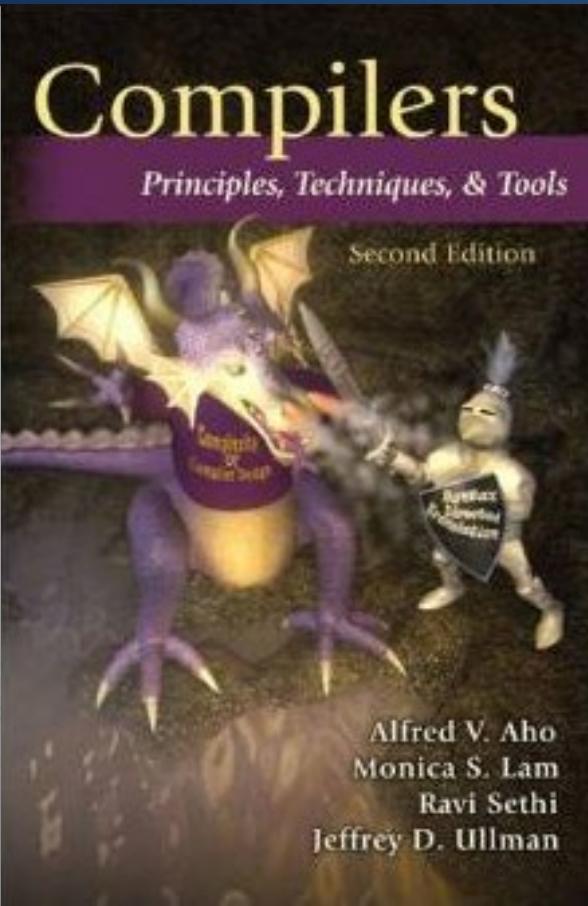
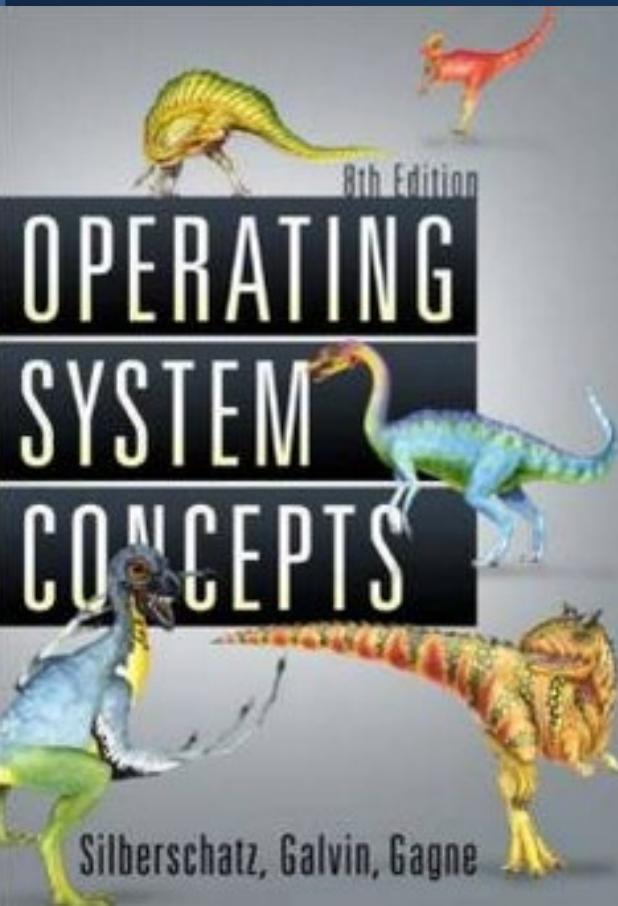


Necessity of Automated SW Analysis

- Fight the Complexity of SW

Moonzoo Kim

KAIST



SW Testing is Very Complex and Difficult Task



Microsoft

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

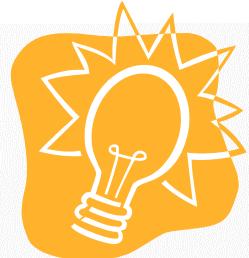
“...The test cases are unbelievably expensive; in fact, there's more lines of code in the test harness than there is in the program itself. Often that's a ratio of about three to one.”

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



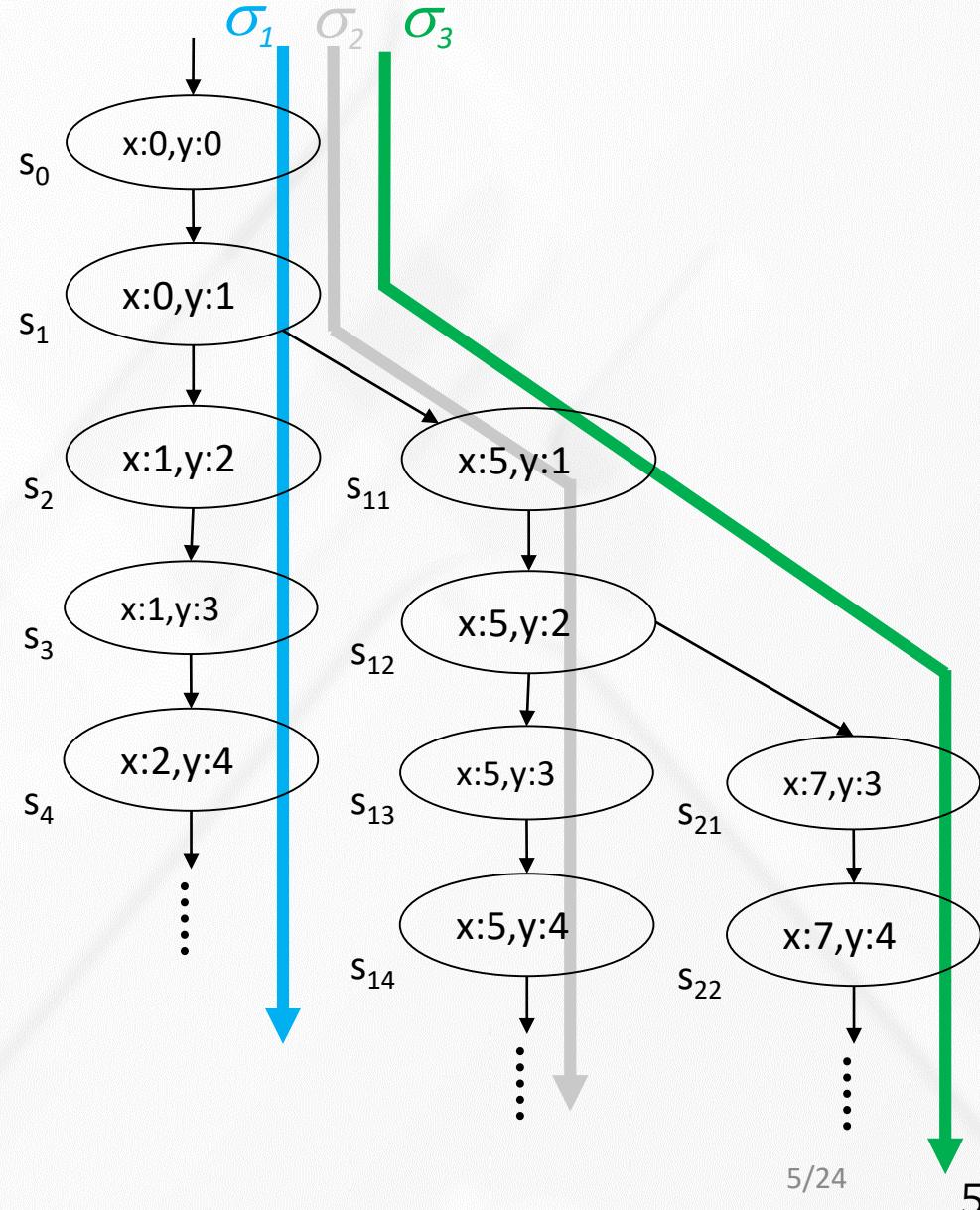
Summary: What is (the essence of) Software?



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

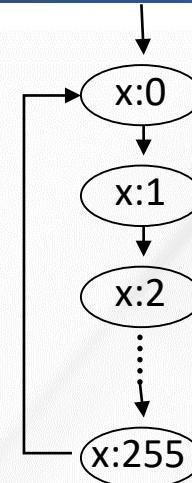
Operational Semantics of Software

- A system has its semantics as a set of system executions σ 's
- A system execution σ is a sequence of states $s_0 s_1 \dots$
 - A state has an environment $\rho_s : Var \rightarrow Val$



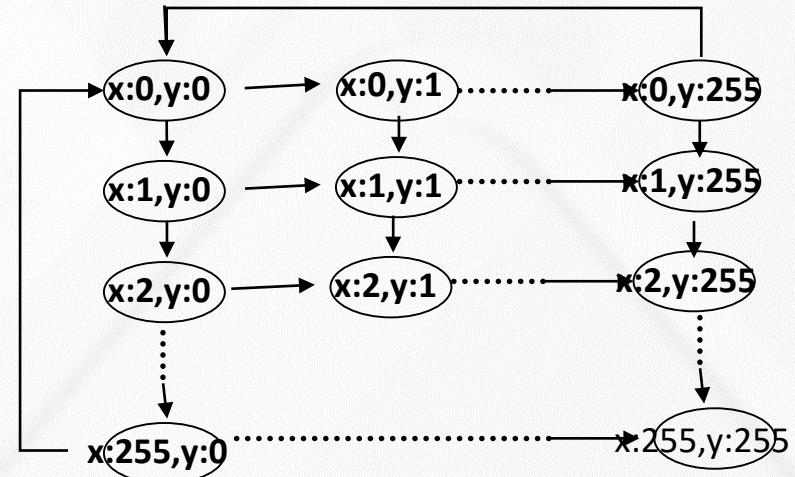
Example

```
active type A() {  
    byte x;  
    again:  
        x=x+1;;  
        goto again;  
}
```



```
active type A() {  
    byte x;  
    again:  
        x=x+1;;  
        goto again;  
}
```

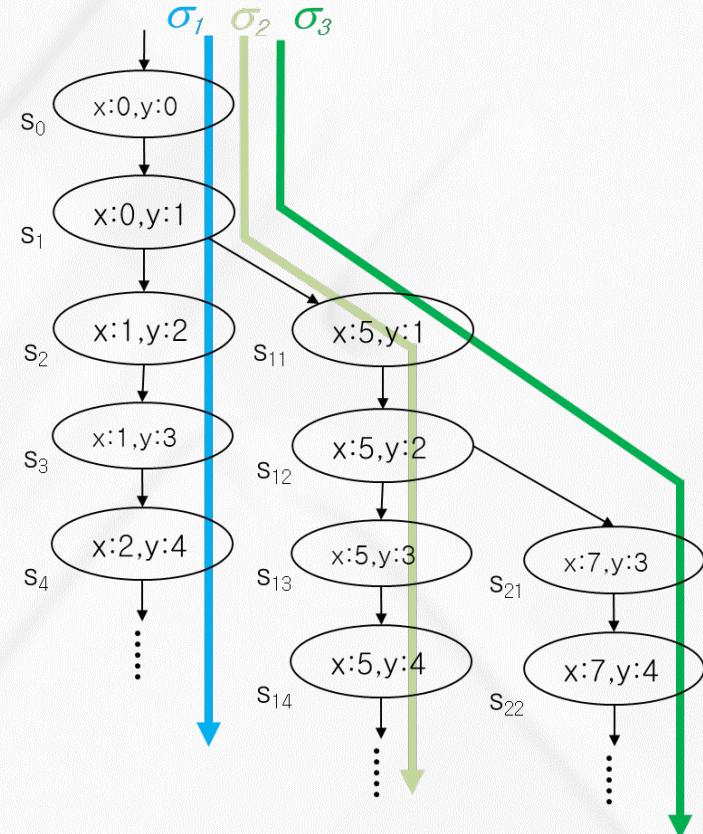
```
active type B() {  
    byte y;  
    again:  
        y++;  
        goto again;  
}
```



Note that model checking analyzes **ALL** possible execution scenarios
while testing analyzes **SOME** execution scenarios

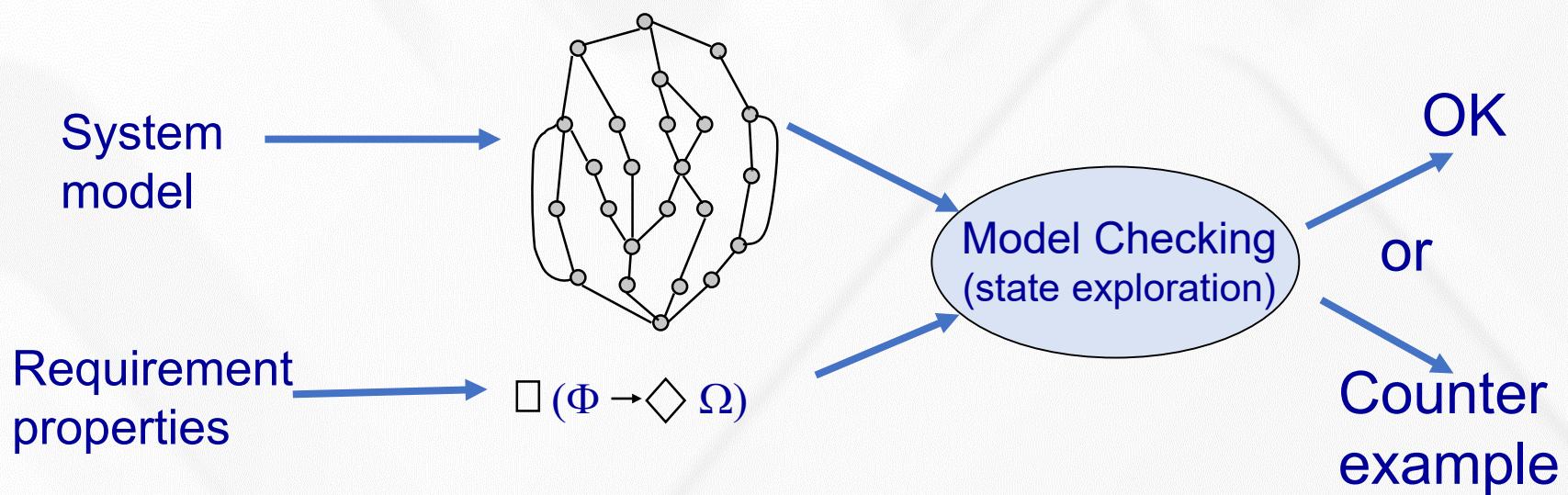
Bug Detection vs. Verification

- Bug detection (testing):
 - a given assert statement (at a given code location) is violated
 - proof: **for a some execution** like σ_1 , a given assert is violated
 - ex. σ_1 violates the assert($2x \neq y$) at s_0, s_2 , and s_4
- Verification (model checking):
 - a given assert statement will be never violated (i.e., **always** satisfied)
 - proof: **for every possible execution** $\sigma_1, \sigma_2, \sigma_3$, and so on, a given assert is satisfied
 - ex. there is no execution σ such that assert($x \geq 0$) is violated.



Verification: State Exploration Method

- Model checking
 - Generate possible states from the model/program and then check whether given requirement properties are satisfied within the state space
 - On-the-fly v.s. generates all
 - Symbolic states v.s. explicit state
 - Model based v.s. code based



Pros and Cons of Model Checking

- Pros

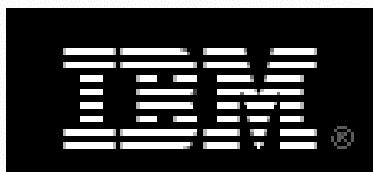
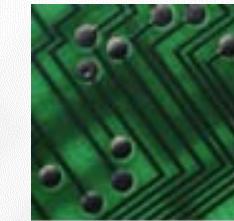
- Fully automated and provide complete coverage
- Concrete counter examples
- Full control over every detail of system behavior
 - Highly effective for analyzing
 - embedded software
 - multi-threaded systems

- Cons

- State explosion problem
- An abstracted model may not fully reflect a real system
- Needs to use a specialized modeling language
 - Modeling languages are similar to programming languages, but simpler and clearer

Companies Working on Model Checking

Microsoft®



cadence®



NEC

Empowered by Innovation



Jet Propulsion Laboratory
California Institute of Technology



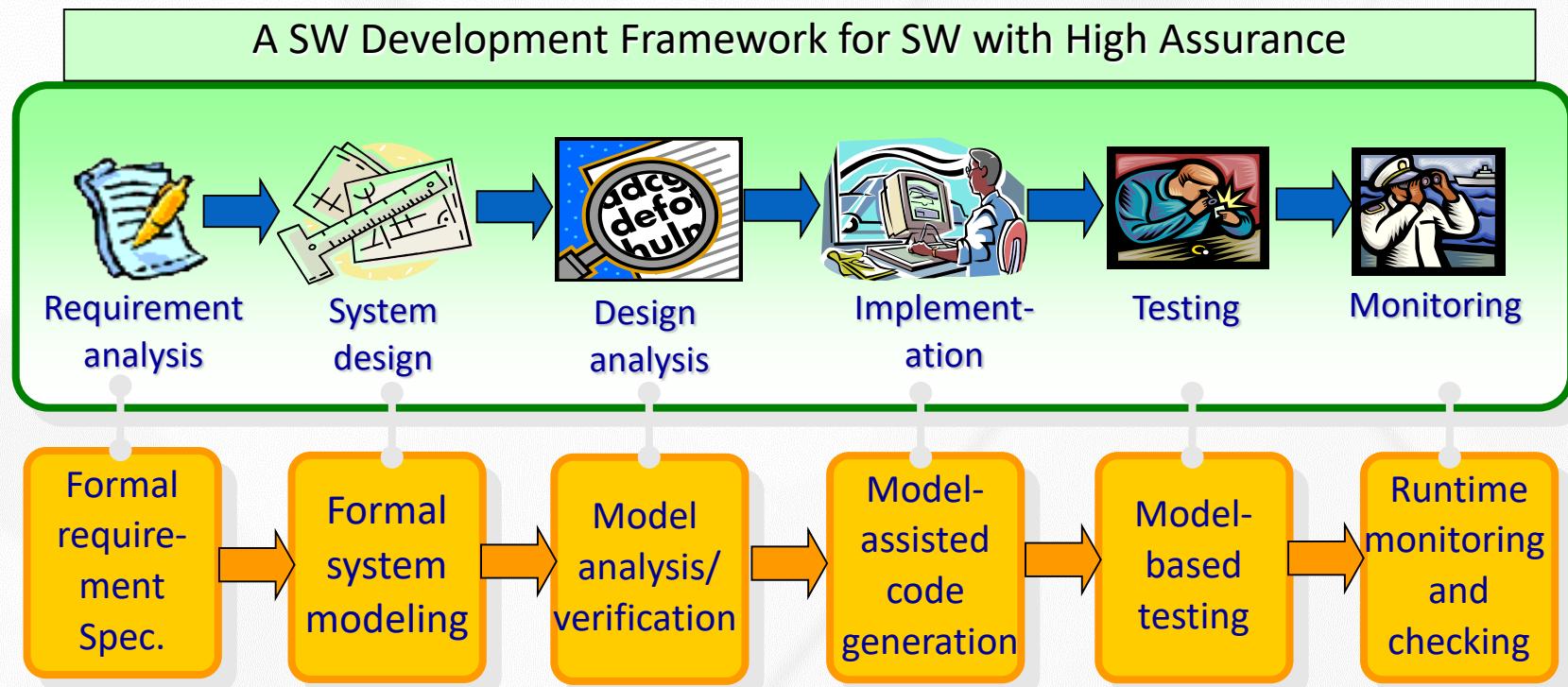
Early Model Checking History



1981	Clarke / Emerson: CTL Model Checking Sifakis / Quielle	10^5
1982	EMC: Explicit Model Checker Clarke, Emerson, Sistla	
1990	Symbolic Model Checking	10^{100}
1992	Burch, Clarke, Dill, McMillan SMV: Symbolic Model Verifier McMillan	
1998	Bounded Model Checking using SAT Biere, Clarke, Zhu	10^{1000}
2000	Counterexample-guided Abstraction Refinement Clarke, Grumberg, Jha, Lu, Veith	

Software Development Cycle

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



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

Necessity of Rigorous Req. Spec.

- Specifications in natural language (ex. 한국어, English, 日本語, etc) can be easily **ambiguous**, **inconsistent**, and **incomplete**
 - Also, connectives in natural languages can have several meaning
 - John eats an apple **and** Mary eats an orange
 - John drove on **and** hit Mary
 - The sun is shining **and** John feels happy
 - John is a Korean **or** a computer scientist
 - John is working **or** he is at home
- and** may have an ordering information in time
- and** may indicate a cause and result
- or** may have an exclusive meaning

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

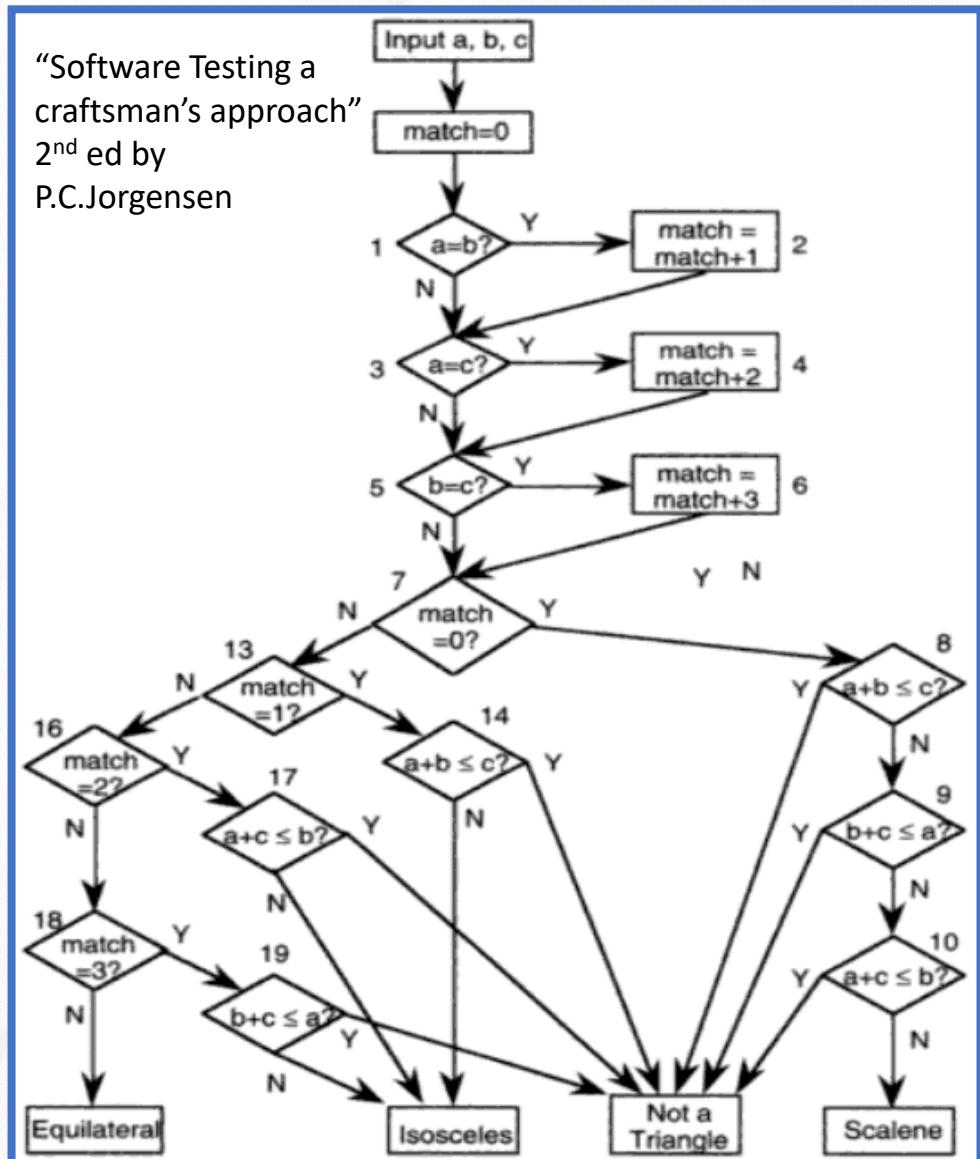
(3,4,5), (2,2,1), (1,1,1) ?

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

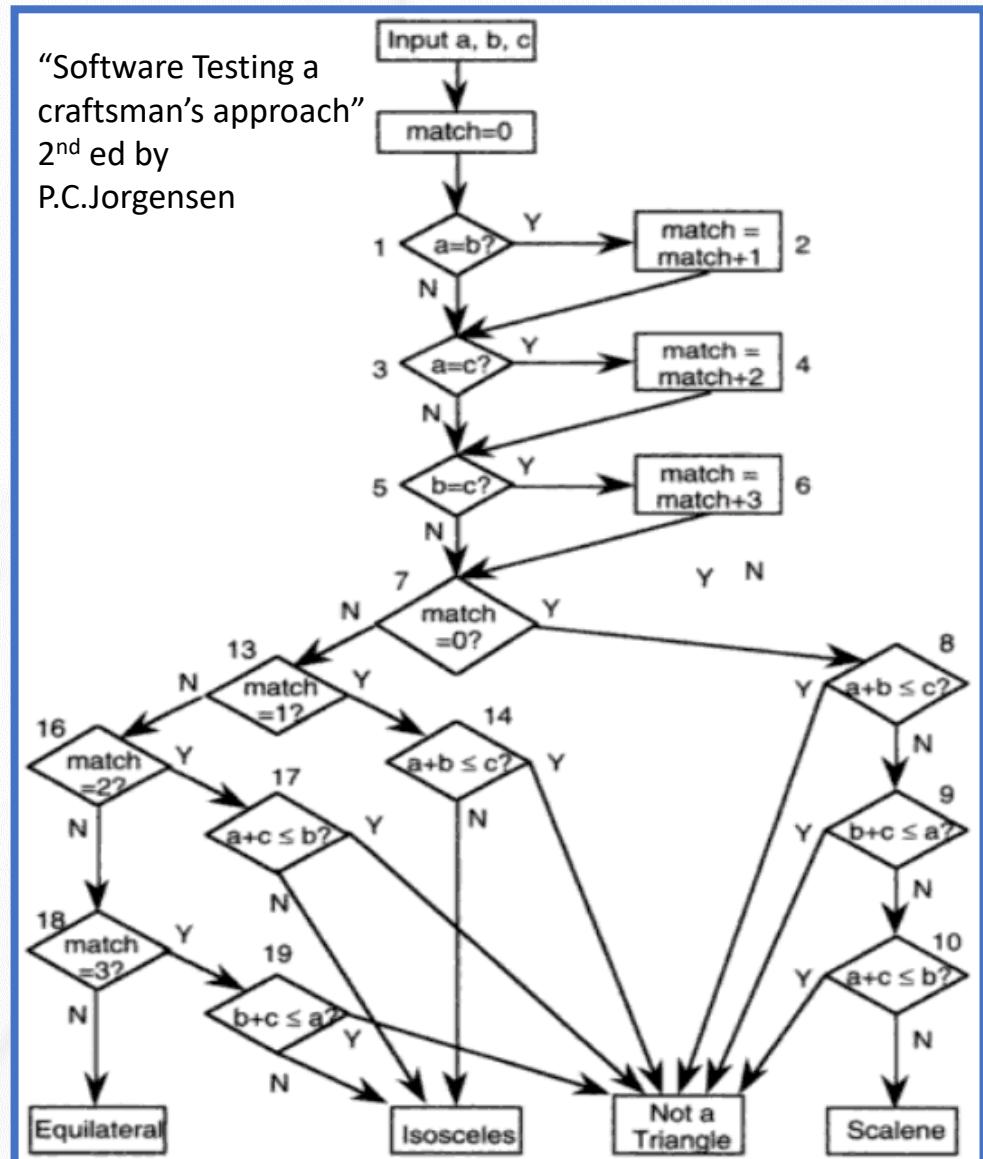
Test Cases for the Triangle Decision

```
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; }
```



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**



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

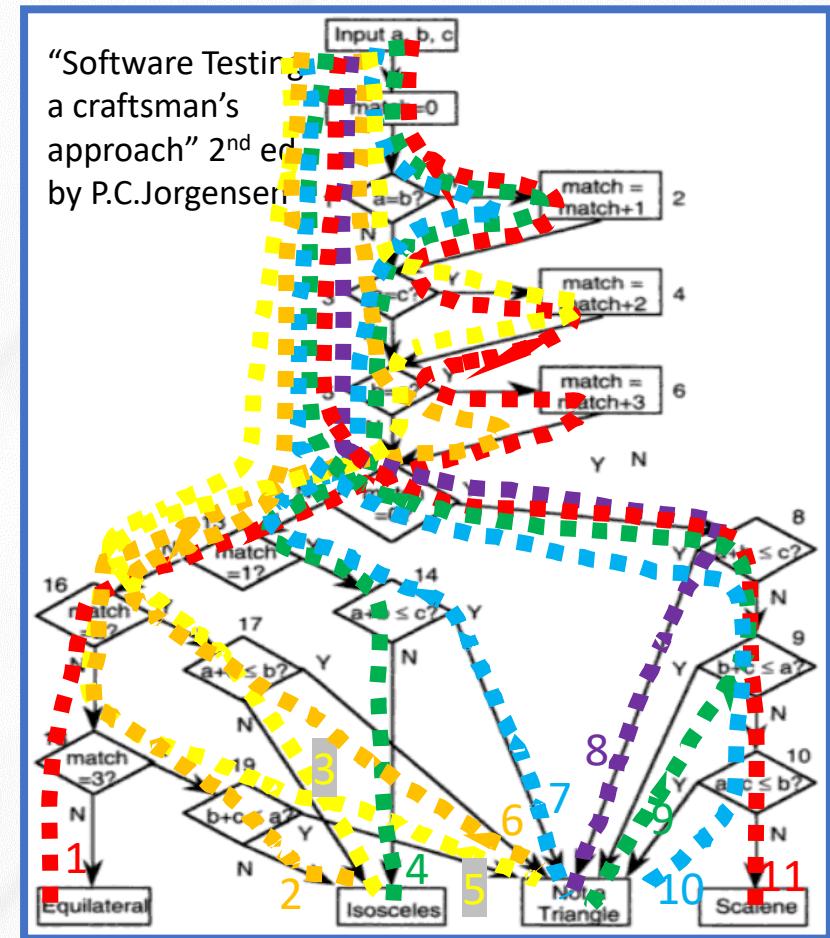
a,b,c = 1,1,2:match=1:result=2:p7

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



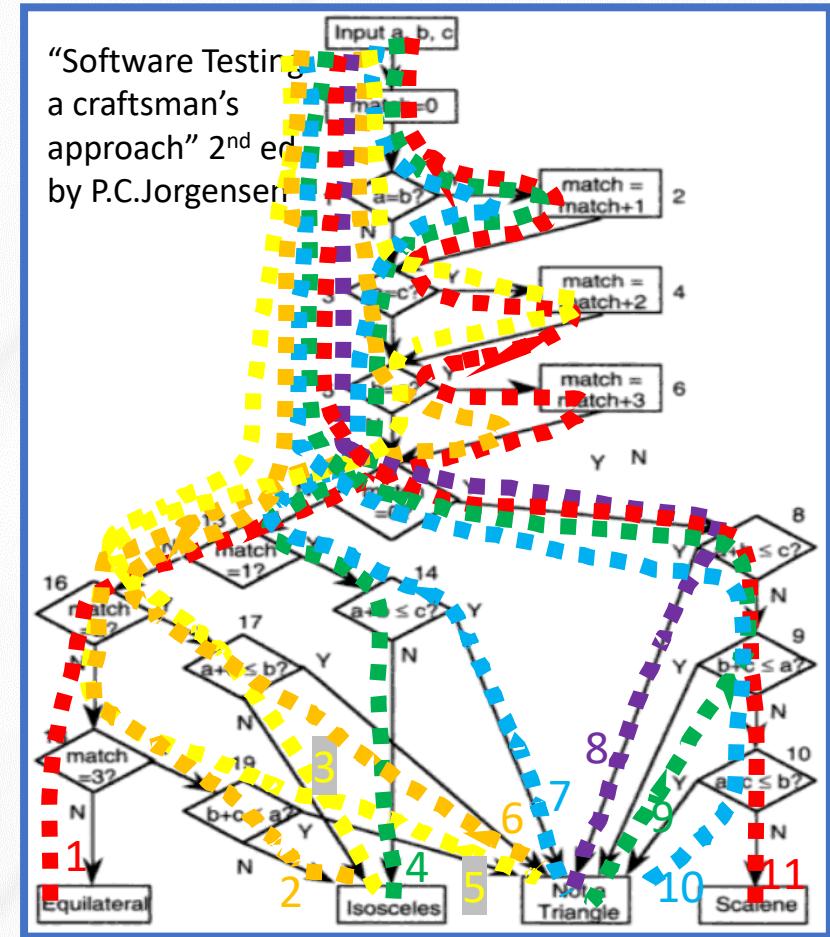
Test Cases for the Triangle Decision

- a) $a \neq b \wedge a \neq c \wedge b \neq c$ (match=0)
- b) $a = b \wedge a \neq c \wedge b \neq c$ (match=1)
- c) $a \neq b \wedge a = c \wedge b \neq c$ (match=2)
- d) $a \neq b \wedge a \neq c \wedge b = c$ (match=3)
- e) $a = b \wedge a = c \wedge b = c$ (match=6)



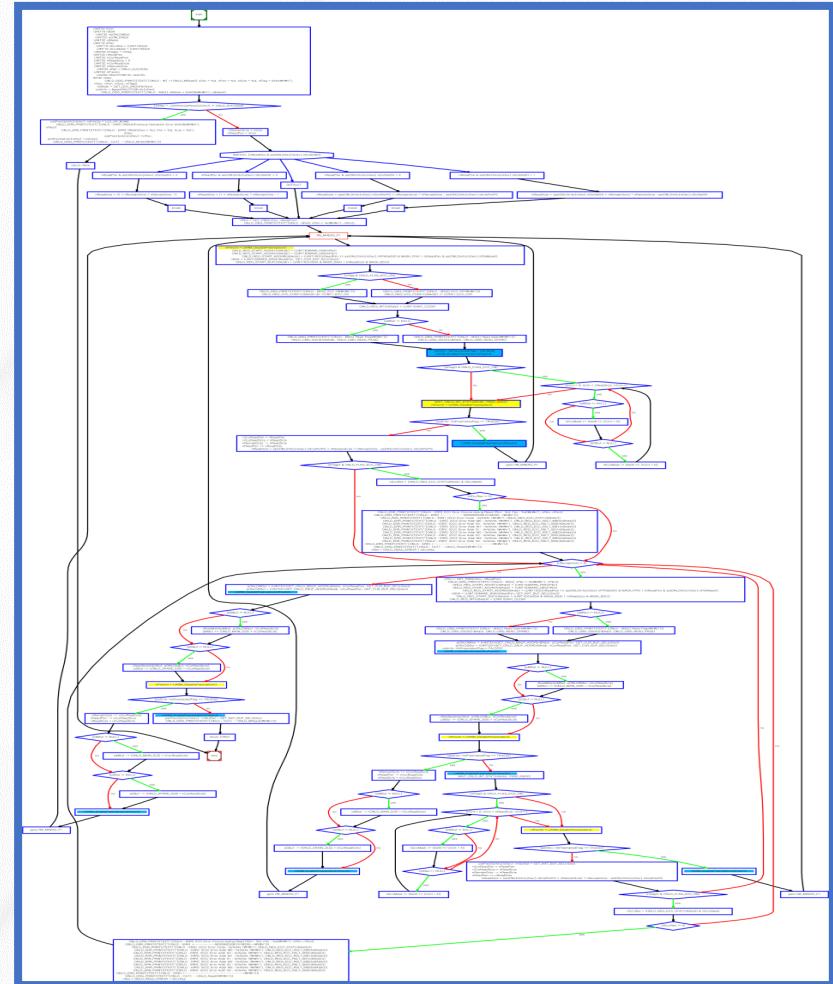
Cartesian
product

- 1) $\text{match} \neq 0 \wedge \text{match} \neq 1 \wedge \text{match} \neq 2 \wedge \text{match} \neq 3$ (EQ)
- 2) $\text{match} \neq 0 \wedge \text{match} \neq 1 \wedge \text{match} \neq 2 \wedge \text{match} = 3 \wedge b+c > a$ (ISO)
- 3) $\text{match} \neq 0 \wedge \text{match} \neq 1 \wedge \text{match} \neq 2 \wedge \text{match} = 3 \wedge b+c \leq a$ (NTR)
- 4) $\text{match} \neq 0 \wedge \text{match} \neq 1 \wedge \text{match} = 2 \wedge a+c > b$ (ISO)
- 5) $\text{match} \neq 0 \wedge \text{match} \neq 1 \wedge \text{match} = 2 \wedge a+c \leq b$ (NTR)
- 6) $\text{match} \neq 0 \wedge \text{match} = 1 \wedge a+b > c$ (ISO)
- 7) $\text{match} \neq 0 \wedge \text{match} = 1 \wedge a+b \leq c$ (NTR)
- 8) $\text{match} = 0 \wedge a+b \leq c$ (NTR)
- 9) $\text{match} = 0 \wedge a+b > c \wedge b+c \leq a$ (NTR)
- 10) $\text{match} = 0 \wedge a+b > c \wedge b+c > a \wedge a+c \leq b$ (NTR)
- 11) $\text{match} = 0 \wedge a+b > c \wedge b+c > a \wedge a+c > b$ (SCL)



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 $S_1 \dots S_n$ 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} ($=2^{32} \times 2^{32}$) 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;}
```

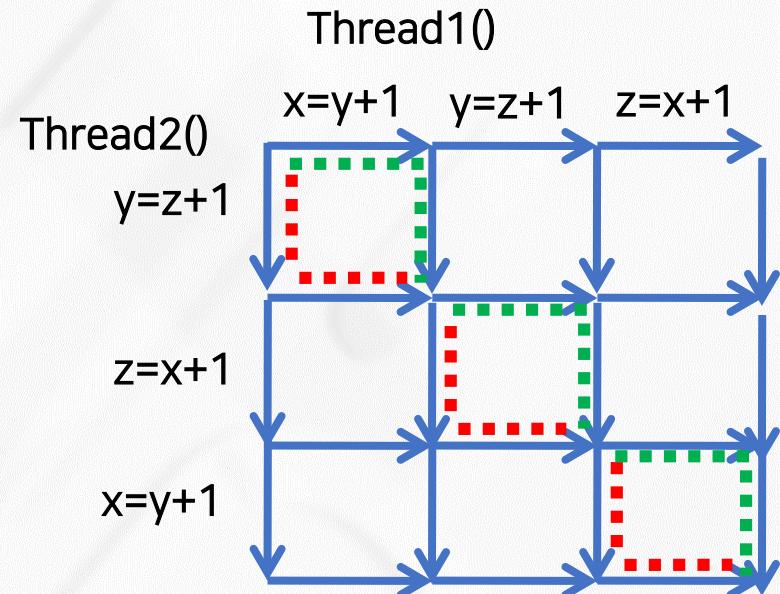
- Total 20 interleaving scenarios

$$= (3+3)!/(3! \times 3!)$$

- 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++
```

*Counter
Example*

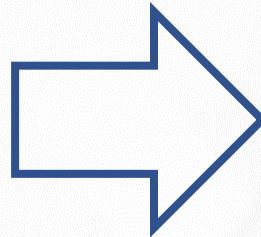
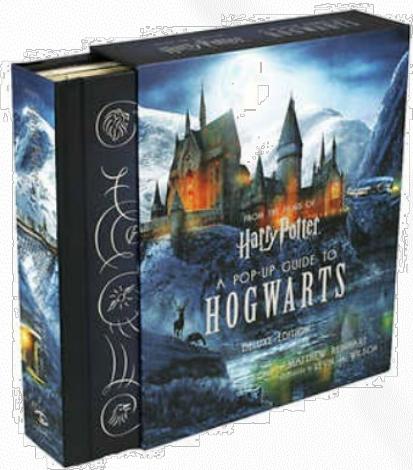
Process 1

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

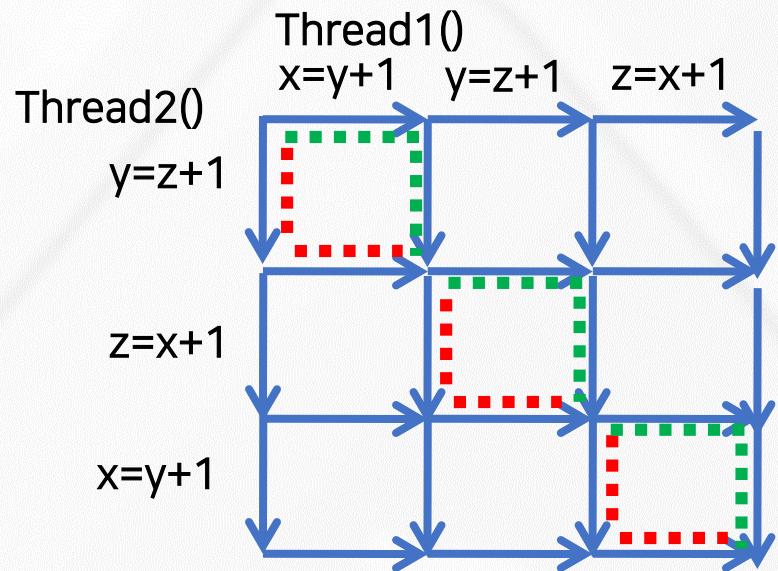
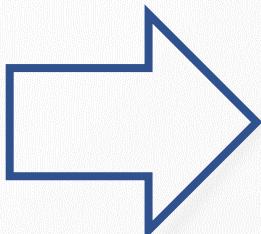
```
y=2  
If (z==2)  
cnt++
```

Violation detected !!!

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;}
```



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

[Initially, balance:10]
-th1: withdraw(10)-
1: if(balance >= 10)
2: balance = 0 - 10;

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

(a) Buggy program code

[Initially, balance:10]
-th1: withdraw(10)-
operation block b:
11:if(getBalance()>=10)
| getBalance()
| 1:lock(m);
| 2:tmp = balance;
| 3:unlock(m);
| 4:return tmp;

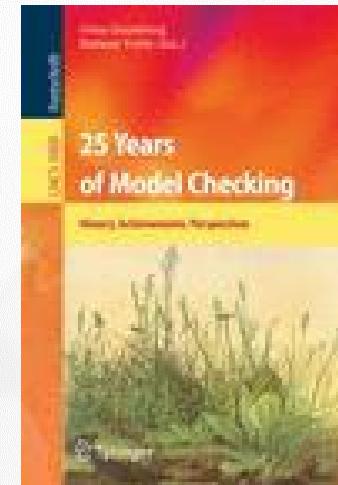
...
-th2 : withdraw(10)-
12: lock(m);
13: balance=10-10;
14: unlock(m);

The invariant is violated as
balance becomes -10.

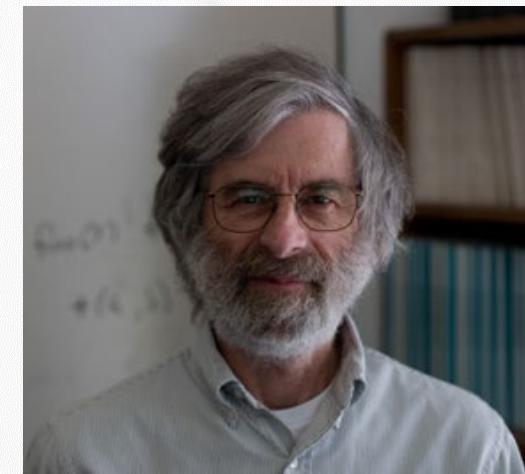
(b) Erroneous execution

Formal Analysis of Software

- 2007 ACM Turing Awardees
 - Prof. Edmund Clarke, Dr. Joseph Sifakis, 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



An Example of a Provable Stmt and a True Stmt

Four derivation rules

$$\frac{\varphi \wedge \psi}{\varphi} \wedge E_l \quad \frac{\varphi \wedge \psi}{\psi} \wedge E_r$$

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi} \wedge I \quad \frac{[\varphi] \quad \psi}{\varphi \rightarrow \psi} \rightarrow I$$

$$\frac{\frac{[\varphi \wedge \psi] \wedge E_r \quad [\varphi \wedge \psi] \wedge E_l}{\psi \wedge \varphi} \wedge I}{\varphi \wedge \psi \rightarrow \psi \wedge \varphi} \rightarrow I$$

Proof

- Syntactic method
- Derivation

$$\vdash \varphi \wedge \psi \rightarrow \psi \wedge \varphi$$

φ	ψ	$\varphi \wedge \psi \rightarrow \psi \wedge \varphi$
T	T	T
T	F	T
F	T	T
F	F	T

Validity

- Semantic method
- Valuation

$$\models \varphi \wedge \psi \rightarrow \psi \wedge \varphi$$

Model Checking vs. Theorem Proving

- In model checking,
 - you describe an abstracted version of your system and you can automatically check some properties.
 - Your model has to be small enough (in term of number of states) to be processed by the tool, and the class of formulas you can express may be limited.
 - Typically, you can check safety properties (such as assert statement in your code).
- In theorem proving (a.k.a., proof assistant),
 - programs are typically verified method-by-method.
 - Given a method and its **contract**, a theorem prover transforms the precondition while symbolically executing the method.
 - Then, it checks whether the transformed precondition is a model of the postcondition (i.e., it implies the postcondition).

Theorem Proving versus Model Checking
EECS 4315

www.eecs.yorku.ca/course/4315/

Model Checking vs. Theorem Proving

- In model checking,
 - Model checking, roughly, tries to use brute force to answer the question and requires **no human interaction** in doing so. You could imagine it feeding every possible input to every process, choosing every possible interleaving of messages and, for every state reachable in such a manner, checking whether the bad thing happens.
- In theorem proving (a.k.a., proof assistant),
 - you try to provide the rationale of why things can't go wrong in form of theorems. However, you also have to **convince the theorem prover** that your reasoning is sound.
 - So first you need to understand what methods of reasoning you are using precisely, and you also need to somewhat understand the way of how the prover "ticks" and what kinds of reasoning steps it can perform automatically.

Significance of Formal SW Modeling and Verification

- 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



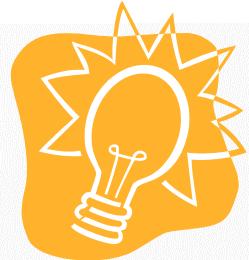
Computing resources are becoming **ubiquitous and cheap**

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



- › To-do: Develop **scientific SW modeling and verification tools** to utilize computing resource effectively and efficiently

Summary: What is (the essence of) Software?



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