

MUTATION TESTING

- White-box, error-based testing technique
- Built-in adequacy criteria
- Tool support
- The goal of mutation testing is to build an adequate sets of tests
 - Finding faults is a side-effect

ERROR-BASED TESTING

- Devise tests that mimic the kinds of mistakes that people make when constructing programs
- *Off-by-one*, spelling, typos, etc.

MUTANTS

- A *mutant* is a program (P') that is similar to the program that you are testing (P)
 - It differs by a single (usually lexical) *mutation*
- For example, imagine that P' was identical to P except that exactly one $+$ was changed to a $*$
 - That is, the programmer had made a typo

RUNNING A MUTATION TEST

- If you run the same test case (T) on both P and P' , one of two things can happen:
 1. P and P' produce different results
 2. They produce identical results

1. DIFFERENT RESULTS

- If P and P' produce different results on T , then they both can't be right
 - Determine which version is correct
 - If P' is correct, then you have detected a bug in P
 - Fix the bug and restart testing
 - If P is correct, then P' must have a bug
 - The only difference is in the mutation we introduced
 - That is, the test has worked; it has found the bug we introduced
 - If this is the case, we say that mutant P' has been *killed* by the test case T
 - Proceed with the next mutant on P

2. SAME RESULTS

- If P and P' produce the same results, this also might occur for two reasons:
 1. The test case T wasn't good enough to demonstrate the difference
 - Perhaps it didn't even execute the line of code containing the mutation
 - In this case we need to come up with another test
 - That is, we keep adding tests until we can kill the mutant
 2. P and P' are *equivalent* programs, and no tests can be devised that can distinguish them

EQUIVALENT MUTANTS

- Let's say that the line of code we mutated was the following

```
X = Y + Z;
```

which we changed to

```
X = Y * Z;
```

- If this line occurs directly following the statement

```
if (Y == 2 && Z == 2)
```

```
    X = Y + Z;
```

then no possible test can ever kill this mutant

- That is, within the body of the `if`, both version produce the value 4 for X

MOTHRA

- Mothra is a suite of tools (developed at Georgia Tech) for performing mutation testing
 - Mutgen: generates mutants
 - A testing harness for runs tests on a set of mutants and records the results
 - Godzilla: generates test cases

USING MOTHRA

- Select and generate a set of mutants
- Compose an initial set of test cases and the corresponding outputs that they generate
- Confirm that the outputs are correct
- Repeat until all mutants are killed:
 - Kill mutants by running the test cases
 - Detect equivalent mutants (effectively killing them)
 - Generate new tests and determine expected results
- When you are done, you have an adequate suite of tests (for the set of errors for which mutants were generated)

PROBLEMS WITH MUTATION TESTING

- Performance: a large number of mutants are generated
 - Test case execution is simulated
- Determining expected results (*oracle*)
 - This is a problem with all kinds of testing
- Generating test cases
 - This is a problem with all kinds of testing
- Equivalencing mutants
 - Forces a deep understanding of the program, leading to a good set of tests

ASSUMPTIONS

- *Competent Programmer Hypothesis*
 - The program being tested is “nearly” correct
 - It attempt to implement a valid specification
- *The Coupling Effect*
 - Large faults, particularly those of a semantic nature, are “coupled” with smaller “lexical / syntactic” faults that can be detected with mutation testing

FORTRAN

- Comments begin with `C`
- Relational operators: `.EQ.`, `.LE.`, etc.
- Logical operators: `.OR.`, etc.
- Numerically labeled statements and `GOTOS`
- Value returned by a `FUNCTION` is kept in a pseudo-variable with the function name
- `PARAMETER` is used for constants
- On I/O, `*` denotes standard input/output
- Fixed loop:
 - `DO stmt #, var = first [, last, incr]`

TRITYPE

- Categorizes triangles:
 - 1 \Rightarrow scalene
 - 2 \Rightarrow isosceles
 - 3 \Rightarrow equilateral
 - 4 \Rightarrow not a triangle
- Statistics (no loops)
 - 28 statements
 - 28 branches
 - 30 conditions
 - 89 paths

CATALOG OF MUTANTS

- AAR - array reference by array reference
- ABS - expression by abs(expression)
- NEGABS - expression by -abs(expression)
- ZPUSH - IF (expression .EQ. 0) then killed
 - Generate a test that forces the expression to have value 0
- ACR - array reference by constant
- AOR - arithmetic operator replacement
- LEFTOP - binary expression by left operand

CATALOG - 2

- RIGHTOP- binary expression by right operand
- MOD - binary expression by mod(left, right)
- ASR - array reference by scalar variable
- CAR - array reference by constant
- CNR - array name by comparable array name
- CRP - constant by constant
- CSR - scalar variable by constant

CATALOG - 3

- DER - DO statement end label replacement
- ONETRIP - DO statement by non-null DO
- DSA - DATA statement constant replacement
- GLR - GOTO label replacement
- LCR - logical operator replacement
- FALSEOP - logical expression by .FALSE.
- TRUEOP - logical expression by .TRUE.
- ROR - relational operator substitution

CATALOG - 4

- RSR - statement by RETURN
- SAN - statement by TRAP
 - Statement coverage
- SAR - scalar variable by array reference
- SCR - constant replaced by scalar variable
- SDL - statement replace by CONTINUE
- SCR - arithmetic constant substitution
- SVR - scalar variable substitution
- UOI - unary operator insertion