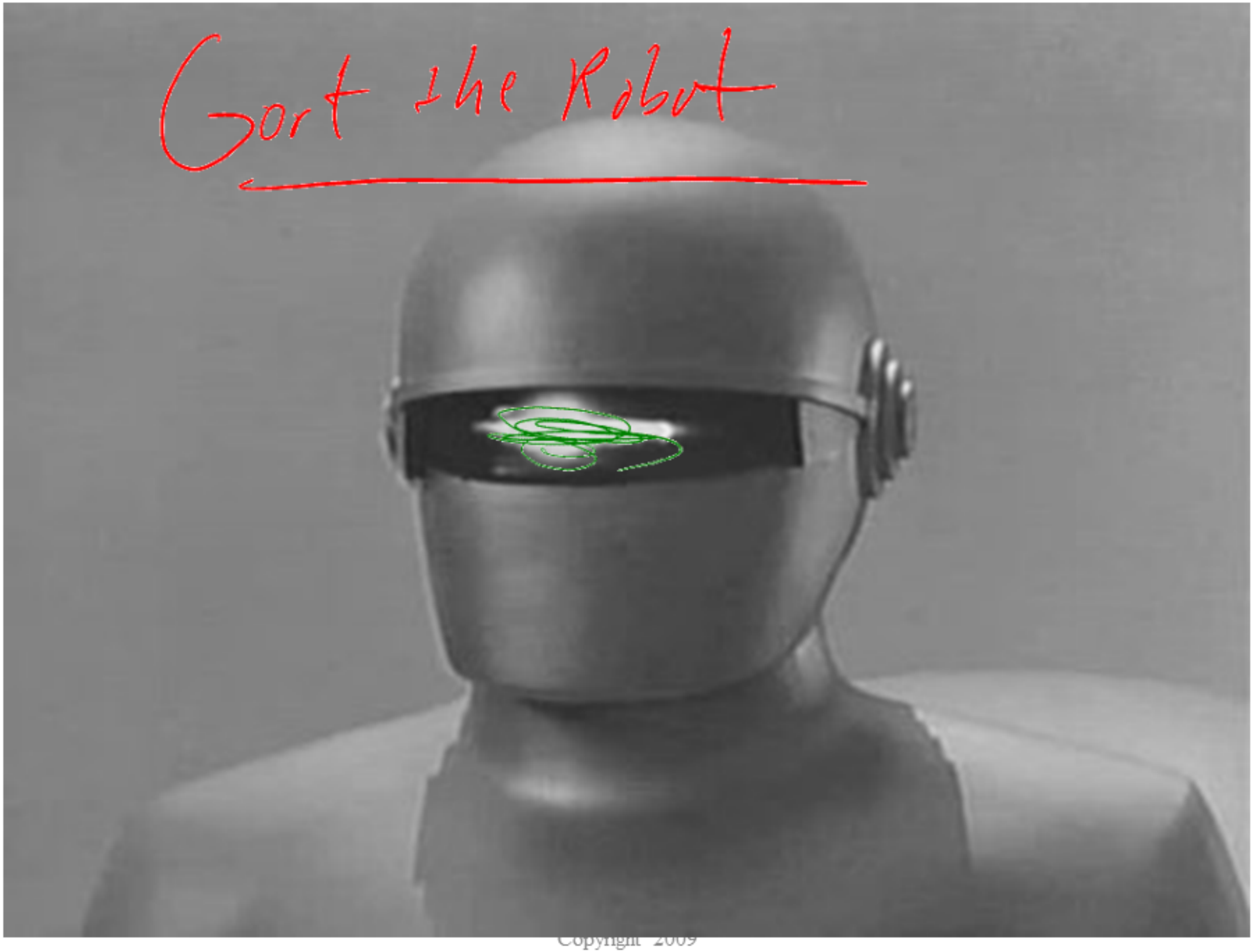
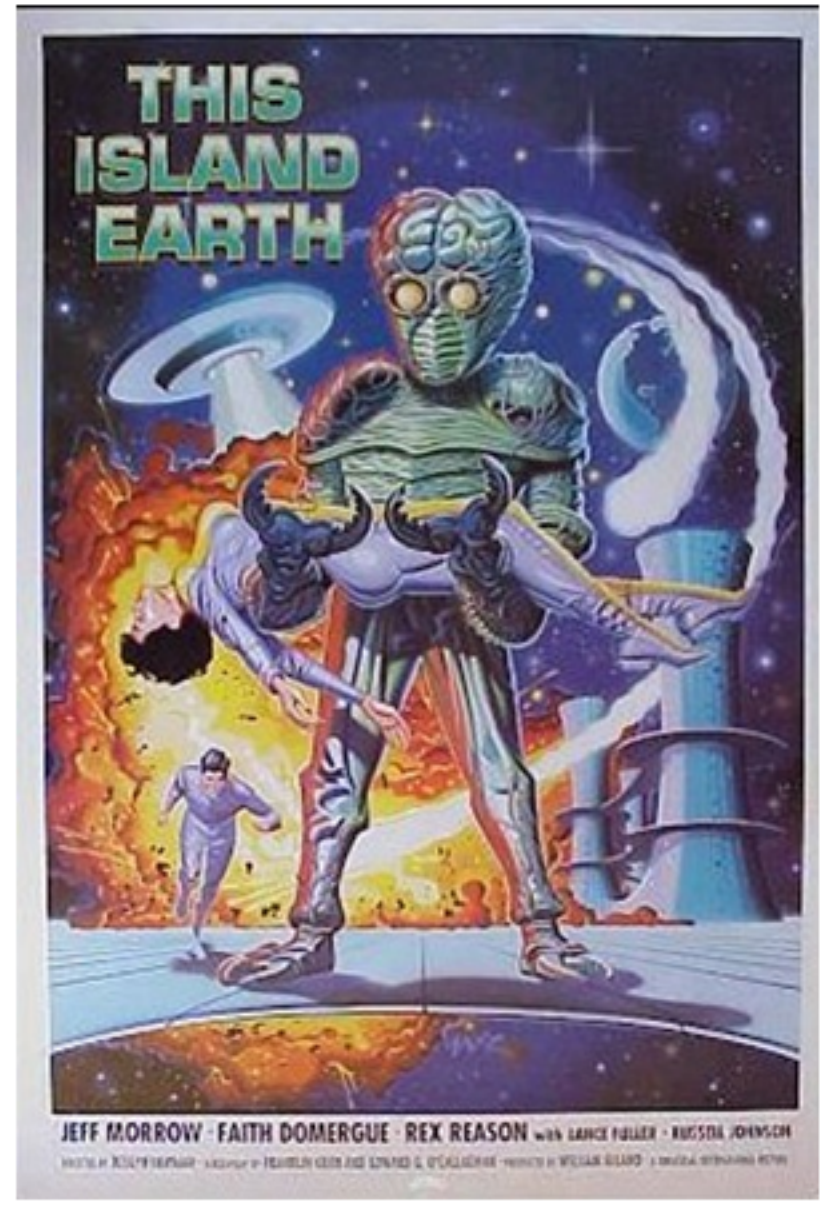


SE2832 Introduction to Software Verification

- Objectives
  - Define the concept of a ground string.
  - Define the term mutation operator.
  - Define the term mutant as it refers to a mutation operator.
  - Define mutation score as it is relative to mutation testing.
  - Explain the concept of a dead mutation.
  - Explain the concept of a stillborn mutant.
  - Draw a flowchart showing the process used to perform mutation testing.
  - List examples of program-level mutation operators.

A Mutant

- Original Program P
- Mutant P' of P
  - A program similar to P
  - P' differs from P by a single mutation
  - Each kind of mutation corresponds to a typical error programmers usually make
    - "Off--by--one", spelling, typos, etc..
-



Definition

- Ground string
  - A string that is in a grammar
- Mutation Operator
  - A rule that specifies syntactic variations of strings generated from a grammar
- Mutant
  - The result of one application of a mutation operator
- Mutation coverage
  - For each mutant m € M, TR contains exactly one requirement, to kill m.

Mutation testing

- Mutation testing introduces faults into software by creating many different versions of the program
  - Each version has one very small change (which introduces a fault) compared to the actual implementation.
- Goal
  - See if the test cases can detect the new fault

Mutation Operators

- Rules applied to a program to create mutants
  - Example: replace each operator with another operator
  - Replace each logical operator with another logical operator
  - Replace each comparison operator with another equivalent operator

Example

```
Boolean isGreaterThanSum(int a, int b, int c)
{   Boolean retVal = false;
    If (a > (b + c))
    {
        retVal = true;
    } Return retVal;
}
```

retval != false

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)

MuJava

Java Mutation Operators

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

Mutation Operator Descriptions

1 Arithmetic Operators

The Java programming language supports five arithmetic operators for all floating-point and integer numbers; (1) +, (2) -, (3) *, (4) /, and (5) %. These operators are all binary. However, both + and - have unary versions. Four short-cut arithmetic operators are defined; (1) op++, (2) ++op, (3) op−−, and (4) −−op.

- **AOR**$_B$: Arithmetic Operator Replacement
  Replace basic binary arithmetic operators with other binary arithmetic operators.
- **AOR**$_U$: Arithmetic Operator Replacement
  Replace basic unary arithmetic operators with other unary arithmetic operators.
- **AOR**$_S$: Arithmetic Operator Replacement
  Replace short-cut arithmetic operators with other unary arithmetic operators.
- **AOI**$_U$: Arithmetic Operator Insertion
  Insert basic unary arithmetic operators.
- **AOI**$_S$: Arithmetic Operator Insertion
  Insert short-cut arithmetic operators.
- **AOD**$_U$: Arithmetic Operator Deletion
  Delete basic unary arithmetic operators.
- **AOD**$_S$: Arithmetic Operator Deletion
  Delete short-cut arithmetic operators.

2 Relational Operators

What can happen to a mutant?

- Stillborn Mutant

  A mutation from which the code is not compilable.

- Killed Mutant

  A mutation which is detected by the test suite.

- Live Mutant

  A mutation not killed by our test suite.

- Equivalent Mutant

  A mutation in which the result is identical.

- Stubborn Mutant

  A mutation which results in different execution but correct.

How does this impact out test cases

```
public int sum (int a, int b)
{
    return a+b;
}
```

[illegible] test case.

a-b ← kill the mutant

a*b ← kill the mutant.

(0,0) ⇒ 0

(2,2) ⇒ 4

(4,2) ⇒ 6

Equivalent Mutant

```
int index=0;
while (...)
{
    . . .;
    index++;
    if (index==10)
        break;
}
```

index < 100000000

if (index >= 10)

Equivalent Mutation

Stubborn mutation

```
int dosomething(int x)
{  int sum=0;
   while(x)
   {
      if(x!=0)
      {
         sum+=x;
      }
      x--;
   }
} return sum;
```

Stubborn Mutation

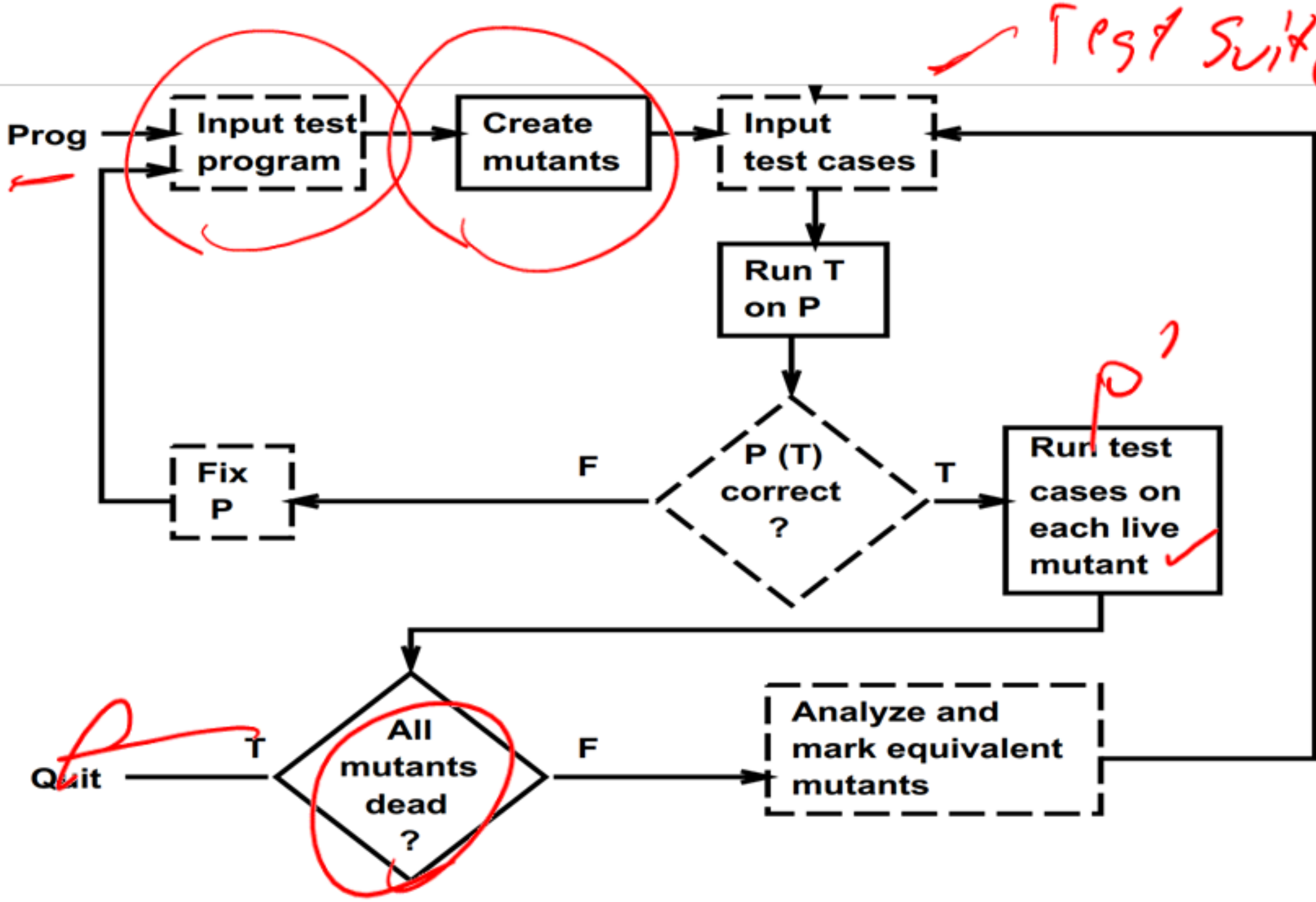
50.2 < 50/100

Mutation Score

$$MS = \frac{\#ofKilledMutants}{\#ofMutants}$$

# of Killed Mutants / Number of Generated Mutants

Mutation testing Complexity



Premise of mutation testing

- In practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault.

Limitations and Problems

- Difficult to identify equivalent mutants
  - Same behavior as the original program
  - They can not be killed
  - Important but very difficult to identify them
- Difficult to kill stubborn non--equivalent mutants
  - May be non--equivalent butt still very difficult to kill
- Computational Cost of Mutation Testing
  - Big number of mutation operators that create a vast number of mutants (not always relevant ones)
  - Typically there is a large number of mutants for even small software units
  - Computationally expensive to test all these mutants against all the test cases
  - Mutation analysis requires large amounts of computation
- Manual Labor when using Mutation Testing
  - Manual equivalent mutant detection is quite tedious
  - Developing mutation adequate test cases can be very labor--intensive

Mutation Tools