Postprint of article in *Proceedings of the International Computer Symposium* (*ICS '04*),
Taipei, Taiwan, pp. 926–932 (2004)

An Empirical Evaluation and Analysis of the Fault-Detection Capability of MUMCUT for General Boolean Expressions[*] [+]

Chang-ai Sun [1], Kwan Yong Sim [2], T.H. Tse [3], and T.Y. Chen [4]

[1, 4] *School of Information Technology*
Swinburne University of Technology
Melbourne, VIC 3122, Australia
{csun, tychen}@swin.edu.au

[2] *School of Engineering*
Swinburne University of Technology
93576 Kuching, Sarawak, Malaysia
ksim@swinburne.edu.my

[3] *Department of Computer Science*
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

Abstract *Boolean expressions are extensively used in software specifications. It is important to generate a small-sized test set for Boolean expressions without sacrificing the fault-detection capability. MUMCUT is an efficient test case generation strategy for Boolean expressions in Irredundant Disjunctive Normal Form (IDNF). In the real world, however, Boolean expressions written by a software designer or programmer are not usually in IDNF. In this paper, we apply MUMCUT to generate test cases for general Boolean expressions and develop a mutation-based empirical evaluation of the effectiveness of this application. The experimental data show that MUMCUT can still detect single seeded faults in up to 98.20% of general Boolean expressions. We also analyze patterns where test cases generated by MUMCUT cannot detect the seeded faults.*

Keywords: Test Case Generation, Boolean Specifications, Software Testing

1. Introduction

Boolean expressions are extensively used to represent the decisions/conditions in a specification or program. It is important to check whether they are implemented correctly for the purpose of quality assurance, since they play an important role in the specification or program.

In the last decade, a lot of research work has been devoted to the testing on Boolean expressions, logic formulas, and predicates [8]. The work on the testing of Boolean expressions can be divided into two major categories: structural approach and fault-based approach.

In the structural approach, the basic idea is to generate a test suite to cover the elements of decision according to the coverage criteria [16]. From the perspective of coverage, structural Boolean expression-oriented testing can be further classified into decision coverage, condition coverage, decision/condition coverage, and path coverage [9]. In decision coverage, for example, test cases are generated so that every program decision has taken the values True and False.

In fault-based testing, test cases are generated to detect specified types of fault in a specification or program [2, 10, 15]. There have been studies related to fault-based testing of Boolean expressions. Weyuker et al. [15] proposed a meaningful impact strategy for testing Boolean formulas in Irredundant Disjunctive Normal Form (IDNF). Offutt and Liu

[+] This research is supported in part by an ARC Discovery Grant (Project No. DP0345147) and a grant of the Research Grants Council of Hong Kong (Project No. CityU 1083/00E).

[10] proposed a procedure to generate test data for SOFL specification where conditions or predicates are assumed to be in Disjunctive Normal Form (DNF). Kuhn [8] described a method for computing the conditions that must be covered by a test set for the test set to guarantee detection of the particular fault class. Tatsuhiro [13] improved on Kuhn's fault hierarchy. Chen and Lau [2] proposed a set of more efficient test case generation and selection strategies for Boolean specifications in IDNF.

We observe that existing fault-based testing techniques are developed for Boolean expressions in DNF or IDNF. In the real world, however, Boolean expressions written by a designer or programmer are usually in the General Form (GF, also called arbitrary form [11]). In additional, we have empirically evaluated fault relationships between GF and IDNF in a companion study [5]. The result shows that, in up to 75.7% of the cases, one fault in GF can result in more than one fault in its equivalent IDNF. Hence, it is of great interest to see how efficient MUMCUT will be for Boolean expressions in GF. We can almost draw a conclusion that there is a big gap between the existing research and realistic situations. This also restricts the applicability of the existing work.

Our main motivation in this paper is to investigate how efficient MUMCUT is when applied to general Boolean expressions. MUMCUT [2] is a fault-based test case generation strategy for Boolean expressions in IDNF, and the test suite generated by MUMCUT can detect seven single faults in a Boolean expression in IDNF. For general Boolean expressions, we want to see whether a test suite generated by MUMCUT can still detect a seeded single fault and empirically evaluate the fault detection capability of MUMCUT.

The rest of this paper is organized as follows: Section 2 introduces MUMCUT and related techniques. Section 3 presents the approach to applying MUMCUT, and Section 4 introduces an empirical evaluation of the effectiveness of MUMCUT. Section 5 introduces the related work and Section 6 concludes the paper.

2. Background

In this section, we introduce the basic concepts and previous work related to this research.

Test Cases and Test Case Adequacy

In software testing practice, testers are required to generate test cases to execute the program. A *test case* is an input on which the program under test is executed during testing. A *test set/suite* is a set of test cases for testing a program [16]. Given a test criterion, we can judge whether a test set is adequate. For Boolean expressions, a test case is an ordered truth-value list where each value is the assignment of a Boolean variable. For example, consider a Boolean expression B1 = "$a + b$", a test case for B1 is "$a = 0$, $b = 0$", and the complete test suite for B1 is {00, 01, 11, 10}. If the decision coverage criterion is used, {00, 10}, {00, 11}, and {00, 10} are the adequate test sets.

Fault-based testing and Mutation Analysis

Software testing aims at ultimately detecting faults in a program [16]. If it is assumed there are some specific fault types in a specification or program and if test cases are generated to detect these faults, the approach is called *fault-based testing*.

In fault-based testing, *mutation analysis* [6] is widely used to verify the adequacy of a test suite based on the specific testing criteria. Given a Boolean expression B, a derivation M is obtained by seeding faults into B. M is called a *mutant* of B and the process to obtain M from B is called *mutation*.

In our experiment, mutation technique is used to derive the Boolean expression mutants. Each mutant contains only one fault when compared with the original Boolean expression. The faults studied in this paper include Expression Negation Fault (*ENF*), Literal Negation Fault (*LNF*), Term Omission Fault (*TOF*), Term Negation Fault (*TNF*), Operator Reference Fault (*ORF*), Literal Omission Fault (*LOF*), Literal Insert Fault (*LIF*), and Literal Reference Fault (*LRF*)[1].

IDNF Transformation

Given a Boolean expression, it can be represented in several forms. A *Boolean expression in DNF* is formed by the disjunctive terms, and a disjunctive term can be formed by the conjunctive literals (Boolean variables). A Boolean expression in DNF is said to be *in IDNF* when none of the Boolean literals or terms can be deleted without altering the value of the Boolean expression for some test cases [2, 15].

Given a general Boolean expression, it can always be transformed into an equivalent one in DNF using laws such as the distributive or commutative law. A Boolean expression in DNF can also be transformed into an equivalent one in IDNF using the algorithm in [7]. For example, a Boolean expression in GF

"$a * (!c + !b + !d) + !a * (c + b + d)$
$+ b * (!c + !d) + !b * (c + d) + !c * d + c * !d$"

can be transformed into

"$a * !c + !a * c + a * !b + b * !c + !a * b$
$+ !b * c + !a * d + !b * d + !c * d + a * !d$
$+ b * !d + c * !d$"

in DNF. It can also be transformed into two different equivalent IDNFs

[1] The eight types of faults are defined in the context of GF, unlike the seven types of faults defined in [2] for IDNF.

"!a * d + !b * d + !c * d + a * !d + b * !d + c * !d"

and

"!a * d + !b * d + b * !d + c * !d + a * !c".

MUMCUT

MUMCUT is a fault-based test case selection strategy for generating test cases from Boolean specifications proposed by Chen and Lau [2]. It integrates the MUTP strategy [4], the MNFP strategy [2], and the CUTPNFP strategy [3]. These strategies were developed based on two important concepts, namely, *Unique True Point* (UTP) and *Near False Point* (NFP) [15]. Accordingly, UTP and NFP are defined on Boolean expressions in IDNF. Compared with the existing test case selection strategies for Boolean expression in IDNF, the MUMCUT strategy is more efficient. For example, MAX-A and MAX-B are two strategies developed by Weyuker et al. [15] based on UTP and NFP. Under the assumption that only one fault is introduced into the implementation, the empirical research shows that the MUMCUT strategy uses on the average 74% and 67% of the test cases required by the MAX-A and the MAX-B strategies, respectively [2].

Companion studies

Recognizing that the realistic Boolean expressions are more often written in GF (that is, the arbitrary form), we have developed an algorithm and a tool to generate a benchmark of Boolean expressions in GF for experimentation [12]. We have also conducted empirical research on the fault relationships between GF and IDNF [5]. More precisely, given one fault in a Boolean expression in GF, we have investigated statistically the number of faults in its equivalent IDNF. We shall not elaborate on the technical details of the companion studies, as they are not directly relevant to the present paper.

3. Applying MUMCUT to Test General Boolean Expressions

Given a Boolean expression in arbitrary form, we first apply an algorithm to transform it into IDNF, then use MUMCUT to generate a set of test cases, and finally use these test cases as a test suite to detect specific faults in the Boolean expression in arbitrary form. End users do not need to know about this process. All they need is to provide Boolean expressions in arbitrary form and to apply the test cases generated by the MUMCUT strategy.

Nevertheless, MUMCUT is not always effective in the context of general Boolean expressions. For a Boolean expression in IDNF $\boldsymbol{B}$ and a test suite $\boldsymbol{T}$ generated by MUMCUT, one can guarantee that all seven single types of fault mentioned in [2] can be detected when $\boldsymbol{T}$ is used as test cases for $\boldsymbol{B}$. For general Boolean expressions, however, one fault will result in simultaneous occurrence of several faults in its equivalence in IDNF. In this situation, the generated test cases satisfying MUMCUT cannot guarantee the detection of all faults introduced into a Boolean expression in IDNF.

4. Empirical Evaluation

In this section, we present an empirical evaluation of the fault detection capability of MUMCUT in the context of general Boolean expressions.

4.1 Principle

We statistically check whether one fault in general Boolean expressions can be detected when a set of test cases generated by MUMCUT (also called *MUMCUT-adequate test cases*) are used. From the perspective of mutation, it is equal to whether the mutants with one fault can be distinguished from the original Boolean expression under these test cases. If answer is yes, we say that the mutant is *killed*.

Given a Boolean expression $\boldsymbol{O}$, a mutant $\boldsymbol{M}$, and a test suite $\boldsymbol{TS} = \{t_1, t_2, \ldots, t_n\}$, we say that $\boldsymbol{M}$ is killed if and only if at least one test case $\boldsymbol{t_i}$ in $\boldsymbol{TS}$ can differentiate the truth values of $\boldsymbol{M}$ and $\boldsymbol{O}$, denoted by *Killed*($\boldsymbol{M}$, $\boldsymbol{O}$, $\boldsymbol{TS}$). In other words,

$$\exists t_i \in TS: \quad O(t_i) \neq M(t_i)$$

Given a Boolean expression $\boldsymbol{O}$, a mutant $\boldsymbol{M}$, and a MUMCUT-adequate test suite $\boldsymbol{TS}$ ($t_1, t_2, \ldots, t_n$) generated based on $\boldsymbol{M}$, we say that *MUMCUT is effective for this mutant* if and only if at least one test case $\boldsymbol{t_i}$ from $\boldsymbol{TS}$ can kill the mutant $\boldsymbol{M}$ with $\boldsymbol{O}$ as the test oracle.

In this experiment, we employ mutation technique to obtain a mutant $\boldsymbol{M}$ of the original Boolean expression in GF $\boldsymbol{O}$. If one or more test cases from the generated test suite $\boldsymbol{TS}$ can distinguish the difference between $\boldsymbol{O}$ and $\boldsymbol{M}$ (that is, $\boldsymbol{M}$ is killed when $\boldsymbol{O}$ is used as the test oracle), we say that MUMCUT is effective. Figure 1 shows how to evaluate whether a MUMCUT-adequate test suite $\boldsymbol{TS}$ can detect a fault in a Boolean expression in GF $\boldsymbol{M}$ (that is, kill $\boldsymbol{M}$) with $\boldsymbol{O}$ as the test oracle.

The critical steps in the experiment include:

1) *Boolean Expression Samples*. To be more convictive, a large number of Boolean expressions in GF are required. We have developed a parameterized generator to construct general Boolean expressions in our companion study [12]. In this experiment, we use this tool to generate the Boolean expression samples.

2) *Mutation*. To control the sizes of mutants while satisfying mutation adequacy to a reasonable extent, we have developed a set of strategies for generating mutants [5]. Only one fault is seeded into each mutant. Eight types of faults as presented in Section 2 are considered in this experiment.

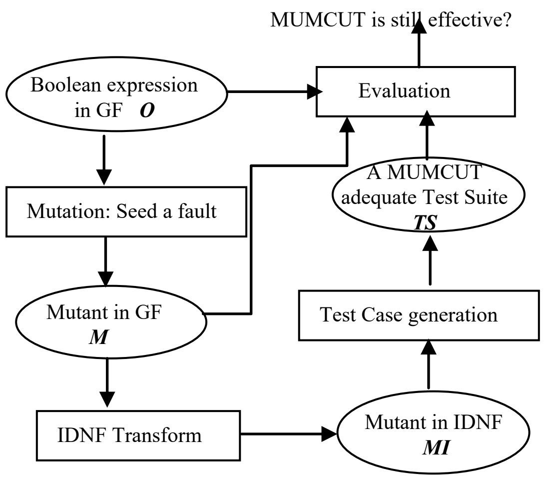

Figure 1. Mutation-based evaluation of the extension of MUMCUT

3) *IDNF Transformation*. The *Converter* [5], a tool developed by us for IDNF transformation, is used to transform Boolean expressions in GF into ones in IDNF.

4) *Test Case Generation*. Given a Boolean expression in IDNF, BEAT [1], a tool for implementing the MUMCUT strategy, is used to generate a MUMCUT-adequate test suite.

5) *Evaluation*. A MUMCUT-adequate test suite ***TS***, which is generated based on the mutant ***MI*** in IDNF, is used to evaluate whether the ***M*** can be killed with ***O*** as the test oracle. Here, ***M*** is the mutant of ***O*** and ***MI*** is the IDNF of ***M***. In the implementation, we will use ***TS*** to kill ***MI*** with ***OI*** (***OI*** is the IDNF of ***O***) as the test oracle because of the following observation:

Suppose ***O*** is a Boolean expression in GF, ***M*** is a mutant of ***O***, ***OI*** and ***MI*** are the IDNF of ***O*** and ***M***, respectively, and ***TS*** is a test suite. We have

Killed(***MI***, ***OI***, ***TS***) = *Killed*(***M***, ***O***, ***TS***)

The first three steps above are also involved in the experiment in our companion study [5]. In the present experiment, we focus on *evaluation*. We use an example to illustrate the evaluation process as follows:

1) Suppose a Boolean expression ***OG*** in arbitrary form is "!*e* * *c* + *c* * (*h* * !*g* * *i* * !*e* + *g* * !*i* * !*d*) * !*d* * !*e* + *g*".

2) Suppose an Operator Reference Fault happens to the first operator and changes it from "*" to "+". A mutant ***MG*** "!*e* + *c* + *c* * (*h* * !*g* * *i* * !*e* + *g* * !*i* * !*d*) * !*d* * !*e* + *g*" is obtained.

3) Transform ***OG*** and ***MG*** into Boolean expressions in IDNF to obtain ***O***("!*e* * c + *g*") and ***M***("!*e* + *c* + *g*").

4) Generate a MUMCUT-based test suite ***T*** *of* ***M*** using BEAT. ***T*** = {000, 010, 110, 011}.

5) Obtain t_1 = "000" from ***T*** and create the literal-assign pair table ***LA*** = {*c* = "0", *e* = "0", *g* = "0"}.

6) Evaluate the truth value ***TM*** of ***M*** with ***LA*** = ***TM*** = "1" (since !*e* + *c* + *g* = 1 + 0 + 0 = 1). Similarly, we evaluate the truth value ***TO*** of ***O*** with ***LA*** = ***TO*** = "0"(since !*e* * *c* + *g* = 1 * 0 + 0 = 0).

7) Return "True" because ***TO*** $\neq$ ***TM.***

In this example, a MUMCUT-adequate test suite kills an ORF mutant. In other words, MUMCUT is still effective in detecting the seeded fault. In some situations, however, MUMCUT may not be effective. Our experiment is to investigate the failure rate of MUMCUT and why it fails.

4.2 Settings

When a Boolean expression contains more than 12 Boolean variables, its IDNF transformation will be very slow. It will also take a long time for BEAT to generate a MUMCUT-adequate test suite for a Boolean expression consisting of more than 15 Boolean variables.

To be practical, we use the Boolean expression generator *BEGen* [12] to generate 800 samples Boolean expressions in GF as the experimental subjects. We restrict the parameter settings of *BEGen* as follows:

- The maximum number of literals is 12 (namely, the characters from *a* to *l* as positive literals and their negations such as "!*a*" as negative literals).
- The maximum number of terms in a Boolean expression is 12.
- The maximum number of literals in a term is 6.
- The seven operators are "*", "+", "!", "+ ( )", "+ !( )", "* ( )", and "* !( )".

4.3 Result and analysis

Table 1 shows the result of the empirical evaluation of the effectiveness of MUMCUT for general Boolean expressions. 800 Boolean expression samples are evenly divided into eight groups and each of them is used for one kind of fault mutation. Column 3 shows the valid mutants in a variety of mutations. Since there may be some redundant components in a Boolean expression, mutants skipping these redundant components are still equivalent to the original Boolean expression. For example, Given a Boolean expression in GF ***BG*** "!*e* + *c* + !*e* * (*h* * !*g* + *g* * !*d*) + *g*", its equivalent IDNF ***BI*** is "!*e* + *c* + *g*". Consider a mutant ***MG*** = "!*e* + *c* + !*e* * (*h* * !*g*) + *g*" of ***BG***. Compared with ***BG***, a Term Omission Fault occurs in ***MG*** (where the term "*g* * !*d*" is missing). The equivalent IDNF ***MI*** of ***MG*** is "!*e* + *c* + *g*". In this situation, ***MG*** is not a valid mutant because ***MI*** is the same as ***BI***. Furthermore, some subtly equivalent IDNFs are difficult to identify. For example, suppose ***BI*** is "!a*!*b* + *a***b* + *a***c*" and ***MI*** is "!a*!*b* + *a***b* + !*b***c*". It appears to be a Literal Reference Fault (LRF) but the two IDNFs are in fact equivalent. In our experiment, we did our best to identify and discard these *equivalent* mutants.

Table 1. Result of empirical evaluation

Mutation Type	BE Sample	Valid Mutant	Killed	Not Killed	Effectiveness
ENF	100	91	91	0	100.00%
LNF	100	775	770	5	98.97%
TOF	100	295	270	25	91.53%
TNF	100	382	377	5	98.695
LOF	100	263	261	2	99.24%
LIF	100	235	229	6	97.45%
LRF	100	336	332	4	98.81%
ORF	100	686	678	8	98.83%
Total	800	3063	3008	55	98.20%

The empirical evaluation result shows that in 98.20% situations MUMCUT is effective when it is used for the fault-based testing on Boolean expressions in GF where there is only one fault is seeded. It means that MUMCUT is still effective in testing Boolean expressions in the realistic situation. In our experiment, there are totally 55 not-killed mutants. When we further examine those instances that cannot be killed by test cases generated by MUMCUT, we discover five failure patterns as follows. Certainly, test cases generated by MUMCUT cannot detect faults in the combination of these five failure patterns either.

Pattern 1: Two-Reflective-Literal Conjunctive Term Omission without Losing Literals

Original Boolean expression: "*abc* + *AB*" → Mutant Boolean expression: "*abc*".

Pattern extension includes:

"*abcT* + !*a*!*b*" → "abc*T*" and

"(*abc* + !*a*!*b*)*T*" → "abc*T*".

Here, "*T*" is a null term or a term without the occurrence of "*a*", "*b*", or "*c*".

Pattern 2: Disjunctive Term Omission without Losing Literals

Original Boolean expression "*ab* + *bc* + *ac*" = "*ab* + *c* (*a* + *b*)" → Mutant Boolean expression "*ab* + *c*".

Pattern extension includes:

"*ab* + *c*(*a* + *b*) *T*" → "*ab* + *c T*" and

"(*ab* + *c*(*a* + *b*)) *T*" →"(*ab* + *c*) *T*".

Here, "*T*" is a null term or a term without the occurrence of "*a*", "*b*", or "*c*".

Pattern 3: Term Omission with Losing Literals

Original Boolean expression: "*ab* + *bc*" → Mutant Boolean expression: "*ab*"

Pattern extension includes:

"*ab T* + *bc*" → "*ab T*",

"*ab* + *bc T*" → "*ab*", and

"(*ab* + *bc*) *T*" → "*ab T*".

Here, "*T*" is a null term or a term without the occurrence of "*a*", "*b*", or "*c*".

Pattern 4: Distributive One's-Complement-Literal Conjunctive Term Omission Without Losing Literals

Original Boolean expression: "*acde* + *bc* + *ab*!*d*" → Mutant Boolean expression: "*acde* + *bc*".

Pattern extension includes:

"*acde T* + *bc* + *ab*!*d*" → "*acde T* + *bc*",

"*acde* + *bc* + *ab*!*d T*" → "*acde* + *bc*", and

"(*acde* + *bc* + *ab*!*d*) *T* "→ "(*acde* + *bc*) *T"*.

Here, "*T*" is a null term or a term without the occurrence of "*a*", "*b*", "*c*", "*d*", and "*e*".

Pattern 5: Concurrent Term and Literal Omission without Losing Literals

Original Boolean expression: "*ab*!*c* + *a*!*bc* + !*abc*" → Mutant Boolean expression: "*ab* + *ac*".

Pattern extension includes:

(*ab*!*c* + *a*!*bc* + !*abc*) *T* → (*ab* + *ac*) *T*.

Here, "*T*" is a null term or a term without the occurrence of "*a*", "*b*", or "*c*".

The empirical evaluation result also indicates that the fault-detection capability of MUMCUT varies with fault types. As for Term Omission Fault (TOF), MUMCUT can only detect faults in 91.53 situations. In our companion work [5], we find that, in most situations when one fault is seeded into a general Boolean expression, it results in more than one fault in the equivalent IDNF. Further, there are more chances where the multiple-literal term is omitted in their equivalent IDNF and the above-mentioned patterns occur with higher frequency. This explains why the fault-detection capacity of MUMCUT for TOF is lower than others. As to the Expression Negation Fault (ENF), any test case can kill a mutant since the truth values of the original Boolean expression and the mutant are always different. The fault-defection capability of MUMCUT for the remaining six types of fault is very close. Table 2 shows the distributions of these five patterns in different fault types.

Table 2. Distributions of the five failure patterns

Mutation Type	P1	P2	P3	P4	P5	Total
ENF	0	0	0	0	0	0
LNF	0	2	2	0	1	5
TOF	3	4	15	3	0	25
TNF	0	0	5	0	0	5
LOF	0	1	1	0	0	2
LIF	2	3	1	0	0	6

LRF	1	2	1	0	0	4
ORF[2]	5	2	4	1	0	8 + 4
Total	11	14	29	4	1	55 + 4

Further, the result also provides a useful guideline for developers. We should pay special attention to the faults that are difficult to detect, such as TOF.

4.4 Threats to validity

In this experiment we developed a mutation strategy to produce the limited size of mutants because of the practicability concerns. The restricted number of mutants may be a limitation of the validity of the empirical evaluation. Another possible limitation in the evaluation of overall effectiveness comes from the mutant ratio of different types of faults, since it is hard to predict which type is more frequent in the real world. Finally, the number of general Boolean expressions may also affect the validity of this evaluation.

5. Related Work

This work complements and extends the research on fault-based testing of Boolean specifications. For the most part, the research has focused on the systematic test generation, selection, and empirical evaluation of test suites, and these test suites are used to detect several special fault types of Boolean expressions in the specific forms, such as DNF, CNF, and IDNF. We outline below the work related to our project.

Weyuker et al. [15] investigated and proposed a family of meaning impact strategies for automatically generating test cases for any implementation intended to satisfy a given specification that is a Boolean formula. These strategies are based on two basic concepts, namely *unique true points* and *near false point*. These strategies are effective to detect five operator faults, such as *variable negation faults*. They also require that the Boolean expressions under test should be in IDNF.

Chen and Lau [3, 4] proposed a set of more efficient test case generation strategies called MUMCUT for Boolean expressions in IDNF, including MUTP, MNFP, and CUTPNFP. These strategies are also based on the *unique true point* and *near false point*, and can detect seven single faults in a Boolean expression. MUMCUT also requires that Boolean expressions under test should be in IDNF.

IDNF Boolean expressions are a small subset of all Boolean expressions in the real world [2, 5, 11]. In this paper, we have applied MUMCUT to generate test cases for general Boolean expressions. The generated test cases are expected to detect common faults. Finally, our work enables MUMCUT to obtain its application in the whole set of Boolean expressions.

To evaluate the effectiveness of the proposed strategy, a lot of empirical studies have been reported [3, 4, 14, 15]. As we know, all existing empirical studies have used the same set of Boolean expression samples, first used by Weyuker et al. [15], to examine the effectiveness of their strategy. These samples consist of twenty Boolean specifications taken from the specification for a real aircraft collision avoidance system (TCAS II). Chen and Lau [3, 4] also used these Boolean specifications samples in their previous empirical evaluation related to MUMCUT, such as MUTP, and CUTPNFP.

In this paper, we use a generator [12] developed by us to produce a large number of Boolean expression samples in arbitrary forms. This enhances the representativeness of experimental samples. Furthermore, our empirical evaluation shows an effectiveness of 98.20%, while the empirical evaluation in [2] reported an effectiveness of 99.8% using 20 Boolean expression samples from [15].

6. Conclusion

We have applied MUMCUT to generate test cases for general Boolean expressions and reported an empirical evaluation of the fault detection capability of MUMCUT in the realistic situation.

MUMCUT is a fault-based test case generation strategy for Boolean expressions in IDNF. When it is applied to a general Boolean expression ***B***, we first transform it into an IDNF ***I*** and then employ the MUMCUT strategy to generate a test suite ***T*** for ***I***. Subsequently, the test suite ***T*** can be used as test cases of the original Boolean expression ***B***.

We have also developed a mutation-based experiment to empirically evaluate effectiveness of MUMCUT. Given a general Boolean expression, mutation technique is used to generate a set of mutants where only one fault is seeded. A test suite generated by MUMCUT is used to decide whether a mutant can be killed. The original Boolean expression is used as an oracle. Our empirical evaluation showed that, in 98.20% of the situations, MUMCUT was effective in the context of fault-based testing of general Boolean expressions.

As future work, we plan to conduct another empirical evaluation of the effectiveness of MUMCUT when more than one fault is seeded into general Boolean expressions. We also plan to develop test case generation strategies or techniques for the failure patterns observed in this study.

[2] In this type of fault, the combination of failure patterns occurs in four mutants.

References

[1] T.Y. Chen, D.D. Grant, M.F. Lau, S.P. Ng, and V.R. Vasa, BEAT: Boolean expression fault-based test case generation, in Proceedings of International Conference on Information Technology: Research and Education, IEEE Computer Society, Los Alamitos, CA, 2003, pp. 64–69

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

[3] T.Y. Chen and M.F. Lau, An empirical evaluation on the greedy CUTPNFP strategy for Boolean specification based testing, in Proceedings of the 5th Joint conference on Information Science (JCIS '00), 2000, pp. 627–630

[4] T.Y. Chen and M.F. Lau, An empirical study on the effectiveness of the greedy MUTP strategy, in Proceedings of Software Engineering Research and Practice (SERP '98), IEEE Computer Society, Los Alamitos, CA, 1998, pp. 338–344

[5] T.Y. Chen, K.Y. Sim, and C.A. Sun, A simulation analysis of fault relationships between Boolean expressions in general form and irredundant disjunctive normal form, in preparation

[6] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, Hints on test data selection: help for the practicing programmer, IEEE Computer, Vol. 11, No. 4, 1978, pp. 34–41

[7] R. Galivanche and S.M. Reddy, A parallel PLA minimization program, in Proceedings of the 24th ACM/IEEE Design Automation Conference, ACM, New York, NY, 1987, pp. 600–607

[8] K. R. Kuhn, Fault classes and error detection capability of specification-based testing, ACM Transactions on Software Engineering and Methodology, Vol. 8, No. 4, 1999, pp. 411–424

[9] N. Juristo, A. M. Moreno, and S. Vegas, Reviewing 25 years of testing technique experiments, Empirical Software Engineering, Vol. 9, No. 1, 2004, pp. 7–44

[10] J. Offutt and S. Liu, Generating test data from SOFL specifications, Journal of Systems and Software, Vol. 49, No. 1, 1999, pp. 49–62

[11] V. Okun, P.E. Black, and Y. Yesha, Comparison of fault classes in specification-based testing, Information and Software Technology, Vol. 46, No. 8, 2004, pp. 525–533

[12] C.A. Sun and K.Y. Sim, A FSM-based parameterized generator of Boolean expressions, 1st International Computer Engineering Conference: New Technologies for the Information Society (ICENCO '04), Cairo, Egypt, 2004.

[13] T. Tsuchiya and T. Kikuno, On fault classes and error detection capability of specification-based testing, ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 1, 2002, pp. 58–62

[14] M.A. Vouk, A. Paradkar, and K.-C. Tai, Empirical studies of predicate-based software testing, in Proceedings of International Symposium on Software Reliability Engineering (ISSRE '96), IEEE Computer Society, Los Alamitos, CA, 1996, pp. 55–65

[15] E.J. Weyuker, T. Goradia, and A. Singh, Automatically generating test data from a Boolean specification, IEEE Transactions on Software Engineering, Vol. 20, No. 5, 1994, pp. 353–363

[16] H. Zhu, A.V. Hall, and H.R. May, Software unit test coverage and adequacy, ACM Computing Surveys, Vol. 29, No. 4, 1997, pp. 366–427