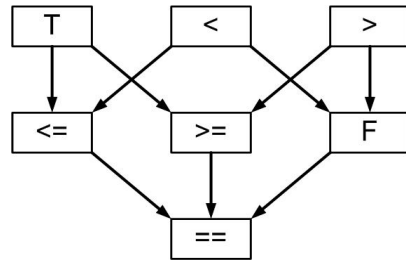
Part A, the  $\geq$  operator

Mutant	Detection condition	Value of $a == b$
$==$ replaced by F	$a == b$	T
$==$ replaced by $\leq$	$a < b$	F
$==$ replaced by $\geq$	$a > b$	F
$==$ replaced by $<$	$a \leq b$	T or F
$==$ replaced by $>$	$a >= b$	T or F
$==$ replaced by T	$a != b$	F
$==$ replaced by $!=$	T	T or F



Part B, the  $==$  operator

Mutant	Detection condition	Value of $a != b$
$!=$ replaced by T	$a == b$	F
$!=$ replaced by $<$	$a > b$	T
$!=$ replaced by $>$	$a < b$	T
$!=$ replaced by $\leq$	$a >= b$	T or F
$!=$ replaced by $\geq$	$a \leq b$	T or F
$!=$ replaced by F	$a != b$	T
$!=$ replaced by $==$	T	T or F



Part C, the  $!=$  operator

Figure 3: Mutants, Detection Conditions, and Class Hierarchy for  $\geq$ ,  $==$ , and  $!=$

the hierarchy. For example, again consider figure 2(A). A test that detects the mutant where  $<$  is replaced with  $!=$  will never detect the mutant where  $<$  is replaced with *false*.

Detecting all ROR mutants guarantees clause coverage. Both T and F appear at least once among the top three rows in the “Value of *a relop b*” column in each table, which achieves clause coverage.

### 3.3. A Cheaper ROR Operator

The detection conditions and mutant hierarchies in section 3.2 lead to an immediate result. The classic definition of the ROR operator [23] should be reformulated to create only the three mutants on top of the mutant hierarchy for the relational operator being mutated. Specifically, the following mutants should be created for each original relational operator. This is an immediate savings of four mutants for each relational operator.

Original Relational Operator	Mutants to Create		
$<$	$<=$	$!=$	<i>false</i>
$>$	$>=$	$!=$	<i>false</i>
$<=$	$<$	$==$	<i>true</i>
$>=$	$>$	$==$	<i>true</i>
$==$	$<=$	$>=$	<i>false</i>
$!=$	$<$	$>$	<i>true</i>

## 4. Improving MCDC Testing with Mutation

The second contribution of this paper is to increase the strength of the logic-based test criteria MCDC. MCDC and ACC [5, 2] test predicates at the clause level. That is, in a predicate  $p = a \wedge b \vee c$ , logic-based criteria test the individual clauses ( $a$ ,  $b$ ,  $c$ ). However, if  $a \equiv (x > y)$ ,  $b \equiv (m <= n)$ , and  $c \equiv (d == e)$ , the logic-based criteria do not test at the lower level of abstraction, inside the clauses. On the other hand, mutation analysis uses the ROR operator to test inside the clauses. The mutations to the relational operators test whether the relations inside the clauses are formulated correctly. This section defines a stronger version of MCDC that leverages the ROR fault hierarchies from section 3.2 to integrate the power of the ROR mutation operator into the logic-based criteria. This result includes an algorithm that

augments an MCDC-adequate test set to also be adequate with respect to RORG, a variant of ROR appropriate for integration with MCDC.

As it turns out, any ROR-adequate test set is guaranteed also to satisfy the weakest version of MCDC [1]. However, the converse is *not* true; an MCDC-adequate test set does not necessarily satisfy ROR coverage. Intuitively, this is because of the difference in the previous paragraph; MCDC treats predicates as boolean functions and ignores relational operators in these expressions. For applications where faults in relational operators are a concern, including ROR can clearly lead to stronger tests.

#### 4.1. Strengthening Logic-Based Criteria

This section introduces a modification of the MCDC criterion to include the additional tests required by the ROR mutation operator to test relational expressions. MCDC coverage of a predicate with  $N$  clauses requires at least  $N + 1$  and no more than  $2N$  tests. Making an MCDC test set ROR-adequate adds at most  $N$  additional tests.

Mutation operators and logic coverage criteria have a theoretical difference that affects this construction. MCDC is considered to be a *semantic* coverage criterion [21], whereas mutation is considered to be *syntactic*. Semantic criteria generate test requirements independently of how the predicates are written, whereas syntactic criteria do not. For example, if the same clause appears twice in a predicate ( $p = a \vee (b \wedge a)$ ), semantic criteria like MCDC and ACC are not affected; when a value for  $a$  is chosen, the same value is used for all occurrences of  $a$ . Mutation, on the other hand, will mutate each occurrence of  $a$  independently. If  $a = a_1 > a_2$ , this means we would have 14 mutants rather than 7, and to kill a mutant, might need the first occurrence of  $a$  to have one value, and the second occurrence a different value.

To successfully integrate a syntactic criterion like ROR mutation with the semantic criterion of MCDC, we need to make certain adaptations. Our approach is to use the concept of a higher-order-mutation operator (HOM) [12], which is a mutation operator that defines more than one change. Our HOM, the Relational Operator Replacement Global (**RORG**), is defined as follows. Consider a boolean expression  $p$  with  $n$  boolean clauses  $c_1, c_2, \dots, c_n$ . Each boolean clause  $c_i$  appears at least once and may appear more than once in  $p$ . For each boolean clause  $c_i$  that uses a relational operator,  $c_i = a_1 \text{ relop } a_2$ , RORG replaces *every* occurrence of  $c_i$  with the same ROR mutations. This means RORG is essentially a semantic, rather than

a syntactic, mutation operator. For example, given the predicate  $p = a1 > a2 \vee (b1 == b2 \wedge a1 > a2)$ , two RORG mutants are:  $p = a1 >= a2 \vee (b1 == b2 \wedge a1 >= a2)$  and  $p = a1 != a2 \vee (b1 == b2 \wedge a1 != a2)$ .

With the RORG adaptation of ROR, we can develop an algorithm that adds RORG-adequacy to an MCDC test set. The idea behind the algorithm is quite simple. We identify tests that satisfy MCDC with respect to each boolean clause  $c$ . The MCDC requirements on these tests are that  $c$  take on the values *true* and *false* under conditions where  $c$  determines the value of predicate  $p$ . We then note that the top three rows of every table in section 3.2 have the same three detection conditions,  $<$ ,  $==$ , and  $>$ . MCDC by itself requires two tests for each clause (*true* and *false*). So at least two of these will have been satisfied by a test where  $c$  determines the value of  $p$ . First, the algorithm identifies whether one of the three detection conditions is not satisfied by the MCDC tests, and which. Then it adds a test to satisfy the third. This process is shown in pseudo-code form in algorithm 1.

**Algorithm Proof Sketch:** The algorithm makes two claims about the resulting test set. The first is that it satisfies MCDC. Since the input test set satisfies MCDC, and no tests are removed from this set, it is clear that the output still satisfies MCDC.

The second claim is that the output test set kills every RORG mutant. We prove this via weak mutation analysis, which is arguably the only generally applicable approach in the context of MCDC. Weak mutation analysis requires *infection* of the subsequent state to kill a mutant. Offutt and Lee [28] found that infection in the context of a mutated predicate is best defined as the final value of the predicate being incorrect. Thus, for each RORG mutant there must be at least one test that causes the predicate to evaluate to the wrong value. Inspection of the detection conditions in figures 2 and 3 shows that to kill all ROR mutants, the relation between  $c_1$  and  $c_2$  in  $c = c_1 \text{ relop } c_2$  must have all the three possible values, namely:  $c_1 < c_2$ ,  $c_1 == c_2$ , and  $c_1 > c_2$ . For these detection conditions to cause predicate  $p$  to evaluate to a different result, it is necessary that  $c$  determines  $p$ . Tests in  $T_c$  satisfy the constraint that  $c$  determines  $p$  by construction. In addition, any new test (possibly) generated by the algorithm for clause  $c$  necessarily also satisfies the constraint that  $c$  determines  $p$ . The algorithm forces the three possible relations between  $c_1$  and  $c_2$  for tests where  $c$  determines  $p$ . Hence RORG is satisfied.  $\square$

Note that the resulting test set does *not* necessarily satisfy ROR. The reason is that if we consider each clause  $c$  syntactically as it appears in the

---

**Algorithm 1** Algorithm to Make MCDC Test Set RORG-Adequate

---

**Require:** Predicate  $p$  and a test set  $T$  that satisfies MCDC (ACC) with respect to  $p$

**Ensure:** A test set that still satisfies MCDC (ACC), but is now also RORG-adequate

```
1: // It does not matter which version of ACC is satisfied
2: // (GACC, CACC, or RACC), or whether masking or
3: // non-masking MCDC is used.
4: for each clause  $c$  in  $p$  do
5:   if  $c$  contains a relational operator  $relop$  then
6:     Identify  $T_c$ , the tests for which  $c$  determines  $p$ 
7:     // Clause  $c$  determines predicate  $p$  if changing
8:     // the value of  $c$ , while leaving all other
9:     // clauses unchanged, changes the value of  $p$  [1].
10:    //  $T_c$  will have at least two tests and possibly more.
11:    //  $c$  will have the value true for at least one test
12:    // and false for at least one test.
13:    Assume  $c = c_1 relop c_2$ 
14:    // We need three tests,  $c_1 < c_2$ ,  $c_1 == c_2$ , and
15:    //  $c_1 > c_2$ , and are assured of having at least two.
16:    for each test  $t_i$  in  $T_c$  do
17:      isCovered['<'] = isCovered['=='] = isCovered['>'] = false
18:      for each  $relop$  in {<, ==, >} do
19:        if  $c_1 relop c_2$  is true for  $t_i$  then
20:          isCovered[' $relop$ '] = true
21:        end if ( $c_1 relop c_2$  is true)
22:      end for (each  $relop$ )
23:    end for (each test)
24:    for each  $relop$  in {<, ==, >} do
25:      if isCovered[' $relop$ '] == false then
26:        Construct a new test by modifying an arbitrary test in  $T_c$ . Leave
        all other variables alone, but change the values for the variables
        in  $c$  so that  $c_1 relop c_2$  is true
27:      end if (isCovered[] == false)
28:    end for (each  $relop$ )
29:  end if ( $c$  contains a  $relop$ )
30: end for (each clause)
```

---

predicate, there is no guarantee that the original MCDC test set has *any* tests in the set  $T_c$ . However, for RORG, each  $T_c$  is guaranteed to always be at least of size 2, and, further, to satisfy predicate coverage. Complexity is on the order of the product of the number of tests and the number of clauses. If the test set is specifically optimized for MCDC, the size of the test set is linear in the number of clauses. Hence, in this case, the algorithm is quadratic in the number of clauses.

#### 4.2. Logic-Based Testing with RORG Example

Consider the predicate  $p = a \wedge b \vee c$ , where  $a = (a_1 < a_2)$ ,  $b = (b_1 \leq b_2)$ , and  $c = (c_1 \neq c_2)$ .

The following test set satisfies the most restrictive MCDC coverage criterion (RACC):  $T = \{t_1, t_2, t_3, t_4\} = \{TTF, TFT, TFF, FTF\}$ . These tests are refined to have the value assignments:

$$t_1: a_1 = 5, a_2 = 6, b_1 = 10, b_2 = 11, c_1 = 21, c_2 = 21$$

$$t_2: a_1 = 5, a_2 = 6, b_1 = 11, b_2 = 10, c_1 = 21, c_2 = 22$$

$$t_3: a_1 = 5, a_2 = 6, b_1 = 11, b_2 = 10, c_1 = 21, c_2 = 21$$

$$t_4: a_1 = 6, a_2 = 5, b_1 = 10, b_2 = 11, c_1 = 21, c_2 = 21$$

We first consider clause  $a$  from line 4 in algorithm 1. The set  $T_a$  of tests where  $a$  determines  $p$  is  $\{t_1, t_4\}$ . Test  $t_1$  satisfies  $a_1 < a_2$  and  $t_4$  satisfies  $a_1 > a_2$ , so the algorithm adds a new test (in line 26) to satisfy  $a_1 == a_2$ . Any test for which  $a$  determines  $p$  will do, so the algorithm starts with  $t_1$ , and modifies the values for  $a_1$  and  $a_2$  to be  $a_1 = 5$  and  $a_2 = 5$ .

For clause  $b$ , the set  $T_b$  of tests where  $b$  determines  $p$  is  $\{t_1, t_3\}$ . Test  $t_1$  satisfies  $b_1 < b_2$  and  $t_3$  satisfies  $b_1 > b_2$ , so the algorithm adds a new test (in line 26) to satisfy  $b_1 == b_2$ . Any test for which  $b$  determines  $p$  will do, so the algorithm starts with  $t_1$ , and modifies the values for  $b_1$  and  $b_2$  to be  $b_1 = 10$  and  $b_2 = 10$ .

Finally, for clause  $c$ , the set  $T_c$  of tests where  $c$  determines  $p$  is  $\{t_2, t_3, t_4\}$ . Test  $t_2$  satisfies  $c_1 < c_2$ ,  $t_3$  satisfies  $c_1 == c_2$ , and  $t_4$  also satisfies  $c_1 == c_2$ , so the algorithm adds a new test (in line 26) to satisfy  $c_1 > c_2$ . Any test for which  $c$  determines  $p$  will do, so the algorithm starts with  $t_2$ , and modifies the values for  $c_1$  and  $c_2$  to be  $c_1 = 22$  and  $c_2 = 21$ .

The resulting MCDC/RORG-adequate test set is shown in table 1. The last three tests marked “New” are added by the algorithm. The original tests they were derived from are included in parentheses, and the modified values are shown in bold. The additional detection conditions that were covered are shown in the last three columns.

Table 1: Example of expanding MCDC tests to be RORG adequate

<b>Test</b>	<b>Value</b>	$a_1$	$a_2$	$b_1$	$b_2$	$c_1$	$c_2$	$a$	$b$	$c$
$t_1$	TTF	5	6	10	11	21	21	<	<	
$t_2$	TFT	5	6	11	10	21	22			<
$t_3$	TFF	5	6	11	10	21	21		>	==
$t_4$	FTF	6	5	10	11	21	21	>		
New	$(t_1)$	<b>5</b>	<b>5</b>	10	11	21	21	==		
New	$(t_1)$	5	6	<b>10</b>	<b>10</b>	21	21		==	
New	$(t_2)$	5	6	11	10	<b>22</b>	<b>21</b>			>

## 5. Improving Logic-Based Testing by Detecting More Faults

MCDC has long been considered the “gold standard” for testing complex predicates in critical systems. It is widely regarded as striking a good balance between a manageable number of test cases and a reasonably thorough probing of the predicate under test. This section makes an argument that undercuts the primacy of MCDC for critical systems. The first part of this argument is empirical; we give data to show just how poorly MCDC tests find faults on predicates extracted from software in the avionics domain. In contrast, minimal-MUMCUT is guaranteed to detect all the logic faults in the Lau and Yu fault hierarchy with a test set that is only slightly larger. The second part of the argument revolves around adding ROR-adequacy to minimal-MUMCUT test sets. An interesting observation on this procedure is that making minimal-MUMCUT ROR-adequate potentially requires fewer additional tests than making MCDC RORG-adequate. Hence, the advantage that MCDC has in terms of number of tests is less when ROR mutation is added.

### 5.1. An Empirical Argument for Preferring minimal-MUMCUT to MCDC

Chilenski [6] extracted 20,256 logical expressions from the airborne software of five different Line Replaceable Units (LRUs, also known as black boxes). The five LRUs belonged to five different airborne systems drawn from two different software models of airplanes (two from one model, three from another). We examined the minimal DNF logical expressions from the same airborne software that contains at least five unique literals, and used the unique-cause version of MCDC [7], RACC [1]. Predicates that have fewer than five literals have fewer possible faults, and a RACC test set will not have

noticeable savings over combinatorial coverage. Unless a criterion requiring combinatorial coverage is used, it is only at five unique literals that a RACC test set can possibly be less than one third of the test set size of combinatorial coverage. These predicates were then analyzed in terms of RACC fault detection with respect to the faults in the Yau and Lu fault hierarchy. The same predicate often occurred multiple times in the source, so the expressions only contained 132 distinct predicates

69% of the predicates had only AND or OR operators, but not both, so RACC tests can detect all faults in the hierarchy. 26% of the predicates had both an AND and an OR operator, but literals did not repeat in different terms and RACC was feasible, so RACC tests can detect all but about half the LIF faults. RACC was feasible for all predicates. For the remaining 5% of the predicates, literals repeat in different terms. A RACC test set for each of these predicates was built, and their ability to detect faults was assessed. Table 2 shows the results on these final seven predicates.

Table 2: RACC Fault Detection by Predicate

Predicate	Literals	Total Faults	Faults Found	Percent
1	5	81	55	68%
2	5	88	78	89%
3	5	88	78	89%
4	5	88	78	89%
5	7	164	130	79%
6	9	249	205	82%
7	13	452	357	79%
Average		173	140	81%

Table 2 shows data from RACC tests designed for the seven remaining predicates. The Literals column shows how many literals each predicate has. The Total Faults column shows how many faults were seeded by using the Lau and Yu fault hierarchy in figure 1. The Faults Found column shows the number of faults found by the RACC tests, and Percent shows the percentage of faults found. The Average row is unweighted. All predicates in table 2 contained either four or five terms.

The next set of predicates we looked at were 19 minimal DNF predicates from an air traffic collision avoidance system (TCAS). Chen, Lau, and Yu [4] used the greedy MUTP algorithm [3] to see how many tests would be needed

to satisfy MUMCUT on these predicates. There were originally 20 predicates but number 12 was excluded due to a missing right parenthesis [31]. The predicates have from 5 to 13 unique literals. However, these predicates were larger, having up to 25 terms, which allowed the authors to examine how increasing the number of terms literals repeat in affects RACC fault detection. RACC tests were created for each predicate and fault detection for each was assessed.

Table 3 gives the number of faults detected by the RACC test sets for the TCAS predicates. The columns are the same as in table 2. Each of the nine DNF fault types were injected in every place possible in each of the 19 predicates.

Table 3: More RACC Fault Detection by Predicate

Predicate	Literals	Total Faults	Faults Found	Percent
1	7	173	79	46%
2	9	548	108	20%
3	12	2493	583	23%
4	5	71	61	86%
5	9	483	267	55%
6	11	342	72	21%
7	10	524	158	30%
8	8	104	44	42%
9	7	46	31	67%
10	13	576	268	47%
11	13	1047	267	26%
12*	N/A	N/A	N/A	N/A
13	12	397	336	85%
14	7	236	162	69%
15	9	599	275	46%
16	12	1980	502	25%
17	11	524	240	46%
18	10	596	274	46%
19	8	212	108	51%
20	7	68	56	82%
Average		580	205	35%

\*Number 12 excluded due to a missing a right parenthesis

Table 4 shows the number of faults found, grouped by the number of

unique literals. As the number of unique literals increases, the RACC tests detected a smaller percentage of faults. Intuitively, as the number of unique literals increases, there are more opportunities for literals to repeat in different terms. As more literals repeat, the RACC test set size will not increase so the additional faults caused by repeated literals are likely to be undetected.

Table 4: RACC Fault Detection Grouped by Unique Literals

Unique Literals	Total Faults	Faults Found	Percent
5	71	61	86%
7*	523	328	63%
8	316	152	48%
9	1630	650	40%
10	1120	432	39%
11	956	312	33%
12	4870	1421	29%
13	1623	535	33%

\*None of the predicates examined had 6 unique literals

As said in section 2, RACC tests are only guaranteed to detect faults of type ENF and TNF [21]. However, in practice, RACC tests do indeed detect some other faults, even though they are not guaranteed to detect such faults. Thus, another question connects theory to practice: what percentage of such faults does RACC actually detect in practice? The answer in this study was that RACC tests actually detected 34% of the faults in the TCAS predicates that RACC tests are neither guaranteed to detect nor guaranteed to not detect. This is close to the 35% given in table 3 because ENF and TNF represent a small percentage of all the faults. That is, only one ENF is possible per predicate and the number of TNFs is equal to the number of terms. The other fault types are much more numerous. The number of tests for Minimal-MUMCUT [15] is about four times the number of tests for RACC for the 19 TCAS predicates. Thus, fault detection can be improved from 35% to 100%, but at a cost of increasing test set size by a factor of four.

### 5.2. Improving Minimal-MUMCUT Testing with Mutation

This section introduces a modification of the minimal-MUMCUT criterion to include the additional tests required by the ROR mutation operator. Since minimal-MUMCUT is a syntactic coverage criterion, we do not need

to adapt the ROR operator to RORG, as we did for MCDC. Instead, we use ROR directly, without change. As an aside, although we develop this section in terms of minimal-MUMCUT, the details of the minimal-MUMCUT algorithm, which are quite complex, do not matter here. In other words, in the following discussion, the algorithm and results are exactly the same if minimal-MUMCUT is replaced with any other test criterion guaranteed to detect the faults in the Lau and Yu fault hierarchy<sup>2</sup>.

The algorithm used to modify minimal-MUMCUT tests is very similar to algorithm 1 for modifying MCDC tests. To kill ROR mutants, we identify relevant subsets of the minimal-MUMCUT test set, and then, for each subset, determine if additional tests are needed. Specifically, for each occurrence of a literal  $c$  in the DNF predicate under test, we identify the minimal-MUMCUT tests that detect the mutants where  $c$  is replaced by *true*. Call these tests  $T_{c=true}$ . We repeat the process to identify tests that detect the mutants where  $c$  is replaced by *false*. Call these tests  $T_{c=false}$ . Each of these sets is guaranteed to be non-empty. The reason is that replacing a literal  $c$  by *true* results in a LOF, and replacing a literal  $c$  by *false* results in a TOF. Hence the non-emptiness of the sets  $T_{c=true}$  and  $T_{c=false}$  follows from the fact that minimal-MUMCUT tests sets detect all LOF and TOF faults.

We again note that the top three rows of every table in section 3.2 have the same three detection conditions,  $<$ ,  $=$ , and  $>$ . Tests in the set  $T_{c=true}$  are guaranteed to satisfy at least one of these detection conditions, and tests in the set  $T_{c=false}$  are guaranteed to satisfy at least one more detection condition. The algorithm determines whether, through serendipity, the third detection condition is also satisfied. If not, it adds a test to satisfy the this condition. This process is shown in pseudo-code form in algorithm 2.

**Algorithm Proof Sketch:** The algorithm makes two claims about the resulting test set. The first is that it satisfies minimal-MUMCUT. Since the input test set satisfies minimal-MUMCUT, and no tests are removed from this set, it is clear that the output still satisfies minimal-MUMCUT.

The second claim is that the output test set kills every ROR mutant. Again, we base our argument on weak mutation testing. As before, the detection conditions in figures 2 and 3 show that to kill all ROR mutants, the

---

<sup>2</sup>A tool to generate minimal-MUMCUT tests for arbitrary minimal DNF expressions is available at the author web site for the authors' textbook [1]: <http://cs.gmu.edu/~offutt/softwaretest/>.

---

**Algorithm 2** Algorithm to Make minimal-MUMCUT Test Set ROR-Adequate

---

**Require:** Predicate  $p$  and a test set  $T$  that satisfies minimal-MUMCUT with respect to  $p$

**Ensure:** A test set that still satisfies minimal-MUMCUT but is now also ROR-adequate

```
1: for each literal  $c$  in  $p$  do
2:   if  $c$  contains a relational operator  $relop$  then
3:     Identify  $T_{c=true}$ , the tests that kill the ROR mutant when  $c$  is re-
4:     placed by  $true$ 
5:     Identify  $T_{c=false}$ , the tests that kill the ROR mutant when  $c$  is re-
6:     placed by  $false$ 
7:     //  $T_{c=true}$  has at least one test and possibly more.
8:     //  $T_{c=false}$  has at least one test and possibly more.
9:     Let  $T_c$  equal  $T_{c=true} \cup T_{c=false}$ 
10:    Assume  $c = c_1 relop c_2$ 
11:    // We need three tests,  $c_1 < c_2$ ,  $c_1 == c_2$ , and
12:    //  $c_1 > c_2$ , and are assured of having at least two.
13:    for each test  $t_i$  in  $T_c$  do
14:      isCovered['<'] = isCovered['=='] = isCovered['>'] =  $false$ 
15:      for each  $relop$  in {<, ==, >} do
16:        if  $c_1 relop c_2$  is  $true$  for  $t_i$  then
17:          isCovered['relop'] =  $true$ 
18:          end if ( $c_1 relop c_2$  is  $true$ )
19:        end for (each  $relop$ )
20:      end for (each test)
21:      for each  $relop$  in {<, ==, >} do
22:        if isCovered['relop'] ==  $false$  then
23:          Construct a new test by modifying an arbitrary test in  $T_c$ . Leave
24:          all other variables alone, but change the values for the variables
25:          in  $c$  so that  $c_1 relop c_2$  is  $true$ 
26:          end if (isCovered[] ==  $false$ )
27:        end for (each  $relop$ )
28:      end if ( $c$  contains a  $relop$ )
29:    end for (each literal)
```

---

relation between  $c_1$  and  $c_2$  in  $c = c_1 \text{ relop } c_2$  must have all three possible values, namely:  $c_1 < c_2$ ,  $c_1 == c_2$ , and  $c_1 > c_2$ . Further, these detection conditions must occur on tests that cause the predicate  $p$  to evaluate to a different result, which is exactly the property of tests in set  $T_c = T_{c=true} \cup T_{c=false}$ . In addition, any new test (possibly) generated by the algorithm for literal  $c$  necessarily also causes  $p$  to evaluate to a different result. The algorithm forces the three possible relations between  $c_1$  and  $c_2$  for tests where  $p$  evaluates to a different result. Hence ROR is satisfied.  $\square$

The complexity analysis is as follows. Identifying  $T_c$  for each literal  $c$  requires examining each test in the minimal-MUMCUT test set. Hence, overall complexity is linear in the product of the number of tests and the number of literals.

**Example:** Consider the predicate  $p = a \wedge b \vee c$ , where  $a = (a_1 < a_2)$ ,  $b = (b_1 \leq b_2)$ , and  $c = (c_1 \neq c_2)$  used in the MCDC example in section 4.1. Predicate  $p$  is already in DNF. The test set from the MCDC example,  $\{t_1, t_2, t_3, t_4\} = \{TTF, TFT, TFF, FTF\}$ , does not satisfy minimal-MUMCUT for  $p$ . The reason is that for term  $c$ , an additional test is needed to satisfy the MUTP part of the minimal-MUMCUT criterion. Augmenting the test set with  $t_5 = FTT$  is sufficient, so define  $T = \{t_1, t_2, t_3, t_4, t_5\}$ . Suppose that, in terms of values, these tests are refined to have the value assignments:

$$\begin{aligned} t_1: a_1 = 5, a_2 = 6, b_1 = 10, b_2 = 11, c_1 = 21, c_2 = 21 \\ t_2: a_1 = 5, a_2 = 6, b_1 = 11, b_2 = 10, c_1 = 21, c_2 = 22 \\ t_3: a_1 = 5, a_2 = 6, b_1 = 11, b_2 = 10, c_1 = 21, c_2 = 21 \\ t_4: a_1 = 6, a_2 = 5, b_1 = 10, b_2 = 11, c_1 = 21, c_2 = 21 \\ t_5: a_1 = 6, a_2 = 5, b_1 = 10, b_2 = 11, c_1 = 22, c_2 = 21 \end{aligned}$$

Tests  $t_1$  through  $t_4$  have the same values as in the MCDC example, and test  $t_5$  is built as a carefully selected variation in terms of values for variables  $c_1$  and  $c_2$ . We will say more about the selection criteria later.

In terms of algorithm 2, the set  $T_{a=true}$  on line 3 is  $\{t_4\}$ , the set  $T_{a=false}$  on line 4 is  $\{t_1\}$ , and hence the set  $T_a$  on line 7 is  $\{t_1, t_4\}$ . Similarly, the set  $T_{b=true}$  on line 3 is  $\{t_3\}$ , the set  $T_{b=false}$  on line 4 is  $\{t_1\}$ , and hence the set  $T_b$  on line 7 is  $\{t_1, t_3\}$ . Since these values are exactly those that were derived for  $T_a$  and  $T_b$  in the MCDC example, and since the remainder of the algorithm is the same, the results of the two algorithms are identical with respect to literals  $a$  and  $b$ . Hence, we do not elaborate them further.

The algorithm's treatment of literal  $c$  differs from the MCDC example. The set  $T_{c=true}$  on line 3 is again  $\{t_3, t_4\}$ , but the set  $T_{c=false}$  on line 4 is

$\{t_2, t_5\}$ . Hence the set  $T_c$  on line 7 is  $\{t_2, t_3, t_4, t_5\}$ . Upon reaching line 20 of algorithm 2, the algorithm finds that all relational operators are covered, and hence no additional tests are needed.

In general, minimal-MUMCUT test sets are larger than MCDC tests. While this might seem to be a liability, strengthening the test set to address relational operators can blunt the difference in test set size. In other words, by choosing values carefully, it is possible to make a minimal-MUMCUT test set ROR-adequate without adding as many tests as are needed to make an MCDC test set RORG-adequate. This was illustrated in the example with respect to literal  $c$ . Even though the MCDC example originally required four tests, three additional tests were required to make the test set RORG-adequate. For minimal-MUMCUT, five tests are required, but only two additional tests are needed to make the test set ROR-adequate. Thus, both MCDC/RORG and minimal-MUMCUT/ROR need seven tests for this example, but the MCDC/RORG tests do **not** detect all faults in the Lau and Yu fault hierarchy. In particular, the MCDC/RORG tests detect neither the  $a \wedge b \vee a \wedge c$  LIF nor the  $a \wedge b \vee !b \wedge c$  LIF. Of course, the minimal-MUMCUT/ROR test set detects all LIFs, including these.

## 6. Recommendations and Conclusions

This paper presents three new results, a way to reduce the number of mutants generated for the ROR operator by eliminating redundancy among mutants, a way to strengthen logic criteria such as MCDC by using the ROR mutation operator to increase precision, and theoretical results supported by empirical data that show that minimal-MUMCUT is significantly stronger than the widely used MCDC.

The first result can be used to improve mutation tools by reducing the number of mutants generated. The traditional ROR operator creates seven mutants for each relational operator, of which four are provably redundant. We are currently modifying muJava's [26] ROR operator in this way. We believe it is likely that other mutation operators create similarly redundant mutants, and similar analysis could greatly reduce the number of mutants created by mutation systems. In mutation testing, each mutant represents a test requirement, so reducing the number of mutants created can have a major impact on the automation of mutation testing.

The second result can be used to strengthen logic test criteria. Logic testing criteria have traditionally only looked at the clause level, treating

each clause as a simple boolean variable. Part of the power of mutation stems from the fact that it looks inside clauses, and tries to determine whether a clause such as  $a > b$  is correctly formulated. By adding one more test for each clause, and using the redundancy proofs for the ROR operator, this paper shows how logic criteria can be extended to gain this power. The algorithm in section 4 shows how this technique can easily be incorporated into an automated test tool.

This result is particularly significant for the MCDC criterion [5], because it is mandated for certain safety-critical software components on aircrafts and air traffic controllers by the US Federal Aviation Administration [29]. The simple extension to MCDC presented in this paper has the ability to strengthen testing of this crucial kind of software, potentially making air travel safer. Although the algorithm is definitive in terms of adding RORG-adequacy to an MCDC-adequate test set, empirical studies are needed to determine how much augmenting MCDC test sets with RORG-adequacy improves fault detection. The algorithm can also be used in the automation of testing virtually any kind of control software, much of which makes heavy use of logical predicates and much of which is safety critical.

The third result is related directly to the MCDC criterion. Although MCDC is widely known, used in safety-critical domains, and required by the US Federal Aviation Administration, the results in this paper show conclusively that minimal-MUMCUT can find far more faults.

Thus, this research leads us to three recommendations:

1. Change the way the ROR mutation is defined in existing and future mutation systems.
2. Augment logic-based test criteria (such as MCDC, minimal-MUMCUT, and ACC) to incorporate relational operator replacement from mutation.
3. Replace the use of MCDC with minimal-MUMCUT, both in practice and in standards documents like FAA-DO178B [29].

## References

- [1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2008. ISBN 0-52188-038-1.
- [2] Paul Ammann, Jeff Offutt, and Hong Huang. Coverage criteria for logical expressions. In *Proceedings of the 14th International Symposium on*

*Software Reliability Engineering*, pages 99–107, Denver, CO, November 2003. IEEE Computer Society Press.

- [3] T. Y. Chen and M. F. Lau. An empirical study on the effectiveness of the greedy MUTP criterion. In *International Conference on Software Engineering: Education and Practice*, pages 338–344, January 1998.
- [4] T. Y. Chen, M. F. Lau, and Y. T. Yu. MUMCUT: A fault-based strategy for testing boolean specifications. In *APSEC '99: Proceedings of the Sixth Asia Pacific Software Engineering Conference*, pages 606–613, Takamatsu, Japan, 1999. IEEE Computer Society Press.
- [5] Jay Jay Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, September 1994.
- [6] John Chilenski. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical report DOT/FAA/AR-01/18, April 2001.
- [7] 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/>.
- [8] Márcio E. Delamaro and José C. Maldonado. Proteum-A tool for the assessment of test adequacy for C programs. In *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*, pages 79–95, New Brunswick, NJ, July 1996.
- [9] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [10] Richard A. DeMillo and Jeff Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
- [11] Angelo Gargantini and Gordon Fraser. Generating minimal fault detecting test suites for boolean expressions. In *AMOST 2010 - 6th Workshop on Advances in Model Based Testing*, pages 37–45, Paris, France, April 2010.

- [12] Yue Jia and Mark Harman. Constructing subtle faults using higher order mutation testing. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 249–258, Beijing, September 2008.
- [13] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions of Software Engineering*, 37(5):649–678, September 2011.
- [14] Garrett Kaminski and Paul Ammann. Using a fault hierarchy to improve the efficiency of DNF logic mutation testing. In *2nd IEEE International Conference on Software Testing, Verification and Validation (ICST 2009)*, pages 386–395, Denver, CO, April 2009.
- [15] Garrett Kaminski and Paul Ammann. Using logic criterion feasibility to reduce test set size while guaranteeing double fault detection. In *Fifth Workshop on Mutation Analysis (IEEE Mutation 2009)*, pages 167–176, Denver, CO, April 2009.
- [16] Garrett Kaminski and Paul Ammann. Using logic criterion feasibility to reduce test set size while guaranteeing fault detection. In *2nd IEEE International Conference on Software Testing, Verification and Validation (ICST 2009)*, pages 356–365, Denver, CO, April 2009.
- [17] Garrett Kaminski and Paul Ammann. Applications of optimization to logic testing. In *CSTVA 2010 - 2nd Workshop on Constraints in Software Testing, Verification and Analysis*, pages 331–336, Paris, France, April 2010.
- [18] Garrett Kaminski and Paul Ammann. Applying MCDC to large DNF logic expressions. In *International Conference on Software Engineering Research and Practice*, pages 411–417, Las Vegas NV, July 2010.
- [19] Garrett Kaminski and Paul Ammann. Reducing logic test set size while preserving fault detection. *Journal of Software Testing, Verification and Reliability*, Wiley, 21(3):155–193, September 2011. Special issue from the 2009 International Conference on Software Testing, Verification and Validation.

- [20] Garrett Kaminski, Paul Ammann, and Jeff Offutt. Better predicate testing. In *Sixth Workshop on Automation of Software Test (AST 2011)*, pages 57–63, Honolulu HI, USA, May 2011.
- [21] Garrett Kaminski, Greg Williams, and Paul Ammann. Reconciling perspectives of logic testing for software. *Journal of Software Testing, Verification and Reliability, Wiley*, 18(3):149–188, September 2008.
- [22] Sunwoo Kim, John A. Clark, and John A. McDermid. Investigating the effectiveness of object-oriented strategies with the mutation method. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 4–100, San Jose, CA, October 2000. Wiley’s Software Testing, Verification, and Reliability, December 2001.
- [23] Kim N. King and Jeff Offutt. A Fortran language system for mutation-based software testing. *Software-Practice and Experience*, 21(7):685–718, July 1991.
- [24] M. F. Lau and Y. T. Yu. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering Methodology*, 14(3):247–276, July 2005.
- [25] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *Fifth Workshop on Mutation Analysis (IEEE Mutation 2009)*, Denver CO, April 2009.
- [26] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava : An automated class mutation system. *Software Testing, Verification, and Reliability, Wiley*, 15(2):97–133, June 2005.
- [27] Jeff Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.
- [28] Jeff Offutt and Stephen D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

- [29] RTCA-DO-178B. Software considerations in airborne systems and equipment certification, December 1992.
- [30] ChangAi Sun, Yunwei Dong, R. Lai, K. Y. Sim, and T. Y. Chen. Analyzing and extending MUMCUT for fault-based testing of general boolean expressions. In *The Sixth IEEE International Conference on Computer and Information Technology*, pages 184–189, Seoul, Korea, September 2006.
- [31] Elaine Weyuker, Thomas Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.