Abstract

A computer program consists of statements, such as IF and WHILE statements, that contain conditions, which are combinations of Boolean and relational expressions. A testing approach, referred to as condition testing, is to test a program by focusing on testing the conditions in this program. A number of condition testing strategies have been developed, but they are not effective for detecting errors in complicated conditions. In this paper, we define two condition testing strategies, based on the detection of Boolean and relational expression errors in a condition. For these two condition testing strategies, we show some theoretical properties and explain why they are practical and effective for testing programs containing complicated conditions.

1. Introduction

A computer program consists of statements, such as IF and WHILE statements, that contain conditions (or predicates), which are combinations of Boolean and relational expressions. A number of testing strategies focusing on testing the conditions in a program have been developed. This testing approach is referred to as the condition testing approach. Existing condition testing strategies, however, are ineffective for detecting errors in complicated conditions. In this paper, we describe two condition testing strategies, based on the detection of Boolean and relational expression errors in a condition.

This paper is organized as follows. The remainder of this section provides several basic definitions. Section 2 surveys existing condition testing strategies. Section 3 reports some of research results on test generation for Boolean expressions. Section 4 shows a normal form, called test-constraints, for defining condition testing strategies. By combining test strategies for Boolean and relational expressions and by applying the notion of test-constraints, sections 5 and 6 define two condition strategies and discuss their effectiveness of error detection. Finally, section 7 concludes this paper. Due to space limitation, the proofs of theorems in this paper are omitted.

* The author is serving as the director of the Software Engineering Program at the National Science Foundation, Washington D.C., 20550.
* This work was supported in part by NSF grant CCR-8907807.
Complete Branch Testing

For a compound condition, say C, this strategy requires that the true and false branches of C itself and of each simple condition in C be executed at least once [Mye79, Tai83]. If C contains n simple conditions, the number of tests for C required by complete branch testing is between two and 2^n. Two tests for C can satisfy this testing strategy if, for C and every simple condition in C, these two tests produce opposite outcomes and thus cover the true and false branches. (Thus, complete branch testing does not necessarily require more tests than branch testing.) At most n+2 tests for C are required to satisfy the complete branch testing strategy, since the first test covers one branch of C and of each simple condition in C and the worst case is that n+1 additional tests are required to cover the remaining n+1 branches of C and simple conditions in C. Note that complete branch testing does not focus on the detection of specific types of errors in a compound condition.

Exhaustive Testing

For a Boolean expression with n, n≥0, variables, this strategy requires all of 2^n possible inputs. This strategy is practical only when n is very small. This testing strategy could be extended to test a condition containing relational expressions, if each relational expression is viewed as a distinct Boolean variable. However, this extended strategy does not deal with the detection of errors in relational expressions.

Relational Operator Testing

For a relational expression, say (E1 < rop> E2), this strategy requires three tests satisfying the following requirements [Kay82]:

(1) one test makes the value of E1 greater than that of E2,
(2) one test makes the value of E1 less than that of E2, and
(3) one test makes the value of E1 equal to that of E2.

If rop is incorrect and E1 and E2 are correct, then this strategy guarantees the detection of the relational operator error. We now consider the application of this testing strategy to test relational expressions in a compound condition. For a condition C with n, n≥0, relational expressions, the number of tests for C required by relational operator testing is between three and 3^n. Three tests for C can satisfy this testing strategy if, for every relational expression in C, these three tests satisfy requirements (1), (2), and (3) respectively. At most 2^n+1 tests for C are required to satisfy this testing strategy, since the first test satisfy one of (1), (2), and (3) for each relational expression in C and the worst case is that 2^n additional tests are required to cover the remaining two requirements for each of the n relational expressions in C. Note that relational operator testing does not focus on the detection of Boolean and arithmetic expression errors.

Relational Expression Testing

This strategy is the same as the relational operator testing strategy except that requirements (1) and (2) are modified as follows [Pos84, How82]:

(1) one test makes the value of (E1 - E2) greater than zero and less than or equal to BOUND, where BOUND is a positive number, and
(2) one test makes the value of (E1 - E2) less than zero and greater than or equal to -BOUND.

A smaller value of BOUND will make the two tests for (1) and (2) more effective for detecting errors in E1 and E2. (If BOUND is set to infinity, then this strategy is equivalent to relational operator testing.) Like relational operator testing, this testing strategy can be applied to test relational expressions in a compound condition. The earlier discussion on relational operator testing holds for relational expression testing except that the latter focus on the detection of both relational operator errors and arithmetic expression errors in a relational expression.

Domain Testing

For a relational expression (E1 < rop> E2) with a total of n distinct arithmetic variables in E1 and E2, the basic domain testing strategy [Tai87] requires that each domain boundary determined by (E1 < rop> E2) be tested with n+1 points. This testing strategy requires much more tests than relational expression (or operator) testing and is more effective for detecting relational expression errors. Like relational expression (or operator) testing, domain testing does not deal with the detection of Boolean expression errors in a compound condition.

3. Test Generation for Boolean Expressions

The problem of how to generate effective tests for Boolean expressions was studied in [Pos84, Tai87, Sub89]. Here we describe some of our research results reported in [Tai87, Sub89]. Let a singular Boolean expression (SBE) be a Boolean expression in which each Boolean variable occurs only once. Algorithms SBE_MIN and SBE_MINSEN, described in [Tai87], are two test generation algorithms for SBE's. For an SBE B with n, n≥0, Boolean variables, SBE_MIN (SBE_MINSEN) produces a test set, denoted as SBE_MIN(B) (SBE_MINSEN(B)), with at most n+1 tests. SBE_MIN(B) and SBE_MINSEN(B) may be different, but they have the same number of tests. It was shown that SBE_MIN(B) is a minimum test set for B that guarantees the detection of multiple Boolean operator errors in B and that SBE_MINSEN(B) is very effective for, if it does not guarantee, the detection of multiple Boolean operator errors in B. To illustrate the test sets produced by SBE_MIN and SBE_MINSEN, let B1 denote the Boolean expression (IIa(12+I3)), where a1, a2 and a3 denote distinct Boolean variables. SBE_MIN(B1) is ((t,t,f), (f,t,f), (f,f,t)) and SBE_MINSEN(B1) is ((t,t,t), (t,f,f), (t,f,t), (f,t,f), (f,f,t)).

565
The following approach, based on the mutation analysis method [DeM78], was used in our experiments for evaluating the effectiveness of test generation algorithms for Boolean expressions. For a Boolean expression B and a test generation algorithm TEST, let TEST(B) be the test set produced by TEST for B. Let BE_SET be a set of n, n>0, mutually non-equivalent Boolean expressions with the same set of Boolean variables. For each Boolean expression B in BE_SET, other Boolean expressions in BE_SET are considered as mutants of B. A mutant B' of a Boolean expression B is said to be killed by TEST(B) if B and B' produce different results on at least one element of TEST(B). (In other words, TEST(B) can distinguish B from B'.) The percentage of dead mutants of B in BE_SET according to TEST(B) is defined as

\[ \text{(the number of B's mutants in BE_SET killed by TEST(B)) / n.} \]

Below are some of our experimental results for evaluating the effectiveness of SBE_MIN and SBE_MINSEN [Ta87, Su89]. SBE_3 denotes a set of 64 mutually non-equivalent SBE's with 3 variables. For each Boolean expression in SBE_3, we applied SBE_MIN (SBE_MINSEN) to generate a test set and computed the percentage of dead mutants in SBE_3 according to this test set. The following table shows the average of the 64 dead mutant percentages by using SBE_MIN (SBE_MINSEN):

<table>
<thead>
<tr>
<th>set of SBE's size</th>
<th>SBE_MIN</th>
<th>SBE_MINSEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBE_3</td>
<td>64</td>
<td>98.61%</td>
</tr>
</tbody>
</table>

(It turns out that for any SBE B with three variables, SBE_MINSEN produces a test set that distinguishes B from any non-equivalent SBE with the same variables.)

Each of SBE_4.1 through SBE_4.5 denotes a set of 51 randomly generated, mutually non-equivalent SBE's with four variables. (The SBE's in these five sets are also mutually non-equivalent.) SBE_4.6 denotes a set of 37 manually generated, mutually non-equivalent SBE's with 4 variables. For each of SBE_4.1 through SBE_4.6, we applied the same approach, as described above for SBE_3, to compute the average of dead mutant percentages by using SBE_MIN (SBE_MINSEN). These averages are shown below:

<table>
<thead>
<tr>
<th>set of SBE's size</th>
<th>SBE_MIN</th>
<th>SBE_MINSEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBE_4.1</td>
<td>51</td>
<td>99.76%</td>
</tr>
<tr>
<td>SBE_4.2</td>
<td>51</td>
<td>99.73%</td>
</tr>
<tr>
<td>SBE_4.3</td>
<td>51</td>
<td>99.69%</td>
</tr>
<tr>
<td>SBE_4.4</td>
<td>51</td>
<td>99.76%</td>
</tr>
<tr>
<td>SBE_4.5</td>
<td>51</td>
<td>99.73%</td>
</tr>
<tr>
<td>SBE_4.6</td>
<td>37</td>
<td>96.77%</td>
</tr>
</tbody>
</table>

Although SBE_MIN and SBE_MINSEN were developed for the detection of Boolean operator errors, our experimental results indicate that the test sets generated by SBE_MIN and SBE_MINSEN are effective for detecting all types of errors in a Boolean expression. Another interesting observation is that even though the test sets produced by SBE_MIN guarantee the detection of multiple Boolean operator errors and those produced by SBE_MINSEN do not, the former are less effective than the latter for detecting all types of errors in a Boolean expression.

4. Test-Constraints for a Condition

As mentioned in section 2, existing condition testing strategies do not deal with the detection of Boolean expression errors in a condition. Since we have developed effective testing strategies for SBE's, one interesting and important question is how to extend these strategies to derive effective testing strategies for complicated conditions. For a condition that is not an SBE, it corresponds to an SBE formed by replacing each operand in the condition with a distinct Boolean variable. For example, the condition \((E_1 \lor E_2) \land (E_3 \lor E_4)\) corresponds to \((I_1 \lor I_2)\), where I1 and I2 are distinct Boolean variables. Note that a test for \((I_1 \lor I_2)\), say \((t, f)\), is not a test for \((E_1 \lor E_2) \land (E_3 \lor E_4)\). In this paper, \((t, f)\) is viewed as a "test-constraint" for \((E_1 \lor E_2) \land (E_3 \lor E_4)\) and the coverage of \((t, f)\) for this condition requires a test making the value of \((E_1 \lor E_2)\) to be true and that of \((E_3 \lor E_4)\) to be false. Below we formally define the concept of test-constraints.

A test-constraint (or constraint, if there is no ambiguity) for a condition with n, n>0, operands is defined as \((D_1, D_2, \ldots, D_n)\), where Di, 0<i<n, is a symbol specifying a constraint on the values in the i-th operand of the condition. A test-constraint D for a condition C is said to be covered or satisfied by a test for C if during the execution of C using this test, the values in the i-th operand of C satisfy the i-th constraint in D.

For a Boolean variable, say B, we use the following symbols to denote different types of constraints on the value of B:

- t the value of B is true,
- f the value of B is false,
- * there is no constraint on the value of B.

For a relational expression R, say \((E_1 < \text{op} > E_2)\), we use the following symbols to denote different types of constraints on the values in R:

- t the value of R is true,
- f the value of R is false,
- > the value of E1 is greater than that of E2,
- = the value of E1 is equal to that of E2,
- < the value of E1 is less than that of E2,
- \(+\text{BOUND}\) the value of \((E_1 - E_2)\) is greater than zero and less than or equal to \(+\text{BOUND}\),
- \(-\text{BOUND}\) the value of \((E_1 - E_2)\) is less than zero and greater than or equal to \(-\text{BOUND}\),
- * there is no constraint on the values in R.
To illustrate the notion of test-constraints, consider the condition \((E1 \geq E2) \mid (E1 > E2)\). The constraint \((t, t)\) for this condition is covered by a test making the value of \((E1 \geq E2)\) to be true and the value of \((E1 > E2)\) to be less than that of \(E4\).

In [Tha89] we showed that many condition testing strategies can be formally defined in terms of test-constraints. (The term "condition-constraints", in lieu of "test-constraints", was used in [Tha89].) In the next two sections, we show how to generate constraints for a condition according to new condition testing strategies. It is possible that a constraint for a condition \(C\) can never be covered by any test for \(C\); such a constraint is said to be an infeasible constraint (or test-constraint) for \(C\). For example, consider the condition \(((E1 \geq E2) \mid (E1 = E2))\). The constraint \((t, t)\) is infeasible for this condition since the value of \(E1\) can never be both greater than and equal to that of \(E2\) at the same time. The existence of infeasible constraints for a condition is often an indication of either errors in the condition or poor structure of the condition. For example, the condition \(((E1 \geq E2) \mid (E1 = E2))\) should be transformed into \((E1 > E2)\). The problem of determining whether a constraint is feasible for a condition is undecidable, since this problem is equivalent to the problem of determining whether a path in a program is feasible (i.e., for a path in a program, whether there exists at least one input that can exercise this path). It is also possible that in a program \(P\), a feasible constraint for a condition \(C\) can never be covered by any input for \(P\); such a constraint is said to be an infeasible constraint (or test-constraint) for \(C\) in \(P\). The problem of determining whether a constraint is feasible for a condition in a program is also undecidable, since it is equivalent to the problem of determining whether a branch of a condition in a program is feasible (i.e., for a condition branch in a program, whether there exists at least one input that exercises this branch). Such undecidable problems exist in condition testing and other types of testing; they are inherent difficulties in softwaretesting.

5. BRO Testing: A Condition Testing Strategy

Based on the Detection of Boolean and Relational Operator Errors

As mentioned earlier, for an SBE, algorithm SBE-MIN generates a minimum test set that guarantees the detection of Boolean operator errors. Also, the relational operator testing strategy guarantees the detection of an incorrect relational operator in a relational expression. By combining algorithm SBE-MIN and relational operator testing, we propose a new condition testing strategy based on the detection of Boolean and relational operator errors; this testing strategy is referred to as BRO (Boolean and Relational Operator) testing.

The BRO testing strategy involves the following steps for testing a program \(P\):

1. For each condition in \(P\), apply algorithm BRO_CONSTRAINTS (see below) to generate a set of test-constraints, and delete as many infeasible test-constraints as possible.

2. Select a test set for \(P\) to cover as many test-constraints as possible.

Below we first present algorithm BRO_CONSTRAINTS and then discuss the theoretical properties of BRO testing.

Algorithm BRO_CONSTRAINTS:

Input: a condition \(C\) with \(n\) operands
Output: BRO_CONSTRAINTS(C), which is a set of test-constraints for \(C\).

(1) Transform \(C\) into its corresponding SBE, say \(B\), by replacing each operand in the condition with a distinct Boolean variable.

(2) Apply algorithm SBE-MIN to \(B\) to produce SBE-MIN(B).

(3) For each constraint in \(SBE-MIN(B)\), say \((X1, X2, ..., Xn)\), where each \(Xi\), \(0 \leq i \leq n\), is "true" or "false", transform it into constraints \((Y1, Y2, ..., Yn)\) and \((Z1, Z2, ..., Zn)\) and add them to BRO_CONSTRAINTS(C), where \(Y1\) and \(Z1\), \(0 \leq i \leq n\), are determined by \(Xi\) and \(Ci\), the \(i\)th operand in \(C\), according to the following rules: (1) denote a Boolean variable and \(E1\) and \(E2\) denote arithmetic expressions.)

<table>
<thead>
<tr>
<th>Xi</th>
<th>Ci</th>
<th>Yi</th>
<th>Zl</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) t</td>
<td>1</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>(b) t</td>
<td>(E1 &lt; E2)</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>(c) t</td>
<td>(E1 = E2)</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>(d) t</td>
<td>(E1 &gt; E2)</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>(e) t</td>
<td>(E1 = E2)</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>(f) t</td>
<td>(E1 = E2)</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>(g) t</td>
<td>(E1 &gt; E2)</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>(h) f</td>
<td>1</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>(i) f</td>
<td>(E1 &lt; E2)</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>(j) f</td>
<td>(E1 = E2)</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>(k) f</td>
<td>(E1 &gt; E2)</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>(l) f</td>
<td>(E1 &lt; E2)</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>(m) f</td>
<td>(E1 = E2)</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>(n) f</td>
<td>(E1 &gt; E2)</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
</tbody>
</table>

The above rules are derived according to the relational operator testing strategy, which requires the coverage of the constraints \(>, =\), and \(<\) for a relational expression. For each combination of \(Xi\) and \(Ci\), either one or two of \(>, =\), and \(<\) are required. In rules (a) through (d), (h), and (l) through (n), \(Y1\) and \(Z1\) are the same and in other rules, \(Y1\) and \(Z1\) are different. If \((X1, X2, ..., Xn)\) and \((Z1, Z2, ..., Zn)\) are the same, then only one constraint is produced and added to BRO_CONSTRAINTS(C). (In each of the above rules, exchanging \(Y1\) and \(Z1\) has effect on the resulting \((Y1, Y2, ..., Yn)\) and \((Z1, Z2, ..., Zn)\), but has no effect on the theoretical properties of BRO testing to be presented later.)
To illustrate the above algorithm, let C1 denote \((E1 \land E2) \land (E3 \land E4)\). The SBE corresponding to C1 is \((E1 \land E2) \land \text{SBE}_\text{MIN}(E1 \land E2)\) is \(\langle t, t, (t, f)\rangle\). (See the example above.)  

\((t, t)\) is transformed into \((>,=)\) according to rules (d) and (c), 
\((t, f)\) is transformed into \((>,<)\) and \((>,>)\) according to rules (d) and (3), 
\((f, t)\) is transformed into \((<,=)\) and \((<,>)\) according to rules (k) and (c).

So \(\text{BO_CONSTRAINTS}(C1)\) is \((>,=)\), \((>,<)\), \((>,>)\), 
\((<,=)\), \((<,>)\). Let \(C2\) denote \((E1=\land E2) \land (E3=\land E4)\). The SBE corresponding to \(C2\) is \((E1 \land E2) \land \text{SBE}_\text{MIN}(E1 \land E2)\) is \(\langle t, f, (f, t), (f, f)\rangle\). In step (3) of the above algorithm \((t, t)\) is transformed into \((>,=)\) and \((<,=)\) according to rules (d) and (i), 
\((t, f)\) is transformed into \((<,>)\) and \((>,>)\) according to rules (j) and (d), 
\((f, t)\) is transformed into \((<,=)\) and \((<,>)\) according to rules (j) and (k).

So \(\text{BO_CONSTRAINTS}(C2)\) is \((>,=)\), \((<,=)\), \((<,>)\), \((>,=)\), \((<,>\), \((>,>)\). 

Now we show some theoretical properties of the algorithm. The motivation of the algorithm is the detection of Boolean and relational operator errors in a condition. Thus, an important question concerning the algorithm is its effectiveness for detecting Boolean and relational operator errors in a condition.

Definition: A test set \(T\) for a condition \(C\) is said to be a \(\text{BO test set for } C\) if \(T\) guarantees the detection of Boolean and relational operator errors in \(C\). (An error in \(C\) is detected by a test if an execution of \(C\) with this test produces an incorrect outcome.)

Definition: A set \(S\) of constraints for a condition \(C\) is said to be a \(\text{BO constraint set for } C\) provided that if a test set \(T\) for \(C\) covers \(S\), then \(T\) is a \(\text{BO test set for } C\).

Theorem 1: Let \(C\) be a condition with \(n, n=0,1,\ldots, n\) operands. \(\text{BO_CONSTRAINTS}(C)\), the constraint set for \(C\) produced by algorithm \(\text{BO_CONSTRAINTS}\), is a \(\text{BO constraint set for } C\) and has at most \(2^n+1\) constraints.

The above theorem indicates that for a condition \(C\), \(\text{BO_CONSTRAINTS}(C)\) provides a guide for the selection of a \(\text{BO test set for } C\). Now we consider the effectiveness of the algorithm for testing a program. To apply the algorithm to a program \(P\), we apply \(\text{BO_CONSTRAINTS}\) to generate a constraint set for each condition in \(P\) and select a test set for \(P\) to cover as many constraints as possible. Assume that a test set \(T\) for \(P\) provides complete coverage of \(\text{BO_CONSTRAINTS}(C)\), where \(C\) is a condition in \(P\). However, \(T\) does not guarantee the detection of Boolean and relational operator errors in \(C\). The is due to the following problem in condition testing. (Similar problems exist in other types of software testing.) Assume that when an execution of \(P\) with input \(X\) reaches condition \(C\), the input for \(C\) detects an error in \(C\) (i.e., this execution takes an incorrect branch of \(C\)). However, this execution of \(P\) does not necessarily expose the error in \(C\), because this execution may still produce the correct output. (An error in \(P\) is detected by input \(X\) only if the output of \(P\) on input \(X\) is incorrect.) A sufficient condition for making this execution to expose the error in \(C\) is that coincidental correctness (see below) does not exist in \(P\).

Definition: The situation that an execution of a program \(P\) produces an incorrect path (or an incorrect branch of a condition) but still produces the correct output is referred to as coincidental correctness.

Definition: Let \(T\) be a test set for a program \(P\) and let \(C\) be a condition in \(P\). \(T(P,C)\) denote the set of inputs for \(C\) during executions of \(P\) on \(T\).

Definition: A test set \(T\) for a program \(P\) is said to be a \(\text{BO test set for } P\) if, for each condition in \(P\), \(T\) guarantees the detection of Boolean and relational operator errors.

Theorem 2: Assume that coincidental correctness does not exist in a program \(P\). Let \(T\) be a test set for \(P\) such that for each condition \(C\) in \(P\), \(T(P,C)\) is a \(\text{BO test set for } C\). Then \(T\) is a \(\text{BO test set for } P\).

From theorems 1 and 2, we have the following theorem on the effectiveness of the algorithm:

Theorem 3: Assume that coincidental correctness does not exist in a program \(P\). Let \(T\) be a test set for \(P\) such that for each condition \(C\) in \(P\), \(T(P,C)\) covers \(\text{BO_CONSTRAINTS}(C)\). Then \(T\) is a \(\text{BO test set for } P\).

6. \(\text{BB Test: A Condition Testing Strategy Based on the Detection of Boolean and Relational Expression Errors}\)

In this section we propose a new condition testing strategy that is an improvement over \(\text{BB test}\) by focusing on the detection of all types of errors in Boolean and relational expressions, not just Boolean and relational operator errors. As mentioned in section 2, the relational expression testing strategy not only guarantees the detection of relational operator errors, but also focus on the detection of errors in a relational expression. By combining algorithm \(\text{SBE_MIN}\) and relational expression testing, we propose a new
condition testing strategy based on the detection of Boolean and relational expression errors; this testing strategy is referred to as BRE (Boolean and Relational Expression) testing.

The BRE testing strategy is the same as the BBO testing strategy except that algorithm BRE_CONSTRAINTS is replaced with algorithm BRE_CONSTRAINTS (see below).

Algorithm BRE_CONSTRAINTS:
Input: a condition C with \( n \in \mathbb{N} \) operands.
Output: BRE CONSTRAINS(C), which is a set of test-conditions for \( C \).

This algorithm is the same as algorithm BBO_CONSTRAINTS except that "BRE_MIND" is replaced with "BRE_MMUSD" and that in rules (b) through (g) and (l) through (n), the symbols "\( \geq \)" and "\( \leq \)" are replaced with "\( \text{BOUND} \)" and "\( \text{-BOUND} \)" respectively, where \( \text{BOUND} \) is a positive number.

In algorithm BRE_CONSTRAINTS, the selection of the value of \( \text{BOUND} \) is important. A smaller value of \( \text{BOUND} \) makes the constraints more effective for error detection, but might make the selection of tests more difficult. For a relational expression (\( \text{E1} \ < \text{op} \ > \text{E2} \)), if the constraint "\( \text{BOUND} \)" is covered by a test, then the constraint \( (\geq) \) is definitely covered by this test, but not vice versa. Similarly, if the constraint "\( \text{-BOUND} \)" is covered by a test, then the constraint \( (<) \) is definitely covered by this test, but not vice versa. Thus, constraints "\( \text{BOUND} \)" and "\( \text{-BOUND} \)" are more difficult to be covered than \( (\geq) \) and \( (<) \), respectively, but the former are more effective than the latter for detecting relational expression errors.

For an SBE, algorithm SBE_MIND, unlike algorithm SBE_MIN, does not necessarily produce a test set that guarantees the detection of Boolean operator errors. Thus, for a condition \( C \), BRE_CONSTRAINTS(C), unlike BBO_CONSTRAINTS(C), is not necessarily a BBO constraint set for \( C \).

Theorem 4: For a condition \( C \), BRE_CONSTRAINTS(C) is more effectively than BBO_CONSTRAINTS(C) for detecting Boolean and relational expression errors in \( C \).

Algorithm BRE_CONSTRAINTS can be modified to produce different sets of test-conditions. One possible variation of BRE_CONSTRAINTS is to use different values of \( \text{BOUND} \) for different conditions in a program. Another possible variation of BRE_CONSTRAINTS is to allow a mixture of "\( \text{BOUND} \)", "\( \text{-BOUND} \)", "\( \geq \)", "\( \leq \)" and "\( = \)" in test-conditions.

7. Conclusion

In this paper, we have defined two condition testing strategies, BRE and BBO testing. These two testing strategies are different from existing condition testing strategies in that they are based on the detection of both Boolean and relational expression errors in a condition. For a condition with \( n \) operands, the number of tests required by BBO or BRE testing is at most \( 2^n \). Based on our empirical studies of algorithms SBE_MIN and SBE_MIND and the theoretical properties of BBO and BRE testing, we believe that BBO and BRE testing are practical and effective for testing programs containing complicated conditions. In [Tai89] the use of BBO testing instead of branch testing for complicated conditions was suggested and several conjectures on the effectiveness of condition testing in general and BBO testing in particular were given. Currently, we are carrying out experiments to evaluate the effectiveness of BBO and BRE testing strategies.

References