CUI Abbottabad

Department of Computer Science

SOFTWARE TESTING

Lecture 19

Mutation Testing

CONTENT

MUTATION TESTING

- MT is also known as **fault-based testing, program mutation, error-based testing,** or **mutation analysis**.
- **Mutation Testing** is a type of software testing in which certain statements of the source code are changed/mutated to check if the test cases are able to find errors in source code.
- The goal of Mutation Testing is to ensure/assess the quality of test cases in terms of robustness that it should fail the mutated source code.
  - In other words, the goal of mutation testing is to find faults on the system under test.

MUTATION TESTING

- **Mutant** is simply the mutated version of the source code. It is the code that contains minute changes.
- Each **mutant** is a copy of the program under test, usually with a small syntactic change, which is interpreted as a fault

  –It is a type of white box testing which is mainly used for unit testing.

  –The changes in mutant program are kept extremely small, so it does not affect the overall objective of the program.
- This method is also called as Fault based testing strategy as it involves creating fault in the program.

A PROGRAM AND THREE MUTANTS -EXAMPLE

Version	Code

P (original)	int sum(int a, int b) {
	return a + b;
	}

Mutant 1	int sum(int a, int b) {
	return a - b;
	}

Mutant 2	int sum(int a, int b) {
	return a * b;
	}

Mutant 3	int sum(int a, int b) {
	return a / b;
	}

	Test data (a,b)			
	(1, 1)	**(0, 0)**	**(-1, 0)**	**(-1, -1)**
P	2	0	-1	-2
M1	0	0	-1	0
M2	1	0	0	1
M3	1	Error	Error	1

BASIC IDEA I

- In Mutation Testing:

1. We take a program and a test suite generated for that program (using other test techniques)
2. We create a number of similar programs (mutants), each differing from the original in one small way, i.e., each possessing a fault
   E.g., replacing an addition operator by a multiplication operator
3. The original test data are then run on the mutants
4. If test cases detect differences in mutants, then the mutants are said to be dead (killed), and the test set is considered adequate

BASIC IDEA II

- A mutant remains live either
- Because it is equivalent to the original program (functionally identical although syntactically different –called an equivalent mutant) or,
- The test set is inadequate to kill the mutant.
- In the latter case, the test data need to be augmented (by adding one or more new test cases) to kill the live mutant
- For the automated generation of mutants, we use mutation operators, that is predefined program modification rules (i.e., corresponding to a fault model)

SOME MUTATION OPERATORS

Operator	Description
ABS	Substitution of a variable *x* by *abs(x)*
ACR	Substitution of a variable array reference by a constant
AOR	Arithmetic operator replacement (*a+b* by *a-b)*
CRP	Substitution of a constant value
ROR	Relational operator replacement (*A and B* by *A or B*)
RSR	Return statement substitution (*return 5* by *return 0*)
SDL	Removal of a sentence

MUTANTS TYPES

- **Survived Mutants:** As we have mentioned, these are the mutants that are still alive after running test data through the original and mutated variants of the source code. These must be killed. They are also known as live mutants.
- **Killed Mutants:** These are mutants that are killed after mutation testing. We get these when we get different results from the original and mutated versions of the source code.
- **Equivalent Mutants:** These are closely related to live mutants, in that, they are 'alive' even after running test data through them. What differentiates them from others is that they have the same meaning as the original source code, even though they may have different syntax.

MUTANT CALCULATION SCORE (CHECK TESTCASES' EFFECTIVENESS)

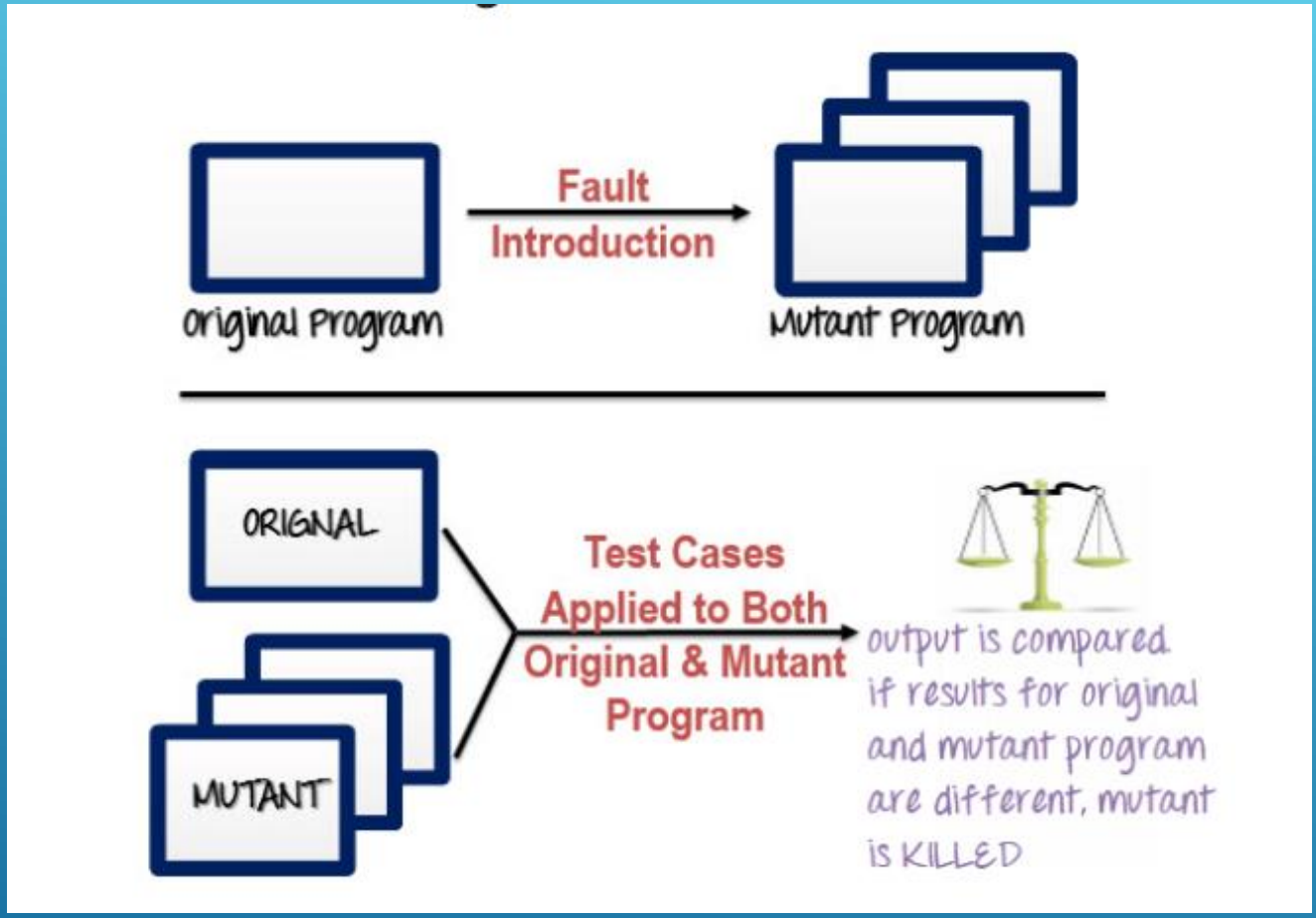
- This is a score based on the number of mutants.
- **It is calculated using the below formula:**

$$\text{Mutation score} = \frac{\text{Number of killed mutants}}{\text{Total number of mutants (survived and killed)}} * 100\%$$

Simply, Mutation Score = (Killed Mutants / Total number of Mutants) * 100

Note that, equivalent mutants are not considered when calculating the mutation score. Mutation score is also known as **mutation adequacy**. Our aim should be to achieve a high mutation score.

HOW TO EXECUTE MUTATION TESTING?



CREATION OF MUTANT PROGRAMS

Original Program	Mutant Program

If (x>y)
Print "Hello"
Else
Print "Hi"
	If(**x<y**)
	Print "Hello"
	Else
	Print "Hi"

BRINGING CHANGES IN PROGRAMS FOR MUTANTS

Operand replacement operators

- If(x>y) replace x and y values

If(5>y) replace x by constant 5

Expression Modification Operators

- If(x==y)

  We can replace == into >= and have mutant program as

  If(x>=y) and inserting ++ in the statement

  If(x==++y)

Statement modification Operators

- Delete the else part in an if-else statement

  Delete the entire if-else statement to check how a program behaves

MUTATION TESTING TYPES

- **Value Mutations:** An attempt to change the values to detect errors in the programs. We usually change one value to a much larger value or one value to a much smaller value. The most common strategy is to change the constants.
- **Decision Mutations:** The decisions/conditions are changed to check for the design errors. Typically, one changes the arithmetic operators to locate the defects and also we can consider mutating all relational operators and logical operators (AND, OR , NOT)
- **Statement Mutations:** Changes done to the statements by deleting or duplicating the line which might arise when a developer is copy pasting the code from somewhere else.

VALUE MUTATION

- Original Vs Mutant

```
let arr = [2,3,4,5]
for(let i=1; i<arr.length; i++){
    if(i%2===0){
        console.log(i*2)
    }
}
```

```
let arr = [2,3,4,5]
for(let i=0; i<arr.length; i++){
    if(i%2===0){
        console.log(i*2)
    }
}
```

DECISION MUTATION

	Original operator	Mutant operator
1	<=	>=
2	>=	==
3	===	==
4	and	or
5	||	&&

STATEMENT MUTATION

- Original Vs Mutant

```
let arr = [2,3,4,5]
for(let i=0; i<arr.length; i++){
    if(i%2===0){
        console.log(i*2)
        console.log(i*2)
    }
}
```

```
let arr = [2,3,4,5]
for(let i=0; i<arr.length; i++
    if(i%2===0){
        console.log(i*2)
    }
}
```

ADVANTAGES

Following benefits are experienced, if mutation testing is adopted:

- It brings a whole new kind of errors to the developer's attention.
- It is the most powerful method to detect hidden defects, which might be impossible to identify using the conventional testing techniques.
- Tools such as Insure++ help us to find defects in the code using the state-of-the-art. Other tools are Stryker and PIT (Parallel Isolated Test).
- Increased customer satisfaction index as the product would be less buggy.
- Debugging and Maintaining the product would be easier than ever.
- It is a powerful approach to attain high coverage of the source program.
- This testing is capable comprehensively testing the mutant program.
- Mutation testing brings a good level of error detection to the software developer.
- This method uncovers ambiguities in the source code and has the capacity to detect all the faults in the program.

DRAWBACKS (SHORTCOMINGS)

- Mutation testing is extremely costly since there are many mutant programs that need to be generated.
- It is not better to do without automation tool as it consumes a lot of time
- Each mutation will have the same number of test cases than that of the original program. So, many mutant programs may need to be tested against the original test suite.
- Not applicable for Blackbox

MANUAL TESTING VS. AUTOMATED TESTING

- A repetitive type of testing is very cumbersome and expensive to perform manually, but it can be automated easily using software tools
- A simple repetitive type of application can reveal memory leaks in a software. However, the application has to be run for a significantly long duration, say, for weeks, to reveal memory leaks. Therefore, manual testing may not be justified, whereas with automation it is easy to reveal memory leaks.
- Stress testing requires a worst-case load for an extended period of time, which is very difficult to realize by manual means.

MANUAL TEST CASE

- It is important to remember that test automation cannot replace manual testing
- Human creativity, variability, and observability cannot be mimicked through automation
- Certain categories of tests, such as usability, interoperability, robustness, and compatibility, are often not suited for automation
- It is too difficult to automate all the test cases; usually 50% of all the system-level test cases can be automated
- There will always be a need for some manual testing, even if all the system-level test cases are automated