An Extended Fault Class Hierarchy for Specification-Based Testing

MAN F. LAU
Swinburne University of Technology
and
YUEN T. YU
City University of Hong Kong

Kuhn, followed by Tsuchiya and Kikuno, have developed a hierarchy of relationships among several common types of faults (such as variable and expression faults) for specification-based testing by studying the corresponding fault detection conditions. Their analytical results can help explain the relative effectiveness of various fault-based testing techniques previously proposed in the literature. This article extends and complements their studies by analyzing the relationships between variable and literal faults, and among literal, operator, term, and expression faults. Our analysis is more comprehensive and produces a richer set of findings that interpret previous empirical results, can be applied to the design and evaluation of test methods, and inform the way that test cases should be prioritized for earlier detection of faults. Although this work originated from the detection of faults related to specifications, our results are equally applicable to program-based predicate testing that involves logic expressions.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Theory, Verification

Additional Key Words and Phrases: Fault class analysis, software testing, specification-based testing, test case generation

The work described in this article was partially supported by a grant from the City University of Hong Kong (project number 7001233) and a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (RGC reference number CityU 1083/00E).
This article is a substantially revised and expanded version of the article "On the Relationships of Faults for Boolean Specification Based Testing," by M. F. Lau and Y. T. Yu, which appeared in *Proceedings of 2001 Australian Software Engineering Conference* [Lau and Yu 2001], © 2001 IEEE.
Authors' addresses: M. F. Lau, Faculty of Information and Communication Technologies, Swinburne University of Technology, Hawthorn 3122, Australia; email: elau@ict.swin.edu.au; Y. T. Yu (corresponding author), Department of Computer Science, City University of Hong Kong, Kowloon Tong, Hong Kong; email: csytyu@cityu.edu.hk.

1. INTRODUCTION

Software testing is undoubtedly a major means of software quality assurance, in spite of the existence of other complementary approaches such as inspection and formal verification. Since the aim of software testing is to demonstrate the existence of errors in software [Myers 1979], selecting test cases that can reveal software faults is of paramount importance.

In recent years, there has been an increasing interest and emphasis on specification-based testing [Chang et al. 1996; Dick and Faivre 1993; Gargantini and Riccobene 2003; Hierons 1997; Offutt et al. 2003; Paradkar et al. 1997; Stock and Carrington 1996] and the use of a fault-based approach to generate test cases from software specifications [Black et al. 2000a; Chen et al. 1999; DeMillo et al. 1978; Kuhn 1999; Offutt et al. 1996; Tai 1996; Tsuchiya and Kikuno 2002]. A common approach is first to hypothesize certain common types of faults that may be committed by the programmer, and then generate test cases that can detect these faults.

Various types of faults have been defined and studied in the literature [Foster 1984; Kuhn 1999; Richardson and Thompson 1993; Tai 1996; Tai and Su 1987; Tsuchiya and Kikuno 2002; Weyuker et al. 1994], including a variable reference fault (a variable replaced by another variable), a variable negation fault (a Boolean variable replaced by its negation), and an expression negation fault (a Boolean expression replaced by its negation). In particular, Kuhn [1999] discovered certain relationships among the conditions that detect these three types of faults. He found that, if the specification is expressed in a Boolean expression in disjunctive normal form, a test case that can detect a variable reference fault can also detect the variable negation fault, which in turn can detect the corresponding expression negation fault. This establishes a hierarchy that relates these types of faults, whose ordering in the hierarchy is consistent with the ordering of the effectiveness of various fault-based testing techniques for detecting these faults as found in previous empirical studies by Vouk et al. [1994] and Weyuker et al. [1994]. Thus, fault class hierarchy provides a useful theoretical framework to explain the empirical results on the effectiveness of fault-based testing techniques [Kuhn 1999].

Tsuchiya and Kikuno [2002] further analyzed the relationship between a variable reference fault and a missing condition fault. A missing condition fault is one which causes a condition in a specification to be omitted from the implementation. They discovered that test cases that can detect a missing condition fault may not be able to detect the corresponding variable reference fault, and vice versa. They also suggested that further work needs to be done on the study and analysis of fault class relationships as well as the effectiveness of fault-based testing strategies, in particular the strategy proposed by Chen and Lau [1997] for the detection of variable reference faults.

Hierons [2002] proposed a new approach for comparing test sets by considering the presence of a test hypothesis or a fault domain. He defined a comparator and provided a framework in which formal statements about the relative effectiveness of test sets and criteria can be made in the presence of known system properties. By using this approach, he rephrased Kuhn's [1999] results in terms of relations between test criteria that distinguish a specification from its variants formed by the introduction of a single fault of the respective classes. He also illustrated how his approach can assist in the incremental generation of tests.

Thus, the analytical results by Kuhn [1999], together with those by Tsuchiya and Kikuno [2002], have clarified the relations among missing condition faults, variable reference faults, variable negation faults, and expression negation faults. These relations have helped to compare the relative effectiveness of test sets [Kuhn 1999; Tsuchiya and Kikuno 2002] and assist in incremental test generation [Hierons 2002]. However, many other faults proposed in the literature, such as those involving logical operators [Black et al. 2000b; Tai 1996; Tai and Su 1987; Weyuker et al. 1994] and terms and literals [Chen and Lau 1997, 2001], have not been considered in these studies. This article extends and complements their work [Kuhn 1999; Tsuchiya and Kikuno 2002] by analyzing the relationships between variable and literal faults, and among literal, operator, term, and expression faults. We discover an extended fault class hierarchy in which a test case that detects a fault F will always detect the corresponding fault(s) at the parent position(s) of F in the hierarchy.

Our analysis is more comprehensive and produces a richer set of findings that have a wide range of applications. Specifically, our analysis not only helps explain previous experimental results, but also helps in designing and evaluating fault-based testing techniques, in prioritizing test cases to be executed so as to achieve earlier detection of more faults, and in studying the relationships between different types of mutation operators in the area of mutation analysis. Detailed discussions on these applications can be found in Section 5.

Section 2 of this article defines our notation, clarifies the terminologies, reviews previous work, and discusses fault classes in software systems. Section 3 analyzes the relationships involving variable faults. Section 4 proves an extended fault class hierarchy relating literal, term, operator, and expression faults. Section 5 discusses the usefulness of our extended fault class hierarchy by illustrating various applications of our results to software testing and analysis. Section 6 concludes the article.

2. FAULT CLASSES

2.1 Basic Notation and Terminology

Kuhn [1999], followed by Tsuchiya and Kikuno [2002], have studied the relationships of different fault classes related to specifications written in Boolean expressions in disjunctive normal form (DNF). We now present the notation and terminologies used in this article for such expressions.

Boolean expressions are those involving Boolean operators such as AND, OR, and NOT (denoted by "$\cdot$", "$+$", and "$^{-}$", respectively), and evaluate to either FALSE or TRUE (denoted by 0 and 1, respectively). The "$\cdot$" symbol will be omitted whenever it is clear from the context. A Boolean expression S in DNF can be written as $S = p_1 + \cdots + p_m$, where m is the number of terms and $p_i (i = 1, \ldots, m)$ denotes the ith term of S. Moreover, let $p_i = x_1^i \cdots x_{k_i}^i$, where k_i is the number of literals in p_i, and $x^i_j (j = 1, \ldots, k_i)$ denotes the jth literal in p_i. A *literal* is an occurrence of a variable. A variable may occur in a Boolean expression as a *positive* literal or a *negative* literal. For example, in the specification $S = ad + \bar{a}c + be$, the variable a occurs as a positive literal a in the first term $p_1 = ad$ and as a negative literal $\bar{a}$ in the second term $p_2 = \bar{a}c$.

A variable or a literal in a Boolean expression represents a *condition* that cannot be further decomposed into simpler Boolean expressions. A condition may be either a simple Boolean variable, such as "*Running*" (which indicates that the engine of an automobile is running) in a Cruise Control System [Atlee and Gannon 1993], or it may actually represent a relational expression such as "$\texttt{\#dom}(s) = \texttt{msize}$" [Hierons 1997] or "$switch = \texttt{up}$" [Chan et al. 1998], even though we denote it by a single letter. A *test case* for testing the implementation I against the specification S of n variables is a vector $\vec{t} = (t_1, \ldots, t_n)$, where $t_i \in \mathbb{B} = \{0, 1\}$. A test case $\vec{t}$ is said to distinguish S and I if $S(\vec{t}) \neq I(\vec{t})$.

2.2 Fault-Based Testing and Mutation Analysis

The input domain of a Boolean expression of n variables is $\mathbb{B}^n$, whose size grows exponentially with n. A test set that can reveal all possible faults has to satisfy the *multiple-condition coverage* (M-CC) criterion [Chilenski and Miller 1994; Myers 1979], which effectively requires all 2^n combinations of the conditions to be tested. This approach is infeasible when n is large, such as in many avionic systems [Chilenski and Miller 1994; Dupuy and Leveson 2000; Jones and Harrold 2003].

In hardware testing, a systematic approach is often adopted, whereby typical manufacturing faults (such as the so-called stuck-at faults) are hypothesized and test data are then derived to detect these faults. Although the problem of testing Boolean expressions is not dissimilar to hardware testing, software faults are more varied in nature [Foster 1984] and harder to predict than hardware faults [Kobayashi et al. 2002], since the former are results of human errors during software development [Kuhn 1999].

In software testing, the fault-based approach [Chen et al. 1999; Foster 1980, 1984; Tai 1996], which focuses on certain restricted classes of well-defined faults, is a widely studied approach for selecting test cases. Fault-based testing is especially effective when the faults model commonly committed mistakes. Execution of fault-based tests without failure guarantees the absence of the prespecified faults.

A commonly accepted technique for fault-based testing is *mutation analysis*, originally applied to source codes [DeMillo et al. 1978; Offutt et al. 1996] but recently also applied to specifications [Black et al. 2000a]. Mutation analysis typically considers faults that cause small changes to the correct version. It rests on two assumptions: (1) the *competent programmer hypothesis* [DeMillo et al. 1978], which states that an incorrect program written by a competent programmer will differ from a correct version by relatively simple faults, and (2) the *fault coupling effect* [Offutt et al. 1996], which hypothesizes that a test set which detects all simple faults in a program will be able to detect more complex faults.

For simplicity, here we outline its application to (Boolean) specifications only; its application to implementations is basically similar. Given a specification S, a *mutation operator* creates a number of *mutants* (representing possible faulty implementations) by making simple syntactic changes to S, one at a time. A mutant I_1 is said to be *killed* by a test case $\vec{t}$ if $S(\vec{t}) \neq I_1(\vec{t})$. An *equivalent mutant* I_2 is logically equivalent to the specification S (that is, $S \equiv I_2$) and hence cannot be killed by any test case. For example, inserting $\bar{a}$ to the second term of $S_0 = a + b$ results in an equivalent mutant $I_0 = a + \bar{a}b$ since $S_0 \equiv I_0$.

Recent empirical studies [Daran and Thévenod-Fosse 1996; Madeira et al. 2000] have shown that even simple syntactic faults can create erroneous program behavior as complex as that produced by real faults, lending support to the representativeness of errors due to mutations. In practice, when the nature and types of faults are not easily identified, mutation-style faults are commonly used as a basis for fault-based testing.

2.3 Definition of Fault Classes

Although extensive empirical studies to identify representative software faults are lacking, some case studies on real faults do exist [DeMillo et al. 1978; Daran and Thévenod-Fosse 1996; Dupuy and Leveson 2000]. In particular, the observations of real faults found in Daran and Thévenod-Fosse [1996], Dupuy and Leveson [2000] and Madeira et al. [2000] are consistent with the common intuition that typical programmer errors include path domain boundary faults due to incorrect control predicates, missing or extra conditions and paths, and the use of incorrect operators or operands [Myers 1979; Tai and Su 1987]. These errors translate to the omission, insertion, or incorrect reference of Boolean operands or operators, when restricted to simple faults relevant to Boolean expressions in DNF. Based on these considerations, we shall analyze fault classes that are described as follows. Our fault classes are similar to those hypothesized in Chen and Lau [2001], Chen et al. [1999], Foster [1984], Kuhn [1999], Paradkar et al. [1997], Tai [1996], Tai and Su [1987], Tsuchiya and Kikuno [2002], Weyuker et al. [1994], Yu and Lau [2002], and Yu et al. [2003], as will be reviewed in Section 2.4.

To define the fault classes, we express the specification in the form $S = p_1 + \cdots + p_m$, where m is the number of terms, $p_i = x_1^i \cdots x_{k_i}^i$ is the ith term of S, x_j^i is the jth literal in p_i, and k_i is the number of literals in p_i. A faulty implementation of S will be denoted by I_F^E, where F is the name of the fault class and E is an appropriate superscript that exactly identifies the fault.

(1) *Expression negation fault* (ENF): The entire Boolean expression S, or a subexpression of it (which is a disjunction of terms, $p_i + \cdots + p_j$), is implemented as its negation. That is, the implementation is $I_{ENF}^{S} = \overline{S}$, or $I_{ENF}^{p_i+\cdots+p_j} = p_1 + \cdots + p_{i-1} + \overline{p_i + \cdots + p_j} + p_{j+1} + \cdots + p_m$, where $1 \leq i < j \leq m$. For example, if $S = ad + \bar{a}c + be$, then $I_{ENF}^{S} = \overline{ad + \bar{a}c + be}$, and $I_{ENF}^{p_1+p_2} = \overline{ad + \bar{a}c} + be$.

(2) *Term negation fault* (TNF): A term p_i is implemented as its negation. That is, the implementation is $I_{TNF}^{p_i} = p_1 + \cdots + p_{i-1} + \overline{p_i} + p_{i+1} + \cdots + p_m$,

where $1 \leq i \leq m$ and $m > 1$. For example, if $S = ad + \bar{a}c + be$, then $I^{p_1}_{TNF} = \overline{ad} + \bar{a}c + be$. Note that, if the expression consists of one term only (when $m = 1$), negating the term will be regarded as an ENF.

(3) *Term omission fault* (TOF): A term p_i is omitted in the implementation. That is, the implementation is $I^{p_i}_{TOF} = p_1 + \cdots + p_{i-1} + p_{i+1} + \cdots + p_m$, where $1 \leq i \leq m$ and $m > 1$. For example, if $S = ad + \bar{a}c + be$, then $I^{p_1}_{TOF} = \bar{a}c + be$.

(4) *Operator reference fault* (ORF): There are two fault subclasses, as follows:
   (a) *Disjunctive operator reference fault* (ORF[+]): A binary Boolean operator "+" immediately following a term p_i is implemented as "·". That is, the implementation is $I^{p_i}_{ORF[+]} = p_1 + \cdots + p_{i-1} + p_i \cdot p_{i+1} + p_{i+2} + \cdots + p_m$, where $1 \leq i < m$. For example, if $S = ad + \bar{a}c + be$, then $I^{p_1}_{ORF[+]} = ad \cdot \bar{a}c + be$.
   (b) *Conjunctive operator reference fault* (ORF[·]): A binary Boolean operator "·" immediately following a literal x^i_j is implemented as "+". That is, the implementation is $I^{x^i_j}_{ORF[\cdot]} = p_1 + \cdots + p_{i-1} + x^i_1 \cdots x^i_j + x^i_{j+1} \cdots x^i_{k_i} + p_{i+1} + \cdots + p_m$, where $1 \leq i \leq m$ and $1 \leq j < k_i$. For example, if $S = ad + \bar{a}c + be$, then $I^{x^1_1}_{ORF[\cdot]} = a + d + \bar{a}c + be$.

(5) *Literal negation fault* (LNF): A literal x^i_j is implemented as its negation. That is, the implementation is $I^{x^i_j}_{LNF} = p_1 + \cdots + p_{i-1} + x^i_1 \cdots x^i_{j-1} \cdot \bar{x}^i_j \cdot x^i_{j+1} \cdots x^i_{k_i} + p_{i+1} + \cdots + p_m$, where $1 \leq i \leq m$, $1 \leq j \leq k_i$ and $k_i > 1$. For example, if $S = ad + \bar{a}c + be$, then $I^{x^1_1}_{LNF} = \bar{a}d + \bar{a}c + be$. Note that if the term p_i consists of a single literal only (when $k_i = 1$), negating the literal will be regarded as a TNF.

(6) *Literal omission fault* (LOF): A literal x^i_j is omitted. That is, the implementation is $I^{x^i_j}_{LOF} = p_1 + \cdots + p_{i-1} + x^i_1 \cdots x^i_{j-1} \cdot x^i_{j+1} \cdots x^i_{k_i} + p_{i+1} + \cdots + p_m$, where $1 \leq i \leq m$. $1 \leq j \leq k_i$ and $k_i > 1$. For example, if $S = ad + \bar{a}c + be$, then $I^{x^1_1}_{LOF} = d + \bar{a}c + be$. Note that, if the term p_i consists of a single literal only (when $k_i = 1$), omitting the literal will be regarded as a TOF.

(7) *Literal insertion fault* (LIF): A literal x is inserted into a term p_i of the Boolean expression. For LIF, insertion of the following is not considered:
   (a) a literal that is already present in the term, because the resulting implementation is equivalent to the original expression; and
   (b) the negation of a literal that is already present in the term, because the result is effectively the same as that due to TOF with that particular term omitted.

   That is, the implementation is $I^{p_i,x}_{LIF} = p_1 + \cdots + p_{i-1} + p_i \cdot x + p_{i+1} + \cdots + p_m$, where $1 \leq i \leq m$ and $x \notin \{x^i_1, \ldots, x^i_{k_i}, \bar{x}^i_1, \ldots, \bar{x}^i_{k_i}\}$. For example, if $S = ad + \bar{a}c + be$, then $I^{p_1,b}_{LIF} = adb + \bar{a}c + be$.

(8) *Literal reference fault* (LRF): A literal x^i_j is replaced by another literal x. For LRF, replacement of a literal by the following is not considered:
   (a) the negation of the same literal, because the result is effectively the same as that due to LNF (if there is more than one literal in the term), TNF (if the term consists of one literal only), or ENF (if the entire expression consists of one literal only);

(b) another literal that is already present in the term, because the result is effectively the same as that due to LOF; or
(c) the negation of another literal that is already present in the term, because the result is effectively the same as that due to TOF.

That is, the implementation is $I_{LRF}^{x_j^i,x} = p_1 + \cdots + p_{i-1} + x_1^i \cdots x_{j-1}^i \cdot x \cdot x_{j+1}^i \cdots x_{k_i}^i + p_{i+1} + \cdots + p_m$, where $1 \leq i \leq m$, $1 \leq j \leq k_i$ and $x \notin \{x_1^i, \ldots, x_{k_i}^i, \bar{x}_1^i, \ldots, \bar{x}_{k_i}^i\}$. For example, if $S = ad + \bar{a}c + be$, then $I_{LRF}^{x_1^1,b} = bd + \bar{a}c + be$.

2.4 Previous Work on Fault Classes

In hardware logic circuit testing where Boolean expressions are extensively used, stuck-at faults are commonly considered a realistic model of manufacturing faults. There are two stuck-at faults: *stuck-at-*`0` *fault* (STF[`0`]) and *stuck-at-*`1` *fault* (STF[`1`]). Obviously, if the expression consists of more than one term, then a literal stucked at `0` will cause the term containing it to vanish and STF[`0`] will become a TOF. On the other hand, if the expression contains a term with more than one literal, then a literal in the term stuck at `1` will have the same effect as if the literal has vanished, in which case STF[`1`] becomes a LOF.

Foster [1984] proposed some rules and procedures for deriving test data from logic expressions. He noted that the resulting test data appear to be sensitive to nonequivalent "errors" such as (a) omission or misplacement of parentheses, (b) omission or misplacement of `NOT` operators, (c) interchange of `AND`/`OR` operators, and (d) insertion of superfluous "variables"[1] and operators. Even though Foster used the term "variable,"[1] apparently he meant literal faults since his rules apply to "each appearance of each variable." His "errors" correspond to our operator faults and literal faults, except for parenthesis "errors."

Tai and Su [1987] suggested that many faults in software do not necessarily correspond to faults in logic circuits. Motivated by Foster's paper [1984] and following up the work in Tai and Su [1987], Paradkar et al. [1997], Tai [1996], and Vouk et al. [1994] noted that an incorrect Boolean expression may contain faults such as (a) an *operator replacement* fault, which replaces one Boolean operator with another, (b) a "*variable negation* fault," which refers to the missing of a negation operator for a literal [Paradkar et al. 1997, page 138] (and hence is actually a literal negation fault), (c) an "expression negation fault," which refers to the missing of a negation operator for an "expression,"[2] and (d) an *incorrect parenthesis*, *incorrect Boolean variable*, and *extra/missing* `NOT` *operator* fault. Except for the incorrect parenthesis fault, the fault types which Tai and others [Tai 1996; Tai and Su 1987; Paradkar et al. 1997; Vouk et al. 1994] considered are essentially the same as the corresponding fault classes described in Section 2.3.

[1]Since different interpretations of the fault classes have been used in previous work, we will keep to our own terminology defined in Section 2.3 as far as possible to avoid confusion. Where appropriate, we put the name of a fault class inside a pair of quotes to signify that it is being used with a different meaning from our terminology.
[2]The meaning of an "expression" was not precisely elaborated in their article.

Weyuker et al. [1994] proposed several testing strategies for Boolean expressions and evaluated them empirically in terms of their ability to detect the following classes of faults: (a) a "variable negation fault," which replaces one occurrence of a variable by its negation [Weyuker et al. 1994, page 360] (and hence is actually a literal negation fault), (b) an "expression negation fault," which replaces an "expression" by its negation, (c) a "variable reference fault," which replaces one occurrence of a variable by another, or by a constant, (d) an operator reference fault, which replaces one Boolean operator with another, and (e) an associative shift fault, which changes the associativity of terms. Again, the faults which they considered are the same as those studied in this article except for the associative shift fault.

Kuhn [1999] studied the detection condition relationships of the following fault classes:

(a) the *variable negation fault* (VNF), which replaces *all occurrences* of a variable with its negation;
(b) the *variable reference fault* (VRF), which replaces *all occurrences* of a variable with another;
(c) the "expression negation fault"[1] ("ENF"),[3] which replaces an "expression" or a group of "expressions" with its negation. Kuhn interprets an "expression" as a *clause*, which is a conjunction of variables. For example, if $S = ad + \bar{a}c + be$, the faulty implementation $S^{be}_{\overline{be}} = ad + \bar{a}c + \overline{be}$ (respectively $S^{ad+\bar{a}c}_{\overline{ad+\bar{a}c}} = \overline{ad + \bar{a}c} + be$) is formed by negating the third clause (respectively the first two clauses). In our terminology, a *clause* is called a *term*, and we say that $S^{be}_{\overline{be}}$ is caused by a TNF, while $S^{ad+\bar{a}c}_{\overline{ad+\bar{a}c}}$ is caused by an ENF.

We follow Kuhn's definition of variable faults (VNF and VRF) in this article. In Section 3, we shall analyze the relations between variable faults and the corresponding literal faults, and also discuss in more detail the work of Kuhn [1999] in relation to our analysis. In the rest of this section, we shall discuss the importance of Boolean expressions and fault classes to the testing of software systems.

2.5 Boolean Expressions and Fault Classes in Software Systems

The specification of a software system generally includes functional requirements such as the constraints on its operations [Dick and Faivre 1993; Hierons 1997], the conditions and events that trigger state transitions [Atlee and Buckley 1996; Offutt et al. 2003], as well as nonfunctional requirements such as safety properties [Atlee and Gannon 1993; Ammann et al. 2001]. In a variety of formalisms, these requirements are written as Boolean expressions or their equivalent forms.

For instance, the SCR specification language, a frequently cited formalism for capturing event-driven system requirements, has been used to specify many practical systems in industrial environments, including avionic and nuclear

[3]The term "ENF" is put inside quotes to indicate that it is being used to mean something different from our terminology described in Section 2.3.

Table I. SCR and Boolean Specification of Part of the Cruise Control System

Current mode	*Ignited*	*Running*	*Toofast*	*Brake*	*Activate*	*Deactivate*	*Resume*	New mode
`Cruise`	t	t	f	@T	–	–	–	`Override`
	t	t	f	–	–	@T	–	

Conditions for the transition from `Cruise` mode to `Override` mode can be written as follows [Offutt et al. 2003]: $S_{\mathtt{C}\mapsto\mathtt{O}} = Ignited \cdot Running \cdot \overline{Toofast} \cdot \overline{Brake} \cdot Brake' + Ignited \cdot Running \cdot \overline{Toofast} \cdot \overline{Deactivate} \cdot Deactivate'$.

Table II. An AND/OR Table in the TCAS II Specification [Heimdahl and Leveson 1996]

		OR	
A	Own-Air-Status$_{\text{v}-36}$ = On-Ground	T	F
N	Traffic-Display-Permitted$_{\text{v}-39}$	·	F
D	Mode-Selector$_{\text{v}-34}$ = Standby	T	·

power plants control systems [Heitmeyer and Bharadwaj 2000]. It uses the *mode transition table* to specify the *conditions* and *events* that trigger the transitions of the system between modes. Table I shows part of the mode transition table of a Cruise Control System [Atlee and Buckley 1996].

Many researchers [Ammann et al. 2001; Atlee and Buckley 1996; Atlee and Gannon 1993; Gargantini and Heitmeyer 1999] have formalized SCR requirements with logic-model semantics so that they can be analyzed by model checkers such as the Symbolic Model Verifier (SMV). For example, Gargantini and Riccobene [2003] formalized the system's safety properties or functional requirements in Boolean expressions in DNF, based on which animation goals are defined in a way analogous to the derivation of test scenarios. Ammann et al. [2001] formalize the notion of dangerous actions into logic expressions, and develop coverage criteria for generating new test sets or assessing existing test sets by means of mutation analysis.

Using the mode transition table of the Cruise Control System [Atlee and Buckley 1996] as an example, Offutt et al. [2003] have shown how to convert a state-based specification into Boolean expressions, based on which test cases can be generated. In Table I an entry @T under the column of an event such as *Brake* represents a *trigger* of it from `FALSE` to `TRUE`. By expanding such a trigger to the form $\overline{Brake} \cdot Brake'$ (where a prime attached to a condition denotes its posttransition value), the conditions for the transition from `Cruise` mode to `Override` mode can be written as the Boolean expression $S_{\mathtt{C}\mapsto\mathtt{O}}$ in Table I [Offutt et al. 2003].

Developed by Leveson et al. [1994], the Requirements State Machine Language (RSML) is another well-known formalism for specifying safety critical systems. It uses a modified Statechart notation in which the conditions for state transitions are represented in the form of AND/OR tables, which are simply a readable version of Boolean expressions in DNF (see Table II for a sample). Experiments have been reported that used different strategies to generate test cases from Boolean expressions in the specification of an aircraft collision avoidance system called *TCAS II* [Kobayashi et al. 2002; Tai 1996; Weyuker et al. 1994; Yu and Lau 2002; Yu et al. 2003].

From model-based specifications such as Z and VDM, Dick and Faivre [1993] and Hierons [1997] generated test cases by using the partition testing approach [Chen and Yu 1994, 1996; Duran and Ntafos 1984]. They transformed the constraints on the individual operations into Boolean expressions in DNF, which represent the boundaries that partition the input domain into disjoint subdomains.

Many other well-known specification methods also rely heavily on such artifacts as cause-effect graphs [Myers 1979; Paradkar et al. 1997], fault-trees [Leveson et al. 1991] and predicates in ADL (Assertion Definition Language) [Chang et al. 1996] that are actually equivalent forms of Boolean expressions, and from these artifacts test cases can be automatically generated. Finally, most program control statements such as if-statements and loop predicates contain Boolean expressions. For critical real-time applications, over half of the executable statements may involve complex Boolean expressions [Chilenski and Miller 1994], and it is extremely important to ensure that these statements are adequately tested. Reports abound in recent years on the studies of criteria for generating test cases automatically from Boolean expressions found in either specifications or implementations of real-life software systems [Chang et al. 1996; Jones and Harrold 2003; Paradkar et al. 1997; Tai 1996].

Even though the fault classes in Section 2.3 are defined by small syntactic changes to the original Boolean expressions, it is not difficult to relate the resulting mutants to the possible implementations which contain real faults due to typical programmer errors. For example, faults of omission are known to be very common [Daran and Thévenod-Fosse 1996; Myers 1979], constituting approximately half of the bug reports posted on Usenet [Kuhn 1999]. The fault classes LOF and TOF are designed to model the *missing condition fault*, which causes the omission of condition(s) in a predicate [Black et al. 2000b; Tai and Su 1987]. Another example is a recent realistic case reported by Dupuy and Leveson [2000], who examined an attitude control software for the HETE-2 (High Energy Transient Explorer) scientific satellite and uncovered an important operator reference fault (ORF) which replaced an `AND` operator by an `OR`.

Finally, the hypothesized fault classes may serve as mutation operators, as has been done in Ammann and Black [2001], Black et al. [2000b], and Gargantini and Riccobene [2001]. Mutation analysis is useful not only for generating test cases, but also for assessing the effectiveness of test sets using a metric known as the *mutation score*, which is the proportion of nonequivalent mutants killed by the test set created [Ammann and Black 2001; Gargantini and Riccobene 2001; Offutt et al. 1996; Weyuker et al. 1994].

3. ANALYSIS OF FAULT CLASSES RELATED TO VARIABLE FAULTS

3.1 Relation Between Variable and Expression Faults

Kuhn [1999] studied the relationships between the following fault classes: variable negation fault (VNF), variable reference fault (VRF) and "expression

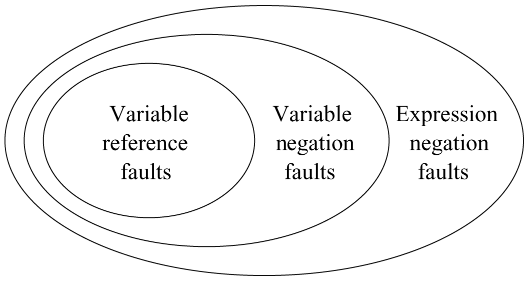

Fig. 1. Detection condition relationships among variable and expression faults [Kuhn 1999].

negation fault" ("ENF"). He noted that, for a Boolean specification S, the condition under which a VNF for variable v will be detected is given by $S_{VNF} = S \oplus S^{v}_{\bar{v}}$, where $\oplus$ denotes the exclusive-or operator, and $S^{v}_{\bar{v}}$ denotes the expression S with *all occurrences* of the variable v replaced by its negation $\bar{v}$. Here we have used the notation defined by Kuhn [1999] as follows: for a Boolean expression P, denote by P^{x}_{e} the expression P with *all occurrences* of variable x replaced by expression e.

Similarly, the detection conditions for VRF and "ENF" are, respectively, given by $S_{VRF} = S \oplus S^{v}_{w}$ (when v is replaced by w) and $S_{\text{“}ENF\text{”}} = S \oplus S^{C_i}_{\overline{C_i}}$, where C_i denotes a clause or a group of clauses in S. Kuhn [1999] stated two theorems which essentially mean that $S_{VRF} \Rightarrow S_{VNF} \Rightarrow S_{\text{“}ENF\text{”}}$. This relationship, depicted in Figure 1, can be expressed as follows:

(1) $[S_{VRF} \Rightarrow S_{VNF}]$. If the variable being replaced in S_{VRF} is the same one being negated in S_{VNF}, any test that detects VRF will also detect VNF. For example, $S \oplus S^{a}_{b} \Rightarrow S \oplus S^{a}_{\bar{a}}$.
(2) $[S_{VNF} \Rightarrow S_{\text{“}ENF\text{”}}]$. If the "expressions" (clauses) containing the variable being negated in S_{VNF} are negated in $S_{\text{“}ENF\text{”}}$, any test that detects VNF will also detect "ENF". For example, $S \oplus S^{a}_{\bar{a}} \Rightarrow S \oplus S^{ad+\bar{a}c}_{\overline{ad+\bar{a}c}}$.

Kuhn [1999] further postulated that VRFs can be subdivided into faults which result in a missing condition (when a variable is replaced by another variable in the same term) and faults which result in an incorrect condition (when a variable is replaced by another which is not in the same term).[4] Subsequently, Tsuchiya and Kikuno [2002] complemented Kuhn's result by clarifying the relationship between missing conditions and VRFs. They showed that, although revealing a missing condition will always detect the corresponding VNF, there is no similar relationship with the corresponding VRF. That is, a test that can detect any missing conditions may not detect some VRF, and vice versa.

3.2 Distinction Between Variable and Literal Faults

The variable reference (respectively negation) fault which Kuhn [1999] studied is a fault which causes *every* occurrence of a variable in the Boolean specification

[4]Note that in our definition of the corresponding literal reference fault (LRF) in Section 2.3, we have explicitly considered literal omission fault (LOF) as a separate case to render the relationship more obvious.

to be replaced by a different variable (respectively by the negation of the same variable). In contrast, empirical studies such as those in Kobayashi et al. [2002] and Weyuker et al. [1994] have considered faults which replace only *one* occurrence of a variable, even though these faults are referred to as *variable (reference or negation) faults*. (See Section 2.4 for a detailed discussion of the definition of fault classes in previous work.)

To emphasize the distinction between these two situations, we have adopted Kuhn's [1999] definition where a *variable fault* replaces *every* occurrence of the variable, and have defined in Section 2.3 a *literal fault* as one which replaces *exactly one occurrence* of a variable. We contend that a theory that attempts to explain previous empirical results (such as those in Kobayashi et al. [2002] and Weyuker et al. [1994]) should also incorporate literal faults that were actually considered in these experiments.

A literal fault changes only one literal of the specification, whereas a variable fault generally changes more than one literal. Therefore, a literal fault results in an implementation which is syntactically closer to the correct implementation than would have been caused by a variable fault. In this sense, a literal fault may be intuitively more fundamental, more elementary, as well as easier to escape the programmer's attention than a variable fault.

The different nature of variable and literal faults will give rise to different possible faulty implementations, as can be shown in the next example.

Example 3.1. Consider the specification $S = ad + \bar{a}c + be$, where the variable a occurs twice: once in the first term as a positive literal, and once in the second term as a negative literal. A variable reference fault (VRF) which replaces variable a by b results in the implementation $S^a_b = bd + \bar{b}c + be$. In contrast, there are two literal reference faults (LRFs) which replace a by b, resulting in the implementations I_1 and I_2, respectively, where (i) $I_1 = bd + \bar{a}c + be$, when the *first* occurrence of a is replaced by b, and (ii) $I_2 = ad + \bar{b}c + be$, when the *second* occurrence of a is replaced by b. Notice that S^a_b is equivalent to neither I_1 nor I_2. Similarly, a variable negation fault (VNF) that replaces a by $\bar{a}$ results in the implementation $S^a_{\bar{a}} = \bar{a}d + \bar{\bar{a}}c + be$, which is different from the versions caused by the two literal negation faults (LNFs) that replace a by $\bar{a}$, namely, $I_3 = \bar{a}d + \bar{a}c + be$ and $I_4 = ad + \bar{\bar{a}}c + be$. Again, $S^a_{\bar{a}}$ is equivalent to neither I_3 nor I_4.

In practice, the class of literal faults may be used to model faults related to a single instance of the infected variable, while variable faults may be used to model faults such as those appearing within a function or a macro that will potentially affect its return value every time it is called.

Example 3.2. Consider again the transition of the Cruise Control System from `Cruise` to `Override` mode as specified in Table I. The condition for this transition can be written as the Boolean expression $S_{\mathtt{C}\mapsto\mathtt{O}}$, where $S_{\mathtt{C}\mapsto\mathtt{O}} = \mathit{Ignited} \cdot \mathit{Running} \cdot \overline{\mathit{Toofast}} \cdot \overline{\mathit{Brake}} \cdot \mathit{Brake}' + \mathit{Ignited} \cdot \mathit{Running} \cdot \overline{\mathit{Toofast}} \cdot \overline{\mathit{Deactivate}} \cdot \mathit{Deactivate}'$ (also see Table I). A correct pseudocode segment for implementing this transition could be written as shown in Figure 2. If the Boolean operator `NOT` in `Line 2` is incorrectly omitted from the code in Figure 2, then this

```
Line 1:   IF   present_mode = Cruise
Line 2:   THEN IF (   (ignited AND running AND (NOT toofast)
Line 3:                AND brake_is_triggered_on)
Line 4:            OR (ignited AND running AND (NOT toofast)
Line 5:                AND deactivate_is_triggered_on))
Line 6:        THEN transit_to_Override_mode ENDIF
Line 7:   ENDIF
```

Fig. 2. A correct pseudocode segment for the mode transition in Table I.

```
Line 1:   DEFINE MACRO precond AS (ignited AND running AND (NOT toofast))
Line 2:   IF   present_mode = Cruise
Line 3:   THEN IF   (    (precond AND brake_is_triggered_on)
Line 4:                OR (precond AND deactivate_is_triggered_on))
Line 5:        THEN transit_to_Override_mode ENDIF
Line 6:   ENDIF
```

Fig. 3. An alternative correct pseudocode segment for the mode transition in Table I.

fault is equivalent to a LNF in $S_{\mathtt{C}\mapsto\mathtt{O}}$, whereby *the first occurrence* of the variable *Toofast* is unnegated. Alternatively, another possible correct code segment implementation of the same transition is shown in Figure 3, where `precond` is a macro that codes the common precondition for the two events to trigger the transition. If the Boolean operator `NOT` in `Line 1` is incorrectly omitted from the code in Figure 3, then this fault is equivalent to a VNF in $S_{\mathtt{C}\mapsto\mathtt{O}}$, whereby *every occurrence* of the variable *Toofast* is unnegated.

Kuhn's [1997] fault class hierarchy (Figure 1) shows only the relations among variable and expression faults. Literal faults, as well as many other faults (such as the operator reference fault [Kobayashi et al. 2002; Weyuker et al. 1994] and term omission fault [Chen and Lau 1997, 2001]), have not been considered in Kuhn's theory.

The above observations naturally lead to the following open questions that constitute the central theme of this article:

(1) If a test case detects a variable fault, does it also always detect the corresponding literal fault, and vice versa?
(2) Is there a fault class hierarchy, similar to Kuhn's, relating the detection conditions for literal faults?
(3) Are the detection conditions of literal faults related in any way to those of other faults (such as operator reference fault) that have been hypothesized in the literature?

We will address the first question in the rest of this section, and the second and third questions in the next section, where our major results will be succinctly summarized in the form of an extended fault class hierarchy (Figure 5). We will illustrate, in Section 5, how our extended theory of fault classes may be useful in many application areas in software testing and analysis.

3.3 Relation Between Variable and Literal Faults

We now investigate the detection condition relationship (a) between LRF and VRF, and (b) between LNF and VNF. Two questions are considered: (1) Will a test case that detects a variable (reference or negation) fault always detect the corresponding literal (reference or negation) faults? (2) Will a test case that detects a literal (reference or negation) fault always detect the corresponding variable (reference or negation) fault? Surprisingly, we find that the answers to both of the two questions are negative, as illustrated from the following examples.

Example 3.3. (VRF vs. LRF) Consider the Boolean specification $S = \bar{a}b + ac$ and the implementations $S_c^a = \bar{c}b + cc$, where (every occurrence of) the variable a has been replaced by c, and $I_5 = \bar{c}b + ac$, where the first occurrence of a has been replaced by c.

Here, the test case 001 ($a = 0$, $b = 0$, $c = 1$) distinguishes between S and S_c^a but not between S and I_5, since $S_c^a(001) = 1 \neq S(001) = 0 = I_5\,(001)$. Thus, this test case detects a VRF but cannot detect a corresponding LRF. Moreover, the test case 011 ($a = 0$, $b = 1$, $c = 1$) can distinguish between S and I_5 but not between S and S_c^a, since $I_5(011) = 0 \neq S(011) = 1 = S_c^a(011)$. Thus, this test case detects a LRF but cannot detect the corresponding VRF.

Example 3.4. (VNF vs. LNF) Consider the Boolean specification $S = ab + \bar{a}c$ and the implementations $S_{\bar{a}}^a = \bar{a}b + \bar{\bar{a}}c$, where (every occurrence of) the variable a has been negated, and $I_6 = \bar{a}b + \bar{a}c$, where the first occurrence of a has been negated.

Here, the test case 101 ($a = 1$, $b = 0$, $c = 1$) distinguishes between S and $S_{\bar{a}}^a$ but not between S and I_6, since $S_{\bar{a}}^a(101) = 1 \neq S(101) = 0 = I_6(101)$. Thus, this test case detects a VNF but cannot detect a corresponding LNF. Moreover, the test case 111 ($a = 1$, $b = 1$, $c = 1$) can distinguish between S and I_6 but not between S and $S_{\bar{a}}^a$, since $I_6(111) = 0 \neq S(111) = 1 = S_{\bar{a}}^a(111)$. Thus, this test case detects a LNF but cannot detect the corresponding VNF.

Thus, the detection conditions for variable faults do not bear simple and direct relationship with those for literal faults. We therefore proceed to study the relationships among literal and other faults.

4. RELATIONSHIPS AMONG FAULT CLASSES

In this section, we shall develop the relationships among all fault classes defined in Section 2.3. We first prove a hierarchy similar to that of Kuhn [1999], and then extend it to other fault classes.

4.1 A Hierarchy of Fault Classes

For any two fault classes F_1 and F_2, we write $F_1 \rightarrow F_2$ to mean that, if a test case $\vec{t}$ detects a fault that belongs to the fault class F_1, then $\vec{t}$ will also detect a corresponding fault that belongs to the fault class F_2. We say that F_1 is *stronger* than F_2 and F_2 is *weaker* than F_1 whenever $F_1 \rightarrow F_2$. Using this notation, Kuhn's [1997] hierarchy in Figure 1 can be written as VRF $\rightarrow$ VNF $\rightarrow$ "ENF".

$$\begin{tikzcd}
\text{ENF} \\
\text{TNF} \arrow[u] \\
\text{LNF} \arrow[u] \\
\text{LRF} \arrow[u]
\end{tikzcd}$$

Fig. 4. A hierarchy of fault classes.

We find that a similar hierarchy exists among literal, term and expression faults, namely, LRF → LNF → TNF → ENF, as shown in Figure 4. We now prove the relationships represented by this hierarchy step by step, with a theorem corresponding to each arrow in Figure 4.

THEOREM 4.1 (LRF → LNF).[5] *If a test case can distinguish S and $I_{LRF}^{x_j^i,x}$ for some i and j, then it can also distinguish S and $I_{LNF}^{x_j^i}$.*

PROOF. Let $S = p_1+\cdots+p_m$ and $I_{LRF}^{x_j^i,x} = p_1+\cdots+x_1^i\cdots x_{j-1}^i\cdot x\cdot x_{j+1}^i\cdots x_{k_i}^i+\cdots+p_m$. Let $\vec{t}$ be a test case that can distinguish S and $I_{LRF}^{x_j^i,x}$ for some i and j. Then $\vec{t}$ must satisfy the following conditions: (1) every term other than p_i evaluates to 0, (2) every literal in p_i other than x_j^i evaluates to 1, and (3) the literals x_j^i and x evaluate to different truth values. Under such circumstances, $S(\vec{t}) = x_j^i(\vec{t})$ and $I_{LNF}^{x_j^i}(\vec{t}) = \bar{x}_j^i(\vec{t})$. Hence, $S(\vec{t})$ and $I_{LNF}^{x_j^i}(\vec{t})$ are different. This completes the proof. □

THEOREM 4.2 (LNF → TNF). *If a test case can distinguish S and $I_{LNF}^{x_j^i}$ for some i and j, then it can also distinguish S and $I_{TNF}^{p_i}$.*

PROOF. Let $S = p_1+\cdots+p_m$ and $I_{LNF}^{x_j^i} = p_1+\cdots+x_1^i\cdots\bar{x}_j^i\cdots x_{k_i}^i+\cdots+p_m$. Let $\vec{t}$ be a test case that can distinguish S and $I_{LNF}^{x_j^i}$ for some i and j. Then $\vec{t}$ must satisfy the following conditions: (1) every term other than p_i evaluates to 0, and (2) every literal other than x_j^i evaluates to 1. Under such circumstances, $I_{TNF}^{p_i}(\vec{t}) = \bar{p}_i(\vec{t})$, which is different from $S(\vec{t}) = p_i(\vec{t})$. This completes the proof. □

THEOREM 4.3 (TNF → ENF). *If a test case can distinguish S and $I_{TNF}^{p_i}$ for some i, then it can also distinguish S and $I_{ENF}^{E(p_i)}$ where $E(p_i)$ is a disjunction of k ($k > 1$) terms in S containing p_i.*

PROOF. Let $S = p_1+\cdots+p_m$. For $I_{TNF}^{p_i}$, the term p_i in S is being negated. For $I_{ENF}^{E(p_i)}$, the expression $E(p_i)$ is being negated where $E(p_i)$ is a disjunction of k ($k > 1$) terms in S and p_i is one of the terms in this disjunction. We can

[5]When Kuhn [1999] stated the theorem involving $S_{VRF} \Rightarrow S_{VNF}$, his proof also included the case when the variable under consideration occurs only once in the Boolean expression. Hence Theorem 4.1 can also be proved in a different way by using the argument in Kuhn's article.

$$\begin{tikzcd}
 & \text{ENF} & \\
 & \text{TNF} \arrow[u] & \\
\text{ORF[+]} \arrow[uur] & \text{LNF} \arrow[u] & \text{ORF[.]} \arrow[ul] \\
\text{TOF} \arrow[u] \arrow[ur] & \text{LRF} \arrow[u] & \text{LOF} \arrow[ul] \arrow[u] \\
\text{LIF} \arrow[u] \arrow[ur] & &
\end{tikzcd}$$

Fig. 5. An extended hierarchy of fault classes.

rearrange the terms in S such that those terms in $E(p_i)$ becomes the first k terms in S and the original p_i becomes the first term p'_1 in S. In other words, $S = p'_1 + \cdots + p'_m$, $E(p_i) = p'_1 + \cdots + p'_k$, and $p'_1 = p_i$.

Let $\vec{t}$ be a test case that can distinguish S and $I_{TNF}^{p'_1} = \overline{p'_1} + p'_2 + \cdots + p'_m$. Then $\vec{t}$ must satisfy the condition that every term other than p'_1 evaluates to 0. Under such circumstances, $I_{ENF}^{E(p_i)}(\vec{t}) = \overline{p'_1(\vec{t})}$. This value is different from $S(\vec{t}) = p'_1(\vec{t})$. This completes the proof. □

4.2 An Extended Hierarchy of Fault Classes

Next, we extend the fault hierarchy to include other fault classes: LIF, LOF, TOF, and ORF. The extended fault hierarchy can be neatly summarized in Figure 5. We now formally state and prove the relations between different pairs of fault classes in the hierarchy one by one.

THEOREM 4.4 (LIF → LRF). *If a test case can distinguish S and $I_{LIF}^{p_i,x}$ for some i, then it can also distinguish S and $I_{LRF}^{x_j^i,x}$ for every $j = 1, \ldots, k_i$, where k_i is the number of literals in p_i.*

PROOF. Let $S = p_1 + \cdots + p_m$ and $I_{LIF}^{p_i,x} = p_1 + \cdots + p_i \cdot x + \cdots + p_m$. Let $\vec{t}$ be a test case that can distinguish S and $I_{LIF}^{p_i,x}$ for some i. Then $\vec{t}$ must satisfy the following conditions: (1) every term other than p_i evaluates to 0, (2) the term p_i evaluates to 1, and (3) the literal x evaluates to 0. Under such circumstances, $S(\vec{t}) = 1$ and $I_{LRF}^{x_j^i,x}(\vec{t}) = p_1 + \cdots + x_1^i \cdots x_{j-1}^i \cdot x \cdot x_{j+1}^i \cdots x_{k_i}^i + \cdots + p_m = 0$ for every $j = 1, \ldots, k_i$. Hence, $S(\vec{t})$ and $I_{LRF}^{x_j^i,x}(\vec{t})$ are different. This completes the proof. □

THEOREM 4.5 (LIF → TOF). *If a test case can distinguish S and $I_{LIF}^{p_i,x}$ for some i, then it can also distinguish S and $I_{TOF}^{p_i}$.*

PROOF. Let $S = p_1 + \cdots + p_m$ and $I_{LIF}^{p_i,x} = p_1 + \cdots + p_i \cdot x + \cdots + p_m$. Let $\vec{t}$ be a test case that can distinguish S and $I_{LIF}^{p_i,x}$ for some i. Then $\vec{t}$ must satisfy the following conditions: (1) every term other than p_i evaluates to 0, (2) the term p_i

evaluates to 1, and (3) the literal x evaluates to 0. Under such circumstances, $S(\vec{t}) = 1$ and $I_{TOF}^{p_i}(\vec{t}) = p_1 + \cdots + p_{i-1} + p_{i+1} + \cdots + p_m = 0$. Hence, $S(\vec{t})$ and $I_{TOF}^{p_i}(\vec{t})$ are different. This completes the proof. □

THEOREM 4.6 (TOF → ORF[+]).

(a) *If a test case can distinguish S and $I_{TOF}^{p_i}$ for some i, then it can also distinguish S and $I_{ORF[+]}^{p_i}$.*

(b) *If a test case can distinguish S and $I_{TOF}^{p_{i+1}}$ for some i, then it can also distinguish S and $I_{ORF[+]}^{p_i}$.*

PROOF. (a) Let $S = p_1 + \cdots + p_m$ and $I_{TOF}^{p_i} = p_1 + \cdots + p_{i-1} + p_{i+1} + \cdots + p_m$. Let $\vec{t}$ be a test case that can distinguish S and $I_{TOF}^{p_i}$ for some i. This implies that $\vec{t}$ must satisfy the following conditions: (1) every term other than p_i evaluates to 0, and (2) the term p_i evaluates to 1. Under such circumstances, $S(\vec{t}) = 1$ and $I_{ORF[+]}^{p_i}(\vec{t}) = p_1 + \cdots + p_{i-1} + p_i \cdot p_{i+1} + p_{i+2} + \cdots + p_m = 0$. Hence, $S(\vec{t})$ and $I_{ORF[+]}^{p_i}(\vec{t})$ are different. This completes the proof.

(b) The proof is similar to part (a) above and is omitted. □

THEOREM 4.7 (TOF → LNF). *If a test case can distinguish S and $I_{TOF}^{p_i}$ for some i, then it can also distinguish S and $I_{LNF}^{x_j^i}$ for every $j = 1, \ldots, k_i$, where k_i is the number of literals in p_i.*

PROOF. Let $S = p_1 + \cdots + p_m$ and $I_{TOF}^{p_i} = p_1 + \cdots + p_{i-1} + p_{i+1} + \cdots + p_m$. Let $\vec{t}$ be a test case that can distinguish S and $I_{TOF}^{p_i}$ for some i. Then $\vec{t}$ must satisfy the following conditions: (1) every term other than p_i evaluates to 0, and (2) the term p_i evaluates to 1 (that is, every literal $x_1^i, \ldots, x_{k_i}^i$ evaluates to 1). Under such circumstances, $S(\vec{t}) = 1$ and $I_{LNF}^{x_j^i}(\vec{t}) = p_1 + \cdots + x_1^i \cdots \bar{x}_j^i \cdots x_{k_i}^i + \cdots + p_m = 0$. Hence, $S(\vec{t})$ and $I_{LNF}^{x_j^i}(\vec{t})$ are different. This completes the proof. □

THEOREM 4.8 (LOF → LNF). *If a test case can distinguish S and $I_{LOF}^{x_j^i}$ for some i and j, then it can also distinguish S and $I_{LNF}^{x_j^i}$.*

PROOF. Let $S = p_1 + \cdots + p_m$ and $I_{LOF}^{x_j^i} = p_1 + \cdots + x_1^i \cdots x_{j-1}^i \cdot x_{j+1}^i \cdots x_{k_i}^i + \cdots + p_m$. Let $\vec{t}$ be a test case that can distinguish S and $I_{LOF}^{x_j^i}$ for some i and j. Then $\vec{t}$ must satisfy the following conditions: (1) every term other than p_i evaluates to 0, (2) every literal in p_i other than x_j^i evaluates to 1, and (3) the literal x_j^i evaluates to 0. Under such circumstances, $S(\vec{t}) = 0$ and $I_{LNF}^{x_j^i}(\vec{t}) = 1$. Hence, $S(\vec{t})$ and $I_{LNF}^{x_j^i}(\vec{t})$ are different. This completes the proof. □

THEOREM 4.9 (LOF → ORF[·]).

(a) If a test case distinguishes S and $I_{LOF}^{x_j^i}$ for some i, j, then it can also distinguish S and $I_{ORF[\cdot]}^{x_j^i}$.

(b) If a test case distinguishes S and $I_{LOF}^{x_{j+1}^i}$ for some i, j, then it can also distinguish S and $I_{ORF[\cdot]}^{x_j^i}$.

PROOF. (a) Let $S = p_1 + \cdots + p_m$ and $I_{LOF}^{x_j^i} = p_1 + \cdots + x_1^i \cdots x_{j-1}^i \cdot x_{j+1}^i \cdots x_{k_i}^i + \cdots + p_m$. Let $\vec{t}$ be a test case that can distinguish S and $I_{LOF}^{x_j^i}$ for some i and j. Then $\vec{t}$ must satisfy the following conditions: (1) every term other than p_i evaluates to 0, (2) every literal in p_i other than x_j^i evaluates to 1, and (3) the literal x_j^i evaluates to 0. Under such circumstances, $S(\vec{t}) = 0$ and $I_{ORF[\cdot]}^{x_j^i}(\vec{t}) = p_1 + \cdots + x_1^i \cdots x_j^i + x_{j+1}^i \cdots x_{k_i}^i + \cdots + p_m = 1$, because $x_{j+1}^i \cdots x_{k_i}^i = 1$. Hence, $S(\vec{t})$ and $I_{ORF[\cdot]}^{x_j^i}(\vec{t})$ are different. This completes the proof.

(b) The proof is similar to part (a) above and is omitted. □

THEOREM 4.10 (ORF[·] → TNF). *If a test case can distinguish S and $I_{ORF[\cdot]}^{x_j^i}$ for some i and j, then it can also distinguish S and $I_{TNF}^{p_i}$.*

PROOF. Let $S = p_1 + \cdots + p_m$ and $I_{ORF[\cdot]}^{x_j^i} = p_1 + \cdots + x_1^i \cdots x_j^i + x_{j+1}^i \cdots x_{k_i}^i + \cdots + p_m$. Let $\vec{t}$ be a test case that can distinguish S and $I_{ORF[\cdot]}^{x_j^i}$ for some i and j. Then $\vec{t}$ must satisfy the following conditions: (1) every term other than p_i evaluates to 0, and (2) p_i and $x_1^i \cdots x_j^i + x_{j+1}^i \cdots x_{k_i}^i$ evaluate to different truth values. In other words, $\bar{p}_i$ and $x_1^i \cdots x_j^i + x_{j+1}^i \cdots x_{k_i}^i$ evaluate to the same truth value. Under such circumstances, $I_{TNF}^{p_i}(\vec{t}) = \bar{p}_i(\vec{t}) = I_{ORF[\cdot]}^{x_j^i}(\vec{t})$ which is different from $S(\vec{t})$. This completes the proof. □

Notice that, although it is true that ORF[·] → TNF, the relationship "ORF[+] → TNF" does NOT hold. In other words, a test case that can distinguish S and $I_{ORF[+]}^{p_i}$ does not always distinguish S and $I_{TNF}^{p_i}$. For example, the test case 0011 ($a = 0, b = 0, c = 1$, and $d = 1$) can distinguish $S = ab + cd$ and $I_{ORF[+]}^{p_1} = abcd$ (in which the Boolean operator "+" immediately following the first term p_1 in S has been replaced by "·"), where $p_1 = ab$, but it cannot distinguish S and $I_{TNF}^{p_1} = \overline{ab} + cd$ (whose first term p_1 has been negated).

THEOREM 4.11 (ORF[+] → ENF). *If a test case can distinguish S and $I_{ORF[+]}^{p_i}$ for some i ($1 \le i < m$), then it can also distinguish S and $I_{ENF}^{E(p_i,p_{i+1})}$ where $E(p_i, p_{i+1})$ is a disjunction of k ($k > 1$) terms in S containing both p_i and p_{i+1}.*

PROOF. Let $S = p_1 + \cdots + p_m$. For $I_{ORF[+]}^{p_i}$, the Boolean operator "+" between the terms p_i and p_{i+1} in S is being replaced by the operator "·". For $I_{ENF}^{E(p_i,p_{i+1})}$, the expression $E(p_i, p_{i+1})$ is being negated where $E(p_i, p_{i+1})$ is a disjunction of k ($k > 1$) terms in S containting p_i and p_{i+1}. We can rearrange the terms in S such that those terms in $E(p_i, p_{i+1})$ becomes the first k terms in S and the original terms p_i and p_{i+1} become the first term p'_1 and the second term p'_2 in S, respectively. In other words, $S = p'_1 + \cdots + p'_m$, $E(p_i, p_{i+1}) = p'_1 + \cdots + p'_k$, $p'_1 = p_i$, and $p'_2 = p_{i+1}$.

Let $\vec{t}$ be a test case that can distinguish S and $I_{ORF[+]}^{p_i}$. Then $\vec{t}$ must satisfy the conditions that (1) every term other than p'_1 and p'_2 evaluates to 0, and (2) p'_1 and p'_2 evaluate to different truth values. Under such circumstances, $I_{ENF}^{E(p_i,p_{i+1})}(\vec{t}) = \overline{p'_1(\vec{t}) + p'_2(\vec{t})}$, which is different from $S(\vec{t}) = p'_1(\vec{t}) + p'_2(\vec{t})$. This completes the proof. □

5. APPLICATIONS TO SOFTWARE TESTING AND ANALYSIS

Here we demonstrate the applicability of our extended fault class hierarchy to interpret more empirical results, analyze testing strategies, prioritize test cases, and improve the cost-effectiveness of mutation analysis.

5.1 Interpretation of Empirical Data

Kuhn's hierarchy theorizes that VRF is stronger than VNF, and VNF is stronger than "ENF" [Kuhn 1999]. He applied his theory to explain the empirical observations in Weyuker et al. [1994] that "VRFs" were detected less readily than "VNFs", and "VNFs" less readily than "ENFs". However, as noted in Section 2.4, the terms "VRF" and "VNF" in Weyuker et al. [1994] actually referred to LRF and LNF, respectively. Also, Kuhn's [1999] "ENF" corresponds to TNF or ENF in this article. As such, our extended fault class hierarchy is more relevant than Kuhn's theory for interpreting the empirical data in Weyuker et al. [1994].

Kobayashi et al. [2002] experimented with the same set of Boolean specifications in Weyuker et al. [1994] to assess non-specification-based testing methods such as random testing and combinatorial testing. They reported the mutation scores of their testing methods for the same fault types studied in Weyuker et al. [1994]. The mutation score for LRF was found to be less than that of LNF, which was in turn less than that of ENF.[6] Their results have provided additional evidence that, in general, LRF is harder to detect than LNF, and LNF is harder to detect than ENF, which agree with our analysis.

Furthermore, we observe that *all* the testing methods experimented with in both Kobayashi et al. [2002] and Weyuker et al. [1994] detected more ENFs than ORFs. This observation cannot be explained by Kuhn's [1999] hierarchy but our extended hierarchy does show that ORF is stronger than ENF.

Black et al. [2000a, 2000b] performed a series of experiments to evaluate the use of mutation operators on logic specifications extracted from the Cruise Control System, Safety Injection System, and a CPU Stack Program. Some of their mutation operators, such as the *expression negation operator* (ENO), generated classes of faults that approximately matched those defined in [Kuhn 1999] and in this article. The empirical results suggested some relationships among their mutation operators that are consistent with both Kuhn's [1999] fault class hierarchy and our extended hierarchy.

5.2 Design and Evaluation of Testing Strategies

Foster [1980, 1984] argued that test cases should be *sensitive* to code errors. For logic expressions, he specifically required each variable value to individually affect the result [Foster 1984], and also gave specific rules for doing so. Subsequent to his work, many strategies have been proposed for the generation of test cases from Boolean expressions. In this section, we apply our fault class

[6]Note that Kobayashi et al. [2002] used the same terminology as in Weyuker et al. [1994] (see Section 2.4), in which the terms "VRF" and "VNF" were used to mean LRF and LNF, respectively.

hierarchy to assess the fault-revealing ability of some of the more well-known testing strategies.

5.2.1 *The Boolean OperatoR (BOR) Strategy.* Tai and others [Paradkar et al. 1997; Tai 1996; Tai and Su 1987; Vouk et al. 1994] have developed a family of testing strategies for Boolean expressions. The core member of Tai's family is the *BOR (Boolean OperatoR) strategy* [Tai and Su 1987], which guarantees the detection of Boolean operator faults: (1) incorrect `AND`/`OR` operators, and (2) missing/extra `NOT` operators. The first group of faults corresponds to ORF, and the second group corresponds to LNF, TNF, or ENF, depending on the scope of the `NOT` operator. These faults comprise the upper half of our fault class hierarchy in Figure 5. Thus, a BOR test set guarantees to detect the weaker faults but not the stronger faults (such as LRF and LOF) at the lower levels of our hierarchy.

5.2.2 *The Modified Condition/Decision Coverage (MC/DC) Criterion.* The modified condition/decision coverage (MC/DC) criterion for software testing has been in use for more than a decade in the avionics industry. Since the work of Chilenski and Miller [1994], it has gained increasing attention from the research community [Dupuy and Leveson 2000; Jones and Harrold 2003; Kuhn 1999]. Although frequently considered a code-based criterion, MC/DC was actually meant "to guide the selection of test cases at all levels of specification, from inital requirements to source code" [Chilenski and Miller 1994, page 193]. Experiences with real software have shown that MC/DC is highly effective. Dupuy and Leveson [2000] found that a MC/DC adequate test set exposed important errors in the attitude-control software for the HETE-2 system. These errors are not detectable by test cases satisfying a weaker coverage criterion. Little is known, however, about the classes of faults that MC/DC *guarantees* to detect.

MC/DC requires that "*each condition has been shown to independently affect the decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions*" [RTCA/DO-178B 1992, quoted in Chilenski and Miller [1994], page 194]. Chilenski and Miller [1994] distinguished between *weak MC/DC* and *strong MC/DC*. Weak MC/DC treats multiple occurrences of a variable in a Boolean expression as a single condition, whereas strong MC/DC requires each occurrence of a variable to satisfy the "independently affect" requirement separately. They noted that strong MC/DC is harder to satisfy but has the advantage of ensuring decision coverage.

In our framework, weak MC/DC interprets each condition as a *variable* (so that multiple instances are treated as the *same* condition), while strong MC/DC interprets each condition as a *literal* (so that each should be separately considered). We find that, however, even the (strong) MC/DC criterion cannot guarantee the detection of LIF, as demonstrated in Example 5.1 below. Therefore, MC/DC provides no guarantee of detecting all faults in our extended hierarchy.

Example 5.1. Consider the Boolean specification $S_{\mathtt{C}\mapsto\mathtt{O}}$ in Table I in Section 2.5, given by

$$\begin{aligned} S_{\mathtt{C}\mapsto\mathtt{O}} &= \mathit{Ignited} \cdot \mathit{Running} \cdot \overline{\mathit{Toofast}} \cdot \overline{\mathit{Brake}} \cdot \mathit{Brake}' \\ &\quad + \mathit{Ignited} \cdot \mathit{Running} \cdot \overline{\mathit{Toofast}} \cdot \overline{\mathit{Deactivate}} \cdot \mathit{Deactivate}', \end{aligned}$$

which specifies the transition of a Cruise Control System [Atlee and Buckley 1996; Offutt et al. 2003] from `Cruise` to `Override` mode.

Let a, b, c, d, e, f, and g denote the conditions *Ignited*, *Running*, *Toofast*, *Brake*, *Brake′*, *Deactivate*, and *Deactivate′*, respectively. Then $S_{\mathtt{C}\mapsto\mathtt{O}} = ab\bar{c}\bar{d}e + ab\bar{c}\bar{f}g$. Consider the test set $T = \{$`1100100, 0100100, 1000100, 1110100, 1101100, 1100000, 1100001, 0100001, 1000001, 1110001, 1100011`$\}$ whose elements are denoted by $\vec{t}_i$ $(i = 1, \ldots, 11)$, respectively. It is easy to verify that $S_{\mathtt{C}\mapsto\mathtt{O}}(\vec{t}_1) = S_{\mathtt{C}\mapsto\mathtt{O}}(\vec{t}_7) = \mathtt{1}$ and $S_{\mathtt{C}\mapsto\mathtt{O}}(\vec{t}_i) = \mathtt{0}$ for all other values of i. Now $\vec{t}_1$ and $\vec{t}_2$ differ only in the value of a, and $S_{\mathtt{C}\mapsto\mathtt{O}}(\vec{t}_1) \neq S_{\mathtt{C}\mapsto\mathtt{O}}(\vec{t}_2)$. Hence the pair of test cases $(\vec{t}_1, \vec{t}_2)$ can show that the literal a in the first term of $S_{\mathtt{C}\mapsto\mathtt{O}}$ independently affects the outcome of $S_{\mathtt{C}\mapsto\mathtt{O}}$. Similarly, the pairs $(\vec{t}_1, \vec{t}_3)$, $(\vec{t}_1, \vec{t}_4)$, $(\vec{t}_1, \vec{t}_5)$, $(\vec{t}_1, \vec{t}_6)$, $(\vec{t}_7, \vec{t}_8)$, $(\vec{t}_7, \vec{t}_9)$, $(\vec{t}_7, \vec{t}_{10})$, $(\vec{t}_7, \vec{t}_{11})$, and $(\vec{t}_7, \vec{t}_6)$, can show, respectively, that the other literals b, $\bar{c}$, $\bar{d}$, and e in the first term and a, b, $\bar{c}$, $\bar{f}$, and g in the second term independently affect the outcome of $S_{\mathtt{C}\mapsto\mathtt{O}}$. Hence T is MC/DC adequate for $S_{\mathtt{C}\mapsto\mathtt{O}}$.

Next, consider the implementation $I_{LIF}^{p_2,\bar{d}} = ab\bar{c}\bar{d}e + ab\bar{c}\bar{f}g\bar{d}$ caused by the LIF which inserts the literal $\bar{d}$ into the second term p_2 of $S_{\mathtt{C}\mapsto\mathtt{O}}$, where $p_2 = ab\bar{c}\bar{f}g$. This implementation is faulty as $I_{LIF}^{p_2,\bar{d}}(\mathtt{1101101}) = \mathtt{0} \neq S_{\mathtt{C}\mapsto\mathtt{O}}(\mathtt{1101101}) = \mathtt{1}$. However, it would be straightforward to show that $S_{\mathtt{C}\mapsto\mathtt{O}}(\vec{t}_i) = I_{LIF}^{p_2,\bar{d}}(\vec{t}_i)$ for $i = 1, 2, \ldots, 11$. Therefore, T cannot distinguish the specification $S_{\mathtt{C}\mapsto\mathtt{O}}$ from the faulty implementation $I_{LIF}^{p_2,\bar{d}}$.

5.2.3 *The Basic Meaningful Impact Strategy (BASIC).* Weyuker et al. [1994] also developed a family of testing strategies, with the *basic meaningful impact strategy* (abbreviated as BASIC) as the core member. BASIC was explicitly designed to detect LNF. It selects at least one test case from each *unique true point set*, UTP_i, and at least one from each *near false point set*, $\mathit{NFP}_{i,j}$, where $j = 1, \ldots, k_i$ and $i = 1, \ldots, m$. A test case $\vec{t} \in \mathbb{B}^n$ is said to be (1) a *unique true point* for the ith term p_i if $\vec{t}$ renders p_i true but all other terms p_j, $(j \neq i)$ false, and (2) a *near false point* for the jth literal x_j^i of the ith term p_i if the outcome is false at $\vec{t}$ but inverting (negating) x_j^i will cause the outcome to become true.

Chen and Lau [1997, 2001] have recently proved that TOF can be detected by the collection of an arbitrary point from each UTP_i set, and LOF can be detected by the collection of an arbitrary point from each $\mathit{NFP}_{i,j}$ set. It follows that BASIC guarantees to detect not only its target fault LNF, but also TOF and LOF. From Figure 5, it is not surprising that BASIC detects all the classes of faults higher up in the hierarchy (TNF, ORF, and ENF) as well, but as noted earlier by Chen, Lau, and Yu [Chen and Lau 1997, 2001; Chen et al. 1999; Yu and Lau 2002], it may not be able to detect the stronger classes of faults (that is, LIF and LRF).

5.2.4 *The MUMCUT Strategy.* Chen and Lau [1997, 2001] developed three strategies to detect LIF and LRF: (1) the MUTP (*multiple unique true points*) strategy, which guarantees the detection of LIF, (2) the CUTPNFP (*corresponding unique true point and near false point pair*) strategy, which guarantees the detection of LRF provided that the required test point pairs exist, and (3) the MNFP (*multiple near false points*) strategy, which supplements the MUTP strategy to detect LRF in case a CUTPNFP test point pair is not found. Later, the <u>MU</u>TP, <u>M</u>NFP, and <u>CUT</u>PNFP strategies were integrated to form the MUMCUT strategy [Chen et al. 1999; Yu and Lau 2002; Yu et al. 2003]. Since the MUMCUT strategy subsumes the BASIC strategy (which detects LOF, LNF and TOF) and is designed to detect LIF and LRF, it follows that the MUMCUT strategy guarantees to detect all the classes of faults in Figure 5.

The MAX-A and MAX-B strategies in Weyuker et al. [1994] select substantially more test cases than BASIC, and the extra test cases were not targeted to specific faults. Thus, although the MAX-A and MAX-B strategies do detect LIF and LRF, they use far more test cases than are necessary [Chen et al. 1999; Yu et al. 2003].

5.2.5 *Comparison of Testing Strategies.* Kuhn [1999] has applied his fault class hierarchy (Figure 1) to evaluate the effectiveness of test methods published in Chilenski and Miller [1994], Foster [1984], Offutt et al. [2003], and Weyuker et al. [1994]. Obviously, our fault class hierarchy (Figure 5) can also be applied in a similar way. Indeed, in the examples in Sections 6 and 7 in Kuhn [1999] (where the test methods were evaluated), all the variable faults were also literal faults, since these examples involved only singular Boolean expressions, in which every variable occurs once in each expression only. As such, our fault class hierarchy is applicable to all of their examples as well.

All the testing strategies discussed in Sections 5.2.1–5.2.4 above were developed prior to the discovery of the fault class hierarchies in Kobayashi et al. [2002] and Kuhn [1999] and in this article. In retrospect, it is now clear that the fault classes based on which a testing strategy was designed has a significant impact on its ability to detect other faults as well. For instance, the MUMCUT testing strategy (which was designed to detect LIF and LRF) turns out to have a greater fault-detecting ability than BASIC (which was designed to detect LNF), and is more cost-effective than the MAX-A and MAX-B strategies (which subsume BASIC with extra test cases that are not fault-based). Clearly, whenever possible, a testing strategy which is designed to target a stronger fault class will likely be able to detect more classes of faults than as designed. Hence, our fault class hierarchy has provided insights to inform both the design and evaluation of fault-based testing strategies.

Intuitively, the more faults a testing strategy guarantees to detect, the harder it is to satisfy the strategy, and the more test cases it usually requires. The programmer has to determine the right balance between the cost of the testing and the need for well-testedness of the software.

Table III lists several Boolean expressions derived from specifications published in the literature and shows the size of test sets required by several common testing strategies. Let $|M|$ denote the size of a test set required by

Table III. Size of Test Sets Required for Several Published Specifications

Boolean Specification	System [Source]
(S1) $a(\bar{b}+\bar{c})d+e$	TCAS II System (Spec 4) [Weyuker et al. 1994]
(S2) $(ac+bd)e\,(f+(i(gj+hk)))$	TCAS II System (Spec 17) [Weyuker et al. 1994]
(S3)* $ab+c+d+e+f+g+h+i+j$	Boiler Control System (Level 1 CEG) [Paradkar et al. 1997]
(S4) $\bar{a}bc+\bar{d}e+\bar{f}\bar{g}h$	Light Control System [Heitmeyer and Bharadwaj 2000]
(S5)† $ab\bar{c}\bar{d}e+ab\bar{c}\bar{f}g$	Cruise Control System [Atlee and Buckley 1996; Offutt et al. 2003]

No. of vars.	Spec.	BOR	MC/DC	BASIC	MUMCUT	MAX-A	MAX-B	M-CC**
5	(S1)	5	7	9	11	28	31	32
11	(S2)	9	15	38	75	946	1015	2048
10	(S3)	11	11	12	19	28	61	1024
8	(S4)	6	11	11	20	222	231	256
7	(S5)	7‡	11	12	20	35	43	128

* Direct conversion of level 1 CEG (cause-effect graph) of the Boiler Control System gives a nonsingular Boolean expression, which we simplify to (S3) so that the BOR strategy can be directly applied.
† (S5) is the same as $S_{\mathsf{c}\mapsto 0}$ used in Example 5.1 in this article.
** M-CC is an abbreviation for the multiple-condition coverage criterion.
‡ (S5) is converted into the singular Boolean expression $ab\bar{c}(\bar{d}e+\bar{f}g)$ before applying the BOR strategy.

the testing method M. From Table III, the following pattern is fairly obvious: $|\text{BOR}| \leq |\text{MC/DC}| \leq |\text{BASIC}| \lessapprox |\text{MUMCUT}| < |\text{MAX-A}| < |\text{MAX-B}| < |\text{M-CC}|$. In particular, the relative sizes of the test sets generated from the first four strategies (BOR, MC/DC, BASIC, and MUMCUT) seem to be reasonably small and manageable, even for the expression with more than 10 variables. In contrast, the test sets generated from the MAX-A, MAX-B, and M-CC strategies can become very large.

5.3 Prioritization of Test Cases

The *test case prioritization* problem [Elbaum et al. 2002; Jones and Harrold 2003; Yu and Lau 2002] attempts to prioritize test cases so as to fulfill a certain predefined goal. One common goal of prioritization is to increase the rate of fault detection of the test set. Our fault class hierarchy naturally suggests that one assign higher priorities to test cases that detect the stronger classes of faults, since these test cases will also detect many other weaker classes of faults. We now illustrate this idea with an example below. To simplify our exposition, we adopt the terminology of *mutation analysis* [Black et al. 2000b; DeMillo et al. 1978] as outlined in Section 2.2.

Example 5.2. Consider the Boolean specification $S_1 = ab + cd$. Table IV shows all 40 mutants of S_1 due to the eight classes of faults defined in Section 2.3.

Since LIF is at the lowest level of the hierarchy in Figure 5, we choose test cases to kill LIF mutants first. Now the test case $\vec{t}_1 = \texttt{1101}$ can kill the LIF mutant $abc + cd$ (in which the literal c has been inserted in the first term). According to Theorems 4.4 and 4.5, $\vec{t}_1$ also kills the LRF mutants $cb + cd$ (in which the literal a has been replaced by c) and $ac + cd$ (in which the literal b

Table IV. Mutants of $S_1 = ab + cd$

Fault Class	Implementation (mutant)	Fault Class	Implementation (mutant)
ENF	$\overline{ab + cd}$	LRF	$cb + cd, \bar{c}b + cd, db + cd, \bar{d}b + cd,$
TNF	$\overline{ab} + cd, ab + \overline{cd}$		$ac + cd, a\bar{c} + cd, ad + cd, a\bar{d} + cd,$
LNF	$\bar{a}b + cd, a\bar{b} + cd, ab + \bar{c}d, ab + c\bar{d}$		$ab + ad, ab + \bar{a}d, ab + bd, ab + \bar{b}d,$
ORF[+]	$abcd$		$ab + ca, ab + c\bar{a}, ab + cb, ab + c\bar{b}$
ORF[·]	$a + b + cd, ab + c + d$	LIF	$abc + cd, ab\bar{c} + cd, abd + cd,$
TOF	cd, ab		$ab\bar{d} + cd, ab + acd, ab + \bar{a}cd,$
LOF	$b + cd, a + cd, ab + d, ab + c$		$ab + bcd, ab + \bar{b}cd$

has been replaced by c), and the TOF mutant cd (in which the first term has been omitted). By Theorem 4.6, $\vec{t}_1$ kills the ORF[+] mutant $abcd$ as well. By Theorems 4.1–4.3, $\vec{t}_1$ further kills the two LNF mutants $\bar{a}b + cd$ and $a\bar{b} + cd$, the TNF mutant $\overline{ab} + cd$ and the ENF mutant $\overline{ab + cd}$.

Furthermore, the same test case ($\vec{t}_1 =$ `1101`) kills the LIF mutant $ab\bar{d} + cd$. From Theorem 4.4, it also kills the LRF mutants $\bar{d}b + cd$ (in which the literal a has been replaced by $\bar{d}$) and $a\bar{d} + cd$ (in which the literal b has been replaced by $\bar{d}$). So this $\vec{t}_1$ alone already kills 12 mutants.

Similarly, the three test cases `0111`, `1110`, and `1011` may be chosen to kill all the remaining LIF mutants. By the relevant theorems in Section 4, the above four test cases together can kill all mutants except LOF and ORF[·] mutants.

According to Figure 5, we should target at LOF next. From Table IV, there are four LOF mutants for S_1, corresponding to the omission of each literal. They can be killed by the two test cases `1001` and `0110`. By Theorem 4.9, these two test cases kill all ORF[·] mutants as well.

In summary, the six test cases (namely, `1101`, `0111`, `1110`, `1011`, `1001`, and `0110`) together can kill all the 40 mutants in Table IV.

In Example 5.2, all mutants are nonequivalent mutants. However, in some cases, equivalent mutants may exist which no test case can kill. When this happens, the corresponding faults at the next level up in the hierarchy will have to be considered, as illustrated in Example 5.3 below.

Example 5.3. Consider $S_2 = abc + abd + \bar{b}\bar{d} + \bar{e}$. For simplicity, we shall consider only mutants that affect the first term of S_2. For the other terms of S_2, the analysis is similar. Table V shows all 30 mutants due to the eight classes of faults (defined in Section 2.3) that affect the first term of S_2.

For the first term of S_2, there are four LIF mutants. Two of them, ($abc\bar{d} + abd + \bar{b}\bar{d} + \bar{e}$) and ($abce + abd + \bar{b}\bar{d} + \bar{e}$), are equivalent mutants and can be ignored. The test case $\vec{t}_1 =$ `11101` can be used to kill the two LIF mutants $abcd + abd + \bar{b}\bar{d} + \bar{e}$ and $abc\bar{e} + abd + \bar{b}\bar{d} + \bar{e}$. According to Theorem 4.4, $\vec{t}_1$ kills the six corresponding LRF mutants in which any literal in the first term is replaced by either d or $\bar{e}$. By the relevant theorems in Section 4, $\vec{t}_1$ also kills all the nine mutants in Table V due to TOF, ORF[+], LNF, TNF, and ENF. Thus, $\vec{t}_1$ alone kills 17 mutants in Table V.

The fault class immediately above LIF that remains undetected in the hierarchy in Figure 5 is LRF. Since each of the literals a, b, and c in the first term may be replaced by either $\bar{d}$ or e, there are six LRF mutants to be killed. Out of these, the mutant ($a\bar{d}c + abd + \bar{b}\bar{d} + \bar{e}$) is equivalent to S_2 and can be ignored.

Table V. Mutants of $S_2 = abc + abd + \bar{b}\bar{d} + \bar{e}$ Due to Faults Involving the First Term

Fault Class	Implementation (mutant)
ENF	$\overline{abc + abd} + \bar{b}\bar{d} + \bar{e}$, $\overline{abc + abd + \bar{b}\bar{d}} + \bar{e}$, $\overline{abc + abd + \bar{b}\bar{d} + \bar{e}}$
TNF	$\overline{abc} + abd + \bar{b}\bar{d} + \bar{e}$
LNF	$\bar{a}bc + abd + \bar{b}\bar{d} + \bar{e}$, $a\bar{b}c + abd + \bar{b}\bar{d} + \bar{e}$, $ab\bar{c} + abd + \bar{b}\bar{d} + \bar{e}$
ORF[+]	$abc \cdot abd + \bar{b}\bar{d} + \bar{e}$
ORF[·]	$a + bc + abd + \bar{b}\bar{d} + \bar{e}$, $ab + c + abd + \bar{b}\bar{d} + \bar{e}$
TOF	$abd + \bar{b}\bar{d} + \bar{e}$
LOF	$bc + abd + \bar{b}\bar{d} + \bar{e}$, $ac + abd + \bar{b}\bar{d} + \bar{e}$, $ab + abd + \bar{b}\bar{d} + \bar{e}$
LRF	$dbc + abd + \bar{b}\bar{d} + \bar{e}$, $\bar{d}bc + abd + \bar{b}\bar{d} + \bar{e}$, $ebc + abd + \bar{b}\bar{d} + \bar{e}$, $\bar{e}bc + abd + \bar{b}\bar{d} + \bar{e}$, $adc + abd + \bar{b}\bar{d} + \bar{e}$, $(a\bar{d}c + abd + \bar{b}\bar{d} + \bar{e})^*$, $aec + abd + \bar{b}\bar{d} + \bar{e}$, $a\bar{e}c + abd + \bar{b}\bar{d} + \bar{e}$, $abd + abd + \bar{b}\bar{d} + \bar{e}$, $ab\bar{d} + abd + \bar{b}\bar{d} + \bar{e}$, $abe + abd + \bar{b}\bar{d} + \bar{e}$, $ab\bar{e} + abd + \bar{b}\bar{d} + \bar{e}$
LIF	$abcd + abd + \bar{b}\bar{d} + \bar{e}$, $(abc\bar{d} + abd + \bar{b}\bar{d} + \bar{e})^*$, $(abce + abd + \bar{b}\bar{d} + \bar{e})^*$, $abc\bar{e} + abd + \bar{b}\bar{d} + \bar{e}$

* These implementations are not faulty since they are equivalent to the specification S_2.

By now, there are 10 live nonequivalent mutants in Table V left: five LRF mutants, three LOF mutants and two ORF[·] mutants. It turns out that the test cases 01101, 10111, and 11001 together can kill all LRF and LOF mutants. By Theorem 4.9, they also kill the two ORF[·] mutants.

In summary, the four test cases (11101, 01101, 10111, and 11001) together can kill all the 27 nonequivalent mutants in Table V.

Previously, Chen and colleagues [Chen and Lau 1997, 2001; Chen et al. 1999; Yu et al. 2003] noted that a MUMCUT test set (which satisfies the MUMCUT testing strategy) guarantees the detection of all the faults described in Section 2.3, but the problem of prioritizing test cases within a MUMCUT test set has not been completely solved yet.

In Chen and Lau [2001], it was shown that (1) the MUTP strategy guarantees the detection of LIF; (2) either of the CUTPNFP strategy or the MNFP strategy guarantees the detection of LOF; (3) the CUTPNFP strategy, supplemented by the MUTP and MNFP strategies if necessary, guarantees the detection of LRF; and (4) the CUTPNFP strategy generally requires fewer test cases than does the MNFP strategy. Therefore, our extended fault class hierarchy suggests that test cases satisfying the M$\underline{\text{U}}$TP strategy should be executed first, followed by those satisfying the $\underline{\text{C}}$UTPNFP strategy, and finally by those satisfying the M$\underline{\text{N}}$FP strategy. We refer to this ordering of test cases as the *UCN order*.

In a recent study, using the set of specifications in [Weyuker et al. 1994], Yu and Lau [2002] compared the fault detection rates of MUMCUT test sets. They found that the UCN order was ranked highest among the six possible execution orders: CNU, CUN, NCU, NUC, UCN, and UNC. Again, their empirical results are consistent with the extended fault class hierarchy developed in this article.

5.4 Impact on Mutation Analysis

Originally proposed as a means of creating test cases [DeMillo et al. 1978], mutation analysis has also been found to be very useful for assessing the strength of a test set generated [Ammann and Black 2001; Offutt et al. 1996]. In the early days, it was primarily used as a code-based testing technique [Daran and Thévenod-Fosse 1996; DeMillo et al. 1978; Offutt et al. 1996], but in the past few years its application has been extended to specification-based testing [Ammann and Black 2001; Black et al. 2000a, 2000b; Kuhn 1999; Weyuker et al. 1994].

There is a close relationship between fault class analysis and mutation testing. First, mutation operators are often designed to model common faults. On the other hand, each fault class corresponds naturally to a potentially useful mutation operator. Conversely, applying a mutation operator gives rise to a fault in that class. The results reported in this article will therefore have direct impact to the analysis and application of mutation testing to logic expressions, particularly in relation to the design and selection of mutation operators.

A major obstacle that limits the practical use of mutation testing is the high cost incurred due to the large number of possibly useful mutants. One way to lower its cost is to reduce the number of mutants, typically through the selective use of mutation operators [Ammann and Black 2001; Offutt et al. 1996]. As noted by Ammann and Black [2001] as well as by Black et al. [2000a, 2000b], if the mutation operators corresponding to the stronger fault classes are used, those corresponding to the weaker fault classes will not be necessary. This can significantly reduce the number of mutants by using only the mutation operators corresponding to the stronger fault classes. Indeed, inspired by this observation and applying Kuhn's [1999] hierarchy, Black et al. [2000a] empirically found a highly effective subset of mutation operators that yields only a fraction of all mutants. Our extended fault class hierarchy is more comprehensive than Kuhn's, providing the relationships among a variety of fault classes (such as ORF and LRF) that were previously unknown. Obviously, it will have more potential applications to the elimination of unnecessary mutation operators for Boolean specifications.

Mutation operators commonly discussed in the research literature, however, do not correspond exactly to the fault classes in the hierarchies developed so far. Our classification of fault classes is more refined than those in previous work. For instance, we distinguish LOF from TOF, while previously they were often pooled together as "missing condition" fault. It is likely that more useful mutation operators can be designed to capture the finer classes of faults that a programmer may commit during software development. Our extended fault class hierarchy can then be used for an informed selection of some mutation operators over others to improve the cost-effectiveness of mutation analysis.

6. CONCLUSION

A hierarchy has recently been developed [Kuhn 1999; Tsuchiya and Kikuno 2002] that captures the relationships among the detection conditions of several fault classes used in specification-based software testing. In this article, we have extended the hierarchy to include other classes of faults considered in the literature. We have clarified the terminology regarding faults associated with Boolean specifications, elucidated imprecise notions, and provided a more refined framework for the analysis of related fault classes. We carefully point out the sometimes neglected distinction between variable and literal faults. We contend that literal faults deserve more attention, since they are intuitively more elementary and more easily escape detection.

A major contribution of this article is the establishment of an extended fault class hierarchy that relates literal, term, operator, and expression faults. Within the hierarchy, a test case that detects a fault of a stronger class will always detect the corresponding fault(s) of a weaker class. One practical implication is that a testing strategy designed for the detection of faults of a stronger class will be more effective, as it will also detect faults of weaker classes. Our results have extended and complemented similar recent work on variable and expression faults [Kuhn 1999; Tsuchiya and Kikuno 2002]. Like these previous works, although our analysis originated from the study of faults related to Boolean specifications, the results are equally applicable to the code-based testing of predicates.

Our extended fault class hierarchy has led not only to a better theoretical understanding of conditions for fault detection, but also to a wide range of applications that will have a potential impact on both practices and further research in software testing and analysis. We have demonstrated that it helps to (1) interpret previous empirical results, (2) suggest ways to design fault-based testing strategies that use fewer test cases, (3) establish a framework for evaluating the effectiveness of testing strategies, (4) provide guidelines for prioritizing test cases, and (5) motivate the design and inform the selection of more useful mutation operators to improve the cost-effectiveness of mutation analysis.

We have considered faults related to Boolean expressions which have been extensively used in many levels of descriptions of software systems, such as the state-based specifications of requirements [Atlee and Gannon 1993; Heimdahl and Leveson 1996; Offutt et al. 2003] and predicates that govern the flow of execution of programs [Foster 1984; Chilenski and Miller 1994; Tai 1996]. Such expressions, which include the equivalent forms derived from state-transition tables [Atlee and Gannon 1993; Offutt et al. 2003] or AND/OR tables [Heimdahl and Leveson 1996; Leveson et al. 1994], partly but often crucially determine the correctness of the system's functional behavior and whether the desirable nonfunctional attributes (such as safety properties) of the system are preserved [Ammann et al. 2001; Leveson et al. 1991]. These systems, characterized by the high complexity of the logic expressions involved, warrant a systematic generation of test cases to ensure the adequacy of testing for uncovering potential faults. We have demonstrated how the fault class relationships can be utilized to effectively reduce or prioritize the test cases for fault detection in these kinds of systems. We look forward to extending our work to other classes of faults, including faults that may occur in other forms of specifications or other types of systems, so that the applicability of fault class relationships can be further broadened.

ACKNOWLEDGMENTS

We would like to thank T. Y. Chen for his valuable comments on this work.

REFERENCES

Ammann, P., Ding, W., and Xu, D. 2001. Using a model checker to test safety properties. In *Proceedings of the 7th Annual International Conference on Eng. of Complex Computer Systems* (ICECCS '01). 212–221.

Ammann, P. E. and Black, P. E. 2001. A specification-based coverage metric to evaluate test sets. *Int. J. Reliab. Qual. Safety Eng. 8*, 4 (Dec.), 275–299.

Atlee, J. M. and Buckley, M. A. 1996. A logic-model semantics for SCR software requirements. In *Proceedings of 1996 International Symposium on Software Testing and Analysis* (ISSTA '96). 280–292.

Atlee, J. M. and Gannon, J. 1993. State-based model checking of event-driven system requirements. *IEEE Trans. Softw. Eng. 19*, 1 (Jan.), 24–40.

Black, P. E., Okun, V., and Yesha, Y. 2000a. Mutation of model checker specifications for test generation and evaluation. In *Proceedings of Mutation 2000*. 14–20.

Black, P. E., Okun, V., and Yesha, Y. 2000b. Mutation operators for specifications. In *Proceedings of the 15th Automated Software Engineering Conference* (ASE 2000). 81–88.

Chan, W., Anderson, R. J., Beame, P., Burns, S., Modugno, F., Notkin, D., and Reese, J. D. 1998. Model checking large software specifications. *IEEE Trans. Softw. Eng. 24*, 7 (July), 498–520.

Chang, J., Richardson, D. J., and Sankar, S. 1996. Structural specification-based testing with ADL. In *Proceedings of 1996 International Symposium on Software Testing and Analysis* (ISSTA '96). 62–70.

Chen, T. Y. and Lau, M. F. 1997. Two test data selection strategies towards testing of Boolean specifications. In *Proceedings of the 21st International Computer Software and Application Conference* (COMPSAC '97). 608–611.

Chen, T. Y. and Lau, M. F. 2001. Test case selection strategies based on Boolean specifications. *Softw. Test. Verif. Reliab. 11*, 3 (Sept.), 165–180.

Chen, T. Y., Lau, M. F., and Yu, Y. T. 1999. MUMCUT: A fault-based strategy for testing Boolean specifications. In *Proceedings of Asia-Pacific Software Engineering Conference* (APSEC '99). 606–613.

Chen, T. Y. and Yu, Y. T. 1994. On the relationship between partition and random testing. *IEEE Trans. Softw. Eng. 20*, 12 (Dec.), 977–980.

Chen, T. Y. and Yu, Y. T. 1996. On the expected number of failures detected by subdomain testing and random testing. *IEEE Trans. Softw. Eng. 22*, 2 (Feb.), 109–119.

Chilenski, J. J. and Miller, S. P. 1994. Applicability of modified condition/decision coverage to software testing. *IEE/BCS Softw. Eng. J. 9*, 5 (Sept.), 193–200.

Daran, M. and Thévenod-Fosse, P. 1996. Software error analysis: A real case study involving real faults and mutations. In *Proceedings of 1996 International Symposium on Software Testing and Analysis* (ISSTA '96). 158–171.

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

Dick, J. and Faivre, A. 1993. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the First International Symposium of Formal Methods Europe (FME 1993)*. Lecture Notes in Computer Science, vol. 670. Springer, Berlin, Germany, 19–23.

Dupuy, A. and Leveson, N. 2000. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In *Proceedings of Digital Aviation Systems Conference* (DASC 2000).

Duran, J. W. and Ntafos, S. C. 1984. An evaluation of random testing. *IEEE Trans. Softw. Eng. 10*, 4 (July), 438–444.

Elbaum, S., Malishevsky, A. G., and Rothermel, G. 2002. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng. 28*, 2 (Feb.), 159–182.

Foster, K. A. 1980. Error sensitive test cases analysis (ESTCA). *IEEE Trans. Softw. Eng. 6*, 2 (May), 258–264.

Foster, K. A. 1984. Sensitive test data for logic expressions. *ACM SIGSOFT Softw. Eng. Notes 9*, 2 (Apr.), 120–125.

Gargantini, A. and Heitmeyer, C. 1999. Using model checking to generate tests from requirements specifications. In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Lecture Notes in Computer Science, vol. 1687. Springer, Berlin, Germany, 146–162.

Gargantini, A. and Riccobene, E. 2001. ASM-based testing: Coverage criteria and automatic test sequence generation. *J. Univers. Comput. Sci. 7*, 11 (Nov.), 1050–1167.

Gargantini, A. and Riccobene, E. 2003. Automatic model driven animation of SCR specifications. In *Proceedings of the Sixth International Conference on Fundamental Approaches to Software Engineering* (FASE 2003). Lecture Notes in Computer Science, vol. 2621. Springer, Berlin, Germany, 294–309.

Heimdahl, M. P. and Leveson, N. G. 1996. Completeness and consistency in hierarchical state-based requirements. *IEEE Trans. Softw. Eng. 22*, 6 (June), 363–377.

Heitmeyer, C. and Bharadwaj, R. 2000. Applying the SCR requirements method to the light control case study. *J. Univers. Comput. Sci. 6*, 7 (Aug.), 650–678.

Hierons, R. M. 1997. Testing from a Z specification. *Softw. Test. Verif. Reliab. 7*, 1 (Mar.), 19–33.

Hierons, R. M. 2002. Comparing test sets and criteria in the presence of test hypotheses and fault domains. *ACM Trans. Softw. Eng. Method. 11*, 4 (Oct.), 427–448.

Jones, J. A. and Harrold, M. J. 2003. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans. Softw. Eng. 29*, 3 (Mar.), 195–209.

Kobayashi, N., Tsuchiya, T., and Kikuno, T. 2002. Non-specification-based approaches to logic testing for software. *Inf. Softw. Tech. 44*, 2 (Feb.), 113–121.

Kuhn, D. R. 1999. Fault classes and error detection capability of specification-based testing. *ACM Trans. Softw. Eng. Method. 8*, 4 (Oct.), 411–424.

Lau, M. F. and Yu, Y. T. 2001. On the relationships of faults for Boolean specification based testing. In *Proceedings of Australian Software Engineering Conference* (ASWEC 2001). 21–28.

Leveson, N. G., Cha, S. S., and Shimeall, T. J. 1991. Safety verification of ADA programs using software fault trees. *IEEE Softw. 8*, 4 (July), 48–59.

Leveson, N. G., Heimdahl, M. P. E., Hildreth, H., and Reese, J. D. 1994. Requirements specification for process-control systems. *IEEE Trans. Softw. Eng. 20*, 9 (Sept.), 684–707.

Madeira, H., Costa, D., and Vieira, M. 2000. On the emulation of software faults by software fault injection. In *Proceedings of International Conference on Dependable Systems and Networks* (DSN 2000). 417–426.

Myers, G. J. 1979. *The Art of Software Testing*, 2nd ed. John Wiley, New York, NY.

Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., and Zapf, C. 1996. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Method. 5*, 2 (Apr.), 99–118.

Offutt, A. J., Liu, S., Abdurazik, A., and Ammann, P. 2003. Generating test data from state-based specifications. *Softw. Test. Verif. Reliab. 13*, 1 (Mar.), 25–53.

Paradkar, A., Tai, K., and Vouk, M. 1997. Specification-based testing using cause-effect graphs. *Ann. Softw. Eng. 4*, 133–157.

Richardson, D. J. and Thompson, M. C. 1993. An analysis of test data selection criteria using the RELAY model of fault detection. *IEEE Trans. Softw. Eng. 19*, 6 (June), 533–553.

RTCA/DO-178B. 1992. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Inc., Washington, DC.

Stock, P. and Carrington, D. 1996. A framework for specification-based testing. *IEEE Trans. Softw. Eng. 22*, 11 (Nov.), 777–793.

Tai, K. C. 1996. Theory of fault-based predicate testing for computer programs. *IEEE Trans. Softw. Eng. 22*, 8 (Aug.), 552–562.

Tai, K. C. and Su, H. K. 1987. Test generation for Boolean expressions. In *Proceedings of the 11th International Computer Software and Application Conference* (COMPSAC '87). 278–284.

Tsuchiya, T. and Kikuno, T. 2002. On fault classes and error detection capability of specification-based testing. *ACM Trans. Softw. Eng. Method. 11*, 1 (Jan.), 58–62.

Vouk, M. A., Tai, K. C., and Paradkar, A. 1994. Empirical studies of predicate-based software testing. In *Proceedings of International Symposium on Software Reliability Engineering*. 55–64.

Weyuker, E. J., Goradia, T., and Singh, A. 1994. Automatically generating test data from a Boolean specification. *IEEE Trans. Softw. Eng. 20*, 5 (May), 353–363.

Yu, Y. T. and Lau, M. F. 2002. Prioritization of test cases in MUMCUT test sets: An empirical study. In *Proceedings of the 7th International Conference on Reliable Software Technologies—Ada-Europe 2002*. Lecture Notes in Computer Science, vol. 2361. Springer, Berlin, Germany, 245–256.

Yu, Y. T., Lau, M. F., and Chen, T. Y. 2003. Automatic generation of test cases from Boolean specifications using the MUMCUT strategy. Submitted for publication.

Received February 2004; revised August 2004; accepted October 2004