Mutation testing

Dr. Aprna Tripathi

Content

Mutation testing history / aim

- Pioneered in the 1970s
- Aim: to locate and expose weaknesses in test suites.

What is Mutation Testing?

- ***Mutation Testing*** is a testing technique that focuses on measuring the adequacy of test cases.
- ***Mutation Testing*** is NOT a testing strategy like *path* or *data-flow* testing. It does not outline test data selection criteria.
- ***Mutation Testing*** should be used in conjunction with traditional testing techniques, not instead of them.

Mutation Testing

- Faults are introduced into the program by creating many versions of the program called *mutants*.
- Each mutant contains a single fault.
- Test cases are applied to the original program and to the mutant program.
- The goal is to cause the mutant program to fail, thus demonstrating the effectiveness of the test case.

Mutation testing overview

- Mutation testing is done by selecting a set of **mutation operators** and then applying them to the source program **one at a time** for each applicable piece of the source code.

- The result of applying one mutation operator to the program is called a ***mutant***.

Mutation testing overview (2)

- If the test suite is able to detect the change in the mutant code, then the **mutant** is said to be ***killed***.

- When this happens, the mutant is considered **dead** and no longer needs to remain in the testing process since the faults represented by that mutant have been detected, and more importantly, it has satisfied its requirement of identifying a **useful test case**.

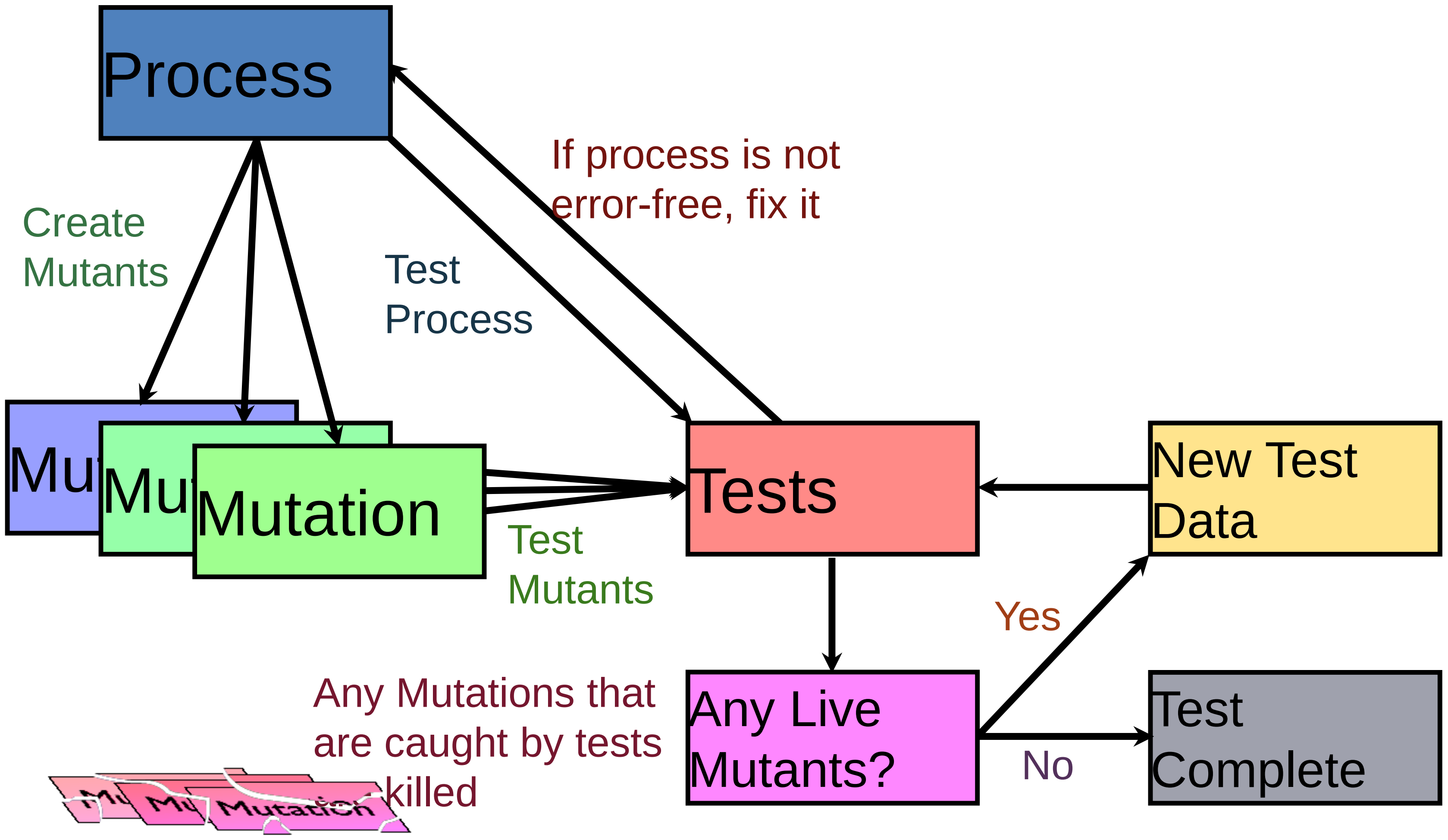
Mutation testing overview (3)

- If a mutant gives the same outcome as the original with the test data, then it is said to be **live**.
- The quality of a set of test cases is measured by the percentage of mutants killed by the test data.

Definition

Mutation testing is a fault-based testing technique that measures the effectiveness of test cases.

The Mutation Process



Test Case Adequacy

- A test case is ***adequate*** if it is useful in detecting faults in a program.
- A test case can be shown to be adequate by finding at least one mutant program that generates a different output than does the original program for that test case.
- If the original program and all mutant programs generate the same output, the test case is ***inadequate***.

Mutant Programs

- Mutation testing involves the creation of a set of mutant programs of the program being tested.
- Each mutant differs from the original program by one *mutation*.
- A *mutation* is a single syntactic change that is made to a program statement.

Example of a Program Mutation

```
1 int max(int x, int y)
2 {
3 int mx = x;
4 if (x > y)
5      mx = x;
6 else
7      mx = y;
8 return mx;
9 }
```

```
1 int max(int x, int y)
2 {
3 int mx = x;
4 if (x < y)
5      mx = x;
6 else
7      mx = y;
8 return mx;
9 }
```

Primary Mutants

- When the mutants are single modifications of the initial program using some operators, they are called primary mutants
- For example

```
.... if (a > b)
x = x + y;
else x = y;
printf("%d", x); ...
```

- Following are the possible mutants for above example:
- M1: x = x – y; M2: x = x / y;
- M3: x = x + 1; M4: printf("%d", y);

Secondary Mutants

- when multiple levels of mutation are applied on the initial program, this class of mutants is called secondary mutants.
- For example program:
- if (a < b) c = a;
- Now, mutants for this code may be as follows:
- M1 : if (a <= b-1) c = a;
- M2: if (a+1 <= b) c = a;
- M3: if (a == b) c = a+1;
- In this case, it is very difficult to identify the initial program from its mutants.

Categories of Mutation Operators

- **Operand Replacement Operators:**
  - Replace a single operand with another operand or constant. *E.g.,*
    - if (5 > y) Replacing x by constant 5.
    - if (x > 5) Replacing y by constant 5.
    - if (y > x) Replacing x and y with each other.
  - *E.g.,* if all operators are *{+,-,*,**,/}* then the following expression *a = b * (c - d)* will generate 8 mutants:
    - 4 by replacing *
    - 4 by replacing -.

Categories of Mutation Operators

- **Expression Modification Operators:**
  - Replace an operator or insert new operators. *E.g.,*
    - if (x == y)
    - if (x >= y)   Replacing == by >=.
    - if (x == ++y)   Inserting ++.

Categories of Mutation Operators

- **Statement Modification Operators:**
  - *E.g.,*
    - Delete the else part of the if-else statement.
    - Delete the entire if-else statement.
    - Replace line 3 by a return statement.

Mutation Score

- The quality of test data is measured by *mutation scores*.
- A mutation score is the percentage of non-equivalent mutants killed.
- A set of test data is *mutation-adequate* if its mutation score is 100%.
- Mutation score is calculated as:

$$MS(P,T) = \frac{M_k}{M_t - M_q}$$

where P is the test program;
T is the set of test data;
M_k is the number of mutants killed by T;
M_t is the total number of mutants generated for the program;
M_q is the number of equivalent mutants for the program.

Example

The original code:

```
if(a && b) c = 1;
else       c = 0;
```

We replace '&&' with '||' and produce the following mutant:

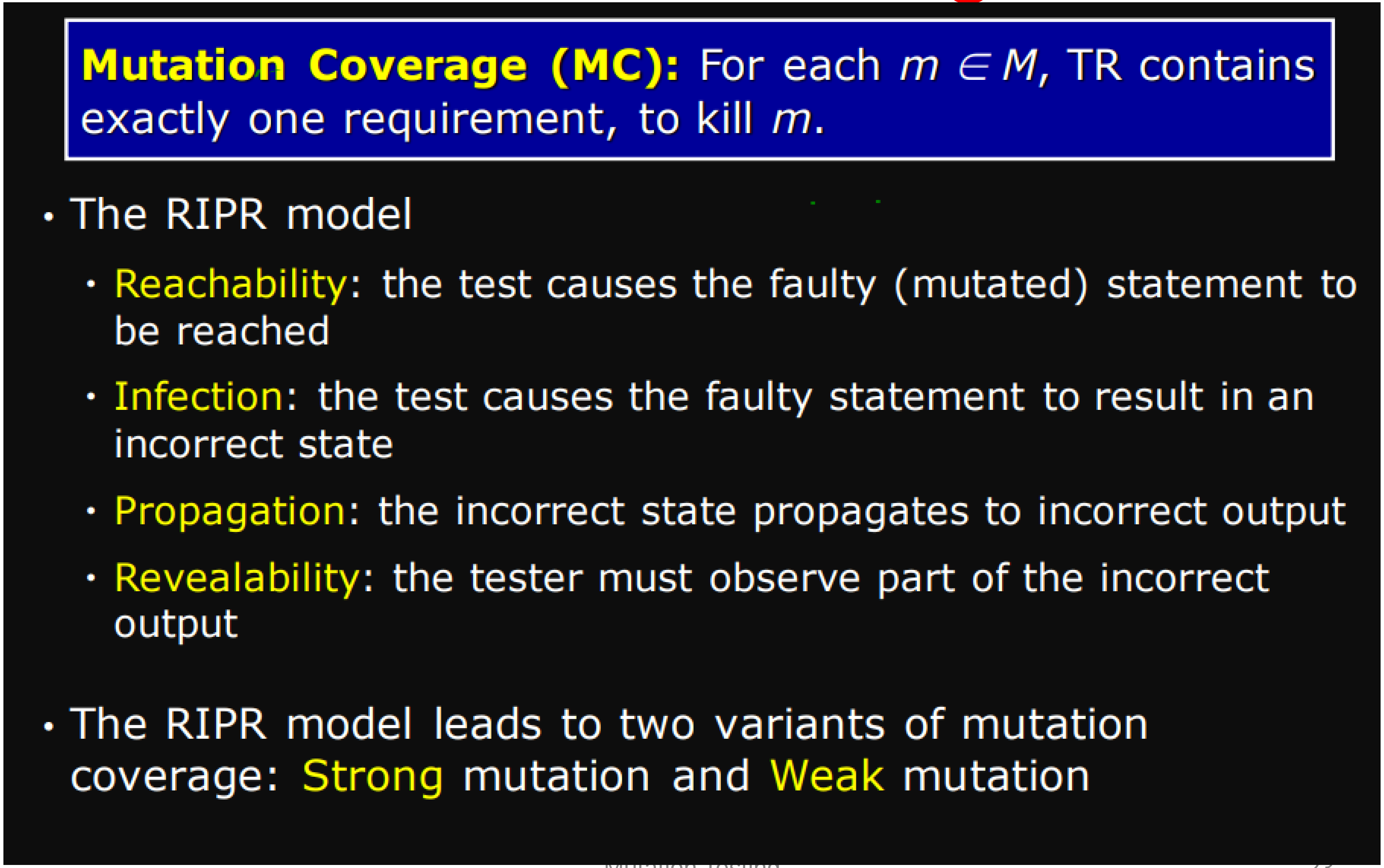
```
if(a || b) c = 1;
else       c = 0;
```

Now, for the test to kill this mutant, the following condition should be met:

1) Test input data should cause different program states for the mutant and the original program. For example, a test with a=1 and b=0 would do this (enough for ***Weak mutation testing***).

2) The value of 'c' should be propagated to the program's output and checked by the test (necessary for ***Strong mutation testing***).

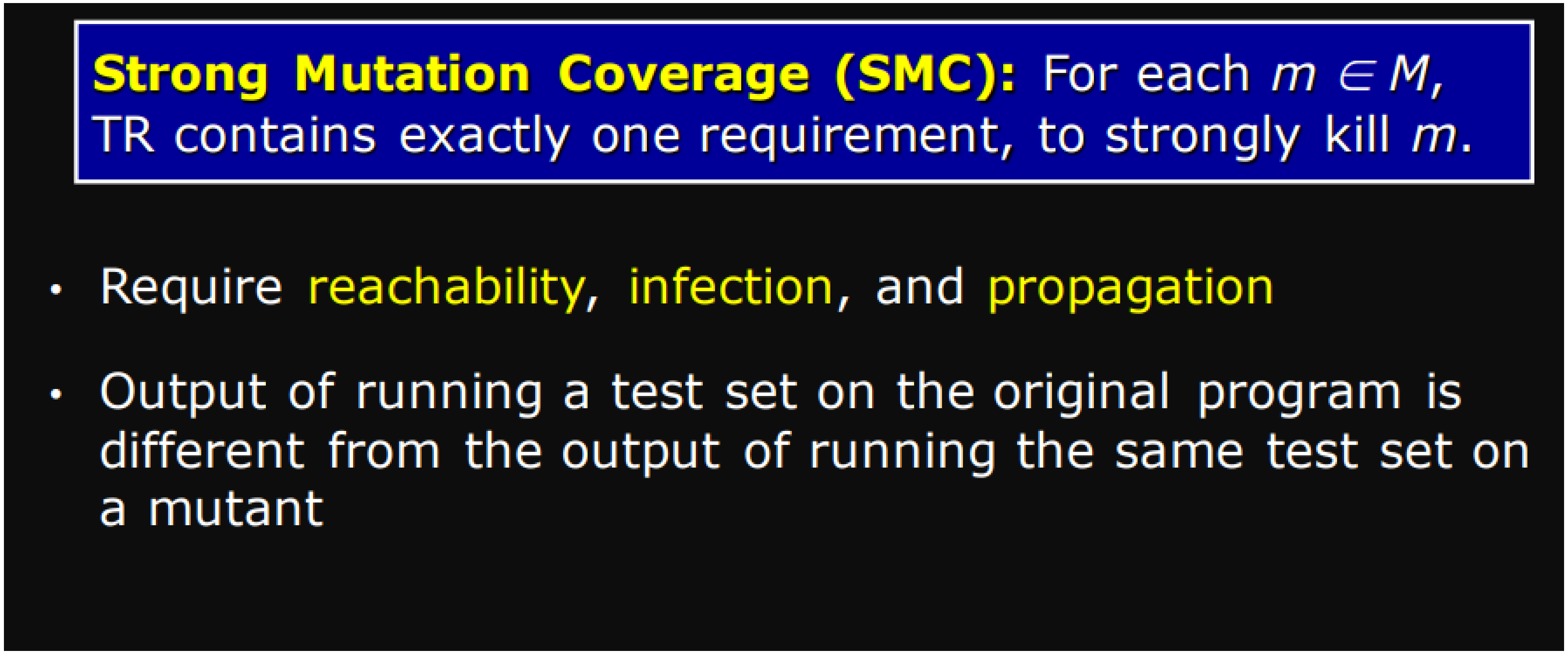
Mutation coverage



Mutation Coverage (MC): For each $m \in M$, TR contains exactly one requirement, to kill m.

- The RIPR model
  - Reachability: the test causes the faulty (mutated) statement to be reached
  - Infection: the test causes the faulty statement to result in an incorrect state
  - Propagation: the incorrect state propagates to incorrect output
  - Revealability: the tester must observe part of the incorrect output
- The RIPR model leads to two variants of mutation coverage: Strong mutation and Weak mutation

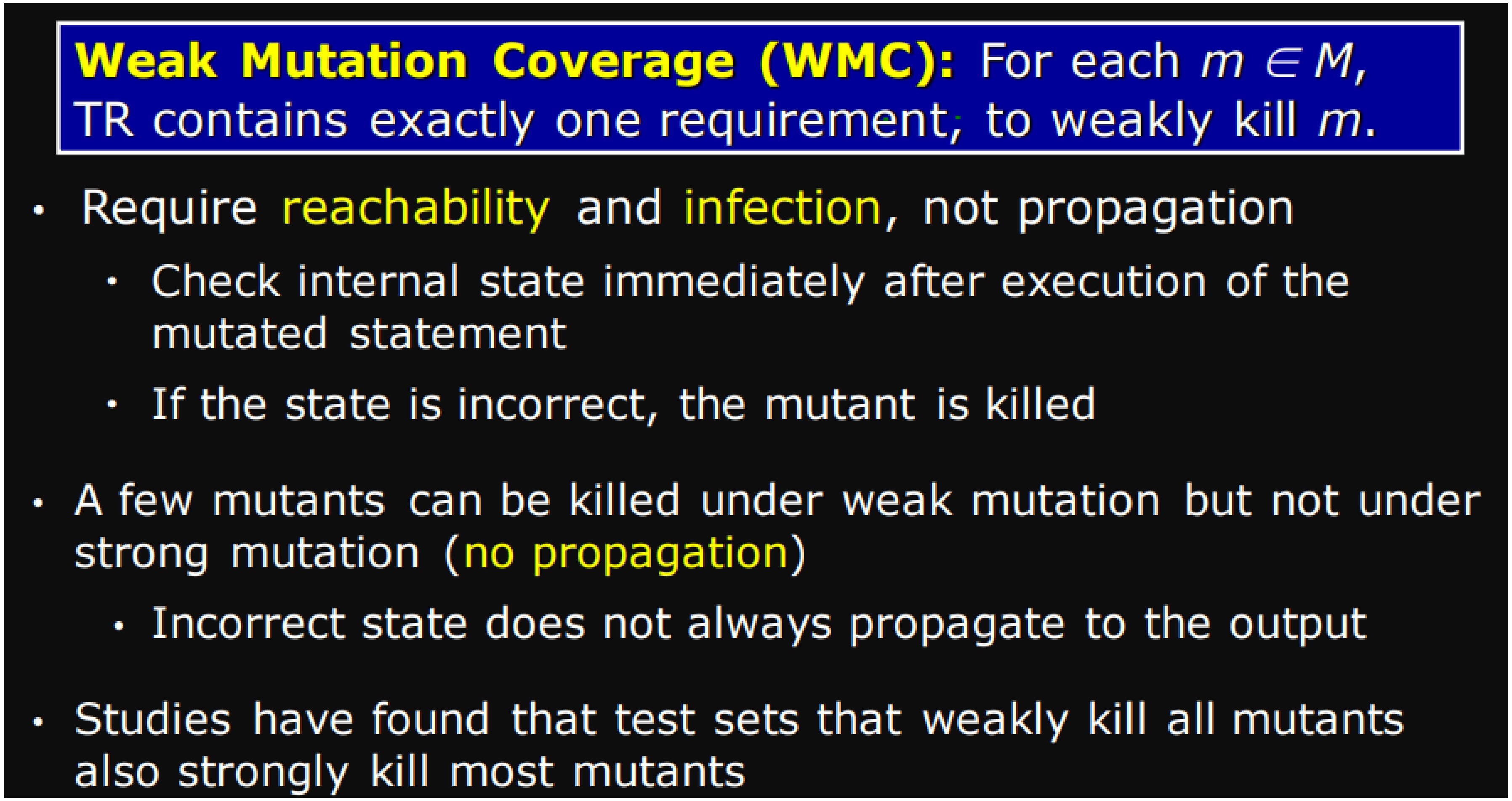
Strong Mutation coverage



Strong Mutation Coverage (SMC): For each $m \in M$, TR contains exactly one requirement, to strongly kill m.

- Require reachability, infection, and propagation
- Output of running a test set on the original program is different from the output of running the same test set on a mutant

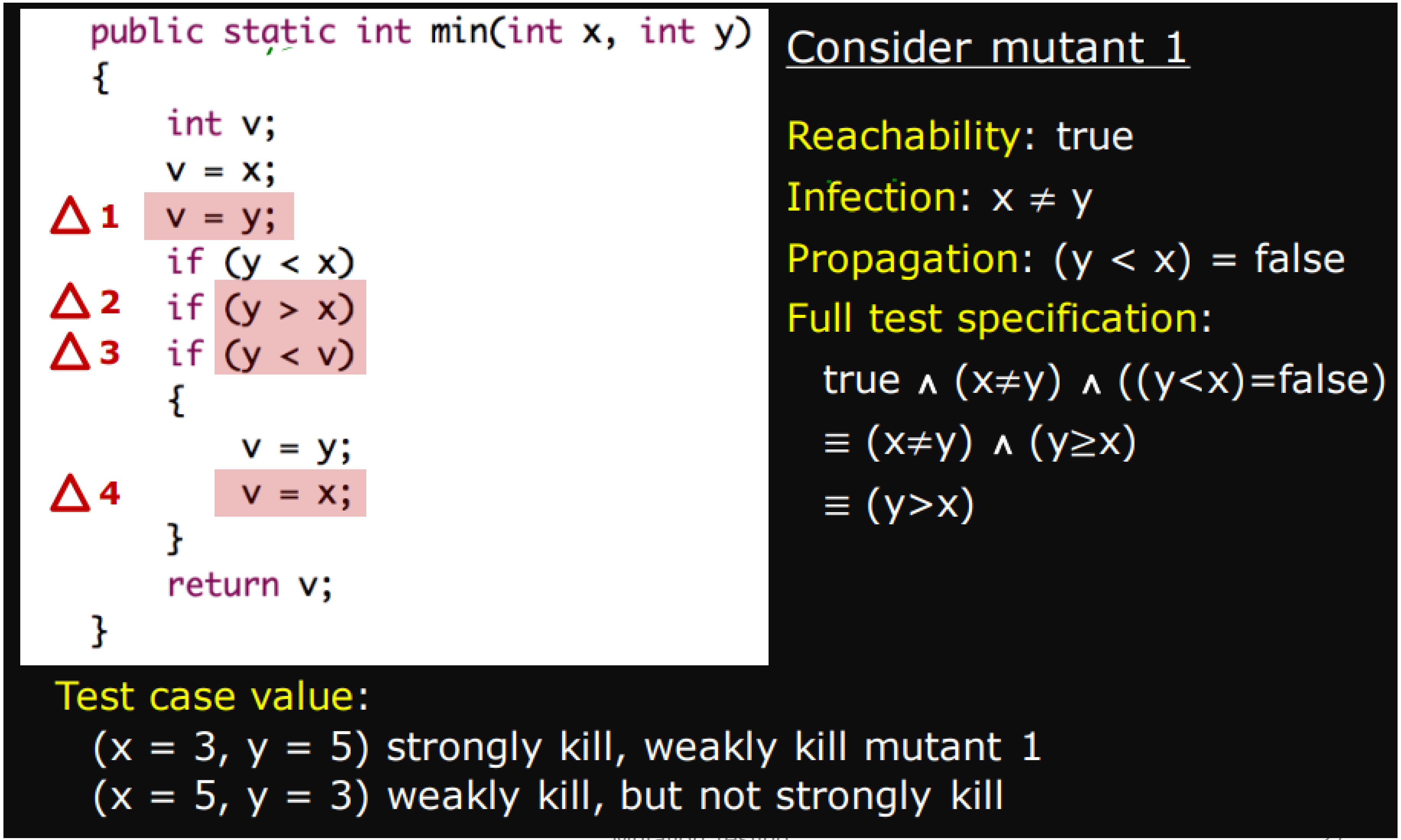
Weak Mutation Coverage



Weak Mutation Coverage (WMC): For each $m \in M$, TR contains exactly one requirement, to weakly kill m.

- Require reachability and infection, not propagation
  - Check internal state immediately after execution of the mutated statement
  - If the state is incorrect, the mutant is killed
- A few mutants can be killed under weak mutation but not under strong mutation (no propagation)
  - Incorrect state does not always propagate to the output
- Studies have found that test sets that weakly kill all mutants also strongly kill most mutants

Example 1



```
    public static int min(int x, int y)
    {
        int v;
        v = x;
△1      v = y;
        if (y < x)
△2      if (y > x)
△3      if (y < v)
        {
            v = y;
△4          v = x;
        }
        return v;
    }
```

Consider mutant 1

Reachability: true

Infection: $x \neq y$

Propagation: $(y < x) = \text{false}$

Full test specification:

$\text{true} \wedge (x \neq y) \wedge ((y < x) = \text{false})$

$\equiv (x \neq y) \wedge (y \geq x)$

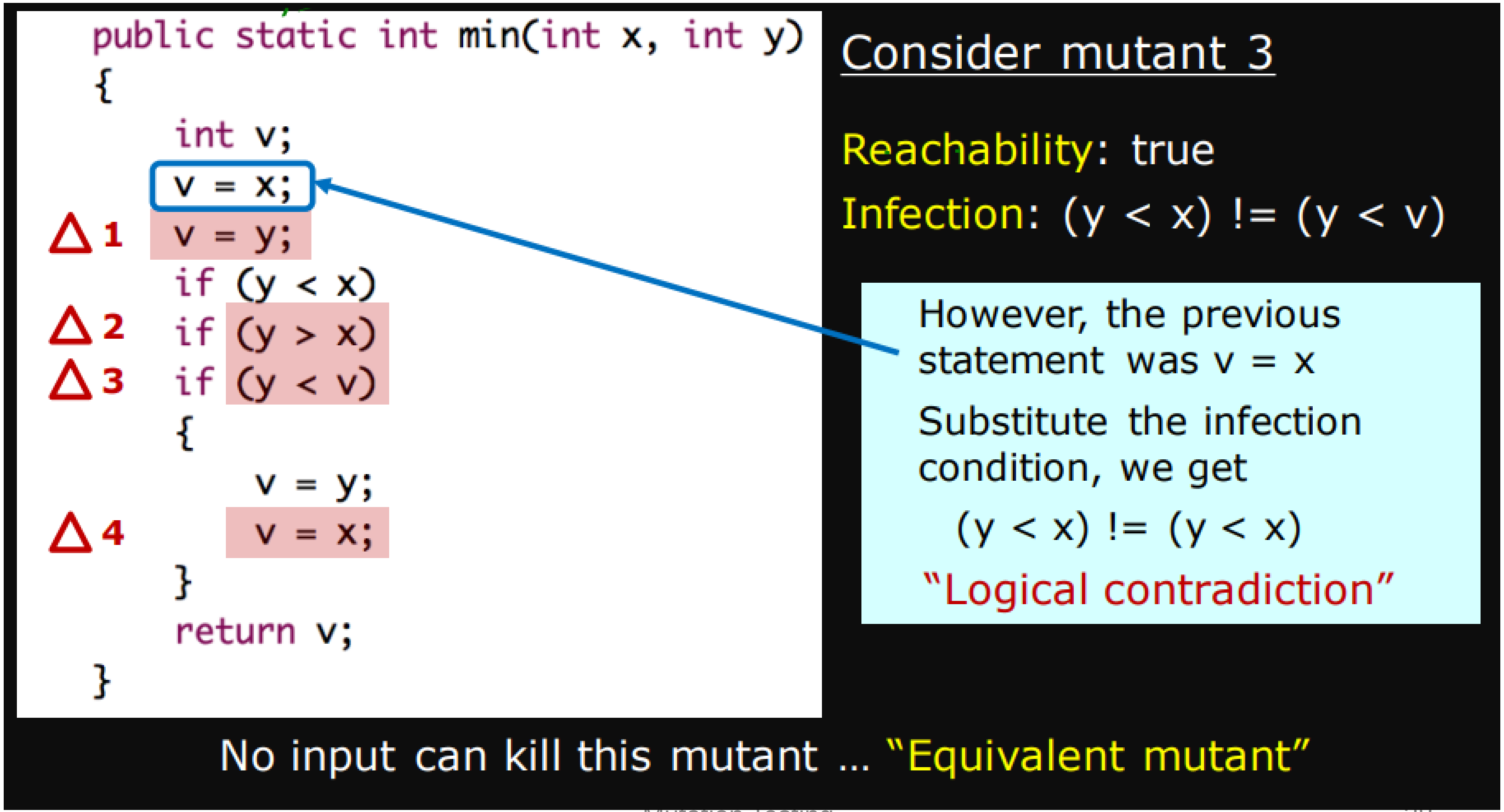
$\equiv (y > x)$

Test case value:

$(x = 3, y = 5)$ strongly kill, weakly kill mutant 1

$(x = 5, y = 3)$ weakly kill, but not strongly kill

Example 2



```
    public static int min(int x, int y)
    {
        int v;
        v = x;
Δ 1     v = y;
        if (y < x)
Δ 2     if (y > x)
Δ 3     if (y < v)
        {
            v = y;
Δ 4         v = x;
        }
        return v;
    }
```

Consider mutant 3

Reachability: true

Infection: (y < x) != (y < v)

However, the previous statement was v = x

Substitute the infection condition, we get

(y < x) != (y < x)

"Logical contradiction"

No input can kill this mutant … "Equivalent mutant"

Strong Vs Weak

```
  public static boolean isEven(int x)
  {
      if (x < 0)
          x = 0 - x;
△1        x = 0;
      if (float)(x/2) == ((float)x)/2.0
          return true;
      else
          return false;
  }
```

Reachability: $x < 0$

Infection: $x \neq 0$

Given a test (x = -6)

Kills the mutant under weak mutation, but not under strong mutation

Propagation:

$$((float)((\underline{0-x})/2)==((float)(\underline{0-x}))/2.0) \mathrel{!=} ((float)(\underline{0}/2)==((float)\underline{0})/2.0)$$

The only value of x that will satisfy this condition is x must not be even

To strongly kill, propagation requires x must be an odd and negative integer

Weak mutation testing

- Weak mutation is an approximation technique that compares the internal states of the mutant and original program immediately after execution of the mutated portion of the program.

- It requires much less computing power to ensure that the test suite satisfies weak mutation testing.

Strong mutation testing

- When strong mutation testing is executing, the values produced by mutated code are passing further and changes (or not) the output of the module or subroutine.
- If the mutant is not killed, then considering whether the test data could be enhanced to kill this live mutant or whether no test data could ever be constructed to kill the mutant.

- Strong mutation testing requires much more computing power.

Selective mutation testing

Selective mutation testing is based on mutants generation only by selected set of mutation operators. Generated by specific set of operators, mutants are more different and represent more different code mistakes.

Traditional mutation operators

1. Statement deletion
2. Boolean expression changes (*true* and *false*)
3. Arithmetic operation changes (+ and *, - and /)
4. Boolean operator changes (> and >=, == and <=)
5. Declared in the same scope variables

   changes (variable types should be the same)

Class-level mutation operators

1. Polymorphic
2. Method Overloading
3. Method Overriding/Hiding in Inheritance
4. Field Variable Hiding in Inheritance
5. Information Hiding (Access Control)
6. Static/Dynamic States of Objects
7. Exception Handling

1. Polymorphic mutations

```
class T { … }
interface K { … }
class S extends T implements K { … }
class U extends S { … }
```

The original code:

```
S s = new S();
```

Mutants:

```
a) T s = new S();
b) K s = new S();
c) S s = new T();
d) S s = new U();
```

2. Method overloading mutations

```
1.public LogMessage(int level,
 String logKey, Object[]inserts)
 {…}

2.public LogMessage(int level,
 String logKey, Object insert) {…}
```

Mutant:

```
public LogMessage(String logKey,
  int level, Object[]inserts) {…}
```

3. Method Overriding/Hiding in Inheritance

```
public class Animal {
    public void speak(){..}
    public void bite(){..}
}
public class Cat extends Animal {
    public void speak() {..}
    public void bite(){..}
}
```

Mutant:

```
public class Cat extends Animal {
    //public void speak() {..}
    public void bite(){..}
}
```

4. Field Variable Hiding in Inheritance

```
public class Animal {
  public String kind;
  public int legCount;
}
public class Cat extends Animal {
  public String kind;
  public int legCount;
}
```

Mutant:

```
public class Cat extends Animal {
  public String kind;
  //public int legCount;
}
```

Possible mutants

5. Information Hiding (Access Control)

The original code:

```
protected Address address;
```

Mutants:

```
a)public Address address;
b)private Address address;
c)Address address; //default (no
 access modifier defined)
```

6. Static/Dynamic States of Objects

The original code:

```
public static int VALUE = 100;
private String s;
```

Mutants:

```
a) public int VALUE = 100;
   //static is removed.
b) private static String s;
   //static is added.
```

7. Exception Handling

The original code:

```
String formatMsg(LogMessage msg){
  …
  try { …
  } catch(MissingResourceException mre){…
  }
  …
}
```

Mutant:

```
String formatMsg(LogMessage msg)
throws MissingResourceException {
…   //try-catch block removed
}
```

Questions

1. Aim of the mutation testing?
2. Mutation testing types?
3. Differences between strong and weak mutation testing?
4. What kind of mutation operators do you know?

References

1. Mutation Testing on wikipedia
2. Class Mutation: Mutation Testing for Object-Oriented Programs
3. A Practical System for Mutation Testing: Help for the Common Programmer
4. .Net mutation testing
5. An Empirical Evaluation of Weak Mutation