NAT'L INST. OF STAND & TECH R.I.C.

A11105 354739

NIST PUBLICATIONS

NISTIR 6140

Fault Classes and Error Detection in Specification Based Testing

D. Richard Kuhn

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899-0001

QC
100
.U56
NO.6140
1998

NIST

NISTIR 6140

Fault Classes and Error Detection in Specification Based Testing

D. Richard Kuhn

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899-0001

February 1998

U.S. DEPARTMENT OF COMMERCE
William M. Daley, Secretary

TECHNOLOGY ADMINISTRATION
Gary R. Bachula, Acting Under Secretary
for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Raymond G. Kammer, Director

Fault Classes and Error Detection in Specification Based Testing

D. Richard Kuhn
National Institute of Standards and Technology
Gaithersburg, Maryland 20899
kuhn@nist.gov

February 24, 1998

Abstract

Specification based testing relies 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 demonstrating their effectiveness of the test generation methods [1] [2] [7] [8].

This paper examines the conditions under which a particular fault class will cause an error for a given predicate. These conditions must be covered by a test set for the test set to guarantee detection of the particular fault class. By deriving the general conditions under which various fault classes cause an error, we show that there is a coverage hierarchy to fault classes. The fault hierarchy is then used to explain experimental results on fault based testing. This work is significant because it provides a method for comparing the effectiveness of test sets designed for various fault models.

General Terms: THEORY, VERIFICATION, TESTING

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.

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] [9] [10]. 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 [11]. In circuit testing, typical manufacturing flaws are hypothesized, then test sets are derived to detect these flaws. For example, a common flaw is a stuck-at-zero fault, in which a logic gate always produces a zero value. This flaw is modeled by replacing the variable that represents the gate in the circuit specification with a zero value. Using the correct specification and the specification with the flaw inserted, tests sets can be constructed automatically to detect the flaw.

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 constucted to detect them. The fault classes defined in [2], [10], and [12] are the following:

- Variable Reference Fault - a boolean variable x is replaced by another variable y, $x \neq y$.
- Variable Negation Fault - a boolean variable x is replaced by $\bar{x}$.
- Expression Negation Fault - a boolean expression p is replaced by $\bar{p}$.
- Associative Shift Fault - a boolean expression is replaced by one in which the association between variables is incorrect, e.g. $x \wedge (y \vee z)$ replaced by $x \wedge y \vee z$.
- Operator Reference Fault - a boolean operator is replaced by another, e.g., $x \wedge y$ replaced by $x \vee y$.

Additional types of faults are defined in [8] :

- Incorrect relational operators - a relational operator (e.g., $>$, $<$) is replaced by a different relational operator.
- Incorrect parentheses;

- Incorrect arithmetic expression;
- Extra binary operators;
- Missing binary operator.

Experimental results have demonstrated the effectiveness of the test generation methods [1] [2] [7] [8].

In this paper, the conditions under which a particular fault class will cause an error 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 an error, 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 explains experimental results on fault based testing. The existence of this hierarchy also helps to explain the effectiveness of fault based testing techniques for detecting a broad range of fault conditions.

The results presented in this paper also have some implications for the *coupling effect* hypothesis [13],[14], one of the principles of mutation testing. The coupling effect hypothesis states that tests which detect simple types of faults will also detect more complex faults. Some empirical evidence exists to show that the coupling effect hypothesis does in fact hold, providing some assurance that fault-based testing is an effective strategy [14]. The existence of a fault hierarchy implies that a limited class of simple tests will also detect faults of other types.

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 an error 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 (see Appendix) 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 $P \oplus P^v_{\bar{v}}$, where P^x_e is predicate P with all free occurrences of variable x replaced by expression e. (P^x_e 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 an error, 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 $S = p \wedge \bar{q} \vee r$, we can compute the conditions under which a variable negation fault for variable q will cause an error, by computing the boolean difference:

$$\begin{aligned} dS^q_{\bar{q}} &= (p \wedge \neg q \vee r) \oplus (p \wedge q \vee r) \\ &= p \wedge \bar{r} \end{aligned}$$

Table 1 shows the correct expression S, the incorrect implementation I and the value of S and I for possible values of p, q and r. Note that only where $p \wedge \bar{r} = 1$ (marked with '*') does the fault make a difference between the value of S and I.

pqr	$p \wedge \bar{q} \vee r$	$p \wedge q \vee r$
000	0	0
001	1	1
010	0	0
011	1	1
100 *	1	0
101	1	1
110 *	0	1
111	1	1

Table 1.

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:

$$\begin{aligned} S \;=\; & x1_1 \wedge x1_2 \wedge \ldots \\ & \vee x2_1 \wedge x2_2 \ldots \\ & \vee xn_1 \wedge xn_2 \ldots \end{aligned} \tag{1}$$

In general, the xi_j variables may not be distinct. For example, we could have $a \wedge b \vee a \wedge c$.

Then the conditions under which, for example, a variable negation fault for variable a will be detected are $S \oplus S^a_{\bar{a}}$. The conditions for detecting variable negation faults (S_{VNF}), variable reference faults (S_{VRF}) and expression negation faults (S_{ENF}) are given below:

$S_{VNF} = S \oplus S^{xi_j}_{\neg xi_j}$

$S_{VRF} = S \oplus S^{xi_j}_{xk_l}$, where xk_l is the variable substituted for xi_j

$S_{ENF} = S \oplus S^{X_i}_{\neg X_i}$,

where X_i is the conjuction $xi_1 \wedge xi_2 \wedge \ldots \wedge xi_n$.

It can readily be shown that $S_{VRF} \Rightarrow S_{VNF} \Rightarrow S_{ENF}$ under very minimal restrictions. Figure 1 shows the relationship between detection conditions for these fault classes.

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

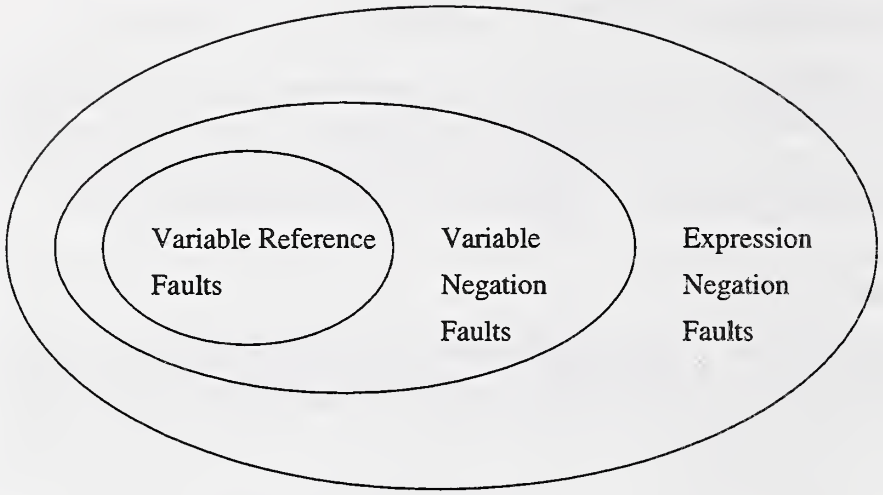

Figure 1: Fault Detection Condition Relationships

Proof:

For readability, the variables in formula(1) will be abbreviated as a_1 for $x1_1$, a_2 for $x1_2$, ...b_1 for $x2_1$, etc.

$$P = a_1 \wedge a_2 \wedge ... \wedge a_k \vee b_1 \wedge b_2 \wedge ... \wedge b_m \vee ... \vee z_1 \wedge z_2 \wedge ... \wedge z_n$$

We want to establish that detection conditions for an arbitrary variable reference fault, $a_1 := b_2$, in this predicate imply the detection conditions for the variable negation fault $a_1 := \bar{a}_1$, i.e.:

$$dP_{b_2}^{a_1} \Rightarrow dP_{\bar{a}_1}^{a_1}$$

The left hand side, $dP_{b_2}^{a_1}$ reduces to:

$$[a_1 \wedge a_2 \wedge ... \wedge a_k \oplus b_1 \wedge a_2 \wedge ... \wedge a_k]$$
$$\wedge (\bar{b}_1 \vee \bar{b}_2 \vee ... \vee \bar{b}_m)$$
$$\wedge ... \wedge (\bar{z}_1 \vee \bar{z}_2 \vee ... \vee \bar{z}_n)$$

The right hand side $dP_{\neg a_1}^{a_1}$ reduces to:
$a_2 \wedge ... \wedge a_k \wedge (\bar{b}_1 \vee \bar{b}_2 \vee ... \vee \bar{b}_m) \wedge ... \wedge (\bar{z}_1 \vee \bar{z}_2 \vee ... \vee \bar{z}_n)$

Note that

$$[a_1 \wedge a_2 \wedge ... \wedge a_k \oplus b_1 \wedge a_2 \wedge ... \wedge a_k] \wedge (\bar{b}_1 \vee \bar{b}_2 \vee ... \vee \bar{b}_m) \wedge ... \wedge (\bar{z}_1 \vee \bar{z}_2 \vee ... \vee \bar{z}_n)$$

$$= \quad [a_1 \oplus b_2] \wedge a_2 \wedge ... \wedge a_k \wedge (\bar{b}_1 \vee \bar{b}_2 \vee ... \vee \bar{b}_m) \wedge ... \wedge (\bar{z}_1 \vee \bar{z}_2 \vee ... \vee \bar{z}_n),$$

and that $[a_1 \oplus b_2] \wedge a_2 \wedge ... \wedge a_k \Rightarrow \wedge a_2 \wedge ... \wedge a_k$, which establishes the result. 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 all expressions containing the variable negated in S_{VNF} are negated in S_{ENF} then $S_{VNF} \Rightarrow S_{ENF}$*

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.

$$dP^{a_1}_{\bar{a}_1} \Rightarrow dP^{(E_1 ... E_k)}_{\neg(E_1 ... E_k)}$$

where $E_1, E_2, ... E_k$ 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 $E_1, E_2, ...$, and clauses not containing a_1 by $R_1, R_2, ...$

The detection conditions for the variable negation fault are then given by:

$$\begin{aligned} E_1[a_1 := \bar{a}_1] \vee E_2[a_1 := \bar{a}_1] \vee ... \vee E_k[a_1 := \bar{a}_1] \vee R_1 \vee ... \vee R_m \\ \oplus E_1 \vee E_2 \vee ... \vee E_k \vee R_1 \vee ... \vee R_m \end{aligned}$$

which is

$$(E_1 \vee E_2 \vee \dots \vee E_k \oplus E_1[a_1 := \bar{a}_1] \vee E_2[a_1 := \bar{a}_1] \vee \dots \vee E_k[a_1 := \bar{a}_1]) \\ \wedge \neg(R_1 \vee \dots \vee R_m$$

The detection conditions for the expression negation fault are then given by:

$$P_{ENF} \equiv E_1 \vee E_2 \vee \dots \vee E_k \vee R_1 \vee \dots \vee R_m \oplus \\ \neg(E_1 \vee E_2 \vee \dots \vee E_k) \vee R_1 \vee \dots \vee R_m$$

Since P_{ENF} reduces to simply $\neg(R_1 \vee \dots \vee R_m)$, clearly $dP^{a_1}_{\bar{a}_1} \Rightarrow dP^{(E_1 \dots E_k)}_{\neg(E_1 \dots E_k)}$ 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: $p \wedge \bar{q} \vee r$. A variable reference fault where q is replaced by r can be detected with conditions shown below:

$$\begin{aligned} S_{VRF} : dS^q_r &= (p \wedge \bar{q} \vee r) \oplus (p \wedge \bar{r} \vee r) \\ &= p \wedge q \wedge \bar{r} \end{aligned}$$

A variable negation fault where q is replaced by $\bar{q}$ is dected with these conditions:

$$\begin{aligned} S_{VNF} : dS^q_{\bar{q}} &= (p \wedge \neg q \vee r) \oplus (p \wedge q \vee r) \\ &= p \wedge \bar{r} \end{aligned}$$

An expression negation fault where $(p \wedge \bar{q})$ is replaced with $\neg(p \wedge \bar{q})$ is detected by $\bar{r}$:

$$\begin{aligned} S_{ENF} : dS^{(p\wedge\bar{q})}_{\neg(p\wedge\bar{q})} &= (p \wedge \neg q \vee r) \oplus (\neg(p \wedge q) \vee r) \\ &= \bar{r} \end{aligned}$$

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

$$p \wedge q \wedge \bar{r} \Rightarrow p \wedge \bar{r} \Rightarrow \bar{r}$$

3.2.2 Example 2

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

$$P : a \wedge c \wedge (d \vee e) \wedge h \vee a \wedge (d \vee e) \wedge \bar{h} \vee b \wedge (e \vee f)$$

A variable reference fault where e is replaced by c is detected by the conditions P_{VRF}:

$$\begin{aligned} P_{VRF} = {} & a \wedge ((c \vee \bar{h}) \wedge (d \vee e)) \vee b \wedge (e \vee f) \\ & \oplus a \wedge (c \wedge h \vee (d \vee c) \wedge \bar{h}) \vee b \wedge (c \vee f) \end{aligned}$$

A variable negation fault where e is replaced by $\bar{e}$ is detected by the conditions P_{VNF}:

$$\begin{aligned} P_{VNF} = {} & a \wedge ((c \vee \bar{h}) \wedge (d \vee e)) \vee b \wedge (e \vee f) \\ & \oplus a \wedge ((c \vee \bar{h}) \wedge (d \vee \bar{e})) \vee b \wedge (\bar{e} \vee f) \end{aligned}$$

$$\begin{aligned} P_{ENF} = {} & a \wedge ((c \vee \bar{h}) \wedge (d \vee e)) \vee b \wedge (e \vee f) \\ & \oplus a \wedge ((c \vee \bar{h}) \wedge \neg(d \vee e)) \vee b \wedge \neg(e \vee f) \end{aligned}$$

It can then be shown that:

$$dP^{e}_{c} \Rightarrow dP^{e}_{\bar{e}} \Rightarrow dP^{(d\vee e)}_{\neg(d\vee e)}$$

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 less well defined associative shift faults (ASF) and operator reference faults (ORF), the relationship is $ASF < VRF < VNF \leq ORF < ENF$.

Why does this relationship hold? We will consider only the conditions for VNF, VRF, and ENF as the conditions for ASF and ORF are somewhat arbitrary and 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_e^x defines the faulty substitution of an expression e for term x, the difference $dS_e^x = S \oplus S_e^x$ 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_e^x As shown in Section 3, $dS_{VRF} \Rightarrow dS_{VNF} \Rightarrow dS_{ENF}$. That is, conditions for dS_{VRF} (the conditions under which a VRF will cause a failure) are the conjunction of dS_{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: $dS_{VRF} \subseteq dS_{VNF} \subseteq dS_{ENF}$, as shown in Figure 1.

Consider Example 1 from the previous section. Detection conditions for variable reference faults and for variable negation faults are shown below.

Variable reference fault detection conditions for p:

$dS_q^p = p \wedge \bar{q} \wedge \bar{r}$
$dS_r^p = p \wedge \bar{q} \wedge \bar{r}$

Variable reference fault detection conditions for q:

$dS_p^q = p \wedge \bar{q} \wedge \bar{r}$
$dS_r^q = p \wedge q \wedge \bar{r}$

Variable reference fault detection conditions for r:

$dS^r_p = r \wedge \bar{p} \vee p \wedge q \wedge \bar{r}$
$dS^r_q = q \wedge \bar{r} \vee p \wedge q \wedge \bar{r} \vee q \wedge \bar{p} \wedge \bar{r} \vee r \wedge \bar{p} \wedge \bar{q}$

Variable negation fault detection conditions for p:
$dS^p_{\bar{p}} = \bar{q} \wedge \bar{r}$

Variable negation fault detection conditions for q:
$dS^q_{\bar{q}} = p \wedge \bar{r}$

Variable negation fault detection conditions for r:
$dS^r_{\bar{r}} = \bar{p} \vee q$

For example, the conditions to detect the variable reference faults where q is substituted for p in specification S are $dS^p_q = p \wedge \bar{q} \wedge \bar{r}$. A variable negation fault for the variable p is detected by $dS^p_{\bar{p}} = \bar{q} \wedge \bar{r}$. Clearly, $dS^p_q \Rightarrow dS^p_{\bar{p}}$. So any test set that detects variable reference faults for p will also detect variable negation faults for p. Because $dVRF \Rightarrow dVNF \Rightarrow dENF$, 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 $p \wedge \bar{q} \wedge \bar{r}$ or by $\bar{p} \wedge \bar{q} \wedge \bar{r}$. 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 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.

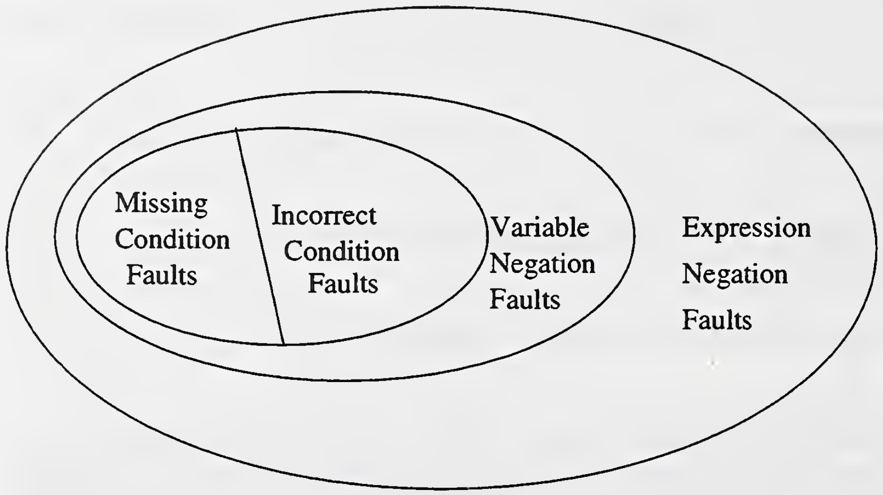

Figure 2: Fault Detection Condition Relationships

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.* Missing condition faults can be regarded as a special case of variable reference fault. For example, consider the predecate $P = A \wedge B \wedge C \vee D \wedge E \wedge F \vee ....$

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.,

$P[A := B] = B \wedge C \vee D \wedge E \wedge F \vee ....$

With a singular condition, e.g. A in

$P = A \vee D \wedge E \wedge F \vee ....,$

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

$P[A := D \wedge E \wedge F] = D \wedge E \wedge F \vee ....$

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 $P = A \wedge B \wedge C \vee D \wedge E \wedge F \vee$..... 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 2. Of the two types of variable reference faults, incorrect condition faults seem to be unlikely in practice.

There is some evidence from empirical investigations of software faults that missing condition faults are extremely common. Marick [20], in an investigation of the competent programmer hypothesis, shows that approximately half of the faults posted on usenet bug reports are faults of ommission, what we have referred to as missing condition faults, while only 23% were simple faults. In circuit testing, the single-stuck-fault is often referred to as the *standard* fault model, because it is one of the most common fault types, and because tests for single stuck faults can detect a number of other fault types. The missing condition fault may serve the same purpose for specification based software testing.

6 MCDC Coverage Via Boolean Differences

Chilenski and Miller [21] analyze Modified Conditions/Decision Coverage (MCDC), which DO-178B [22] 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. An example is given in the next section. 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 x in some boolean decision P. Then the boolean difference of P with respect to x,

$$dP/dx \;=\; P \oplus P^{x}_{\bar{x}}$$

gives the conditions under which P depends on the value of x. Thus by choosing an assignment of truth values such that dP/dx is satisfied, and then choosing x to be true and then false, two tests are generated that satisfy MCDC with respect to x. 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 $n+1$.

It is worth noting that the boolean difference approach can also be used to specify the conditions under which a test set derived by any means satisfies MCDC. Consider a candidate test set T. For each condition x, there must be at least one pair of test cases in T that differ only in the truth value of x, or else MCDC cannot be satisfied. If at least one such pair, with x excluded, satisfies dP/dx for each variable x, then, and only then, MCDC is achieved.

6.1 Example

This subsection develops an example, $A \wedge (B \vee C)$, with both the Pairs Table approach and via boolean difference.

The Pairs Table approach in [21] begins with constructing a truth table for $A \wedge (B \vee C)$, 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.

Case	ABC	Result	A	B	C
1	111	1	5		
2	110	1	6	4	
3	101	1	7		
4	100	0		2	3
5	011	0	1		
6	010	0	2		
7	001	0	3		
8	000	0			

Table 2. Pairs Table for $A \wedge (B \vee C)$

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

1. $dP/dA \;=\; A \wedge (B \vee C) \oplus \bar{A} \wedge (B \vee C) = B \vee C$
2. $dP/dB \;=\; A \wedge (B \vee C) \oplus A \wedge (\bar{B} \vee C) = A \wedge \bar{C}$
3. $dP/dC \;=\; A \wedge (B \vee C) \oplus A \wedge (B \vee \bar{C}) = A \wedge \bar{B}$

Test sets are generated as follows:

1. From dP/dA, select $B \vee C$ 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) $\{110, 010\}$
   (c) $\{101, 001\}$
2. From dP/dB, select $A \wedge \bar{C}$ true and B both true and false, yielding $\{110, 100\}$.

3. From dP/dC, select $A \wedge \bar{B}$ true and C both true and false, yielding $\{101, 100\}$.

Next, combine the test sets generated above. There are three possible test cases:

1. $\{111, 110, 101, 100, 011\}$
2. $\{110, 101, 100, 010\}$
3. $\{110, 101, 100, 001\}$

The second and third possibilities are more desirable since they use the minimum number $(n+1)$ of tests.

6.2 Coupled Conditions

Sometimes, conditions cannot be varied independently. Two conditions are *strongly coupled* if varying one always varies the other. Two conditions are *weakly coupled* if varying one sometimes, but not always, varies the other. *Weak* MCDC treats strongly coupled conditions as one condition. Clearly, the boolean difference approach can satisfy weak MCDC coverage by simply replacing strongly coupled conditions with a single condition.

Strong MCDC requires that each condition to be treated as if it were independent. In particular, repeated instances of a single condition are treated separately. Again, the boolean difference approach applies by simply considering each condition to be a separately named variable.

6.3 Comparing Test Methods

The results can be used to compare the theoretical effectiveness of published test methods. One published test generation method, Offutt 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 $A \vee B \vee C \vee \ldots$, 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 $A \wedge B \wedge C \wedge$ ..., 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 condition to detect a missing condition fault for x_i are given by:

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

This results in an expression of the form

$$\bar{x}_i \wedge x_j \wedge x_k ... \wedge \neg(\vee_{x_i \notin X_m} X_m).,$$

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

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

$$\bar{a} \wedge b \wedge c \wedge \neg(d \wedge e \wedge f)$$
$$\bar{b} \wedge a \wedge c \wedge \neg(d \wedge e \wedge f)$$
$$\bar{c} \wedge a \wedge b \wedge \neg(d \wedge e \wedge f)$$
$$\bar{d} \wedge e \wedge f \wedge \neg(a \wedge b \wedge c)$$
$$\bar{e} \wedge d \wedge f \wedge \neg(a \wedge b \wedge c)$$
$$\bar{f} \wedge d \wedge e \wedge \neg(a \wedge b \wedge c)$$

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 missing condition faults are dominated by variable negation faults, Offutt and Liu's method should be more efficient than these.

7 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 can be explained by 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, a superclass of missing condition faults. This is to be expected because a test for a variable negation 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.

8 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. Marv Zelkowitz provided helpful comments on demonstrating the practical benefits of the results.

References

[1] K.A. Foster. Sensitive test data for logical expressions. *ACM SIGSOFT software engineering notes*, 9(2), 1984.

[2] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5), 1994.

[3] A.M. Paradkar and K.C. Tai. Test generation for boolean expressions. In *Proc. IEEE International symposium on software reliability engineering*, 1995.

[4] A.M. Paradkar and K.C. Tai. Automatic test generation for predicates. In *Proc. IEEE International symposium on software reliability engineering*, 1996.

[5] A.M. Paradkar, K.C. Tai, and M.A. Vouk. Automatic test generation for predicates. *IEEE Transactions on Reliability*, 45(4), 1996.

[6] P. Stocks and D. Carrington. A framework for specification based testing. *IEEE Transactions on Software Engineering*, 9(22), 1996.

[7] A. J. Offutt and S. Liu. Generating test data from SOFL specifications. Technical report, George Mason University, 1997.

[8] M.A. Vouk, A.M. Paradkar, and K.C. Tai. Empirical studies of predicate-based software testing. In *Proc. IEEE International symposium on software reliability engineering*, pages 55–65, Nov. 1994.

[9] L.J. Morrell. Theoretical insights into fault-based testing. In *Proceedings of the Second Workshop oin Software Testing, Verification, and Analysis*. ACM/SIGSOFT, 1988.

[10] D.J. Richardson and M.C. Thompson. The relay model of error detection and its application. In *Proceedings of the Second Workshop oin Software Testing, Verification, and Analysis*. ACM/SIGSOFT, 1988.

[11] M. Abramovici, M.A. Breuer, and A.D. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, 1990.

[12] D.J. Richardson and M.C. Thompson. An analysis of test data selection criteria using the relay model of fault detection. *IEEE Transactions on Software Engineering*, 19(5), 1993.

[13] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: help for the practicing programmer. *IEEE Computer*, 11(4), 1978.

[14] A.J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on software engineering methodology*, 1(1), 1992.

[15] I.S. Reed. A class of multiple-error correcting codes and the decoding scheme. *Transactions of the Institute of Radio Engineers*, IT-4, 1954.

[16] S.B. Akers. On a theory of boolean functions. *SIAM Journal*, 7(4), 1959.

[17] F.M. Brown. *Boolean Reasoning.* Kluwer Academic Publishers, 1990.

[18] D.A. Huffman. Solvability criterion for simultaneous logical equations. Technical Report AD 156-161, Massachusetts Institute of Technology, Jan. 1958.

[19] D.R. Kuhn. A technique for analyzing the effects of changes in formal specifications. *BCS Computer Journal*, 35(6), 1992.

[20] B. Marick. Two experiments in software testing. Technical Report UIUCDCS-R-90-1644, University of Illinois at Urbana-Champaign, 1990.

[21] J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5), 1994.

[22] RTCA. Software considerations in airborne systems and equipment. Technical Report DO-178B, RTCA Inc., December 16 1992.

Appendix

Boolean Difference

The boolean difference [15], [16], can be used to calculate the dependency of a boolean function on a literal x_i of that function. The boolean difference of F with respect to x_i, dF/dx_i, gives the conditions under which the value of F will change if the value of x_i changes.

For a function $F = f(x_1, ..., x_i, ..., x_n)$, the boolean difference of F with respect to x_i is

$$dF/dx_i = f(x_1, ..., x_i, ..., x_n) \oplus f(x_1, ..., \bar{x}_i, ..., x_n).$$

This is equivalent to $dF/dx_i = f(x_1, ..., 0, ..., x_n) \oplus f(x_1, ..., 1, ..., x_n)$,

which follows from the fact that x_i must be either 0 or 1. The difference dF/dx_i is an expression that does not contain x_i.

A useful property of the boolean difference is that

$$dF/dx_i = \begin{cases} 1 & \text{if } F \text{ is unconditionally dependent on } x_i \\ 0 & \text{if } F \text{ is unconditionally independent on } x_i \\ F' & \text{an expression not containing } x_i\text{, otherwise} \end{cases}$$

The boolean difference of a function $F = f(F_1, ..., F_n)$, with respect to one of its component functions F_i is $dF/dF_i = f(F_1, ..., F_i, ..., F_n) \oplus f(F_1, ..., \neg F_i, ..., F_n)$.

The *partial boolean difference* gives the effect on the truth value of a boolean formula of a component of the formula, through a particular term. For a formula $F = f(F_1, ..., F_n)$, the partial boolean difference of F with respect to F_i with respect to a variable x_j of F_i, is $dF/d(x_j|F_i) = dF/dF_i \wedge dF_i/dx_j$

Predicate Difference

The predicate difference [19] for a predicate P with respect to variable substitution $x := e$, denoted dP^x_e, is $P \oplus P^x_e$.

The properties of the predicate difference are similar to those of the boolean difference. However, the boolean difference with respect to a term gives the conditions under which a change in the value of the term will change the value of the boolean function. A boolean term can change only from x to $\neg x$. The change to a predicate depends on the expression substituted for x. Thus a predicate difference is with respect to a particular change $x := e$ (the substitution of expression e for free variable x), rather than simply with respect to x. Note also that the predicate difference with respect to a change $x := e$ may still contain x:

$$dP^x_e = \begin{cases} 1 & \text{if } P \text{ is unconditionally dependent on } x_i \\ 0 & \text{if } P \text{ is unconditionally independent on } x_i \\ F' & \text{an expression, possibly containing } x\text{, otherwise} \end{cases}$$

If dP_e^x is not 0 and not 1, then the resulting formula can be solved for 1 to determine the conditions under which P_e^x will be dependent on x. Note that if x is a boolean term and $e = \bar{x}$ in a propositional formula, the predicate difference is equivalent to the boolean difference.

Partial Predicate Difference

The predicate difference of a predicate formula $F = f(F_1, ..., F_n)$, consisting of component formulas connected by $\wedge, \vee, or \Rightarrow$ with respect to one of its component formulas F_i is

$$dF/dF_i = f(F_1, ..., F_i, ..., F_n) \oplus f(F_1, ..., \neg F_i, ..., F_n).$$

The partial predicate difference gives the effect on a formula of a component of the formula, through a change in a particular term. For a formula $F = f(F_1, ..., F_n)$, the partial predicate difference of F with respect to F_i with respect to a change in a variable $x_j := e$ of F_i, is

$$dF/d(F_i)_e^{xj} = dF_{\neg F_i}^{F_i} \wedge dF_{ie}^{xj}$$

The Relationship Between Predicate Differences and Boolean Differences

The boolean difference can be viewed as an "upper bound" on the result of changes to an individual variable in a component formula. The change from a variable in a component formula is never larger than the change that results from negating the entire component formula.

The predicate difference of a predicate formula $F = f(F_1, ..., F_n)$, consisting of component formulas connected by $\wedge, \vee, or \Rightarrow$ with respect to one of its component formulas F_i is

$$dF_{\neg F_i}^{F_i} = f(F_1, ..., F_i, ..., F_n) \oplus f(F_1, ..., \neg F_i, ..., F_n)$$

which is equivalent to the boolean difference of F with respect to F_i. The following theorem [19] shows the relationship between this boolean difference with respect to a component formula and the partial predicate difference with respect to a variable of the component formula.

Theorem 8.1 *A substitution $x := e$ in a component formula F_i (where x is some variable in F_i) of a formula F will change the value of F only if a change in the value of F_i will change the value of F:*

$$dF/d(F_i)^x_e \Rightarrow dF^{F_i}_{\neg F_i}.$$

Applying modus tollens gives the following corollary:

Theorem 8.2 *If the boolean difference is 0, then the predicate difference is 0 as well:* $(dF^{F_i}_{\neg F_i}) = 0 \Rightarrow dF/d(F_i)^{x_j}_e = 0$.

Anti-difference

We define the *anti-difference* of a predicate P to be the set of predicates S_i for which $dS_i = P$. The anti-difference will be taken with respect to some substitution $x := e$,

$$\alpha P^e_x = \{S_i\}$$

where $dS_i/d(x := e) = P$

For fault-based testing the anti-difference is significant because it defines the class of predicates for which a test condition will detect an error. For example, the test condition $P = x \wedge \bar{z}$ will detect variable negation errors in predicates S_i where $dS_i/d(y/\bar{y}) = P$