**BEAT: Boolean Expression fAult-based Test Case Generator**

T.Y. Chen, D.D. Grant, M.F. Lau*, S.P. Ng, V.R. Vasa

School of Information Technology
Swinburne University of Technology
Hawthorn, VIC. 3122, Australia

{tchen,dgrant,elau,sng,rvasa}@it.swin.edu.au

---

**Abstract** This paper presents a system which generates test cases from Boolean expressions. The system is based on the integration of several fault-based test case selection strategies developed by us. Our system generates test cases that are guaranteed to detect all single operator fault and all single operand faults when the Boolean expression is in irredundant disjunctive normal form. Apart from being an automated test case generation tool developed for software testing practitioners, this system can also be used as a training or self-learning tool for students as well as software testing practitioners.

**Index Terms**— Black-box testing, Boolean expression, fault-based testing, specification-based testing, test case generator

---

**I. INTRODUCTION**

Boolean expression plays several important roles in software development. On one hand, Boolean expressions can be found in software specifications. For example, Leveson et al. [4] use "AND-OR" table to specify the requirements of a Traffic Collision Avoidance System, TCAS II. As explained in Weyuker et al. [7], an "AND-OR" table is just one of the many possible representations of a Boolean expression. On the other hand, program predicates such as pre-conditions and conditional statements can be modelled as Boolean expressions (for example, see Chilenski and Miller [2]).

Recently, several fault-based test case selection strategies [1],[2],[3],[6],[7] have been proposed. These strategies are considered to be fault-based because test cases are generated to detect particular types of fault. For example, Tai [6] proposes to use attribute grammar to generate test cases that can detect Boolean operator faults and relational operator faults related to the predicates in source code. On the other hand, Weyuker et al. [7] adopt an approach which is different from Tai’s approach in the sense that they propose to generate test cases from specifications rather than from the implementation or the program source. One of the advantages is that the generated test cases do not depend on the program source. They propose a family of strategies to automatically generate test cases from Boolean specifications, that is, specifications written in Boolean expressions.

Chen and Lau [1] develop three test case selection strategies to generate test cases from Boolean specifications. The generated test cases can guarantee to detect seven types of faults to be described in Section II. These strategies are the MUTF strategy, the CUTPNFP strategy and the MNFP strategy [1]. This paper describes the BEAT (Boolean Expression Fault-based Test case generator) system which generates test cases from Boolean expression according to these three strategies. Moreover, we demonstrate how BEAT can be used to generate test cases from Boolean specifications and we discuss how the system can be used to help students and software testing practitioners to acquire the knowledge of generating test cases from Boolean expressions.

Section II presents the notation, terminology and the seven types of fault considered in this paper. In Section III, we describe the three fault-based test case selection strategies developed by us [1] and, propose guidelines on using these strategies as well as integrating these strategies together for the generation of comprehensive test sets. Section IV describes the BEAT system. In Section V, we illustrate with an example how BEAT can be used as a test case generator and a learning tool. Section VI concludes the paper.

**II. BACKGROUND**

A. Notation and Terminology

Throughout this paper, 1 and 0 are used to denote "TRUE" and "FALSE", respectively. The set of all truth values is denoted by B, that is, B = {1, 0}. For a Boolean expression with n variables, there are altogether 2^n possible inputs, making it difficult to conduct exhaustive testing even when n is moderately large. The set of all possible inputs is denoted by B^n. The Boolean operators AND, OR and NOT are denoted by '*' and '−', respectively. The operator '−' may be omitted from the expression whenever there is no ambiguity.

A Boolean expression can be represented in many different ways. The two most common standard forms are the disjunctive normal form, DNF (also called the sum-of-product form) and the conjunctive normal form, CNF (also called the product-of-sum form). For example, the Boolean expression ab+ac is in DNF whereas (a+b)(a+c) is in CNF. There is a dual relationship between CNF and DNF.

Test case generation strategies proposed by Weyuker et al. [7] and Chen and Lau [1] are mainly based on the irreducible disjunctive normal form (IDNF), a special form of DNF. A Boolean expression in DNF is said to be irreducible if (1) none of its terms may be omitted from the expression; and (2) none of its literals may be omitted from any term in the
expression [5]. For example, the Boolean expression $a + b + ab + c$ is in IDNF whereas $abc + b + ab + c$ is not because these two Boolean expressions are equivalent. In this paper, we concentrate our discussion in the form of DNF.

Let $S$ be a Boolean expression in DNF given by $S = p_1 + p_2 + \ldots + p_m$, where $m$ is the total number of terms in $S$ and $p_i (i = 1, \ldots, m)$ is the $i$-th term of $S$. A point $\tilde{r} \in \mathbb{B}^k$ is a true (false) point of $S$ if $S$ evaluates to 1 (0) on $\tilde{r}$.

For each term $p_i$ ($i = 1, \ldots, m$) of $S$, a point $\tilde{r} \in \mathbb{B}^k$ is a unique true point of $p_i$ if (1) $p_i$ evaluates to 1 and (2) all other terms evaluate to 0 on $\tilde{r}$. Moreover, a true point of $S$ is an overlapping true point of $S$ if it is not a unique true point. The set of all unique true points of $p_i (i = 1, \ldots, m)$ of $S$ are denoted by $\text{UTP}(S)$. The set of all unique true points of $S$ is denoted by $\text{UTP}(S)$, that is, $\text{UTP}(S) = \bigcup_i \text{UTP}(S)$. The set of all overlapping true points of $S$ is denoted by $\text{OTP}(S)$.

Suppose that the $i$-th term $p_i$ in $S$ is given by $p_i = x_1^{(i)} x_2^{(i)} \ldots x_k^{(i)}$, where $x_j^{(i)}$ denotes the $j$-th literal in $p_i$ ($j = 1, \ldots, k_i$) and $k_i$ is the total number of literals in $p_i$. For $j = 1, \ldots, k_i$, $p_{i,j} = x_1^{(i)} \ldots \hat{x}_j^{(i)} \ldots x_k^{(i)}$ is used to denote the term obtained from $p_i$ by negating its $j$-th literal $x_j^{(i)}$. A point $\tilde{r} \in \mathbb{B}^k$ is a near false point for the $j$-th literal $x_j^{(i)}$ of the $i$-th term $p_i$ in $S$ if (1) $p_{i,j}$ evaluates to 1 and (2) $S$ evaluates to 0. Moreover, a false point of $S$ is a remaining false point of $S$ if it is not a near false point at all. The set of all near false points for the $j$-th literal $x_j^{(i)}$ of the $i$-th term $p_i$ in $S$ is denoted by $\text{NFP}_{i,j}(S)$. The collection of all near false points of $S$ is denoted by $\text{NFP}(S)$, that is, $\text{NFP}(S) = \bigcup_{i,j} \text{NFP}_{i,j}(S)$. The set of all remaining false points of $S$ is denoted by $\text{RFP}(S)$.

For any Boolean specification $S$, test cases are selected from the Boolean space $\mathbb{B}^k$ to verify whether the software developer actually implements the software correctly. These test cases will then be used to verify the corresponding implementation $I$. If a test case $\tilde{r} \in \mathbb{B}^k$ can distinguish between $S$ and $I$ (that is, $S(\tilde{r}) \neq I(\tilde{r})$), $\tilde{r}$ is considered to be a good test case because it can demonstrate a failure in the implementation.

B. Various Types of Faults

The following seven types of fault with a single operator or operand fault are considered in previous research [1],[3],[7].

1. **Operator Reference Fault (ORF)**

   The Boolean operator ‘+’ (‘-’) is wrongly implemented as ‘·’ (‘·’). For example, $ab + cd$ is incorrectly implemented as $abcd + a + b + cd$.

2. **Expression Negation Fault (ENF)**

   The Boolean expression is wrongly implemented as its negation.

3. **Term Omission Fault (TOF)**

   A particular term is omitted during the implementation. For example, $ab + cd + ef$ is wrongly implemented as $ab + cd$.

4. **Literal Negation Fault (LNF)**

   A literal in a particular term is wrongly implemented as its negation. For example, $abc + de$ is implemented as $\overline{abc} + de$.

5. **Literal Omission Fault (LOF)**

   A literal in a particular term is omitted during the implementation such as $abcd + de$ being implemented as $abc + de$.

6. **Literal Insertion Fault (LIF)**

   A literal not appearing in a particular term is inserted into that term. For example, $abc + de$ is incorrectly implemented as $abcd + de$.

7. **Literal Reference Fault (LRF)**

   A literal in a particular term is replaced by another literal not appearing in the term during the implementation. For example, $abc + de$ is incorrectly implemented as $abd + de$.

III. TEST CASE SELECTION STRATEGIES USED IN BEAT

This section describes the three test case selection strategies, which are used in the BEAT system. They are the MUTP strategy, the CUTPNFP strategy and the MNFP strategy [1]. They have been proposed for the detection of the seven types of fault mentioned in Section II. We illustrate each of these strategies using the Boolean expression $ab + cd + ef$ as an example.

The MUTP (Multiple Unique True Point) strategy requires the selection of unique true points from every $\text{UTP}(S)$ such that all possible truth values (that is, 0 and 1) of every variable not occurring in $p_i$ are covered. It guarantees the detection of ENF, TOF, LNF, LIF, and those ORF where the OR operator ('·') is wrongly implemented as the AND operator ('·'). For $ab + cd$, a set of unique true points for the first term $ab$ is $\text{UTP}(S) = \{1100, 1110, 0111, 1010, 1110, 1011, 1111\}$. In order to satisfy the MUTP strategy on the first term $ab$ of the Boolean expression $ab + cd$, we need to select unique true points from $\text{UTP}(S)$ to cover 0 and 1 for $c$ and $d$. One of the many possible test sets is $\{1101, 1110\}$. Similarly, a test set that can satisfy the MUTP strategy for the second term $cd$ is $\{0111, 1111\}$. Thus, $\{1101, 1110, 0111, 1111\}$ satisfies the MUTP strategy for the expression $ab + cd$.

The CUTPNFP (Corresponding Unique True Point and Near False Point pair) strategy requires a unique true point to be selected from $\text{UTP}(S)$ and a near false point to be selected from $\text{NFP}(S)$ so that the unique true point and near false point only differ in the corresponding value of $x_j^{(i)}$ in $p_i$, of $S$, for each possible pair of $\text{UTP}(S)$ and $\text{NFP}(S)$ ($i = 1, \ldots, m$ and $j = 1, \ldots, k_i$), where $k_i$ denotes the number of literals in the $i$-th term $p_i$ of $S$. In other words, a pair of unique true point and near false point is selected. Such a pair of unique true point and near false point is said to be a corresponding pair of unique true point and near false point. Whenever the CUTPNFP strategy is satisfied, the corresponding LRF can be detected, as well as ENF, TOF, ORF, LNF, and LOF. A possible set of test cases satisfying the CUTPNFP strategy for $ab + cd$ may consist of the test cases $1101, 0101, 1001, 0111,$
0101, and 0110 from $UTP_1(S)$, $NFP_{1,4}(S)$, $NFP_{1,6}(S)$, 
$UTP_2(S)$, $NFP_{2,4}(S)$, and $NFP_{2,6}(S)$, respectively.

The MNFP (Multiple Near False Point) strategy requires the selection of near false points from $NFP_{1,4}(S)$ such that all possible truth values of every variable not occurring in $p_i$ are covered. It guarantees the detection of ENF, LNF, LOF, and possible truth values of every variable not occurring in

[Equations and variables]

It consists of $ORF$ where the Boolean operator '+' is replaced by the operator '+'. When the CUTPNFP strategy cannot be applied, both MUTP and MNFP strategies can be combined to guarantee the detection of the corresponding LRF [1]. A possible set of test cases satisfying the MNFP strategy may consist of 0101, 0110 (both from $NFP_{1,4}(S)$), 1001, 1010 (both from $NFP_{1,6}(S)$), 0101, 1001 (both from $NFP_{2,4}(S)$), 0110, and 1010 (both from $NFP_{2,6}(S)$).

As mentioned, none of the three strategies alone can guarantee to detect all seven types of fault. However, when they are applied together, all these faults could be detected. The BEAT system is designed to allow users to choose any combination of the three strategies for generating test cases from the input Boolean expression. Moreover, if users choose more than one strategy, they can specify the ordering of the strategies to be applied. In such situations, an incremental approach rather than taking the union of the resulting test sets generated by each individual strategy. Let us illustrate the incremental approach with the special order of applying the MUTP, the CUTPNFP and the MNFP strategies. First, we generate a test set that satisfies the MUTP strategy. Second, for the satisfaction of the CUTPNFP strategy, we need to find corresponding pairs of unique true point and near false point. Since some unique true points are already found in the first stage, we use these generated unique true points to find extra near false points in forming the corresponding pairs. If all corresponding pairs can be found, we proceed to the next stage. However, after all those generated unique true points are considered and there are still some corresponding pairs that cannot be found, we need to generate extra unique true points and near false points to satisfy the CUTPNFP strategy. This process continues until all corresponding pairs can be found or all unique true points have been considered. Finally, for the satisfaction of the MNFP strategy, extra near false points are generated, if necessary, so that the collection of those near false points generated in the CUTPNFP stage and the newly generated near false points collectively satisfies the MNFP strategy. In each phase, we use a greedy approach to generate those unique true points or those near false points that can satisfy the corresponding strategy. Due to the limitations on the length of the paper, the detailed steps are omitted.

IV. THE BEAT SYSTEM

BEAT is a fault-based test case generation tool developed in Java. It generates test cases from a Boolean expression. The current version is developed as an applet and can be accessed using a Web browser that has a Java plug-in version 1.3 or later installed. The system accepts general Boolean expressions and translates it into IDNF, if necessary. Test cases are then generated according to the chosen fault-based strategies.

To support extensibility and maintainability, the entire tool has been developed modularly and consists of four main components, namely, Expression Parser, Expression Transformer, Test Set Generator and User Interface. Fig. 1 shows how these four components interface with each other. In what follows, we briefly describe the components, their responsibilities and their limitations.

The Expression Parser is to parse the input Boolean expressions. The Boolean Expression Parser component is developed using JavaCC version 1.0, which is a full-featured Java based Compiler-Compiler. This tool allows for the generation of a parser based on a formal extend BNF grammar specification. It enables us to enhance and extend the grammar of the input expressions. Moreover, Boolean operators 'AND', 'OR', and 'NOT' are inputted as '·', '+', and '!' (See Table I).

The Expression Transformer component makes use of the Expression Parser component to transform a Boolean expression into an equivalent IDNF. After the transformation, capitalized variable is used to represent the negation of the corresponding variable. For example, "'a" will be shown as "A".

Based on the resultant IDNF, the Test Set Generator component generates test sets according to the chosen strategies. Users may choose to generate a test set that satisfies a particular strategy (say, the MUTP strategy) or any combinations of the three strategies (say, the MUTP strategy and the MNFP strategy but not the CUTPNFP strategy).

The User Interface component glues all the above three components together to ensure a simple and easy-to-use interface for users.

Fig. 2 shows a screen shot of the BEAT system with $(a + \overline{a}b)(c + d)$ as input. It should be noted that "NOT a" should be entered as "'a" as mentioned before. The output of the DNF and IDNF are shown in Fig. 2. The resultant IDNF is $ac+ad+bc+bd$. The second half of the screen shows further information related to the IDNF, including the number of terms, the number of variables, the total number of possible inputs, the sets $UTP_1(S)$ for each term, the sets $NFP_{1,4}(S)$ for

![Diagram of components](image)

**TABLE I BOOLEAN EXPRESSION FOR BEAT**

<table>
<thead>
<tr>
<th>Boolean Expression</th>
<th>Inputs to BEAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(a \text{ OR } b) \text{ AND } (\overline{b} \text{ OR } c)$</td>
<td>$(a + b) : (\overline{b} + c)$</td>
</tr>
<tr>
<td>NOT $(a \text{ OR } b)$</td>
<td>$(\overline{a} + b)$</td>
</tr>
<tr>
<td>$(a \text{ AND } b) \text{ OR } c$</td>
<td>$(a \cdot b) + c$</td>
</tr>
</tbody>
</table>
Collision, a traffic collision avoidance unit along the road either left or right in the junction whereas vehicles travelling road junction. Vehicles travelling along the road RI must consider the following traffic control monitoring system:

\[ NFP(S) \]

A Boolean expression three strategies, a possible test set returned by the set

\[ RFP(S) \]

there are altogether four sets of unique true points, one for each term, namely \( UTP_1(S), UTP_2(S), UTP_3(S), \) and \( UTP_4(S) \). As shown in Fig. 2, the unique element in \( UTP_1(S) \) for the third term \( bc \) is 0110, meaning that, \( a \) is “FALSE” (0), \( b \) is “TRUE” (1), \( c \) is “TRUE” (1), and \( d \) is “FALSE” (0). As a reminder, the system always sorts the variable alphabetically so that the ‘lowest’ variable (in this case, \( a \)) will always correspond to the first character of the binary string. Similarly, there are altogether eight different sets of near false points, \( NFP_{ij}(S) \) (for \( i = 1, 2, 3, 4 \) and \( j = 1, 2 \)). Figure 2 shows that the set \( NFP_{2,2}(S) \) has 2 elements, namely 1100 and 0100. Moreover, the set \( OTP(S) \) of overlapping true points contains 5 elements, namely 0111, 1111, 1110, 1101, and 1011, and the set \( RFP(S) \) of remaining false point contains only 1 element, which is 0000.

Suppose a user selects to generate test cases that satisfy all three strategies, a possible test set returned by BEAT for the Boolean expression \( (a + ab)(c + d) \) is \{1010, 0010, 0011, 1000, 1100, 0001, 0110, 0100, 0101\}.

V. USAGE OF THE BEAT SYSTEM

A. Test Case Generation

To illustrate the test case generation of BEAT, let us consider the following traffic control monitoring system:

Fig. 3 shows two traffic lights L1 and L2 installed in a T-road junction. Vehicles travelling along the road R1 must turn either left or right in the junction whereas vehicles travelling along the road R2 must go straight. In order to avoid traffic collision, a traffic collision avoidance unit (TCAU) will be used to monitor the traffic lights in the junction. The road junction is unsafe if any of the following situations occurs:

1. L1 is “Amber” and L2 is not “Red”,
2. L1 is “Green” and L2 is not “Red”,
3. L2 is “Amber” and L1 is not “Red”, or
4. L2 is “Green” and L1 is not “Red”.

Under any of the above situations, the TCAU will signal an alarm in the traffic control office. The TCAU has six inputs, namely L1R, L1A, L1G, L2R, L2A, and L2G. The inputs L1R, L1A, and L1G represent that the traffic light of L1 is “Red”, “Amber” and “Green”, respectively. The signal of each input is either “TRUE” or “FALSE”. For example, if L1R is “TRUE”, the traffic light of L1 is “Red”. It is assumed that exactly one of L1R, L1A, and L1G is “TRUE” at any time. Otherwise, the traffic light L1 is not in proper working condition and it shall signal the control office for repair. Hence, when the light of L1 is not “Red”, it is either “Amber” or “Green”. Similar situations apply for traffic light L2.

In this example, we only concentrate on the specification of the TCAU rather than the controlling of the proper working of the traffic lights L1 and L2. One way of specifying the “unsafe” condition of the TCAU system in normal expression can be given by:

\[ Unsafe_S = (L1A AND (L2A OR L2G)) \]
\[ OR (L1G AND (L2A OR L2G)) \]
\[ OR (L2A AND (L1A OR L1G)) \]
\[ OR (L2G AND (L1A OR L1G)) \]

It can be further simplified to the following:

\[ Unsafe_S = (L1A AND L2A) OR (L1A AND L2G) \]
\[ OR (L1G AND L2A) OR (L1G AND L2G) \]

(1)

The four variables in (1) are L1A, L1G, L2A, and L2G. If we choose to generate test cases for all three strategies, one possible test case generated by BEAT is \{1010, 0010, 0011, 1000, 1100, 0001, 0110, 0100, 0101\} where, for
example, the test case '1010' means that L1A is 'TRUE' (1), L1G is 'FALSE' (0), L2A is 'TRUE' (1) and L2G is 'FALSE' (0).

8. BEAT as a learning tool

The BEAT system also serves as a Web-based learning tool generating test cases from Boolean expressions. Since the conversion of a Boolean expression to various standard forms requires knowledge of Boolean algebra, and most developed test case selection strategies are based on the mathematical vigour of set theory, generating test cases from Boolean expressions is usually regarded as a very tedious task even for experienced practitioners, in particular when there are many complicated Boolean expressions. For beginners, such tasks are often error-prone; there is no easy way to detect any mistake in the process and it is difficult to validate the final test sets.

After learning the methodologies, students or trainees require practicing exercises to achieve their proficiency and competency. The BEAT system provides a very economical solution to satisfy their needs. They can use BEAT to practise their skills on

1. transforming a given Boolean expression to its equivalent DNF or IDNF, and
2. generating test cases that satisfy the corresponding test case selection strategies.

First, in the context of transforming a Boolean expression into DNF and IDNF, since the BEAT system also displays the corresponding equivalent DNF and IDNF of the original expression, it assists students in checking the correctness of their constructed expressions in DNF and IDNF. This helps to improve their learning process.

Second, in the context of generating test cases from Boolean expressions, it is common for students and even software testing practitioners to make mistake due to the tediousness of the generation process. As a learning tool of test case generating system, BEAT displays the various sets of points P^n in addition to the set of test cases for the input Boolean expression. This includes the sets of unique true points for each term, the sets of near false points for each literal in each term, the set of overlapping true points and the set of remaining false points. By studying these useful sets of points, students can follow the subtle relationships among the MUTP, CUTPNFP and MNFP strategies, and improve their understanding. The BEAT system also benefits all users, including the experienced practitioners, by providing an efficient checking mechanism for the test case generation process.

VI. CONCLUSION AND FUTURE WORK

This paper reports the development of the BEAT system. The system adopts a fault-based approach to automatically generate test cases from Boolean expressions. It is based on the integration of the MUTP, CUTPNFP and MNFP strategies [1].

During the development phase of the BEAT system, we recognise that, by combining these three strategies, the generated test sets may not be the smallest nor have the highest fault-detecting efficiency in terms of detecting more faults with fewer test cases. Nonetheless, we demonstrate how the system can be used to generate test cases effectively from Boolean expressions for users of various levels of competency.

We also discuss how the system can be used as a learning tool by helping students and software testing novices to better understand the process of generating test cases from Boolean expressions using our proposed strategies.

In the future, we plan to measure the effectiveness of BEAT as a learning tool for students studying a software testing course in university. Based on the outcomes of the experiment, we should be able to further improve the efficiency, performance and user-friendliness of BEAT. In addition to being a self-learning tool, the BEAT system can also serve as a self-assessment tool for students and trainees during the course of study or training to generate test cases from Boolean expressions. However, this is considered as beyond the scope of this paper and hence not included here.

REFERENCES