IACSIT International Journal of Engineering and Technology Vol.1,No.5,December,2009
ISSN: 1793-8236

Toward A Theory of Test Case Reduction in Specification Based Software Testing

Rakesh Kumar Kulvinder Singh

***Abstract*—During the software development, numbers of mistakes are committed by software developers consequential to the insertion of a number of faults in the program. The behavior of a faulty program may be different from expected one. Since testing to detect all imaginable faults is impossible because of large numbers of test cases are required. Fault based testing strategies detects only pre-defined types of faults. Kuhn's fault class hierarchies provide the focus of fault-based testing strategies on detecting particular faults. We have purposed supplementary fault detection strategies by considering Coupling Effect Hypothesis and the Competent Programmer Hypothesis to detect faults with small number of test cases. Our results extend Kuhn's fault class hierarchy combine with Black's fault detection strategies that provide a focus on testing strategies for detection of faults. The resulting tests are also effective for detecting faults in other classes with the sane test cases.**

Key words—**Fault-based testing; Fault classes, Specification-based testing;**

I. INTRODUCTION

Software testing is crucial and decisive in ensuring the quality of software. A formal specification provides the entire knowledge about a system and valuable information for testing programs. The aim of fault-based testing is to generate tests to detect faults in software [1], [10]. The mutation method is a fault-based testing strategy that measures the quality/adequacy of testing by examining whether the test set used in testing can reveal certain types of faults. Given a program p, a mutant is some variant $\hat{p}$ of p. A mutant is generated by applying some mutation operators i.e. rules that allow us to transform programs. A mutant produced by applying one instance of one operator only is called first-order mutants. By applying a mutation operator to a mutant, a mutant of a mutant is generated known as second order mutant. By mutating a second order mutant, a third order mutant is obtained and so on. These "higher order" mutants are not considered in Mutation Testing.

Using only first-order mutants has been justified in two ways. Firstly, it is argued that if our test finds the small differences defined by first-order mutants, then it is likely that it will find larger differences defined by higher-order mutants. It is based on The Coupling Effect Hypothesis i.e. "Large program faults, particularly those of a semantic nature are coupled with smaller syntactic faults that can be detected with mutation testing". Secondly, it is also argued that real programmers make small mistakes and thus that real programs are like first-order mutants of correct programs. It is based on The Competent Programmer Hypothesis i.e. "In general programmers are competent i.e., the programs they write are nearly correct. The program differs from a correct version in only a few small ways. Kuhn's hierarchy [1] of fault implies that some faults may be left during software testing. Earlier results [2], [6] were restricted for specifications in disjunctive normal form (DNF). Vadim [10] strategy is capable of removing the restriction to DNF. By use of fault-based testing using DNF specifications may fail to notice faults that can be detected if testing were done from the original specifications [1, 8]. Kuhn developed the hierarchy based on detection conditions for fault classes. We have used Kuhn's and Vadim et al [10] approach. We consider only the conditions for Logical Operator Reference Faults (LRF), CIF, Clause Negation Fault (CNF) and Expression Negation Fault (ENF) depending upon the particular operators or association faults chosen. Note that the conditions under which a particular fault will cause a failure are defined by the difference of the specification with respect to the particular fault. We also extend the hierarchy to include additional fault such as LRF, ENF, Clause Reference Fault (CRF), and CNF and stuck off faults (STF). The use of fault conditions enables us to analyze existing testing methods. For instance, we find that the basic meaningful impact strategy is stronger in that it tests for LRF and not variable negation faults.

II. TYPES OF FAULT

In the software development process, software developers may make a numbers of mistakes resulting into the introduction of number of faults in the program. The Faults may involve Boolean variables, Boolean operators, relational operators, logical operator or arithmetic expressions. The different faults which may occur are:

(i) Clause Reference Fault (CRF) – Clause 'a' is replaced with another clause 'b'. For example, the specification (x >7) V (y < 3) is implemented as (x > 9) V (y < 3).

(ii) Clause Negation Fault (CNF) - Clause a is replaced by its negation $\bar{a}$.

(iii) Clause Insertion Fault (CIF) - Clause b is inserted, for example clause a is replaced by $a\ op\ b$, where

Faculty of Computer Science, Department of Computer Science and Applications(D.C.S.A), Kurukshetra University, Kurukshetra (K.U.K)- India
Faculty of Computer Engineering, University Institute of Engineering & Technology (U.I.E.T), Kurukshetra University, Kurukshetra (K.U.K),India

$b$another clause, and op is is is either conjunction or disjunction. There are two subclasses of this class.

(iv) Clause Conjunction Fault (CCF) - Clause a is replaced by $a \wedge b$.

(v) Clause Disjunction Fault (CDF) - Clause a is replaced by $a \vee b$.

(vi) Relational Operator Reference Fault (RRF) - Relational operator is replaced by any other relational operator. Note that replacing a relational operator with its opposite is the same as negating the whole relational expression.

(vii) Off-By-1 Fault (OFF) - in a relational expression E1*op*E2, replace the arithmetic expression E2 with E2 + 1 or E2 − 1.

(viii) Stuck-At Fault (STF) - stuck-at-0 replaces a clause with 0, stuck-at-1 replaces it with 1.

(ix) Expression Negation Fault (ENF) - replace an expression E by$\bar{E}$.

(x) Missing Expression Fault (MEF) - a predicate is omitted during implementation. MEF includes both where a clause is missing and where a compound predicate is missing.

(xi) Logical Operator Reference Fault (LRF) - a Boolean operator is replaced by another operator, e.g., x $\vee$ y is replaced by x $\wedge$ y.

(xii) Associative Shift Fault (ASF) - change the associability of terms. For example, replace (ab) $\vee$ c with a (b $\vee$ c).

(xiii) Term Omission Fault (TOF): A particular term is omitted during the implementation. For example $(a \wedge b) \vee (c \wedge d) \vee (e \wedge f)$ are wrongly implemented as$(a \wedge b) \vee (c \wedge d)$.

(xiv) Literal Negation Fault (LNF): A literal in a particular term is wrongly implemented as its negation.

(xv) Literal Omission Fault (LOF): A literal in a particular term is omitted during the implementation such as $(a \wedge b \wedge c) \vee (d \vee e \wedge f)$ being implemented as$(a \wedge b) \vee (d \vee e \wedge f)$.

(xvi) Literal Insertion Fault (LIF): A literal not appearing in a particular term is inserted into that term. For example, $(a \wedge b) \vee (d \vee e \wedge f)$ is incorrectly implemented as$(a \wedge b \wedge c) \vee (d \vee e \wedge f)$.

(xvii) Literal Reference Fault (LRF): A literal in a particular term is replaced by another literal not appearing in the term during the implementation.

Where

$\bar{a}$ (horizontal line above an operand)	Negation
$\vee$	Disjunction
$\wedge$	Conjunction
$\oplus$	Exclusive-or
$\leftrightarrow$	Equivalence
$\rightarrow$	Implication
$<, \leq, =, \neq, >, \geq$	*Operator*

And clause is either a Boolean variable or a relational expression [10]. A relational expression is of the form A operator B, where A and B are arithmetic expressions A compound predicate consists of one or more binary Boolean operators and their operands. A predicate is either a clause or a compound predicate. These fault classes correspond closely to first-order mutants that may occur in software specifications, where one occurrence of first-order mutants may be an error while another occurrence is correct [10].

III. PREVIOUS WORK

The objective of only using first-order mutants is reduction of efforts i.e. if we do not restrict ourselves to first-order mutants then the total number of mutants is likely to be extremely large. To imagine all types of faults is not possible but some faults classes can be hypothesized and test sets can be constructed. Kuhn [1] purposed the techniques for analyzing the effects of faults in specifications. Let S denote a specification predicate considered to be correct and S′ a faulty version of it. A test detects the fault if and only if it causes S′ to evaluate to a different value than S, formally when S $\oplus$ S′. The notation S^{X}_{E} signifies that a predicate X of specification S is replaced by a predicate E. Kuhn's [2] classification of detection condition for the fault is dS^{X}_{E} = S $\oplus$ S^{X}_{E}, in other words, S^{X}_{E} evaluates to a different value than S and is referred to as the detection condition for the fault. By considering this condition Kuhn [2] compared the detection conditions for different faults like variable reference fault (VRF), variable negation fault (VNF), and expression negation fault (ENF) and he purposed that if any test that detects a VRF for some variable also detects a VNF for the same variable, also the test that detects the VNF for some variable also detects an ENF for the expression in which the variable occurs. A number of researchers [2, 6, 8, 9, 10] used Kuhn's technique to compare fault classes.

The restriction of disjunctive normal form (DNF) was removed by Vadim et al [10] and considered another classes of faults likes CRF, CNF, ENF, CCF, and CDF and. They proved that a test case that detects CRF can also detect CNF and a test case that detects CNF can also detect ENF. Detection condition is an effective and concise analytical tool for studying faults in formal specifications. They refine the fault detection conditions.

Different faults like LIF, LRF, LOF TOF, LNF, TNF, and ENF were considered by [6]. They showed that if a test case detects literal insertion fault will also detect literal reference fault, literal omission faults and a test case that detects LRF, TOF, or LOF can also detect LNF. They also showed that if a test case that detects LNF can also detect TNF, a test case that detects TNF can also detect ENF.

Research on the tests that detect Missing Clause Fault (MCF) will also detect VNF was considered in [9]. They also discovered that tests that detect MCF may not be able to detect VRF, and vice versa. They also proved that a test set that detects MCFs for single variable terms, as well as VRFs, is sufficient to detect both VRFs and MCFs.

IV. FAULT CONDITIONS

With the following truth table, we can analyze fault conditions for various fault classes. For any predicates x, y, and z, the following identities are built from the above truth table and used throughout in this paper:

$x \wedge y \oplus y \wedge z = (x \oplus y) \wedge z$ (1)

$x \wedge y \oplus y = \bar{x} \wedge y$ (2)

$(x \oplus y) \wedge \bar{z} = (x \vee y) \oplus (y \vee z)$ (3)

$x \oplus y \oplus \bar{x} \oplus y = 1$ (4)

$x \oplus y = (x \vee y) \oplus (x \wedge y)$ (5)

$(x \wedge y) \rightarrow z = 1$ (6)

$x \oplus \bar{x} = 1$ (7)

Let S denote a specification predicate and a is a clause in S, b is another valid clause and E is an expression in S. The notation S_F is used to represent the detection condition for an arbitrary fault belonging to fault class F. The detection conditions [10] for fault classes CRF, CNF, ENF, CCF, CDF and LRF are summarized as.

$S_{CRF} = ds_b^a$ Clause Reference Fault
$S_{CNF} = dS_{\bar{a}}^{a}$ Clause Negation Fault
$S_{ENF} = dS_{\bar{E}}^{E}$ Expression Negation Fault
$S_{CCF} = dS_{a \wedge b}^{a}$ Clause Conjunction Fault
$S_{CDF} = dS_{a \vee b}^{a}$ Clause Disjunction Fault
$S_{LRF} = dS_{a \vee b}^{a \wedge b}$ Logical Operator Reference Fault

First consider the relationship between Logical Operator Reference Fault (LRF) and clause negation faults (CNF).

(i) Any test case that detects a Logical Operator Reference Fault (LRF) for a clause in a predicate will also detect the clause negation faults (CNF) for the same clause, then $S_{LRF} \rightarrow S_{CNF}$.

Proof: For a predicate S and a clause a occurring in S, and in Logical Operator Reference Fault (LRF) a operator is replaced by another operator, e.g., $a \wedge b$ is replaced by $a \vee b$. Then LRF is given by:

$$dS_{a \vee b}^{a \wedge b} = (a \vee b) \oplus (a \wedge b)$$

$$= (a \oplus b)$$

The clause negation faults (CNF) where a is replaced by its Negation $\bar{a}$ and given by

$dS_{\bar{a}}^{a} = a \oplus \bar{a} = 1$ Then

$dS_{a \vee b}^{a \wedge b} \rightarrow dS_{\bar{a}}^{a}$ holds, where b is another valid clause and $b \neq a$.

Writing with detection condition [10] i.e.

$dS_B^A = (A \oplus B) \wedge \frac{dS}{dA}$ Where A is a predicate in S and is replaced by another predicate **B.** We have

$$(a \vee b) \oplus (a \wedge b) \wedge \frac{dS}{dA} \rightarrow (a \oplus \bar{a}) \wedge \frac{dS}{dA}$$

$$(a \oplus b) \wedge \frac{dS}{dA} \rightarrow \frac{dS}{dA}$$

With equation (6) the above expression will holds. So if any test case that detects a LRF for a clause in a predicate will also detect the CNF for the same clause.

(ii) Any test case that detects a Logical Operator Reference Fault (LRF) for a clause in a predicate will also detect the Expression negation faults (ENF) for the same clause, then $S_{LRF} \rightarrow S_{ENF}$.

Proof: For a predicate S and a clause a occurring in S, and in LRF a operator is replaced by another operator, e.g., $a \wedge b$ is replaced by $a \vee b$. Then LRF is given by:

$dS^{a\wedge b}_{a\vee b} = (a \vee b) \oplus (a \wedge b)$

$= (a \oplus b)$

And the Expression Negation fault (ENF) is inserting an Expression $\bar{E}$ in place of Expression E.
$dS^{E}_{\bar{E}}$ = (E$\oplus \bar{E}$) where E is replaced by $\bar{E}$ then
$dS^{a\wedge b}_{a\vee b} \leftrightarrow dS^{E}_{\bar{E}}$ hold.
Writing with detection condition [10] i.e.
$dS^{A}_{B} = (A \oplus B) \wedge \frac{dS}{dA}$
Where A is a predicate in S and is replaced by another predicate **B.** We have

$(a \vee b) \oplus (a \wedge b) \wedge \frac{dS}{dA} \rightarrow (E \oplus \bar{E}) \wedge \frac{dS}{dE}$

$(a \oplus b) \wedge \frac{dS}{dA} \rightarrow \frac{dS}{dE}$ **where** $E \oplus \bar{E} = 1$

With equation (6) the above expression will holds. So if any test case that detects a LRF for a clause in a predicate will also detect the ENF for the same clause.

(iii) **Any test case that detects a Clause Insertion faults (CIF) for a clause in a predicate will also detect the Logical Operator Reference Fault (LRF) for the same clause and vice versa, then $S_{CCF} \vee S_{CDF} \leftrightarrow S_{LRF}$**

Proof: For a predicate S and a clause a occurring in S, and in Logical Operator Reference Fault (LRF) a operator is replaced by another operator, e.g., $a \wedge b$ is replaced by $a \vee b$. Then the logical operator reference fault is:

$dS^{a\wedge b}_{a\vee b} = (a \vee b) \oplus (a \wedge b)$

$= (a \oplus b)$

And the clause Insertion fault (CIF) is insert a clause b, which is, replace a clause a by a *op* b, where b is another clause, *op* is either conjunction or disjunction.
= $a \oplus (a \wedge b)$ where a is replaced by $(a \wedge b)$ **or**
= $a \oplus (a \vee b)$ where a is replaced by $(a \vee b)$ then
$dS^{a\wedge b}_{a\vee b} \leftrightarrow dS^{a}_{a\wedge b} \vee dS^{a}_{a\vee b}$ hold.

Writing with detection condition [10] i.e.
$dS^{A}_{B} = (A \oplus B) \wedge \frac{dS}{dA}$
Where A is a predicate in S and is replaced by another predicate **B.** We have

$(a \vee b) \oplus (a \wedge b) \wedge \frac{dS}{dA} \leftrightarrow (a \oplus (a \wedge b)) \wedge \frac{dS}{dA}$

$\vee (a \oplus (a \vee b)) \wedge \frac{dS}{dA}$

$(a \oplus b) \wedge \frac{dS}{dA} \leftrightarrow (a\bar{b}) \wedge \frac{dS}{dA} \vee (\bar{a}\, b) \wedge \frac{dS}{dA}$
$= (a\bar{b} \vee \bar{a}\, b) \wedge \frac{dS}{dA}$
$= (a \oplus b) \wedge \frac{dS}{dA}$

From this expression $S_{CCF} \vee S_{CDF} \leftrightarrow S_{LRF}$ will holds. So if any test case that detects a LRF for a clause in a predicate will also detect the CIF for the same clause.

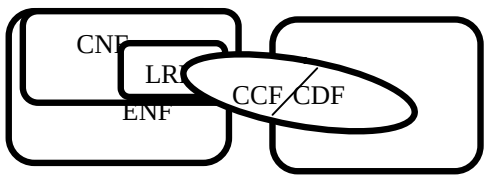
(iv) **Any test case that detects a Clause Insertion Fault (CIF) for a clause in a predicate will also detect the Stuck-At-0 and Stuck-At-1 faults for the same clause, then $S_{CIF} \rightarrow S_{STF}$**

Proof: clause Insertion fault (CIF) is insert a clause b, which is, replace a clause a by a *op* b, where b is another clause, *op* is either conjunction or disjunction.
= $a \oplus (a \wedge b)$ where a is replaced by $(a \wedge b)$ **or**
= $a \oplus (a \vee b)$ where a is replaced by $(a \vee b)$
$dS^{a}_{a\wedge b} = (a \oplus (a \wedge b)) \wedge \frac{dS}{dA}$
$= (a \wedge \bar{b}) \wedge \frac{dS}{dA}$
And stuck-at-0 fault replaces a clause with 0; stuck-at-1 fault replaces it with 1.
$dS^{a\vee b}_{a\vee 0} = (a \vee b) \oplus (a \vee 0) \wedge \frac{dS}{dA}$
$= (a \wedge \bar{b}) \wedge \frac{dS}{dA}$

$dS^{a\wedge b}_{a\wedge 1} = (a \wedge b) \oplus (a \wedge 1) \wedge \frac{dS}{dA}$
$= (a \wedge \bar{b}) \wedge \frac{dS}{dA}$
Then
$dS^{a}_{a\wedge b} \rightarrow dS^{a\vee b}_{a\vee 0}$ holds. So if any test case that CIF for a clause in a predicate will also detect the stuck-at-0 and stuck-at-1 faults for the same clause.

The above relationship between LRF, CIF, CNF and ENF is shown in figure 1. This relationship between fault classes implies that the number of tests needed is much less, because of overlap between detection conditions.

Figure1 depicts the relationships between fault classes



V.CONCLUSION

Software testing is the most costly phase in the software development and efforts should be made to reduce the cost. The proposal made in the paper helps in reducing the test cases. By extending the fault hierarchy, we have been provided with to detect a corresponding fault from a class, thus improving the effectiveness of fault based testing. The technique presented in this paper helps us for detection of faults with the minimum numbers of test case thus reducing the overall software testing cost.

REFERENCES

[1] D. R. Kuhn, "A technique for analyzing the effects of changes in formal specifications", The Computer Journal 35 (6) (1992) 574–578.
[2] D. R. Kuhn, "Fault classes and error detection in specification based testing", ACM Transactions on Software Engineering Methodology 8 (4) (1999) 411–424.
[3] H. D. Mills, "On the statistical validation of computer programs", Software Productivity, Little, Brown, Boston, 1983, pp. 71–81.
[4] J. Offutt, Y. Xiong, S. Liu, Criteria for generating specification-based tests, Proceedings of the Fifth IEEE Fifth International Conference on Engineering of Complex Computer Systems (ICECCS '99), IEEE Computer Society Press, Las Vegas, NV, 1999, pp. 119–131
[5] K.-C. Tai, Theory of fault-based predicate testing for computer programs, IEEE Transactions on Software Engineering 22 (8) (1996) 552–562.
[6] M. F. Lau, Y. T. Yu, "On the relationships of faults for Boolean specification based testing", Australian Software Engineering Conference, IEEE CS Press, 2001, pp. 21–28.
[7] P. E. Ammann, P. E. Black, and W. Majurski. "Using model checking to generate tests from specifications" Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98), pages 46–54. IEEE Computer Society, Dec. 1998.
[8] P. E. Black, Vadim Okun, Yaacov Yesha, "Mutation operators for specifications", 15th IEEE International Conference on Automated Software Engineering (ASE2000), IEEE Computer Society, Grenoble, France, 2000, pp. 81–88 .
[9] T. Tsuchiya, T. Kikuno, "On fault classes and error detection in specification based Testing", ACM Transactions on Software Engineering Methodology 11 (1) (2002) 58–62.
[10] Vadim Okun , Paul E. Black, Yaacov Yesha," Comparison of Fault Classes in Specification-Based Testing" Elsevier Science, 2 April 2