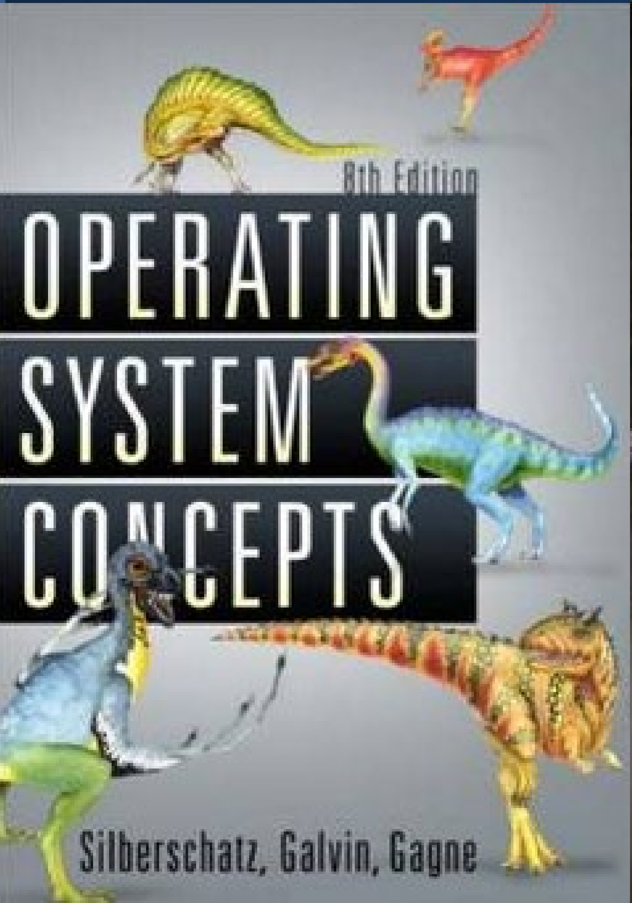
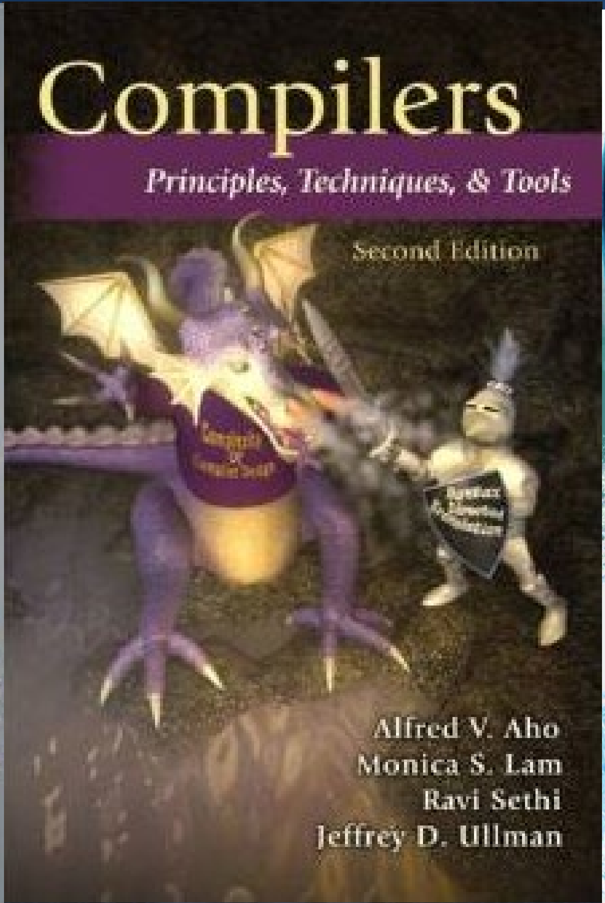
# Necessity of Automated SW Testing
# - Fight the Complexity of SW

Moonzoo Kim

KAIST





# SW Testing is Very Complex and Difficult Task

**The ratio of time spent for developing and testing SW products is 1:3**

"… We have as many testers as we have developers. Testers basically test all the time, and developers basically are involved in the testing process about half the time…"

**The ratio of program code written for SW products and test harness is 1:3**

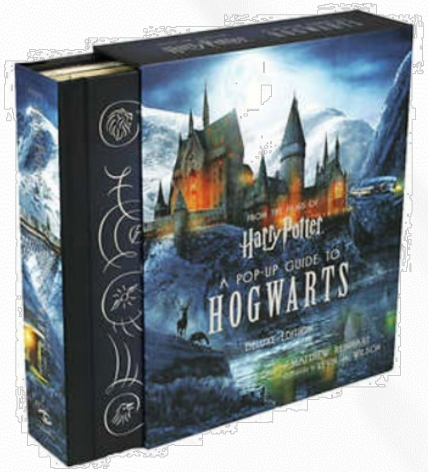
"…The test cases are unbelievably expensive; in fact, there's more lines of code in the test harness than there is in the program itself. Often that's a ratio of about three to one."

*Object-Oriented Programming, Systems Languages, and Applications, Seattle, Washington, 2002*

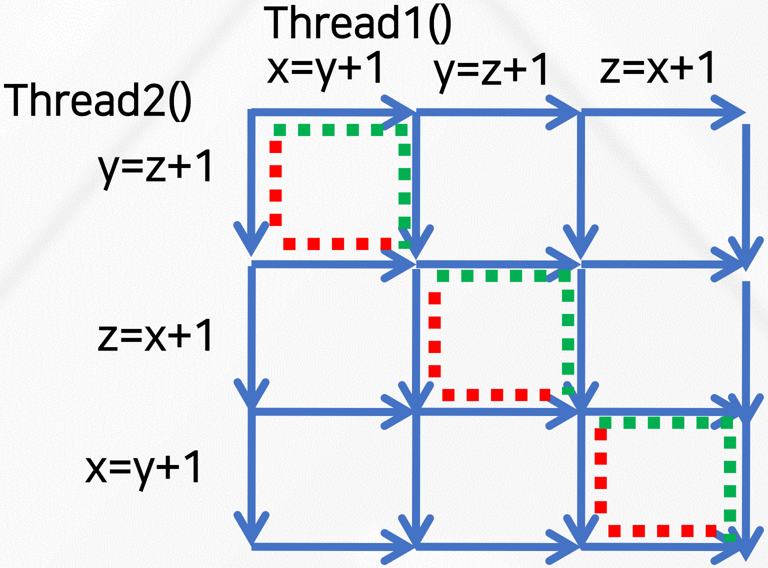
# Summary: What is (the essense of) Software?

1. Software = a large set of unique executions
2. SW testing = to find an execution that violates a given requirement among the large set
   - A human brain is poor at enumerating all executions of a target SW, but computer is good at the task
3. Automated SW testing
   = to enumerate and analyze the executions of
   SW systematically (and exhaustively if possible)

# Static SW Code vs. Dynamic SW Executions

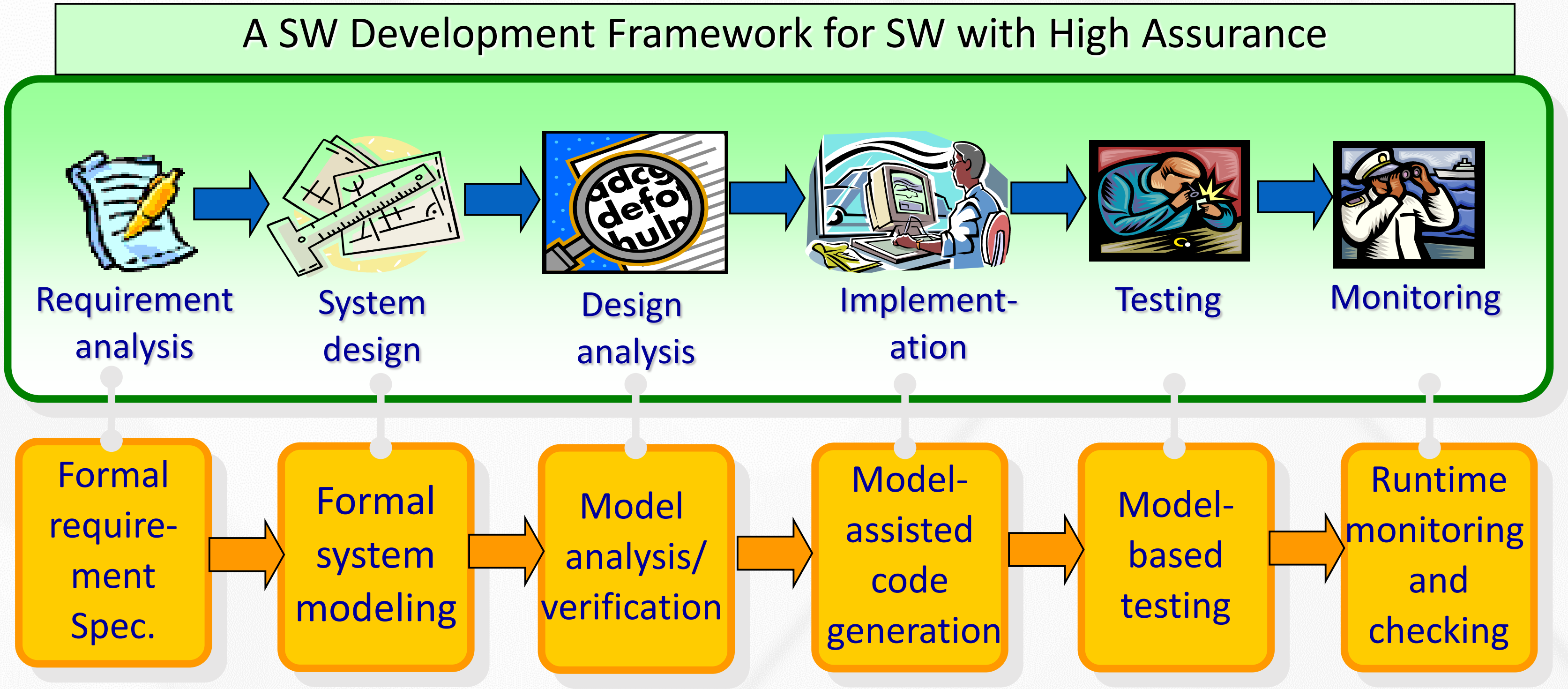


```
int x=0, y=0, z =0;
void Thread1()
{x=y+1; y=z+1; z= x+1;}
void Thread2()
{y=z+1; z=x+1; x=y+1;}
```



# Software Development Cycle

- A practical end-to-end formal framework for software development



A SW Development Framework for SW with High Assurance

Requirement analysis → System design → Design analysis → Implementation → Testing → Monitoring

Formal requirement Spec. → Formal system modeling → Model analysis/ verification → Model-assisted code generation → Model-based testing → Runtime monitoring and checking

# Software v.s. Magic Circle (마법진)

- Written by a software developers line by line
- Requires programming expertise
- SW executes complicated tasks which are far more complex than the code itself
- The software often behaves in unpredicted ways and crash occurs

- Written by a human magician line by line
- Requires magic spell knowledge
- Summoned monsters are far more powerful than the magic spell itself
- The summoned demon is often uncontrollable and disaster occurs

네이버 웹툰 "그 판타지 세계에서 사는법" by 촌장

# Requirement Specification Problems

- Ambiguity
  - Expression does not have unique meaning, but can be interpreted as several different meaning.
    - Ex. For a natural number input, do X
      - What if a 0 is given? Is 0 a natural number?
- Incompleteness
  - Relevant issues are not addressed , e.g. what to do when user errors occur or software faults show.
    - Ex. For a positive integer input, do Y
      - What if a negative input is given?
- Inconsistency
  - Contradictory requirements in different parts of the specification.
    - Ex. For a non-negative input, execute Z, and for a non-positive input, do not execute Z
      - What if 0 is given?

# Example (retail chain management software)

- If the sales for the current month are below the target sales, then a report is to be printed,
    - unless the difference between target sales and actual sales is less than half of the difference between target sales and actual sales in the previous month
    - or if the difference between target sales and actual sales for the current month is under 5 percent.

# Example 2: Leap year (윤년) detection

- The Februray of a leap year has 29th day (i.e., an extra day).
- Given year, print “Leap year” if the following conditions hold:

a) if a year is divisible by 4, it is a leap year. Otherwise, it is not.

b) if a year is divisible by both 4 and 100, it is not a leap year.

c) if a year is divisible by 400, it is a leap year.

- 예시:

```
isLeapYear(2008)    Leap year
isLeapYear(2100)    Not a leap year
isLeapYear(2021)    Not a leap year
isLeapYear(2000)    Leap year
```

# Ex. Testing a Triangle Decision Program

Input : Read three integer values from the command line. The three values represent the length of the sides of a triangle.

Output : Tell whether the triangle is

- Scalene (부등변삼각형 ) : no two sides are equal
- Isosceles (이등변삼각형) : exactly two sides are equal
- Equilateral(정삼각형) : all sides are equal

Create a Set of Test Cases for this program

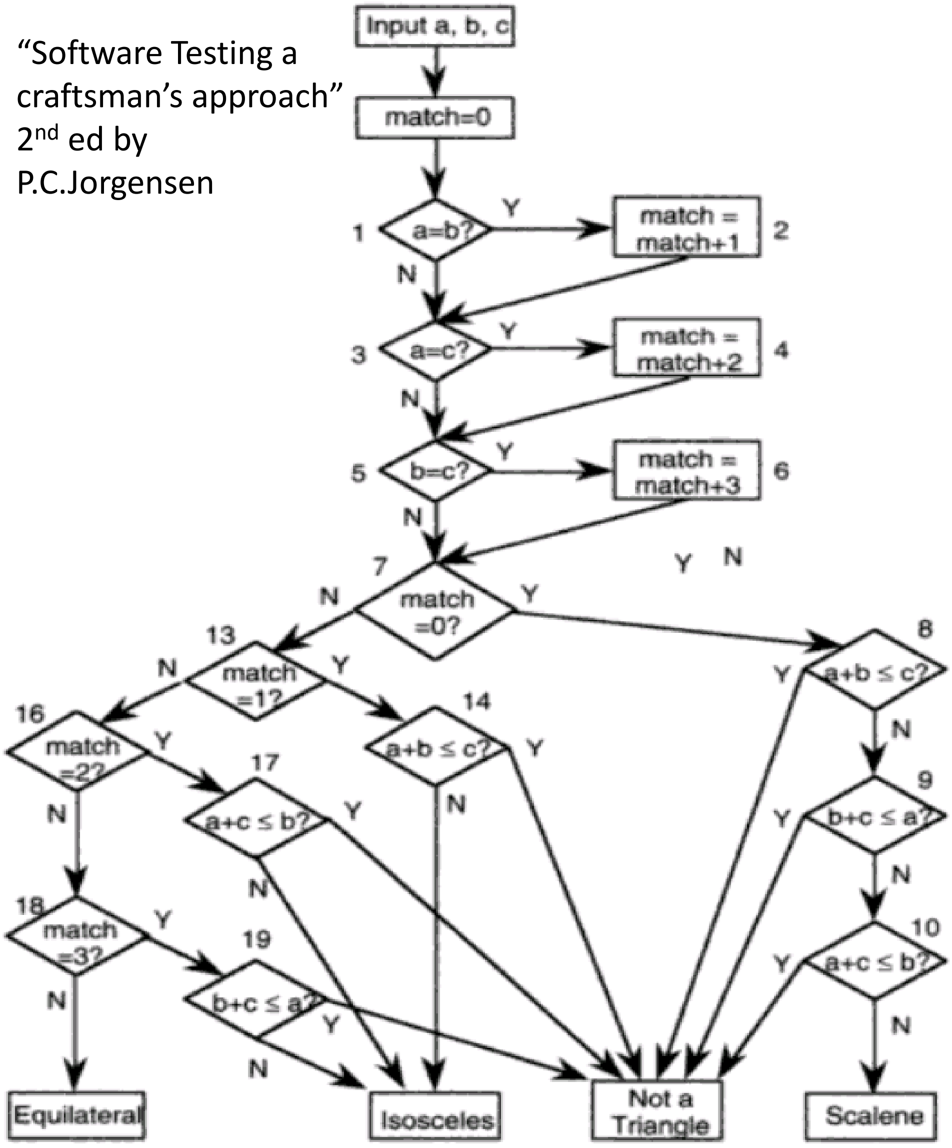
(3,4,5), (2,2,1), (1,1,1) ?

# Precondition (Input Validity) Check

- Condition 1: $a > 0, b > 0, c > 0$
- Condition 2: $a < b + c$
  - Ex. (4, 2, 1) is an invalid triangle
  - Permutation of the above condition
    - $a < b +c$
    - $b < a + c$
    - $c < a + b$
- What if b + c exceeds $2^{32}$ (i.e. overflow)?
  - long v.s. int v.s. short. v.s. char
- Developers often fail to consider implicit preconditions
  - Cause of many hard-to-find bugs

# Test Cases for the Triangle Decision

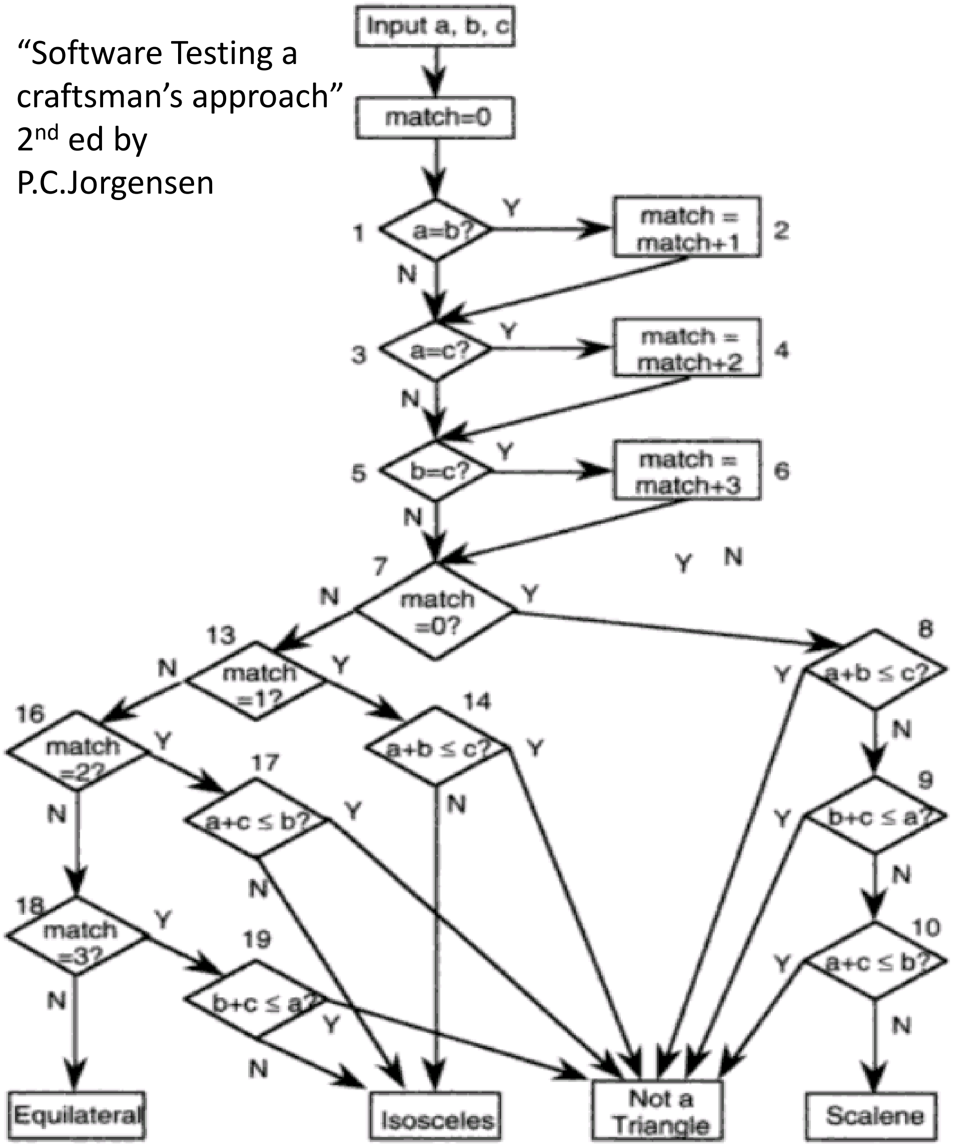
```
int triangle(int a, int b, int c) {
    int match=0, result=-1;
1:   if(a==b) match=match+1;
3:   if(a==c) match=match+2;
5:   if(b==c) match=match+3;
7:   if(match==0) {
8:       if( a+b <= c) result=2;
9:       else if( b+c <= a) result=2;
10:      else if(a+c <= b) result =2;
         else result=3;
       } else {
13:      if(match == 1) {
14:          if(a+b <= c) result =2;
             else result=1;
         } else {
16:          if(match ==2) {
17:              if(a+c <=b) result = 2;
                 else result=1;
             } else {
18:              if(match==3) {
19:                  if(b+c <= a) result=2;
                     else result=1;
                 } else result = 0;
         } }}
return result; }
```



“Software Testing a craftsman’s approach” 2nd ed by P.C.Jorgensen

# Test Cases for the Triangle Decision

- # of test cases required?
  ① 4
  ② 11
  ③ 50
  ④ 100
- # of feasible unique execution paths?
  - 11
- The goal of testing
  - Generate 11 test cases that exercise the 11 unique execution paths



"Software Testing a craftsman's approach" 2nd ed by P.C.Jorgensen

# Test Cases for the Triangle Decision

a,b,c = 1,1,1:match=6:result=0:p1

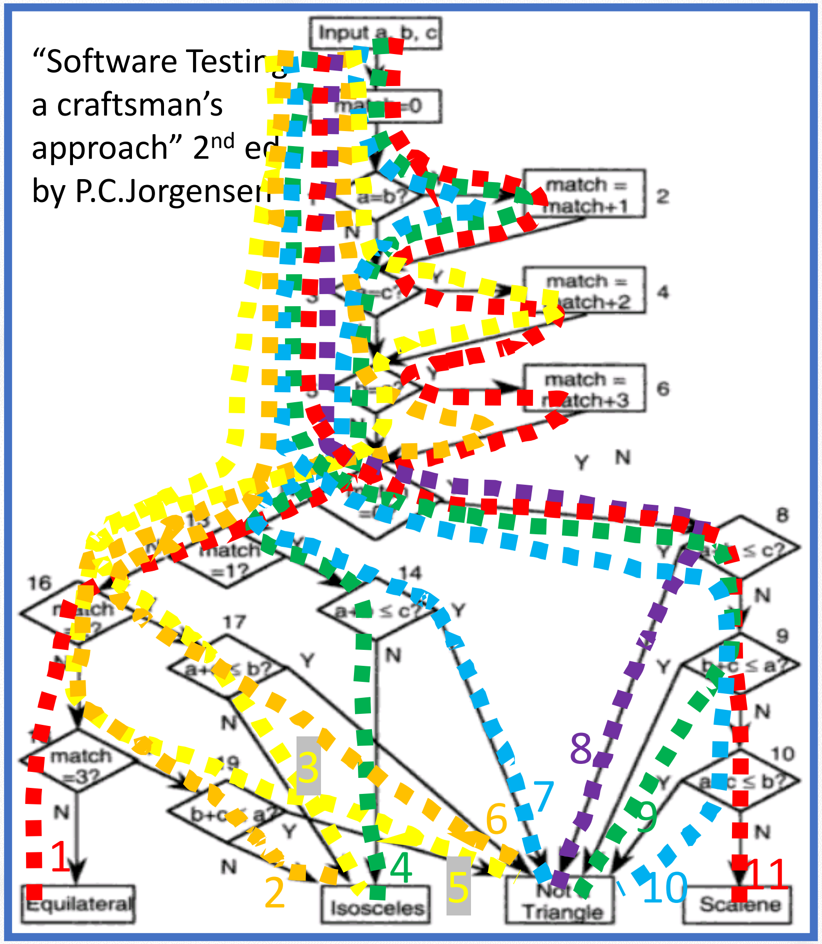
a,b,c = 3,2,2:match=3:result=1:p2
a,b,c = 2,1,2:match=2:result=1:p3
a,b,c = 2,2,1:match=1:result=1:p4

a,b,c = 2,1,1:match=3:result=2:p5
a,b,c = 1,2,1:match=2:result=2:p6
a,b,c = 1,1,2:match=1:result=2:p7

a,b,c = 2,1,3:match=0:result=2:p8
a,b,c = 3,2,1:match=0:result=2:p9
a,b,c = 2,3,1:match=0:result=2:p10

a,b,c = 4,3,2:match=0:result=3:p11

“Software Testing a craftsman’s approach” 2nd ed by P.C.Jorgensen



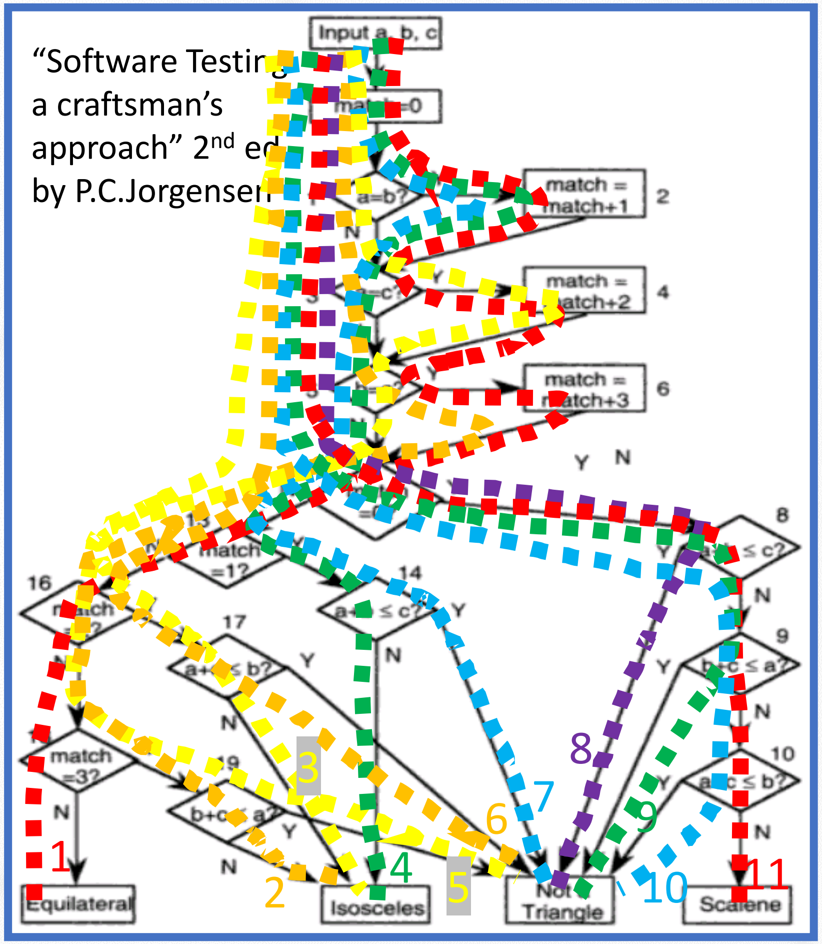
3

# Test Cases for the Triangle Decision

a) a≠b /\ a≠b /\ a≠b  (match=0)
b) a=b (match=1)
c) a=c (match=2)
d) b=c (match=3)
e) a=b /\ a=b /\ a=b (match=6)

Cartesian product

1) match≠0 /\ match≠1 /\ match≠2 /\ match≠3  (EQ)
2) match≠0 /\ match≠1 /\ match≠2 /\ match=3
    /\ b+c > a (ISO)
3) match≠0 /\ match≠1 /\ match≠2 /\ match=3
    /\ b+c ≤ a (NTR)
4) match≠0 /\ match≠1 /\ match=2 /\ a+c > b  (ISO)
5) match≠0 /\ match≠1 /\ match=2 /\ a+c ≤ b (NTR)
6) match≠0 /\ match=1 /\  a+b > c (ISO)
7) match≠0 /\ match=1 /\  a+b ≤ c (NTR)
8) match=0 /\ a+b ≤ c (NTR)
9) match=0 /\ a+b > c /\ b+c ≤ a (NTR)
10) match=0 /\ a+b > c /\ b+c > a /\ a+c ≤ b (NTR)
11) match=0 /\ a+b > c /\ b+c > a /\ a+c > b (SCL)

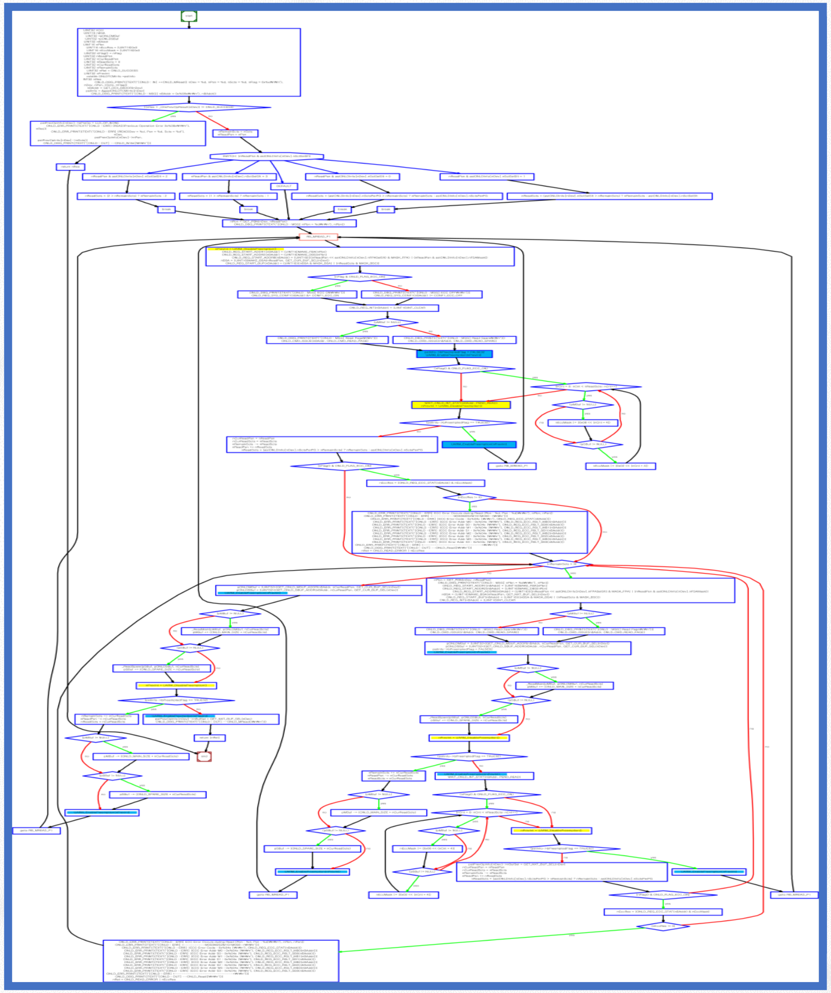


“Software Testing a craftsman’s approach” 2nd ed by P.C.Jorgensen

3

# Test Cases for the Triangle Decision

- # of test cases required?
  - ① 4
  - ② 11
  - ③ 50
  - ④ 100
- # of feasible unique execution paths?
  - 11
- The goal of testing
  - Generate 11 test cases that exercise the 11 unique execution paths



# More Complex Testing Situations (1/3)

- Software is constantly changing
  - What if “integer value” is relaxed to “floating  value” ?
    - Round-off errors should be handled explicitly
  - What if new statements $S_1$ … $S_n$ are added to check whether the given triangle is a right angle triangle (직각삼각형)?
    - Will you test all previous tests again?
    - How to create minimal test cases to check the changed parts of the target program

# More Complex Testing Situations (2/3)

- Regression testing is essential
  - How to select statements/conditions affected by the revision of the program?
  - How to create test cases to cover those statements/conditions?
  - How to create efficient test cases?
    - How to create a minimal set of test cases (i.e. # of test cases is small)?
    - How to create a minimal test case (i.e. causing minimal execution time)?
  - How to reuse pre-existing test cases?

# More Complex Testing Situations (3/3)

- However, conventional coverage is not complete
  - Ex.  int adder(int x, int y) { return 3;}
    - Test case (x=1,y=2) covers all statements/branches of the target program and detects no error
    - In other words, all variable values must be explored for complete results
- Formal verification aims to guarantee completeness
  - Model checking analyzes all possible x, y values through $2^{64}$ ($=2^{32}$ x $2^{32}$) cases
  - However, model checking is more popular for debugging, not verification

# Concurrency

- Concurrent programs have very high complexity

  due to non-deterministic scheduling

- Ex. int x=0, y=0, z =0;

  void Thread1() {x=y+1; y=z+1; z= x+1;}

  Void Thread2() {y=z+1; z=x+1; x=y+1;}

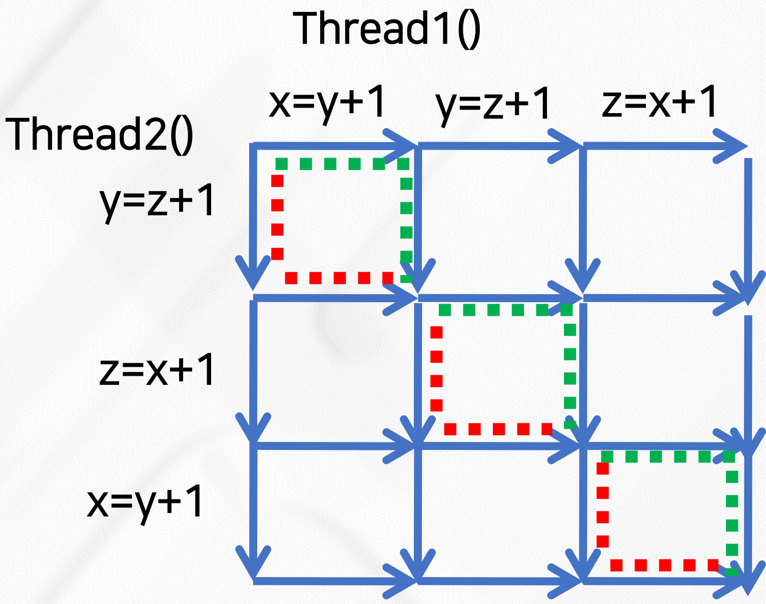
  - Total 20 interleaving scenarios

    = (3+3)!/(3!x3!)

  - However, only 11 unique outcomes

    - assert(x+y+z > 5)???

    - assert(x+y+z < 15)???



Trail1: 2,1,2
Trail2: 2,1,3
Trail3: 2,2,3
Trail4: 2,3,3
Trail5: 2,4,3
Trail6: 3,2,3

Trail7: 3,2,4
Trail8: 4,3,2
Trail9: 4,3,5
Trail10: 5,4,3
Trail11: 5,4,6

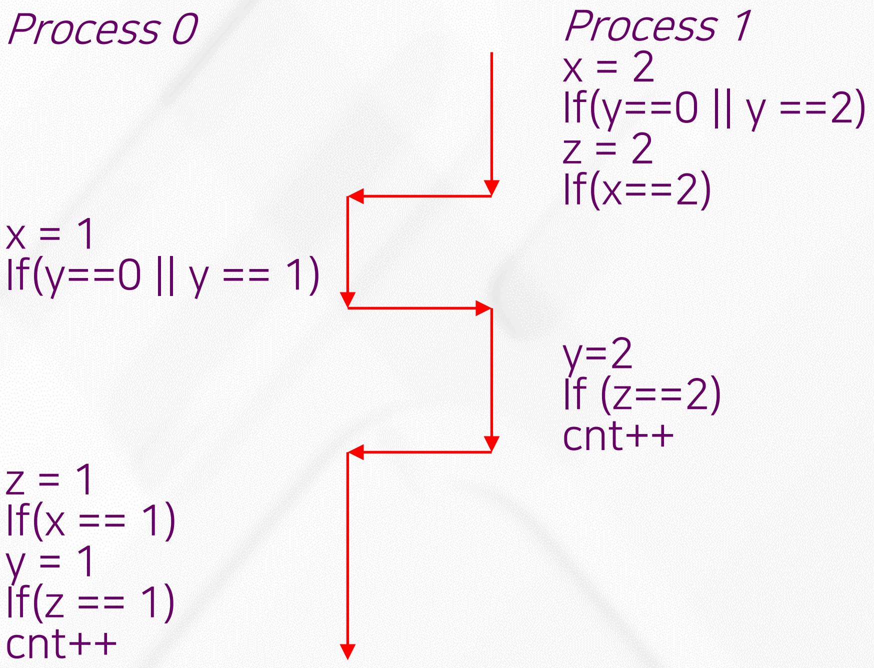
# An Example of Mutual Exclusion Protocol

```
char cnt=0,x=0,y=0,z=0;

void process() {
     char me=_pid +1; /* me is 1 or 2*/
again:
       x = me;
       If (y ==0 || y== me) ;
       else goto again;

       z =me;
       If (x == me) ;
       else goto again;

       y=me;
       If(z==me);
       else goto again;

       /* enter critical section */
       cnt++;
       assert( cnt ==1);
       cnt --;
       goto again;
}
```

*Software locks*

*Critical section*

*Mutual Exclusion Algorithm*



*Process 0*

x = 1
If(y==0 || y == 1)

z = 1
If(x == 1)
y = 1
If(z == 1)
cnt++

*Process 1*

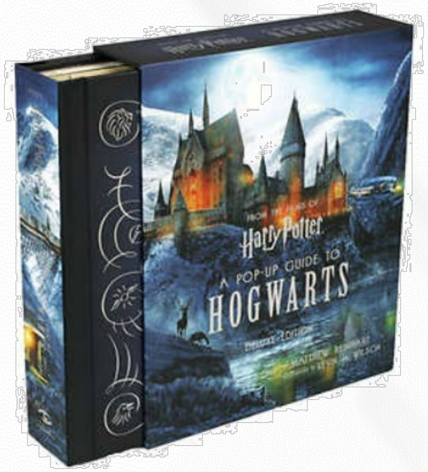
x = 2
If(y==0 || y ==2)
z = 2
If(x==2)

y=2
If (z==2)
cnt++

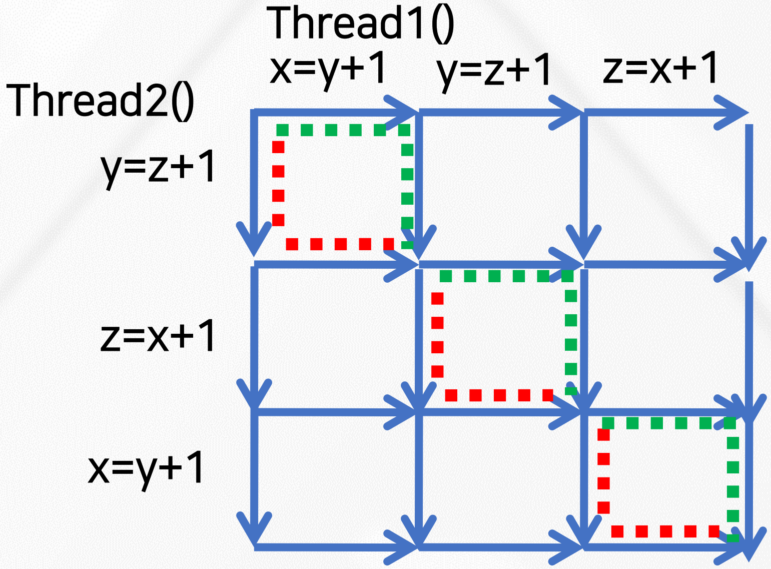
*Violation detected !!!*

*Counter Example*

# Static SW Code vs. Dynamic SW Executions

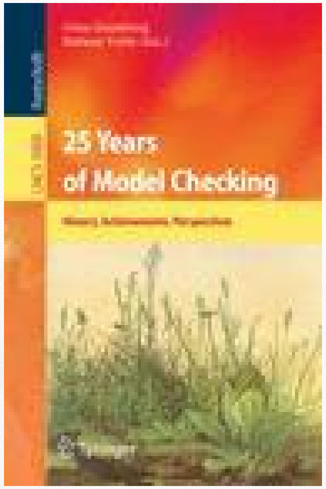


```
int x=0, y=0, z =0;
void Thread1()
{x=y+1; y=z+1; z= x+1;}
void Thread2()
{y=z+1; z=x+1; x=y+1;}
```

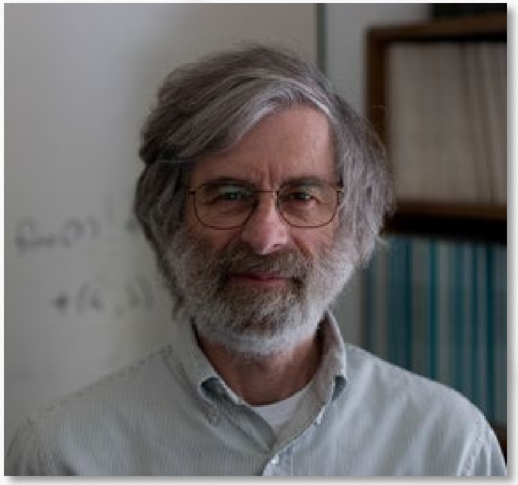


# Formal Analysis of Software

- 2007 ACM Turing Awardees
  - Prof. Edmund Clarke, Dr. Joseph Sipfakis, Prof. E. Allen Emerson
  - For the contribution of migrating from pure model checking research to industrial reality



- 2013 ACM Turing Awardee
  - Dr. Leslie Lamport
  - For fundamental contributions to the theory and practice of distributed and concurrent systems
    - Happens-before relation, sequential consistency, Bakery algorithm, TLA+, and LaTeX



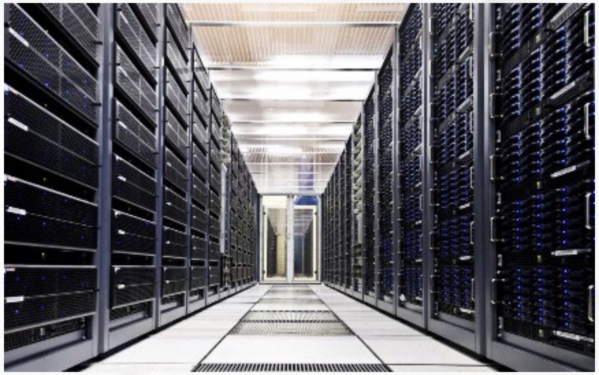
# Significance of Automated SW Testing

- Software has become more ubiquitous and more complex at the same time

**Human resources** are becoming **less reliable and more expensive** for highly complex software

**Computing resources** are becoming **ubiquitous and cheap**

Amazon AWS price: you can use thousands of CPUs @ 0.03$/hr for 2.5Ghz Quad-core CPU



› To-do: Develop **automated and scientific software testing tools** to utilize computing resource effectively and efficiently

# Summary: What is (the essense of) Software?

1. Software = a large set of unique executions
2. SW testing = to find an execution that violates a given requirement among the large set
   - A human brain is poor at enumerating all executions of a target SW, but computer is good at the task
3. Automated SW testing

   = to enumerate and analyze the executions of

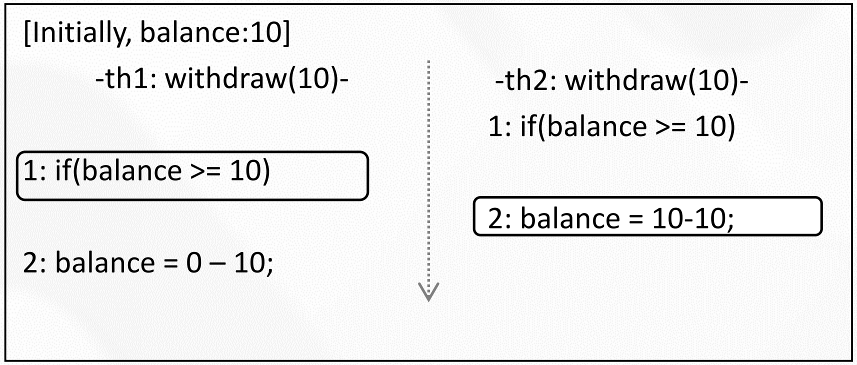
   SW systematically (and exhaustively if possible)

# More Concurrency Bugs

- Data race bugs

```
class Account_DR {
  double balance;
  // INV:balance should be always non-negative

   void withdraw(double x) {
1:   if (balance >= x) {
2:      balance = balance–x;}
      …
   }}
```

(a)  Buggy program code



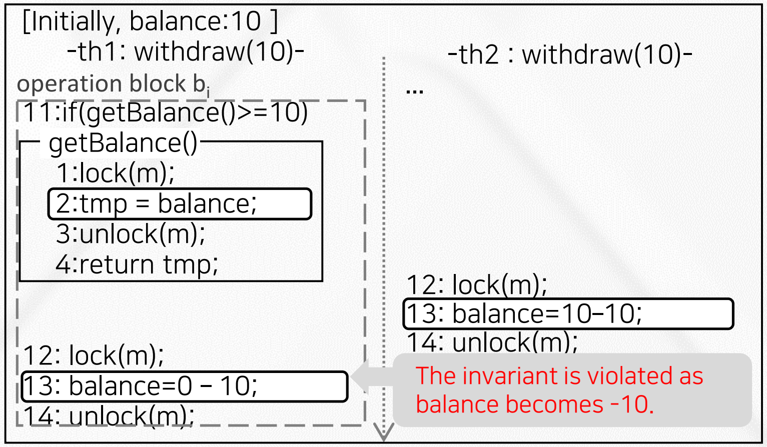
The invariant is violated as balance becomes -10.

(b) Erroneous execution

- Atomicity bugs

```
class Account_BR {
 Lock m;
 double balance;
 // INV: balance should be non-negative

 double getBalance() {          void withdraw(double x){
   double tmp;                    /*@atomic region begins*/
1:  lock(m);                    11: if (getBalance() >= x){
2:  tmp = balance ;             12:   lock(m);
3:  unlock(m);                  13:   balance = balance – x;
4:  return tmp; }               14:   unlock(m); }
                                  /*@atomic region ends*/
                                  … }
```

(a) Buggy program code



The invariant is violated as balance becomes -10.

(b) Erroneous execution