Software Testing Technique
Chapter 4

Mutation Testing

Junjie Chen（陈俊洁）
Office: 55-A317
Email: junjiechen@tju.edu.cn

2022

Test Adequacy

- Consider a program P written to meet a set R of functional requirements. We notate such a P and R as (P, R). Let R contain n requirements labeled R1, R2,…, Rn .
- Suppose now that a set T containing k tests has been constructed to test P to determine whether or not it meets all the requirements in R . Also, P has been executed against each test in T and has produced correct behavior.
- We now ask:
  - Is T good enough?
  - This question can be stated differently as: Has P been tested thoroughly?, or as: Is T adequate?

Program mutation

- **Suppose that program P has been tested against a test set T and P has not failed on any test case in T. Now suppose we do the following:**

Changed to

P  **P'**

- **What behavior do you expect from P' against tests in T?**
- **P' is known as a mutant of P.**
- **There might be a test t in T such that P(t)≠P'(t). In this case we say that t distinguishes P' from P. Or, that t has killed P'.**
- **There might be not be any test t in T such that P(t)≠P'(t). In this case we say that T is unable to distinguish P and P'. Hence P' is considered live in the test process.**

- **If there does not exist any test case t in the input domain of P that distinguishes P from P' then P' is said to be equivalent to P.**
- **If P' is not equivalent to P but no test in T is able to distinguish it from P then T is considered inadequate.**
- **A non-equivalent and live mutant offers the tester an opportunity to generate a new test case and hence enhance T.**

Test adequacy using mutation

- **Given a test set T for program P that must meet requirements R, a test adequacy assessment procedure proceeds as follows:**
  - **Step 1: Create a set M of mutants of P. Let M={M_0, M_1…M_k}. Note that we have k mutants.**
  - **Step 2: For each mutant M_i find if there exists a *t* in T such that $M_i(t) \neq P(t)$. If such a t exists then M_i is considered killed and removed from further consideration.**
  - **Step 3: At the end of Step 2 suppose that $k_1 \leq k$ mutants have been killed and $(k-k_1)$ mutants are live.**
    - **Case 1: $(k-k_1)=0$: T is adequate with respect to mutation.**
    - **Case 2: $(k-k_1)>0$ then we compute the mutation score (MS) as follows:**
      - **$MS=k_1/(k-e)$**
      - **Where *e* is the number of equivalent mutants. Note: $e \leq (k-k_1)$.**

Test enhancement using mutation

- **One has the opportunity to enhance a test set T after having assessed its adequacy.**
  - **Step 1: If the mutation score (MS) is 1, then some other technique, or a different set of mutants, needs to be used to help enhance T.**
  - **Step 2: If the mutation score (MS) is less than 1, then there exist live mutants that are not equivalent to P. Each live mutant needs to be distinguished from P.**
  - **Step 3: Hence a new test *t* is designed with the objective of distinguishing at least one of the live mutants; let us say this is mutant m.**
  - **Step 4: If *t* does not distinguish m then another test t needs to be designed to distinguish m. Suppose that t does distinguish m.**
  - **Step 5: It is also possible that t also distinguishes other live mutants.**
  - **Step 6: One now adds t to T and re-computes the mutation score (MS).**
  - **Repeat the enhancement process from Step 1.**

Error detection using mutation

- As with any test enhancement technique, there is no guarantee that tests derived to distinguish live mutants will reveal a yet undiscovered error in P. Nevertheless, empirical studies have found to be the most powerful of all formal test enhancement techniques.
- The next simple example illustrates how test enhancement using mutation detects errors.
- Consider the following function foo that is required to return the sum of two integers x and y. Clearly foo is incorrect.

```
int foo(int x, y){
    return (x-y);
}
```


This should be return (x+y)

- **Now suppose that foo has been tested using a test set T that contains two tests:**

T={ t1: <x=1, y=0>, t2: <x=-1, y=0>}

- **First note that foo behaves perfectly fine on each test in, i.e. foo returns the expected value for each test case in T. Also, T is adequate with respect to all control and data flow based test adequacy criteria.**
- **Let us evaluate the adequacy of T using mutation. Suppose that the following three mutants are generated from foo.**

M1:
```
int foo(int x, y){
return (x+y);
}
```

M2:
```
int foo(int x, y){
return (x-0);
}
```

M3:
```
int foo(int x, y){
return (0+y);
}
```

- **Note that M1 is obtained by replacing the - operator by a + operator, M2 by replacing y by 0, and M3 by replacing x by 0.**

- **Next we execute each mutant against tests in T until the mutant is distinguished or we have exhausted all tests. Here is what we get.**

T={ t1: <x=1, y=0>, t2: <x=-1, y=0>}

Test (t)	foo(t)	M1(t)	M2(t)	M3(t)
t1	1	1	1	0
t2	-1	-1	-1	0
		Live	Live	Killed

- After executing all three mutants we find that two are live and one is distinguished. Computation of mutation score requires us to determine of any of the live mutants is equivalent.
- Let us examine the following two live mutants.

M1:
```
int foo(int x, y){
return (x+y);
}
```

M2:
```
int foo(int x, y){
return (x-0);
}
```

- Let us focus on M1. A test that distinguishes M1 from foo must satisfy the following condition:

$$x-y \neq x+y \text{ implies } y \neq 0.$$

- Hence we get t3: <x=1, y=1>

- **Executing foo on t3 gives us foo(t3)=0. However, according to the requirements we must get foo(t3)=2. Thus t3 distinguishes M1 from foo and also reveals the error.**

M1:

```
int foo(int x, y){
return (x+y);
}
```

M2:

```
int foo(int x, y){
return (x-0);
}
```

Distinguishing a mutant

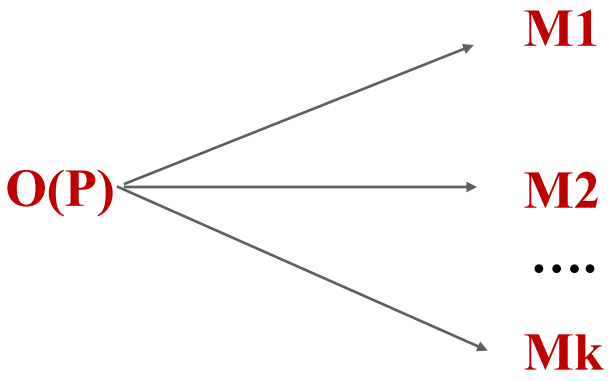
- **A test case t that distinguishes a mutant *m* from its parent program P program must satisfy the following RIPR model:**
  - **Reachability: *t* must cause *m* to follow a path that arrives at the mutated statement in *m*.**
  - **Infection: If S_{in} is the state of the mutant upon arrival at the mutant statement and S_{out} the state soon after the execution of the mutated statement, then $S_{in} \neq S_{out}$.**
  - **Propagation: If difference between S_{in} and S_{out} must propagate to the output of *m* such that the output of *m* is different from that of P.**
  - **Revealability : The tester must observe part of the incorrect output**

Equivalent mutants

- **The problem of deciding whether or not a mutant is equivalent to its parent program is undecidable. Hence there is no way to fully automate the detection of equivalent mutants.**
- **The number of equivalent mutants can vary from one program to another. However, empirical studies have shown that one can expect about 5% of the generated mutants to the equivalent to the parent program.**
- **Identifying equivalent mutants is generally a manual and often time consuming--as well as frustrating--process.**

Mutant operators

- A mutant operator *O* is a function that maps the program under test to a set of k (zero or more) mutants of P.



- A mutant operator creates mutants by making simple changes in the program under test.
- For example, the "variable replacement" mutant operator replaces a variable name by another variable declared in the program. An "relational operator replacement" mutant operator replaces relational operator with another relational operator.

Mutant operators: Examples

Mutant operator	In P	In mutant
Variable replacement	z=x*y+1;	x=x*y+1; z=x*x+1;
Relational operator replacement	if (x<y)	if(x>y) if(x<=y)
Off-by-1	z=x*y+1;	z=x*(y+1)+1; z=(x+1)*y+1;
Replacement by 0	z=x*y+1;	z=0*y+1; z=0;
Arithmetic operator replacement	z=x*y+1;	z=x*y-1; z=x+y-1;

Mutants: First order and higher order

- **A mutant obtained by making exactly "one change" is considered first order.**
- **A mutant obtained by making two changes is a second order mutant. Similarly higher order mutants can be defined. For example, a second order mutant of z=x+y; is x=z+y; where the variable replacement operator has been applied twice.**
- **In practice only first order mutants are generated for two reasons: (a) to lower the cost of testing and (b) most higher order mutants are killed by tests adequate with respect to first order mutants.**

Mutant operators: basis

- **A mutant operator models a simple mistake that could be made by a programmer**
- **Several error studies have revealed that programmers--novice and experts--make simple mistakes. For example, instead of using x<y+1 one might use x<y.**
- **While programmers make "complex mistakes" too, mutant operators model simple mistakes.**

Mutant operators: Goodness

- **The design of mutation operators is based on guidelines and experience. It is thus evident that two groups might arrive at a different set of mutation operators for the same programming language. How should we judge whether or not that a set of mutation operators is "good enough?"**
- **Informal definition:**
  - **Let S1 and S2 denote two sets of mutation operators for language L. Based on the effectiveness criteria, we say that S1 is superior to S2 if mutants generated using S1 guarantee a larger number of errors detected over a set of erroneous programs.**
- **Generally one uses a small set of highly effective mutation operators rather than the complete set of operators.**
- **Experiments have revealed relatively small sets of mutation operators for C and Fortran. We say that one is using "constrained" or "selective" mutation when one uses this small set of mutation operators.**

Competent programmer hypothesis (CPH)

- **CPH states that given a problem statement, a programmer writes a program P that is in the general neighborhood of the set of correct programs.**
- **A more reasonable interpretation of the CPH is that the program written to satisfy a set of requirements will be a few mutants away from a correct program.**
- **The CPH assumes that the programmer knows of an algorithm to solve the problem at hand, and if not, will find one prior to writing the program.**
- **It is thus safe to assume that when asked to write a program to sort a list of numbers, a competent programs knows of, and makes use of, at least one sorting algorithm. Mistakes will lead to a program that can be corrected by applying one or more first order mutations**

Tools for mutation testing

- As with any other type of test adequacy assessment, mutation based assessment must be done with the help of a tool.
- There are few mutation testing tools available freely. Two such tools are ***Proteum*** for C from Professor Maldonado and ***muJava*** for Java from Professor Jeff Offutt.
- A typical tool for mutation testing offers the following features
  - A selectable palette of mutation operators.
  - Management of test set T.
  - Execution of the program under test against T and saving the output for comparison against that of mutants.
  - Generation of mutants.
  - Mutant execution and computation of mutation score using user identified equivalent mutants.
  - Incremental mutation testing: i.e. allows the application of a subset of mutation operators to a portion of the program under test.

Mutation testing

- **Adequacy assessment using mutation is often recommended only for relatively small units, e.g. a class in Java or a small collection of functions in C.**
- **However, given a good tool, one can use mutation to assess adequacy of system tests.**
- **The following procedure is recommended to assess the adequacy of system tests.**
  - **Step 1: Identify a set U of application units that are critical to the safe and secure functioning of the application. Repeat the following steps for each unit in U.**
  - **Step 2: Select a small set of mutation operators. This selection is best guided by the operators defined by Eric Wong or Jeff Offutt.**
  - **Step 3: Apply the operators to the selected unit.**
  - **Step 4: Assess the adequacy of T using the mutants so generated. If necessary, enhance T.**
  - **Step 5: Repeat Steps 3 and 4 for the next unit until all units have been considered.**
  - **We have now assessed T, and perhaps enhanced it. Note the use of incremental testing and constrained mutation (i.e. use of a limited set of highly effective mutation operators).**

Summary of Mutation Testing

- **Mutation testing is the most powerful technique for the assessment and enhancement of tests.**
- **Mutation, as with any other test assessment technique, must be applied incrementally and with assistance from good tools.**
- **Identification of equivalent mutants is an undecidable problem--similar the identification of infeasible paths in control or data flow based test assessment.**
- **While mutation testing is often recommended for unit testing, when done carefully and incrementally, it can be used for the assessment of system and other types of tests applied to an entire application tests.**
- **Mutation is a highly recommended technique for use in the assurance of quality of highly available, secure, and safe systems.**