PA2516: COVERAGE AND INTERVENTION 3: FAULT-BASED (MUTATION) TESTING

Kai Petersen,
Blekinge Institute of Technology

Based on Gordan Fraser's slides, Mutation testing
as well as Ammann & Offutt - Introduction to Software Testing

KEY QUESTION

- Tests show only the presence, not the absence of faults
- We have seen how we find tests based on coverage (considering requirements and source code structure)
- How can we know how good our tests are?
  - Calculation: #found faults/#total faults, but: we do not know the total number of faults
- How to solve this?
  - Use known or inserted faults to evaluate the goodness of test cases, also known as mutation testing

FAULT-BASED TESTING

- Mutation testing utilizes faults in the software to:
  - generate test data
  - evaluation of testing effectiveness

DEFINITIONS

- Mutant: Single change in a program leads to a new mutant that is syntactically legal
- We would like to introduce mutants that lead to a different behaviour of the program, but the program should still compile (syntactical correctness)
- Goal: Kill mutants
    - If a test case causing the mutant to fail means that the mutant is dead
    - A mutant is called stubborn if the existing set of test cases are insufficient to kill it
    - Test cases that identified mutants are useful
    - Consensus that mutants increase the strength of test cases

MUTATION TESTING EXAMPLE

Test verdict

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

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

Pass

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

Mutant with an operator change

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

Fail

GENERATING MUTANTS

Mutation operators

ABSOLUTE VALUE INSERTION

Def: Each arithmetic expression (and subexpression) is modified by the functions abs(), negAbs(), and failOnZero()

```
int gcd(int x, int y) {
    int tmp;
    while(y != 0) {
      tmp = x % y;
      x = y;
      y = tmp;
   }

   return x;
}
```

Abs()

```
int gcd(int x, int y) {
    int tmp;
    while(y != 0) {
      tmp = x % y;
      x = abs(y);
      y = tmp;
   }

   return x;
}
```

negAbs()

```
int gcd(int x, int y) {
    int tmp;
    while(y != 0) {
      tmp = x % y;
      x = y;
      y = tmp;
   }

   return -abs(x);
}
```

failOnZero()

```
int gcd(int x, int y) {
    int tmp;
    while(y != 0) {
      tmp = x % y;
      x = y;
      y = tmp;
   }

   return 0;
}
```

ARITHMETIC OPERATOR REPLACEMENT

Def: Each occurrence of one of the arithmetic operators $+, -, *, /, **$, and % is replaced by each of the other operators. In addition, each is replaced by the special mutation operators leftOp, rightOp, and mod.

```
int gcd(int x, int y) {
    int tmp;
    while(y != 0) {
      tmp = x % y;
      x = y;
      y = tmp;
  }
  return x;
}
```

```
int gcd(int x, int y) {
    int tmp;
    while(y != 0) {
      tmp = x * y;
      x = y;
      y = tmp;
  }
  return x;
}
```

Also, left and right operators are dropped, i.e. tmp = x; and tmp = y;

RELATIONAL OPERATOR REPLACEMENT

Def: Each occurrence of one of the relational operators (<, ≤, >, ≥, =, =) is replaced by each of the other operators and by falseOp and trueOp.

```
int gcd(int x, int y) {
    int tmp;
    while(y != 0) {
      tmp = x % y;
      x = y;
      y = tmp;
  }

  return x;
}
```

```
int gcd(int x, int y) {
    int tmp;
    while(y > 0) {
      tmp = x % y;
      x = y;
      y = tmp;
  }

  return x;
}
```

Also, true and false are put directly into the expression, i.e. while(true) as well as while(false)

CONDITIONAL OPERATOR REPLACEMENT

Def: Each occurrence of each logical operator (and–&&, or–, and with no conditional evaluation–&, or with no conditional evaluation–|, not equivalent – ^) is replaced by each of the other operators; in addition, each is replaced by falseOp, trueOp, leftOp, and rightOp.

```
if (a && b)

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

SHIFT OPERATOR REPLACEMENT

Def: Each occurrence of one of the shift operators <<, >>, and >>> is replaced by each of the other operators. In addition, each is replaced by the special mutation operator leftOp.

m<<a

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

Excursion: The bitwise-shift operator

Syntax: [variable]<<[number of places]

```
int mult_power(int number, int power)
{
    return number<<power;
}
```

Example: 00001111
Shift by 2: 00111100

Distinction between signed and unsigned shift operator to be made.
<< and >> are signed shift operators
<<< and >>> are unsigned shift operators

LOGICAL OPERATOR REPLACEMENT

in real life

Def: Each occurrence of each bitwise logical operator (bitwise and (&), bitwise or (|), and exclusive or (^)) is replaced by each of the other operators; in addition, each is replaced by leftOp and rightOp.

```
x = m & n;
```

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

ASSIGNMENT OPERATOR REPLACEMENT

Def: Each occurrence of one of the assignment operators (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=) is replaced by each of the other operators.

```
x += 3;
```

```
x -= 3;
x *= 3;
x /= 3;
x %= 3;
x &= 3;
x |= 3;
x ^= 3;
x <<= 3;
x >>= 3;
x >>>= 3;
```

UNARY OPERATOR INSERTION

Def: Each unary operator (arithmetic +, arithmetic −, conditional !, logical ~) is inserted before each expression of the correct type.

```
x = 3 * a;
```

```
x = 3 * +a;
x = 3 * -a;
x = +3 * a;
x = -3 * a;
```

UNARY OPERATOR DELETION

Def: Each unary operator (arithmetic +, arithmetic −, conditional !, logical ~) is deleted.

if !(a > -b)

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

SCALAR VARIABLE REPLACEMENT

Def: Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.

```
x = a * b;
```

```
x = a * a;
a = a * b;
x = x * b;
x = a * x;
x = b * b;
b = a * b;
```

GENERATING MUTANTS

Integration/Interface Mutation

INTEGRATION MUTATION

- Integration mutation: Often misunderstandings on
  - delivery side: deliver what is not desired by the caller (e.g caller expects records in ascending order, but delivered in random order)
  - receiver side: recipient makes assumptions on what is delivered (e.g. expects to receive data in a specific format, which was not the case)
  - Integration (interface) mutation: Mutate connections between components

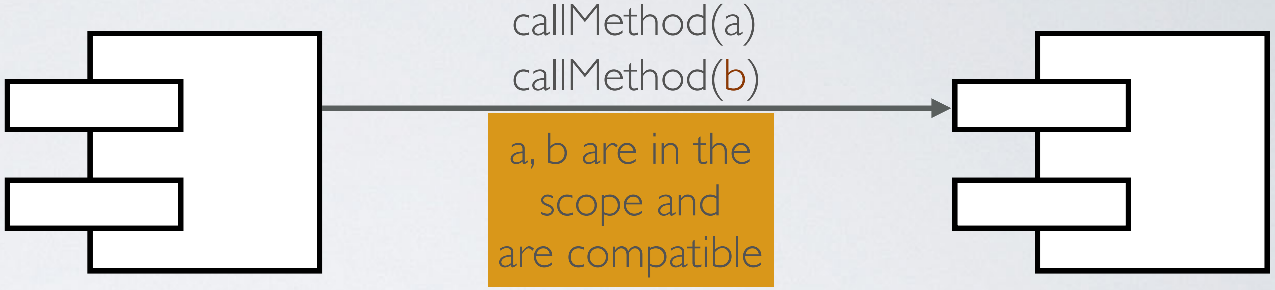
TYPES OF MUTATION OPERATORS

- T1: Change a calling method by modifying the values sent to a called method
- T2: Change a calling method by modifying the call
- T3: Change a called method by modifying the values that enter and leave a method
- T4: Change a called method by modifying the return statements

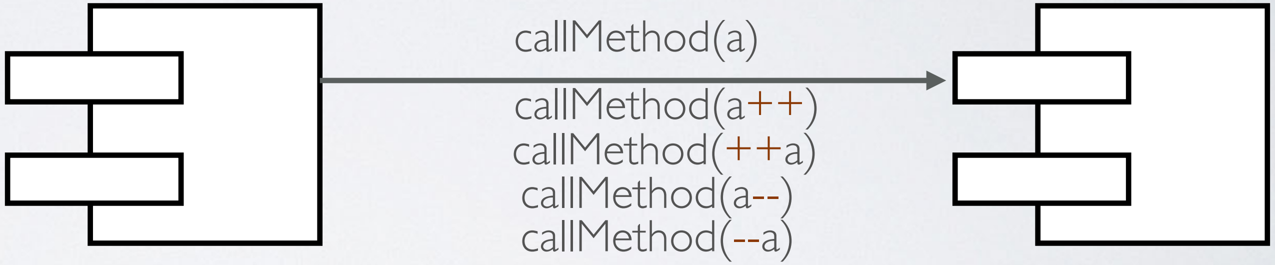
T1/T2

BLEKINGE TEKNISKA HÖGSKOLA • BTH •
in real life

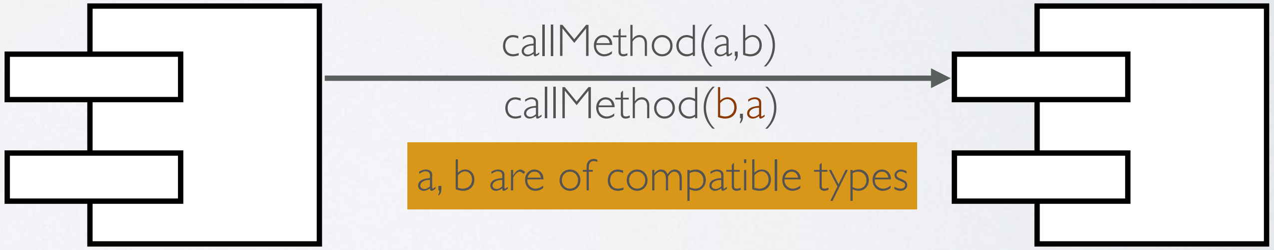
Integration parameter variable replacement: Each parameter in the method call is replaced by another variable in the scope of the method call



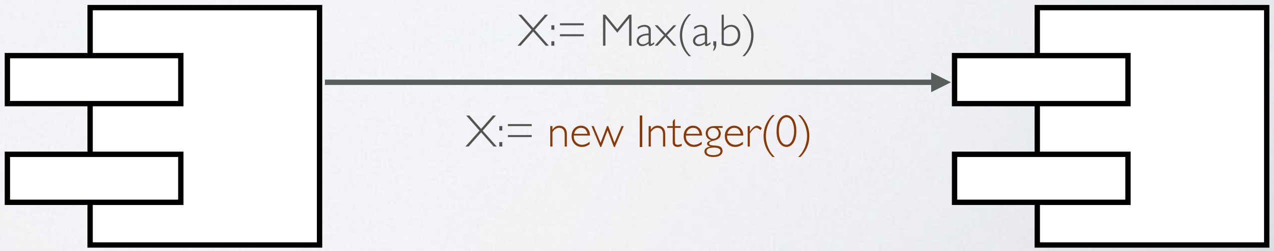
Integration unary insertion: Each expression in the method call is modified by inserting all possible unary operators in front and behind it



Integration Parameter Exchange: Each parameter in a method call is exchanged with each parameter of compatible types in that method call.

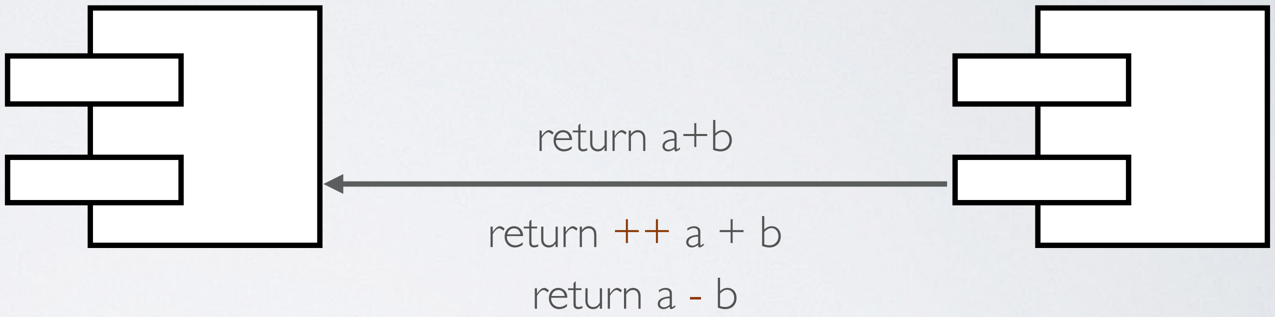


Integration method call deletion: Each method call is deleted. If the method returns a value and it is used in an expression, the method call is replaced with an appropriate constant value.



T3/T4

Integration Return Expression Modification: Each expression in each return statement in a method is modified by applying the unary operator insertion (UOI) and arithmetic operator replacement (AOR) operators.



GENERATING MUTANTS

OO Mutation

Earlier we only considered mutation within methods or functions, we now consider object oriented concepts related to information hiding, inheritance, polymorphism, dynamic binding, and method overloading

EXCURSION: OO CONCEPTS

- Encapsulation: Information hiding = restriction to member variables (modification of variables through methods)
- Method overriding: Method of subclass can be redefined (different implementation), but has the same name, arguments, and result type
- Variable hiding: Variable defined in the child class with same name hides variable in the inheriting class
- Polymorphism: Interface that can take different types (templates/generics), e.g. in sorting one could hand over different object types (strings, ints, etc.)
- Method overloading: In the **same class** we use the same name for constructors and methods, but with different parameters
- Method overriding: Child class declares a method with **the same name** as in the mother class and hence overrides the method of the parent class

OO MUTATIONS ENCAPSULATION

Access Modifier Change: The access level for each instance variable and method is changed to other access levels.

Original

myclass
private int myVar;

Mutants

myclass
public int myVar;

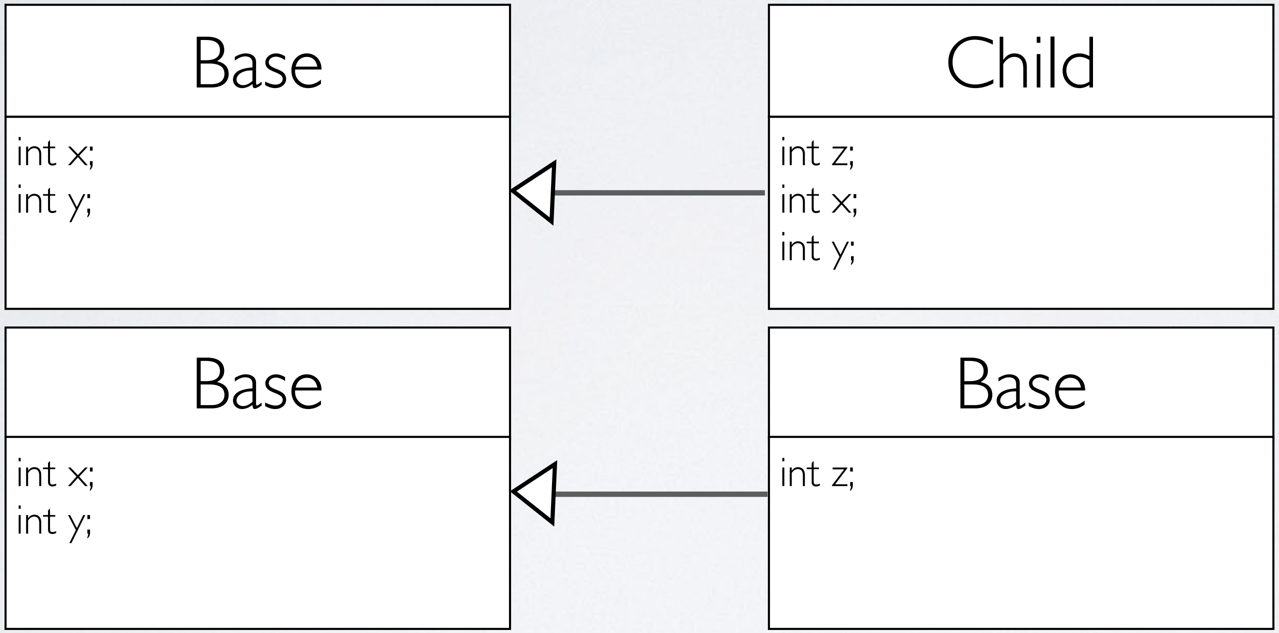
myclass
protected int myVar;

myclass
int myVar;

Goal: Check whether the accessibility of variables is correct

OO MUTATIONS INHERITANCE

Hiding variable deletion: Each declaration of an overriding or hiding variable is deleted.



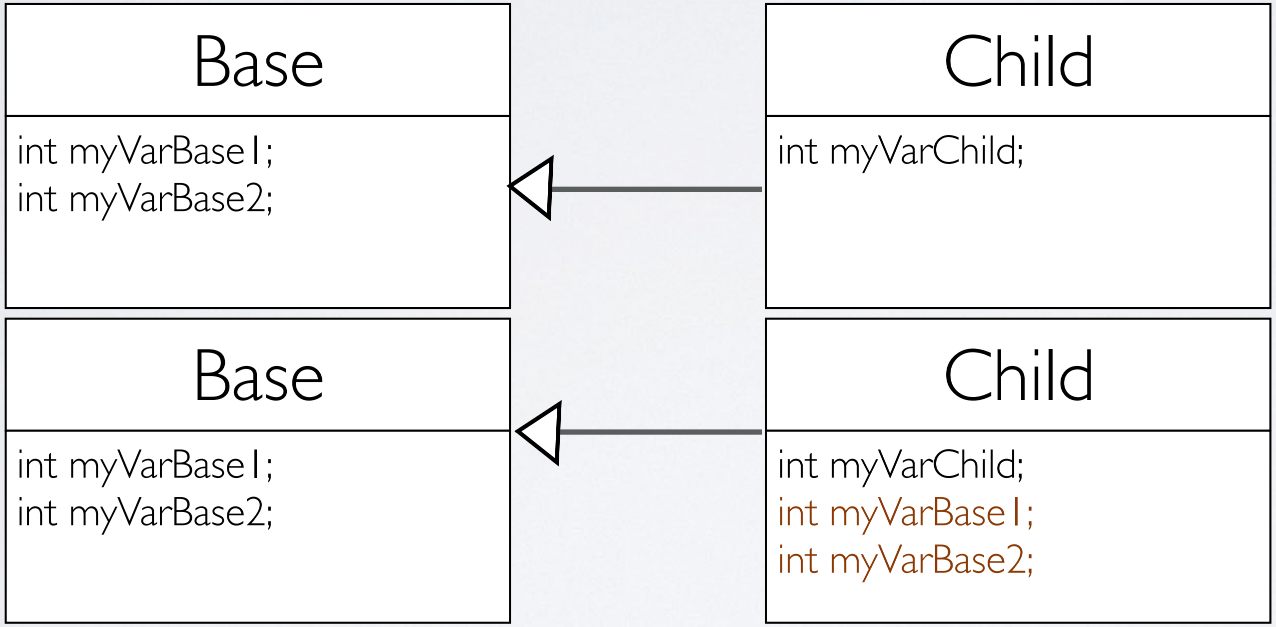
Original

Mutant

Goal: Check whether tests can detect wrong references (common mistake made)

OO MUTATIONS INHERITANCE

Hiding variable insertion: A declaration is added to hide the declaration of each variable declared in an ancestor.



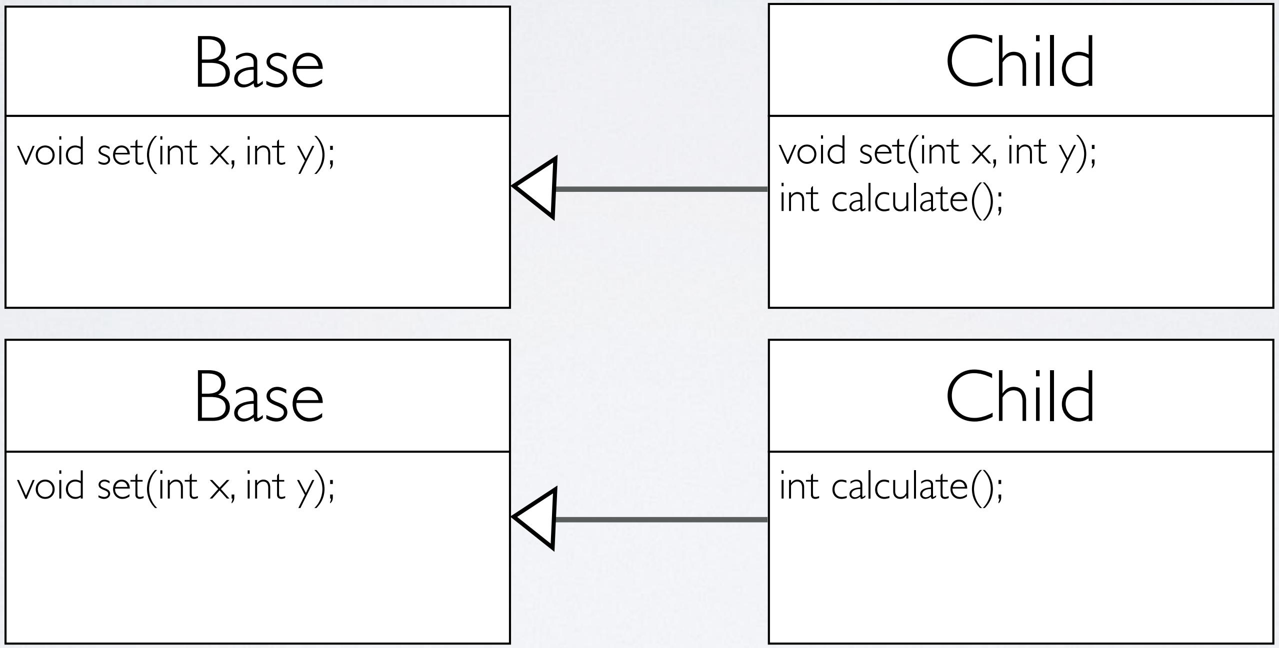
Original

Mutant

Goal: Find test cases that show that reference to the overriding variable is incorrect

OO MUTATIONS INHERITANCE

Overriding method deletion: Each entire declaration of an overriding method is deleted.



Original

Mutant

Goal: Allows to evaluate whether the invocation of the method is to the intended method

OO MUTATIONS CONT.

- Overridden method moving
- Overridden method rename
- Super keyword deletion
- Parent constructor deletion
- Actual type change
- Declared type change
- Parameter type change
- Reference type change
- Overloading method deletion
- Argument order change
- Argument number change
- Keyword deletion
- Static modifier change
- Variable initialisation deletion
- Default constructor delete
- Overloading method change

WEAK AND STRONG MUTATION

- Criteria of reachability, interaction, and propagation**RIP**
  - Reachability (R): Location of the fault must be reachable
  - Infection (I): Fault is executed and leads to an incorrect state in the program
  - Propagation (P): Output of the program is incorrect
- Strong mutation: An incorrect state should propagate to the output of the program (should fulfil R+I+P)
- Weak mutation: Test case reaches the mutant, but does not require propagation to the output (should fulfil R+I)

EXAMPLE RIP

```
public int lastIndexOf(int[] x, int y) {
for (int i = x.length-1; i > 0; i--){
        if( x[i]==y ) return i;
        }

        return -1;
}
```

State (program counter)	Line	Expected	Actual
S1	lastIndexOf()	x=[1]	x=[1]
S2	for (int i =…)	i=0	i=0
S3		Program counter = if (…)	Program counter = return -1

Error state in S3, but we have the expected output

Input: x=[1], y=2
Expected output: -1

BASED ON WEAK AND STRONG MUTATION WE DEFINE COVERAGE CRITERIA

- Weak mutation coverage: For each mutant there exist a test case to weakly kill the mutant.
- Strong mutation coverage: For each mutant there exist a test case to strongly kill the mutant.

EQUIVALENT MUTANTS

- Mutations are equivalent if the behaviour stays the same after introducing a mutation to the code.

```
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];
}
```

The function will give the correct max value for > as well as >=

- Notes on equivalent mutants
  - hard and costly to detect (equivalence mutant problem) —> e.g. Adamopoulos et al. propose a solution
  - if we could find the equivalent mutants, we could reduce the overall testing effort
  - what is more expensive, finding the equivalent mutants and removing them, or including them in the test suit? -> did not find an answer to that question

FIRST AND SECOND ORDER MUTATIONS

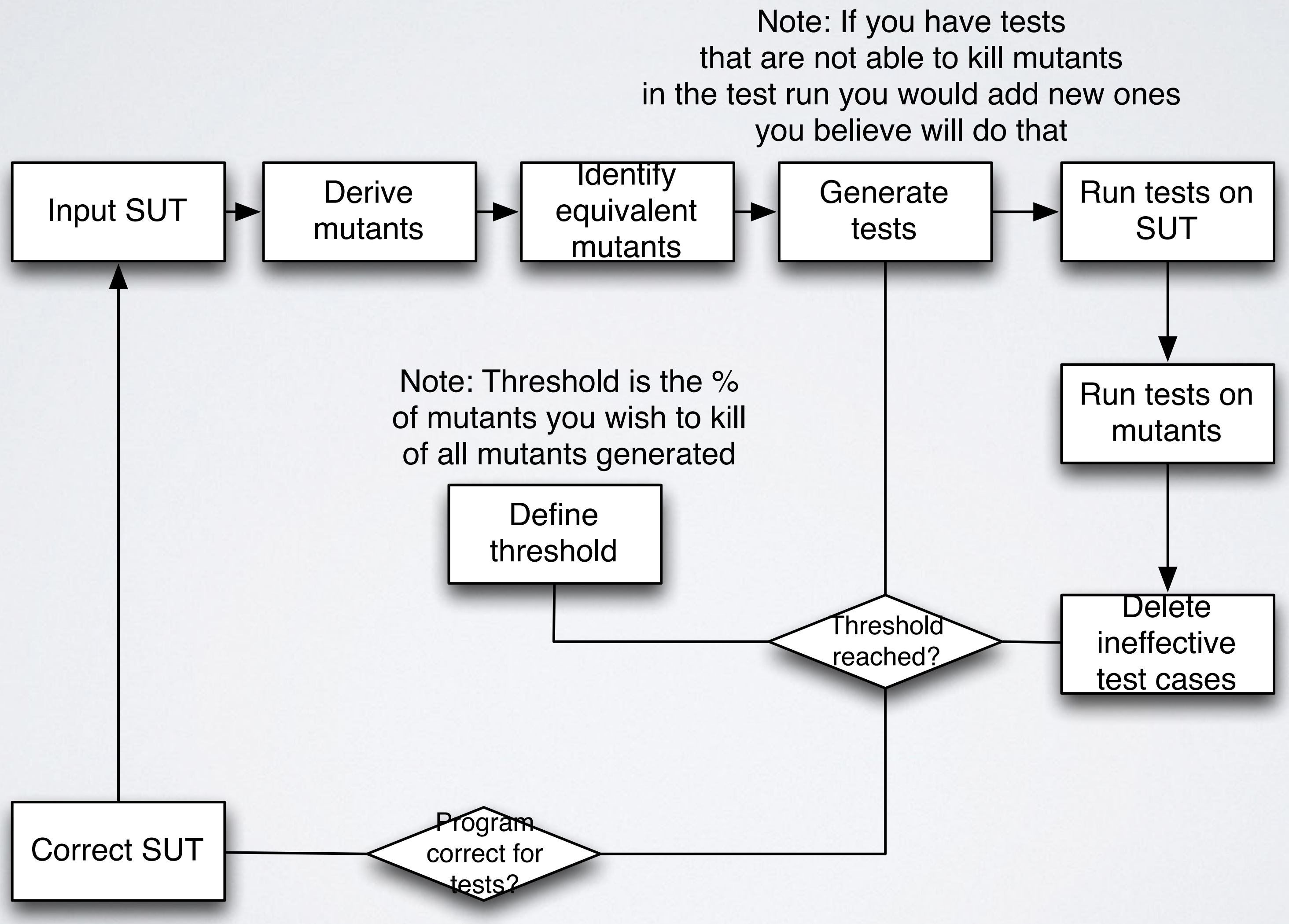
- Definitions:
  - First order mutation: insertion of a single mutant
  - Higher order mutation: Insertion of two more more mutants
- Problem of first order mutants: Trivial and easy to detect (does not consider fault combinations that make the testing interesting/subtle)
- First order mutants do not represent a realistic situation

Yue Jia, Mark Harman: Higher Order Mutation Testing. Information & Software Technology 51(10): 1379-1393 (2009)

EVALUATING MUTANTS

- Mutation score = #dead mutants/ # non-equivalent mutants (mutant coverage)

MUTATION TESTING PROCESS



Mutation testing is considered one of the strongest approaches to arrive at an effective test suite

–But: What is the challenge with it?

EVALUATION OF HIGHER ORDER MUTATIONS

```
a + b > c
```

Original

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

Mutants (selection)

REASONS FOR LOW TIME EFFICIENCY

- High number of mutations for each mutation operator
- To be tested dynamically, mutants have to be compiled and tests have to be run against each mutant
- Consequence:
  - Need strategies for selection
  - Need for high computing power (one of the reasons because there was a large break in research on mutation, invented in 1976, continued in the 90s/2000s)

STRATEGIES I

- Strategies
  - Coverage
  - Mutation sampling
  - Selective mutation
  - Parallelization
  - Weak coverage
  - Mutate bytecode

STRATEGIES II

- Coverage
  - Determine statement coverage
  - Only execute those test cases that reach the statement including a mutation
- Strong vs. weak mutation
  - Compare internal states after mutation
  - Reported to safe 50% execution time
- Mutant schema
  - Create a mutant "product line", using a meta mutant
  - One compilation, mutant activation and deactivation (e.g. switch function)

STRATEGIES III

- Sampling
  - Strategy 1: Identify test cases that kill all mutants (very effective, but not efficient)
  - Strategy 2: Only focus on test cases focusing on those mutation operators that subsume all other mutation operators
- For Strategy 2, a subset of mutants will detect more than 99% of all mutants
  - ABS(), Arithmetic operator replacement, Conditional operator replacement, Relational operator replacement, Unary operator replacement

TOOLS

- Evo-suite - http://www.evosuite.org/
  - Research shows that (based on a sample selection of programs) mutation testing is scalable for the application of novel techniques
  - Generated 1,380,302 mutants for 8,963 classes
- Jester - http://jester.sourceforge.net/

Recommended reading: Gordon Fraser and Andrea Arcuri. Achieving Scalable Mutation-based Generation of Whole Test Suites. Empirical Software Engineering

OVERALL REFLECTIONS

- Research provides evidence that superior test suites can be generated (compared with other test design techniques)
  - structural code coverage
  - data-flow coverage
- Convincing work is done (e.g. through search-based algorithms) to make mutation testing beneficial and cost efficient
- Though: Need more industria application (often open source studied)