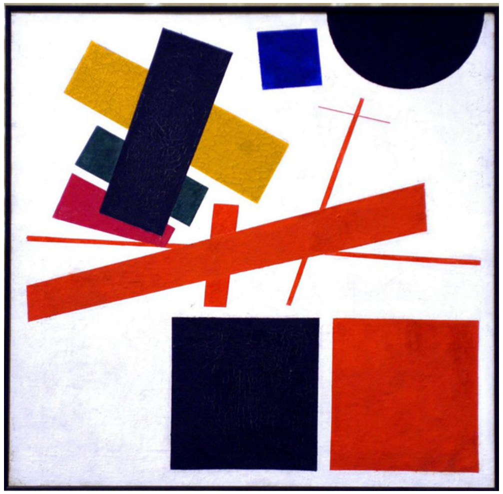
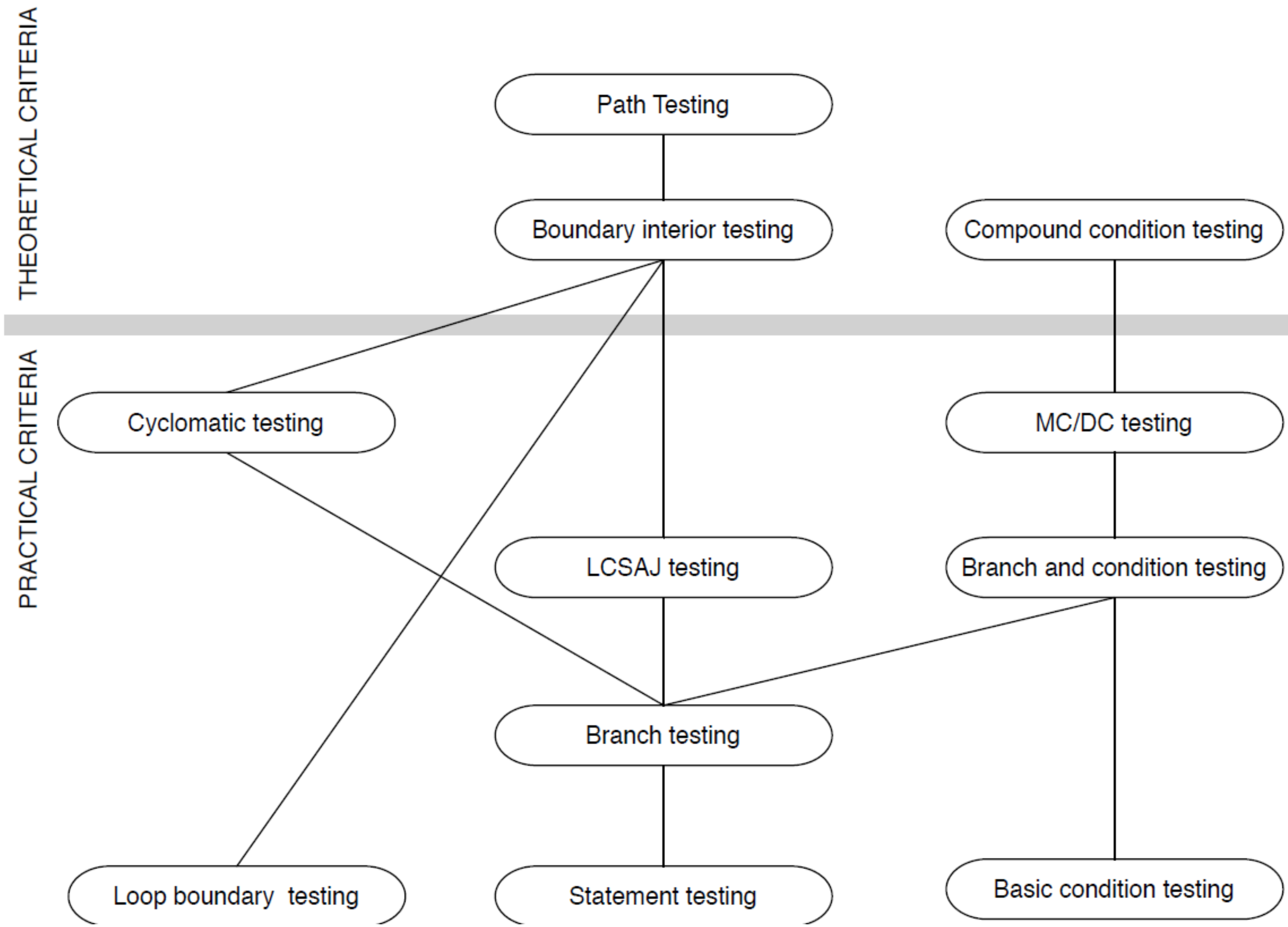
CPEN 422
Software Testing and Analysis



Fault-based Adequacy and Mutation Testing

How Good Are My Test Cases?



```java
/* Make sure Double.NaN is returned iff n = 0 */
public void testNaN1() {
    StandardDeviation std = new StandardDeviation();
    assertTrue(Double.isNaN(std.getResult()));
    std.increment(1d);
    assertEquals(0d, std.getResult(), 0);
}
```

```java
public void testNaN2() {
    StandardDeviation std = new StandardDeviation();
    Double.isNaN(std.getResult());
    std.increment(1d);
    std.getResult();
}
```

Coverage is not changed!

How Good Are My Test Cases?

- Executing all the code is not enough
  - 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

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

Competent Programmer Hypothesis

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

Coupling Effect

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

```
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;
}
```

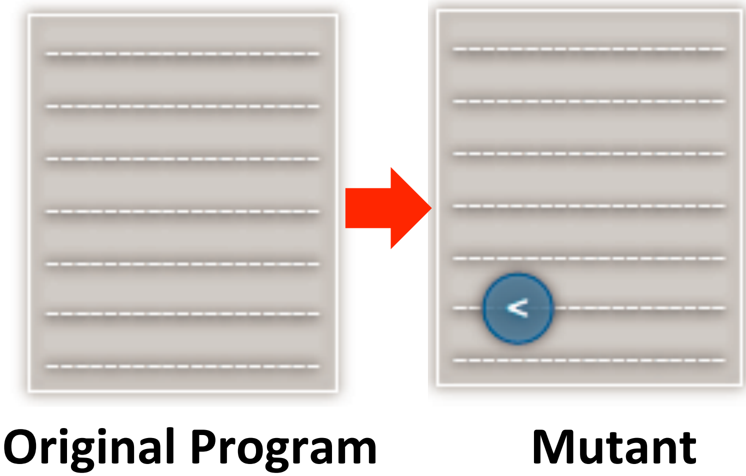
Mutant

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

Test

Mutants

- Slightly changed version of original program



- Syntactic change
  - Valid (compilable code)
- Simple
  - Programming “glitch”, “typo”

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

AOR – Arithmetic Operator Replacement

```
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;
}
```

+, -, *, /, x, y

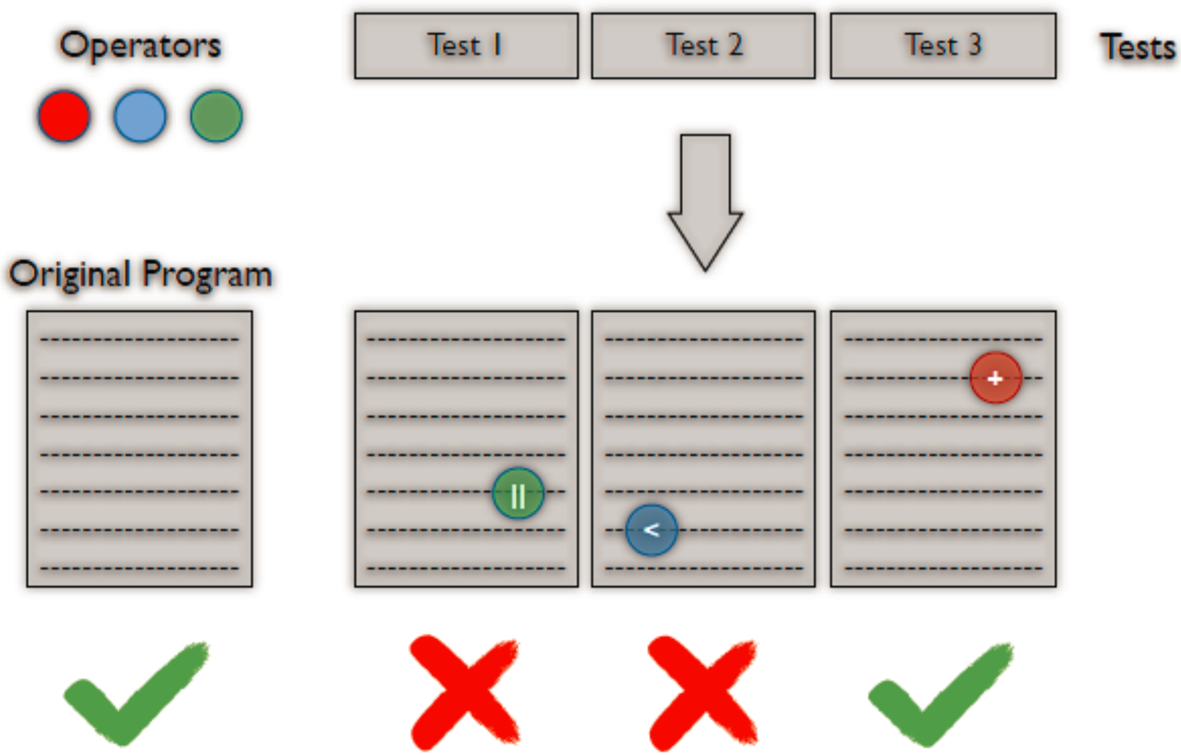
SVR - Scalar Variable Replacement

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

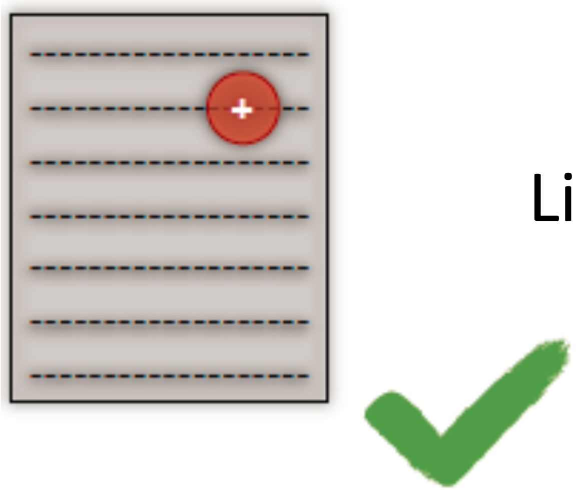
```
int gcd(int x, int y) {
int tmp;
while(y != 0) {
    tmp= y % y;
    x = y;
    y = tmp;
}
return x;
}
```

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

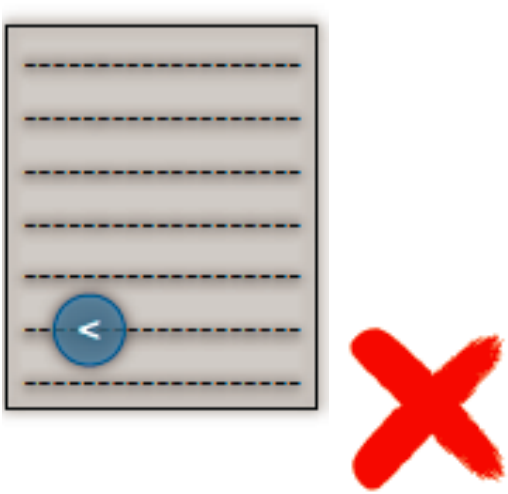
Evaluating the Adequacy of a Test Suite



- Executing each mutant against tests in T until the mutant is detected or we have exhausted all tests.
- Detected Mutant == Killed Mutant
- Undetected Mutant == Live Mutant



Live mutant - we need more tests



Killed mutant - of no further use

foo is required to return the sum of two integers x and y. Clearly foo is incorrect.

```
int foo(int x, y){
   return (x-y);
}
```

Foo should return (x+y)

foo has been tested using a test suite T:

T={ t1: <x=1, y=0>, assert(1),
t2: <x=-1, y=0>, assert(-1)}

```
int foo(int x, y){
   return (x-y);
}
```

- foo returns the expected value for each test case.
- T is adequate with respect to all control and data flow based test adequacy criteria.

Evaluating the adequacy of T using mutation

Three Mutants: M1, M2, M3

```
int foo(int x, y){
   return (x-y);
}
```

M1

```
int foo(int x, y){
   return (x+y);
}
```

M2

```
int foo(int x, y){
   return (x-0);
}
```

M3

```
int foo(int x, y){
   return (0-y);
}
```

Foo should return (x+y)

```
int foo(int x, y){
    return (x-y);
}
```

T={ t1: <x=1, y=0>, assert(1),

t2:<x=-1, y=0>, assert(-1)}

M1

```
int foo(int x, y){
    return (x+y);
}
```

M2

```
int foo(int x, y){
    return (x-0);
}
```

M3

```
int foo(int x, y){
    return (0-y);
}
```

Test (t)	foo(t)	M1(t)	M2(t)	M3(t)
t1<x=1,y=0> e:1	1			
t2<x=-1,y=0> e:-1	-1			
Live/Killed?	Live			

Foo should return (x+y)

```
int foo(int x, y){
   return (x-y);
}
```

T={ t1: <x=1, y=0>, assert(1),
t2:<x=-1, y=0>, assert(-1)}

M1

```
int foo(int x, y){
   return (x+y);
}
```

M2

```
int foo(int x, y){
   return (x-0);
}
```

M3

```
int foo(int x, y){
   return (0-y);
}
```

Test (t)	foo(t)	M1(t)	M2(t)	M3(t)
t1<x=1,y=0> e:1	1	1	1	0
t2<x=-1,y=0> e:-1	-1	-1	-1	0
Live/Killed?	Live	Live	Live	Killed

M1:

```
int foo(int x, y){
   return (x+y);
}
```

t1: <x=1, y=0>, assert(1),

t2:<x=-1, y=0>, assert(-1)

A test that detects M1 from foo must satisfy the following condition: $x-y \neq x+y$ implies $y \neq 0$

M1:

```
int foo(int x, y){
   return (x+y);
}
```

t1: <x=1, y=0>, assert(1),
t2:<x=-1, y=0>, assert(-1)

A test that detects M1 from foo must satisfy the following condition: $x-y \neq x+y$ implies $y \neq 0$

t3: <x=1, y=1>

M1:

```
int foo(int x, y){
   return (x+y);
}
```

t1: <x=1, y=0>, assert(1),
t2:<x=-1, y=0>, assert(-1)

A test that detects M1 from foo must satisfy the following condition: $x-y \neq x+y$ implies $y \neq 0$

t3: <x=1, y=1>, assert(2)

```
int foo(int x, y){
   return (x-y);
}
```

foo(t3)=0 ≠ 2

M1:

```
int foo(int x, y){
   return (x+y);
}
```

t1: <x=1, y=0>, assert(1),

t2:<x=-1, y=0>, assert(-1)

A test that detects M1 from foo must satisfy the following condition: $x-y \neq x+y$ implies $y \neq 0$

t3: <x=1, y=1>

```
int foo(int x, y){
   return (x-y);
}
```

foo(t3)=0 ≠ 2

t3 distinguishes M1 from foo and also reveals the error

Guaranteed error detection

Let P' be a mutant of P and t a test in the input domain of P:

- P' is an error revealing mutant if any test t that distinguishes P' from P ($P'(t) \neq P(t)$), also causes P to fail ($P(t) \neq$ the expected response).

Is M1 an error revealing mutant? What about M2?

M1

```
int foo(int x, y){
   return (x+y);
}
```

M2

```
int foo(int x, y){
   return (x-0);
}
```

M1 and M2 are error-revealing mutants;

Consider M2:

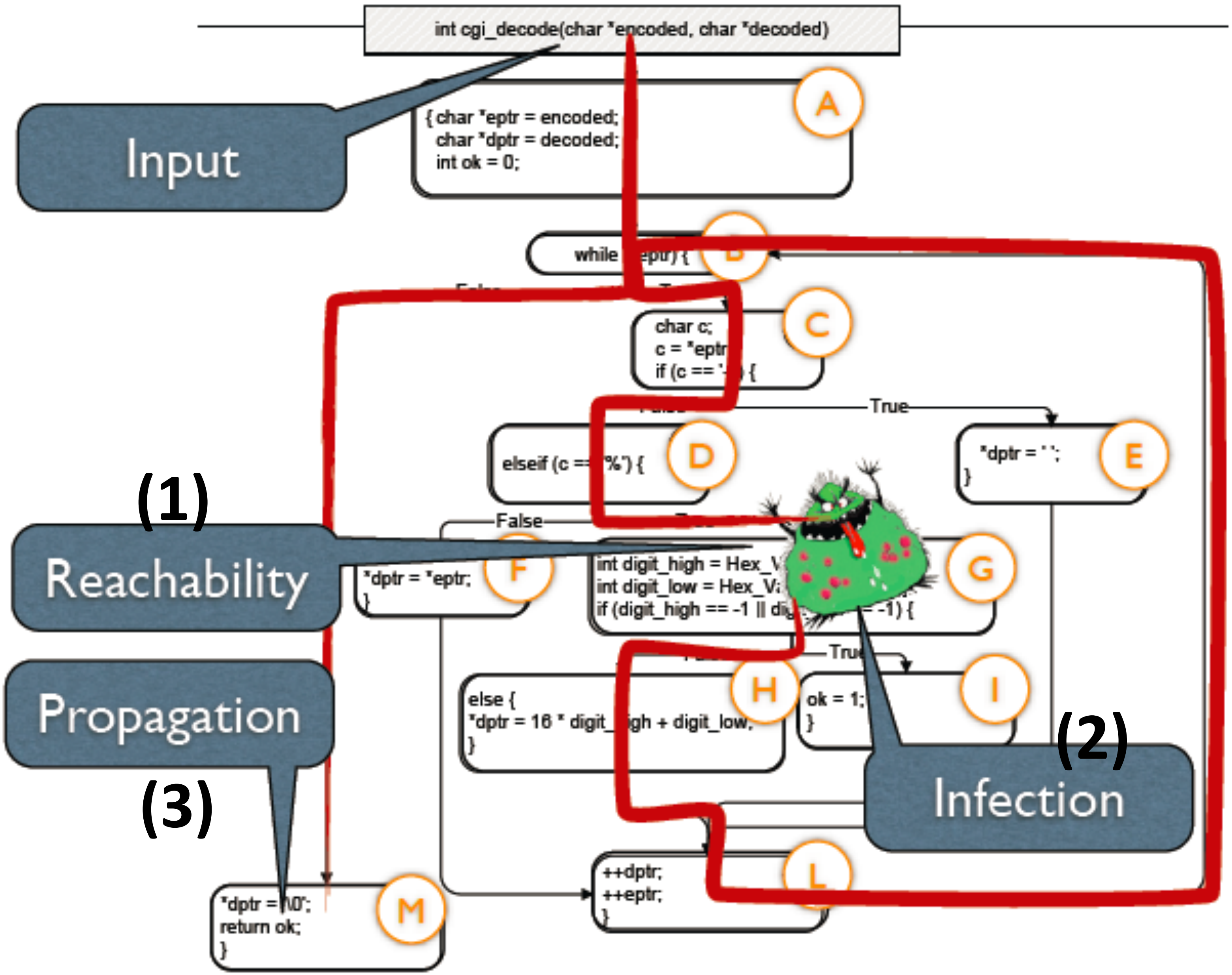
Expected: x+y

P: x-y

P’: x-0

x-y != x-0 => y != 0 => x-y != x+y for test cases in which y!=0.

Detecting a mutant:



Do t1 and t2 satisfy reachability, infection, propagation for M3?

t1: <x=1, y=0>

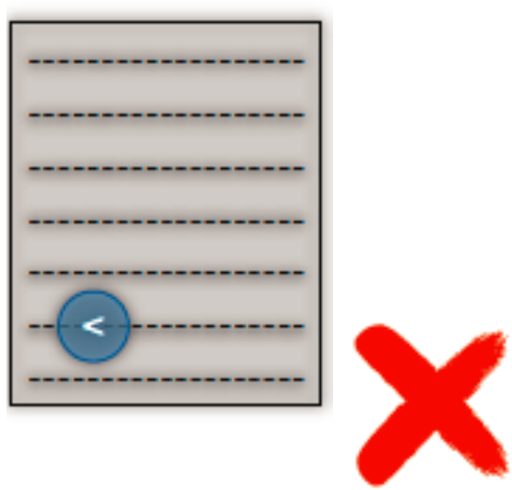
t2: <x=-1, y=0>

M3

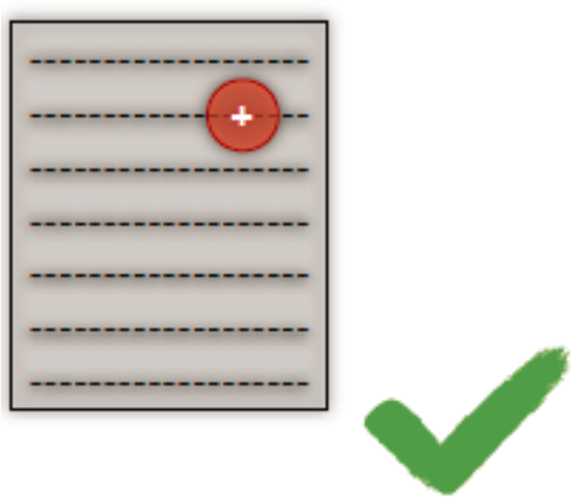
```
int foo(int x, y){
   return (x-y);
}
```

```
int foo(int x, y){
   return (0+y);
}
```

As long as x != 0 it infects the internal state of the mutated part of the code.



$$\text{Mutation Score} = \frac{\text{Killed Mutants}}{\text{Total Mutants}}$$



Equivalent Mutants

- Mutation = syntactic change
- The change might leave the semantics unchanged

Equivalent Mutants

- Mutation = syntactic change
- The change might leave the semantics unchanged
- Equivalent mutants are hard to detect

Equivalent Mutants

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

Equivalent Mutants

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

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
             minVal = B;
         }
         return (minVal);
}
```


```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
             minVal = B;
         }
         return (minVal);
}
```

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
             minVal = B;
         }
         return (minVal);
}
```

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < minVal)
        {
             minVal = B;
         }
         return (minVal);
}
```

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
            minVal = B;
        }
        return (minVal);
}
```

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < minVal)
        {
            minVal = B;
        }
        return (minVal);
}
```

What is the infection condition?

Can the mutant be killed?

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
            minVal = B;
         }
         return (minVal);
}
```

Infection condition: (B < A) != (B < minVal)

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < minVal)
        {
            minVal = B;
         }
         return (minVal);
}
```

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
             minVal = B;
         }
         return (minVal);
}
```

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < minVal)
        {
             minVal = B;
         }
         return (minVal);
}
```

Infection condition: (B < A) != (B < minVal)

minVal == A

(B < A) == (B < minVal)

No input can kill this mutant

The mutant and the original program always produce the same output

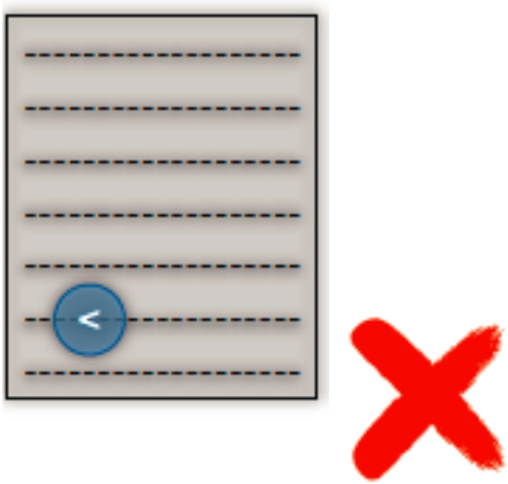
- No test data can distinguish between the two

The mutant and the original program always produce the same output

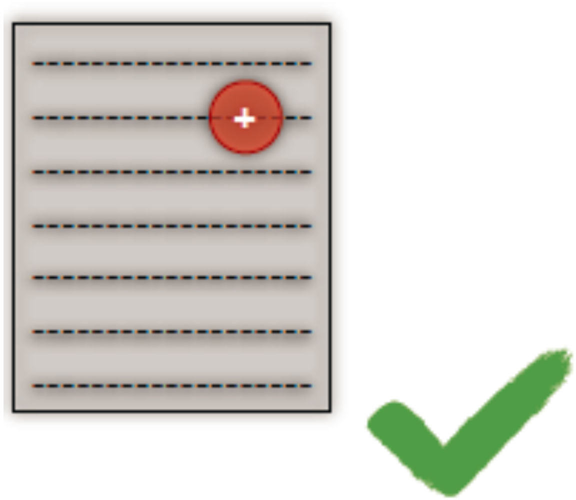
- No test data can distinguish between the two

We only want to kill the non-equivalent mutants, else:

Reaching 100% mutation score would be impossible



$$\text{Mutation Score} = \frac{\text{Killed Mutants}}{\text{Total Mutants} - \text{Equivalent Mutants}}$$



What is the mutation score of test suite T?

```
int foo(int x, y){
   return (x-y);
}
```

M1

```
int foo(int x, y){
   return (x+y);
}
```

M2

```
int foo(int x, y){
   return (x-0);
}
```

M3

```
int foo(int x, y){
   return (0-y);
}
```

Test (t)	foo(t)	M1(t)	M2(t)	M3(t)
t1<x=1,y=0>	1	1	1	0
t2<x=-1,y=0>	-1	-1	-1	0
		Live	Live	Killed

What is the mutation score of test suite T?

$$\frac{1}{3-0} = 0.33$$

```
int foo(int x, y){
   return (x-y);
}
```

M1

```
int foo(int x, y){
   return (x+y);
}
```

M2

```
int foo(int x, y){
   return (x-0);
}
```

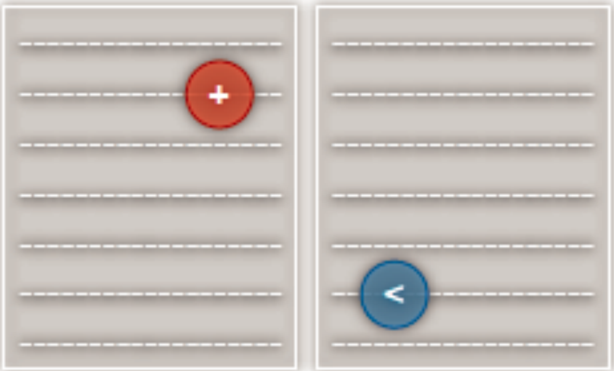
M3

```
int foo(int x, y){
   return (0-y);
}
```

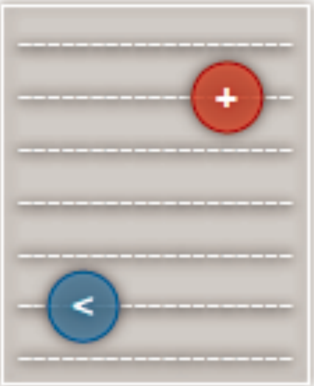
Test (t)	foo(t)	M1(t)	M2(t)	M3(t)
t1<x=1,y=0>	1	1	1	0
t2<x=-1,y=0>	-1	-1	-1	0
		Live	Live	Killed

Order of mutants

- First order mutant (FOM)
  - Exactly one mutation



- Higher order mutant (HOM)
  - Mutant of mutant



Coupling Effect

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

Coupling Effect + Competent Programmer

Focus on First Order Mutants

Performance Problems

- Many mutation operators possible
  - Each mutation operator results in many mutants

- Each test case needs to be executed against every mutant

Improvements

- Mutant sampling
- Selective mutation

- Weak mutation
- Use coverage
- Impact

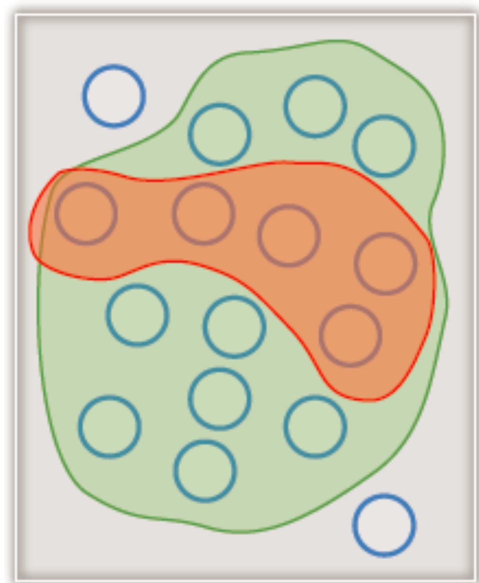
- Mutate bytecode
- Parallelize

Selective Mutation

- Use only a subset of mutation operators instead of all operators

Full Set:
Test cases that kill all mutants

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



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
  - Easier to kill: Less analysis
  - Does not guarantee propagation

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
            minVal = B;
        }
        return (minVal);
}
```

```
int Min (int A, int B)
{
        int minVal;
        minVal = B;
        if (B < A)
        {
            minVal = B;
        }
        return (minVal);
}
```

```
int Min (int A, int B)
{
	int minVal;
	minVal = A;
	if (B < A)
	{
		minVal = B;
	}
	return (minVal);
}
```

```
int Min (int A, int B)
{
	int minVal;
	minVal = B;
	if (B < A)
	{
		minVal = B;
	}
	return (minVal);
}
```

Reachability : ?

Infection : ?

Propagation: ?

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
            minVal = B;
         }
         return (minVal);
}
```

⇒

```
int Min (int A, int B)
{
        int minVal;
        minVal = B;
        if (B < A)
        {
            minVal = B;
         }
         return (minVal);
}
```

Reachability : true

Infection : ?

Propagation: ?

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
            minVal = B;
         }
         return (minVal);
}
```

```
int Min (int A, int B)
{
        int minVal;
        minVal = B;
        if (B < A)
        {
            minVal = B;
         }
         return (minVal);
}
```

Reachability : true

Infection : $A \neq B$

Propagation: ?

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
            minVal = B;
         }
         return (minVal);
}
```

```
int Min (int A, int B)
{
        int minVal;
        minVal = B;
        if (B < A)
        {
            minVal = B;
         }
         return (minVal);
}
```

Reachability : true

Infection : $A \neq B$

Propagation: $(B < A) =$ false

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
            minVal = B;
        }
        return (minVal);
}
```

```
int Min (int A, int B)
{
        int minVal;
        minVal = B;
        if (B < A)
        {
            minVal = B;
        }
        return (minVal);
}
```

Reachability : true

Infection : $A \neq B$

Propagation: $(B < A) =$ false

(A = 5, B = 3) will weakly kill the mutant but not strongly!

Available Mutation Testing Tools

- Mutandis (JavaScript)
- MutateMe (PHP)
- CSAW (C)
- Java
  - PIT (http://pitest.org/quickstart/mutators/)
  - MuJava (Java)
    - Class level and method level mutation operators
  - Javalanche (Java)
    - Invariant and impact analysis
  - MuClipse (Java)
    - Weak mutation, Eclipse plug-in

Mutandis: JavaScript Mutation Testing Tool

- Developed at SALT lab @ UBC
- JavaScript specific mutation operators
  - https://github.com/saltlab/mutandis/
- Leverages static and dynamic program analysis
- Guided mutation generation
  - Error-prone parts of the code
  - Likely to influence the program's output

Mutation Testing in Summary

Pros:

- Great degree of automation
- Providing an interactive test environment
  - Tester can locate and remove errors

Mutation Testing in Summary

Pros:

- Great degree of automation
- Providing an interactive test environment
  - Tester can locate and remove errors

Cons:

- Large computation resources (time and space)
- Human cost of examining large numbers of mutants for possible equivalence

Questions?