Evaluating and comparing fault-based testing strategies for general Boolean specifications: A series of experiments

Evaluating and Comparing Fault-Based Testing Strategies for General Boolean Specifications: A Series of Experiments

Chang-ai Sun, Yimeng Zai
School of Computer and Communication Engineering
University of Science and Technology Beijing, China

Huai Liu
Australia-India Research Centre for Automation Software Engineering
RMIT University, Melbourne VIC Australia
Email: casun@ustb.edu.cn, huai.liu@rmit.edu.au

August 6, 2014

Abstract

A great amount of fault-based testing strategies have been proposed to generate test cases for detecting certain types of faults in Boolean specifications. However, most of the previous studies on these strategies were focused on the Boolean expressions in the disjunctive normal form, even the irredundant disjunctive normal form — little work has been conducted to comprehensively investigate their performance on general Boolean specifications. In this study, we conducted a series of experiments to evaluate and compare eighteen fault-based testing strategies using over four thousand randomly generated fault-seeded Boolean expressions. In the experiments, a testing strategy is regarded as effective and efficient if it can detect most of the seeded faults using a small number of test cases. Our experimental results show that if a testing strategy is highly effective and efficient when testing the Boolean expressions in the irredundant disjunctive normal form, it also shows high effectiveness and efficiency on general Boolean expressions. It is found that one family of fault-based testing strategies, namely MUMCUT, normally deliver the best performance among all the eighteen strategies. Our study provides an in-depth understanding and insight of fault-based testing for general Boolean expressions.

keywords: software testing; Boolean specification; test case generation; fault-based testing.

1 Introduction

Software testing is a widely used quality assurance approach, the basic idea of which is to select a limited number of test cases as inputs, and then observe whether the outputs of the program under test are as expected, with a fault being reported when unexpected outputs are detected. In most situations, the number of possible inputs for a program may be extremely large, so it is impossible to conduct the exhaustive testing. In order to make testing efficient, various strategies have been proposed to select part of program inputs as test cases such that software failures can still be effectively detected [29].

Decisions and conditions, which determine different execution paths of the program under test, are the crucial part in software specifications [33]. Boolean expressions are commonly used to describe these decisions and conditions. For instance, assume a branch statement written in C is "if ((i > 0) && (j < 10)) i = i + m; else j = j - m", its condition is "(i > 0) && (j < 10)". We treat "i > 0 " and "j < 10" as Boolean variables "a_1" and "a_2", respectively, then we have the Boolean expression "$a_1 \wedge a_2$" for this condition. Boolean expressions are conceptually simple, and it is not very difficult to automatically generate test cases based on them. Nevertheless, it is still extremely expensive to exhaustively test a Boolean expression when it involves multiple Boolean variables. Given that an expression contains n Boolean variables, there will be 2^n distinct test cases. When n is large, it is almost infeasible to conduct exhaustive testing [13].

For decades, a lot of efforts have been made on how to efficiently test Boolean specifications [18, 40, 38, 6, 7]. Weyuker et al. [40] proposed a family of test case generation strategies, namely, MIN, ONE, MANY-A, MANY-B, MAX-A, and MAX-B. They evaluated the effectiveness of these strategies using mutation analysis technique [2]. A mutant was generated by seeding a single fault into a Boolean expression. Five types of faults were considered in their study. The experimental results show that the test sets (a test set refers to a set of test cases) generated by these strategies have different sizes and different effectiveness on detecting certain types of faults. For example, MAX-B is able to detect most of seeded faults, while it requires the largest number of test cases; on the other hand, MIN requires the fewest test cases, while its fault-detection effectiveness is the lowest among all strategies.

It has been widely accepted that testing cannot prove that a program is fault-free, but it is possible to guarantee the absence of certain types of faults. Fault-based testing [28] aims at designing test cases especially for some specific faults: If a program passes all these test cases, the program can be regarded as free of the specific fault types. Chen and Lau [6] investigated seven types of faults in Boolean expressions, and proposed three strategies, namely MUTP, MNFP, and CUTPNFP. These fault-based testing strategies significantly increase the efficiency of testing: They not only guarantee the detection of all seven fault types, but also reduce the size of the test set as compared with MAX-B. Yu et al. [41] further developed several methods for implementing the MUMCUT strategy (the integration of MUTP, MNFP, and CUTPNFP).

Among the existing test case generation strategies for Boolean specifications, most of them assumed that the Boolean specifications under test are in the irredundant disjunctive normal form (IDNF). In reality, developers may write Boolean specifications in any form, which indicates that faults can be introduced into a specification of any form, and some constraints may exist among Boolean variables [19]. To be practical, there should not be any restriction on the form of a Boolean specification. As observed in our previous work [7], a single fault in a general Boolean expression can give rise to a very large number of faults in the equivalent expression of IDNF. So far, little work has been conducted to evaluate the effectiveness of the IDNF-oriented testing strategies on detecting faults in general Boolean expressions [33].

In this paper, we conduct a series of experimental studies to comprehensively evaluate the fault-detection effectiveness and efficiency of fault-based test case generation strategies on general Boolean specifications. The paper is organised as follows. In Section 2, we introduce the underlying concepts of test case generation for Boolean specifications. In Section 3, we discuss the design of experiments and the threats to validity of our study. In Section 4, we report the evaluation results and provide a comparative analysis. We discuss the related work in Section 5 and conclude the paper in Section 6.

2 Preliminaries

2.1 Notation and terminology

A Boolean expression has a Boolean value of either TRUE (normally also denoted by ‘1’) or FALSE (‘0’). There are two main components for Boolean expressions, namely Boolean variables and Boolean operators. A Boolean variable has the value of either 1 or 0. The mostly used Boolean operators include AND (normally also denoted by ‘$\cdot$’), OR (‘+’), and NOT (‘$\bar{\ }$’). Boolean expressions can also include some parentheses to change the precedence of operators or association of Boolean variables.

A Boolean expression can be represented in various forms. One popularly used form is disjunctive normal form (DNF), which can also be called sum-of-products form, for example, $a\bar{b} + bc$. In a Boolean expression, a *literal* refers to the occurrence of a Boolean variable or its negation, such as a, $\bar{b}$, b, and c in the above example. The product of literals is called a *term*, such as $a\bar{b}$ and bc in the above example. A DNF expression f is in IDNF if and only if removing any Boolean literal or conjunction from the expression could potentially change the value of f. A test case for a Boolean expression refers to the value assignment to all involved variables. For example, a test case for $a\bar{b}+bc$ is 010, which refers to a=0, b=1, and c=0.

Test cases for a Boolean expression can be categorised into true points (which cause the expression to produce the value ‘1’) and false points (which cause the expression to produce the value ‘0’). True points can be further categorised into *unique true points* (*UTPs*) and *overlapping true points* (*OTPs*). Suppose that a Boolean expression in IDNF is denoted by $p_1 + p_2 + \cdots + p_m$, where p_i

$(i = 1, 2, \cdots, m)$ refers to the i^{th} term of the expression. The unique true points for p_i refer to the test cases that cause p_i to be 1 but all other terms to be 0. The overlapping true points refer to the true points that are not unique for any term. False points can be further categorised into *near false points*(*NFPs*) and *remaining false points* (*RFPs*). A test case is referred to as a near false point for the j^{th} literal of p_i if it causes p_{ij} to be 1 but the whole Boolean expression to be 0, where p_{ij} is a term that is obtained by negating the j^{th} literal of p_i. The remaining false points refer to the false points that are not the near false points for any literal. Consider the Boolean expression a(b+c)+bc, its DNF representation is *ab+ac+bc* and there are three terms, namely *ab*, *ac*, and *bc*. Its unique true points are {110, 011, 101}; its overlapping true points are {111}; its near false points are {010, 001, 100}; and its remaining false points are {000}.

2.2 Test case generation strategies

In recent years, various test case generation strategies for Boolean expressions have been developed [33]. They can be further grouped into *syntactic* and *semantic* strategies [23]. The former usually restricts Boolean expressions under test to be a certain form, for instance DNF or IDNF. In this study, we are going to evaluate and compare the fault-detection effectiveness and efficiency of those syntactic fault-based test case generation strategies when they are used for general Boolean expressions. Therefore, semantic strategies [13, 1] will not be included for experiments. Some researchers [3, 20] recently proposed to use the modern satiability solvers to reduce the test case generation effort. However, their approach is effectively based on mutation testing technique [15]. Mutation-based testing is actually a "special" case of fault-based testing: The former generates test cases based on a large number of mutants; while the latter is based on certain types of faults, which imply a large number of faulty versions. In a word, this mutation-based testing strategy is not a general fault-based one, so we did not include it in our experiments. Next, we briefly introduce representative fault-based test case generation strategies. All of them are illustrated by the Boolean expression a(b+c)+bc.

Weyuker et al. [40] proposed the basic meaningful impact strategy for generating test cases from Boolean expressions. The intuition underlying their approach is that given a Boolean expression, a test set should be chosen such that, if possible, each literal occurrence in the expression demonstrates its meaningful impact on the outcome. They developed six algorithms based on this strategy, as listed in the following.

- ONE:
  - Select one unique true point for each term;
  - Select one near false point for each literal.

  For the example expression, its unique true points are 110, 101, and 011; for the term *ab*, its near false points are 100 and 010; for the term *ac*, its near false points are 100 and 001; for the term *bc*, its near false points are 010 and 001. Therefore, its ONE test suite is {110, 011, 101, 010, 001, 100}.

- MIN:
  - Select one unique true point for each term;
  - Construct the minimum set of near false points for covering all literals. Note that different literals may have common near false points, so it is possible to use fewer near false points to cover all literals.

  One possible MIN test suite for the example expression is {110, 011, 101, 001}.

- MANY-A:
  - Select $\max\{\lceil \log_2 N_u \rceil, 1\}$ unique true points for each term, where N_u is the number of unique true points for the term;
  - Select $\max\{\lceil \log_2 N_n \rceil, 1\}$ near false points for each literal, where N_n is the number of near false points for the literal.

  For the example expression, there is only one unique true point for each term; thus, three unique true points (namely 110, 101, and 011) are selected. For each literal, only one near false point is available for selection and finally three near false points (namely 010, 001, and 100) are selected. The MANY-A test suite for the example expression is {110, 011, 101, 010, 001, 100}.

- MANY-B:
  - Select $\max\{\lceil \log_2 N_u \rceil, 1\}$ unique true points for each term, where N_u is the number of unique true points for the term;
  - Select $\max\{\lceil \log_2 N_n \rceil, 1\}$ near false points for each literal, where N_n is the number of near false points for the literal;
  - Select $\max\{\lceil \log_2 N_o \rceil, 1\}$ overlapping true points, where N_o is the number of overlapping true points for the Boolean expression;
  - Select $\max\{\lceil \log_2 N_r \rceil, 1\}$ remaining false points, where N_r is the number of remaining false points for the Boolean expression.

  For the example expression, three unique true points (namely 110, 101, and 011) and three near false points (namely 010, 001, and 100) are selected; only one overlapping true point (namely 111) and only one remaining false point (namely 000) are selected. Thus, the MANY-B test suite for the example expression is {110, 011, 101, 010, 001, 100, 111, 000}.

- MAX-A:
  - Select all unique true points;

  - Select all near false points.

  For the example expression, its MAX-A test suite is {110, 011, 101, 010, 001, 100}.

- MAX-B:

  - Select all unique true points;
  - Select all near false points;
  - Select $\max\{\lceil \log_2 N_o \rceil, 1\}$ overlapping true points, where N_o is the number of overlapping true points for the Boolean expression;
  - Select $\max\{\lceil \log_2 N_r \rceil, 1\}$ remaining false points, where N_r is the number of remaining false points for the Boolean expression.

  For the example expression, its MAX-B test suite is {110, 011, 101, 010, 001, 100, 111, 000}.

Chen and Lau [6] investigated seven types of faults, namely ENF (Expression Negation Fault), LNF (Literal Negation Fault), Term Omission Fault (TOF), Operator Reference Fault (ORF), Literal Omission Fault (LOF), Literal Insertion Fault (LIF), and Literal Reference Fault (LRF). They found that the MAX-B strategy guarantees the detection of all the seven types of faults, but it requires a large amount of test cases. Chen and Lau proposed three fault-based test case generation strategies, namely MUTP (multiple unique true points), MNFP (multiple near false points), and CUTPNFP (corresponding unique true point and near false point pair), as presented in the following.

- MUTP: for each term, select the unique true points that can cover both 0 and 1 values of every variable not appearing in the term.

  For the example expression, its MUTP test suite is {110, 101, 011}.

- MNFP: for each literal, select the near false points that can cover both 0 and 1 values of every variable not appearing in the term containing the literal.

  For the example expression, its MNFP test suite is {010, 100, 001}.

- CUTPNFP: for each term and each literal in the term, select a pair of unique true point and near false point that only differ in the corresponding value of the literal.

  For the example expression, its CUTPNFP test suite is {110, 010, 100, 101, 001, 011}.

The integration of MUTP, MNFP, and CUTPNFP, namely MUMCUT [8], guarantees the detection of all seven types of faults while using fewer test cases than MAX-B. In other words, MUMCUT is more efficient than MAX-B. Yu et al. [41] proposed several methods for automatically generating MUMCUT test sets using the techniques of greedy heuristics, divide-and-conquer, and incremental expansion. In their methods, the MUMCUT test set can be constructed by incrementally generating test cases by means of MUTP, MNFP, and CUTPNFP. Four different methods are proposed according to different permutation orders, namely MUMCUT-CUN, MUMCUT-UCN, MUMCUT-UNC, MUMCUT-NCU, where the letters 'U', 'N', and 'C' represent MUTP, MNFP, and CUTPNFP, respectively. For the example expression, all these four strategies produce the same test suite, namely {110, 010, 100, 101, 001, 011}.

In our previous work [36], we investigated the fault-detection capabilities of MUMCUT when it is used to test general Boolean expressions, and we discovered some patterns of faults in general Boolean expressions that cannot be detected by MUMCUT. We also investigated the reason why such faults cannot be detected by the existing fault-based strategies [34]. Based on the investigations on undetected faults, we proposed some guidelines for extending MUMCUT such that more faults can be detected. The basic intuition of these extension guidelines is that the test cases should have certain degree of diversity. The concept of diversity has been widely used in other test case selection methods [9, 5, 21]. The details of the MUMCUT extensions proposed in our previous work are presented as follows. Without loss of generality, these extensions are based on the version of MUMCUT-CUN. Assume that the test set of MUMCUT is denoted as $TS = \{ts_1, ts_2, ..., ts_m\}$, and the remaining points $TR = \{t_1, t_2, ..., t_r\}$ where $m < 2^n$, $r = 2^n - m$, and n is the number of Boolean variables within the Boolean expression under test.

- MUMCUT-NCases:
  - Set $TS_{NCases} = \emptyset$;
  - Select the point where all Boolean variables have the value of 1, and the point where all Boolean variables have the value of 0, and add these points to the set TS_{NCases};
  - If $n > 2$, then select $n - 2$ points t_i from TR and add them to the set TS_{NCases}, where t_i should satisfy the following constraint: $max_{j=1..m}(|t_i - ts_j|) \geq max_{k=1..r \wedge k \neq i}(max_{j=1..m}(|t_k - ts_j|))$, where $|t_i - ts_j|$ denotes the number of Boolean variables in t_i and ts_j with different values;
  - Return $TS \bigcup TS_{NCases}$.

  For the example expression, its MUMCUT-NCases test suite is {110, 011, 101, 010, 001, 100, 111, 000}.

- MUMCUT-M2NFP:
  - For any two literals (j_1^{th} and j_2^{th}) in p_i, select the *pair near false points* (denoted as *2NFP*) that can cover both 0 and 1 values of every variable not appearing in p_i. 2NFP is an extension of one literal *near false point*, and formally, a test case is referred to as 2NFP for the j_1^{th} and j_2^{th} literals of p_i if it caused $p_{i,j_1 j_2}$ to be 1 but

the whole Boolean expression to be 0, where p_{i,j_1j_2} is a term that is obtained by negating the j_1^{th} and j_2^{th} literal of p_i. The above strategy is referred to as *M2NFP*, which is an extension of MNFP with respect to pair literals, and the resulting test set is denoted as TS_{M2NFP}.

  - Return $TS \bigcup TS_{M2NFP}$.

  For the example expression, its MUMCUT-M2NFP test suite is {110, 011, 101, 010, 001, 100}.

- MUMCUT-MS2NFP:

  - Select any two successive literals (j_1^{th} and j_2^{th}) in p_i, select the *successive pair near false points* (denoted as *S2NFP*) that can cover both 0 and 1 values of every variable not appearing in p_i. *S2NFP* is the same to *2NFP* except that the former requires that two literals must successively occur in the term. The strategy is referred to as *MS2NFP*, and the resulting test set is denoted as TS_{MS2NFP}.
  - Return $TS \bigcup TS_{MS2NFP}$.

  For the example expression, its MUMCUT-MS2NFP test suite is {110, 011, 101, 010, 001, 100}.

- MUMCUT-NCases&M2NFP: Merge the test set of MUMCUT-NCases into the one of MUMCUT-M2NFP.

  For the example expression, its MUMCUT-NCases&M2NFP test suite is {110, 011, 101, 010, 001, 100, 111, 000}.

- MUMCUT-NCases&MS2NFP: Merge the test set of MUMCUT-NCases into the one of MUMCUT-MS2NFP.

  For the example expression, its MUMCUT-NCases&MS2NFP test suite is {110, 011, 101, 010, 001, 100, 111, 000}.

It is expected that the fault-detection effectiveness could be improved by enhancing the diversity among selected test cases [34]. However, the fault-detection effectiveness and efficiency of the above extensions to MUMCUT have not yet been evaluated, which is to be reported later in this paper.

3 Experimental Design

In this section, we first state the research questions targeted in the study, then explain how the experiment is designed and implemented according to research questions, and finally discuss the threats to validity of our study.

3.1 Research questions

In this study, we attempt to answer the following basic research questions about fault-based test case generation strategies on general Boolean expressions.

- RQ1: How effective is a fault-based test case generation strategy in detecting faults in general Boolean expressions?

  The fault-detection effectiveness is a major metric for evaluating a testing method. Normally, the more faults a testing method can detect, the more effective it is. In this study, we evaluate the fault-detection effectiveness of a testing method via mutation analysis technique [2]. Given a Boolean expression in the general form, a set of mutants can be generated, each of which is related to a seeded fault. A mutant m is said to be *killed* by a test case t if t causes m to produce a value different from that of the original expression. The fault-detection effectiveness is measured by the number of killed mutants (that is, the number of detected faults).

- RQ2: Which is the most efficient strategy for testing general Boolean expressions?

  In reality, if a testing method requires a high cost (such as a large number of test cases, too many computing resources required, etc.), its high fault-detection effectiveness may become meaningless. As an extreme example, the exhaustive testing has the highest fault-detection effectiveness, but it is practically infeasible. We expect that a good testing method should have a high fault-detection efficiency, meaning that it can detect a large number of faults at low cost. In this study, we evaluate the fault-detection efficiency of a testing method by the ratio between the number of detected faults and the number of used test cases. Through our study, we will recommend the most efficient testing method for general Boolean expressions.

- RQ3: To what extent do our proposed extensions improve the fault-detection effectiveness of MUMCUT for testing general Boolean expressions, and what is the cost for such improvement?

  As mentioned above, some faults in the general Boolean expressions cannot be detected by MUMCUT [36], and hence some extensions were proposed to improve its fault-detection effectiveness [34]. However, their improvements are not evaluated yet. Through this study, we want to answer whether these extensions are able to improve the fault-detection effectiveness without significantly increasing the number of used test cases.

3.2 Experimental settings

We conducted a series of experiments to answer the above research questions. The settings of our experiments are described as follows.

The main purpose of this study is to evaluate and compare the effectiveness and efficiency of various fault-based test case generation strategies on general

Boolean expressions. In most previous studies, Boolean specifications extracted from a real-life software were used as the subjects of experiments. Though real-life examples are important for empirical studies, a few real-life Boolean specifications cannot guarantee that the full picture would be given. Technically speaking, an extremely large amount of Boolean expressions should be used for getting a statistically reliable conclusion. However, in practice, empirical studies with lots of real-life examples are time-consuming and labor-intensive. In our study, the subjects are the general Boolean expressions that were generated by an FSM (finite state machine)-based parameterised generator [35]. The structure of the generated Boolean expressions can be adjusted through the parameters, including *max term number* (TN), *max term length* (TL), *max literal number* (LN), and *max operator number* (OP). We found the transformation of general Boolean expressions to their IDNF representation is very time-consuming when LN was larger than ten. In our experiments, we set TN to nine, TL to five, LN to nine, and OP to seven, just because of the limitation of experimental environments (i.e. low computing capacity). A total of 1000 expressions were generated in a random manner, which guarantees a large number of subjects without any human bias.

Common faults that might occur during the programming process have been carefully investigated and summarized [31, 32]. The described typical programming errors involve missing or extra literals/variables, and the use of incorrect operators and operands. Some researchers have recently introduced ten fault types for general Boolean expressions, including ENF, SA0, SA1, VNF, ASF, ORF, VRF, MVF, CCD, and CDF [24, 10]. However, these terminologies are a bit confusing. For instance, VNF (Variable Negation Fault) is quite misleading: It is unsure whether it refers to the negation of all occurrences of a variable or one occurrence of a variable. A very precise definition of every fault type is provided in [32]. In this study, we used ten types of faults for general Boolean expressions in our previous work [7]. We described the ten types of faults (illustrated by the previous example $a\bar{b} + bc$, except for the POF fault) as follows.

- ENF (expression negation fault): The whole or part of the expression is mistakenly negated. For example, $\overline{a\bar{b} + bc}$.
- TNF (term negation fault): A term is mistakenly negated. For example, $a\bar{b} + \overline{bc}$.
- TOF (term omission fault): A term is mistakenly omitted. For example, $a\bar{b}$.
- ORF (operator reference fault): An operator is mistakenly replaced by another type of operator. For example, $a + \bar{b} + bc$.
- LNF (literal negation fault): A literal is mistakenly negated. For example, $ab + bc$.
- LOF (literal omission fault): A literal is mistakenly omitted. For example, $a + bc$.

- LIF (literal insertion fault): Another literal is mistakenly inserted into a term. For example, $a\bar{b}c + bc$.

- LRF (literal reference fault): A literal is mistakenly replaced by another literal. For example, $a\bar{b} + ac$.

- POF (parentheses omission fault): A pair of parentheses are mistakenly omitted. For example, $(a + b)\,c$ might be mistakenly implemented as $a+bc$.

- PIF (parentheses insertion fault): A pair of parentheses are mistakenly inserted into the expression. For example, $a\left(\bar{b} + bc\right)$.

We adopted the mutation analysis technique [2] to construct the faulty versions (namely mutants) for each Boolean expression. For each mutant, only one fault (namely a single syntactical change) was mimicked. This single fault assumption is due to the two hypotheses behind mutation analysis, namely the *competent programmer hypothesis* and *the coupling effect* [14, 30]. The first hypothesis states that competent programmers tend to develop programs close to the correct version, which implies that the faults made by competent programmers are merely a few simple faults. The second hypothesis states that test data that can detect simple types of fault are sensitive enough to detect more complex types of faults. Even one single fault assumption was applied, the number of literal related mutants (including LNF, LOF, LIF, and LRF) is still very large. To control the number of mutants, we used the following strategy, as illustrated by the fault type LNF. For each term in the expression, if the number of literals is equal to or larger than three, three mutants will be generated by negation of first, middle, and last literal in the term; Otherwise, mutants will be generated by negation all possible literals (when a term has only one literal, its mutant corresponds to a TOF fault). Note that for LIF and LRF, only the Boolean variable that does not occur in the current term will be used. Table 1 reports the number of mutants generated based on each type of fault.

In this study, we attempt to comprehensively evaluate and compare most of the existing fault-based testing strategies for Boolean specifications. The strategies under study include ONE, MIN, MANY-A, MANY-B, MAX-A, MAX-B, MUTP, MNFP, CUTPNFP, MUMCUT-CUN, MUMCUT-UCN, MUMCUT-UNC, and MUMCUT-NCU. At the same time, we like to have an insight into the performance improvement made by the extensions of MUMCUT and the cost incurred by the improvement. Therefore, MUMCUT-NCases, MUMCUT-M2NFP, MUMCUT-MS2NFP, MUMCUT-NCases&M2NFP, and MUMCUT-NCases&MS2NFP are also included in our study.

Our experiments were conducted through the following procedure.

1. Choose a type of fault, and a test case generation strategy.

2. Generate mutants based on the fault type. In our study, the number of mutants is exchangeable to the number of faults.

3. Generate test cases using the testing strategy under study.

Table 1: Number of mutants generated

Type of fault	Number of original expressions	Number of mutants
ENF	100	314
TNF	100	416
TOF	100	294
ORF	100	714
LNF	100	778
LOF	100	269
LIF	100	238
LRF	100	342
POF	100	88
PIF	100	549
All faults	1000	4002

4. Use the test cases (generated in Step 3) to test mutants (generated in Step 2). When a test case cause a mutant to produce an output different from that of the original expression, the mutant is said to be killed (in other words, a fault is detected). After all test cases are executed on all mutants, record the total number of killed mutants (in other words, the number of detected faults).

3.3 Threats to Validity

The threats to validity of our study are discussed as follows.

The threat to internal validity is related to how the experiments were designed and implemented. Firstly, in our study, mutation operators were used to simulate possible faults of general Boolean expressions. Although mutation analysis has been widely used to evaluate the effectiveness of various testing techniques [2], the mimicked faults (namely mutants) are possibly different from the real-life faults. Secondly, the limited number of sample expressions may threat the validity of evaluations. In our experiments, some strategies were adopted to control number of mutants, and some parameters (such as the maximum number of literals) are concerned when the tool was used to produce expression samples, which may not cover all possible cases. Also, it is possible that randomly generated Boolean expressions can show difference when compared with real-life ones. Thirdly, mutation analysis are restricted to simple mutants that are created by making a single syntactical change because of the single fault assumption [14, 30]. However, multiple faults in a single expression are possible yet rare. Fourthly, the occurrences of different types of realistic faults are varying, while faults mimicked by means of mutation operators were randomly generated. Such a distribution difference may result in a deviation of the effectiveness evaluation result of the test case generation strategies when they are used in practice [37]. Finally, the threat to internal validity is also related to the implementations. We used a tool [35] to generate Boolean expressions and mutants as the subjects. The programs that implement test case generation methods were developed based on previous source code, and have been cross checked by different individuals. We are confident that our experiments were correctly implemented.

The threat to external validity is related to the subjects. In most situations, Boolean literals in an expression may represent complex concepts or computations; this actually indicates that the application of fault-based test generation strategies to realistic software involves three steps, namely abstracting Boolean expressions, generating test cases using the strategies, and converting (or refining) the derived test cases into test data. Accordingly, it would be a good idea to collect an extremely large number of Boolean expressions from code or models and use them as base for the experiments. However, it is very expensive in both time and labor and thus infeasible. In our study, instead of adopting real-life specifications, we used Boolean expressions that were randomly generated by a mature tool [35] and a large amount of Boolean expressions were automatically generated at a low cost. We are confident that these randomly constructed Boolean expressions comprehensively reflect various attributes and features of general Boolean specifications without human bias.

The threat to construct validity is related to the measurement. In our study, we used the number of killed mutants to measure the fault-detection effectiveness. This measurement has been widely acknowledged as a fair metric for evaluating a testing method. In addition, we measured the fault-detection efficiency using the ratio of the number of killed mutants to the number of test cases. This metric is straightforward and intuitively appealing.

There is little threat to conclusion validity in our study. A large number of Boolean expressions have been used as the subjects, and ten types of faults have been considered. Our experiments provided a huge amount of data, which helped us obtain a statistically reliable conclusion. Statistical tests were also conducted to validate the statistical significance of our experimental results. Finally, constraints may exist in Boolean expressions [40, 19]. An interesting question is to investigate the impact of constraints on fault-based test case generation strategies, which requires to simulate possible constraints by introducing variable dependencies in randomly generated Boolean expressions, and thus is left for our future work.

4 Experimental Results

4.1 RQ1: Fault-detection effectiveness

The experimental results about the fault-detection effectiveness (measured by the percentage of the number of detected faults over the total number of seeded faults) are summarised in Figures 1 and 2. Figure 1 shows how effectively these strategies are in detecting each type of fault; while Figure 2 shows the fault-

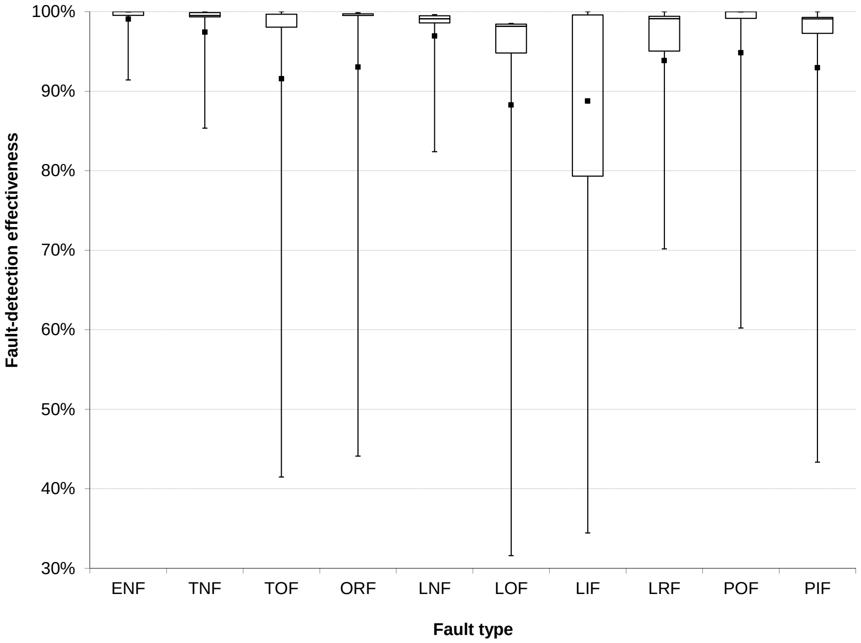

Figure 1: Fault-detection effectiveness on each fault type

detection effectiveness of each fault-based testing strategy. In these figures, box-plots are used to represent the statistical distribution of fault-detection effectiveness. For each box, the upper and lower bounds denote the third and first quartile of the fault-detection effectiveness, respectively, while the middle line inside the box represents the median value. The top and bottom whiskers out of the box denote the maximum and minimum values, respectively, and a square dot represents the mean value of the fault-detection effectiveness.

Based on the detailed experimental data and Figure 1, we can have the following observations.

- ENF is the easiest fault type to be detected. Among all 18 strategies, 13 can detect all faults. Even for the worst case (MUTP), 91.4% of ENF faults can be detected.

- LOF is the most difficult fault type to be detected. No testing strategy can detect over 99% of LOF faults, and only 31.6% of LOF faults are detected for the worst case (MUTP).

- The difficulty for detecting LIF seems to be similar to that of LOF. The value range for the fault-detection ratios on LIF is from 34.45% to 100%, just marginally higher than that of LOF (from 31.6% to 98.51%).

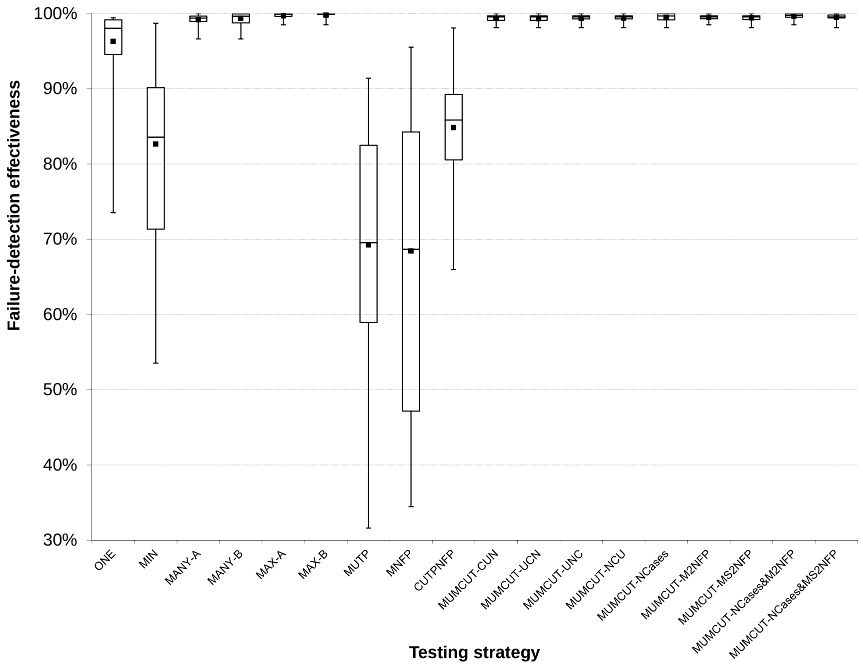

Figure 2: Fault-detection effectiveness of each testing strategy

Kapoor and Bowen [24] conjectured that ENF might be the weakest one among eight fault types for general Boolean expressions; in other words, the detection of any other fault type can also guarantee the detection of ENF. Though this conjecture has been proven wrong by Chen et al. [10], our observation with respect to ENF demonstrated that ENF at least can be revealed very easily as compared with other fault types. Lau and Yu [27] proved that LIF and LOF are the strongest fault types for Boolean specifications in the IDNF. Our experimental results showed that even for general Boolean specifications, LIF and LOF are still the most difficult fault types to be detected.

To further analyse the fault-detection effectiveness, we also conducted Bonferroni means separation tests on all testing strategies. Bonferroni test is an approach for multiple comparisons, based on which, we could verify whether the performance difference between the testing techniques under study is statistically significant. Generally speaking, Bonferroni test is a conservative approach that prevents data from being mistakenly identified as statistically significant. This approach has been widely used to quantify how various software engineering techniques are different from one another [16]. After the tests, these strategies are ranked and classified into different groups, as summarised in Table 2. The strategies in the same group imply that their difference in fault-detection effectiveness is not statistically significant. Note that one strategy can belong to

Table 2: Bonferroni mean separation tests on the fault-detection effectiveness for all testing strategies

				Testing strategy	Fault-detection effectiveness on all types of fault
A	B			MAX-B	99.80%
A	B			MAX-A	99.70%
A	B			MUMCUT-NCases&M2NFP	99.65%
A	B			MUMCUT-NCases&MS2NFP	99.50%
A	B			MUMCUT-M2NFP	99.48%
A	B			MUMCUT-NCases	99.45%
A	B			MUMCUT-MS2NFP	99.43%
A	B			MUMCUT-NCU	99.40%
A	B			MANY-B	99.38%
A	B			MUMCUT-UNC	99.38%
A	B			MUMCUT-CUN	99.35%
A	B			MUMCUT-UCN	99.33%
A	B			MANY-A	99.20%
	B	C		ONE	96.30%
		C	D	CUTPNFP	84.86%
		C	D	MIN	82.66%
			D	MUTP	69.24%
			D	MNFP	68.44%

different groups. For example, the ONE strategy is in Group B, which implies that its fault-detection effectiveness is not significantly different from the 13 best strategies in Group A (from MAX-B to MANY-A in Table 2, also in Group B). On the other hand, the ONE strategy is also in Group C, so its fault-detection effectiveness cannot be statistically distinguished from CUTPNFP and MIN, which, however, are significantly different from the 13 best strategies.

Based on Figure 2 and Table 2, we can make the following observations.

- MAX-B is the best method in terms of fault-detection effectiveness.
- The testing strategies of MANY-A, MANY-B, MAX-A, MUMCUT-CUN, MUMCUT-UCN, MUMCUT-UNC, MUMCUT-NCU, MUMCUT-NCases, MUMCUT-M2NFP, MUMCUT-MS2NFP, MUMCUT-NCases&M2NFP, and MUMCUT-NCases&MS2NFP have similar fault-detection effectiveness to MAX-B. All these 13 strategies have fault-detection effectiveness larger than 99% when considering all types of faults.
- The ONE testing strategy has a “not-so-bad” fault-detection effectiveness (fault-detection ratio larger than 96% when considering all types of faults).

However, the MIN strategy always has lower fault-detection effectiveness than the ONE strategy.

- It is hard to distinguish the fault-detection effectiveness of MUTP and MNFP.

Intuitively speaking, for almost all test case generation strategies, there is a trade-off between the number of used test cases and the number of detected faults. MAX-B always uses the most test cases (demonstrated in previous studies [40, 6]) among all 18 strategies, so it is not surprising at all that MAX-B can detect the most faults. The MIN strategy was proposed to minimise the number of test cases of the ONE strategy. Though both strategies guarantee the coverage of unique true points and near false points for all terms and literals, ONE always has a larger number of test cases as well as a larger number of detected faults than MIN. Each of MUTP, MNFP, and CUTPNFP was proposed to detect certain types of faults in IDNF Boolean specifications. They must be collectively used (that is, MUMCUT) for delivering a similar fault-detection effectiveness as MAX-B. Therefore, it is understandable that neither of them can provide a high fault-detection effectiveness on general Boolean specifications.

4.2 RQ2: Fault-detection efficiency

The experimental results for fault-detection efficiency is summarized in Figure 3, in which we use box-plots to represent the statistical distribution of fault-detection efficiency (measured by the ratio of the number of detected faults to the number of used test cases). Similar to Figures 1 and 2, the upper, middle, and lower lines of each box denote the third quartile, median value, and the first quartile of the fault-detection efficiency, respectively; while the top whisker, bottom whisker, and square dot represent the max, min, and mean value, respectively. As pointed out in Section 3.1, the main aim of the research related to RQ2 is to find the most efficient testing strategy in the sense of detecting most faults using a small number of test cases. As shown in Section 4.1, some of the testing strategies are not able to detect most of the faults, so we eliminated them when studying the fault-detection efficiency — we only included the results for the 13 best testing strategies (from MAX-B to MANY-A in Group A in Table 2). Bonferroni analysis was also performed to rank and group these strategies in terms of their fault-detection efficiency, as summarised in Table 3.

From the experimental data, we can have the following observations.

- In general, the four basic MUMCUT methods (MUMCUT-CUN, MUMCUT-UCN, MUMCUT-UNC, and MUMCUT-NCU) have the highest fault-detection efficiency among all testing strategies.

- Among all four basic MUMCUT methods, MUMCUT-UCN is generally the most efficient testing strategy, though its efficiency is not significantly higher than other three.

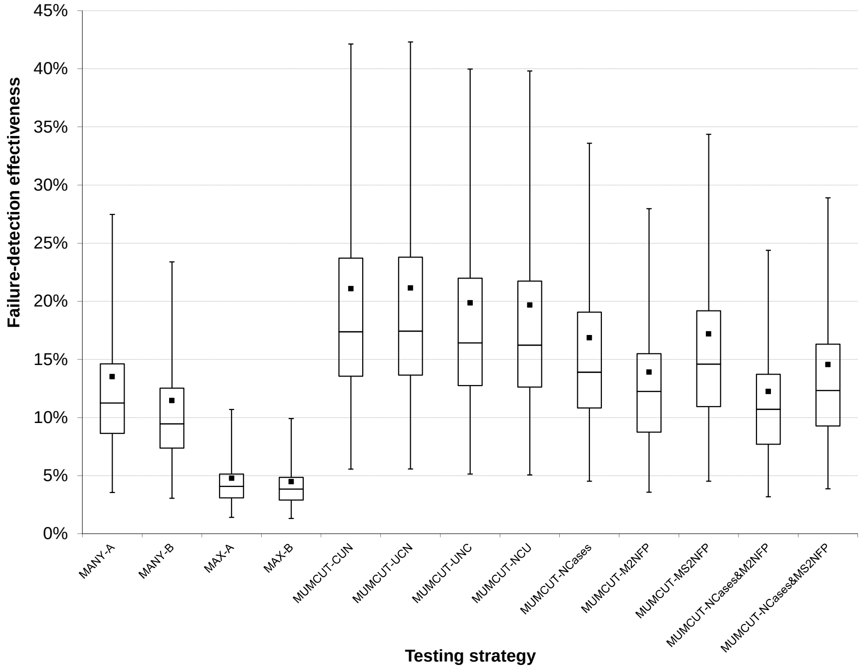

Figure 3: Fault-detection efficiency of the 13 best testing strategies

Table 3: Bonferroni mean separation tests on the fault-detection efficiency for the 13 best testing strategies

	Testing strategy	Fault-detection efficiency on all types of fault
A	MUMCUT-UCN	21.15%
A	MUMCUT-CUN	21.08%
A	MUMCUT-UNC	19.86%
A	MUMCUT-NCU	19.67%
A B	MUMCUT-MS2NFP	17.20%
A B	MUMCUT-NCases	16.87%
A B	MUMCUT-NCases&MS2NFP	14.56%
A B	MUMCUT-M2NFP	13.91%
A B	MANY-A	13.51%
A B	MUMCUT-NCases&M2NFP	12.25%
A B	MANY-B	11.46%
B C	MAX-A	4.78%
B C	MAX-B	4.49%

Table 4: Overall performance improvement of MUMCUT extensions

Testing strategy	Improvement of fault-detection effectiveness over MUMCUT-CUN	Improvement of fault-detection efficiency over MUMCUT-CUN
MUMCUT-NCases	0.101%	-0.199%
MUMCUT-M2NFP	0.126%	-0.340%
MUMCUT-MS2NFP	0.075%	-0.184%
MUMCUT-Ncases&M2NFP	0.301%	-0.419%
MUMCUT-Ncases&MS2NFP	0.151%	-0.309%

- MAX-B always has the lowest fault-detection efficiency among all the ranked strategies, and the efficiency of MAX-A is just marginally higher than that of MAX-B.

Based on the above observations, we can conclude that though MAX-B detects the most faults, its efficiency is very low. On the other hand, MUMCUT should be used for consistently delivering a high fault-detection efficiency in testing general Boolean specifications. In particular, MUMCUT-UCN is recommended to be the best choice.

4.3 Performance improvement and cost of MUMCUT extensions

As mentioned above, some faults in the general Boolean expressions cannot be detected by MUMCUT [36], and hence some extensions were proposed to improve its fault-detection effectiveness [34]. However, their improvements are not evaluated yet. Through this study, we want to answer whether these extensions are able to improve the fault-detection effectiveness without significantly increasing the number of used test cases.

We have provided a comprehensive comparison of all fault-based testing strategies for general Boolean expressions in terms of the fault-detection effectiveness and efficiency. We further analyse to what extent our MUMCUT extensions [34] can improve the performance over the original MUMCUT-CUN, as summarised in Table 4, where the performance improvements in terms of fault-detection effectiveness and efficiency are given in the 2nd and 3rd columns, respectively.

Generally speaking, the five MUMCUT extensions (MUMCUT-NCases, MUMCUT-M2NFP, MUMCUT-MS2NFP, MUMCUT-NCases&M2NFP, and MUMCUT-NCases&MS2NFP) have marginally higher fault-detection effectiveness than MUMCUT-CUN. Among them, MUMCUT-Ncases&M2NFP has the highest fault-detection effectiveness. However, the marginal improvement of the fault-detection effectiveness is actually at the cost of much more test cases. As shown in Table 4, all MUMCUT extensions have lower efficiency than MUMCUT-CUN.

Such observations reinforce that there is a trade-off between the fault-detection effectiveness and the number of test cases, and the MUMCUT methods fairly balance the trade-off.

5 Related Work

In this section, we introduce closely related work in fault-based testing of Boolean expressions.

5.1 Fault classes and their detection hierarchy

Since testing can manifest the presence of faults, an exhaustive testing is expected to be adequate to detect all possible faults. However, it is prohibitively expensive to realize complete adequacy testing [15]. Fault-based testing is a relative adequacy testing technique, which first assumes certain types of faults, and then designs test suites to detect these faults [28]. Boolean expressions play an important role in software specifications, and numerous efforts have been reported on how to efficiently test different types of faults present in Boolean specifications [33].

One category of research efforts have been made to identify the detection hierarchy of fault classes of Boolean specifications under some form. Fault class A is said to subsume fault class B, if and only if the test suite that can detect A must be able to detect B, denoted as $A \Rightarrow B$. That is, $TS(A) \supseteq TS(B)$ where $TS(A)$ and $TS(B)$ denote test suite that can guarantee the detection of fault class A and B, respectively. If this subsumption relation exists, one just needs to focus on test case generation for fault class A without considering the detection of fault class B. Therefore, understanding such a detection relationship is important, because it may be helpful to not only explain the experimental results, but also reduce the testing cost.

A pioneering work in this direction was done by Kuhn [25], who first reported the hierarchy of several fault classes where Boolean specifications are assumed to be DNF. Kuhn's hierarchy identifies the following subsumption relationships: (1) $VRF \Rightarrow VNF$; and (2) $VNF \Rightarrow ENF$. Here, VRF (Variable Reference Fault) refers to that a variable v_i is substituted by the other one v_j, where $v_i \neq v_j$. VRF is further classified as MCF (Missing Condition Fault) and ICF (Incorrect Condition Fault). Tsuchiya and Kikuno [39] further extended Kuhn's hierarchy by analysing the relationship between VRF and MCF, namely (1) $MCF \Rightarrow VNF$, when the fault happens to the same variable; (2) $VRF \Rightarrow MCF$ when the fault happens to the same variable and the faulty term has more than one variable. The identified hierarchy only considers variable related faults, which means all occurrences of a missing or referenced variable in the expression. However, it is possible that a fault is only related to a literal, namely one occurrence of a variable.

Lau and Yu [27] extended the previous work to include the relationships between term and literal related fault classes. They established a hierarchy, which shows the following relationships: (1) $LIF \Rightarrow LRF \Rightarrow LNF \Rightarrow TNF \Rightarrow ENF$; (2) $LIF \Rightarrow TOF \Rightarrow ORF[+] \Rightarrow ENF$; (3) $LOF \Rightarrow ORF[\cdot] \Rightarrow TNF$; (4) $TOF \Rightarrow LNF$; and (5) $LOF \Rightarrow LNF$. Here, ORF[+] refers to that a binary Boolean operator "+" (OR) immediately following a term is replaced by "·" (AND), and ORF[·] refers to that a binary Boolean operator "·" immediately following a literal is replaced by "+".

The above hierarchies assume that Boolean specifications are represented to be in DNF or IDNF. Okun et al. [31] extended the hierarchy to include clause related fault classes, and the proposed hierarchy applies to arbitrary expressions. A clause is either a Boolean variable or a relational expression. The proven relationships include: (1) CRF (Clause Reference Fault) ⇒ CNF (Clause Negation Fault); and (2) CNF ⇒ ENF. Kapoor and Bowen [24] attempted to extend the hierarchy to general Boolean specifications. They studied ten types of faults and proposed a hierarchy of fault classes for general Boolean specifications. However, their hierarchy was then proven by Chen et al. [10] to be incorrect. As a result, the correct relationships include: (1) $CDF \Rightarrow SA1$; (2) $CCF \Rightarrow SA0$; and (3) $VRF \Rightarrow VNF$.

A Boolean specification can be represented in different forms (such as DNF, IDNF, and general form), and faults that may be committed by programmers may involve Boolean variables, Boolean operators, Boolean literals, or terms [33]. Kaminski et al. [23] summarized common types of faults. However, most fault classes are hypothesized with respect to a given representation form, which calls for different analysis of the fault hierarchy, and affects the development of test case generation strategies. Recently, Paul and Lau [32] proposed a set of uniform definitions of fault classes, and they can be applied irrespective of the syntactic nature of Boolean expressions.

5.2 Test case generation strategies

The other category of research efforts have been made to develop efficient test case generation strategies for Boolean specifications [33]. The basic issue is on how to generate and select an efficient set of test cases so that the size of selected test cases is as small as possible, while they can detect faults as many as possible. Various test case generation strategies for Boolean specifications have been proposed, and they are grouped into *syntactic* and *semantic* [23]. The syntactic strategies normally require Boolean specifications under test to be in a specific form, and most notable ones include the family of basic meaningful impact strategies [40], MUMCUT [6], and MUMCUT extensions [34]. The semantic strategies do not require the Boolean expression to be in a particular form, and typical ones include sensitive strategies [18], MC/DC [13], BRO strategies [38], and the predicate coverage criteria such as predicate coverage (PC), clause coverage (CC), multiple condition coverage (MCC), active clause coverage (ACC), general active clause coverage (GACC), correlated active clause coverage (CACC), restricted active clause coverage (RACC), inactive clause coverage (ICC), general inactive clause coverage (GICC), and restricted inactive clause coverage (RICC) [1]. Kaminski et al. [23] analyzed the subsumption hierarchy among most test strategies, which covers the family of basic meaningful impact strategies [40], the family of MUMCUT [6], and the predicate coverage criteria [1].

A recent trend is to study the testing of general Boolean specifications. In our previous work [36], we evaluated the fault detection capability of MUMCUT strategy when it is used for general Boolean expressions, and reported five patterns of faults that cannot be detected by MUMCUT. We further analysed why the five patterns of faults cannot be detected by MUMCUT, and proposed extensions [34]. We further investigated the effectiveness of MUMCUT using randomly generated Boolean expressions [7], but have not compared MUMCUT with other testing methods. In this work, we took one step further to evaluate and compare the effectiveness of representative fault-based (syntactic) test case generation strategies when they are used for general Boolean expressions. The evaluation and comparison covered more test case generation strategies (including meaningful impact strategies, MUMCUT and MUMCUT variants, and MUMCUT extensions proposed in [34]), and covered more types of faults (ten types are considered, while only eight types are considered in [36]). Furthermore, the evaluation and comparison took a large size of randomly generated Boolean expressions as the subjects, which is different from those reported in existing literature.

One single fault assumption has been widely adopted in fault-based testing. However, a single fault in a general Boolean expression can give rise to a very large number of faults in the equivalent expression of IDNF, as observed in our previous work [7]. Lau et al. [26] investigated the detection conditions of combinations of two single faults in the expressions of IDNF, in which one is related to term and the other is related to literal, and evaluated the fault detection capability of some test case generation strategies (including ONE, MAX-A, MAX-B, and MUMCUT) using 38 double fault expressions. They found that all such faulty expressions, except two, can be detected by these strategies for single fault detection. Unlike their evaluation, our experiments actually covered multiple faults and also combinations of literal, terms, and operators related faults, and compared the fault detection capability of more test strategies that were designed for single fault detection using a large number of randomly generated expressions as subjects.

Gargantini and Fraser [20] proposed a test case generation approach for general Boolean expressions. The approach is actually a mutation-based method, and aims at detecting all ten types of faults investigated in previous studies [24, 10]. The generated test set is required to be capable of killing all possible mutants that are constructed by seeding certain types of faults into the Boolean expression under test. Unlike some other strategies such as MUMCUT, it is not theoretically proven that this approach can guarantee the detection of all faulty versions corresponding to certain fault types. The case study was conducted based on the mutants generated from 19 Boolean specifications [40], which are from only one subject program.

In some situations, constraints may exist among Boolean variables in a Boolean expression, and thus possibly result in some invalid test cases. Gargantini [19] illustrated this issue by providing several interesting examples, and proposed three strategies to handle the constraints, including (1) generating test cases without considering constraints and then removing invalid test cases; (2) modeling constraints as Boolean predicates, including predicates as further conjoint to the original expression, and generating test cases for the derived expression; (3) using a constraint solving technique to generate only valid test cases. The proposed strategies were evaluated and compared using seven Boolean expressions with variable dependencies selected from the specification of TCAS II [40]. In this study, we did not explicitly address constraints of Boolean expressions; however, a large number of sample expressions may partially cover some constraints that are treated as conjoints. In our future work, we will further investigate the impact of constraints on all fault-based test case generation strategies.

5.3 Evaluation and comparison of test strategies

Numerous studies were conducted to compare different test case generation strategies for Boolean expressions. For instance, Feng et al. [17] evaluated the subsumption relations among four types of testing strategies, including the testing strategy based on the ONE criterion, partition testing, decision table-based testing, and fault-based testing. Their study was based on two subject programs, and did not involve many Boolean expression-oriented testing strategies, such as MUMCUT, MAX-B, etc. Chen et al. [11] investigated the effectiveness of two IDNF-oriented testing strategies on general Boolean specifications, and proposed some extensions for the two strategies under study. However, their study only involved ONE and MUMCUT, and was conducted based on only one subject program. Similarly, Yu et al. [41] compared the performance of MUMCUT with MAX-A and MAX-B using a set of 80 Boolean expressions (20 from the specification of TCAS II [40], 20 from an LRU subject program, and 40 random). As compared with these previous studies, our work is the first one to evaluate and compare the effectiveness of a full family of fault-based test case strategies on a large amount of general Boolean specifications.

All fault-based test case generation strategies under evaluation have been implemented. Chen et al. [4] developed the BEAT (Boolean expression fault-based test case generator) system, which can be used to generate test cases from Boolean expression according to MUTP, MNFP, CUTPNFP, MUMCUT-CUN, MUMCUT-UCN, MUMCUT-UNC, and MUMCUT-NCU. Cheng [12] reported the implementation of MUMCUT extensions proposed in [34]. Arcaini et al. [3] proposed to improve the performance of test case generation for Boolean expression by using SAT/SMT solvers, which may be beneficial to the implementation of test case generation strategies. They also published a set of random Boolean expressions and a tool that can be used to generate test cases for ten types of faults proposed in [24, 10]. To support the evaluation of test case generation strategies for Boolean expressions, we developed a finite state machine-based parameterised generator [35], which can be used to generate a large size of general Boolean expressions, and the structure characteristics of the generated

Boolean expressions are adjustable. Similarly, Jenkins et al. [22] developed a fault evaluator to aid experimental evaluation of fault-based testing techniques for Boolean expressions.

6 Conclusion

Boolean expressions are widely used for describing conditions and decisions in software specifications. Many strategies have been proposed to generate test cases for detecting different types of faults in Boolean expressions. Most of the existing fault-based testing strategies are based on Boolean expressions in IDNF. However, any form can be used to represent a Boolean specification. In this paper, we evaluated and compared the effectiveness and efficiency of various fault-based test case generation strategies on general Boolean specifications.

In our experimental studies, we used a large amount of randomly generated Boolean expressions as the subjects. As compared with the subjects used in previous studies [40, 20, 11], these randomly generated expressions may reflect more aspects of Boolean specifications in the general form, and thus can give us a more comprehensive picture of the fault-detection effectiveness and efficiency of the testing strategies under investigation. On the other hand, it is possible that randomly generated Boolean expressions can show difference when compared with real-life ones. Our study showed that among all investigated strategies, four basic MUMCUT methods, five MUMCUT extensions, MANY-A, MANY-B, MAX-A, and MAX-B have the highest fault-detection effectiveness. Though these strategies detect a similar number of faults, MUMCUT-UCN uses the fewest test cases, and thus delivers the highest fault-detection efficiency. Another interesting observation from the experiments is that the extensions marginally improve the fault-detection effectiveness of the basic MUMCUT, while significantly increasing the number of used test cases. This further implies that there does exist the trade-off between fault-detection effectiveness and the testing cost. In summary, MUMCUT is the most efficient test case generation strategy, not only for Boolean specifications in IDNF, but also for general Boolean specifications. Therefore, MUMCUT is recommended to practitioners as the best fault-based testing strategy.

In our future work, we would like to take a step further towards the application of fault-based testing strategies in practice. Up to now, various fault-based testing strategies have been proposed to test Boolean expressions using fewer test cases, and their effectiveness has been comprehensively evaluated via a series of experiments presented in this work. However, conditions or decisions in a real-life program usually involve relational or arithmetical expressions which are defined on non-Boolean variables. How these strategies can be employed to select test cases for execution has not been well addressed, and calls for further efforts. In this context, we would like to further investigate their adoption in unit testing. This usually involves the abstraction of predicates as a Boolean expression, generation of abstract tests using the existing fault-based testing strategies, and refinement of abstract tests into applicable ones via constraint solver techniques. Additionally, constraints may happen to Boolean expressions as demonstrated by some of Boolean expressions of TCAS II [40], we are interested to know the impact of constraints on all fault-based test case generation strategies.

References

[1] P. Ammann and J. Offutt. *Introduction to software testing.* Cambridge University Press, Cambridge, UK, 2008.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 402–411, St. Louis, MO, USA, 15-21 May 2005. ACM, New York, NY, USA.

[3] P. Arcaini, A. Gargantini, and E. Riccobene. Optimizing the automatic test generation by sat and smt solving for boolean expressions. In *Proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 388–391, Lawrence, KS, USA, 6-10 November 2011. IEEE Computer Society, Los Alamitos, California, USA.

[4] T. Y. Chen, D. D. Grant, M. F. Lau, S. P. Ng, and V. R. Vasa. Beat: A web-based boolean expression fault-based test case generation tool. *International Journal of Distance Education Technologies*, 4(2):44–56, 2006.

[5] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse. Adaptive random testing: The ART of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

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

[7] T. Y. Chen, M. F. Lau, K. Y. Sim, and C.-A. Sun. On detecting faults for boolean expressions. *Software Quality Journal*, 17(3):245–261, 2009.

[8] T. Y. Chen, M. F. Lau, and Y. T. Yu. Mumcut: A fault-based strategy for testing boolean specifications. In *Proceedings of the 6th Asia Pacific Software Engineering Conference (APSEC 1999)*, pages 606–613, Takamatsu, Japan, 7-10 December 1999. IEEE Computer Society, Los Alamitos, California, USA.

[9] T. Y. Chen and Y. T. Yu. Constraints for safe partition testing strategies. *The Computer Journal*, 39(7):619–625, 1996.

[10] Z. Y. Chen, T. Y. Chen, and B. W. Xu. A revisit of fault class hierarchies in general boolean specifications. *ACM Transactions on Software Engineering and Methodology*, 20(3):13:1–13:11, 2011.

[11] Z. Y. Chen, B. W. Xu, and C. H. Nie. Comparing fault-based testing strategies of general boolean specifications. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007) - Volume 01*, pages 621–622, Beijing, China, 24-27 July 2007. IEEE Computer Society, Los Alamitos, California, USA.

[12] Q. Cheng. *Performance Analysis and Improvements of MUMCUT for General Boolean Specifications.* University of Science and Technology Beijing, Beijing, China, 2011. Master Thesis.

[13] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

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

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

[16] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[17] X. Feng, D. L. Parnas, T. H. Tse, and T. O'Callaghan. A comparison of tabular expression-based testing strategies. *IEEE Transactions on Software Engineering*, 37(5):616–634, 2011.

[18] K. A. Foster. Sensitive test data for logic expressions. *ACM SIGSOFT Software Engineering Notes*, 9(2):120–125, 1984.

[19] A. Gargantini. Dealing with constraints in boolean expression testing. In *Proceedings of 3rd Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA 2011)*, pages 322–327, Berlin, Germany, 21-25 March 2011. IEEE Computer Society, Los Alamitos, California, USA.

[20] A. Gargantini and G. Fraser. Generating minimal fault detecting test suites for general boolean specifications. *Information and Software Technology*, 53(11):1263–1273, 2011.

[21] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology*, 22(1):6:1–6:42, 2013.

[22] W. Jenkins, S. Vilkomir, and W. Ballance. Fault evaluator: A tool for experimental investigation of effectiveness in software testing. In *Proceedings of IEEE International Conference on Progress in Informatics and Computing (PIC 2000)*, pages 1077–1083, Shanghai, China, 10-12 December 2010. IEEE Computer Society, Los Alamitos, California, USA.

[23] G. Kaminski, W. Gregory, and P. Ammann. Reconciling perspectives of software logic testing. *Software Testing, Verification and Reliability*, 18(3):149–188, 2008.

[24] K. Kapoor and J. P. Bowen. Test conditions for fault classes in boolean specifications. *ACM Transactions on Software Engineering and Methodology*, 16(3):10:1–10:12, 2007.

[25] D. R. Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 8(4):411–424, 1999.

[26] M. Lau, Y. Liu, and Y. Yu. Detecting double faults on term and literal in boolean expressions. In *Proceedings of the Seventh International Conference on Quality Software (QSIC 2007)*, pages 117–126, Portland, Oregon, USA, 11-12 October 2007. IEEE Computer Society, Los Alamitos, California, USA.

[27] M. F. Lau and Y. T. Yu. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 14(3):247–276, 2005.

[28] L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, 1990.

[29] G. J. Myers. *The Art of Software Testing.* John Wiley & Sons Inc., New York, NY, USA, 1979.

[30] J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, 1992.

[31] V. Okun, P. E. Black, and Y. Yesha. Comparison of fault classes in specification-based testing. *Information and Software Technology*, 46(8):525–533, 2004.

[32] T. Paul and M. Lau. Redefinition of fault classes in logic expressions. In *Proceedings of International Conference on Quality Sofware (QSIC 2012)*, pages 144–153, Xian, China, 27-29 August 2012. IEEE Computer Society, Los Alamitos, California, USA.

[33] C.-A. Sun and Q. S. Cheng. A survey on fault-based testing techniques for boolean expressions. *Chinese Journal of Computer Science*, 40(3):16–23, 2013.

[34] C.-A. Sun, Y. Dong, R. Lai, K. Y. Sim, and T. Y. Chen. Analyzing and extending mumcut for fault-based testing of general boolean expressions. In *Proceedings of the 6th IEEE International Conference on Computer and Information Technology (CIT 2006)*, page 184, Seoul, Korea, 20-22 September 2006. IEEE Computer Society, Los Alamitos, California, USA.

[35] C.-A. Sun and K. Y. Sim. An FSM-based parameterized generator for general boolean expressions. In *Proceedings of International Computer Engineering Conference (ICENCO 2004)*, pages 119–126, Cairo, Egypt, 29-30 December 2004. Cairo University, Cairo, Egypt.

[36] C.-A. Sun, K. Y. Sim, T. Tse, and T. Chen. An empirical evaluation and analysis of the fault-detection capability of mumcut for general boolean expressions. In *Proceedings of International Computer Symposium (ICS 2004)*, pages 926–932, Taipei, Taiwan, 15-17 December 2004. TTU CSE Extenics and Fuzzy System Laboratory, Taipei, Taiwan.

[37] C.-A. Sun, G. Wang, K.-Y. Cai, and T. Y. Chen. Distribution-aware mutation analysis. In *Proceedings of 9th IEEE International Workshop on Software Cybernetics (IWSC 2012)*, pages 170–175, Izmir, Turkey, 16-20 July 2012. IEEE Computer Society, Los Alamitos, California, USA.

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

[39] T. Tsuchiya and T. Kikuno. On fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 11(1):58–62, 2002.

[40] E. J. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, 1994.

[41] Y. T. Yu, M. F. Lau, and T. Y. Chen. Automatic generation of test cases from Boolean specifications using the mumcut strategy. *The Journal of Systems and Software*, 79(6):820–840, 2006.