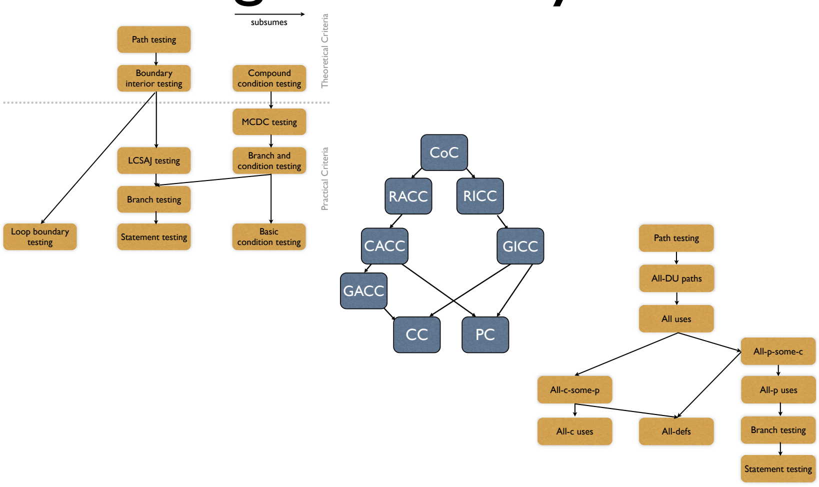
1

How good are my tests?



2

The most common way to determine the quality of a test suite is to measure its coverage - we've already seen quite a number of different coverage criteria.

```
/**
 * Make sure Double.NaN is returned iff n = 0
 *
 */
public void testNaN() {
    StandardDeviation std = new StandardDeviation();
    assertTrue(Double.isNaN(std.getResult()));
    std.increment(1d);
    assertEquals(0d, std.getResult(), 0);
}
```

3

This is a unit test case for the StandardDeviation class in Common-Math.

```java
/**
 * Make sure Double.NaN is returned iff n = 0
 *
 */
public void testNaN() {
    StandardDeviation std = new StandardDeviation();
    Double.isNaN(std.getResult());
    std.increment(1d);
    std.getResult();
}
```

Coverage is not changed!

4

This version of the unit test has the identical coverage as the test on the previous slide - but it will detect no faults (except program crashes).

The Oracle Problem

- Executing all the code is not enough
- We need to check the functional behavior
- Does this thing actually do what we want?
- Automated oracles can be spec, model
- Else, manual oracles have to be defined

5

Coverage misses one important aspect: The Oracle Problem. A test oracle is the entity that decides whether a test case passed or failed.

How good are my tests?

- Coverage = how much of the code is executed
- But how much of the code is checked?
- We don't know where the bugs are
- But we know the bugs we have made in the past!

6

Learning from Mistakes

- Key idea: Learning from earlier mistakes to prevent them from happening again
- Key technique: *Simulate earlier mistakes* and see whether the resulting defects are found
- Known as *fault-based testing* or *mutation testing*

7

```
int do_something(int x, int y)
{
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int y)
{
  if(x < y)
    return x-y;
  else
    return x*y;
}
```

Mutant

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test

8

We have a program under test, and test cases that exercise the program. The program passes all our tests - so how good is the program tested? We insert a simple fault in the program, and create a mutant version. On this mutant we execute the same test cases again. If the mutant fails the test cases then we see that our test case checks against this type of fault at this location. If the mutant passes the test we need more tests.

Mutants

- Mutant
  Slightly changed version of original program
- Syntactic change
  Valid (compilable code)
- Simple
  Programming "glitch"
- Based on faults
  Fault hierarchy

9

Generating Mutants

- Mutation operator
  Rule to derive mutants from a program
- Mutations based on real faults
  Mutation operators represent typical errors
- Dedicated mutation operators have been defined for most languages
- For example, > 100 operators for C language

10

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

11

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = abs(y);
    y = tmp;
  }

  return x;
}
```

12

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return -abs(x);
}
```


ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return 0;
}
```


AOR - Arithmetic Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

AOR - Arithmetic Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x * y;
    x = y;
    y = tmp;
  }

  return x;
}
```

+, -, *, /, %, **, x, y

16

AOR does not only replace the operators, but also the two special cases where x + y is mutated to "x" and to "y", i.e. dropping the other operand.

ROR - Relational Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

17

ROR - Relational Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y > 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

<, >, <=, >=, =,
!=, false, true

18

In addition to replacing the relational operator with other operators, this mutation operator also replaces the entire relational expression with true and with false.

COR - Conditional Operator Replacement

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
if(a ^ b)
if(false)
if(true)
if(a)
if(b)
```


SOR - Shift Operator Replacement

```
x = m << a
```

```
x = m >> a
x = m >>> a
x = m
```


LOR - Logical Operator Replacement

```
x = m & n
```

```
x = m | n
x = m ^ n
x = m
x = n
```

ASR - Assignment Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

22

ASR - Assignment Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x -= y;
    y = tmp;
  }

  return x;
}
```

+=, -=, *=, /=, %=, &=, |=, ^=,<<=,>>=,>>>=

23

UOI - Unary Operator Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

24

UOI - Unary Operator Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % +y;
    x = y;
    y = tmp;
  }

  return x;
}
```

+, -, !, ~,++,--

25

UOD - Unary Operator Deletion

```
if !(a > -b)
```

```
if (a > -b)
if !(a > b)
```

26

SVR - Scalar Variable Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

27

SVR - Scalar Variable Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = y % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

tmp = x % y
tmp = x % x
tmp = y % y
x = x % y
y = y % x
tmp = tmp % y
tmp = x % tmp

28

OO Mutation

- So far, operators only considered method bodies
- Class level elements can be mutated as well:

```
public class test {
  // ..
  protected void do() {
    // ...
  }
}
```

```
public class test {
  // ..
  private void do() {
    // ...
  }
}
```

29

OO Mutation

- AMC - Access Modifier Change
- HVD - Hiding Variable Deletion
- HVI - Hiding Variable Insertion
- OMD - Overriding Method Deletion
- OMM - Overridden Method Moving
- OMR - Overridden Method Rename
- SKR - Super Keyword Deletion
- PCD - Parent Constructor Deletion
- ATC - Actual Type Change
- DTC - Declared Type Change
- PTC - Parameter Type Change
- RTC - Reference Type Change
- OMC - Overloading Method Change
- OMD - Overloading Method Deletion
- AOC - Argument Order Change
- ANC - Argument Number Change
- TKD - this Keyword Deletion
- SMV - Static Modifier Change
- VID - Variable Initialization Deletion
- DCD - Default Constructor 2

30

Interface Mutation

- Integration testing
- Change calling method by modifying the values that are sent to a called method
- Change a calling method by modifying the call
- Change a called method by modifying the values that enter/leave the method
- Change a called method by modifying statements that return from the method

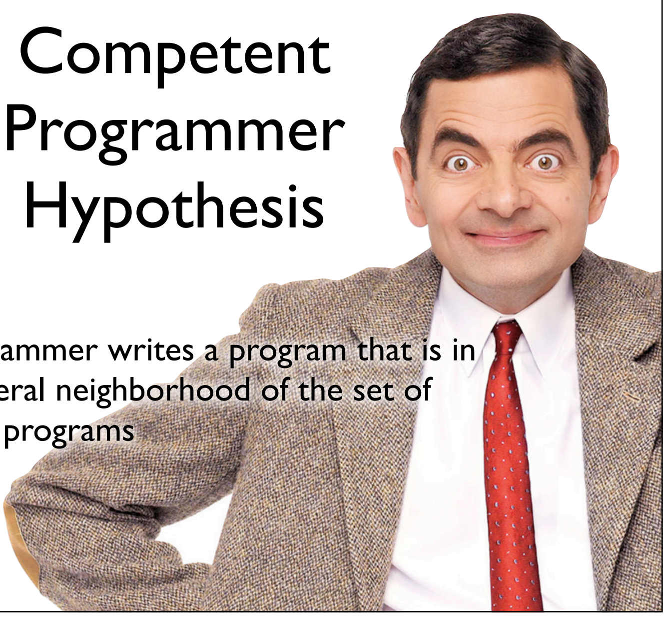
31

Order of Mutants

- First order mutant (FOM)
  Exactly one mutation
- Each mutation operator yields a set of FOMs
- Number of FOMs
  ~ number of data references * number of data objects
- Higher order mutant (HOM)
  Mutant of mutant
- $\#HOM = 2^{\#FOM} - 1$

32

Competent Programmer Hypothesis



A programmer writes a program that is in the general neighborhood of the set of correct programs

33

Coupling Effect

Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.

34

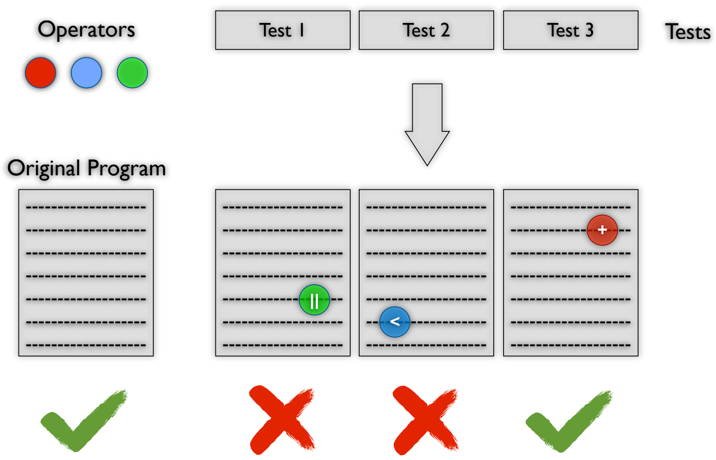
Mutation testing focuses on First Order Mutants

Competent Programmer Hypothesis

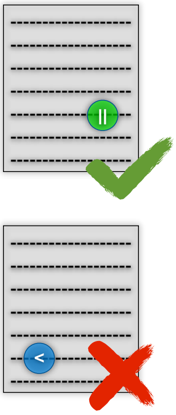
Coupling Effect

35

Because of the competent programmer hypothesis and coupling effect mutation testing in general only considers first order mutants.

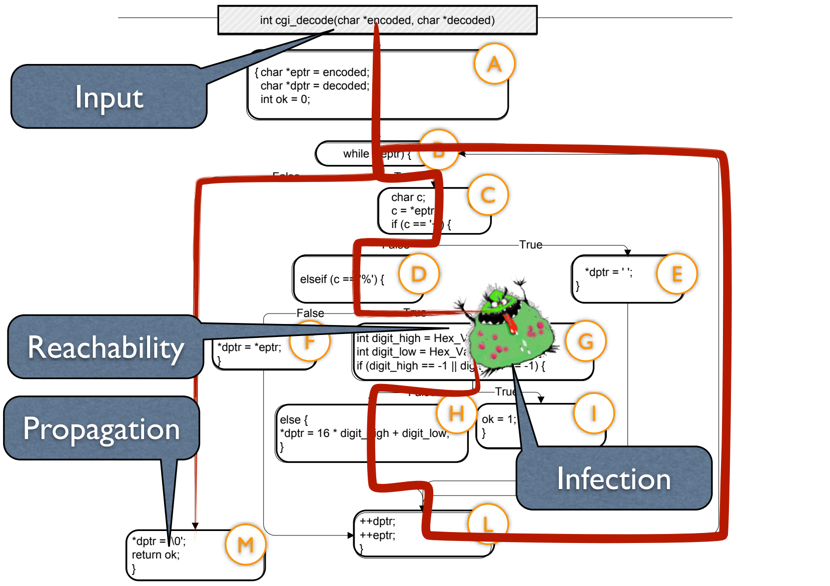


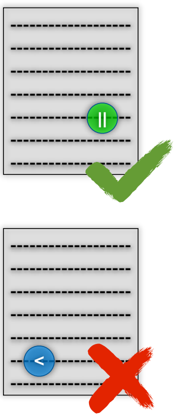
36



Live mutant - we need more tests

Dead mutant - of no further use





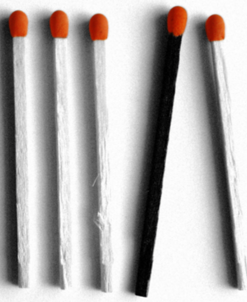
Mutation Score:

$$\frac{\text{Killed Mutants}}{\text{Total Mutants}}$$

- **Mutation analysis**: Assessing the quality of a test suite
- **Mutation testing**: Improving the test suite using mutants

40

Equivalent Mutants



- Mutation = syntactic change
- The change might leave the semantics unchanged
- Equivalent mutants are hard to detect (undecidable problem)
- Might be reached, but no infection
- Might infect, but no propagation

41

Example I

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[i] > values[r])
      r = i;
  }

  return values[r];
}
```

42

Example I

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 0; i<values.length; i++) {
    if (values[i] > values[r])
      r = i;
  }

  return values[r];
}
```

43

In this mutant, the loop starts from 0 instead of 1. This mutant is equivalent because it only introduces an additional comparison of the first element to itself - this cannot change the functional behavior.

Example I

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[i] >= values[r])
      r = i;
  }

  return values[r];
}
```

44

This is another equivalent mutant: The value of the maximum stays the same regardless of whether the comparison is < or <=

Example I

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[r] > values[r])
      r = i;
  }

  return values[r];
}
```

45

This mutant is not equivalent.

Example 2

```
if(x > 0) {
  if(y > x) {
    // ...
  }
}
```

46

Example 2

```
if(x > 0) {
  if(y > abs(x)) {
    // ...
  }
}
```

47

In the second predicate x can only have values greater than 0 because of the first predicate, so this mutant is equivalent.

Frankl's Observation

> We also observed that […] mutation testing was *costly*.
> Even for these small subject programs, the human effort needed to check a large number of mutants for equivalence was *almost prohibitive*.

P. G. Frankl, S. N. Weiss, and C. Hu.
All-uses versus mutation testing:
An experimental comparison of effectiveness.
Journal of Systems and Software, 38:235–253, 1997.

48

Compiler Optimizations

- Some mutations are similar to compiler optimizations / de-optimizations
- Optimizations are functionally equivalent
- Use compiler optimization techniques to remove equivalent mutants
- Examples
  Mutants in dead code • Mutation of def without use
- Works for ~10% of equivalent mutants

49

Mutant Constraints

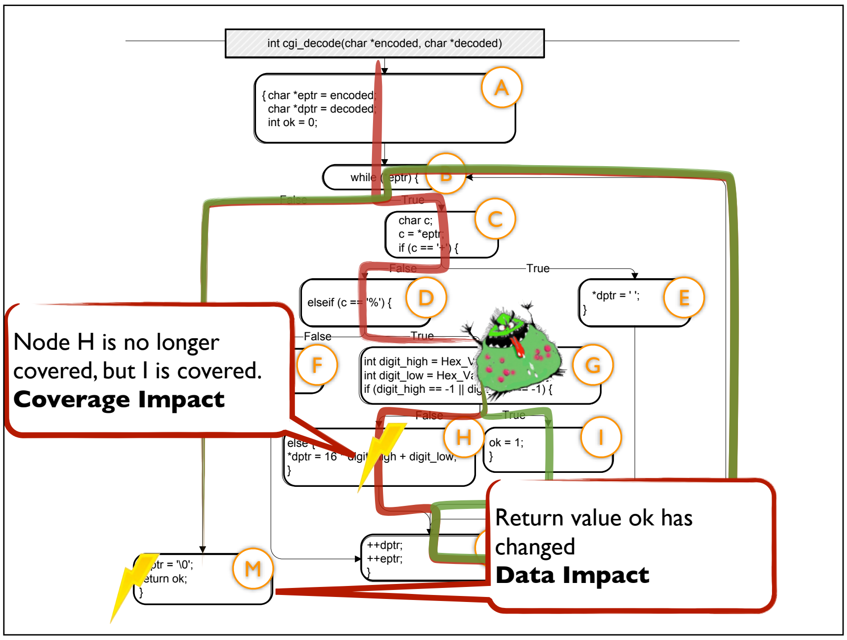
- State infection can be represented as a constraint system
- Equivalent mutant problem = feasible path problem
- Heuristics can detect some infeasible path constraints
- Works for ~40% of equivalent mutants

50

Impact of mutations

- A mutant is killed if an oracle checks one of the places it propagates to
- If a mutant propagates to many places, chances of detecting it are higher
- Impact = measurement of how much/far a mutant propagates
- High impact but not detected: Check your oracles...

51

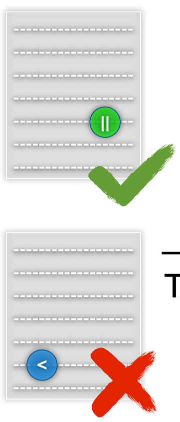


52

Impact

- Measurement
  Number of methods with changed coverage
  Number of methods with changed return values
  Number of violated dynamic invariants
- Mutants with high impact are less likely to be equivalent
- Prioritize mutants according to impact

53



Mutation Score:

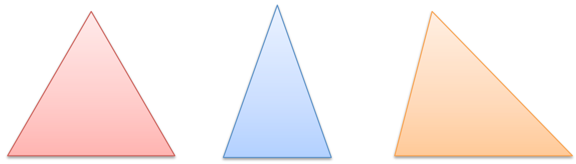
$$\frac{\text{Killed Mutants}}{\text{Total Mutants - Equivalent mutants}}$$

54

Now that we are aware of the equivalent mutant problem we can refine the definition of the mutation score slightly: We only want to kill the non-equivalent mutants, else reaching 100% mutation score would be impossible.

Example

Classify triangle by the length of the sides



55

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

56

This is an example implementation of the triangle example. If one of the triangle sides is negative or the inputs don't satisfy the triangle invariant, then we return invalid (4). If they're equilateral we return 1, 2 if two sides are isosceles, and 3 if the triangle is scalene. We start off by creating a branch coverage test suite for the program.

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a - b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✗

(2, 2, 3) ✗

(2, 3, 4) ✗

57

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

- (0, 0, 0) ✓
- (1, 1, 3) ✓
- (2, 2, 2) ✓
- (2, 2, 3) ✓
- (2, 3, 4) ✓

58

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b || b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

- (0, 0, 0) ✓
- (1, 1, 3) ✓
- (2, 2, 2) ✓
- (2, 2, 3) ✗
- (2, 3, 4) ✓

59

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || !(a == c)) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

- (0, 0, 0) ✓
- (1, 1, 3) ✓
- (2, 2, 2) ✓
- (2, 2, 3) ✓
- (2, 3, 4) ✗

60

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓
(1, 1, 3) ✓
(2, 2, 2) ✓
(2, 2, 3) ✓
(2, 3, 4) ✓

61

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓
(1, 1, 3) ✓
(2, 2, 2) ✓
(2, 2, 3) ✓
(2, 3, 4) ✓

62

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b++ == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓
(1, 1, 3) ✓
(2, 2, 2) ✓
(2, 2, 3) ✓
(2, 3, 4) ✗

63

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c++) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)
(1, 1, 3)
(2, 2, 2)
(2, 2, 3)
(2, 3, 4)

64

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a++ == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)
(1, 1, 3)
(2, 2, 2)
(2, 2, 3)
(2, 3, 4)

65

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 1, 1)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(4, 3, 2)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(1, 1, 1)

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c++) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(2, 3, 2)

(0, 0, 0)
(1, 1, 3)
(2, 2, 2)
(2, 2, 3)
(2, 3, 4)

66

Here we see the additional test cases we need in order to kill these four mutants.

This is an equivalent mutant - we cannot kill it.

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a++ == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

equivalent!

(4, 3, 2)

(1, 1, 1)

(2, 3, 2)

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

67

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

Our tests still haven't found the bug!

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

(0, 1, 1)

(4, 3, 2)

(1, 1, 1)

(2, 3, 2)

68

Performance

69

```
int triangle(int a, int b, int c) {
  if (a <= 0 || b <= 0 || c <= 0) {
      return 4; // invalid
  }
  if (! (a + b > c && a + c > b && b + c > a)) {
      return 4; // invalid
  }
  if (a == b && b == c) {
      return 1; // equilateral
  }
  if (a == b || b == c || a == c) {
      return 2; // isosceles
  }
  return 3;  // scalene
}
```

70

a + b > c

```
a - b > c           a - b >= c      a - b > 0
a * b > c           a - b < c       ++a - b > c
a / b > c           a - b <= c      a - ++b > c
a % b > c           a - b = c       a - b > ++c
a > c               a - b != c      --a - b > c
b > c               b - b > c       a - --b > c
abs(a) - b > c      a - a > c       a - b > --c
a - abs(b) > c      c - b > c       ++(a - b) > c
a - b > abs(c)      a - c > c       --(a - b) > c
abs(a - b) > c      a - b > a       -a - b > c
-abs(a) - b > c     a - b > b       a - -b > c
a - -abs(b) > c     a - b > c       a - b > -c
a - b > -abs(c)     0 - b > c       -(a - b) > c
-abs(a - b) > c     a - 0 > c       0 > c
```

71

How many mutants can you find for the expression a + b > c? This slide lists 42, but this is not an exhaustive list - we could create even more mutants.

Performance Problems

- Many mutation operators possible
  Proteum - 103 Mutation Operators for C
  MuJava - Adds 24 Class level Mutation Operators
- Each mutation operator results in many mutants
  Depending on program under test
- Each mutant needs to be compiled
- Each test case needs to be executed against every mutant

72

Improvements

Do fewer

- Mutant sampling
- Selective mutation

Do smarter

- Parallelize
- Weak mutation
- Use coverage
- Impact

Do faster

- Mutate bytecode
- Mutant schemata

73

Using Coverage

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;   // sc
}
```

Only these tests execute mutants in this line!

(0, 0, 0)
(1, 1, 3)
(2, 2, 2)
(2, 2, 3)
(2, 3, 4)
(0, 1, 1)
(4, 3, 2)
(1, 1, 1)
(2, 3, 2)

74

If we mutate the last if expression, then there is no point in executing all of the test cases against the mutants derived from the expression. Only some of the test cases will actually execute the mutation. If a test case does not execute the mutation, then there is no way it could kill it. Therefore, before mutation analysis we determine statement coverage for each of the test cases, and during mutation analysis only execute those test cases for a mutant that actually reach the mutation.

Strong vs. Weak Mutation

- Strong mutation
  Mutation has propagated to some observable behavior
- Weak mutation
  Mutation has affected state (infection)
- Compare internal state after mutation
- Does not guarantee propagation
- Reported to safe 50% execution time

75

Strong vs. Weak Mutation

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

Weak mutation

Strong mutation

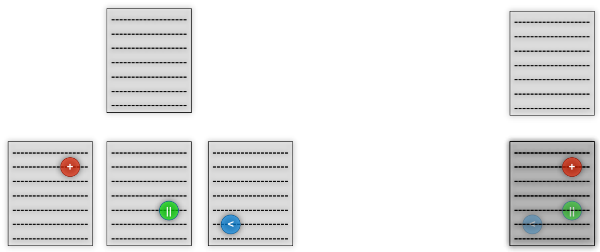
```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x * y;
    x = y;
    y = tmp;
  }

  return x;
}
```

76

Mutant Schemata



Mutant 1: Compile Execute

Mutant 2: Compile Execute

Mutant 3: Compile Execute

Meta-Mutant:
Compile
Switch on Mutant 1
Execute
Switch on Mutant 2
Execute
Switch on Mutant 3
Execute

77

Mutant schemata create one big meta-mutant instead of many separate mutants. The advantage is that the compilation step only has to be done once for the whole meta-mutant. The meta mutant has an extra parameter to activate mutants, and the overhead for this is only very small.

Mutant Schemata

`a + b > c` → `arithOp(a, b, 42) > c`

```
int arithOp(int op1, int op2, int location) {
  switch(variant(location)) {
    case aoADD:   return op1 + op2;
    case aoSUB:   return op1 - op2;
    case aoMULT:  return op1 * op2;
    case aoDIV:   return op1 / op2;
    case aoMOD:   return op1 % op2;
    case aoLEFT:  return op1;
    case aoRIGHT: return op2;
  }
}
```

Select active mutant

78

This is a possible way to implement a mutant schema. Each occurrence of an arithmetic operation is replaced with a call to the function arithOp, together with the operands and information on the location of the call). The function variant (location) returns the original operation if no mutant at location is active, else it returns the currently activated mutant.

Mutant Schemata

```
a + b > c
```

→

```
relOp(arithOp(a, b, 42), c, 42)
```

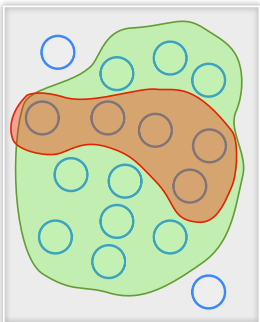
```
boolean relOp(int op1, int op2, int location) {
  switch(variant(location)) {
    case roLT:   return op1 < op2;
    case roGT:   return op1 > op2;
    case roLTE:  return op1 <= op2;
    case roGTE:  return op1 >= op2;
    case roEQ:   return op1 == op2;
    case roNEQ:  return op1 != op2;
  }
}
```

79

Selective Mutation

Full Set:
Test cases that kill all mutants

Sufficient Subset:
Test cases that kill these mutants will kill all mutants



80

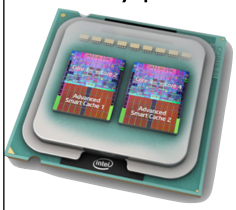
Selective Mutation

- Use only a subset of mutation operators instead of all operators
- Subset is sufficient
- Detecting mutants of sufficient subset will detect >99% of all mutants
- ABS, AOR, COR, ROR, UOI

81

Do Smarter/Faster

Mutation testing is inherently parallelizable



```
6:  iload_1 // a
7:  ifeq 14
10: iload_2 // b
11: ifne 23
14: iload_3 // c
15: ifeq 34
// ...
23: invokevirtual  #4;
// ...
34: invokevirtual  #5;
```

Mutating bytecode avoids recompilation

Sample subset of mutants

82

Higher Order Mutants

83

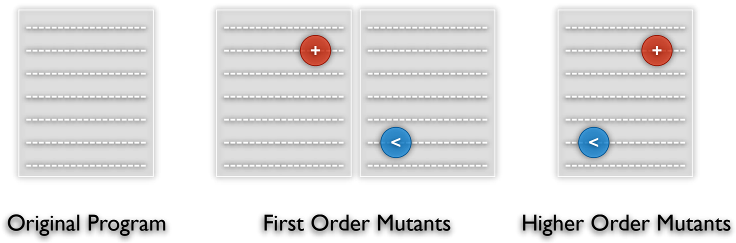
First Order Restriction

FOMs are easy to detect and kill

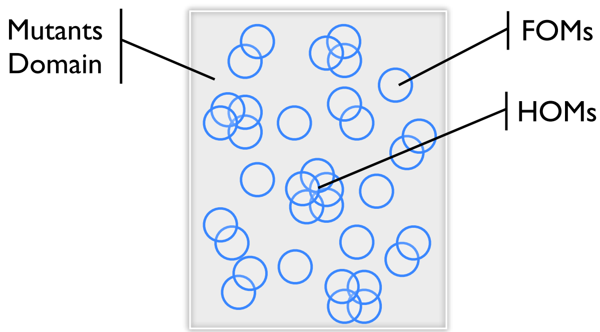
e.g. + → -

Only a small proportion of the FOMs are likely to simulate real faults.

84

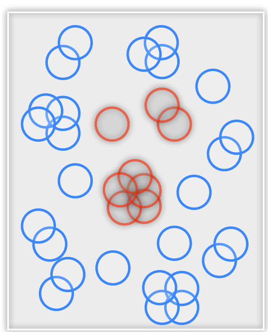


FOMs and HOMs



Higher Order Mutation Testing

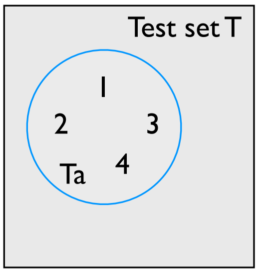
Search for a small set of highly fit mutants within an enormous space, rather than to enumerate a complete set.



HOM classification

Most common case

FOM a is killed by
{ 1, 2, 3, 4 }



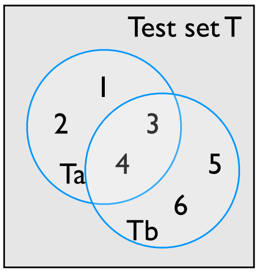
88

HOM classification

Most common case

FOM a is killed by
{ 1, 2, 3, 4 }

FOM b is killed by
{ 3, 4, 5, 6 }



89

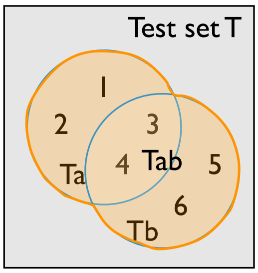
HOM classification

Most common case

FOM a is killed by
{ 1, 2, 3, 4 }

FOM b is killed by
{ 3, 4, 5, 6 }

HOM ab is killed by
{ 1, 2, 3, 4, 5, 6 }



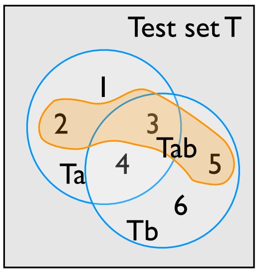
90

The most common case for HOMs is where any test case that kills one of the FOMs the HOM is constructed from will also kill the HOM.

HOM classification

Types of HOMs

Subsuming HOM



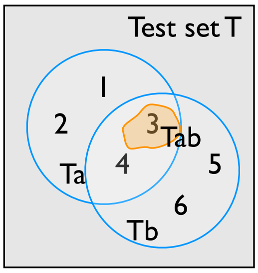
A more interesting case is where only some of the test cases that kill the constituent FOMs can kill the HOM. This is a subsuming HOM, and represents an interesting case where the FOMs influence each other.

HOM classification

Types of HOMs

Subsuming HOM

Strongly Subsuming



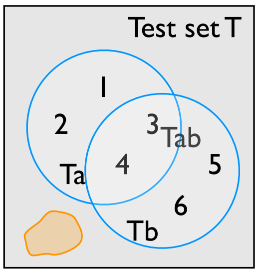
A particularly interesting case is where the set of test cases that kills the HOM is contained in the intersection of the sets of test cases of the FOMs. Any test case that kills the HOM is guaranteed to also kill all the FOMs.

Types of HOM

Subsuming HOM

Strongly Subsuming

De-Coupled

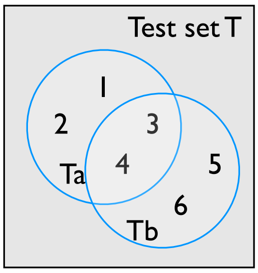


A decoupled HOM cannot be killed by any of the test cases that would kill the FOMs.

Types of HOM

Types of HOMs

- Subsuming HOM
- Strongly Subsuming
- De-Coupled
- Equivalent

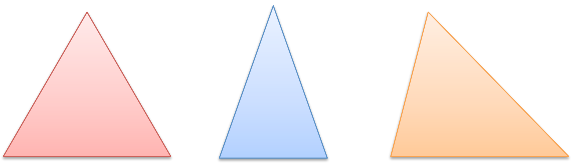


94

Of course a HOM can also be equivalent, even if the FOMs are not equivalent. In other words, the FOMs cancel each other's effects.

Case Study: Triangle

Classify triangle by the length of the sides



Equilateral Isosceles Scalene

95

```
int trian(int a, int b, int c) {
  if(a<=0||b<=0||c<=0)
    return INVALID;
  int trian = 0;
  if(a==b) trian = trian+1;
  if(a==c) trian = trian+2;
  if(b==c) trian = trian+3;
  if (trian == 0)
    if (a + b < c || a + c < b || b + c < a)
      return INVALID;
    else
      return SCALENE;
  if (trian > 3)
    return EQUILATERAL;
  if(trian==1 && a+b>c)
    return ISOSCELES ;
  else if(trian==2 && a+c>b)
    return ISOSCELES;
  elseif(trian==3 && b+c>a)
    return ISOSCELES;

  return INVALID;
}
```

```
a==c && a+b>c
&& a+c<=b

b==c && a+b>a
&& b+c<=a

a==b &&
a+b>c

a==b &&
a+b<=c
```

`if(trian > 1 && a+b>c)`

(2, 4, 2)
(4, 2, 2)
(2, 2, 3)

`if(trian == 1 && a+b<=c)`

(2, 2, 3)
(2, 2, 4)

`if(trian > 1 && a+b<=c)`

(2, 2, 3)

96

Here is an example of a strongly subsuming HOM on a different implementation of the triangle program. Only test cases that satisfy the constraints in the intersection of the constraints of the FOMs can kill the HOM.

(2,4,2) INVALID
(4,2,2) INVALID
(2,2,3) ISOCELES
(2,2,4) INVALID

HOMs in Practice

Program	LoC	FOM	SHOM	SSHOMs
Triangle	50	601	14.79%	0.24%
Tcas	150	744	10.21%	0.11%
Schedule2	350	1,603	32.81%	0.27%
Schedule	400	1,213	15.96%	0.39%
Totinfo	500	2,316	20.61%	0.24%
Replace	550	4,195	20.22%	0.31%
Printtokens2	600	1,714	16.54%	0.10%
Printtokens	750	1,237	24.33%	0.01%
Gzip	5,500	12,027	12.38%	0.08%
Space	6,000	68,843	7.29%	0.21%

Case studies show that subsuming HOMs are quite common. The more interesting strongly subsuming HOMs are rare - but given the number of possible mutants, 0.3% is actually a usefully large number.

HOM Conclusions

- Many HOMs are simple to kill (coupling effect)
- But some HOMs are very interesting
- Because there are so many HOMs, there are many interesting ones as well
- The ratio of equivalent HOMs is better than for FOMs
- How to get good HOMs? Open research problem

Mutation vs. Statement Coverage

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

SDL achieves statement coverage

Mutation vs. Branch Coverage

```
int gcd(int x, int y) {
  int tmp;

  while(false) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

CPR achieves
branch coverage

100

Mutation Testing vs Coverage

- Statement coverage - SDL (statement deletion operator)
- Branch coverage - CPR (constant for predicate replacement)
- Clause coverage - ROR+COR+LOR
- CoC is not subsumed
- GACC is subsumed by ROR+COR+LOR

101

Estimating #Defects

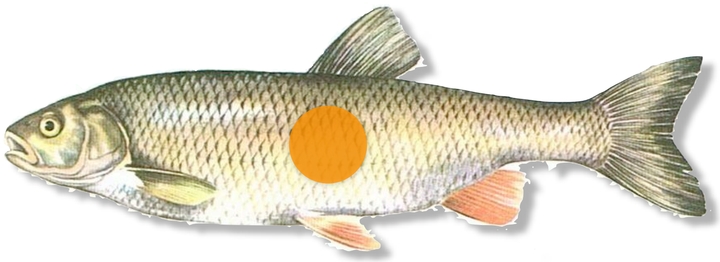
- How many defects remain in our software?
- With mutation testing, we can make an *estimate* of remaining defects

102

Let’s consider a lake. How many fish are in that lake?

103

Fish Tag



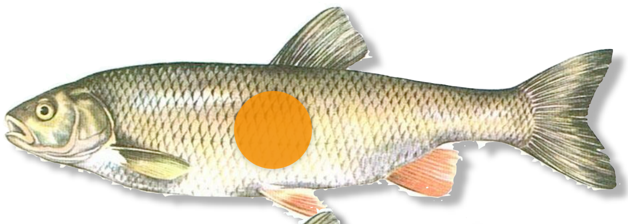
- We catch 1,000 fish and tag them

Simple. We catch a number of fish (say, 1000), tag them, and throw them back again.

104

Counting Tags

50



300

Let’s assume over the next week, we ask fishermen to count the number of tags. We find 300 untagged and 50 tagged fish.

105

Estimate

$$\frac{1{,}000}{\text{untagged fish population}} = \frac{50}{300}$$

…and we can thus estimate that there are about 6,000 remaining untagged fish in the lake.

106

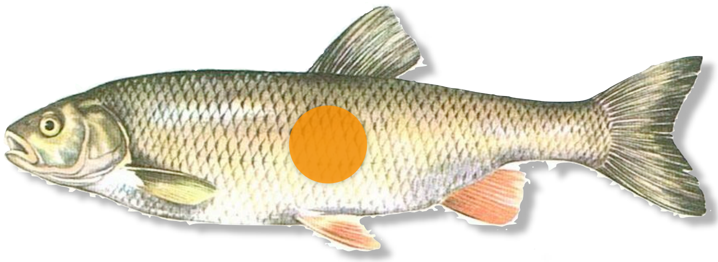
That’s how we can tell how many fish there are.

107

Now let’s assume our lake is not a lake, but our program.

108

A Mutant



- We seed 1,000 mutations into the program

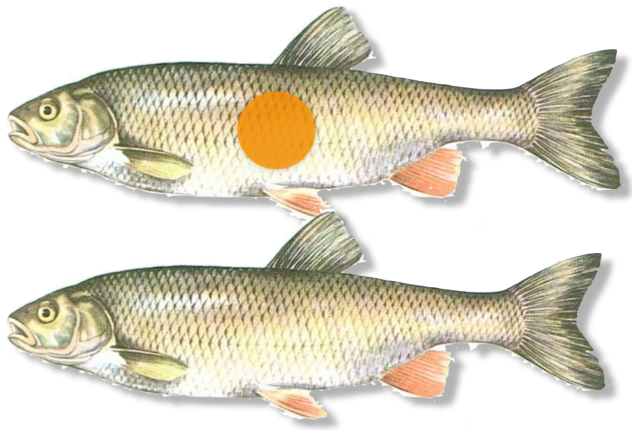
Simple. We catch a number of fish (say, 1000), tag them, and throw them back again.

109

Counting Mutants

50

300



Our test suite finds 50 mutants, and 300 natural faults.

110

Estimate

$$\frac{1{,}000}{\text{remaining defects}} = \frac{50}{300}$$

…and we can again estimate that there are about 6,000 remaining defects in our program. (A test suite finding only 50 out of 1,000 mutations is a real bad sign.)

111

Mutation testing focuses on First Order Mutants

Competent Programmer Hypothesis

Coupling Effect

Improvements

Do fewer

- Mutant sampling
- Selective mutation

Do smarter

- Parallelize
- Weak mutation
- Use coverage

Do faster

- Mutate bytecode
- Mutant schemata

Example 2

```
if(x > 0) {
  if(y > abs(x)) {
    // ...
  }
}
```

HOM classification

Many types of HOM

Subsuming HOM

Strongly Subsuming