CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Lecture 11: Mutation Testing

Gregory Gay
DIT636/DAT560 - February 19, 2024

Space Shuttle Challenger

- Seal failure in rocket booster causes explosion, killing seven astronauts.
- Investigation found technical and organizational issues.
  - Became a case example studied in many forms of engineering.
  - **Learn from your failures.**

Fault-Based Testing

- By studying faults in previous designs, we can prevent similar faults in new designs.
- Many testing techniques based on what we ***think should happen***.
- We can also design tests based on knowledge of ***what has gone wrong in other programs***.

Implemented in Language Design

- Automated Garbage Collection
  - Prevents dangling pointers, memory leaks, other memory management faults.
- Automatic Array Bounds Checking
  - Does not prevent bad indexes from being used, but ensures they are noticed and limits damage.
- Type Checking
  - Prevent malformed value use in input or computations.

Fault-Based Testing

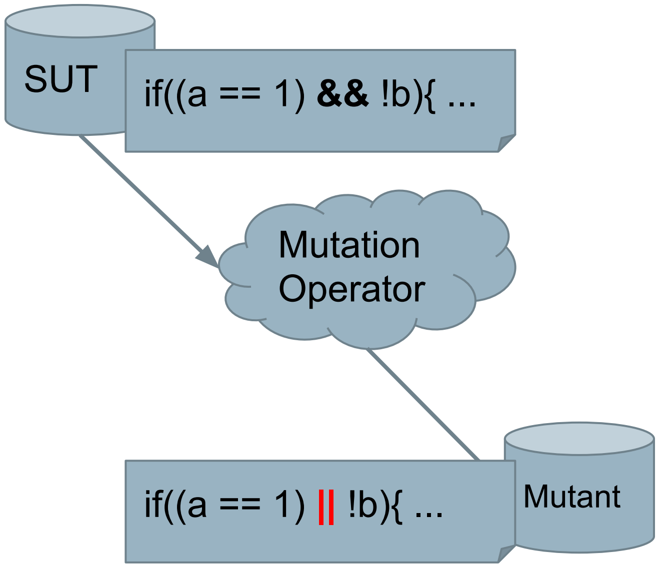
- Consider the types of faults we expect to see.
  - Create **mutated** versions of the program.
  - See if tests fail for those mutated versions.
- **Fault Seeding**
  - Deliberately creating programs with faults to see if our tests are good enough to detect them.
  - May help us find new faults in the unmutated program.

Uses of Fault Seeding

- Fault seeding can be used to:
  - Judge the adequacy of a test suite.
  - Design test cases to augment a suite.
- Provides evidence that we have done a good job.
  - If our tests have not found faults, are there no more major issues, or are they bad tests?

Mutation Testing

- Encode common faults as **mutation operators**.
  - Insert the modeled fault into program statements.
- Produces a **mutant**.
  - A clone of the program with 1+ seeded faults.



Mutants

- **“First-Order Mutants” (our focus)**
  - One line modified.
  - Easy to create, many tools to insert them.
  - Most common, but not as realistic.
- “Higher-Order Mutants”
  - Multiple lines modified.
  - Harder to create, not well understood.
  - May be more realistic.

Mutation Operators

Mutation Operators

- Intended to model common types of faults.
- Designed to be applied to any type of code, without human intervention.
- Tend to be simple syntactic faults.
  - Replacing one variable reference with another.
  - Changing a comparison from < to <=.
  - Referencing a parent class instead of a child.

Operand Modifications

- X for Y replacement
  - Replace constant *C1* with constant *C2*.
    - `int X = 72;` -> `int X = 135;`
  - Replace constant *C* with variable *S*.
    - `int Y = 135; int X = 72;` -> int Y = 135; int X = **Y**;
  - Replace variable *S* for constant *C*.
    - `int X = Y;` -> `int X = 72;`
  - Replace variable *S1* with variable *S2*.
    - `int X = Y;` -> `int X = Z;`

Operand Modifications

- X for Y replacement
  - Replace variable/constant with array reference *A[I]*.
    - `int X = Y; -> int X = A[4];`
  - Replace array reference *A[I]* with variable/constant.
    - `int X = A[4]; -> int X = Y;`
  - Replace array reference with another array reference.
    - Either another array or another index in the same array.
    - `int X = A[4]; -> int X = A[10];`

Expression Modifications

- Arithmetic Operators
  - Binary operators: `x (+, -, *, /, %) y`
  - Unary operators: `+x, -x`
  - Shortcut operators: `x++, ++x, x--, --x`
- Arithmetic Operator Replacement
  - Replace binary/unary/shortcut operator with another.
    - `Z = X + Y;` -> `Z = X - Y;`
  - Replace shortcut/unary operator with a unary/shortcut.
    - `Z = --X;` -> `Z = -X;`

Expression Modifications

- Arithmetic Operator Insertion
  - Insert an additional operator into an expression.
  - `int Z = X; -> int Z = X + Y;`
  - `int Z = X; -> int Z = X++;`
- Arithmetic Operator Deletion
  - Remove an operator from an expression.
  - `int Z = X + Y; -> int Z = X;`
  - `int Z = X++; -> int Z = X;`

Expression Modifications

- Conditional Operators
  - Binary: `x (&&, ||, &, |, ^) y`
  - Unary: *`(~, !)x`*
- Relational Operators
  - `x (>, >=, <, <=, ==, !=) y`
- Shift Operators
  - `x (>>, <<, >>>>) y`
- Operator Replacement, Insertion, Deletion
  - Works like arithmetic operators.

Expression Modifications

- Shortcut Operators
  - `x (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=) y`
  - Shortcut Operator Replacement
- Absolute Value Insertion
  - Replace a subexpression with *abs(e)*.
    - `int Z = X + Y; -> int Z = abs(X + Y);`
- Constant for Predicate Replacement
  - Replace boolean predicate with a constant value *(T/F)*.
    - `bool Z = (A || B) && C; -> bool Z = (A || true) && C;`

Statement Modifications

- Statement Deletion
  - Remove a random statement from the program.
- Switch Case Replacement
  - Replace the label of one case with another.
- End Block Shift
  - Move closing brackets to an earlier or later location.

Encapsulation/Inheritance

- Access Modifier Change
  - Change a modifier to *(public/protected/private)*
  - **public** void DoThis(int x) -> **private** void DoThis(int x)

Inheritance Modifications

- Overriding Method Deletion
  - Delete an overriden method from a subclass.
  - References call the version inherited from a parent.

  - ```
    Class Child implements Parent { …
    @Override public int doThis(){ .. } …
     int X = doThis(); }
    ->
    Class Child implements Parent { …
    int X = doThis(); }
    ```
    ```

Inheritance Modifications

- Super Keyword Insertion/Deletion
  - Inserts or deletes the `super()` keyword.

  - ```
    @Override
    public void doSomething(){
        super(); … } ->
    @Override
    public void doSomething(){
        … }
    ```
    ```

Inheritance Modifications

- Overridden Method Calling Position Change
  - Overridden methods can call the parent method.
  - Moves calls to the parent version to other positions.

  - ```
    @Override
    public int doThis(){
        int x = super(); int y = 5; ...  }    ->
    @Override
    public int doThis(){
        int y = 5; ... int x = super();  }
    ```
    ```

Inheritance Modifications

- Overridden Method Renamed
  - Rename a method in the parent class that was overridden by the child.
  - Ensures that the overridden version is always called instead of the parent version.
  - `Class Parent { … public void doThis(); } Class Child implements Parent { … @Override public void doThis(); }`
    `->`
    `Class Parent { … public void doThat(); } Class Child implements Parent { … public void doThis(); }`

Inheritance Modifications

- Explicit Parent Constructor Call Deletion
  - Deletes *super()* call in a constructor.
  - To detect, tests must trigger an incorrect initial state.

  - Class Child implements Parent {
    int x;
    public Child () { **super();** ... } } ->

    Class Child implements Parent {
    int x;
    public Child () { ... } }

Polymorphism Modifications

- New Method Call with Child Class Type
  - Replace a declaration with a valid child instance.
    - `Parent a = new Parent();` -> `Parent a = new Child();`
- Variable Declaration With Parent Class Type
  - Change the declared type of a variable to its parent.
    - `Child a = new Child();` -> `Parent a = new Child();`
    - `boolean equals(Child c){..}` -> `boolean equals(Parent c){..}`

Polymorphism Modifications

- Type Cast Operator Insertion/Deletion
  - Cast the type of an object reference to the parent or child of the original type.
    - `p.toString() -> ((Child) p).toString()`
  - Or delete a type cast operator.
    - `((Child) p).toString()-> p.toString()`

- Cast Type Change
  - Changes a cast to another valid data type.
  - `((SomeChild) c).toString() -> ((OtherChild) c).toString()`

Polymorphism Modifications

- Reference Assignment with Other Compatible Type
  - Change an object reference to point to another compatible variable.

-

```
Object obj;
String s = “hello”;
Integer i = new Integer(4);
obj=s;
```

->

```
Object obj;
String s = “hello”;
Integer i = new Integer(4);
obj=i;
```

Polymorphism Modifications

- Overloading allows 2+ methods to have the same name if they have different signatures.
- Overloading Method Contents Change
  - Replace the body of a method with the body of another method with the same name.

  - ```
    public void doThis (int x) { … int Z … }
    public void doThis (int x, int y) { … int W … }     ->
    public void doThis (int x) { … int W … }
    public void doThis (int x, int y) { … int Z … }
    ```
    ```

Polymorphism Modifications

- Overloading Method Deletion
  - Deletes one of the overloading methods.
  - ```
    public void doThis (int x) { … }
    public void doThis (int x, int y) { … }     ->
    public void doThis (int x) { … }
    ```
- Argument of Overloading Method Change
  - Changes order or number of arguments in an invocation, as long as there is a version that will accept the list.
  - ```
    public void doThis (int x, int y) { … }     ->
    public void doThis (int y, int x) { … }
    ```

Language-Specific Modifications

- Mutation operators can be written for a particular language.
- Java:
  - *this* insertion/deletion
  - Static modifier insertion/deletion
  - Member variable initialization deletion
  - Default constructor deletion
  - Getter/Setter method replacement

Let’s Take a Break

Mutation Testing

Mutation Testing

- Select mutation operators.
- Generate mutants by applying mutation operators.
- Execute tests against original class and mutants.
  - A mutant is **killed** if the test passes on the original program and fails on the mutant.
  - A mutant not killed is considered **live**.

Mutation Testing

- Mutation operators reflect small syntactic mistakes.
  - **Programmers do make such mistakes!**
- However, many faults are *conceptual* mistakes.
  - Mistaken assumptions about requirements.
  - Forgotten requirements.
- **Is mutation testing a reasonable technique for judging test adequacy?**

Viability of Mutation Testing

- Mutation testing is valid if seeded faults are **representative** of real faults.
- *Competent Programmer Hypothesis*
  - A faulty program differs from a correct program only by small textual changes.
  - If so, we only have to distinguish the program from all such small variants.
  - Assumption: the SUT is “close to” correct.

Coupling Effect

- Many faults **are** small syntactical errors.
- Conceptual faults often manifest as syntax errors.
- Complex faults result in larger textual differences.
  - However, mutation testing is still valid **if** test cases for simple issues can detect complex issues.
  - *Coupling Effect Hypothesis* - complex faults can be modeled as a set of small faults.

Coupling Effect

- A complex change is a series of small changes.
  - If one change not covered up by others, a test that exposes it can also detect a more complex change.
- Mutation testing effective if **both** competent programmer and coupling effect hypotheses hold.

Judging Test Sensitivity

- Mutants are often simpler than real faults.
- Mutation is still good at judging **sensitivity of your tests to minor changes in the code**.
  - If tests can distinguish mutants from the real code, then your tests execute the code *thoroughly*.
  - If you miss mutants, you can add new tests to detect them and make your suite more sensitive.

Mutant Quality

To be used in testing, mutants must be:

- Syntactically correct (***valid***)
  - Mutants must compile and execute.
- Plausible (***useful***)
  - Must provide valuable information on how the system works for testers working to improve the system.
- **A mutant can be valid, but not useful.**
  - All or almost all tests fail.

Mutant Quality

Mutants might remain live if:

- They are *equivalent* to the original program.
  - `for(i=0; i < 10; i++) ->`
  - `for(i=0; i != 10; i++)`
  - Identifying equivalency is NP-hard.
- Test suite is *inadequate* for that mutation.
  - `(a <= b)` and `(a >= b)` cannot be differentiated if `a==b` in the test case.

Mutation Coverage

Adequacy of suite can be measured as:

$$\frac{(\#\ \text{mutants killed})}{(\text{total mutants})}$$

- Helps ensure that the test suite is *robust* against the modeled mutation types.
  - Ensures that suite is sensitive to small changes in code.

Mutation and Structural Coverage

Can subsume structural coverage.

- Statement Coverage
  - Apply statement deletion to each statement.
- Branch Coverage
  - Apply constant replacement to each predicate.
    - (set to true/false)
  - To kill a “true” mutant, a test must execute the original with a false value.

Practical Considerations

Mutation testing is **expensive**.

- Must run *all* tests against *all* mutants.
- Many mutants typically generated.
  - One mutation operator applied per mutant.
  - May be dozens - hundreds per class.
- Can randomly choose X mutants or operators.

Statistical Mutation Testing

- A test suite that kills *some* mutants may be as effective as one that kills *all* mutants.
- Obtain a statistical estimate of the ability of the suite to detect mutations.
  - Randomly generate N mutants.
  - Samples must be a valid statistical model of occurrence frequencies of real faults.
  - Target 100% coverage over the sample.

Mutation Testing at Google

- Very large codebase, so using all mutants or using mutants often impractical.
  - Skip lines not covered by tests.
  - Skip "uninteresting" lines.
    - Logging, testing, timing, loop conditions.
- Used during code reviews.
  - Present undetected mutants to suggest new tests or potential code mistakes.

Activity

1. How many mutations are possible for Relational Operator Replacement, Arithmetic Operator Replacement
2. Apply relational operator replacement operation to line 4, choose input that will show different output from original.
3. Design an equivalent mutant.
4. Design a valid, but not useful mutant.

```
public int[] makePositive(int[] a){
    int threshold = 0;
    for(int i=0; i < a.length; i++){
        if(a[i] < threshold){
            a[i]= -a[i];
        }
    }
    return a;
}
```

Activity - Solution

- How many mutations are possible:
  - Relational Operator Replacement:
    - `for(int i=0; i < a.length; i++){`
      - (>=, >, <=, ==, !=), 5 mutations
    - `if(a[i] < threshold){`
      - (>, >=, <=, ==, !=), 5 mutations

Activity - Solution

- How many mutations are possible:
  - Arithmetic Operator Replacement
    - `for(int i=0; i < a.length; i++){`
      - Shortcut replacement, (++*i*, *i*--, --*i*), 3 mutations
    - `a[i]= -a[i];`
      - Unary replacement, (+a[i]), 1 mutation
      - Unary to shortcut replacement, *(a[i]*++, ++*a[i]*, *a[i]*--, --*a[i]*), 4 mutations

Activity - Solution

- Apply the relational operator replacement operation to statement 4:
  - `if(a[i] < threshold){    ->`
  - `if(a[i] == threshold){`
- Choose test input that would kill that mutant.
  - a[-1,0,1]
  - -1 would not become positive.

Activity - Solution

- **Design an equivalent mutant.**
  - Can do so by applying the relational operator replacement operation to statement 4:
    - `if(a[i] < threshold){` becomes:
    - `if(a[i] <= threshold){`
  - Since threshold=0, and -0 = 0, no test would detect.
  - Does not help us test, as the fault cannot cause a failure.

Activity - Solution

- **Design a valid, but not useful mutant.**
  - Compiles, but trivially fails.
  - Apply relational operator replacement to statement 4:
    - `if(a[i] < threshold){` becomes:
    - `if(a[i] > threshold){`
    - Any positive numbers are made negative, all negative remain negative. Almost any test would detect this.
  - **Many** mutants are not useful.

PIT Demo

We Have Learned

- Mutation testing inserts faults to judge test suite sensitivity and adequacy.
- Mutation operators automatically create faulty versions of a program.
  - Operators model expected syntactic faults.
- Tests are judged according to their ability to detect faults - useful sensitivity analysis.

Next Time

- Testing in Industry
  - **Please attend!**
- Exercise Session: Mutation Testing
- Assignment 2 due February 25.

UNIVERSITY OF GOTHENBURG

CHALMERS

UNIVERSITY OF TECHNOLOGY