Fault Classes and Error Detection Capability of Specification Based Testing

D. Richard Kuhn
National Institute of Standards and Technology
Gaithersburg, Maryland 20899-8970 USA

Some varieties of specification based testing rely upon methods for generating test cases from predicates in a software specification [1] [2] [3] [4] [5] [6] [7]. These methods derive various test conditions from logic expressions, with the aim of detecting different types of faults. Some authors have presented empirical results on the ability of specification based test generation methods to detect failures [1] [2] [7] [8]. This paper describes a method for computing the conditions that must be covered by a test set for the test set to guarantee detection of the particular fault class. It is shown that there is a coverage hierarchy to fault classes that is consistent with, and may therefore explain, experimental results on fault based testing. The method is also shown to be effective for computing MCDC-Adequate [9] tests.

Categories and Subject Descriptors: D.2.4 [Software]: software engineering—program verification; D.2.1 [Software]: software engineering—requirements/specifications; D.2.5 [Software]: software engineering—testing and debugging

General Terms: THEORY, VERIFICATION, TESTING

1. INTRODUCTION

A number of methods have been proposed for generating test cases from predicates in a specification or program [1] [2] [3] [4] [5] [6] [7] [10] [11]. These methods derive various test conditions from logic expressions, with the aim of detecting a variety of fault types. This approach is analogous to standard digital circuit test methods [12]. In circuit testing, typical manufacturing flaws are hypothesized, then test sets are derived to detect these flaws.

With software, the situation is similar, but the set of possible fault classes is much larger. Because the difference between an implementation and its specification is the result of human error, some types of faults may be virtually impossible to predict in advance. Nevertheless, some fault classes can be hypothesized and test sets can be constructed to detect them. The fault classes defined in [2], [11], and [13] are the following: Variable Reference Fault - a boolean variable \( z \) is replaced by another variable \( y \), \( z \neq y \); Variable Negation Fault - a boolean variable \( z \) is replaced by \( \overline{z} \); Expression Negation Fault - a boolean expression \( p \) is replaced by \( \overline{p} \); Associative Shift Fault - a boolean expression is replaced by one in which the association between variables is incorrect, e.g., \( z \land (y \lor z) \) replaced by \( z \land y \lor z \); Operator Reference Fault - a boolean operator is replaced by another, e.g., \( z \land y \) replaced by \( z \lor y \). Additional types of faults are defined in [8]: Incorrect relational operators, Incorrect parentheses, Incorrect arithmetic expression, Extra binary operators, Missing binary operators. Experimental results have been used in evaluating the effectiveness of various test generation methods [1], [2], [7], [8], [14], although the fault classes considered are all derived from specifications and some
implementation faults may not fit neatly into these categories.

In this paper, the conditions under which a particular fault class will cause a failure for a given predicate are calculated. We show that the calculated conditions must be covered by a test set for the test set to guarantee detection of the particular fault class. By deriving the conditions under which various fault classes will cause a failure, we show that there is a hierarchy to fault classes. The ordering of the hierarchy matches the ordering of effectiveness of fault-based testing techniques established in empirical studies by Weyuker et al. [2], and Vouk et al. [8]. Thus the fault hierarchy helps to explain experimental results on fault based testing.

2. DETECTION CONDITIONS

The detection conditions for a predicate \( P \) are the conditions under which a change to \( P \) will affect the value of the predicate \( P \). A test will detect a failure if and only if a faulty predicate \( P' \) evaluates to a different value than the correct predicate \( P \). That is, where \( \neg (P \Leftrightarrow P') \), or \( P \oplus P' \), where \( \oplus \) is exclusive-or.

This is simply the boolean difference of \( P \) with respect to \( P' \) [15] [16], also called the boolean derivative [17] [18], or predicate difference [19] when \( P \) contains expressions rather than strictly boolean terms. To determine, for example, the conditions under which a variable negation fault for variable \( v \) will be detected, we simply compute \( P \oplus P^v \), where \( P^v \) is predicate \( P \) with all free occurrences of variable \( x \) replaced by expression \( e \). \( (P^v \) may also be written as \( P[x := e] \).

Other types of faults can be analyzed in the same way, letting \( P' \) be the predicate \( P \) with the fault inserted. Given a particular fault hypothesized for a particular specification, it is possible to compute the conditions under which the fault will cause a failure, i.e., conditions under which the fault will cause the expression to evaluate to a different value than if the fault had not occurred. For example, suppose the specification is \( S = p \land q \lor r \). We can compute the conditions under which a variable negation fault for variable \( q \) will cause a failure, by computing the boolean difference:

\[
\begin{align*}
\delta S^q_f &= (p \land \neg q \lor r) \oplus (p \land q \lor r) \\
&= p \land r
\end{align*}
\]

Weyuker, Goradia and Singh [2] describe an algorithm that computes test conditions for detecting variable negation faults, and propose various strategies to generate data for these conditions. Although their algorithm was designed to detect variable negation faults, Weyuker et al. show that their approach detects other fault types as well.

3. HIERARCHY OF FAULT CLASSES

This section develops a hierarchy of fault classes based on the conditions under which a particular type of faults are detected. It is then shown that this hierarchy can be used to explain the empirical results for fault based testing described by Foster [1], Weyuker et al. [2], and Vouk et al. [8].

3.1 Fault Classes

We first determine the detection conditions for the various fault classes under different assumptions. Let \( S \) be a specification in disjunctive normal form:
\[
S = x_1 \land x_2 \land \ldots \lor x_{21} \land x_{22} \ldots \lor x_n \land x_{n2} \ldots \tag{1}
\]

In general, the \(x_i\) variables may not be distinct. For example, we could have \(a \land b \lor a \land c\). Then the conditions under which, for example, a variable negation fault for variable \(a\) will be detected are \(S \oplus S_i^2\). The conditions for detecting variable negation faults (\(S_{VNF}\)), variable reference faults (\(S_{VRF}\)) and expression negation faults (\(S_{ENF}\)) are given below. (Note that the only expressions in ENF are clauses.)

\[
S_{VNF} = S \oplus S_{-xi_i}^a
\]

\[
S_{VRF} = S \oplus S_{-xi_i}^a, \text{ where } zk_i \text{ is the variable substituted for } xi_i, zk_i \neq xi_i
\]

\[
S_{ENF} = S \oplus S_{X_i}^a,
\text{ where } X_i \text{ is the conjuction } xi_1 \land xi_2 \land \ldots \land xi_m. \text{ It can readily be shown that } S_{VRF} \Rightarrow S_{VNF} \Rightarrow S_{ENF} \text{ under very minimal restrictions.}
\]

**Theorem 3.1.** If the variable replaced in \(S_{VRF}\) is the same variable negated in \(S_{VNF}\) then \(S_{VRF} \Rightarrow S_{VNF}\).

Proof: For readability, the variables in formula(1) will be abbreviated as \(a_1\) for \(x_1\), \(a_2\) for \(x_{21}\), \ldots, \(b_1\) for \(x_{21}\), etc.

\[
P = a_1 \land a_2 \land \ldots \land a_k \lor b_1 \land b_2 \land \ldots \land b_m \lor \ldots \lor z_1 \land z_2 \land \ldots \land z_m
\]

**Case 1:** Substituting a variable from a different clause. We want to establish that detection conditions for an arbitrary variable reference fault, \(a_1 := b_1\), in this predicate imply the detection conditions for the variable negation fault \(a_1 := a_1\), i.e., \(dP_{b_1}^a \Rightarrow dP_{a_1}^a\).

The left hand side, \(dP_{b_1}^a\), reduces to:

\[
[a_1 \land a_2 \land \ldots \land a_k \oplus b_1 \land a_2 \land \ldots \land a_k] \land (b_1 \lor b_2 \lor \ldots \lor b_m) \land \ldots \land (z_1 \lor z_2 \lor \ldots \lor z_m)
\]

The right hand side \(dP_{a_1}^a\), reduces to:

\[
a_2 \land \ldots \land a_k \land (b_1 \lor b_2 \lor \ldots \lor b_m) \land \ldots \land (z_1 \lor z_2 \lor \ldots \lor z_m)
\]

Note that \([a_1 \land a_2 \land \ldots \land a_k \oplus b_1 \land a_2 \land \ldots \land a_k] \land (b_1 \lor b_2 \lor \ldots \lor b_m) \land \ldots \land (z_1 \lor z_2 \lor \ldots \lor z_m) = [a_1 \oplus b_1] \land a_2 \land \ldots \land a_k \land (b_1 \lor b_2 \lor \ldots \lor b_m) \land \ldots \land (z_1 \lor z_2 \lor \ldots \lor z_m)
\]

and that \([a_1 \oplus b_1] \land a_2 \land \ldots \land a_k \Rightarrow a_2 \land \ldots \land a_k\).

**Case 2:** Substituting a variable from the same clause. We want to establish that detection conditions for an arbitrary variable reference fault, \(a_1 := a_2\), in this
predicate imply the detection conditions for the variable negation fault \( a_1 := a_1 \), i.e.: \( dP_{a_1}^1 \Rightarrow dP_{a_1}^2 \).

The left hand side, \( dP_{a_1}^2 \), reduces to:
\[
[a_1 \land a_2 \land \ldots \land a_k \land \neg b_1 \lor b_2 \lor \ldots \lor b_m] \land \ldots \land
(z_1 \lor z_2 \lor \ldots \lor z_n)
\]
which is
\[
a_1 \land a_2 \land \ldots \land a_k \land (b_1 \lor b_2 \lor \ldots \lor b_m) \land \ldots \land (z_1 \lor z_2 \lor \ldots \lor z_n)
\]
which clearly implies the right hand side. Q.E.D.

**Corollary 3.2.** Any test that detects a variable reference fault for a variable \( x \) in a predicate will also detect a variable negation fault for the same variable.

Now consider the relationship between variable negation faults and expression negation faults.

**Theorem 3.3.** If expressions containing the variable negated in \( S_{VNF} \) are negated in \( S_{ENF} \) then \( S_{VNF} \Rightarrow S_{ENF} \), where the expressions are clauses in the correct specification \( S \).

Proof: We want to establish that detection conditions for a variable negation fault in this predicate for an arbitrary variable \( a_1 \) imply the detection conditions for an expression negation fault for expressions including \( a_1 \).

\[
dP_{a_1}^2 \Rightarrow dP_{\neg(E_1 \ldots E_k)}^1
\]
where \( E_1, E_2, \ldots E_k \) are all expressions containing \( a_1 \).

We assume that variable \( a_1 \) may occur in more than one clause. That is, some of \( b_j, d_k, \) etc., may be the same variable as \( a_1 \). Let the formula be rearranged so that all clauses containing \( a_1 \) occur first, followed by clauses not containing \( a_1 \).

Then abbreviate clauses containing \( a_1 \) by \( E_1, E_2, \ldots \), and clauses not containing \( a_1 \) by \( R_1, R_2, \ldots \).

The detection conditions for the variable negation fault are then given by:
\[
E_1[a_1 := a_1] \lor E_2[a_1 := a_1] \lor \ldots \lor E_k[a_1 := a_1] \land R_1 \land \ldots \land R_m
\]
\[
\equiv E_1 \lor E_2 \lor \ldots \lor E_k \land R_1 \land \ldots \land R_m
\]
which is
\[
(E_1 \lor E_2 \lor \ldots \lor E_k \land R_1 \land \ldots \land R_m)
\]

The detection conditions for the expression negation fault are then given by:
\[
P_{ENF} \equiv E_1 \lor E_2 \lor \ldots \lor E_k \lor R_1 \lor \ldots \lor R_m \equiv \neg(E_1 \lor E_2 \lor \ldots \lor E_k) \lor R_1 \lor \ldots \lor R_m
\]

Since \( P_{ENF} \) reduces to simply \( \neg(R_1 \lor \ldots \lor R_m) \), clearly \( dP_{a_1}^2 \Rightarrow dP_{\neg(E_1 \ldots E_k)}^1 \).

Similarly, if the variable is negated in some, but not all, of the \( E_k \), the result also holds. If the clauses in which the variable is negated are \( E_1 \ldots E_j \), then (placing these first)
Fault Classes and Error Detection Capability of Specification Based Testing

\[ P_{VNF} = (E_1 \lor E_2 \lor \ldots \lor E_j \oplus E_1[a_1 := a_1] \lor E_2[a_1 := a_1] \lor E_2[a_1 := a_1]) \land \neg(E_{j+1} \lor E_k \lor R_1 \ldots R_m) \]

and

\[ P_{ENF} = \neg(E_{j+1} \lor E_k \lor R_1 \ldots R_m) \text{ Q.E.D.} \]

**Corollary 3.4.** Any test that detects a variable negation fault for a variable \( x \) in a predicate will also detect an expression negation fault for the expression in which the variable occurs.

### 3.2 Examples

This section provides two examples. The first one is simple enough to make the hierarchy of fault detection conditions obvious. The second is a realistic example, taken from the FAA Traffic Collision Avoidance System software specification, as reported in [2].

#### 3.2.1 Example 1.

Consider the expression from Section 2: \( p \land q \lor r \). A variable reference fault where \( q \) is replaced by \( r \) can be detected with conditions shown below:

\[ S_{VRF} : dS^p_{q} = (p \land q \lor r) \oplus (p \land q \lor r) \]

\[ = p \land q \land r \]

A variable negation fault where \( q \) is replaced by \( q \) is detected with these conditions:

\[ S_{VNF} : dS^q_{q} = (p \land q \lor r) \oplus (p \land q \lor r) \]

\[ = p \land q \]

An expression negation fault where \( (p \land q) \) is replaced by \( \neg(p \land q) \) is detected by \( r \):

\[ S_{ENF} : dS^{(p \land q)}_{\neg(p \land q)} = (p \land q \lor r) \oplus (\neg(p \land q) \lor r) \]

\[ = r \]

Clearly, \( S_{VRF} \Rightarrow S_{VNF} \Rightarrow S_{ENF} \), i.e., the following relationship holds:

\[ p \land q \land r \Rightarrow p \land q \Rightarrow r \]

#### 3.2.2 Example 2.

For a realistic example, consider the following formula from [2]:

\[ P : a \land c \land (d \lor e) \land h \lor a \land (d \lor e) \land h \lor b \land (e \lor f) \]

A variable reference fault where \( e \) is replaced by \( c \) is detected by the conditions \( P_{VRF} \): \n
\[ P_{VRF} = a \land ((c \lor h) \land (d \lor e)) \lor b \land (e \lor f) \oplus a \land (c \land h \land (d \lor c) \land h) \lor b \land (c \lor f) \]

A variable negation fault where \( e \) is replaced by \( \neg e \) is detected by the conditions \( P_{VNF} \):

\[ P_{VNF} = a \land ((c \lor h) \land (d \lor e)) \lor b \land (e \lor f) \oplus a \land ((c \lor h) \land (d \lor e)) \lor b \land (e \lor f) \]

An expression negation fault where \( (d \lor e) \) is replaced by its negation is detected by the conditions \( P_{ENF} \):
\[ P_{\text{ENF}} = a \land ((c \lor \bar{h}) \land (d \lor e)) \lor b \land (e \lor f) \equiv a \land ((c \lor \bar{h}) \land \neg (d \lor e)) \lor b \land \neg (e \lor f) \]

It can then be shown that: \( dP_z \Rightarrow dP_s \Rightarrow dP^{(d \lor e)}_{- (d \lor e)} \)

4. ANALYSIS OF EMPIRICAL DATA

The empirical data presented in [2] show that tests detected 100% of ENF and a slightly smaller percentage of the other faults. Testing detected fewer variable negation faults than expression faults, and fewer variable reference faults than variable negation faults. For variable reference faults (VRF), variable negation faults (VNF), and expression negation faults (ENF), the relationship is \( VRF < VNF < ENF \). Including the associative shift faults (ASF) and operator reference faults (ORF), the relationship is \( ASF < VRF < VNF < ORF < ENF \).

Why does this relationship hold? We will consider only the conditions for VNF, VRF, and ENF as the conditions for ASF and ORF depend on the particular operators or association faults chosen by the tester. Note that the conditions under which a particular fault will cause a failure are defined by the boolean difference of the specification with respect to the particular fault. Where \( S^p_x \) defines the faulty substitution of an expression \( e \) for term \( x \), the difference \( dS^p_x = S \oplus S^p_x \) defines the conditions under which the fault will cause a failure. Weyuker et al.'s meaningful impact testing draws tests from the conditions defined by \( dS^p_x \), i.e., the detection conditions for variable negation faults. As shown in Section 3, \( dS_{\text{VRF}} \Rightarrow dS_{\text{VNF}} \Rightarrow dS_{\text{ENF}} \). That is, conditions for \( dS_{\text{VRF}} \) (the conditions under which a VRF will cause a failure) are the conjunction of \( dS_{\text{VNF}} \) and additional conditions. So every condition that tests for a VRF also tests for a VNF. Likewise, every test for a VNF is also a test for an ENF. In terms of test conditions, the relationship between the fault classes is: \( dS_{\text{VRF}} \subseteq dS_{\text{VNF}} \subseteq dS_{\text{ENF}} \).

Consider Example 1 from the previous section. The conditions to detect the variable reference faults where \( q \) is substituted for \( p \) in specification \( S \) are \( dS^p_x = p \land q \land \tau \). A variable negation fault for the variable \( p \) is detected by \( dS^p_x = q \land \tau \). Clearly, \( dS^p_x \Rightarrow dS^p_x \). So any test set that detects variable reference faults for \( p \) will also detect variable negation faults for \( p \). Because \( d\text{VRF} \Rightarrow d\text{VNF} \Rightarrow d\text{ENF} \), the VRF fault class can be considered "stronger" than the VNF fault class, which is in turn stronger than the ENF fault class.

Recall that empirical results showed that variable reference faults were detected less successfully than variable negation faults, which in turn were detected less successfully than expression negation faults. A VNF for \( p \) can be detected by either \( p \land q \land \tau \) or by \( p \land q \land \tau \). Depending on which is chosen, the test vector may or may not also detect a VRF for \( p \). On the other hand, the VNF test set for \( p \) will always detect an ENF for an expression containing \( p \). The empirical results are thus consistent with the hierarchy developed in Section 3.

The results described in this paper suggest that fault-based testing can be made more efficient by designing test generation algorithms to target the strongest fault class. The relationship between the VRF, VNF, and ENF fault classes implies immediately that not more than \( n(n-1) \) tests are required to detect all faults in any of these classes, for an expression with \( n \) distinct variables, or \( m(n-1) \) tests for \( m \) occurrences of \( n \) variables. This is because each variable can be replaced by any of the other variables in a variable reference fault. In practice the number of tests
needed is much less, because of overlap between detection conditions. As the next section shows, however, tests computed for another fault model, missing condition faults, can detect faults in the other classes at a cost that is linear in the number of conditions.

5. MISSING CONDITION FAULTS

While the results presented in previous sections are interesting from a theoretical standpoint, a natural question to ask is whether the various fault types described in Section 1 are realistic models of faults that occur in software. In this section we consider a type of fault that does occur in software, and its relationship to other fault types. One of the most common implementation errors is the failure to validate input data, or check preconditions. We will refer to this type of fault as a missing condition fault, i.e., a fault where the specification contains one or more conditions not implemented by the programmer. Missing condition faults can be regarded as a special case of variable reference fault. For example, suppose the correct predicate is: 

\[ P = A \land B \land C \lor D \land E \land F \land \ldots \]

A missing condition fault in which \( A \) is not implemented is equivalent to the variable reference fault in which \( A \) is replaced by (for example) \( B \), i.e.,

\[ P[A := B] = B \land C \lor D \land E \land F \lor \ldots \]

With a singular condition, e.g. \( A \) in

\[ P = A \lor D \land E \land F \lor \ldots , \]

the missing condition fault is equivalent to \( A \) being replaced by one of the conjunct expressions in the DNF formula, e.g.

\[ P[A := D \land E \land F] = D \land E \land F \lor \ldots \]

Variable reference faults can now be divided into those in which the variable substitution results in a missing condition fault and others in which one condition is replaced by another, e.g., where \( A \) is replaced by \( E \) in \( P = A \land B \land C \lor D \land E \land F \lor \ldots \). The second type of fault could be described as an incorrect condition.
fault, since the boolean variables typically represent some relation or condition in a specification. The hierarchy can thus be extended as shown in Figure 1.

There is some evidence from empirical investigations of software faults that missing condition faults are extremely common. Marick [20] shows that approximately half of the faults posted on usenet bug reports are faults of omission, while only 23% were simple faults. Although not all faults of omission are necessarily missing condition faults, tests based on this fault model do appear to be effective in finding errors. Experiments have been conducted in generating tests from specifications by using missing condition faults. Preliminary results show that the tests generated provide coverage that is as good or better than typically produced by manually created tests, as measured by statement coverage and branch coverage metrics [14].

6. MCDC COVERAGE VIA BOOLEAN DIFFERENCES
Chilenski and Miller [21] analyze Modified Conditions/Decision Coverage (MCDC), which DO-178B [9] defines as follows:

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 independently affect the decision's outcome by varying just that condition while holding fixed all other possible conditions.

Chilenski and Miller present a 'Pairs Table' approach to identifying MCDC adequate test sets. In the Pairs Table approach, a truth table is defined for the boolean decision of interest. Rows in the truth table are numbered. An additional column is added for each condition. The entry in this column for a particular row is the row or rows for which i) the condition of interest is the only variable that changes, and ii) the boolean decision changes truth value as well. Often, many of the entries in a Pairs Table are blank. The possibility of multiple entries arises when short-circuit operations are considered.

MCDC coverage is obtained by selecting enough rows in the truth table such that each condition column has a 'Pair' selected. That is, for each column, the chosen rows must include a pair of rows such that when the relevant condition changes, the value of the boolean decision changes as well. Chilenski and Miller state that for a boolean expression with n conditions, a minimum of n + 1 tests 'can usually be achieved'.

Explicitly constructing truth tables has significant drawbacks. This section presents an alternative approach that uses boolean differences to develop a specification for the circumstances under which MCDC is achieved.

Consider a particular condition z in some boolean decision P. Then the boolean difference of P with respect to z, \(dP^z\), \(P \oplus P^z\) gives the conditions under which P depends on the value of z. Thus by choosing an assignment of truth values such that \(dP^z\) is satisfied, and then choosing z to be true and then false, two tests are generated that satisfy MCDC with respect to z. Repeating the procedure for each condition yields a total of 2n tests. Careful selection of these tests may reduce the total number of tests to n + 1.
Consider an example, \( A \land (B \lor C) \), developed with both the Pairs Table approach and via boolean difference. The Pairs Table approach in [21] begins with constructing a truth table for \( A \land (B \lor C) \), for all possible values of the variables \( A, B, \) and \( C \). The columns labeled \( A, B, \) and \( C \) show which test cases (first column) can be used to show the independence of the condition (second column). For example, the independence of \( A \) can be shown by pairing test case 1 with test case 5.

<table>
<thead>
<tr>
<th>Case</th>
<th>ABC</th>
<th>Result</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>111</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>110</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>101</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>011</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>010</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>001</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>000</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Pairs Table for \( A \land (B \lor C) \)

The boolean difference approach is as follows. First, the boolean differences with respect to \( A, B, \) and \( C \) are calculated:

1. \( dP_A^A = A \land (B \lor C) \oplus \overline{A} \land (B \lor C) = B \lor C \)
2. \( dP_B^B = A \land (B \lor C) \oplus A \land \overline{(B \lor C)} = A \land \overline{C} \)
3. \( dP_C^C = A \land (B \lor C) \oplus A \land (B \lor C) = A \land B \)

Test sets are generated as follows:

1. \( T_A \): From \( dP_A^A \), select \( B \lor C \) true (three possibilities) and \( A \) both true and false, yielding three choices for tests (the notation indicates the assignments of truth values to \( A, B, \) and \( C, \) respectively): (a) \( \{111,011\} \), (b) \( \{110,010\} \), and (c) \( \{101,001\} \).
2. \( T_B \): From \( dP_B^B \), select \( A \land \overline{C} \) true and \( B \) both true and false, yielding \( \{110,100\} \).
3. \( T_C \): From \( dP_C^C \), select \( A \land B \) true and \( C \) both true and false, yielding \( \{101,100\} \).

Next, combine the test sets generated above. There are three possible test cases: (1) \( \{111,110,101,100,011\} \) (i.e., \( T_A(a), T_B, T_C \)); (2) \( \{110,101,100,010\} \) \((T_A(b), T_B, T_C)\); (3) \( \{110,101,100,001\} \) \((T_A(c), T_B, T_C)\). The second and third possibilities are more desirable since they use the minimum number \((n + 1)\) of tests.

7. COMPARING TEST METHODS

The results can be used to compare the theoretical effectiveness of published test methods. One published test generation method, Ofluit and Liu's [7], uses the following procedure, where predicates are assumed to be in disjunctive normal form:

—At the disjunctive level, where predicates are of the form \( A \lor B \lor C \lor \ldots \), generate test values by holding all disjuncts but one false, then vary each one to be true in turn.
At the conjunctive level, where predicates are of the form $A \land B \land C \land \ldots$, first find values that cause each clause to be true, then generate additional tests by holding all conjuncts but one true and vary each one to be false in turn.

As it turns out, this procedure is equivalent to generating missing condition faults for each of the variables in the predicates being tested. That is, for each variable $x_i$, the conditions to detect a missing condition fault for $x_i$ are given by:

$$f(x_1, \ldots, x_i, \ldots, x_n) \oplus f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n).$$

This results in an expression of the form

$$x_i \land x_j \land x_k \land \ldots \land \neg(\lor_{x_i \neq x_m} x_m),$$

where $x_j, x_k$ are other variables in the conjunct containing $x_i$.

For example, suppose the predicate is $a \land b \land c \lor d \land e \land f$. Computing the detection conditions for missing condition faults for each variable gives the following set of expressions:

$$a \land b \land c \land \neg(d \land e \land f)$$
$$b \land a \land c \land \neg(d \land e \land f)$$
$$c \land a \land b \land \neg(d \land e \land f)$$
$$d \land e \land f \land \neg(a \land b \land c)$$
$$e \land d \land f \land \neg(a \land b \land c)$$
$$f \land d \land e \land \neg(a \land b \land c)$$

Computing such an expression for each variable is thus equivalent to using the test generation method of [7]. By comparison, the methods described by Foster and by Weyuker et al. use algorithms that are equivalent to computing boolean differences for variable negation faults. Since tests that detect missing condition faults will also detect variable negation faults, Offutt and Liu's method should be at least as efficient than these, and may be more efficient for some faults.

8. CONCLUSIONS

This paper has developed a hierarchy of fault models used in specification based software testing. Tests that detect missing condition faults will detect variable negation faults, and tests that detect variable negation faults will detect expression negation faults. These results suggest that test generation methods that focus on detecting missing condition faults will also detect a variety of other fault types.

Experimental results presented by various authors are consistent with the hierarchy. Experiments show that expression negation faults are detected more readily than variable negation faults, which in turn are detected more readily than variable reference faults, of which missing condition faults are an important subclass. This is to be expected because a test for a variable reference fault will also detect the other fault types, while the converse is not necessarily true. Experimental results are thus in alignment with results presented in the paper, and suggest that specification based testing should give priority to the detection of missing condition faults.

9. ACKNOWLEDGEMENTS

I am grateful to Paul Ammann for many suggestions and for contributing work on MCDC testing. Paul Black did a careful analysis of the results. Mary Zelkowitz
provided helpful comments on demonstrating the practical benefits of the results. ACM reviewers suggested many improvements.

REFERENCES