Heuristic Approaches for Equivalent Mutant Problem
Inderjot1 Dr. Rakesh Kumar2
1Research Scholar 2Professor
1,2Department of Computer Science & Application Engineering
1,2Kurukshetra University, Kurukshetra

Abstract— Mutation testing works on seeding faults into the program using mutation operators. These faulty programs are known as mutants. Test suites are applied on these mutants to recognize if they are killed or not. Test suite that kill mutants are said to be adequate. Sometimes, it is difficult to kill all the mutants because of equivalent mutants. Equivalent mutants are the mutants syntactically different and semantically similar to the original program. In this paper some heuristics have been proposed to avoid equivalent mutant problems.

Key words: Equivalent Mutants, Mutants, Mutation Operators, Mutation Testing, Test Suite

I. INTRODUCTION

Mutation testing is a fault based testing approach. Mutation testing is a method of white box testing. Mutation testing was invented by Richard Lipton in 1971[1]. Mutation testing refers to seeding slight change into source code [2]. Each faulted version of source code is known as Mutants. Mutants are of two types [3]:

1) First order mutant- mutants that are generated by seeding a single fault in the original program.
2) Higher order mutants- mutants that are generated by seeding more than one fault in the original program.

Mutation testing uses two principles-

1) Competent programmer hypothesis-it states that mutants are created by seeding faults in the program. The faults should be like as committed by competent programmers [4].
2) Coupling effect hypothesis- It states that simple faults (first order mutants) are coupled to form complex faults (higher order mutants). Test suite which can able to detect the simple fault can also detect the complex fault [4].

Mutation testing provides a test adequacy criterion. To evaluate the quality of test-suite, test-suite is executed over the mutants. Output of original program is compared against mutant. If compared outputs are different then mutant is said to be killed by the test suite. If a mutant is not killed by any test suite that type of mutants are said to be equivalent mutants. A test suite must satisfy three conditions (RIP model) [1] to kill mutants-

1) Reachability- mutated statement must be executed by the test suite.
2) State Infection- mutated statement must have different state than original.
3) State Propagation- mutant must have different final state from the original program.

Mutation adequacy score is used to measure the effectiveness of test suite in terms of its ability to detect faults. Mutation adequacy score is calculated by [5]-

\[
\text{Mutation adequacy score} = \frac{D}{M} - E
\]

Where, D- no. of killed mutants, M- Total no. of mutants, E- no. of Equivalent mutants

Mutation adequacy score results between real values of 0.0 and 1.0. High adequacy score represents adequate test suite. Calculating exact mutation score is difficult due to the presence of equivalent mutants.

II. LITERATURE REVIEW

Equivalent Mutant Problem is the major obstacle of mutation testing. The problem of determining whether a mutant is equivalent to the original program is theoretically un-decidable. Equivalent mutants are the mutants syntactically different and semantically similar to the original program. A program M is derived from program P by slight change. If P and M both produce same result on every test-suite then M is said to be equivalent mutant of P.

For example-

```
int max(int[a]);
{
    int b,j;
    b=0;
    n=a.length;
    for(i=0;i<n;i++)
    {
        [a[i]>=a[b]]
        b=i;
    }
    return a[b];
}
```

To overcome equivalent mutant problem there are 17 relevant techniques divided into three categories [6]-

A. Detecting Techniques:
1) Compiler optimization techniques [7].
2) Mathematical constraints to automatically detect equivalent mutants [8],[9].
3) Program slicing to assist in the detection of equivalent mutants [10].
5) Margrave’s change impact analysis [12].
6) Lesar model-checker for eliminating equivalent mutants [13].

B. Avoiding Techniques:
1) Selective mutation [14].
2) Avoiding equivalent mutant generation using program dependence analysis [15].
3) Co-evolutionary search techniques [16].
4) Equivalency conditions to eliminate equivalent mutants for object-oriented mutation operators [17].
5) Fault hierarchy to improve the efficiency of DNF logic mutation testing [18].
6) Distinguishing the equivalent mutants by semantic exception hierarchy [19].
7) Higher mutation testing [20].
C. Suggesting Techniques:
1) Using Bayesian-learning based guidelines to help to determine equivalent mutants [21].
2) Examining the impact of equivalent mutants on coverage [22].
3) Using the impact of dynamic invariants [23].
4) Examining changes in coverage to distinguish equivalent mutants [24].

Recently researchers focused on avoiding and suggesting techniques. Detecting techniques are hard to implement. Detecting techniques never results erroneous like avoiding and suggesting techniques.

III. HEURISTICS FOR AVOIDING EQUIVALENT MUTANTS
As discussed, while mutants are generated, there are chances that equivalent mutants may be created. To detect the equivalent mutant is an un-decidable problem. The better option can be, don’t create equivalent mutant. In the following text a number of heuristics have been proposed to avoid the creation of equivalent mutants. Table I. [25] shows list of mutant operators in Operator-class.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OAAA</td>
<td>Arithmetic assignment mutation</td>
<td>OAAN</td>
<td>Arithmetic operator mutation</td>
</tr>
<tr>
<td>OABA</td>
<td>Arithmetic assignment by Bitwise assignment</td>
<td>OABN</td>
<td>Arithmetic by bitwise operator</td>
</tr>
<tr>
<td>OAEA</td>
<td>Arithmetic assignment by plain assignment</td>
<td>OALN</td>
<td>Arithmetic by logical operator</td>
</tr>
<tr>
<td>OARN</td>
<td>Arithmetic operator by relational operator</td>
<td>OASA</td>
<td>Arithmetic assignment by shift assignment</td>
</tr>
<tr>
<td>OASN</td>
<td>Arithmetic operator by shift operator</td>
<td>OBAA</td>
<td>Bitwise assignment by arithmetic assignment</td>
</tr>
<tr>
<td>OBAN</td>
<td>Bitwise operator by arithmetic operator</td>
<td>OBBN</td>
<td>Bitwise operator mutation</td>
</tr>
<tr>
<td>OBBA</td>
<td>Bitwise assignment mutation</td>
<td>OBEA</td>
<td>Bitwise assignment by plain assignment</td>
</tr>
<tr>
<td>OBLN</td>
<td>Bitwise operator by logical operator</td>
<td>OBNG</td>
<td>Bitwise negation</td>
</tr>
<tr>
<td>OBRN</td>
<td>Bitwise operator by relational operator</td>
<td>OBSA</td>
<td>Bitwise assignment by shift assignment</td>
</tr>
<tr>
<td>OCNG</td>
<td>Logical context negation</td>
<td>OEAA</td>
<td>Plain assignment by arithmetic assignment</td>
</tr>
<tr>
<td>Oeba</td>
<td>Plain</td>
<td>OESA</td>
<td>Plain</td>
</tr>
</tbody>
</table>

Using some of the above operators, equivalent mutants are generated. Equivalent mutants are produced at certain conditions. Warnings are produced to avoid the generation of equivalent mutants. Some of them are as follows-

A. OAAA- Arithmetic Assignment Mutation (A+=B → A=B). For Example-

```c
int a, b;
scanf("%d %d", &a, &b);
if (b == 0)
    { a = b; printf("%d", a); } // Original program
```

```c
int a, b;
scanf("%d %d", &a, &b);
if (b == 0)
    { a = b; printf("%d", a); } // Equivalent Mutant
```

Be careful if the value of operands is zero. During generation of mutants using OAAA value of operands must greater than zero.
B. OAAN- Arithmetic Operator Mutation (A*B → A+B). For Example-

```
int a,b,c;
scanf("%d %d",&a,&b);
if(a==0)
{
 a+=b;
 printf("%d",a);
}
Original program
```

Equivalent Mutant

```
int a,b,c;
scanf("%d %d",&a,&b);
if(a==2&&b==2)
{
 c=a*b;
 printf("%d",a);
}
Equivalent Mutant
```

Be careful if the values of operands are 0 and 2. During generation of mutants using OAAN, value of operands must greater than 0 and not equal to 2.

C. OAEA- Arithmetic Assignment by Plain Assignment (A+=B → A=B). For Example-

```
int a,b;
scanf("%d %d",&a,&b);
if(a==0)
{
 a+=b;
 printf("%d",a);
}
Original program
```

Equivalent Mutant

```
int a,b;
scanf("%d %d",&a,&b);
if(a==0)
{
 a=b;
 printf("%d",a);
}
Equivalent Mutant
```

Be careful if the value of operands is 0. During generation of mutants using OAEA, value of operands must greater than zero.

D. OBAN- Bitwise Operator by Arithmetic Operator (A&B → A*B). For Example-

```
int a,b,c;
// if input of a and b is '0' or '1'
scanf("%d %d",&a,&b);
if (a==b)
{
 c=a^b;
 printf("%d",c);
}
Original program
```

Equivalent Mutant

```
int a,b,c;
scanf("%d %d",&a,&b);
if((a==b) & (a^b))
{
 c=a&b;
 printf("%d",c);
}
Equivalent Mutant
```

Be careful if the value of operands is same. During generation of mutants using OBAN, value of operands must be different.

E. OBNB- Bitwise Operator Mutation (A&B →A|B). For Example-

```
int a,b,c;
scanf("%d %d",&a,&b);
if(a==b)
{
 c=a&b;
 printf("%d",c);
}
Original program
```

Equivalent Mutant

```
int a,b,c;
scanf("%d %d",&a,&b);
if(b=c)
{
 b&c;
 printf("%d",b&c);
}
Equivalent Mutant
```

Be careful on replacing bitwise OR by bitwise AND operator, operands must have different values.

F. OBLN- Bitwise Operator by Logical Operator (A|B → A||B). For Example-

```
char sub;
int marks;
scanf("%d %c ",&marks,&sub,);
if((marks>80)||(sub="P" && marks>70)||(sub="C" && marks>80 ))
{
 printf("student will choose medical/non-medical");
}
else
{
 printf("student will choose commerce");
}
Original program
```

Equivalent Mutant

```
char sub;
int marks;
scanf("%d %c ",&marks,&sub,);
if((marks>80)||(marks>70)||(sub="C" && marks>80 ))
{
 printf("student will choose medical/non-medical");
}
else
{
 printf("student will choose commerce");
}
Equivalent Mutant
```

Be careful if value of operands uses values as true or false using bitwise OR. If bitwise OR replaced with logical OR using true or false values then they will produce same output.

G. OBNG- Bitwise Negation (~X Op Y → ~ (X Op Y)). For Example-

```
int a,b,c;
scanf("%d %d",&a,&b);
c=~(a^b);
printf("%d",c);
```

```
int a,b,c;
scanf("%d %d",&a,&b);
if(b^c)
{
 printf("b& c not equal");
}
else
{
 printf("b&c are equal");
}
```

Be careful on using bitwise complement operator with bitwise XOR operator.

H. OBRN- Bitwise Operator By Relational Operator (A^B → A!=B). For Example-

```
int a,b,c;
scanf("%d %d",&a,&b);
if(b^c);
printf("b& c not equal");
else
printf("b&c are equal");
```

```
int a,b,c;
scanf("%d %d",&a,&b);
if(b!=c);
printf("b& c not equal");
else
printf("b&c are equal");
```

Be careful if bitwise XOR replaced with "!=" relational operator then equivalent mutant generated.

I. OEA- Plain assignment by arithmetic assignment (a=b → a+=b). For Example-

```
int a,b;
scanf("%d %d",&a,&b);
if (a==0)
{
 a+=b;
 printf("%d",a);
}
Original program
```

```
int a,b;
scanf("%d %d",&a,&b);
if(a==0)
{
 a+=b;
 printf("%d",a);
}
Equivalent Mutant
```

Be careful on using OEAA operator if value of the operand is zero.
J. **OLBN- Logical Operator by Bitwise Operator** (A&\&B \rightarrow A&\&B). For Example-

```c
int a,b,c;
scanf("%d %d %d",a,b,c);
if((a>b)&(a>c))
    printf("a is greater");
Elseif((a<b)\&\&(c<b))
    printf("b is greater");
Elseif((a<c)\&\&(b<c))
    printf("c is greater");
Else
    printf("two of them are equal");
```

Original program

```
int a,b,c;
scanf("%d %d %d",a,b,c);
if((a>b)||(a>c))
    printf("a is greater");
Elseif((a<b)\&\&(c<b))
    printf("b is greater");
Elseif((a<c)\&\&(b<c))
    printf("c is greater");
Else
    printf("two of them are equal");
```

Equivalent mutant

Be careful during the replacement of bitwise AND by logical AND if conditions result true or false.

K. **OLLN- Logical Operator by Relational Operator** (A&\&B \rightarrow A||B). For Example-

```c
int x,y,z;
scanf("%d %d %d",x,y,z);
if((x>y)\&\&(x>=z))
    printf("x>=z");
Elseif((x<y)\&\&(x!=z))
    printf("x!=z");
Else
    printf("x==z");
```

Original program

```
int x,y,z;
scanf("%d %d %d",x,y,z);
if((x>y)||(x>=z))
    printf("x>=z");
Elseif((x<y)||(x!=z))
    printf("x!=z");
Else
    printf("x==z");
```

Equivalent mutant

Be careful when single OR operator in a conditional statement makes the whole result of conditional statement true if any condition in it is true, regardless of AND operator.

L. **OLRN- Logical Operator by Relational Operator** (A&\&B \rightarrow A<\&B). For Example-

```c
int a,b,c;
scanf("%d %d %d",a,b,c);
a=b=c;
if(!a)
    printf("&c not equal");
Else
    printf("b\&c not equal");
```

Original program

```
int a,b,c;
scanf("%d %d %d",a,b,c);
a=b=c;
if(!a)
    printf("&c not equal");
Else
    printf("b\&c not equal");
```

Equivalent mutant

Be careful when a "!" logical operator is used with relational operator which makes condition equal to zero.

M. **ORRN- Relational Operator Mutation** (A<\&B \rightarrow A<\&B). For Example-

```c
int a,b,c;
scanf("%d %d %d",a,b,c);
if(a<b)
    c=a;
Else
    c=b;
```

Original program

```
int a,b,c;
scanf("%d %d %d",a,b,c);
if(a<b)
    c=a;
Else
    c=b;
```

Equivalent mutant

Be careful on using ORRN mutant operator for finding largest of two "\" \" should not replace by "\"\"."


