Extending the fault classification hierarchy for software with relational operators

Jeonghyun Kim[1], Kyunghee Choi[1] and Gihyun Jung[2]
[1]Graduate School of Information and Communication, [2]School of Electrics Engineering
Ajou University
430, San-hak hall, Ajou university, suwon-city
South Korea

Abstract: - Testing is a simple and direct way for making software more reliable. Many specification-based software testing mechanisms utilize test points generated based on fault hierarchies, classes of which are categorized by common faults frequently introduced during software implementation. The testing method is effective to identify Boolean faults but has a severe drawback, which is not applicable to testing software with relational operators such as “>” or “!=”. This paper carefully investigates the characteristic of relational operator faults and proposes a unique way to find relational operator reference faults in software. Its feasibility is also shown by applying the proposed method to a HVAC (Heating/Ventilating/Air-Conditioning) system of commercial vehicle.

Key-Words: - Specification-based test, Boolean fault, relational operator reference fault. Fault classes hierarchy

1 Introduction

Testing is an important part for improving the quality of software and takes a key role for improving the quality as softwares include more diverse functions both de-signers and customers want. Testing becomes more critical when faults may cause fatal results like brake control failure in vehicle control mechanism. In general, software programs include various expressions to implement their functionalities. Boolean expression is one of the basic expressions frequently used in software. Most software programming languages from machine-level to high level languages like C, C++ or JAVA define and use Boolean expressions to implement such requirements. Moreover, in specification-based software testing methodology, Boolean expressions including relational expressions are frequently imported in specification models which describe the functionality of software. Thus to improve software reliability, it is inevitable to test Boolean expressions in software. Many studies have been done for identifying faults in Boolean expressions, classifying the faults frequently presented in Boolean expressions and generating test points to detect the faults based on the classification.

Authors of [3] and [6] proposed fault hierarchies for classifying Boolean faults such as *Expression Negation Fault, Variable Negation Fault, Variable Reference Fault* and *Missing Condition Fault*, based on fault detection conditions. They proved that the test points to detect faults in a higher class can detect faults in lower classes. Weyuker EJ and Goradia T, Singh A [2] suggested test vector generating strategies that can successfully detect the faults in Boolean expressions embedded in softwares using Unique True Points (UTP) and Near False Points (NFP). M. F. Lau and Y. T. Yu [4],[8] revealed the relationship between eight Boolean fault classes. And T. Y. Chen and M. F. Lau proposed test point generation strategies, CUTPNFP, MUTP and MNFP, based on the fault hierarchy in [8],[9]. The test points can efficiently detect *Literal Reference Faults* (LRF) and *Literal Insertion Faults* (LIF) in irredundant *Disjunctive Normal Form* (DNF) of Boolean expressions. Vadim Okun, Paul E. Black and Yaacov Yesha [7] also proposed a strategy to detect the faults of expressions in arbitrary form as well as DNF.

Few previous works have suggested solutions to detect faults in Boolean expressions with relational operators such as “greater than” or “not equal”. Considering that most practical softwares contain many relational operators, it is important to have strategies to detect the faults introduced by relational operators.

This paper proposes a strategy to find Relational Operator Reference Faults (RORF) in Boolean expressions. The proposed strategy also present a way to generate test points, based on UTP and NFP as in [2] and detect RORF using the test points. Since the proposed strategy uses UTP and NFP as CUTPNFP, MUTP and MNFP do, the extra overhead to generate test points is not significant and the number of additional test points is not large either.

In the Second section, the notations to be used throughout this paper and the previous works are summarized. Third section defines relational operator reference fault that is the problem we try to solve in the paper. The method to detect RORF is suggested in Section four. How the suggested method can be applied to real systems is shown through a small sample system in the section. The final section wraps up this paper.

2 Notations and previous works

This section summarizes the notations to be utilized in the paper. As mentioned before, we focus on software with Boolean expressions and thus the notations we will summarize are all about Boolean expressions. A *variable* represents an unknown quantity that has the potential to change. A *literal* is an occurrence of a Boolean variable or its negation in an expression. A Boolean variable is expressed as either *positive* or *negative* literal. For example, a Boolean expression $S = a \cdot b + \overline{a} \cdot c$ has three Boolean variables, a, b, c. And a and $\overline{a}$ are positive literal and negative literal. *Disjunctive Normal Form* (DNF) is one of standard Boolean expressions. What an expression is in DNF means that the expression is an OR sequence of conjunctions of literals. For example, an expression S $= p_1 + p_2 \cdots + p_{m-1} + p_m$ is one in DNF where $p_i = x_1 \cdot x_2 \cdot \ldots . \cdot x_n$ and x_i is a literal, where $n \geq 1$. In the case, we say S has *m* terms. DNF expression may not be unique for an expression. An *irredundant DNF* expression is one that if any one of literals in a Boolean expression is removed or modified, the meaning of expression is changed. A Boolean expression has a unique *irredundant DNF* [5] and a Boolean expression can be translated into an equivalent Boolean expression in irredundant disjunctive normal form.[10][11]

A set of *test points* for S is an assignment of values to variables that make S 'True' or 'False' and are used to verify whether S is correct or wrong. Test points that make S true are called *True Points* (TP) and ones that make S false are called *False Points* (FP). If a TP makes the ith term of an Boolean expression true but all other terms except the ith term false, we call the TP a *Unique True Point* (UTP). The meaning of $UTP_i(S)$ is combination(s) of literals that makes p_i true but all other terms except p_j false in S. The number of literal combinations of $UTP_i(S)$ may be zero or multiple if S is of DNF. For example, for S = abc + de, $UTP_1(S)$ = {(11100), (11101), (11110)}. *Overlapping True Points* (OTP) are TP's except UTP's.

False points (FP) are test points that make S false. *Near False Points*, $NFP_{i,j}(S)$ are defined as false points such that only the ith term becomes true but all other terms except the ith term stay false when the jth literal of the ith term is negated. For example, $NFP_{1,2}(S)$ for a system, S = abc + de, are {(10100), (10101), (10110)}. *Remaining False Point* (RFP) are FP's except NFP's.

Man F. Lau and Yuen T. Yu of [8] addressed that the typical faults introduced by programmers during programming step are 'incorrect operators or operands', 'missing extra conditions and paths' and 'incorrect control predicates'. And they classified the faults in views of 'omission', 'insertion' and 'incorrect reference' of Boolean operands or operators. Kuhn [3], Chan and Lau [5] and Tsuchiya and Kikuno[6] similarly classified the faults. Lau and Yu classified faults of Boolean expressions in DNF into eight classes as follows;

1) *Express Negation Fault* (ENF):
   Fault introduced by negating S or multiple terms
2) *Term Negation Fault* (TNF):
   Fault introduced by negating a term
3) *Literal Negation Fault* (LNF):
   Fault introduced by negating a literal of a term
4) *Term Omission Fault* (TOF):
   Fault introduced by omitting a term
5) *Operator Reference Fault* (ORF):
   Fault introduced by exchanging '+' with '·'
   or vice versa.
6) *Literal Omission Fault* (LOF):
   Fault introduced by omitting a literal of a term
7) *Literal Insertion Fault* (LIF):
   Fault introduced by inserting a literal into a term
8) *Literal Reference Fault* (LRF):
   Fault introduced by replacing a literal of a term
   with a different one

Man F. Lau and Yuen T. Yu showed that the eight fault classes can construct a fault hierarchy. According to the hierarchy, the faults in higher classes can be detected by the test points by which the faults in lower fault classes can be detected. For example, if the faults belonging to LIF are detected by a test point set, the test point set can also detect the faults in TOF, which is a higher fault class. A test point set for detecting the faults in LRF can also detect the faults in LNF, which is a higher fault class than LRF.

In paper [5], T. Y. Chen and M. F. Lau proved that UTP and NFP can detect all faults in the eight fault classes if a software consists of Boolean variables and Boolean operators and is of irredundant DNF. They proposed a method to produce test points for finding faults in the lowest classes of hierarchy: LIF, LRF and LOF. Since the test points can detect the faults in the lowest classes, they can also find all faults in higher classes.

But unfortunately the fault class hierarchy and the test point generation method are not applicable to software with relational operators. Meanwhile many real softwares include various relational operators. Thus for the hierarchy and the test point generation method to be more practical, it is inevitable to extend them to be able to deal with relational operators.

3 Presenting relational expression

The expression of relational operations to be dealt in this paper is "***var1 op var2***", where ***var1*** is a variable, ***op*** is one of relational operators among $\{ >, \geq, <, \leq, =, \neq \}$, and ***var2*** is another variable. Here, what we mean a *Relational Operator Reference Fault* (RORF) of a Boolean expression, S, is the fault by replacing a relational operator in S with a different relational operator.

A relational expression "***var1 op var2***"can be considered as a literal in the view that "***var1 op var2***" returns 'True' or 'False' when substituting ***var1*** and ***var2*** with proper test points as a literal does. For finding TP and FP and RORF, we will use modified software expression S_{mod} instead of S, where S_{mod} is exactly same as S except replacing a relational expression in S with a literal that is not included in the literal list of S.

TP and FP of S_{mod} are found by the exactly same manner for those of S. Assume we have a Boolean expression $S = a \cdot (k > 5) + c$. If relational operator $(k > 5)$ is replaced with (arbitrarily chosen) 'b', the modified system S_{mod} is expressed as $S_{mod} = a \cdot b + c$. The true points of S_{mod} are (1,1,1), (1,1,0), (0,0,1), (0,1,1) and (1,0,1). Here, the input '1' of 'b' corresponds to k value greater than 5 and '0' to k smaller than or equal to 5. Table 1 and 2 show the TP and FP of S and S_{mod}.

True points of S_{mod}	True points of S
(1,1,1)	$(1, k \mid k > 5, 1)$
(1,1,0)	$(1, k \mid k > 5, 0)$
(0,0,1)	$(0, k \mid k \leq 5, 1)$
(0,1,1)	$(0, k \mid k > 5, 1)$
(1,0,1)	$(1, k \mid k \leq 5, 1)$

Table 1. True points of S and S_{mod}

False points of S_{mod}	False points of S
(0,0,0)	$(0, k \mid k \leq 5, 0)$
(1,0,0)	$(1, k \mid k \leq 5, 0)$
(0,1,0)	$(0, k \mid k > 5, 0)$

Table 2. False points of S and S_{mod}

The authors of [5] showed that if a software S is of irredundant DNF, there are at least one UTP(S) and NFP(S). If S is of irredundant DNF, then S_{mod} is also of irredundant DNF. Thus S_{mod} has at least one $UTP_i(S_{mod})$ and $NFP_{i,j}(S_{mod})$. An expression with a literal replaced for a relation expression has infinite number of $NFP_{I,j}(S)$ and $UTP_i(S)$ since there infinite number of values that make S true or false.

4 Strategy for detecting RORF (MUNC algorithm)

In this section, a strategy called *Multiple Unique true points and Near false points Combination* (MUNC) is proposed for detecting RORF in Boolean expressions that are of irredundant DNF and contain relational expressions. A blower speed control module in a HVAC system is utilized as a test system for verifying the feasibility of proposed strategy.

4.1 MUNC algorithm

Let $S = p_1 + \cdots + p_i + \cdots + p_m$ be a Boolean expression of irredundant DNF with relational expressions. The relational operation we are dealing with is "*var1 op var2*". Here we assume that *op* is one of six relational operators, which are '>' , '=' , '<', '≥' , '≠' , '≤'. Let a term p_i include a relational expression "*var1 op var2*" at the jth literal position of p_i term. That is, $p_i = x_1 \cdot x_2 \ldots x_{j-1} \cdot (var1\ op\ var2) \cdot x_{j+1} \ldots x_m$, where $1 \leq j \leq m$. After substituting the jth relational expression with a literal x that is not included in the literal list of S, we have the modified S, $S_{mod} = p_1 + \cdots + q_i + \cdots + p_m$, where $q_i = x_1 \cdot x_2 \cdots x_{j-1} \cdot x \cdot x_{j+1} \cdots x_m$. x has either 'True' or 'False' depending on the non-algebraic values of *var1* and *var2* in S. If there are multiple terms including relational operations in S and the relational operations are independent of each other, S_{mod} can be driven from S in the same manner.

Since S_{mod} is an Boolean expression in irredundant DNF form, we can find $UTP_i(S_{mod})$ and $NFP_{i,j}(S_{mod})$ [5]. By replacing *x* with the corresponding values of *var1* and *var2*, we can also find $UTP_i(S)$ and $NFP_{i,j}(S)$. If it is possible to identify *op* in the term by using the values of *var1* and *var2*, $UTP_i(S)$ and $NFP_{i,j}(S)$ can be used to find both the typical eight types of faults [8] and RORF in Boolean expressions.

Input values of *var1* / **Relational Operation**	*Var1 = var2 – α*	*var1 = var2*	*var1= var2+ α*
var1 > var2	False	False	True
var1 ≤ var2	True	True	False
var1 = var2	False	True	False
var1 ≠ var2	True	False	True
var1 < var2	True	False	False
var1 ≥ var2	False	True	True

Table 3. Test results in relational operations

Table 3 illustrates that each of the relational operations with the six relational operators returns a unique combination of 'True" and 'False' by substituting *var1* with *var2*- α, *var2* and *var2*+ α, respectively in "*var1 op var2*" in S, where *α* is an arbitrary positive constant. For example, if "*var1 = var2"* (in the second row) is correctly implemented, the expression must return "False", "True" and "True" when *var1* is substituted with *var2*- α, *var2* and *var2*+ α, respectively. The return value combination is different from other return values. That is, by substituting *var1* values with *var2*- α, *var2* and *var2*+ α, it is possible to identify *op* in the relational operation.

If the relational operator {one of '>' , '=' , '<', '≥' , '≠' , '≤'} hidden in literal *x* in S_{mod} is replaced with a different one, a RORF is introduced. The RORF can be detected by the test points in the following four Summaries.

Summary 1. If either '>' or '<' is replaced with a different one out of {'>', '<', '=' , '≥' , '≠' , '≤'}, then the test points to detect the RORF are set of {t_{11}, t_{12}, t_{13}} such that "$t_{11} \in UTP_i(S)$", "$t_{12} \in NFP_{i,j}(S)$, where *var1 = var2*(i.e. *var1* and *var2* have the same input value) ", "$t_{13} \in NFP_{i,j}(S)$, where $t_{13} \neq t_{12}$".

Proof) By the definition of UTP and NFP, the test point set, "$t_{11} \in UTP_i(S)$", "$t_{12} \in NFP_{i,j}(S)$, where *var1 = var2*" and "$t_{13} \in NFP_{i,j}(S)$, where $t_{13} \neq t_{12}$" satisfy the input values of *var1* of the first row (*'var1 > var2'*), which are *'var1 = var2 + α', 'var1 = var2'* and *'var1 = var2 – α'*, respectively in Table 3. And also the test point sets satisfy the input conditions of the fifth row (*'var1 < var2'*), which are *'var1 = var2 – α', 'var1 = var2'* and *'var1 = var2 + α'*.

Meanwhile, *var1 < var2* and *var1 > var2* return outputs ('false', 'false', 'true') and ('true', 'false', 'false') to input values of *'var1 = var2 – α', 'var1 = var2'* and *'var1 = var2 + α'*. And the outputs are different from any other output combinations in Table 3. It means that the input values of *var1* can identify '>' and '<' from other relational operators.

Consequently, test points sets "$t_{11} \in UTP_i(S)$", "$t_{12} \in NFP_{i,j}(S)$, where var1 = var2" and "$t_{13} \in NFP_{i,j}(S)$, where $t_{13} \neq t_{12}$" can detect the RORF introduced by replacing either '>' or '<' with a different one out of {'>', '<', '=' , '≥' , '≠' , '≤'}.

Summary 2. If either '≥' or '≤' is replaced with a different one out of {'>', '<', '=' , '≥' , '≠' , '≤'}, then the test points to detect the RORF are set of {t_{21}, t_{22}, t_{23}} such that "$t_{21} \in UTP_i(S)$", "$t_{22} \in UTP_i(S)$, where *var1 = var2* and $t_{22} \neq t_{21}$", "$t_{23} \in NFP_{i,j}(S)$".

Proof) Same as Summary 1 except test point sets.

Summary 3. If either '=' is replaced with a different one out of {'>', '<', '≥' , '≠' , '≤'}, then the test points to detect the RORF are set of {t_{31}, t_{32}, t_{33}} such that "$t_{31} \in UTP_i(S)$, where *var1 = var2* "$t_{32} \in NFP_{i,j}$ (S), where *var1 > var2*", "$t_{33} \in NFP_{i,j}(S)$, where *var1 < var2*".

Proof) Same as Summary 1 except test point sets.

Summary 4. If either '≠' is replaced with a different one out of {'>', '<', '≥' , '≠' , '≤'}, then the test points to detect this RORF are set of {t_{41}, t_{42}, t_{43}} such that "$t_{41} \in UTP_i(S)$, where *var1 > var2*", "$t_{42} \in UTP_{i,}(S)$, where *var1 < var2*", "$t_{43} \in NFP_{i,j}(S)$, where *var1 = var2*".

Proof) Same as Summary 1 except test point sets.

The number of test points for MUNC to detect a

RORF in expression with *n* Boolean variables and *m* terms with at most *r* relational expressions in each term is bounded by 3*mr*. This is because each relational expression needs either (two $UTP_i(S)$ and one $NFP_{ij}(S)$) or (one $UTP_i(S)$ and two $NFP_{i,j}(S)$). And to find either $UTP_i(S)$ or $NFP_{ij}(S)$, we need *mr* test points in maximum.

When the proposed MUNC strategy is used along with MUTP, the test point generation strategy proposed by Chen to detect RORF, the extra test points required to detect RORF is *2mr* in maximum since MUTP has to generate *mr* test points for $UTP_i(S)$. MUNC needs extra *mr* test points when it is used along with CUTPNFP strategy that generates 2*mr* test points for $UTP_i(S)$ and $NFP_{i,j}(S)$. In the case that MUNC is used with MNFP, it needs to generate 2*mr* test points since MNFP requires *mr* test points for $NFP_{i,j}(S)$.

MUTP, CUTPNFP and MNFP have to be applied together to detect the typical eight types of faults in Boolean expression. For them, 2*mr* test points are required. Thus the maximum number of required test points needed for detecting RORF by the proposed MUNC is bound by *mr* when they are used to detect the typical eight types of faults and RORF.

4.2 Applying the proposed algorithm to a simple blower speed control module

Here is a simple example system to show how efficiently the proposed algorithm works. A simple work model of 'Blower Speed Control Module' in a 'HVAC system' for commercial vehicle is depicted in Fig 1. The state diagram illustrates the state transition from a state that blower speed is fast to another state that blower speed is slow.

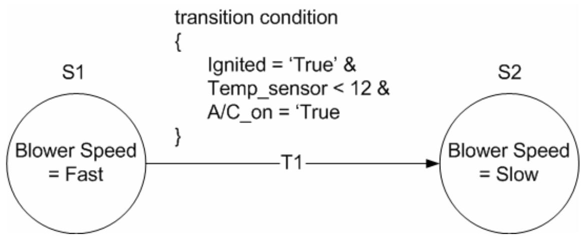

Fig. 1. Blower speed state transition in HVAC system

Ignited is a flag to indicate vehicle ignition state and *Ignited = 'True'* presents that the vehicle engine is running. *A/C_on* is also a flag to indicate operating state of air-conditioner and A/C_on = 'True' presents that the air conditioner is running. *Temp_sensor* has a numerical number to indicate the temperature of ambient air in degree (℃).

At the condition that a vehicle is ignited, air conditioner is on and the vehicle temperature is below 12 degree, the blower speed goes from fast to slow. The model can be coded as follows;

```
If (Current Blower Speed = Fast){
  If (Ignited & (Temp_Sensor < 12) & A/C_on){
            New Blower Speed = Slow
}}
```

The above specification from fast blower speed to slow blower speed, S, can be written in Boolean form expression: S = *Ignited* · (*Temp_Sensor* < 12) · *A/C_on*. Let substitute variables '*Ignited*', '*Temp_Sensor* < 12' and '*A/C_on*' with 'a', 'b' and 'c' for convenience. Note that relational expression '*Temp_Sensor* < 12' has been substituted by a literal 'b'. Then S_{mod} becomes $S_{mod} = a \cdot b \cdot c$.

In the case that a RORF is introduced in S by accidentally replacing ''*Temp_Sensor* < 12'' with ''*Temp_Sensor* > 12'', (that is, S has been actually implemented as S_{imp} = *Ignited* · (*Temp_Sensor* < 12) · *A/C_on* and $S_{mod} = a \cdot b' \cdot c$), the RORF can be found by a test point 't_{11} in $UTP_1(S)$', a test point 't_{12} in $NFP_{1,2}(S)$, where *Temp_Sensor* = 12', and a test point 't_{13} in $NFP_{1,2}(S)$, where $t_{12} \neq t_{13}$' according to **Summary 1**. Let choose $t_{11} = (1, 11, 1)$, $t_{12} = (1, 12, 1)$ and $t_{13} = (1, 13, 1)$. Then $S(t_{11})$, $S(t_{12})$ and $S(t_{13})$ should be 'True', 'False' and 'False'. But they force the implemented system to return 'False', 'False' and 'True'. The test outputs in implemented system are different from what should be. Consequently, we can say that there is a relational fault in the implemented system. The position where the fault appears can be found by UTP and NFP. Since the (incorrectly) implemented S has a fault with $UTP_1(S_{imp})$ and $NFP_{1,2}(S_{imp})$, we know that the fault occurs at the 2nd literal of the 1st term. The literal corresponds to a relational operation is '*Temp_Sensor* < 12'.

5 Conclusion

We proposed a strategy to overcome the drawback of method by T. Y. Chen and M. F. Lau. Their method only detects the typical eight faults of software with only Boolean operators and Boolean variables. Our proposed method can detect relational operator reference faults, which are easily observed in many real applications. The proposed strategy identifies Relational Operator Reference Faults through unique combinations of outputs produced by relational operators. The feasibility of the proposed strategy is shown in a small application. The proposed method can be used to detect RORF as well as the faults in the typical eight fault classes frequently shown in software when it is applied to software with other well known test point generation strategies such as MUTP, CUPNFP and MUTP.

Based on the previous works and our proposed strategy, the further work has been progressed. The future work includes finding methods to detect other software faults such as more complicated relational expression faults, associative shift faults and stuck-at faults.

References:

[1] Kuo-Chung Tai, Mladen A. Vouk, Amit M. Paradkar, Peng Lu, “Evaluation of a Predicate-Based Software Testing Strategy”, *IBM Systems Journal*, 33(3):445-457, 1994.

[2] Weyuker EJ, Goradia T, Singh A, “Automatically generating test data from a Boolean specification”, *IEEE Transactions on Software Engineering*, 20(5):353-363, 1994.

[3] D. Richard Kuhn, “Fault Classes and Error Detection Capability of Specification Based Testing”, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 8(4):411 - 424 , 1999.

[4] M. F. LAU, Y. T. Yu, “On the Relationships of Faults for Boolean Specification Based Testing”, *Australian Software Engineering Conference*, 2001, pp. 21-30

[5] T. Y. Chen and M. F. Lau, "Test point selection strategies based on Boolean specifications", , *Journal of Software Testing, Verification and Reliability*, 11(3):165-180, 2001.

[6] Tatsuhiro Tsuchiya and Tohru Kikuno, “On Fault Classes and Error Detection Capability of Specification-Based Testing”, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1):58-62, 2002.

[7] Vadim Okun, Paul E. Black, and Yaacov Yesha, “Comparison of Fault Classes in Specification-Based Testing”, *Information and Software Technology*, Elsevier, 46(8):525-533, 2004.

[8] Man F. Lau and Yuen T. Yu, “An extended fault class hierarchy for specification-based testing”, *ACM Transactions on Software Engineering and Methodology (TOSEM)*”, 14(3):247-276, 2005.

[9] T. Y. Chen, M. F. Lau and Y. T. Yu, "MUMCUT: a fault-based strategy for testing Boolean specifications", *In Proceedings of Asia-Pacific Software Engineering Conference* 1999, pp. 606-613, December 1999, IEEE CS Press.

[10] V. Gurvich, L. L. Khachiyan, “On generating the irredundant conjunctive and disjunctive normal forms of monotone Boolean functions”, *Discrete Applied Mathematics*, volume 96-97, Issue 1, pp. 363-373, October 1999.

[11] Quine WV. “The problem of simplifying truth functions”, *America Mathematical Monthly* 1952; 59:521-531