COMPOUND PREDICATES TESTING USING PREDICATE-BASED STRATEGIES

Satyabrata Das
Department Of Computer Science & Engineering,
Aryan Institute Of Engineering & Technology, Bhubaneswar
Namrata Khamari
Department Of Computer Science & Engineering,
Nm Institute Of Engineering & Technology, Bhubaneswar
Biraja Nayak
Department Of Computer Science & Engineering,
Capital Engineering College, Bhubaneswar
Susmita Mohapatra
Department Of Computer Science & Engineering,
Raajdhani Engineering College, Bhubaneswar

Abstract: Predicate testing is a common approach to software testing, which requires certain types of tests for each predicate in a program. The main difficulty in predicate testing is testing compound predicates. One way to test a compound predicate C is to traverse all combinations of "t" and "f" for each simple predicate in C. But this approach requires a large number of tests. Recently, a predicate testing approach that reduces the number of tests required to detect possible errors in compound predicates has been proposed. This approach includes three predicate-based test generation criteria, called BOR (boolean operator), BRO (boolean and relational operator), and BRE (boolean and relational expression) testing criteria. Also, the paper describes a system that implements this approach for testing compound predicates in C++ programs. Finally, the paper presents the results of the experiments that have been carried out to evaluate the error-exposing ability of the system, and the testing criteria employed in it, and to compare the percentage coverage of these criteria.

Keywords: Software testing; Predicate testing; Compound predicates;

I. INTRODUCTION

One way to show the correctness of programs is to use exhaustive testing. But, this is not a practical approach because it is very costly. In order to make the testing process more practical, we must find criteria for choosing representative test cases that are comprehensive enough to span the problem domain and provide us with the required degree of confidence that the software works correctly and reliably. At the same time, this test set must be small enough so that the testing can be performed within the available schedule and cost constraints. **Predicate testing** is a common approach to software testing, which requires certain types of tests for each predicate (or condition) in a program. A number of predicate testing strategies have been proposed, such as branch testing, and domain testing, [1, 5]. A predicate 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 E_1 rop E_2, where E_1 and E_2 are arithmetic expressions and rop is one of the six relational operators {<, ≤, >, ≥, =, ≠}. A compound predicate consists of one or more binary Boolean operators {AND, OR}, two or more operands, and possibly NOT operators and parentheses. A Boolean expression is a predicate with no relational expressions.

The predicate is incorrect when it contains one or more errors involving boolean operators, boolean variables, relational operators, arithmetic expressions or parentheses. The main difficulty in predicate testing is testing compound predicates. One way to test a compound predicate C, that contains n, n > 0, AND/OR operators, is to traverse all combinations of "t" and "f" for each simple predicate in C, which requires 2^{n+1} tests, and to traverse all combinations of "<", "=", and ">" for each relational expression in C, which requires 3^{n+1} tests. Another way to test a compound predicate is strong mutation testing [2, 3]. For the predicate C, a set of mutants is generated by creating possible errors in C. To "kill" a nonequivalent mutant C^* of C, a test is chosen to distinguish C^* from C. The number of nonequivalent mutants of C due to boolean and/or relational operator errors is an exponential function of the number of boolean operators in C. If each mutant of C is allowed to have a single boolean or relational operator error only, then the number of mutants of C is a linear function of the number of AND/OR operators in C.

Tai [6] has proposed a predicate testing approach that reduces the number of tests required to detect possible errors in compound predicates. His approach includes three predicate-based test generation and selection criteria for compound predicates, called the BOR (boolean operator), the BRO (boolean and relational operator), and the

BRE (boolean and relational expression) testing criteria. For the compound predicate C, his approach requires at most n+2 tests for the detection of boolean operator errors in C, and at most 2*n+3 tests for the detection of boolean and relational operator errors in C. A software testing system has been built that implements the proposed predicate testing approach for testing compound predicates in C++ programs. The paper is organized as follows: Section 2 describes the Tai's three predicate-based testing criteria, i.e. the BOR, BRO, and BRE testing criteria, and the algorithms of generating minimum tests to cover these criteria. Section 3 describes the proposed predicate testing approach, and the system that implements it. Section 4 presents the results of the experiments that have been carried out to evaluate the error-exposing ability of the system, and the predicate-based testing criteria.

II. THE PREDICATE-BASED TESTING CRITERIA

This section describes the Tai's three predicate-based test generation and selection criteria for detecting errors in compound predicates, i.e. the BOR, BRO, and BRE testing criteria, and the algorithms of generating minimum tests to cover these criteria.

There are three types of errors that can be found in a compound predicate:

1. Boolean operator errors (incorrect AND/OR operator or missing/extra NOT operator),
2. Relational operator errors (incorrect relational operator),
3. off-by-ε , off-by-ε^* , and off-by-ε^+ errors in an arithmetic expression, which are defined as follows:

Assume that the relational expression (E_1 rop_1 E_2) is incorrect and its correct version is (E_3 rop_2 E_4), where E_1=E_2 and E_3=E_4 are nonequivalent, continuous functions with the same set of variables.

- (E_1 rop_1 E_2) is said to have an **off-by-ε error** if any test making E_3=E_4 makes $|E_1-E_2| = \varepsilon$.
- (E_1 rop_1 E_2) is said to have an **off-by-ε^* error** if any test making E_3=E_4 makes $|E_1-E_2| \geq \varepsilon$.
- (E_1 rop_1 E_2) is said to have an **off-by-ε^+ error** if any test making E_3=E_4 makes $|E_1-E_2| > \varepsilon$.

Note that an off-by-ε or off-by-ε^+ error implies an off-by-ε^* error and that the detection of an off-by-ε^* error implies that detection of off-by-ε or off-by-ε^+ error.

BR-constraints for predicates:

A BR-constraint for a predicate specifies a restriction on the value of each boolean variable or relational expression in the predicate, [6]. For a boolean variable say B, the symbols {t, f} are used to denote different types of restrictions on the value of B, where t indicates that the value of B is true, and f indicates that the value of B is false. For a relational expression, say E rop E', the symbols {t, f, <, =, >, +ε, -ε} are used to denote different types of restrictions on the value of the relational expression, where:

- **t** denotes that the value of E rop E' is true.
- **f** denotes that the value of E rop E' is false.
- **<** denotes that E - E' < 0.
- **=** denotes that E - E' = 0.
- **>** denotes that E - E' > 0.
- **+ε** denotes that 0 < E - E' ≤ +ε.
- **-ε** denotes that -ε ≤ E - E' < 0.

A constraint X 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 restriction in X. For example, the constraint {=, <} for a predicate (E_1>=E_2) OR (E_3>E_4) is covered by a test making E_1=E_2 and E_3<E_4. The result of any test for C that covers X is denoted as C(X).

A set S of constraints for a predicate C is said to be covered by a test set T if each constraint in S for C is covered by at least one test in T. The set S can be divided into two subsets S_t(C) and S_f(C), where, S_t(C) = {X in S | C(X)=t} and S_f(C) = {X in S | C(X)=f}. In order to generate a minimum constraint set for a compound predicate C, it must be transformed into its syntax tree, denoted as ST(C). In this tree, each leaf node denotes a boolean variable / relational expression and each non leaf node denotes a boolean operator. In ST(C), an immediate descendant of a non leaf node N is referred to as the input node of N. In his algorithms for generating minimum constraint sets, Tai used a new operator **A** ⊗ **B** on two sets A and B. A ⊗ B, called *the onto from A to B*, is a minimal set of (u, v) in A × B such that every element in A is chosen as u at least once and every element in B as 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 just one of them.

The BOR testing criterion:

The BOR testing criterion for a predicate C requires a set of tests to guarantee the detection of boolean operator errors. The following algorithm generates a minimum BOR constraint set S for C that should be covered by a test set T in order to satisfy the BOR testing criterion.

Algorithm BOR_min:

Let C be a predicate. For each leaf node in ST(C), its BOR constraint set is {(t), (f)}. The non-leaf nodes in ST(C) are visited from bottom to top. After visiting the root node of ST(C), a BOR constraint set for C is available. When node N is visited, the BOR constraint set S for N is constructed by using one of the following rules:

(bor$_1$) Let N be an OR node with N_1 and N_2 as its input nodes. Let S_1 and S_2 be the BOR constraint sets for N_1 and N_2 respectively.

$S\_f = S_1\_f \otimes S_2\_f$, and

$S\_t = (S_1\_t \times \{f_2\}) \cup (\{f_1\} \times S_2\_t)$

where (f_1, f_2) is any element of S_f, and f_1 and f_2 are elements of S_1_f and S_2_f, respectively.

(bor$_2$) Let N be an AND node with N_1 and N_2 as its input nodes. Let S_1 and S_2 be the BOR constraint sets for N_1 and N_2 respectively.

$S\_t = S_1\_t \otimes S_2\_t$, and

$S\_f = (S_1\_f \times \{t_2\}) \cup (\{t_1\} \times S_2\_f)$

where (t_1, t_2) is any element of S_t, and t_1 and t_2 are elements of S_1_t and S_2_t, respectively.

(bor$_3$) Let N be a NOT node with N_1 as its input node. Let S_1 be the BOR constraint set for N_1.

$S\_t = S_1\_f$, and $S\_f = S_1\_t$.

The BRO testing criteria:

The BRO testing criterion for a predicate C requires a set of tests to guarantee the detection of boolean and relational operator errors. The requirement of relational operator testing can be defined as the coverage of the constraint set {(>), (=), (<)}. The constraint set {(+ε), (=), (-ε)} is a special case of {(>), (=), (<)}, and can be used to detect off-by-ε, off-by-ε*, and off-by-ε+ errors. The following algorithm generates a minimum BRO constraint set S for C that should be covered by a test set T in order to satisfy the BRO testing criterion.

Algorithm BRO_min:

Algorithm BRO-min is algorithm BOR-min modified as follows:

- For each boolean variable in predicate C, its BRO constraint set is {(t), (f)}.
- For each relational expression (E_1 rop E_2) in C, its BRO constraint set is S = {(>), (=), (<)} and separate S into S_t and S_f according to rop. For example, if rop is ">", then S_t ={(>)}, and S_f ={(=), (<)}.
- Change the names of the rules (bor_1), (bor_2), and (bor_3) to (bro_1), (bro_2), and (bro_3), respectively.

The BRE testing criteria:

The BRE testing criterion with parameter ε, ε>0, for a compound predicate requires a set of tests to guarantee the detection of (single or multiple) boolean operator errors, relational operator errors, and off-by-ε+ errors. (The parameter ε in BRE testing is often omitted.) The following algorithm generates a minimum BRE constraint set S for C that should be covered by a test set T in order to satisfy the BRE testing criterion.

Algorithm BRE_min:

Algorithm BRE-min is algorithm BOR-min modified as follows:

- For each boolean variable in predicate C, its BRE constraint set is {(t), (f)}.
- For each relational expression (E_1 rop E_2) in C, its BRE constraint set is S = {(+ε), (=), (-ε)}, and separate into S_t and S_f according to rop in the same way as in algorithm BRO_Min.
- Change the names of the rules (bor_1), (bor_2), and (bor_3) to (bre_1), (bre_2), and (bre_3), respectively.

III. THE PROPOSED APPROACH AND THE AUTOMATED SUPPORT SYSTEM

A software testing system has been built to test C++ programs, using the proposed approach described in Sections 2. The system is written in Borland C++, and it consists of three main phases:

1. Classifying and reformatting phase.
2. Static analysis and instrumentation phase.
3. Execution and reporting phase.

In the first phase, program statements are classified and some of them are reformatted, Figure 1, without affecting the program structure to enable the insertion of the probes.

St. No.	Blk No.	Statement
1	1	#include <iostream.h>
2	1	void main( )
3	1	{
4	1	float a,b,c;
5	1	int i,n;
6	1	cout<<"Enter no. of test cases ?";
7	1	cin>>n;
8	2	for(i=1;i<=n;i++)
9	2	{
10	3	cout<<"Enter values of a,b,c ?";
11	3	cin>>a>>b>>c;
12	3	cout<<endl;
13	3	cout<<a<<","<<b<<","<<c<<"is";
14	3	if(a>0 && b>0 && c>0)
15	3	{
16	4	if(a+b>c && b+c>a && c+a>b)
17	4	{
18	5	if(a==b && a==c)
19	5	{
20	6	cout<<"equilateral"<<endl;
21	6	}
22	7	else
23	7	{
24	7	if(a==b || b==c || c==a)
25	7	{
26	8	cout<<"isosceles"<<endl;
27	8	}
28	9	else
29	9	{
30	9	cout<<"scalene"<<endl;
31	9	}
32	10	}
33	11	}
34	12	else
35	12	{
36	12	cout<<"not a triangle"<<endl;
37	12	}
38	13	}
39	14	else
40	14	{
41	14	cout<<"not a triangle"<<endl;
42	14	}
43	15	}
44	16	}

Figure 1 Example program

The main tasks of the static analysis and instrumentation phase are as follows:

1. Collect information about the relational and boolean expressions in control statements, such as if, while, for, switch, etc.
2. For each program predicate, build the syntax tree, and generate the BOR, BRO, and BRE constraint sets, by implementing the algorithms: BOR_Min, BRO_Min, and BRE_Min, respectively, described in Section 2.

The generation of the required constraint set for a compound predicate C is performed as follows:

(a) If C contains only a boolean variable, then the BOR constraint set for this predicate is {0,1}, where 0 and 1 represent f and t, respectively.

(b) If C contains only a relational expression, then its BOR constraint set is {0,1}, its BRO constraint set is {“<”, “=”, ”>”}, and its BRE constraint set is {“LHS-RHS=-1”, “LHS-RHS=0”, “LHS-RHS=1”}, where LHS and RHS represent the left-hand and right-hand sides of the relational expression, respectively.

(c) If C contains one or more operators AND/OR, then its syntax tree ST(C) is built. In this tree, each leaf node denotes a Boolean variable or a relational expression and each non leaf node denotes a boolean operator.

- For each leaf node construct and store the BOR, BRO, and BRE constraint sets as in the cases (a) and (b) above.
- Then, non leaf nodes are visited from the bottom to the root. For each non leaf node construct the BOR, BRO, and BRE constraint sets from the constraint sets of its input nodes by applying the BOR_Min, BRO_Min, and BRE_Min algorithms, respectively.

- After visiting the root node of ST(C), BOR, BRO, and BRE constraint sets for C will be available.

3. Insert a software probe in each basic block of the program to record its number when it is traversed during program execution.
4. Insert software probes before any control statement in the program to record information about its predicate during program execution.

At the end of the analysis phase, the following lists are produced:

- The BOR, BRO, and BRE constraint sets associations of each predicate.
- The number of all BOR, BRO, and BRE test cases.

Figure 2 shows part of the static analysis report generated by the system for the example program. It shows the number of all BOR, BRO, and BRE test cases, and the BOR, BRO, and BRE constraint sets associations generated by the system for the predicate a+b>c && b+c>a && c+a>b at statement 16 in block 4.

```
*********************************************************
 No. of BOR test cases : 17
 No. of BRO test cases : 27
 No. of BRE test cases : 27
*********************************************************
FOR EXP.  a+b>c&&b+c>a&&c+a>b AT ST. NO. 16  IN BLOCK 4
*********************************************************
BOR TEST CASES ARE:
111      011      101      110
BRO TEST CASES:
>>>      =>>      <>>      >=>      ><>      >>=      >><
BRE TEST CASES:
LHS1-RHS1=1,   LHS2-RHS2=1,   LHS3-RHS3=1,
LHS1-RHS1=0,   LHS2-RHS2=1,   LHS3-RHS3=1,
LHS1-RHS1=-1,  LHS2-RHS2=1,   LHS3-RHS3=1,
LHS1-RHS1=1,   LHS2-RHS2=0,   LHS3-RHS3=1,
LHS1-RHS1=1,   LHS2-RHS2=-1,  LHS3-RHS3=1,
LHS1-RHS1=1,   LHS2-RHS2=1,   LHS3-RHS3=0,
LHS1-RHS1=1, LHS2-RHS2=1, LHS3-RHS3=-1
*********************************************************
```

Figure 2. Part of the static analysis report generated by the system for the example program

The function of the execution and reporting phase is monitoring the fulfillment of the BOR, BRO, and BRE testing criteria, during test runs of the program being tested. During program execution with test runs, the inserted probes record:

1. The value of each boolean expression and/or each relational expression in the traversed control statements.
2. The values of the left-hand side and the right-hand side of each relational expression in the traversed control statements.

At the end of a test run, the system reports on:

- The uncovered BOR, BRO, and BRE constraint sets, for each predicate:
- The percentage coverage of BOR, BRO, and BRE constraint sets.

Figure 3 shows a part of the report produced by the system after a test run for the example program. The figure shows the percentage coverage of BOR, BRO, and BRE constraint sets, for the whole program. It also shows the uncovered constraint sets for the predicate a+b>c && b+c> a&& c+a>b.

```
******************  RUN ( 1 )  ************************
*  35.3 % OF BOR TEST CASES ARE COVERED.
*  25.9 % OF BRO TEST CASES ARE COVERED.
*  22.2 % OF BRE TEST CASES ARE COVERED.
```

```
*******************************************************
FOR EXP.  a+b>c&&b+c>a&&c+a>b AT ST. NO. 16  IN BLOCK 4
*******************************************************
Not  covered  BOR test cases:
011        110
Not  covered  BRO test cases:
=>>         <>>         ><>         >>=         >><
Not  covered  BRE test cases:
LHS1-RHS1=0,  LHS2-RHS2=1,  LHS3-RHS3=1,
LHS1-RHS1=-1,  LHS2-RHS2=1,  LHS3-RHS3=1
LHS1-RHS1=1,  LHS2-RHS2=0,  LHS3-RHS3=1,
LHS1-RHS1=1, LHS2-RHS2=-1, LHS3-RHS3=1,
LHS1-RHS1=1,  LHS2-RHS2=1,  LHS3-RHS3=0,
LHS1-RHS1=1, LHS2-RHS2=1, LHS3-RHS3=-1
*******************************************************
```

Figure 3. A part of the system report after a test run of the example program

The tester then re-runs the program with test data until the BOR, BRO, and BRE testing criteria, are covered, if possible.

IV. THE EXPERIMENTS

This section describes the experiments that have been carried out in order to evaluate the error-exposing ability of the system, and the predicate-based testing criteria. In these experiments, fourteen C++ programs were selected and seeded with errors one at a time. In each case, the erroneous program was analyzed, instrumented, and executed with the test data. The output of this execution was compared with the correct output, which was obtained by executing the correct program with the same data. The success of the system in discovering the error is judged by the occurrence of a deviation in the actual output. A criterion is considered to be effective in discovering an error if the data chosen to fulfil it caused a deviation in the actual output. The errors that were seeded into programs in these experiments fall into two categories: domain errors and computation errors. The definitions of these two categories have been given by Howden [4]. Table 1 shows the types of these errors and their frequencies in the experiments.

Table 1. The types of seeded errors and their frequencies

Code	Error type	Error Frequency
C	***Computation errors***	
C1	wrong variable reference	4
C2	incorrect constant value	22
C3	statement wrongly placed	1
D	***Domain errors***	
D1	wrong relational operator	50
D2	a variable replaced by a constant	3
D3	wrong variable reference	21
D4	incorrect constant value	33
D5	statement wrongly placed	2

The results of the experiments showed that the three predicate-based criteria have discovered 127 out of 142 errors, which represents 89.4% of all seeded errors. Table 2 shows the percentage of errors of each type discovered by the three predicate-based criteria: BOR, BRO, and BRE. It can be seen from Table 2 that:

- BRO has the highest ability of discovering the errors of type D1.
- BOR and BRO have discovered the same percentage of the errors of type D2, D3, and D5, which is higher than BRE.
- BRO have discovered the same percentage of the errors of type D4, which is higher than BOR and BRE.
- BOR have discovered the same percentage of the errors of type D6 and C2, which is higher than BRO and BRE.
- BRO, and BRE have discovered the same percentage of the errors of type C1, which is higher than BOR.
- All criteria have discovered all of the errors of type C3.

Table 2 Percentage of errors of each type discovered by the three predicate-based criteria

Error-type	Predicate-Based			
	BOR	BRO	BRE	Predicate-Based
C1	50.0%	75.0%	75.0%	75.0%
C2	90.9%	86.4%	81.8%	90.9%
C3	100%	100%	100%	100%
D1	86.0%	94.0%	86.0%	96.0%
D2	66.7%	66.7%	0%	66.7%
D3	76.2%	76.2%	57.1%	76.2%
D4	84.8%	87.9%	84.8%	90.9%
D5	100%	100%	50.0%	100%
D6	83.3%	33.3%	0%	83.3%

Table 3 Percentage coverage of the three predicate-based criteria,

Program	No. of test cases			Percentage coverage		
	BOR	BRO	BRE	BOR	BRO	BRE
Prog 01	13	19	19	100.0	100.0	89.4
Prog 02	8	12	12	100.0	83.3	83.3
Prog 03	8	12	12	100.0	100.0	100.0
Prog 04	8	12	12	100.0	100.0	100.0
Prog 05	12	18	18	100.0	94.4	94.4
Prog 06	12	18	18	100.0	94.4	94.4
Prog 07	4	6	6	100.0	83.3	50.0
Prog 08	17	27	27	100.0	81.5	59.3
Prog 09	4	6	6	100.0	83.3	83.3
Prog 10	4	6	6	100.0	100.0	100.0
Prog 11	6	9	9	100.0	66.7	66.7
Prog 12	28	42	42	96.4	97.6	97.6
Prog 13	14	22	22	100.0	68.2	59.1
Prog 14	14	21	21	100.0	95.2	95.2

Another experiment has been conducted. In this experiment, each of the fourteen programs was executed with test data to reach 100% coverage of the BOR criterion, if possible, and the percentage coverage of the other criterion were noted. Table 3 shows the results of this experiment. It can be seen from this table that covering the BOR criterion does not always guarantee the coverage of the other predicate-based criteria. This means that more test data are needed to cover the other criteria, and this in turn increases the possibility of discovering more errors. These results indicate that these criteria complement each other.

V. CONCLUSION

The predicate testing approach, proposed by Tai [6], reduces the number of tests required to detect possible errors in compound predicates. It considers the boolean and relational operators errors. The paper presented a brief description of the BOR, the BRO, and the BRE testing criteria included in Tai's approach. Then, the paper described the proposed predicate testing approach. Also, the paper described the components of a system that has been built to implement this approach for testing compound predicates in C++ programs. Experiments have been carried out to evaluate the error-exposing ability of the constructed system, and the predicate-based testing criteria. The results of the experiments indicate that the system is very effective in discovering the errors in compound predicates. Also, the results indicate that covering some criteria does not always guarantee the coverage of the others. This means that more test data are needed to cover the other criteria, and this in turn increases the possibility of discovering more errors. Thus, the criteria employed in the system complement each other.

REFERENCES

1. B. Beizer, Software testing techniques, 2nd edition, Van Nostrand Reinhold, New York, 1990.
2. T.A. Budd, R.J. Lipton, F.G. Sayward, and R.A. DeMillo, "Mutation analysis", Technical Report No. 155, Department of Computer Science, Yale University, 1979.
3. R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on test data selection: help for the practicing programmer", IEEE Computer, vol. 11, pp. 34-41, 1978.
4. W.E. Howden, "Reliability of the path analysis testing strategy", IEEE Transactions on Software Engineering, vol. SE-2, pp. 208-215, 1976.
5. W.E. Howden, Functional program testing and analysis, McGraw-Hill Book Co., 1987.
6. K.-C. Tai, "Theory of fault-based predicate testing for computer programs", IEEE Transactions on Software Engineering, vol. 22, pp. 552-562, 1996.