Predicate-Based Test Generation for Computer Programs

Kuo-Chung Tai
Computer Science Department
North Carolina State University
Raleigh, NC 27659-8206, USA
kct@csc.ncsu.edu

Abstract

One common approach to software testing, referred to as **predicate testing**, is to require certain types of tests for each predicate (or condition) in a program. Effective testing of predicates in a program results in effective detection of faults in the program. However, existing predicate testing strategies are ineffective or impractical for testing compound predicates, which are predicates with one or more AND/OR operators.

In this paper, two fault-based testing strategies for compound predicates are defined, **BOR** (*B*oolean *O*perato*R*) **testing** and **BRO** (*B*oolean and *R*elational *O*perator) **testing**. It is shown that for a predicate with n, n>0, AND/OR operators, at most n+2 (2*n+3) tests are needed to satisfy BOR (BRO) testing. Preliminary experimental results indicate that BOR and BRO testing are effective for the detection of various types of faults in a predicate and provide more specific guidance than branch testing for test generation.

1. Introduction

Predicates (or conditions) in a program divide the input domain of this program into partitions and define the paths of this program. One common approach to software testing, referred to as **predicate testing**, is to require certain types of tests for each predicate in a program. Let C be the predicate of an if or while statement in a program P.

```
        ...
         |
         C
 true /     \ false
   ...       ...
```

Assume that an execution of P with input X reaches C. If the outcome of C is incorrect due to fault(s) in C or statement(s) executed before C is reached, then this execution exercises an incorrect path of P and is very likelyto produce an incorrect result. (The possibility of "coincidental correctness", i.e., an execution of P with an incorrect path but a correct result, is very small.) Thorough testing of C can detect not only faults in C or statements executed before C is reached, but also faults in statements executed after C is reached. Therefore, effective testing of predicates in a program results in effective detection of faults in the program.

A number of predicate testing strategies have been proposed. However, these strategies are either ineffective or impractical for testing compound predicates, which are predicates with one or more binary boolean operators. In this paper, we describe a practical approach to generating effective tests for compound predicates. This paper is organized as follows. The remainder of this section provides several basic definitions. Section 2 surveys existing predicate testing strategies and explains why they are ineffective or impractical for testing compound predicates. Section 3 describes a **BR-constraint** for a compound predicate, which consists of a constraint for each *B*oolean variable or *R*elational expression in the compound predicate. Section 4 defines two fault-based testing strategies for compound predicates: **BOR** (*B*oolean *O*perato*R*) **testing** and **BRO** (*B*oolean and *R*elational *O*perator) **testing**. Section 5 shows how to generate a minimum set of BR-constraints for a compound predicate in order to guide the generation of tests for BOR or BRO testing. Section 6 discusses how to apply BOR or BRO testing to a program. Section 7 presents experimental results to demonstrate the effectiveness of BOR and BRO testing. Section 8 concludes this paper. The proofs of theorems in this paper are omitted and can be found in [Tai92a]

A predicate in a program is either a simple or compound predicate. A **simple predicate** is a boolean variable or a relational expression, possibly with one or more NOT ("~") operators. A **relational expression** is of the form

E1 <rop> E2

where E1 and E2 are arithmetic expressions and <rop> is one of six possible relational operators: "<", "<=", "=", "/=" (non-equality), ">", and ">=". (Non-arithmetic expressions, such as character strings and sets, are not discussed in this paper.) A **compound predicate** consists of at least one binary boolean operator, two or more operands, and possibly NOT operators and parentheses.

This work was supported in part by the National Science Foundation under grant CCR-8907807.

0270-5257/93 $03.00 © 1993 IEEE

The binary boolean operators allowed in a predicate include OR ("|") and AND ("&"), each having two operands. A **simple operand** in a compound predicate refers to an operand without binary boolean operators. A **compound operand** in a predicate refers to an operand with at least one binary boolean operator. A **boolean expression** is a predicate with no relational expressions. In this paper, Bi, i>0, denotes a boolean variable, Ei, an arithmetical expression, <rop> or <rop$_i$>, a relational operator, and <bop> or <bop$_i$>, a binary boolean operator.

It is assumed that a predicate has no syntactic faults. If a predicate is incorrect with respect to what it is intended to do, then one or more of the following types of faults occur:
(1)boolean operator fault (incorrect AND/OR operator or a missing/extra NOT operator)
(2)boolean variable fault (incorrect boolean variable)
(3)boolean parenthesis fault (incorrect location of a parenthesis)
(4)relational operator fault (incorrect relational operator) and
(5)arithmetic expression fault. (The various types of faults in an arithmetic expression are not discussed in this paper.)
Faults of types (1), (2) and (3) and their combinations are referred to as **boolean expression faults**. Faults of types (4) and (5) and their combinations are referred to as **relational expression faults**.

An incorrect predicate contains either a **single fault** or **multiple faults** of the same or different types. The existence of one or more faults in a predicate C is said to be **detected by a test** if an execution of C with this test produces (executes) an incorrect outcome (branch) of C. A test set for C is said to **guarantee the detection** of a certain type of fault in C if the existence of such fault(s) in C can be detected by at least one element of this test set, provided that C does not contain faults of other types. Assume that predicate C* has the same set of variables as C and is not equivalent to C. A test (a test set) is said to **distinguish** C from C* if C and C* produce different outcomes on the test (on at least one element of the test set). Assume that C contains faults and C" is the correct version of C. A test set is said to be **insensitive** to the faults in C if this test set cannot distinguish C from C".

2. Existing Predicate Testing Strategies

In this section, we first describe a number of existing testing strategies for simple predicates and then explains why intuitive extensions of such strategies are ineffective or impractical for testing compound predicates.

2.1 Testing Strategies for Simple Predicates

Branch Testing This strategy requires that the true and false branches of a boolean variable or relational expression be executed at least once.

Relational Operator Testing For a relational expression, say (E1 <rop> E2), this strategy requires three tests satisfying the following requirements [How82]:
(1) one test makes $E1 > E2$,
(2) one test makes $E1 < E2$, and
(3) one test makes $E1 = E2$.
If <rop> is incorrect and E1 and E2 are correct, then this strategy guarantees the detection of the incorrect relational operator.

Relational Expression Testing This strategy is the same as relational operator testing except that requirements (1) and (2) are modified as follows [Fos80,How82]:
(1) one test makes $0 < (E1 - E2) <= \varepsilon$,
(2) one test makes $-\varepsilon <= (E1 - E2) < 0$,
where ε is a positive number and used as a parameter of relational expression testing. A smaller value of ε will make the two tests for (1) and (2) more effective for detecting faults in E1 and E2. Relational expression testing is more stringent than relation operator testing and thus is more effective for detecting relational expression faults.

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 [Whi80] requires that each domain boundary determined by (E1 <rop> E2) be tested with n+1 points. This testing strategy is more stringent than relational expression testing.

2.2 Testing Strategies for Compound Predicates

(External) Branch Testing For a compound predicate C, this strategy requires that the true and false branches of C be executed at least once. Branch testing is ineffective for testing a compound predicate since the number of tests required for branch testing is always two, regardless of the complexity of this predicate.

Complete Branch Testing For a compound predicate C, this strategy requires that the true and false branches of every simple or compound operand in C as well as C itself be executed at least once. Although complete branch testing is more stringent than branch testing, it can be satisfied by using two tests for C, which cover two different branches of each operand in C.

Exhaustive Branch Testing For a compound predicate C, this strategy requires that all combinations of true and false branches of simple operands in C be executed at least one. If C consists of n AND/OR operators, then C contains n+1 simple operands and exhaustive branch testing for C requires 2**(n+1) tests.

The above testing strategies for a compound predicate are based on the structure, not the possible faults, of this compound predicate. As mentioned earlier, faults in a compound predicate consist of boolean and relational expression faults. External and complete branch testing of a compound predicate are ineffective for fault detection, while exhaustive branch testing is practical only if the number of AND/OR operators is small. Below we consider an intuitive extension of relational (expression) operator testing for a compound predicate.

Complete Relational Operator (Expression) Testing
For a compound predicate C, this strategy requires relational operator (expression) testing for each relational expression in C. However, this strategy does not guarantee the detection of incorrect relational operators in C. (Note that the existence of faults in C is detected only if an incorrect outcome of C is produced.) Consider the following compound predicate denoted as C#

(E1 = E2) & (E3 /= E4).

Let T1 be a set of three tests t11, t12, and t13 for C# such that

t11 makes E1 = E2 and E3 = E4,
t12 makes E1 > E2 and E3 > E4, and
t13 makes E1 < E2 and E3 < E4.

T1 satisfies relational operator testing for each simple operand of C#, but it cannot distinguish C# from the following two predicates:

(E1 = E2) & (E3 < E4) and
(E1 /= E2) & (E3 = E4),

which differ from C# in relational operators only. (The above two predicates produce the false value, like C#, on each of t11, t12 and t13.) Although T1 guarantees the detection of incorrect boolean operator in each of (E1=E2) and (E3/=E4) according to the outcomes of (E1=E2) and (E3/=E4), respectively, it fails to do so according to the outcome of (E1=E2)&(E3/=E4).

The above example illustrates the major problem for the detection of faults in a compound predicate: **how to propagate an incorrect outcome of an operand in a compound predicate to the outcome of the predicate.** In order to **reveal** faults in an operand of a compound predicate, the values of other operands in the predicate must satisfy certain constraints. In the next section, we introduce a notion for specifying such constraints.

3. BR-constraints for Predicates

For a predicate with n, n>0, simple operands

$(\langle opd_1 \rangle \langle bop_1 \rangle \langle opd_2 \rangle \ldots \ldots \langle bop_{n-1} \rangle \langle opd_n \rangle)$,

where $\langle opd_i \rangle$, i>0, denotes the *i*th simple operand, a **BR-constraint** (or **constraint**, if there is no ambiguity) is defined as (D1,D2,...,Dn), where Di, 0<i<=n, is a symbol specifying a constraint on the *b*oolean variable or *r*elational expression in $\langle opd_i \rangle$. (Note that a simple operand is a boolean variable or relational expression, possibly with one or more NOT operators.)

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, say (E1 <rop> E2), we use the following symbols to denote different types of constraints on the values in the relational expression:

- t the value of the relational expression is true,
- f the value of the relational expression is false,
- \> the value of (E1 − E2) is greater than zero,
- = the value of (E1 − E2) is equal to zero,
- < the value of (E1 − E2) is less than zero,
- +ε the value of (E1 − E2) is greater than zero and less than or equal to ε,
- −ε the value of (E1 − E2) is less than zero and greater than or equal to −ε,
- * there is no constraint on the value of the relational expression.

A constraint D for a predicate C is said to be **covered (or satisfied) by a test** if during the execution of C with this test, the value of each boolean variable or relational expression in C satisfies the corresponding constraint in D. Consider the constraint (=,<) for predicate ((E1>= E2)|¬(E3>E4)). The coverage of (=,<) for this predicate requires a test making E1=E2 and E3<E4. (Note that the occurrence of "¬" in ¬(E3>E4) does no affect the coverage requirement for "<".) The coverage of (t,+ε) for this predicate requires a test making E1>=E2 and 0 < E3−E4 <= ε.

A set S of BR-constraints for a predicate C is said to be **covered (or satisfied) by a test set** T if each constraint in S is covered for C by at least one test in T. Note that a test in T may cover two or more constraints in S. Many of the predicate testing strategies discussed in section 2 can be defined in terms of BR-constraints. Below are several examples:

branch testing for (E1 <rop> E2): {(t), (f)}
relational operator testing for (E1 <rop> E2):
{(>), (=), (<)}
complete branch testing for
((E1 <rop1> E2) <bop> (E3 <rop2> E4)):
{(t,*), (f,*), (*,t), (*,f)}
complete relational operator testing for
((E1 <rop1> E2) <bop> (E3 <rop2> E4)):
{(>,*), (=,*), (<,*), (*,>), (*,=), (*,<)}.

4. Two Testing Strategies for Compound Predicates : BOR and BRO Testing

We have investigated the problem of test generation for boolean expressions [Tai87]. The **boolean operator testing** (or **BOR testing**) strategy for a boolean expression is to require a set of tests to guarantee the detection of boolean

operator faults, including incorrect AND/OR operators and missing or extra NOT operators (with the assumption that the boolean expression does not contain faults of other types). We have shown that if a test set for a boolean expression is effective for detecting boolean operator faults, then it is also effective for detecting other types of boolean expression faults (i.e., boolean variable and parenthesis faults) as well as the combinations of these types of faults. The **BOR testing** strategy for a predicate is similar to that for a boolean expression, i.e., it requires a test set to guarantee the detection of boolean operator faults in the predicate (with the assumption that the predicate does not contain faults of other types).

Since compound predicates are combinations of boolean and relational expressions, we consider combining BOR testing with testing strategies for relational expressions. Below we define a combination of BOR testing with relational operator testing.

Definition: The ***b*oolean and *r*elational *o*perator testing** (or **BRO testing**) strategy for a predicate is to require a set of tests to guarantee the detection of boolean and relational operator faults (or **BRO faults**), including incorrect AND/OR operators, missing or extra NOT operators, and incorrect relational operators (with the assumption that the predicate does not contain faults of other types).

Definition: A test set T for a predicate C is said to be a **BOR (BRO) test set** for C if T satisfies the BOR (BRO) testing strategy for C.

Definition: A set S of BR-constraints for a predicate C is said to be a **BOR (BRO) constraint set** for C provided that if a test set T for C covers S, then T is a BOR (BRO) test set for C.

BOR and BRO testing are different from those mentioned in section 2 in that the former do not provide explicit guidance on the selection of tests. In the next section, we will show how to generate a BOR or BRO constraint set for a compound predicate.

5. Generation of BR-Constraint Sets for BOR and BRO Testing

First we give several definitions and theorems. In sections 5.1 and 5.2, we present algorithms for generating BOR and BRO constraint sets, respectively, for a compound predicate.

Let u=(u_1, ..., u_m) and v=(v_1,...,v_n), where m,n >0, be two lists of elements. The concatenation of u and v, denoted as (u,v), is (u_1,...,u_m,v_1,...,v_n). Let A and B be two sets of lists. A$B denotes the union of A and B, A*B the product of A with B, and ¦A¦ the size of A. A%B, called the **onto** from A to B, is a minimal set of (u,v) such that u (v) is in A (B) and every element in A (B) appears as u (v) at least once. Thus, ¦A%B¦ is the maximum of ¦A¦ and ¦B¦. If both A and B have two or more elements, A%B has several possible values and returns any one of them. For example, assume that C={(a),(b)} and C={(c),(d)}. C%D has two possible values: {(a,c),(b,d)} and {(a,d),(b,c)}. Assume that E={(a),(b)} and F={(c),(d),(e)}. E%F has six possible values: {(a,c),(b,d),(a,e)}, {(a,c),(b,d),(b,e)}, {(a,c),(a,d),(b,e)}, {(b,c),(a,d),(b,e)}, {(b,c),(a,d),(a,e)}, and {(b,c),(b,d),(a,e)}.

Let X denote a constraint, which consists of "t", "f", ">", "=", and "<", for a predicate C. The value produced by C on any input covering X is the same, which is denoted as C(X). (Thus, a constraint for C can be viewed as an input of C.) X is said to **cover** or **be associated with** the true (false) branch of C if C(X) = true (false). Assume that S is a constraint set for C. S can be divided into two sets S_t and S_f, where S_t(C) = {X in S ¦ C(X) = t} and S_f(C) = {X in S ¦ C(X) = f}.

Let w be an input covering X for C. The value produced by C on w is C(X). Let C" be a predicate that has the same set of input variables as C. The value produced by C" on w can be denoted as C"(X) under one of the following two conditions:

(1) C" differs from C only in boolean operators and each constraint in X is either "t" or "f", and
(2) C" differs from C only in boolean and/relational operators and each constraint in X is one of "t" and "f", for a boolean variable in C, or one of ">", "=" and "<", for each relational expression in C.

Assume that (1) or (2) is true. (In fact, (1) and (2) are true in the theories of BOR and BRO testing, respectively.) A test satisfying X for C (C") can distinguish C (C") from C" (C) if and only if C(X) /= C"(X). Thus, X is said to **distinguish** C from C" if C(X)/=C"(X). For a constraint set S for C, C"(S)=C(S) if for every constraint X in S, C"(X)=C(X); otherwise, C"(S)/=C(S) and S is said to **distinguish** C from C".

For a boolean variable, its BOR or BRO constraint set is defined as {(t), (f)}. For a relational expression (E1 <rop> E2), its BOR constraint set is defined as {(t), (f)} and its BRO constraint set {(>), (=), (<)}. Let C1 and C2 be predicates. Assume that S1 and S2 are BOR (BRO) constraint sets for C1 and C2 respectively. Below we show that we can easily construct BOR (BRO) constraint sets for (C1¦C2) and (C1&C2) by using S1 and S2.

Theorem 1: Let C be (C1¦C2), where C1 and C2 are predicates. Assume that S1 and S2 are BOR (BRO) constraint sets for C1 and C2 respectively. Define
F(C) = S1_f % S2_f and
T(C) = (S1_t * {f2}) $ ({f1} * S2_t),
where f1 is in S1_f, f2 in S2_f, and (f1,f2) in F(C). T(C)$F(C) is a BOR (BRO) constraint set for C.

To help understand the above definitions of F(C) and T(C), we show that theorem 1 is true with the following assumptions: (1) S1 and S2 are BRO constraint sets for C1 and C2 respectively, (2) either C1 or C2, but not both, may have BRO faults, and (3) "¦" is correct. Assume that C1 contains BRO faults, at least one element, say x1, of S1_t or S1_f produces an unexpected value for C1. By the definitions of T(C) and F(C), T(C)$F(C) contains (x1,x2), where x2 is either f2 or an element of S2_f. Since C2(x2) = false, we have C((x1,x2)) = C1(x1). Thus (x1,x2) produces an unexpected value for C, indicating the existence of faults in C. Similarly, T(C)$F(C) reveals possible BRO faults in C2.

Theorem 2: Let C be (C1&C2), where C1 and C2 are predicates. Assume that S1 and S2 are BOR (BRO) constraint sets for C1 and C2 respectively. Define

T(C) = S1_t % S2_t and

F(C) = (S1_f * {t2}) $ ({t1} * S2_f),

where t1 is in S1_t, t2 in S2_t, and (t1,t2) in T(C). T(C)$F(C) is a BOR (BRO) constraint set for C.

To illustrate theorems 1 and 2, consider the construction of a BOR constraint set for the compound predicate C# defined in section 2

(E1 = E2) & (E3 /= E4).

Let C1 and C2 be (E1 = E2) and (E3 /= E4) respectively. For BOR testing of C1 and C2, S1_t = S2_t= {(t)} and S1_f = S2_f = {(f)}. Following theorem 2, T(C#) = S1_t % S2_t = {(t,t)}. Since t1 = t2 = t, F(C#) = {((f,t) (t,f)}. Thus, {(t,t), (f,t), (t,f)} is a BOR constraint set for C#. Note that (I1&I2), where I1 and I2 denote distinct boolean variables, has the same BOR constraint set as C#.

Now consider the construction of a BRO constraint set for C#. For BRO testing of C1, S1_t={(=)} and S1_f={(>),(<)}. For BRO testing of C2, S2_t={(>),(<)} and S2_f={(=)}. Following theorem 2, T(C#) =

S1_t % S2_t = {(=)} % {(>),(<)} = {(=,>),(=,<)}.

Since T(C#) has two elements, we choose (=,>) as (t1,t2). Therefore, F(C#) = (S1_f * {(>)}) $ ({(=)} * S2_f) =

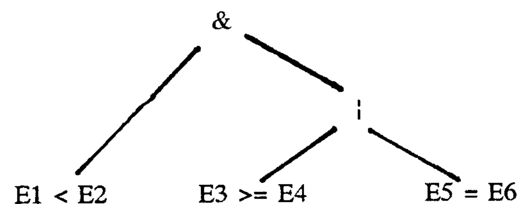
{(>,>), (<,>), (=,=)}.

Thus, {(=,>), (=,<), (>,>), (<,>), (=,=)} is a BRO constraint set for C#.

For the following predicate denoted as C@:

((E1 < E2) & ((E3 >= E4) ¦ (E5 = E6))),

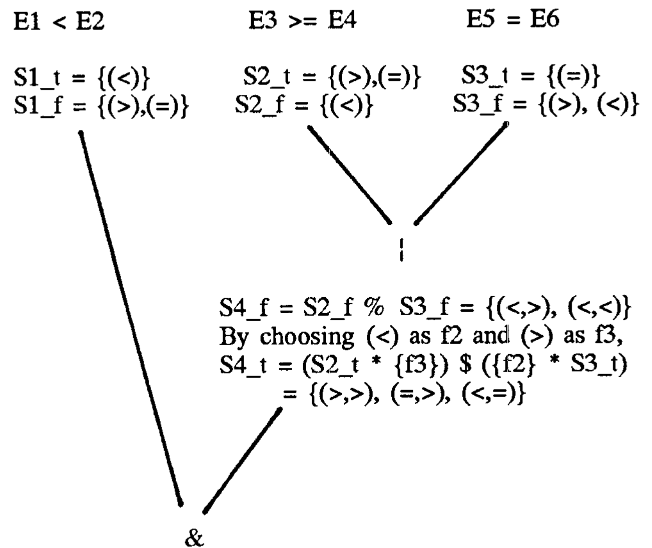
we show how to construct a BRO constraint set. The construction is done by making a bottom-to-top visit of the syntax tree of C@, which is shown below:



A BRO constraint set for C@ can be constructed as follows:

(1) generate BRO constraint sets S1, S2, and S3 for (E1<E2), (E3>=E4), and (E5=E6), respectively.
(2) apply theorem 1 to S2 and S3 and construct a constraint set S4 for ((E3>=E4) ¦ (E5=E6)).
(3) apply theorem 2 to S1 and S4 and construct a constraint set S5 for C@.

Below is the reverse of the syntax tree of C@ with associated constraint set for each node of the syntax tree.



S5_t = S1_t % S4_t = {(<,>,>), (<,=,>), (<,<,=)}

By choosing (<) as t1 and (>,>) as t4,

S5_f = (S1_f * {t4}) $ ({t1} * S4_f)

= {(>,>,>), (=,>,>), (<,<,>), (<,<,<)}.

S5_t $ S5_f is a BRO constraint set for C@.

By applying the same approach, we can show that {(t,t,f), (t,f,t), (f,t,f), (t,f,f)} is a BOR constraint set for C@ as well as ((I1&(I2¦I3)), where I1, I2 and I3 denote distinct boolean variables. The above approach can be formally defined by using the notion of an attribute grammar. First, the syntax of predicates is defined by a context-free grammar G. Each symbol C in G is assigned two attributes T(C), denoting a set of constraints with the true value, and F(C), denoting a set of constraints with the false value. For each production in G, two attribute rules are defined, one for T(C) and the other for F(C). The evaluation of attributes for a predicate can be accomplished in conjunction with a left-to-right, bottom-to-top parsing of the predicate according to the productions in G. When the evaluation is completed, a constraint set for the predicate is produced.

5.1 Algorithm BOR_GEN for Generation of BOR Constraint Sets

This algorithm is defined as an attribute grammar in terms of productions and attribute rules. For the sake of simplicity, the context-free grammar G for predicates is ambiguous and does not contain productions for arithmetic expressions. The symbol C in grammar G denotes a predicate and is assigned two attributes BOR_t(C) and BOR_f(C). The symbol I in grammar G denote a boolean variable and the symbol E an arithmetical expression. If a production has multiple occurrences of C, the second and third occurrences of C are replaced with C1 and C2 respectively in order to distinguish between the attributes associated with different occurrences of C.

For the productions C -> I or C -> E <rop> E, where <rop> is a relational operator
  BOR_t(C) = {(t)} and BOR_f(C) = {(f)}.
For the production C -> ~C1
  BOR_t(C) = BOR_f(C1) and BOR_f(C) = BOR_t(C1).
For the production C -> (C1 ¦ C2)
  BOR_f(C) = BOR_f(C1) % BOR_f(C2), and
  BOR_t(C) = (BOR_t(C1) * {f2}) $
        ({f1} * BOR_t(C2)),
  where f1 is in BOR_f(C1), f2 in BOR_f(C2), and (f1,f2) in BOR_f(C).
For the production C -> (C1 & C2)
  BOR_t(C) = BOR_t(C1) % BOR_t(C2), and
  BOR_f(C) = (BOR_f(C1) * {t2}) $
        ({t1} * BOR_f(C2)),
  where t1 is in BOR_t(C1), t2 in BOR_t(C2), and (t1,t2) in BOR_t(C).

Theorem 3: Algorithm BOR_GEN produces a minimum BOR constraint set for a predicate.

Theorem 4: For a predicate with n, n>0, AND/OR operators, let m be the size of the constraint set produced by algorithm BOR_GEN. 2*SQRT(n+1) <= m <= n+2, where SQRT denotes the square root function.

5.2 Algorithm BRO_GEN for Generation of BRO Constraint Sets

This algorithm uses the same grammar as algorithm BOR_GEN. However, the symbol C in grammar G is assigned two attributes BRO_t(C) and BRO_f(C). The attribute rules are given below.

For the production C -> I,
  BRO_t(C) = {(t)} and BRO_f(C) = {(f)}.
For the production C -> E > E,
  BRO_t(C) = {(>)} and BRO_f(C) = {(=), (<)}.
For the production C -> E = E
  BRO_t(C) = {(=)} and BRO_f(C) = {(>), (<)}.
For the production C -> E < E
  BRO_t(C) = {(<)} and BRO_f(C) = {(>), (=)}.
For the production C -> E >= E or C -> E => E
  BRO_t(C) = {(>), (=)} and BRO_f(C) = {(<)}.
For the production C -> E <= E or C -> E =< E
  BRO_t(C) = {(<), (=)} and BRO_f(C) = {(>)}.
For the production C -> E /= E
  BRO_t(C) = {(>), (<)} and BRO_f(C) = {(=)}.
For the production C -> ~C1
  BRO_t(C) = BRO_f(C1) and BRO_f(C) = BRO_t(C1).
For the production C -> (C1 ¦ C2)
  BRO_f(C) = BRO_f(C1) % BRO_f(C2), and
  BRO_t(C) = (BRO_t(C1) * {f2}) $
        ({f1} * BRO_t(C2)),
  where f1 is in BRO_f(C1), f2 in BRO_f(C2), and (f1,f2) in BRO_f(C).
For the production C -> (C1 & C2)
  BRO_t(C) = BRO_t(C1) % BRO_t(C2), and
  BRO_f(C) = (BRO_f(C1) * {t2}) $
        ({t1} * BRO_f(C2)),
  where t1 is in BRO_t(C1), t2 in BRO_t(C2), and (t1,t2) in BRO_t(C).

Theorem 5: Algorithm BRO_GEN produces a minimum BRO constraint set for a predicate.

Theorem 6: For a predicate with n, n>0, AND/OR operators, let m be the size of the constraint set produced by algorithm BRO_GEN. 2*SQRT(n+1) <= m <= 2*n+3, where SQRT denotes the square root function.

6. Applications of BOR and BRO Testing

Since predicates appear in both the specification and implementation of a program, BOR and BRO testing can be used for both specification-based and program-based testing. To apply BOR (BRO) testing to a program P involves
(1) application of algorithm BOR_GEN (BRO_GEN) to generate a BOR (BRO) constraint set for each predicate in P and
(2) generation of tests of P to cover each of these constraints at least once.
Note that step (2) is different from the generation of tests to cover each branch of P at least once. The reason is that a branch may have to be executed several times in order to cover each of the constraints associated with the branch at least once.

A path of P can be defined as a sequence of branches in P. Let a **constraint-path** of P be a path of P with each branch being replaced with a constraint. One method for step (2) is to (2.1) select a set of constraint-paths of P to cover each of the constraints generated in (1) at least once, and (2.2) generate a test for each selected constraint-path. (Distinct constraint-paths of P may have the same path.) This method is time-consuming since there exist very few powerful tools for automatic selection of paths and tests.

An alternative is to test P with an existing test set T, which may contain randomly generated tests from the input domain of P. During or after the testing of P with T, the constraints covered by T are determined. Then the uncovered constraints in P are used to guide the selection of constraint-paths and the generation of additional tests for P.

The implementation of program-based BOR or BRO testing requires a tool for measuring the test coverage of constraints in a program. Such a tool is slightly more complicated than that for measuring the test coverage of branches in a program. One major reason is that during an execution of a compound predicate, the outcome of each boolean variable or relational expression needs to be recorded. A BRO testing tool for Pascal has been implemented and used to carry out several experiments (see section 7.2).

One major difficulty in software testing is that a selected path of a program may be infeasible (i.e., no input for this path). Furthermore, the problem of determine whether a path is feasible is generally undecidable. Such difficulties also exist for constraint-paths. A constraint-path is infeasible due to one or more constraints on this path.

A constraint for a predicate C is said to be **infeasible for C** if it can never be covered by any test for C. For example, the constraint (t,t) is infeasible for predicate ((E1>E2) ¦ (E1=E2)), since the value of E1 can never be both greater than and equal to that of E2 at the same time. If all simple operands in predicate C are independent from each other (i.e., no common variables), then every constraint for C is feasible for C.

A constraint for a predicate in a program P is said to be **infeasible for P** if it can never be covered by any test for P. (An infeasible constraint for a predicate is infeasible for any program containing this predicate.) Assume that P contains the following segment:

if X>Y then if ((X<=Z) ¦ (Z>Y)) then ...;

The constraint (<,<) is feasible for ((X<=Z) ¦ (Z>Y)) and its coverage requires a test making X < Z < Y. However, such a test can never reach ((X<=Z) ¦ (Z>Y)) since it cannot satisfy the predicate "X>Y". Thus, the constraint (<,<) for ((X<=Z) ¦ (Z>Y)) is infeasible for P.

If any of the constrains generated by algorithm BOR_GEN (BRO_GEN) for predicates in a program is infeasible, then it is impossible to cover all of the generated constraints and thus there is no guarantee for the detection of boolean (and relational) operator faults in the program. The existence of infeasible constraints in a program does not necessarily imply the existence of faults. However, the identification of infeasible constraints in a program may help fault detection and location. Unfortunately, the problem of determining whether a constraint for a predicate is feasible for the predicate or for a program containing the predicate is generally undecidable. Tools for program analysis, verification, and symbolic execution can be used to detect many, if not all, infeasible constraints.

7. Empirical Studies of BOR and BRO Testing

In this section, we present the results of two experiments. The first experiment shows the effectiveness of BOR testing for compound predicates. The second experiment compares the coverage of BRO constraints versus that of branches for a set of multi-version programs with two different test sets.

7.1 Effectiveness of BOR Testing for Compound Predicates

We describe some of our earlier results in [Tai87] on the testing of boolean expressions. Let a **singular boolean expression** (or **SBE**) be a boolean expression in which each boolean variable occurs only once. For an SBE B with n, n>0, AND/OR operators, algorithm BOR_GEN produces a minimum BOR test set, which contains at most n+2 tests. (A constraint for a boolean expression is actually a test.) However, this test set does not guarantee the detection of incorrect parentheses and/or boolean variables.

We generated a set of 64 non-equivalent SBEs with three boolean variables (i.e., two AND/OR operators) and a set of 255 non-equivalent SBEs with four boolean variables (i.e., three AND/OR operators). The SBEs in each set have the same boolean variables. For each of these SBEs, we applied BOR_GEN to generate a test set, say T, and determine the number of other SBEs in the same set that can be distinguished by T. Our results show that for an SBE with three (four) boolean variables, the test set produced by BOR_GEN contains at most four (five) tests and is able to distinguish this SBE from, on the average, at least 98.8% (98%) of non-equivalent SBEs. Note that exhaustive testing of an SBE with three (four) boolean variables can distinguish this SBE from all non-equivalent SBEs, but it requires eight (sixteen) tests.

The results indicate that if a test set for a boolean expression (or a predicate) is effective for detecting boolean operator faults, then it is also effective for detecting other types of boolean expression faults. Thus, BOR (BRO) testing of a compound predicate should be very effective for the detection of boolean expression faults in the compound predicate. Since BRO testing are also effective for detecting incorrect relational expressions, it is expected to be very effective for detecting combinations of boolean and relational expression faults.

7.2 Coverage of BRO Constraints versus Branches

A tool called BGG was initially developed at North Carolina State University by Dr. M. Vouk and his students to measure the test coverage of statements, branches, and various types of data flow metrics for Pascal programs [Vou89]. Recently, BGG has been extended to generate BRO constraint sets for predicates in a Pascal program and to measure the test coverage of BRO constraints in the Pascal program according to a given test set.

To compare the effectiveness of BRO testing versus branch testing, we chose a set of five functionally equivalent Pascal programs, ranging from 228 to 488 Pascal statements. These five programs, denoted as L1 through L5, were tested by two different test sets, one with 1,000 randomly selected tests and the other with 100 manually selected functional tests (i.e., special value tests). Figures 1 (3) and 2 (4) show the test coverage curves of these five programs for BRO constraints and branches, respectively, based on the number of random tests (functional tests). The following table gives, for each of Figures 1 through 4, the average test coverage of these five programs.

	average coverage of BRO constraints	average coverage of branches
random testing	48%	63%
special value testing	69%	89%

Note that for random (functional) testing, the average coverage of BRO constraints is about 15% (20%) less than that of branches. These results indicate that BRO testing is significantly more stringent than branch testing and that BRO constraints provide more guidance than branches for test generation.

8. Conclusion

In this paper, we have defined BOR and BRO testing strategies for a compound predicate and have shown how to generate a minimum set of BR-constraints for a compound predicate in order to guide the generation of tests for BOR or BRO testing. Preliminary experimental results indicate that BOR and BRO testing are very effective for error detection. For a compound predicate with n AND/OR operators, BOR (BRO) testing requires at most n+2 (2*n+3) tests and thus is practical to apply. Research is underway for more theoretical and experimental studies of BOR and BRO testing for both specification-based and program-based testing. A detailed comparison of various structure-based and fault-based testing strategies for compound predicates is given in [Tai92b].

BOR and BRO testing can be easily applied to small programs or modules of large programs. For integration or system testing of a large program, BOR or BRO testing can be applied to selected modules, predicates, or constraints in this program according to various considerations such as complexity, reliability, use of global variables, code type (new, modified or reused), ... etc. As an example, the integration testing of modules M1, M2 and M3 with global variables X and Y may require the coverage of all BRO constraints for predicates accessing X and Y.

One problem with BRO testing (and its variants) is that the constraint set produced by algorithm BRO_GEN for a compound predicate is usually not unique. The choice of a BRO constraint set for a predicate can affect the coverage of BRO constraints for a given test set. We are developing a new strategy called **adaptive BRO testing**. A predicate C is associated with BRO_MORE(C), a minimum set of constraints to be covered for satisfying BRO testing. As the testing of C progresses, we keep on revising BRO_MORE(C), based on how C has been tested. When BRO_MORE(C) becomes empty, no more testing is needed for C. By doing so, we minimize the number of tests required for BRO testing of C.

One possible extension of BRO testing is to modify algorithm BRO_GEN so that occurrences of ">" and "<" in the attribute rules are replaced by "+ε" and "−ε" respectively, where $\varepsilon > 0$. The motivation is that "+ε" and "−ε" are more effective than ">" and "<", respectively, for detecting relational expression faults involving other than incorrect relational operators. (The smaller the value of ε, the more effective such constraints for fault detection.) The use of such constraints for testing a predicate is called ***b*oolean and *r*elational *e*xpression testing with parameter ε** (or **BRE(ε) testing**), and the resulting algorithm is referred as BRE(ε)_GEN. BRO testing is equivalent to BRE(ε) testing with ε being infinite. More discussion on BRE testing is given in [Tai92a].

Another extension of BOR/BRO testing is its combination with data flow testing [Fra88]. As an example, the definition of a BR-constraint can be extended to include some requirements for data flow testing. Assume that in the following predicate

(E1 = E2) & (E3 /= E4)

variables x, y and z are used. A constraint, say (<,=), for the above predicate can be extended to become (<,=,Dx,Dy,Dz), where Dx, Dy, and Dz denote definitions of x, y and z, respectively, that can reach the predicate. The coverage of (<,=,Dx,Dy,Dz) for the above predicate requires a test making E1<E2 and E3=E4 and also making the definitions of x, y and z at Dx, Dy and Dz, respectively, to be live at the predicate.

One major problem in software testing is the detection of extra or missing predicates in a program. (Such faults create extra or missing paths in a program.) Another major problem is the detection of longer or shorter predicates, i.e.,

predicates containing extra or missing AND/OR operators (and their associated operands). (Such faults create incorrect paths in a program.) The detection of extra, missing, longer, or shorter predicates in a program is usually extremely difficult because such faults can be detected only when several predicates in the program are tested together under certain situations. Since BOR and BRO testing require thorough testing of compound predicates, it is expected to be effective for detecting extra, missing, longer, or shorter predicates in a program.

BOR and BRO testing are fault-based testing strategies because they are based on the detection of boolean and relational expression faults. In [Mor90] Morell presented a general mode of fault-based testing and identified two orthogonal attributes: extent and breadth. BOR and BRO testing have local extent since only faults in a predicate are considered and they have infinite breadth since the allowed number of faults in a predicate is not bounded by a constant.

In the mutation testing for a predicate C, a mutant C* is selected by identifying one or more faults in C. To kill C*, a set of "constraints" corresponding to the faults in C* is derived and a test satisfying these constraints is chosen [DeM91]. This approach is time-consuming if a large number of mutants is selected. **Note that a BOR (BRO) constraint set for C shows how to construct a test set that guarantees the killing of all mutants of C having boolean (and relational) operator faults**. Thus, BOR/BRO testing can significantly improve the effectiveness and performance of mutation testing.

In [Ric88] a formal model for error detection, called the RELAY model, was introduced. A number of classes of faults, including boolean and relational operator faults, are considered. For each type of fault, a revealing condition set is identified such that the satisfaction of these conditions guarantees the detection of an error for the fault. The RELAY model was limited to the detection of errors resulting from a single fault in a module. BOR (BRO) testing can be applied to the RELAY model to deal with multiple boolean (and relational) operator faults .

Acknowledgment

The author would like to thank Dr. Mladen Vouk, Wayne Staats, and James O'Connor for extending BGG to generate BRO constraints and measure test coverage of BRO constraints, for carrying out the experiments reported in section 7.2, and for providing comments to improve the clarity of this paper.

References

[DeM91] DeMillo, R. A., and Offut, A. J., "Constraint-based automatic test data generation" IEEE Trans. Software Engineering, Vol. SE-17, No. 9, Sept. 1991, 900-910.

[Fos80] Foster, K. A., "Error sensitive test cases analysis (ESTCA)," IEEE Trans. Software Engineering, Vol. SE- 6, No. 3, May 1980, 258-264.

[Fra88] Frankl, P. G., and Weyuker, E. J., "An applicable family of data flow testing criteria," IEEE TSE, Vol. 14, No. 10, Oct. 1988, 1483-1498.

[How82] Howden, W. E., "Weak mutation testing and completeness of test cases," IEEE Trans. Software Engineering, Vol. SE-8, No. 4, July 1982, 371-379.

[Mor90] Morell, L. J., "A theory of fault-based testing", IEEE Trans. TSE., Vol. 16, No. 8, Aug. 1990, 844-857.

[Ric88] Richardson, D. J., and Thompson, M. C., "THe RELAY model for error detection and its application", Proc. 2nd Workshop on Software Testing, Verification, and Analysis, 1988, 223-230.

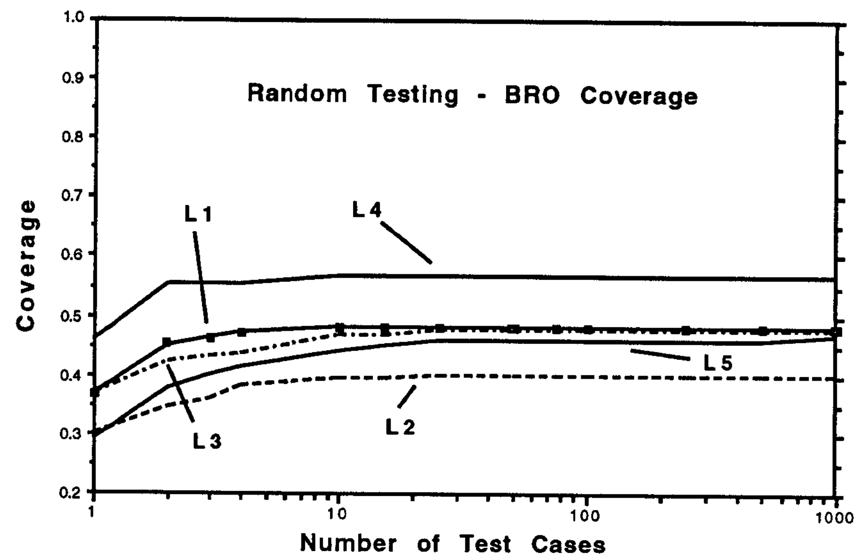
[Tai87] Tai, K. C., and Su, H. K., "Test generation for boolean expressions," Proc. COMPSAC '87, 1987, 278-283.

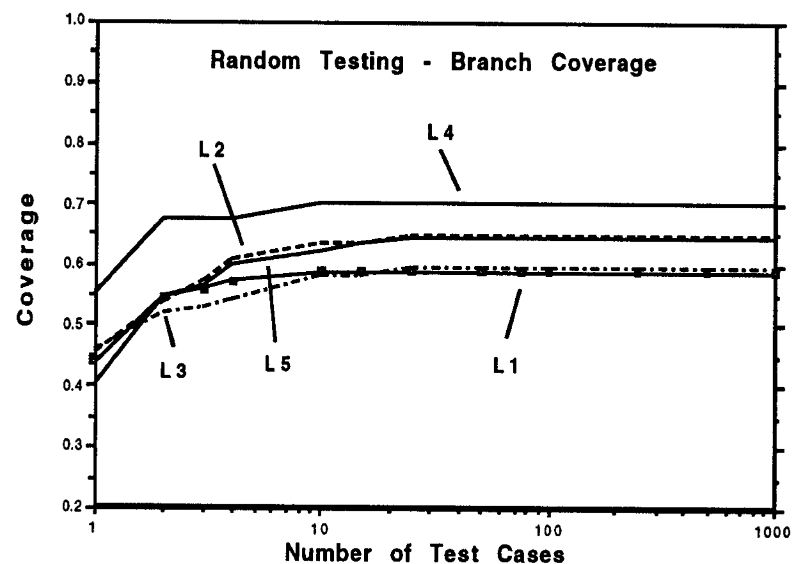
[Tai92a] Tai, K. C., "Predicate-based test generation for computer programs", Technical Report 92-23, Dept. of Computer Science, North Carolina State University, 1992.

[Tai92b] Tai, K. C., "Structure-based and Fault-Based Testing Strategies for Compound Predicates", Technical Report 92-24 , Dept. of Computer Science, North Carolina State University, 1992.

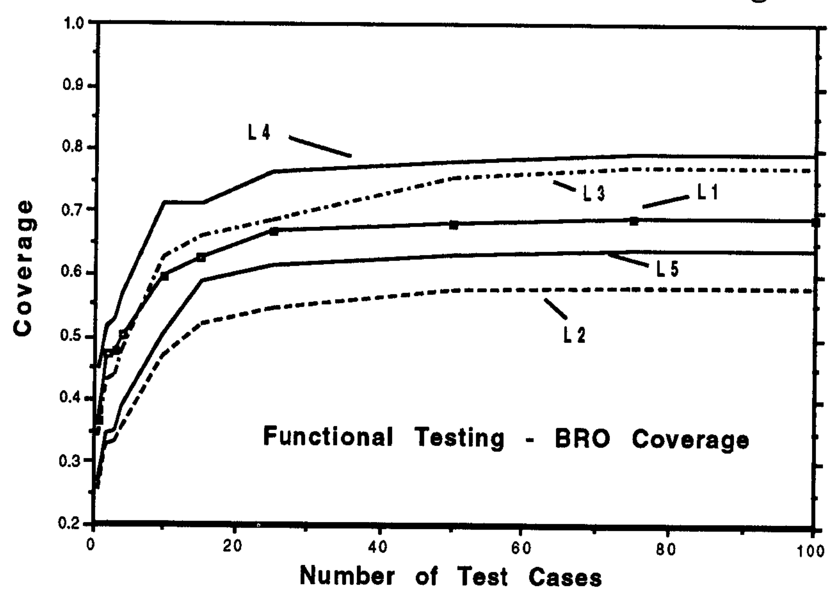
[Vou89] Vouk, M. A., Coyle, R. E., "BGG: A testing coverage tool " Proc. 7th Northwest Software Quality Conference, 1989, 212-233

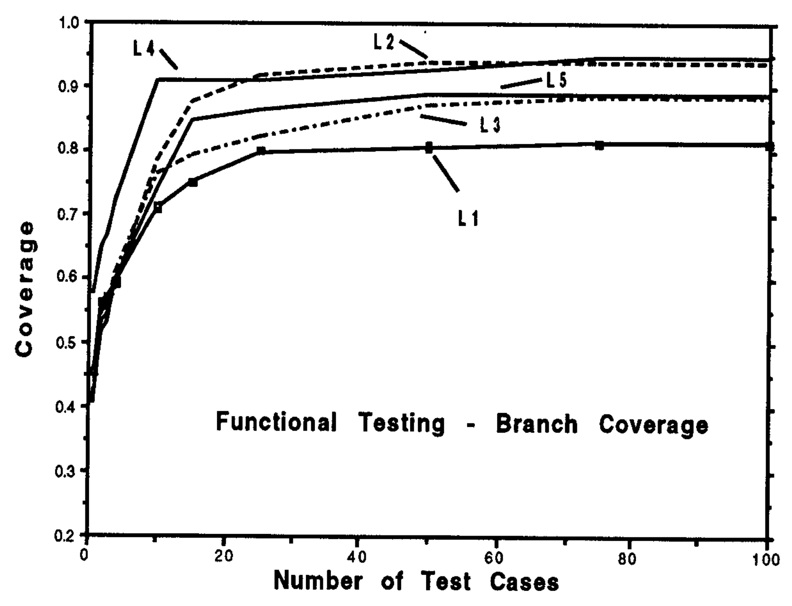
[Whi80] White, L. J., and Cohen, E. I., "A domain strategy for computer program testing," IEEE Trans. Software Engineering, Vol. SE-6, May 1980, 247-257.





Figures 1 and 2





Figures 3 and 4