Study Notes for Comp 4350: Software Engineering II - Lecture 9

Test Coverage

- **Test Coverage**: Quantitative measure of how much code is executed by test cases.
- **Effectiveness**: Test cases' ability to detect bugs.
- Coverage metrics do not directly measure test case effectiveness.

Fault Localization

- Identifying the exact location of a defect within the code.

Mutation Testing

- **Mutation Testing**: Introduce artificial defects by modifying the code to verify the ability of test cases to detect changes.
- Mutants: Variants of the program with small modifications (injected bugs) to assess test effectiveness.
- Steps in mutation testing:
  - **Step 1**: Create mutants by modifying the original program.
  - **Step 2**: Run test cases through each generated mutant.
  - **Step 3**: Compare results of the original program to the mutants.
  - **Step 4**: Determine if mutants are killed (detected by test) or alive (test suite ineffective).

Mutation Operators

- Govern the automated generation of mutants.
- Examples:
  - Statement modification (deletion/insertion/duplication).
  - Boolean and arithmetic expression replacement.
  - Variable replacement and constant manipulations.
  - Specific to OO languages: Changing access modifiers, overloading, overriding methods, etc.
- Mutation Operators selection depends on the programming language and design paradigms.

Mujava

- A mutation testing tool for class-level and method-level mutations.
- Mujava Reference

Mutation Testing Types

- **Statement Mutation**: Cut/paste errors leading to code modification.
- **Value Mutation**: Modification of primary parameter values.
- **Decision Mutation**: Changes in control statements.

Mutation Coverage

- **Mutation Score**: `MS = 100 * D / N` where D=Dead mutants, N=Total number of mutants.
- **Mutation Adequacy**: A test suite with a score of 100%.
- **Inadequate Test Set**: Enhance or augment the test suite if not all mutants are killed.

Live Mutants

- Indicative of inadequate test cases.
- **Equivalent Mutants**: Mutants functionally identical to the original program.

Avoiding Equivalent Mutants

- Use constraints and data dependencies to detect equivalent mutants and avoid generating them.

Mutation Testing Assumptions

- **Competent Programmer Assumption**: Experienced programmers introduce faults due to small syntactic errors.
- **Coupling Effect Assumption**: Small changes can lead to complex errors.

Mutant Variants

- **Equivalent Mutant**: Produces the same output as the original program.
- **Stillborn Mutant**: Killed by the compiler due to being syntactically incorrect.
- **Trivial Mutant**: Killed by almost any test case.

Triggering a Mutant

- **Reachability**: The test must reach the mutated code during execution.
- **Infection**: Mutated code must cause the program state to be incorrect.
- **Propagation**: Incorrect state must lead to incorrect program output.

Enhance Test Cases Through Mutation Testing

- The process involves examining failed test cases and enhancing the test suite to address mutants not previously killed.

Mutation Testing Summary

- Provides a measure for test case quality.
- Aims to kill mutants (find test cases that expose the mutant as a defect).
- Computationally intensive with potentially many mutants generated.
- Applicable to white-box testing but not black-box testing.

References

- TechTarget: Mutation Testing Definition
- Guru99: Mutation Testing

Class Notes

Slide 1

Title: Comp 4350 Software Engineering II Lecture 9
Dr. Shaowei Wang

Slide 2

Title: Outline
Applications for testing coverage
Faulty localization
Mutation testing

Slide 3

Title: An Example of Using Tarantula
Only running on 1/5 of the pass cases.
Running on all the pass cases.

Slide 4

Title: Outline
Applications for testing coverage
Faulty localization
Mutation testing

Slide 5

Mutation testing

Slide 6

Title: Measuring Test Cases
What we have learned so far?
Quantitatively measure how much code is covered by test cases?
Do the coverage metrics directly measure the effectiveness of test cases (where your test case could detect bug)?
No.

Slide 7

Title: Test Case Effectiveness
Detecting bug is the most effective way to measure test case.
Test suite A can detect more bugs than test suite B.
Given one bug, some test cases in suite A fail, but all test cases in suite B pass.
The best test suite can detect every bug; whenever a bug is introduced, at least one test case will fail.

Slide 8

What if we manually inject bugs and let the test cases to detect?

Slide 9

Title: Mutation testing
Artificially inject bugs by modifying (mutating) some statements in the code, so we have many different versions of the code. Each version has a bug.
Check if the test cases can find the bugs, i.e., some test cases fail and some do NOT fail.

Slide 10

```
int max(int x, int y){
    int Max;
    if (x > y)
        Max = x;
    else
        Max = y;
    return Max;
}
Input sets:
{x=3, y=2;}  => return 3 => cannot detect bug
{x=3, y=2; x=2, y=3;} => always return 2, detect the bug

int max(int x, int y){
    int Max;
    if (x > y)
        Max = x;
    else
        Max = x;   // mutant
    return Max;
}
```

Slide 11

Title: Steps in Mutation Testing
Input: an original program and a target test suits
Step 1: Modify statements in the original program and create a number of variates (i.e., mutants).
Step 2: Test cases are run through the each generated mutant.
Step 3: Compare the results of the original and the mutant.
Step 4:
=>The mutant is killed if different results are found (i.e., dead mutant). The test suite is good as it detects the change (i.e., injected defect).
=> The mutant is alive if the results are the same: the test suite is NOT effective.

Slide 12

For the automated generation of mutants, we use mutation operators, i.e., predefined program modification rules (corresponding to a fault model)

Slide 13

Title: Simple Example – mutation operations
(Gp:) _1 minVal = b;
(Gp:) _2 if (a < b);
(Gp:) _3 if (b < minVal);
(Gp:) _4 bomb();
(Gp:) _5 minVal = a;

Slide 14

Title: Specifying Mutation Operators
Ideally, we would like the mutation operators to be representative of (and generate) all realistic types of faults that could occur in practice
Mutation operators change with programming languages, design and specification paradigms (much overlap though)
In general, the number of mutation operators is large as they are supposed to capture all possible syntactic variations in a program

Slide 15

Title: Example of Mutation Operators
Statement deletion
Statement duplication
Statement insertion
Replacement of Boolean expressions with true or false
Replacement of arithmetic operations with others (e.g., + with *, - with /)
Replacement of conditional expressions (e.g., > with ≥, == with ≤)
Negation of conditional expressions
Replacement of variables (must be type compatible)
Replacement of constant with variable or vice versa
Replacement of array reference with constant or variable or vice versa
Replace increments with decrements or vice versa
Replace constructor calls with null
Replacement of return statement
For a complete list supported by the Pitest tool: see http://pitest.org/quickstart/mutators/

Slide 16

Title: Example of Mutation Operators
Specific to object-oriented programming languages:
Replacing a type with a compatible subtype (inheritance)
Changing the access modifier of an attribute or method
Changing the instance creation expression (inheritance)
Changing the order of parameters in the definition of a method
Changing the order of parameters in a call
Removing an overloading method
Reducing the number of parameters
Removing an overriding method

Slide 17

Title: Mujava
Class level mutation
Method level mutation
More information

Slide 18

Title: Type of mutation testing
Statement mutation
Value mutation
Decision mutation

Slide 19

Title: Mutation coverage
Mutation Score = 100 * D / N
D: Dead mutants (killed mutants)
N: Number of mutants
A set of test cases is mutation adequate if its mutation score is 100%.
Number of mutants tends to be large, even for small programs.
If not all mutants are killed, enhance your test suit.

Slide 20

Why some mutants still remain live?

Slide 21

Title: Why some mutants still remain live?
The test set is inadequate to kill the mutant (the test set does not cover the modified spots). For this case, the test data need to be re-examined, possibly augmented (increase test set) to kill the live mutant

Slide 22

```
int max(int x, int y){
    int Max;
    if (x > y)
        Max = x;
    else
        Max = y;
    return Max;
}
Input sets:
{x=3, y=2;}  => return 3 => cannot detect bug, because the mutated
statement is not executed.

int max(int x, int y){
    int Max;
```

```
    if (x > y)
        Max = x;
    else
        Max = x;   // mutant
    return Max;
}
```

Slide 23

Title: Why some mutants still remain live?
The test set is inadequate to kill the mutant (the test set does not cover the modified spots). For this case, the test data need to be re-examined, possibly augmented (increase test set) to kill the live mutant
It is equivalent to the original program (functionally identical though syntactically different – equivalent mutant)

Slide 24

Title: Example – Equivalent mutants
(int min(int a, int b) { ...
(Gp:) _2 if (b < minVal);
Replace a with minVal, does not change the semantic of the program since minVal = a;

Slide 25

Title: Mutation coverage
Mutation Score = 100 * D / (N - E)
D: Dead mutants (killed mutants)
N: Number of mutants
However, detecting equivalent mutants is challenging.
Use constraints to Detect Equivalent Mutants

Slide 26

Title: Example – Equivalent mutants
(int min(int a, int b) { ...
(Gp:) _2 if (b < minVal);
Maintain a constraints {minVal = a}

Slide 27

Title: Mutation coverage
Mutation Score = 100 * D / (N - E)
D: Dead mutants (killed mutants)
N: Number of mutants
E: Number of equivalent mutants
However, detecting equivalent mutants is challenging.
Use constraints to Detect Equivalent Mutants
Avoid generating equivalent mutants

Slide 28

Title: Example – Equivalent mutants
(int min(int a, int b) { ...
(Gp:) x = y-x;
Any mutant on the assignment of x before x=0 would cause equivalent mutant.
Use data dependency to know the scope of variable values can reach a particular program point

Slide 29

Title: Assumptions for mutation testing
What about more complex errors, involving several statements?
Competent programmer assumption: most software faults introduced by experienced programmers are due to small syntactic errors
Coupling effect assumption: simple changes can lead to complex errors

Slide 30

Title: Different Mutants
Equivalent mutant: always produces the same output as the original program
Stillborn mutant: syntactically incorrect, killed by compiler

Slide 31

Title: Example - Stillborn mutant
(int min(int a, int b) { ...
(Gp:) _6 minVal = ToString(b);
Return type of failOnZero() is String

Slide 32

Title: Different Mutants
Equivalent mutant: always produces the same output as the original program
Stillborn mutant: syntactically incorrect, killed by compiler
Trivial mutant: killed by almost any test case (i.e., covered by every test case)

Slide 33

Title: Example - Trivial mutant
(public int getMin (int [] numbers) { ...
(Gp:) Return 0;
A=[1,2,3,4,5] fail
B=[1,1,3,56,0] fail

Slide 34

Title: What is the condition for a test case to trigger a mutant?
Reachability: reach the fault seeded (mutated code) during execution
Infection: Cause the program state to be incorrect
Propagation: Cause the program output to be incorrect

Slide 35

```
int min(int A, int B)
{ // original
  int minVal;
  minVal = A;
  if (B < A) {
      minVal = B;
  }
  return minVal;
}
int min(int A, int B)
{ // mutant
  int minVal;
  minVal = B;
  if (B < A) {
      minVal = B;
  }
  return minVal;
}
Replace one variable with another
Reachability: unavoidable
Propagation: wrong minVal needs to return to the caller; that is, we
cannot execute the body of the if statement, so need B >= A
Infection: need B != A
Test case => B>A
```

Slide 36

Title: Example –use mutation testing to enhance test cases
The program prompts the user for a positive number in the range of 1 to 20 and then for a string of that length. The program then prompts for a character and returns the position in the string at which the character was first found or a message indicating that the character was not present in the string.
The user has the option to search for more characters.

```
FindPos(x, a, c){
    // a is the given string, x is the length of a, c is the target
character
    ...
    if (x > 20 || x<1)
        print(“input integer between 1 and 20”)
    found := FALSE;
    i := 1;
    x = sizeof(a)
    while(not(found) and (i _ x)) do
    begin
        if a[i] = c then
            found := TRUE
        else
```

```
            i := i + 1
    end
    ...
}
```

Slide 37

Title: Test Cases
x is the length of the string
a is the string
c is the character we are looking for
Response indicates whether we want to continue the search

Slide 38

Title: Mutant 1
Re-run original test data set
(Gp:) _1 found := TRUE;
Failure: “character a appears at position 1” != n
Mutant 1 is killed.

Slide 39

Title: Mutant 2
Running our original test data set fails to reveal the fault
Mutant is not killed _ test cases are not adequate
Need to increase our string length and search for a character further along it.
(Gp:) _2 x := 1;

Slide 40

Title: Mutant 2
Running our original test data set fails to reveal the fault
(Gp:) _2 x := 1;
Failure: “Character v does not occur in string”
Mutant is killed _ test cases are adequate

Slide 41

Title: Mutant 3
Again, our test data fails to kill the mutant
Need to augment the test data to search for a character in the middle of the string.
(Gp:) _3 i := i + 2;

Slide 42

(Gp:) _3 i := i + 2;

Slide 43

Title: Mutation Testing: summary

Measures the quality of test cases

Provides the tester with a clear target (mutants to kill)

Computationally intensive, a possibly very large number of mutants is generated

It is not applicable for black-box testing

Slide 44

Title: References

1

2