Comparison of Fault Classes in Specification-Based Testing

Vadim Okun [a,b] Paul E. Black [a] Yaacov Yesha [b]

[a] *National Institute of Standards and Technology, Gaithersburg, MD 20899, USA*

[b] *University of Maryland Baltimore County, Baltimore, MD 21250, USA*

Abstract

Our results extending Kuhn's fault class hierarchy provide a justification for the focus of fault-based testing strategies on detecting particular faults and ignoring others. We develop a novel analytical technique which allows us to elegantly prove that the hierarchy applies to arbitrary expressions, not just those in disjunctive normal form. We also use the technique to extend the hierarchy to a wider range of fault classes. To demonstrate broad applicability, we compare faults in practical situations and analyze previous results. In particular, using our technique, we show that the basic meaningful impact strategy of Weyuker et al. tests for stuck-at faults, not just variable negation faults.

Key words: Fault-based testing; Specification-based testing; Kuhn's hierarchy; Fault classes

1 Introduction

Fault-based testing focuses on generating tests to detect faults in software [26,15,5,25]. It can often guarantee the absence of particular faults, which is an important advantage over other testing approaches. Since testing for all conceivable faults is impossible, fault based testing targets prespecified classes of faults. Experience shows that resulting tests are also effective for detecting faults in other classes [1].

Is there an analytical foundation to expect such effectiveness? Kuhn's hierarchy [12] of fault classes implies that some faults may be skipped during testing.

Email addresses: vokun1@cs.umbc.edu (Vadim Okun), paul.black@nist.gov (Paul E. Black), yayesha@cs.umbc.edu (Yaacov Yesha).

Previous results [12,23,1,13] were proved for specifications in disjunctive normal form.

We are able to remove the restriction to disjunctive normal form. This is significant because specifications are seldom written entirely in disjunctive normal form, yet some testing literature pertains solely to normalized specifications. However, fault-based testing from normalized specifications may miss faults which would have been detected if testing were done from the original specifications [24,1]. So having the option of testing from the original specification is better in some cases and is never worse.

Kuhn developed the hierarchy based on detection conditions for fault classes. We apply the RELAY model [21] to refine Kuhn's approach. In particular, we define the detection condition as the conjunction of origination and propagation conditions. This enables us to prove relationships between fault classes for arbitrary predicates instead of being restricted to disjunctive normal form. We also extend the hierarchy to include additional fault classes. Our analysis is not restricted to Boolean specifications. In particular, some faults occurring in relational expressions are considered.

The use of fault conditions enables us to analyze existing testing methods. For instance, we find that the basic meaningful impact strategy [24] is stronger in that it tests for stuck-at faults and not variable negation faults as was claimed by the authors. The strategy happens to also detect variable negation faults because those are "easier to detect" than the stuck-at faults.

The remainder of this introduction presents definitions and notation, and describes relevant fault classes and our model of faults.

1.1 Definitions and Notation

In this paper, a *test case* is an assignment of values to variables in an expression.

The following notation is used throughout the paper. A horizontal line above an operand represents negation. $\vee$ and $\wedge$ represent disjunction and conjunction, respectively. Occasionally, when clear from the context, $\wedge$ is omitted. $\oplus$ and $\leftrightarrow$ stand for exclusive-or and equivalence, respectively. $\rightarrow$ represents implication. Among the Boolean operators, negation has the highest precedence, and $\wedge$ has higher precedence than other binary operators. 1 and 0 denote "true" and "false," respectively.

A *clause* [1] is either a Boolean variable or a relational expression, possibly negated. A *relational expression* is of the form $E\ op\ F$, where E and F are arithmetic expressions and op is one of $<, \leq, =, \neq, >$, or $\geq$.

A *compound predicate* consists of one or more binary Boolean operators and their operands, and possibly negation operators and parenthesis.

A *predicate* [2] is either a clause or a compound predicate.

For example, $x < 5$ is a clause, and $((x < 5) \vee (y > x)) \wedge (\overline{f \vee g})$ is a compound predicate. If a clause appears more than once in a predicate, we consider each occurrence to be a distinct clause. This affects our definition of fault classes as explained in the next section.

A *Boolean formula* consists of Boolean variables and possibly Boolean operators and parenthesis. In other words, a Boolean formula is a predicate with no relational expressions.

1.2 Fault Classes

Faults may involve Boolean variables, Boolean operators, relational operators, or arithmetic expressions. We first list faults that involve a clause, then later list faults that affect compound predicates. In the following fault classes, a clause is implemented incorrectly as a (possibly empty) predicate.

- Clause Reference Fault (CRF) - replace a clause c with another clause, d. For example, the specification $(x < 5) \vee (y > 3)$ is implemented as $(x < 4) \vee (y > 3)$.
- Clause Negation Fault (CNF) - replace a clause c by its negation $\bar{c}$.
- Clause Insertion Fault (CIF) - insert a clause d, that is, replace a clause c by $c \circ d$, where d is another clause, $\circ$ is either conjunction or disjunction. There are two subclasses of this class.
  · Clause Conjunction Fault (CCF) - replace a clause c by $c \wedge d$.
  · Clause Disjunction Fault (CDF) - replace a clause c by $c \vee d$.
- Relational Operator Reference Fault (RRF) - replace a relational operator by any other relational operator. Note that replacing a relational operator with its opposite is the same as negating the whole relational expression.
- Off-By-1 Fault (OFF) - in a relational expression $E_1 operator E_2$, replace the arithmetic expression E_2 with $E_2 + 1$ and $E_2 - 1$.
- Stuck-At Fault (STF) - stuck-at-0 replaces a clause with 0, stuck-at-1 replaces it with 1.

[1] The terms *condition* or *simple predicate* are also used in the literature.
[2] Sometimes we use the term *expression*.

- Missing Clause Fault (MCF) - a clause is omitted during implementation. For instance, the specification $c \wedge d \vee e$ is implemented as $c \vee e$.

Boolean formulas are often used to formally specify real-world systems [14]. For this reason, testing based on Boolean specifications is often studied exclusively [24,2]. The results in this paper can easily be adapted to Boolean specifications since many fault classes in Boolean formulas are special cases of the fault classes listed above. In particular, in case of Boolean specifications or when only Boolean variables are involved in the substitution, CRF, CNF, and CIF become variable reference fault (VRF), variable negation fault (VNF), and variable insertion fault (VIF), respectively. VRF, RRF, OFF, and STF are subclasses of CRF. CRF also includes some unlikely faults such as replacing a Boolean variable with a relational expression. For this reason, we consider its subclasses separately. CNF represents realistic faults: a relational expression may be implemented as its negation. Whenever a proof is given for CNF, a similar proof can be given for VNF.

Most of the fault classes involve replacing a clause. Even though the same clause may occur more than once in an expression, a single fault is a change to just one of the occurrences, not to all of them simultaneously. This approach to faults is also taken by [23] and [13]. These fault classes correspond closely to faults that may occur in software specifications, where one occurrence of a clause or a variable may be replaced as a result of an error while another occurrence is correct. This is in contrast with hardware design, where, for example, a stuck-at-0 fault on a line of a logic circuit results in all occurrences of the corresponding clause being replaced with 0. So in this paper, whenever we refer to a clause (variable) in an expression, we mean a single clause (variable) occurrence.

Some additional fault classes, where a predicate (possibly compound) is implemented incorrectly, are as follows.

- Expression Negation Fault (ENF) - replace an expression X by $\overline{X}$.
- Missing Expression Fault (MEF) - a predicate is omitted during implementation. MEF includes both where a clause is missing and where a compound predicate is missing.
- Logical Operator Reference Fault (LRF) - a Boolean operator is replaced by another operator, e.g., $x \wedge y$ is replaced by $x \vee y$.
- Associative Shift Fault (ASF) - change the associativity of terms. For example, replace $(ab) \vee c$ with $a(b \vee c)$.

The rest of this paper is organized as follows. Section 2 presents background, including Kuhn's fault hierarchy. Section 3 defines the fault conditions and explains our analytical approach. Section 4, the heart of the paper, uses the approach to prove the fault class hierarchy. Section 5 applies the fault conditions and fault hierarchy to compare several specific fault classes commonly occurring in predicates, consider some previous empirical observations, and analyze the basic meaningful impact strategy of Weyuker et al. Section 6 summarizes the results and presents conclusions.

2 Related Work

Mutation analysis [5] is a fault-based testing technique that uses "mutation operators" to introduce small changes, or mutations, into the program or specification, producing a mutant, and then chooses a test case to distinguish the mutant from the original. For mutation testing to be practical, mutation operators model a limited number of typical single fault classes. A fault is called *simple* or *single* if the faulty version differs from the original by exactly one syntactic change.

Mutation testing is one of a myriad of testing criteria proposed in the literature. A *testing criterion* [8] specifies what properties of a program or specification must be exercised to constitute a thorough test. Criteria for generating tests from state-based specifications are presented in [19]. *Subsumption* relationship is a widely accepted method for comparing different testing criteria. A criterion C_1 is said to subsume another criterion C_2 if and only if any test set that satisfies C_1 also satisfies C_2. A subsumption hierarchy for several path selection criteria was developed in [4]. Many criteria based on logical control flow through a program [3] are subsumed by mutation testing [18].

There is an extensive body of research that studied conditions for detecting a fault from the program output [7,6,16,21,25,10]. The RELAY model [21] defines the revealing conditions under which a fault is detected. First, a potential error originates at the smallest subexpression containing the fault, that is, the subexpression evaluates incorrectly. Then the potential error transfers through computations and information flow. Finally, a failure is revealed in the outputs. The model provides a mechanism for developing failure conditions that guarantee fault detection. In particular, the transfer conditions for Boolean operators are defined. The transfer condition guarantees that a potential failure is not masked out by the computation of a parent operator. We apply the RELAY model to construct the detection conditions for predicates.

2.1 Kuhn's Fault Hierarchy

Kuhn [11] invented the technique of predicate differences for analyzing the effects of faults in specifications. Briefly, it is as follows. Let S denote a specification predicate hypothesized to be correct and S' a faulty version of it. A test detects the fault if and only if it causes S' to evaluate to a different value than S, formally when $S \oplus S'$. The predicate $S \oplus S'$ is referred to as the detection condition for the fault. The predicate difference is a generalization of the Boolean difference [20] used in hardware testing.

Several researchers [12,23,1,13] used Kuhn's technique to compare fault classes in Boolean specifications restricted to disjunctive normal form, that is, a disjunction of terms. A *term* is a conjunction of literals, a *literal* is an occurrence of a variable or its negation.

Kuhn [12] compared the detection conditions for variable reference fault (VRF), variable negation fault (VNF), and expression negation fault (ENF) and proved that they form a hierarchy with respect to detectability. That is, any test that detects a VRF for some variable also detects a VNF for the same variable, and any test that detects a VNF for some variable also detects an ENF for the expression in which the variable occurs. Tsuchiya and Kikuno [23] proved that tests that detect missing clause fault (MCF) will also detect VNF. Tsuchiya and Kikuno also showed that tests that detect MCF may not be able to detect VRF, and vice versa. They also showed that a test set that detects MCFs for single variable terms, as well as VRFs, is sufficient to detect both VRFs and MCFs.

Lau and Yu [13] extended the hierarchy to include several other fault classes that can occur in Boolean specifications. They considered literal insertion fault (LIF), literal reference fault (LRF), literal and term omission faults (LOF and TOF), literal, term, and expression negation faults (LNF, TNF, and ENF). They concluded that a test case that detects LIF can also detect LRF and TOF, a test case that detects either LRF, TOF, or LOF can also detect LNF, a test case that detects LNF can also detect TNF, a test case that detects TNF can also detect ENF.

All these results apply to Boolean specifications in disjunctive normal form. This paper studies more general fault classes. For instance, the literal insertion fault in [13] is a special case of the clause conjunction fault when specifications are restricted to disjunctive normal form.

Detection condition is an effective and concise analytical tool for studying faults in formal specifications. It can be thought of as a formalization (in the case of predicates) of "necessity condition" [6] which is described as follows: "for a test to differentiate a mutant program from the original program, the execution state of the mutant program must differ from that of the original program after some execution of the mutated statement". We refine the fault detection conditions.

3 Fault Conditions

Since we use the following identities throughout the rest of the paper, we present them here together. For any predicates f, g, h:

$$\begin{aligned} f \wedge h \oplus g \wedge h &= (f \oplus g) \wedge h && (1) \\ f \wedge h \oplus h &= \bar{f} \wedge h && (2) \\ (f \vee h) \oplus (g \vee h) &= (f \oplus g) \wedge \bar{h} && (3) \\ (f \vee h) \oplus h &= f \wedge \bar{h} && (4) \\ f \oplus h \oplus \bar{f} \oplus h &= 1 && (5) \\ (f \vee g) \oplus (f \wedge g) &= f \oplus g && (6) \\ ((f \wedge g) \rightarrow g) &= 1 && (7) \end{aligned}$$

Identities (2) and (4) follow from (1) and (3), respectively. With these identities at hand, we can analyze fault conditions for various fault classes.

3.1 Origination Condition

Suppose X is the smallest subpredicate of a specification S corresponding to a fault, that is, X is implemented as a predicate E. Then the *origination condition* for the fault is $X \oplus E$, in other words, E evaluates to a different value than X.

For example, a specification $S = (x < 5) \vee a$ may be implemented incorrectly as $(x < 4) \vee a$, that is, the implementation contains an Off-by-1 fault. Then, the smallest subpredicate of S corresponding to the fault is the clause $(x < 5)$, and the origination condition is $(x < 5) \oplus (x < 4)$ or $x = 4$. On the other hand, suppose that S is implemented as $(x < 5) \wedge a$, a logical operator reference fault. Then, the smallest subpredicate of S corresponding to the fault is S itself, and the origination condition is $((x < 5) \vee a) \oplus ((x < 5) \wedge a)$ or $(x < 5) \oplus a$.

3.2 Propagation Condition

In what cases will the value of a predicate be affected if one part is faulty? Concretely, if R is some predicate such as $P \wedge Q$, $P \vee Q$, or $P \oplus Q$, what value of Q will let a change in the value of P lead to a change in the value of R? For completeness, we include the case of $R = \overline{P}$. Formally, let $R = op(P, [Q])$ denote either $R = \overline{P}$ or $R = P \circ Q$, where $\circ$ is a binary Boolean operator.

The *propagation condition* guarantees that the value of R will change if the value of P changes. Denote the propagation condition for operator op as

$$\widehat{Q} = op(1, [Q]) \oplus op(0, [Q])$$

An alternate but equivalent definition of $\widehat{Q}$ is

$$\widehat{Q} = op(P, [Q]) \oplus op(\overline{P}, [Q])$$

For instance, when $R = P \vee Q$, the propagation condition is

$$(P \vee Q) \oplus (\overline{P} \vee Q) = \overline{Q} \quad \text{by (3)}$$

Using identities (1), (3) and (5), the propagation conditions for fundamental operators are as follows:

$$\widehat{Q} = \begin{cases} 1 \text{ if } R = \overline{P} \text{ or } R = P \oplus Q \\ Q \text{ if } R = P \wedge Q \\ \overline{Q} \text{ if } R = P \vee Q \end{cases}$$

Propagation conditions for the other binary Boolean operators fall into one of the three categories above, since they can be expressed using the fundamental operators without duplicating the clause occurrences involved. That is, $P \to Q = \overline{P} \vee Q$, $P \leftrightarrow Q = \overline{P} \oplus Q$.

More generally, we can define the propagation condition for a subpredicate X of some larger predicate. It guarantees that a fault in X is not masked by the computation of parent expressions. In other words, it is the condition under which the value of specification S will change if the value of its subpredicate X changes.

Let $P_0, \ldots, P_n$ be predicates, such that $S = P_0$, $P_{i-1} = op_i(P_i, [Q_i])$, $i = 1 \ldots n$, $X = P_n$. The series of predicates P_i can be seen as the path in the expression tree of S from the root to X. Each Q_i is the subtree on the branch which is not on the path. The propagation condition for a fault in X is the conjunction of the propagation conditions for each op_i:

$$\frac{dS}{dX} = \widehat{Q_1} \wedge \widehat{Q_2} \wedge \cdots \wedge \widehat{Q_n}$$

Suppose a specification

$$F = (x \leftrightarrow y)w \vee (\overline{vzw}) \qquad (8)$$

Table 1
Computing the propagation condition for clause z in $(x \leftrightarrow y)w \vee (\overline{vzw})$.

i	P_i	Q_i	op_i	$\widehat{Q_i}$
1	$\overline{vzw}$	$(x \leftrightarrow y)w$	$Q_1 \vee P_1$	$\overline{Q_1}$
2	vzw	none	$\overline{P_2}$	1
3	z	vw	$P_3 \wedge Q_3$	Q_3

has a variable reference fault where z is replaced by x. The propagation condition for a fault in z can be computed from Table 1. There, i is the index in P_i. It follows that

$$\begin{aligned}\frac{dF}{dz} &= \widehat{Q_1}\widehat{Q_2}\widehat{Q_3} = \overline{Q_1} \wedge Q_3 \\ &= \overline{(x \leftrightarrow y)w} \wedge vw = (\overline{x \leftrightarrow y} \vee \overline{w})vw = (x \oplus y)vw\end{aligned}$$

There may be more than one occurrence of the same clause in a predicate, for instance, variable w occurs twice in (8). This makes the notation $\frac{dF}{dw}$ ambiguous. However, the concept of clause replacement is unambiguous since each occurrence is considered to be a distinct clause and a fault is a change to just one of the occurrences. Rather than use an awkward but unambiguous notation, we trust that the reader will understand that the claims made in this paper have to do with replacing one clause at a time, never several clauses simultaneously.

It turns out that, given two predicates R and P on a path in the expression tree of specification S, such that R is an ancestor of P on the path, if the propagation condition for P is satisfied, then the propagation condition for R is guaranteed to be satisfied. This is stated formally as Lemma 1.

Lemma 1 *Let R be a subpredicate of predicate S. If P is a subpredicate of R, then*

$$\frac{dS}{dP} \rightarrow \frac{dS}{dR}.$$

Proof. Let $P_0, \ldots, P_k, \ldots, P_n, 0 < k < n$, be predicates, such that $S = P_0$, $P_{i-1} = op_i(P_i, [Q_i])$, $i = 1 \ldots n$, $R = P_k$, $P = P_n$. Then

$$\begin{aligned}\frac{dS}{dP} &= \widehat{Q_1} \wedge \cdots \wedge \widehat{Q_k} \wedge \widehat{Q_{k+1}} \wedge \cdots \wedge \widehat{Q_n} \\ \frac{dS}{dR} &= \widehat{Q_1} \wedge \cdots \wedge \widehat{Q_k}\end{aligned}$$

In view of (7), the Lemma holds. Q.E.D.

3.3 Detection Condition

The notation $S^{\mathrm{X}}_{\mathrm{E}}$ signifies that a subpredicate X of specification S is replaced by a predicate E. Kuhn's original definition [12] of the detection condition for the fault is $dS^{\mathrm{X}}_{\mathrm{E}} = S \oplus S^{\mathrm{X}}_{\mathrm{E}}$, in other words, $S^{\mathrm{X}}_{\mathrm{E}}$ evaluates to a different value than S. For example, if $E = \bar{X}$, an expression negation fault, the detection condition is $dS^{\mathrm{X}}_{\bar{\mathrm{X}}} = S \oplus S^{\mathrm{X}}_{\bar{\mathrm{X}}}$.

For example, the detection condition for the fault where z is replaced by x in (8) is

$$dF^{\mathrm{z}}_{\mathrm{x}} = ((x \leftrightarrow y)w \vee (\overline{vzw})) \oplus ((x \leftrightarrow y)w \vee (\overline{vxw}))$$

It follows that, for instance, a test case $(x, y, z, v, w) = (1, 0, 0, 1, 1)$ will detect the fault because this assignment of values to variables satisfies $dF^{\mathrm{z}}_{\mathrm{x}}$.

This illustrates a limitation of this definition: the formula is not easy to manipulate, especially when one would like to prove that a property of detection conditions holds for any specification. Proofs in [12] for the restricted case of disjunctive normal form involve manipulating large formulas. Our reformulation, although semantically equivalent to Kuhn's definition, allows for more generally applicable, yet more succinct, proofs.

We define the *detection condition* as a conjunction of origination condition and propagation condition:

$$dS^{\mathrm{X}}_{\mathrm{E}} = (X \oplus E) \wedge \frac{dS}{dX} \tag{9}$$

For example, the detection condition for the fault where z is replaced by x in (8) is

$$dF^{\mathrm{z}}_{\mathrm{x}} = (z \oplus x) \wedge \frac{dF}{dz} = (z \oplus x)(x \oplus y)vw$$

4 Analytical Comparison of Fault Classes

This section uses fault conditions to derive the relationships between several fault classes. Some practical implications of these results are presented in Section 5.

Table 2
Detection conditions for several fault classes.

$\mathcal{S}_{\mathrm{CRF}}$	$dS^{\mathrm{x}}_{\mathrm{y}}$	Clause Reference Fault
$\mathcal{S}_{\mathrm{CNF}}$	$dS^{\mathrm{x}}_{\bar{\mathrm{x}}}$	Clause Negation Fault
$\mathcal{S}_{\mathrm{ENF}}$	$dS^{\mathrm{X}}_{\overline{\mathrm{X}}}$	Expression Negation Fault
$\mathcal{S}_{\mathrm{CCF}}$	$dS^{\mathrm{x}}_{\mathrm{x}\wedge\mathrm{y}}$	Clause Conjunction Fault
$\mathcal{S}_{\mathrm{CDF}}$	$dS^{\mathrm{x}}_{\mathrm{x}\vee\mathrm{y}}$	Clause Disjunction Fault

The notation $\mathcal{S}_{\mathrm{F}}$ is used to represent the detection condition for an arbitrary fault belonging to fault class F. The detection conditions for fault classes CRF, CNF, ENF, CCF, and CDF are summarized in Table 2. There, x is a clause in S, y is another valid clause [3], and X is an expression in S.

First consider the relationship between clause reference faults (CRF) and clause negation faults (CNF).

Theorem 1 *If the clause replaced in $\mathcal{S}_{\mathrm{CRF}}$ is the same clause negated in $\mathcal{S}_{\mathrm{CNF}}$, then $\mathcal{S}_{\mathrm{CRF}} \rightarrow \mathcal{S}_{\mathrm{CNF}}$.* [4]

Proof. By Table 2, we must establish that, for a predicate P and a clause x occurring in P, $dP^{\mathrm{x}}_{\mathrm{y}} \rightarrow dP^{\mathrm{x}}_{\bar{\mathrm{x}}}$ holds, where $y \neq x$ is another valid clause.

Rewriting with (9), we have

$$((x \oplus y) \wedge \frac{dP}{dx}) \rightarrow ((x \oplus \bar{x}) \wedge \frac{dP}{dx})$$

Since $x \oplus \bar{x} = 1$, and in view of (7), the Theorem holds. Q.E.D.

By Theorem 1, a test case, that is, an assignment of values to variables, which makes $dP^{\mathrm{x}}_{\mathrm{y}}$ true, also makes $dP^{\mathrm{x}}_{\bar{\mathrm{x}}}$ true. This is stated as the following corollary.

Corollary 1 *Any test case that detects a clause reference fault for a clause in a predicate will also detect the clause negation fault for the same clause.*

It must be noted that $dP^{\mathrm{x}}_{\mathrm{y}} \rightarrow dP^{\mathrm{x}}_{\bar{\mathrm{x}}}$ in Theorem 1 does not guarantee the existence of a test for the clause reference fault. For instance, if $P^{\mathrm{x}}_{\mathrm{y}}$ and P evaluate the same on their entire domain, then no test exists for the fault, even though there may be a test for the corresponding clause negation fault.

[3] While Kuhn [12] restricts y to be a variable in S, this paper only requires $S^{\mathrm{x}}_{\mathrm{y}}$ to be syntactically legal, that is, y has to be a valid clause. This applies to other faults involving clauses.
[4] Some papers indicate fault class domination by an arrow. In contrast, $\mathcal{S}_{\mathrm{CRF}}$ and $\mathcal{S}_{\mathrm{CNF}}$ are predicates, and the theorem states a logical implication.

But $dP^{\mathrm{x}}_{\mathrm{y}}$ is universally false in this case, so the theorem is still valid. However, if there actually is a test case for the clause reference fault, that test will detect the clause negation fault.

Another interesting relationship is between clause negation faults (CNF) and expression negation faults (ENF).

Theorem 2 *If the clause negated in $\mathcal{S}_{\mathrm{CNF}}$ occurs in the expression negated in $\mathcal{S}_{\mathrm{ENF}}$, then $\mathcal{S}_{\mathrm{CNF}} \rightarrow \mathcal{S}_{\mathrm{ENF}}$.*

Proof. By Table 2, we must establish that, for a predicate P, with a clause x occurring in a subpredicate E of P, $dP^{\mathrm{x}}_{\bar{\mathrm{x}}} \rightarrow dP^{\mathrm{E}}_{\overline{\mathrm{E}}}$ holds.

Rewriting with (9), we have

$$((x \oplus \bar{x}) \wedge \frac{dP}{dx}) \rightarrow ((E \oplus \overline{E}) \wedge \frac{dP}{dE})$$

Since the exclusive-or of a predicate with its negation is trivially true, this can be rewritten as

$$\frac{dP}{dx} \rightarrow \frac{dP}{dE}$$

Since clause x is a subpredicate of E, the theorem follows from Lemma 1. Q.E.D.

Corollary 2 *Any test case that detects a clause negation fault for a clause in a predicate will also detect an expression negation fault for an expression in which the clause occurs.*

Consider the relationship between the clause reference fault (CRF) class and the two clause insertion fault classes. Informally, tests for a clause conjunction fault (CCF) and tests for the corresponding clause disjunction fault (CDF) partition the set of test cases that detect the corresponding clause reference fault. Figure 1 presents this relationship. In terms of detection conditions, it means:

(1) The disjunction of $\mathcal{S}_{\mathrm{CCF}}$ and $\mathcal{S}_{\mathrm{CDF}}$ is equivalent to $\mathcal{S}_{\mathrm{CRF}}$.
(2) $\mathcal{S}_{\mathrm{CCF}}$ and $\mathcal{S}_{\mathrm{CDF}}$ are never satisfied simultaneously.

This is formalized in Theorem 3.

Theorem 3 *If a clause x is replaced with another clause y in $\mathcal{S}_{\mathrm{CRF}}$, and the same clause x is replaced with $x \wedge y$ in $\mathcal{S}_{\mathrm{CCF}}$, with $x \vee y$ in $\mathcal{S}_{\mathrm{CDF}}$, then*

$$((\mathcal{S}_{\mathrm{CCF}} \vee \mathcal{S}_{\mathrm{CDF}}) \leftrightarrow \mathcal{S}_{\mathrm{CRF}}) \wedge (\overline{\mathcal{S}_{\mathrm{CCF}}} \vee \overline{\mathcal{S}_{\mathrm{CDF}}})$$

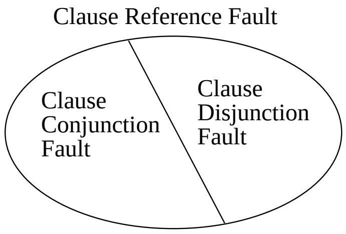

Fig. 1. Relationship between tests for a clause reference fault and tests for the corresponding clause insertion faults.

Proof. By Table 2, we must establish that, for a predicate P and a clause x occurring in P,

$$((dP^{\mathrm{x}}_{\mathrm{x \wedge y}} \vee dP^{\mathrm{x}}_{\mathrm{x \vee y}}) \leftrightarrow dP^{\mathrm{x}}_{\mathrm{y}}) \wedge (\overline{dP^{\mathrm{x}}_{\mathrm{x \wedge y}}} \vee \overline{dP^{\mathrm{x}}_{\mathrm{x \vee y}}})$$

where $y \neq x$ is another valid clause.

By (9), the detection conditions for CRF, CCF, and CDF are

$$\begin{aligned}
dP^{\mathrm{x}}_{\mathrm{y}} &= (x \oplus y) \wedge \frac{dP}{dx} \\
dP^{\mathrm{x}}_{\mathrm{x \wedge y}} &= (x \oplus (x \wedge y)) \wedge \frac{dP}{dx} = x\bar{y} \wedge \frac{dP}{dx} \quad \text{by (2)} \\
dP^{\mathrm{x}}_{\mathrm{x \vee y}} &= (x \oplus (x \vee y)) \wedge \frac{dP}{dx} = \bar{x}y \wedge \frac{dP}{dx} \quad \text{by (4)}
\end{aligned}$$

The disjunction of the detection conditions for CCF and CDF is

$$dP^{\mathrm{x}}_{\mathrm{x \wedge y}} \vee dP^{\mathrm{x}}_{\mathrm{x \vee y}} = (x\bar{y} \wedge \frac{dP}{dx} \vee \bar{x}y \wedge \frac{dP}{dx}) = (x \oplus y) \wedge \frac{dP}{dx} \tag{10}$$

Additionally,

$$\overline{x\bar{y} \wedge \frac{dP}{dx}} \vee \overline{\bar{x}y \wedge \frac{dP}{dx}} = \bar{x} \vee y \vee x \vee \bar{y} \vee \overline{\frac{dP}{dx}} = 1 \tag{11}$$

In view of (10) and (11), the Theorem holds. Q.E.D.

Corollary 3 *Any test case that detects a clause insertion fault in a predicate where a clause x is implemented as $x \vee y$ or as $x \wedge y$, y is another valid clause, will also detect the clause reference fault where the same clause x is implemented as y.*

Corollary 4 *Any test case that detects a clause reference fault in a predicate where a clause x is implemented as another valid clause y will also detect* *either the clause conjunction fault where the clause x is implemented as $x \wedge y$ or the clause disjunction fault where the clause x is implemented as $x \vee y$, but not both.*

Putting the results of this section together, Figure 2 depicts the hierarchy of tests that detect various fault classes in predicates. Note that this hierarchy applies to arbitrary predicates. It is not restricted to predicates in disjunctive normal form.

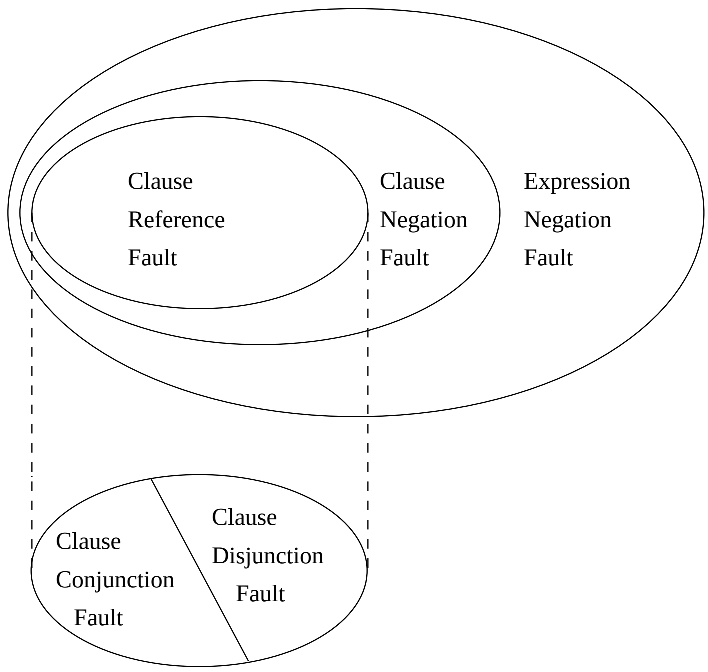

Fig. 2. Hierarchy of Fault Classes

5 Applications

The fault hierarchy in Figure 2 is rather general. Why should testing researchers care? This section presents examples of applying the hierarchy and fault conditions to specific cases. Section 5.1 discusses how the results apply to faults involving Boolean variables, as well as those occurring in relational expressions. Section 5.2 explains and discusses previous empirical observations about other fault classes including logical operator reference and missing clause faults. Section 5.3 analyzes the basic meaningful impact strategy [24].

Boolean variables, such as flags, commonly occur in predicates. Additionally, some specifications are written using Boolean formulas. Since the results in the previous section were proved for a more general case of clauses in predicates, they apply directly to Boolean formulas. These specific applications generalize similar results which were obtained earlier in [12,13] for Boolean formulas limited to disjunctive normal form.

While faults in relational expressions were thoroughly investigated [7,22], use of fault conditions allows us to compare these fault classes from another, more formal, perspective. The clause reference fault class includes relational operator reference faults and off-by-1 faults. Therefore, by Corollary 1,

- Any test case that detects a relational operator reference fault for a clause in a predicate will also detect a clause negation fault for the same clause.
- Any test case that detects an off-by-1 fault for a clause in a predicate will also detect a clause negation fault for the same clause.

For a relational expression, there are five possible relational operator reference faults for each of the other relational operators, as well as two possible off-by-1 faults. Fault conditions can be used to compare these faults. Consider, for instance, a specification S with a clause $E < F$, where E and F are arithmetic expressions. The detection condition for an off-by-1 fault which replaces $E < F$ with $E < F - 1$ is

$$\begin{aligned} dS^{\mathrm{E<F}}_{\mathrm{E<F-1}} &= ((E < F) \oplus (E < F - 1)) \wedge \frac{dS}{d(E < F)} \\ &= (E = F - 1) \wedge \frac{dS}{d(E < F)} \end{aligned}$$

since $E < F - 1$ evaluates to a different value than $E < F$ only when E is equal to $F - 1$. On the other hand, the detection condition for a relational operator fault that replaces $E < F$ with $E > F$ is

$$dS^{\mathrm{E<F}}_{\mathrm{E>F}} = ((E < F) \oplus (E > F)) \wedge \frac{dS}{d(E < F)}$$

Since $E < F$ and $E > F$ are never satisfied simultaneously,

$$dS^{\mathrm{E<F}}_{\mathrm{E>F}} = ((E < F) \vee (E > F)) \wedge \frac{dS}{d(E < F)}$$

$$= (E \neq F) \wedge \frac{dS}{d(E < F)}$$

Since $(E = F - 1) \rightarrow (E \neq F)$, $dS^{\mathrm{E<F}}_{\mathrm{E<F-1}} \rightarrow dS^{\mathrm{E<F}}_{\mathrm{E>F}}$.

It follows that any test case that detects an off-by-1 fault which replaces a clause $E < F$ with $E < F - 1$ will also detect a relational operator fault which replaces the same clause with $E > F$.

In a similar fashion, it is possible to derive a set of relationships between relational operator and off-by-1 faults for various relational expressions.

5.2 Analysis of Previous Observations

Gopal and Budd [9] note that a logical operator reference fault (LRF), where $\vee$ is substituted for $\wedge$ and vice versa, tends to be trivial to detect. Indeed, in view of (6), the detection condition for such a fault in a specification $S = P \vee Q$ is

$$dS^{\mathrm{P \vee Q}}_{\mathrm{P \wedge Q}} = (P \vee Q) \oplus (P \wedge Q) = P \oplus Q,$$

so this fault is detected by any test where P and Q evaluate differently. This explains Gopal and Budd's observation.

The detection condition for a corresponding missing expression fault (MEF) is

$$dS^{\mathrm{P \vee Q}}_{\mathrm{P}} = (P \vee Q) \oplus P = \bar{P} \wedge Q,$$

and $dS^{\mathrm{P \vee Q}}_{\mathrm{P}} \rightarrow dS^{\mathrm{P \vee Q}}_{\mathrm{P \wedge Q}}$. Hence, a test that detects an MEF for an operand of an $\vee$ operator will also detect an LRF where the operator is replaced by $\wedge$. A similar result can be obtained when $\wedge$ is replaced by $\vee$.

The above does not mean that all logical reference faults can be ignored. Consider an LRF where $\vee$ is replaced by $\oplus$. The detection condition is

$$dS^{\mathrm{P \vee Q}}_{\mathrm{P \oplus Q}} = (P \vee Q) \oplus (P \oplus Q) = P \wedge Q,$$

so the fault is relatively hard to detect. This is reasonable since $\vee$ differs from $\oplus$ in only one of four positions of the truth table, while it differs from $\wedge$ in two positions.

Stuck-at fault (STF) is one of the subclasses of the clause reference fault. Therefore, any test that detects a stuck-at fault for a clause in a predicate will also detect a clause negation fault for the same clause. It was suggested in [12] that missing clause fault (MCF) can be regarded as a special case of variable reference fault. However, it is more appropriate to compare MCF with STF. For instance, if a specification contains a conjunction $x \wedge y$, the result of an MCF where clause y is not implemented at all is equivalent to the result of an STF where y is replaced with 1. Similarly, in $x \vee y$ or $x \oplus y$, the result of an MCF for clause y is equivalent to the result of an STF where y is replaced with 0. Implication is a special case. Consider specification S with implication $x \rightarrow y$. There are two cases. First, the result of an MCF where clause x is not implemented at all is equivalent to the result of an STF where x is replaced with 0. Second, the detection condition for an MCF for y is

$$dS_{\mathrm{x}}^{\mathrm{x}\rightarrow\mathrm{y}} = ((x \rightarrow y) \oplus x) \wedge \frac{dS}{dy} = (\bar{x} \vee \bar{y}) \wedge \frac{dS}{dy}.$$

On the other hand, the detection condition for an STF where y is replaced with 1 is

$$dS_{1}^{\mathrm{y}} = ((x \rightarrow y) \oplus (x \rightarrow 1)) \wedge \frac{dS}{dy} = x\bar{y} \wedge \frac{dS}{dy}.$$

Since $x\bar{y} \rightarrow (\bar{x} \vee \bar{y})$, $dS_{1}^{\mathrm{y}} \rightarrow dS_{\mathrm{x}}^{\mathrm{x}\rightarrow\mathrm{y}}$. To summarize, if a test generation strategy guarantees detection of both stuck-at-0 and stuck-at-1 faults for a clause, it will also guarantee detection of the missing clause fault for the same clause.

5.3 On the Basic Meaningful Impact Strategy

Weyuker et al. [24] designed the *meaningful impact strategy* for testing Boolean formulas in *irreducible* disjunctive normal form. A formula is said to be in irreducible disjunctive normal form when none of the formula's literals or terms can be deleted without altering the formula's value for some test case.

We briefly repeat here the relevant definitions. More details are in [24]. Let F be a Boolean formula in irreducible disjunctive normal form with n variables and m product terms: $p_1 \vee p_2 \vee \ldots \vee p_m$. Each term is a conjunction of literals. Recall that a literal is a single occurrence of a variable or its negation.

The points of the input space are divided into two categories: *true points* and *false points* are those that cause the formula to evaluate to 1 and 0, respectively. True points for the term p_i are the points of the input space that cause p_i to evaluate to 1. Denote these points by R_i. The *unique true points* for the term p_i are those points that are in R_i but do not belong to any other R_j. Denote these points by U_i.

Let $p_{i,j}$ denote the product-term obtained by complementing the jth literal of the product-term p_i. Denote the set of true points for $p_{i,j}$ by $D_{i,j}$. Denote the points in $D_{i,j}$ that are false points for F by $N_{i,j}$.

The basic meaningful impact strategy is defined as follows [24]:

(1) Select one test point from each nonempty U_i of F.
(2) Select one test point from each $N_{i,j}$ of F.

Weyuker et al. [24] claim that the strategy is testing directly for a variable negation fault. In fact, the strategy is stronger: it is testing for stuck-at faults.

To show this, we first compute the propagation conditions for a fault in an arbitrary term and for a fault in an arbitrary literal for a specification in disjunctive normal form. Since F can be rewritten as

$$p_i \vee (p_1 \vee \ldots \vee p_{i-1} \vee p_{i+1} \ldots \vee p_m),$$

it follows that the propagation condition for a fault in p_i is

$$\begin{aligned} \frac{dF}{dp_i} &= \overline{p_1 \vee \ldots p_{i-1} \vee p_{i+1} \vee \ldots p_m} \\ &= \overline{p_1} \wedge \ldots \overline{p_{i-1}} \wedge \overline{p_{i+1}} \wedge \ldots \overline{p_m} \end{aligned} \tag{12}$$

Let $p_i = l_1 \ldots l_k$, where l_j denotes the jth literal in p_i. Then F can be rewritten as

$$l_j l_1 \ldots l_{j-1} l_{j+1} \ldots l_k \vee (p_1 \vee \ldots \vee p_{i-1} \vee p_{i+1} \ldots \vee p_m)$$

The propagation condition for a fault in l_j is

$$\frac{dF}{dl_j} = l_1 \ldots l_{j-1} l_{j+1} \ldots l_k \wedge \frac{dF}{dp_i} \tag{13}$$

In view of (12), the detection condition for a stuck-at-0 fault which replaces any literal in term p_i with 0 is

$$(p_i \oplus 0) \wedge \frac{dF}{dp_i} = p_i \wedge \overline{p_1} \wedge \ldots \overline{p_{i-1}} \wedge \overline{p_{i+1}} \wedge \ldots \overline{p_m}$$

This defines the set U_i of unique true points for the term p_i.

In view of (13), the detection condition for a stuck-at-1 fault which replaces literal l_j of term p_i with 1 is

$$(l_j \oplus 1) \wedge \frac{dF}{dl_j} = \overline{l_j} l_1 \cdots l_{j-1} l_{j+1} \cdots l_k \wedge \frac{dF}{dp_i}$$

This defines the set $N_{i,j}$, since $\overline{l_j} l_1 \cdots l_{j-1} l_{j+1} \cdots l_k$ defines the set $D_{i,j}$ of true points for $p_{i,j}$.

The basic meaningful impact strategy happens to also detect the variable negation faults because, as we observed in Section 5.2, test cases that detect stuck-at faults will also detect variable negation faults.

6 Conclusions

Our reformulation of the detection condition for a fault in specification S where a subpredicate X is replaced with another predicate E as a conjunction of origination condition and propagation condition:

$$dS^{\mathrm{X}}_{\mathrm{E}} = (X \oplus E) \wedge \frac{dS}{dX}$$

allows us to elegantly prove that the fault hierarchy holds for general expressions, not just expressions in disjunctive normal form. It also allows us to extend the fault hierarchy to a wider range of fault classes.

This extended hierarchy, diagrammed in Figure 2, permits us to skip an explicit test for a fault from an "easier to detect" class in the hierarchy, provided that we detect a corresponding fault from a "harder to detect" class, thus improving the effectiveness of fault based testing. The work is applicable to faults involving Boolean variables, as well as those occurring in relational expressions.

We use the fault conditions and our analysis to explain many previous empirical observations. In particular, we prove that a test that detects a missing expression fault will also detect the corresponding logical operator reference fault where $\wedge$ is replaced by $\vee$ and vice versa, and a test set that detects both stuck-at-0 and stuck-at-1 faults for a clause will also detect the corresponding missing clause fault.

We prove that the basic meaningful impact strategy of Weyuker et. al is testing for stuck-at faults; it detects variable negation faults because tests that detect stuck-at faults also detect variable negation faults.

The analytical technique presented in this paper can be applied to other such problems. In particular, it can be used to compare the semantic sizes [17] of fault classes. For instance, the relationship between clause negation fault and expression negation fault implies that the former is smaller semantically. In this case, clause negation fault also represents a smaller syntactic change than the expression negation fault.

References

[1] P. E. Black, V. Okun, Y. Yesha, Mutation operators for specifications, in: 15^{th} IEEE International Conference on Automated Software Engineering (ASE2000), IEEE Computer Society, Grenoble, France, 2000, pp. 81–88.

[2] T. Y. Chen, M. F. Lau, Test case selection strategies based on boolean specifications, Software Testing, Verification and Reliability 11 (3) (2001) 165–180.

[3] J. J. Chilenski, S. P. Miller, Applicability of modified condition/decision coverage to software testing, Software Engineering Journal (1994) 193–200.

[4] L. A. Clarke, A. Podgurski, D. J. Richardson, S. J. Zeil, A formal evaluation of data flow path selection criteria, IEEE Transactions on Software Engineering 15 (11) (1989) 1318–1332.

[5] R. A. DeMillo, R. J. Lipton, F. G. Sayward, Hints on test data selection: Help for the practicing programmer, IEEE Computer 11 (4) (1978) 34–41.

[6] R. A. DeMillo, A. J. Offutt, Constraint-based automatic test data generation, IEEE Transactions on Software Engineering 17 (9) (1991) 900–910.

[7] K. A. Foster, Error sensitive test cases analysis (ESTCA), IEEE Transactions on Software Engineering 6 (3) (1980) 258–264.

[8] J. B. Goodenough, S. L. Gerhart, Toward a theory of test data selection, IEEE Transactions on Software Engineering 1 (2) (1975) 156–173.

[9] A. Gopal, T. Budd, Program testing by specification mutation, Tech. Rep. TR 83-17, University of Arizona (Nov. 1983).

[10] T. Goradia, Dynamic impact analysis: Analyzing error propagation in program executions, Ph.D. thesis, Dept. of Computer Science, New York University (1988).

[11] D. R. Kuhn, A technique for analyzing the effects of changes in formal specifications, The Computer Journal 35 (6) (1992) 574–578.

[12] D. R. Kuhn, Fault classes and error detection in specification based testing, ACM Transactions on Software Engineering Methodology 8 (4) (1999) 411–424.

[13] M. F. Lau, Y. T. Yu, On the relationships of faults for boolean specification based testing, in: 2001 Australian Software Engineering Conference, IEEE CS Press, 2001, pp. 21–28.

[14] N. G. Leveson, M. P. Heimdahl, H. Hildreth, J. Reese, Requirements specification for process control systems, IEEE Transactions on Software Engineering SE-20 (9) (1994) 684–707.

[15] H. D. Mills, On the statistical validation of computer programs, in: Software Productivity, Little, Brown, Boston, 1983, pp. 71–81, also Technical Report FSC 72-6015, IBM Federal Systems Division, 1972.

[16] L. J. Morell, A theory of error-based testing, Dissertation, Dept. of Computer Science, University of Maryland (Aug. 1984).

[17] A. J. Offutt, J. H. Hayes, A semantic model of program faults, in: International Symposium on Software Testing and Analysis, 1996, pp. 195–200.

[18] A. J. Offutt, J. M. Voas, Subsumption of condition coverage techniques by mutation testing, Tech. Rep. ISSE-TR-96-01, George Mason University (1996).

[19] J. Offutt, Y. Xiong, S. Liu, Criteria for generating specification-based tests, in: Proceedings of the Fifth IEEE Fifth International Conference on Engineering of Complex Computer Systems (ICECCS '99), IEEE Computer Society Press, Las Vegas, NV, 1999, pp. 119–131.

[20] I. Reed, Boolean difference calculus and fault finding, SIAM Journal Applied Mathematics 24 (1) (1973) 134–143.

[21] D. J. Richardson, M. C. Thompson, An analysis of test data selection criteria using the relay model of fault detection, IEEE Transactions on Software Engineering 19 (6) (1993) 533–553.

[22] K.-C. Tai, Theory of fault-based predicate testing for computer programs, IEEE Transactions on Software Engineering 22 (8) (1996) 552–562.

[23] T. Tsuchiya, T. Kikuno, On fault classes and error detection in specification based testing, ACM Transactions on Software Engineering Methodology 11 (1) (2002) 58–62.

[24] E. Weyuker, T. Goradia, A. Singh, Automatically generating test data from a boolean specification, IEEE Transactions on Software Engineering 20 (5) (1994) 353–363.

[25] S. J. Zeil, Perturbation techniques for detecting domain errors, IEEE Transactions on Software Engineering 15 (6) (1989) 737–746.

[26] H. Zhu, P. A. V. Hall, J. H. R. May, Software unit test coverage and adequacy, ACM Computing Surveys 29 (4) (1997) 366–427.