SE-307 Review

SE-307 Software Testing Techniques

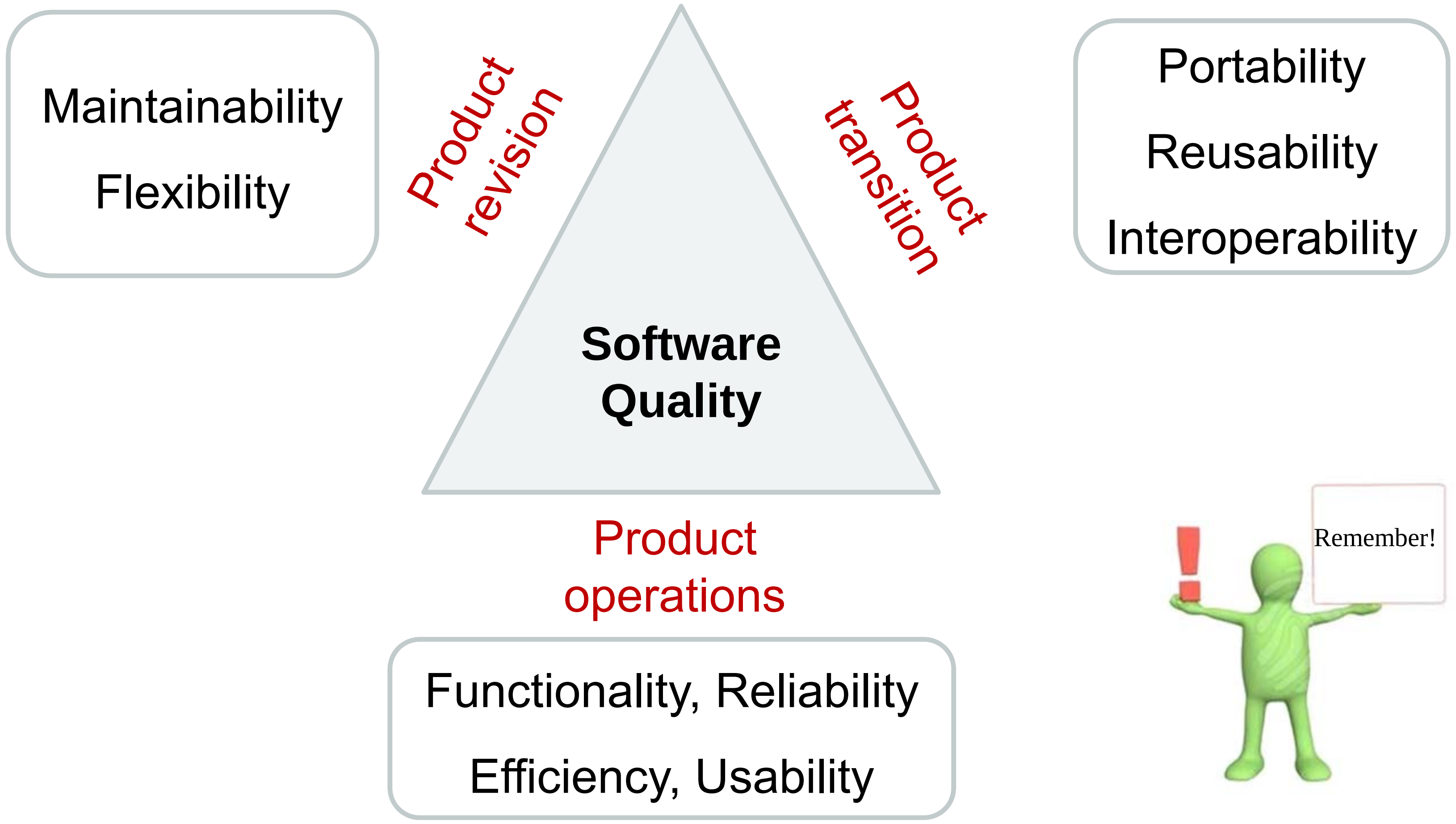
http://my.ss.sysu.edu.cn/wiki/display/SE307/Home

Instructor: Dr. Wang Xinming, School of Software, Sun Yat-Sen University

Knowledge Point #1

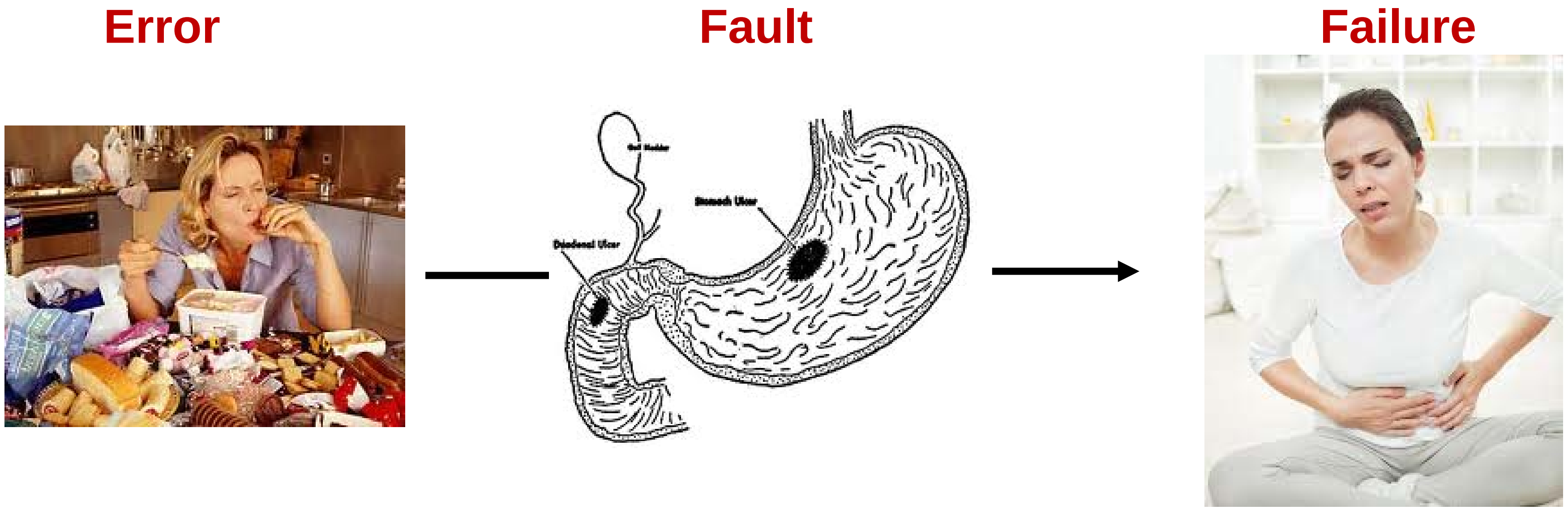
- Software Quality and Software Bugs
  - Software quality factors
  - Failure, fault (bug), and error
  - Where are bugs coming from?
  - Facts on bugs
  - Why it is difficult to write bug-free programs

Software Quality Factors



Failure, Fault and Error

- Failure
  Observable incorrect behavior or state of a given system.
- Fault (also known as “bug” and “defect”)
  A defect in a system whose presence causes the failure.
- Error
  The human mistake that produce a fault.



Where are bugs coming from

- Who did it?
  - **Client:** Inappropriate software reuse, communication error.
  - **Requirement engineer**: Omission, misunderstanding, or deliberate deviations of software requirements.
  - **Designer**: Wrong algorithms, wrong handling of boundary conditions, omission of system states, missing abnormal operation handling.
  - **Programmer**: Misunderstanding of design doc, mental lapse while coding, non-compliance with documentation and coding instructions.
  - **Quality assurance team**: Incomplete test plan, not document and report detected faults or failures, not promptly correct faults, incomplete correction of detected errors.
  - **Operational staff**: Procedure errors.
  - **Technical writer**: Omission of software functions, not up-to-date, error in descriptions.

Facts on Bug

- Bugs are much more than coding mistakes
- The earlier a bug is found, the cheaper it costs
- Bugs are costly to fix.
- Not all software bugs get fixed after they are exposed.
- ……

Difficulty of writing bug-free programs

Complexity

- Many parameters
- No two parts of software are alike
- System goes through many states during execution
- Inherent non-linearity

Conformity

- System should conform with standards imposed by
  - other components, such as hardware, or
  - external bodies, such as existing software

Changeability

- Constant need for change

Knowledge Point #2

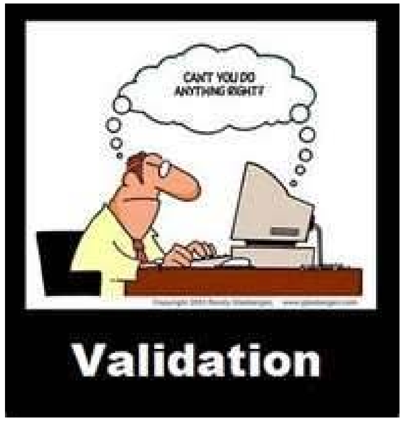
- Quality Assurance and Quality Control
  - Quality Assurance vs. Quality Control
  - Verification vs. Validation
  - Software verification techniques
  - Definition of software testing

Quality Control & Quality Assurance

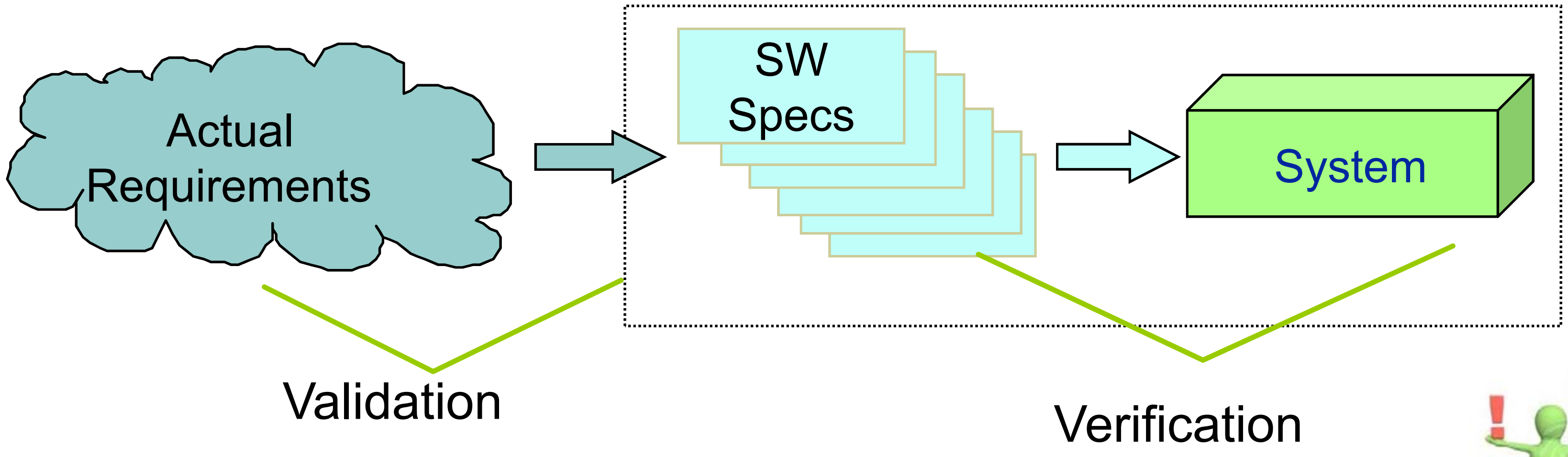
- Quality Assurance:
  - *A set of activities designed to ensure that the development and/or maintenance process is adequate to ensure a system will meet its objectives.*
- Quality Control:
  - *A set of activities designed to evaluate a developed work product.*
- Differences:
  - QA focus on the process elements of a project. QC activities focus on finding defects in specific deliverables.
  - QA makes sure you are doing the right things, the right way, QC makes sure the results of what you've done are what you expected.
  - QA is ***process oriented*** and QC is ***product oriented***.
- Two kind of QC: *verification* and *validation*

Verification and Validation

- Validation:
  does the software system meets the user's real needs?
  are we building the right software?



- Verification:
  does the software system meets the requirements specifications?
  are we building the software right?



Software Verification Techniques

- **Inspection**
  - A constructive review of the artifacts, typically the code (code review). Also known as static testing
- **Model checking**
  - Proving that the programs in question to be absolutely correct with respect to some formal properties on *any* input
- **Program analysis**
  - Identifying certain problematic code patterns that are known to be bug-prone.
- **Testing**
  - Finding bugs by executing input cases to a program and comparing the outputs with some reference system.

Definitions of testing

- IEEE definition:
  - The process of exercising or evaluating a system or system components by manual or automated means to verify that it's satisfies specific requirements or to identify difference between expected and actual results.

- *The art of software testing, Glenford J. Myers*
  - Testing is the process of executing a program with the intent of finding errors."
  - Successful (positive) test: exposes an error

What testing is not

- "Testing is the process of demonstrating that an errors are not present"
  - Testing can never show that an error is not present.
- "Testing is the process of establishing confidence that the program does what it intends to do"
  - Testing can never achieve this end.

Knowledge Point #3

- Testing: the roadmap
  - IS-A relations
  - Concepts in testing
  - The four dimensions of testing techniques
  - Testing as a career

IS-A relations

- Software Quality
  - Quality Assurance
  - Quality Control
    - Validation
    - Verification
      - Inspection
      - Model checking
      - Program analysis
      - Testing

Concepts in Testing

Quality under concern: obedience

QC: God and angels

Test input:
apples and snake

Test oracle:
assert(!apple_eaten())

Subjects under test:
Adam and Eve

Test harness:
Garden of Eden

Test outcome: fail

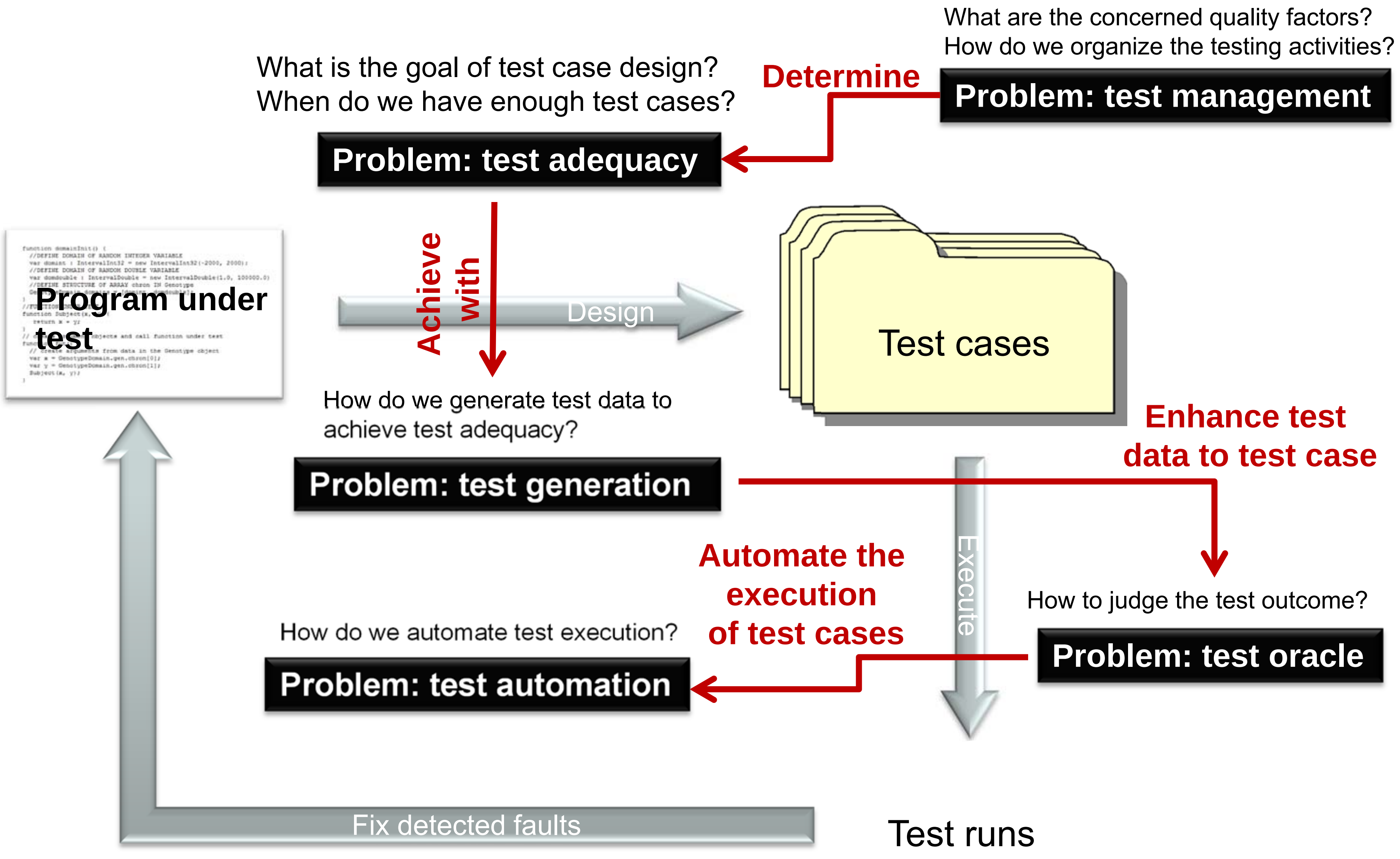
So Many Testing Techniques!

- How shall I learn all of them?
  - Unit testing, Integration testing, System testing, Acceptance testing
  - Functional testing, Non-functional testing
  - Black-box testing, White-box testing, Model-based testing
  - Alpha testing, Beta testing
  - Specification-based testing, Code-based testing
  - Structural testing, Behavior testing
  - Developer testing, QA team testing, User testing
  - Validation testing, Verification testing
  - Regression testing, Smoke testing, Continuous testing
  - Performance testing, Security testing, Reliability testing
  - Web testing, Database testing, Multi-threaded testing

Four dimensions of Testing Techniques

- **P**roblems
  - Fundamental problems: test adequacy, test oracle, test generation
  - Important problems: test automation, test management
- **M**ethods
  - Exploratory testing, black-box testing, white-box testing, model-based testing
- **O**bjectives
  - Unit testing, Integration testing, System testing, Acceptance testing (alpha testing, beta testing), Performance testing (stress testing, load testing), Reliability testing, Regression testing, Security testing, Compatibility testing, ...
- **D**omains
  - GUI testing, Web testing, Database testing, Multi-threaded testing ...

Problems（PMOD）


What are the concerned quality factors?
How do we organize the testing activities?
Problem: test management
Determine
What is the goal of test case design?
When do we have enough test cases?
Problem: test adequacy
Achieve with
Program under test
Design
Test cases
How do we generate test data to achieve test adequacy?
Problem: test generation
Enhance test data to test case
Execute
How to judge the test outcome?
Problem: test oracle
Automate the execution of test cases
How do we automate test execution?
Problem: test automation
Fix detected faults
Test runs

Methods（PMOD）

- **Exploratory method** （探索性测试方法）
  - Explore the software while designing and executing tests simultaneously.
- **Black box method**（黑盒测试方法）
  - Treat the program as a black box, ignoring its internal structure.
- **White box method** （白盒测试方法）
  - Logical analysis of code element (e.g. basic block, branch, path ...).
- **Model-based method**（基于模型的测试方法）
  - Based on a formal model of the software.

Objectives（PMOD）

Objective	Purpose
Functional testing （功能测试）	Verify the functionalities of the system against the requirement specification.
Regression testing (回归测试)	Verify that changes to the software do not break the functionalities of the previous version.
Load testing （并发测试）	Verify the behavior of the system under both normal and anticipated peak load conditions.
Stress testing （压力测试）	Verify the behavior of the system beyond its capacity.
Reliability testing （可靠性测试）	Verify that the failure rate of the software is under an acceptable level.
Usability testing （可用性测试）	Verify that the ease of using and learning the software.
Validation testing （确认测试）	Verify that the software delivers what the users expect (alpha testing and beta testing).

Domain（PMOD）

- Operation system
  - Kernel, device driver
- Database application
  - SQL table, transaction processing, ACID
- Web application
  - Javascript, Ajax, HTML5
- Multi-threaded application
  - Data race, atomicity, transactional memory, asychronized tasks
- Cloud application
  - Map-reduce, distributed computing
- Mobile application
  - Android, touch-screen, sensors…
- ……

Testing as a career

- What is the most important skill for test engineer?
  - Programming

- Remarks by Bill Gates
  - "…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."

Knowledge Point #4

- Test oracle
  - Concept
  - Requirement specification as test oracle
  - Reference implementation as test oracle
  - Program assertion as test oracle
  - Metamorphic relation as test oracle
  - Execution profiling as test oracle
  - Heuristic and statistical test oracle

Oracle

The Oracle offers candy to Neo, the following dialogue ensues …

Neo: Do you already know if I'm going to take which candy?
The Oracle: Wouldn't be much of an Oracle if I didn't.

- The Oracle in Matrix
  - is a program
  - predicts choices and outcomes based on input from the Zion mainframe
- Test Oracles …

Requirement Specification

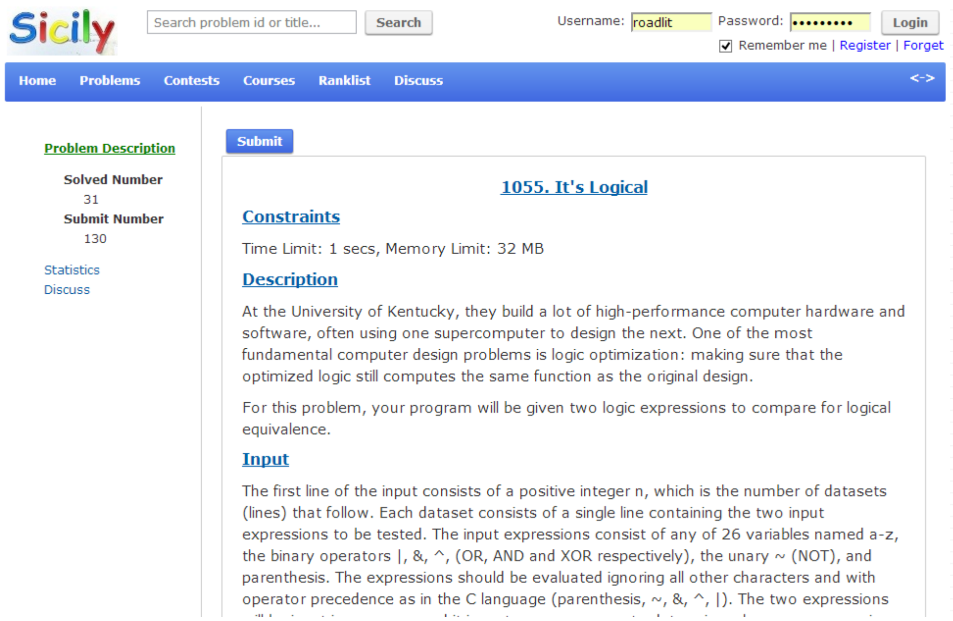
- Extract valid input/output pairs from specifications to verify `intended behavior' of the system.
  - Simulate what the program does with your brain.
- Example: problems in Sicily

You havn't any signature yet.

Sicily酱现在知道每天过题数最多的人啦
点击这里进入。

Sicily Online Judge System(Rev 20120716-961)

rright © 2005-2012 Informatic Lab in SYSU. All rights reserved.



Solved Number
31
Submit Number
130

1055. It's Logical

Constraints

Time Limit: 1 secs, Memory Limit: 32 MB

Description

At the University of Kentucky, they build a lot of high-performance computer hardware and software, often using one supercomputer to design the next. One of the most fundamental computer design problems is logic optimization: making sure that the optimized logic still computes the same function as the original design.

For this problem, your program will be given two logic expressions to compare for logical equivalence.

Input

The first line of the input consists of a positive integer n, which is the number of datasets (lines) that follow. Each dataset consists of a single line containing the two input expressions to be tested. The input expressions consist of any of 26 variables named a-z, the binary operators |, &, ^, (OR, AND and XOR respectively), the unary ~ (NOT), and parenthesis. The expressions should be evaluated ignoring all other characters and with operator precedence as in the C language (parenthesis, ~, &, ^, |). The two expressions

Reference Implementation

- Reference implementation
  - Another program that does exactly (or similarly) what the program under test does.
- Situations where reference implementation can be available:
  - **Case 1: deliberately built** to mimic the behavior of the program under test
  - **Case 2: alternative products**
  - **Case 3: historical version**

Internet Explorer

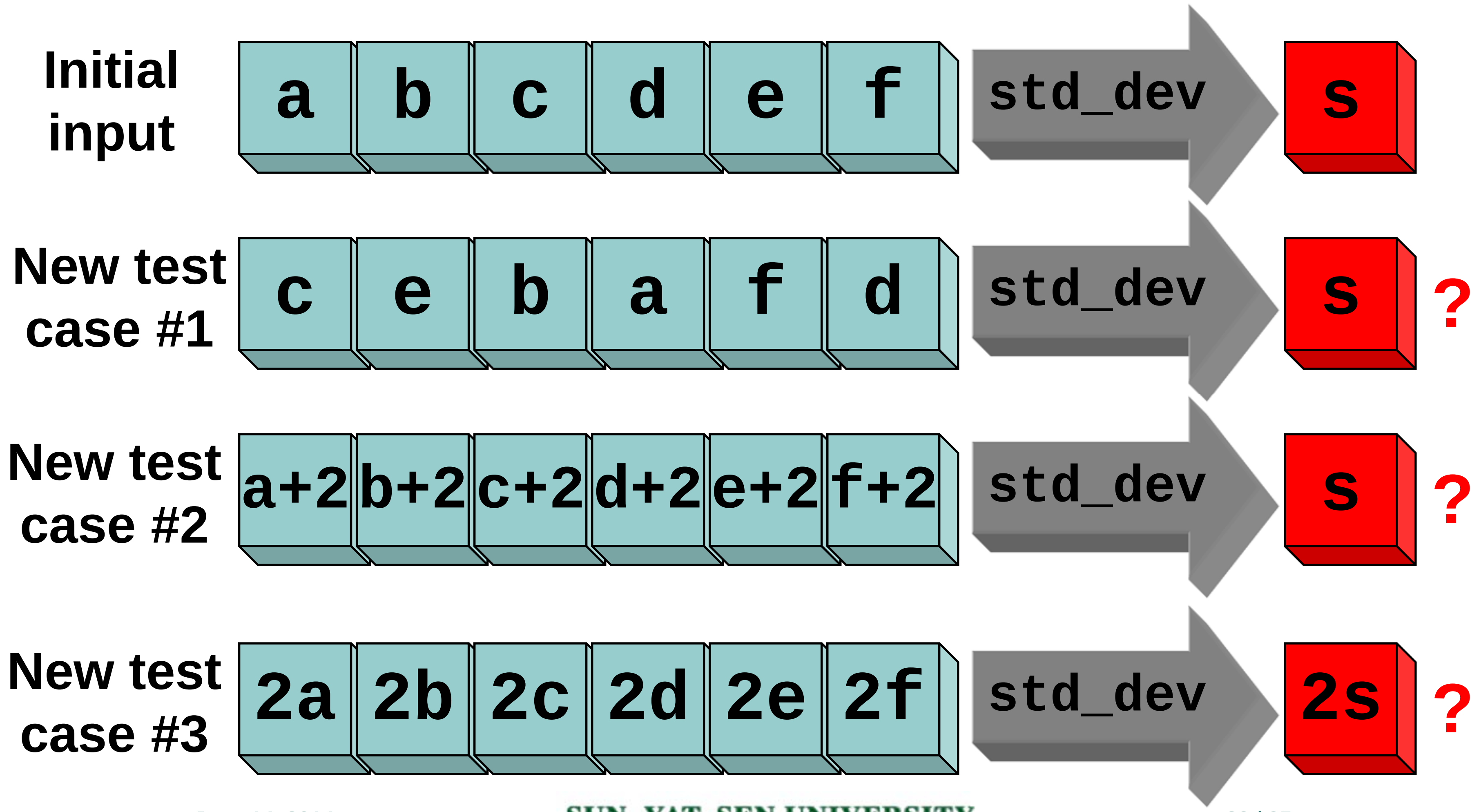
Firefox

Program Assertions

- Java `assert` statement
  - Check *safety* property
- Design by contract
  - Pre-condition
  - Post-condition
  - Class invariant
- Java Modeling Language (JML) to specify contracts

Meta-morphic Relation

- Consider a function to determine the standard deviation of a set of numbers



Execution Profiling

- Execution profilers use instrumentation or sampling to monitor and record program execution details:
  - How many statements have been executed
  - Time elapsed in each function and its descendants
  - How many amounts of memory are allocated/released
  - How many variables have been read/written
  - ...
- Idea: some of them can be directly used to implement a partial oracle

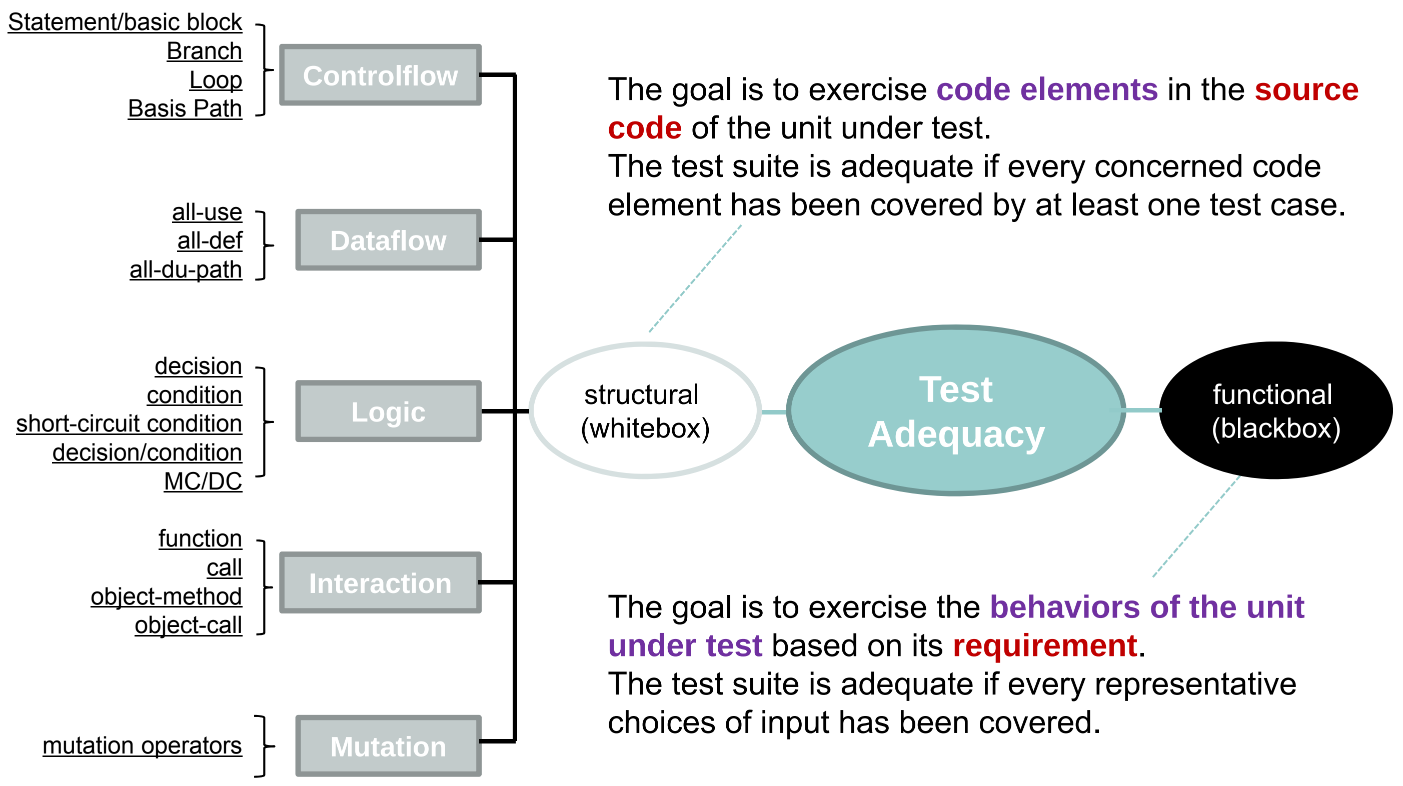
Heuristic and Statistic Oracle

- Assumption:
  - In most of the case, the fault is inactive.
- An intuitive solution to identify failed test runs
  - Learning the norms
  - Identifying the outliers

Who is a terrorist?

Knowledge Point #5

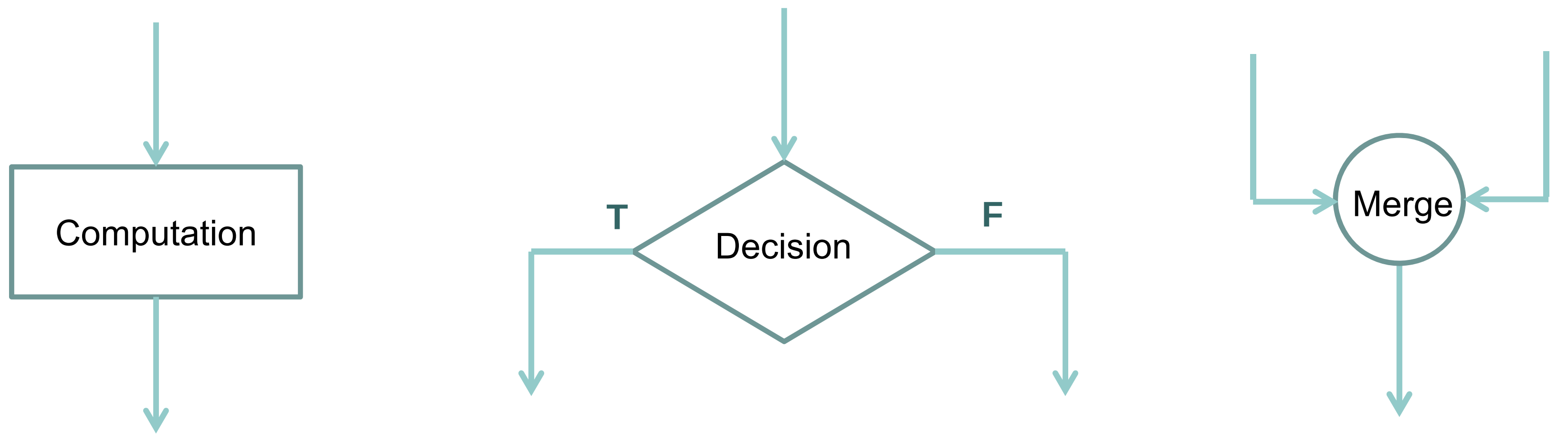
- White-box Test adequacy



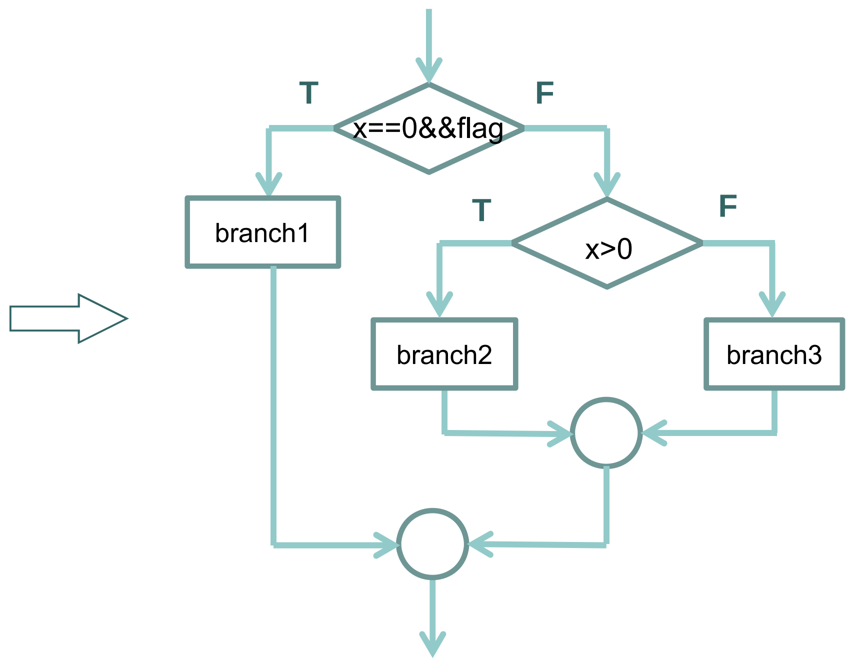
The goal is to exercise **code elements** in the **source code** of the unit under test.
The test suite is adequate if every concerned code element has been covered by at least one test case.

The goal is to exercise the **behaviors of the unit under test** based on its **requirement**.
The test suite is adequate if every representative choices of input has been covered.

Control Flow Graph



if (x==0 && flag) { branch 1 }
else if (x>0) { branch 2}
else { branch 3 }
….



Control Flow Coverage

- The basic idea behind control flow coverage is to define the test adequacy criteria as graph elements that we design test data to “cover”.
  - How to generate the test data to cover these elements belongs to another fundamental problem: test generation.

- Types of control flow coverage that can be defined as elements CFG:
  - Statement coverage（语句覆盖）
  - Basic block coverage（基本块覆盖）
  - Branch coverage（分支覆盖）
  - Loop coverage（循环覆盖）
  - All path coverage（全路径覆盖）
  - Basis path coverage（基本路径覆盖）

Basic Block Coverage（基本块覆盖）

- 100% statement coverage ⇔ 100% basic block coverage
  - Then why bother introducing another coverage criteria?
- Problem of statement coverage:

```
if(…){
   one statement
}else{
   ninety nine statements
}
```

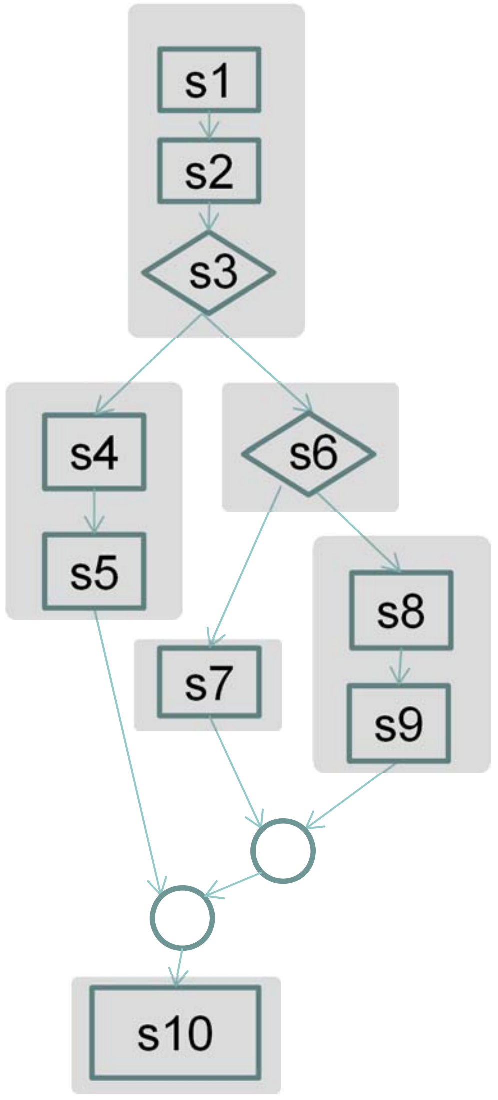
If we have tested one path, then the test process is either 1% (too pessimistic) or 99% (too optimistic).

- Another problem: instrumentation cost
- Basic block: a better unit to measure coverage
  - Much more commonly used in the industry than statement coverage

Definition: Basic Block

- A ***basic block*** is a maximal sequence of statements having the following properties:
  - (单入口) It can only be entered through the first statement;
  - (单出口) It can only exit through the last statement;
  - (顺序执行) Whenever the first statement is executed, the remaining statements are executed in the given sequential order.

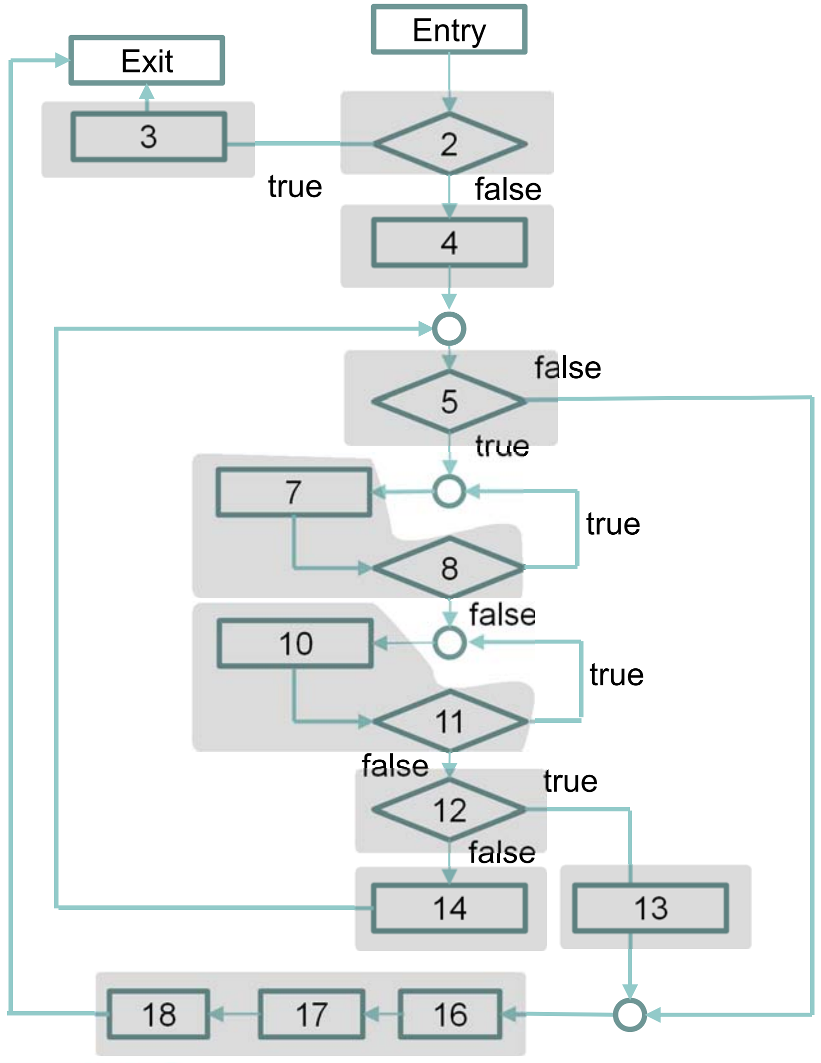
```
s1
s2
if (s3){
    s4
    s5
}else if(s6){
    s7
}else{
    s8
    s9
}
s10
```



CFG & Basic Block Coverage

- Draw the CFG for the following code and identify basic blocks

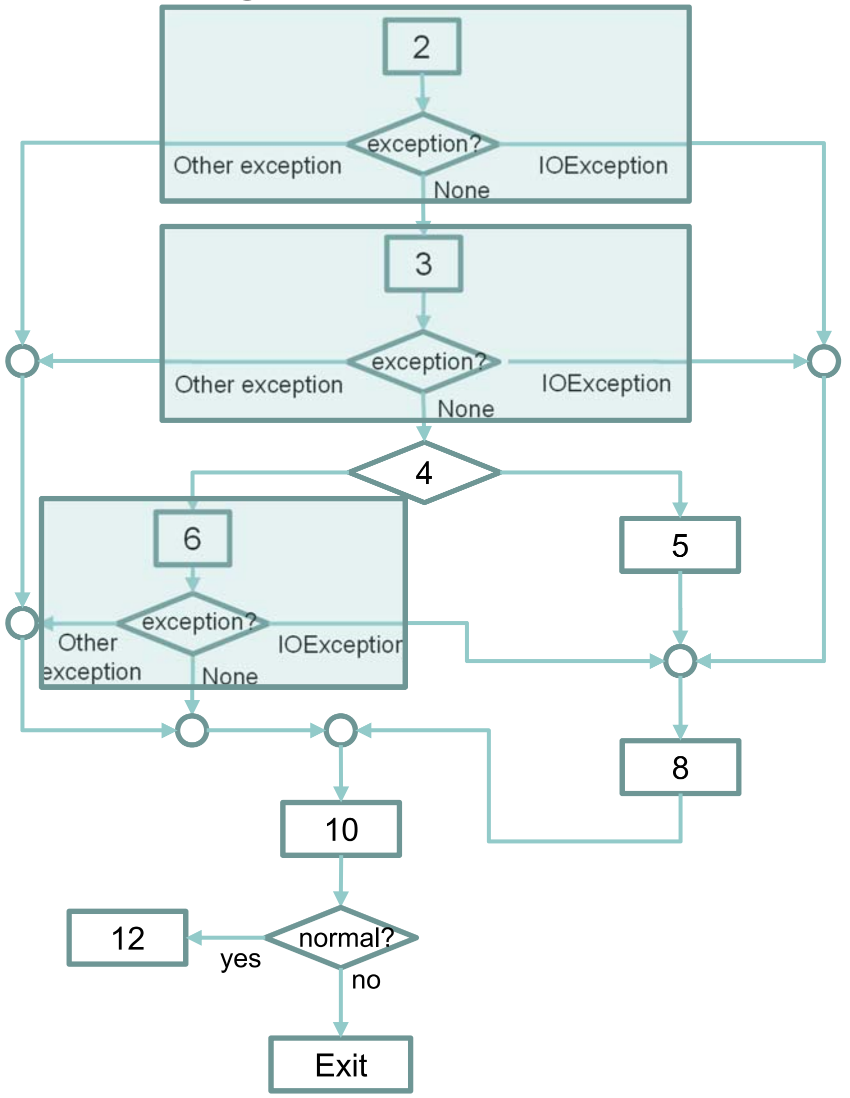
```
// External input-output array: int a[]
   void quicksort( int m, int n ) {
1:      int i, j, v, x;
2:      if ( n <= m )
3:         return;
4:      i = m-1; j = n; v = a[n];
5:      while(1) {
6:         do
7:           i=i+1;
8:         while( a[i] < v );
9:         do
10:          j=j-1;
11:        while( a[j] > v );
12:        if ( i >= j )
13:           break;
14:        x = a[i]; a[i] = a[j]; a[j] = x;
15:     }
16:     x = a[i]; a[i] =a [n]; a[n] = x;
17:     quicksort( m, j );
18:     quicksort( i+1, n );
   }
```



Branch Coverage （分支覆盖）

- Branch coverage for exception handling code

```
1:  try {
2:     in.open(“testfile.txt”)
3:     str = in.readline()
4:     if(str == null)
5:        throw new IOException();
6:     in.close();
7:  }catch (IOException e) {
8:     system.out.println(“hello!”);
9:  }finally{
10:    system.out.println(“hello!”);
11: }
12: …
```



How many test cases do we need to cover every edge (branch coverage)?

“Happy” path: 1,2,3,4,6,10,12
Exception at 2: 1,2,8,10,12 and 1,2,10
Exception at 3: 1,2,3,8,10,12 and 1,2,3,10
Throw at 5: 1,2,3,4,5,8,10
Exception at 6: 1,2,3,4,6,8,10,12 and 1,2,3,4,6,10

Loop Coverage (循环覆盖)

Find test cases that achieve 100% loop coverage for the following code:

```
i = strlen(fname) -1;
while (i>0 && fname[i] != '.') {i--;}
```

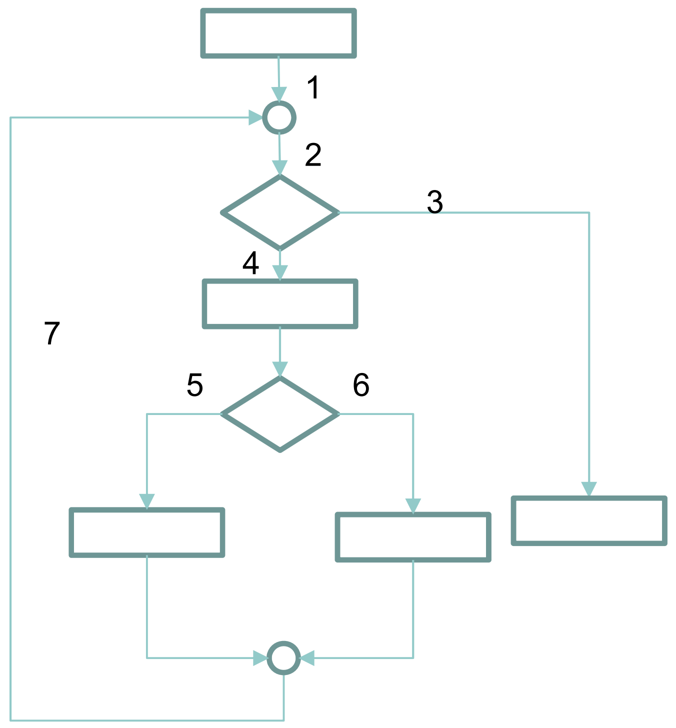
To do “loop coverage”, list some of the possible test cases:

(0 times) →	fname contains: “report1.”
(1 time) →	fname contains: “report1.t”
(more than 1 time) →	fname contains: “report1.txt”
(max – 1 times) →	fname contains: “a.txt”
(max times) →	fname contains: “report1”

Note: max loop count here is the *length of the string minus 1*

Basis Path Coverage (基本路径覆盖)

```
int i=20;
while (i<10) {
   System.out.printf("i is %d", i);
   if (i%2 == 0){
      System.out.println("even");
   } else {
      System.out.println("odd");
   }
}
```



- The size of basis path set (Cyclomatic number)?
  - E – N + 2 = 10 – 9 = 3
- **Example:**

  Path1 = 1,2,3

  Path2 = 1,2,4,5,7,2,3

  Path3 = 1,2,4,6,7,2,3

Consider Path5: 1,2,4,5,7,2,4,6,7,2,3

Path3 - Path1 = 4,5,7,2

Path2 - Path1 = 4,6,7,2

Path1 + **4,5,7,2** + **4,6,7,2** =

1,2,**4,5,7,2**,**4,6,7,2**,3 = Path5

Therefore:

Path 5 = (Path3-Path1)+(Path2-Path1) + Path1

Subsumption Relation

- We say that coverage criterion A subsumes coverage criterion B if test suite that satisfies A must at the same time satisfies B.

$$\begin{tikzcd}
& \text{All path coverage} \arrow[dl] \arrow[dr] & \\
\text{Basis path coverage} \arrow[d] & & \text{Loop coverage} \\
\text{Branch coverage} \arrow[d] & & \\
\text{Statement coverage = Basic block coverage = Line coverage} & &
\end{tikzcd}$$

Types of logic coverage

- Decision coverage (i.e. branch coverage) 判定覆盖（也就是分支覆盖）
- Condition coverage 条件覆盖
- Short-circuiting condition coverage 短路求值语义下的条件覆盖
- Condition/decision coverage (C/D) 条件判定覆盖
- Multiple-condition coverage 条件组合覆盖
- Modified condition/decision coverage (MC/DC)修改的条件判定覆盖

Condition Coverage

- Condition coverage concerns the coverage of each condition taking both `true` and `false`.
  - Does not consider constant condition, such as (true) and (x==x).
  - Does not consider short-circuiting.
    - Apply to languages without short-circuiting, e.g. Visual Basic
  - Condition coverage does not subsume decision coverage.
  - What is the number of test cases to achieve 100% condition coverage?
    - Always 2

```
if ( (A || B) && C ) { …}
else {…}
```

Test suite that satisfy condition coverage:

A=true, B=false, C=false
A=false, B=true, C=true

```
if(A && B) {…}
else {…}
```

Test suite that satisfy condition coverage:
A=true, B=false
A=false, B=true

Problem: branch coverage not achieved

Short-circuiting Condition Coverage

- The same with condition coverage, except that we now require each condition *being actually evaluated* to `true` and `false`.
  - Apply to languages with short-circuiting, e.g. C, C++, Java
  - Short-circuiting condition coverage subsumes decision coverage.
    - Why? Think about it.

```
if ( (A || B) && C ) { …}
else {…}
```

Test Cases for Condition Coverage:

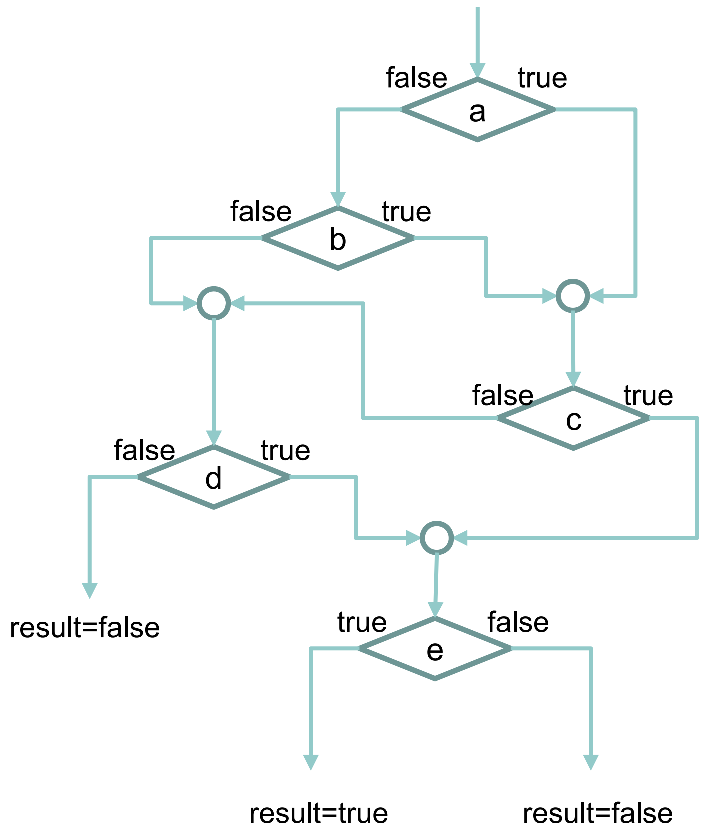
A=true, B=false, C=false
A=false, B=true, C=true

```
if ( (A || B) && C ) { …}
else {…}
```

Test Cases for Short-circuiting Condition Coverage:

A=true, B=not-eval, C=false
A=false, B=true, C=true
A=false, B=false, C=not-eval

Example



((a||b) && c || d) && e

Three test cases to satisfy short-circuiting condition coverage (cover all edges):

```
a=false, b=false, c=-,     d=false, e=-
a=false, b=true,  c=false, d=true,  e=true
a=true,  b=-,     c=true,  d=-,     e=false
```

Notes:

1) it is weaker than MC/DC (need at least 6).
2) it is stronger than condition coverage (need 2).

Questions:

*What is the **minimal** number of test cases and how to find them?*
Transform it into the minimal flow problem and use the Ford-Fulkerson algorithm to solve it. Read: *http://bioinformatics.oxfordjournals.org/content/suppl/2012/04/29/bts258.DC1/supplemental_material_v1_1.pdf*

*Why short-circuiting condition coverage **subsumes** decision coverage?*

Modified Condition/Decision Coverage

- MC/DC requires that for each condition, both its evaluation `true` and `false` are shown to determine the outcome of the decision.
  - That is, the outcome of a decision changes as a result of changing a single condition.
  - Does not consider short-circuiting.
    - (A || B) && C: **A=false, B=true, C=false**
  - For each condition *C*, select two test cases such that the truth values of all the conditions, except for *C*, are the same, and the decision outcome evaluates to be `true` for one and `false` for the other.

Example of MC/ DC Coverage

With these values for G and (x<y), (a>b) determines the value of the predicate

With these values for (a>b) and (x<y), G determines the value of the predicate

With these values for (a>b) and G, (x<y) determines the value of the predicate

((a > b) or G) and (x < y)

1	T	F	T	T
2	F	F	T	F
3	F	T	T	T
4	F	F	T	F
5	T	T	T	T
6	T	T	F	F

duplicate

Subsumption Relation of Logic Coverage

- We say that coverage criterion A subsumes coverage criterion B if test suite that satisfies A must at the same time satisfies B.

$$\begin{tikzcd}
& \text{Multiple condition coverage} \arrow[d, Rightarrow] & \\
& \text{MC/DC coverage} \arrow[d, Rightarrow] & \\
& \text{Short-circuiting condition coverage} \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
\text{Condition coverage} & & \text{Branch coverage}
\end{tikzcd}$$

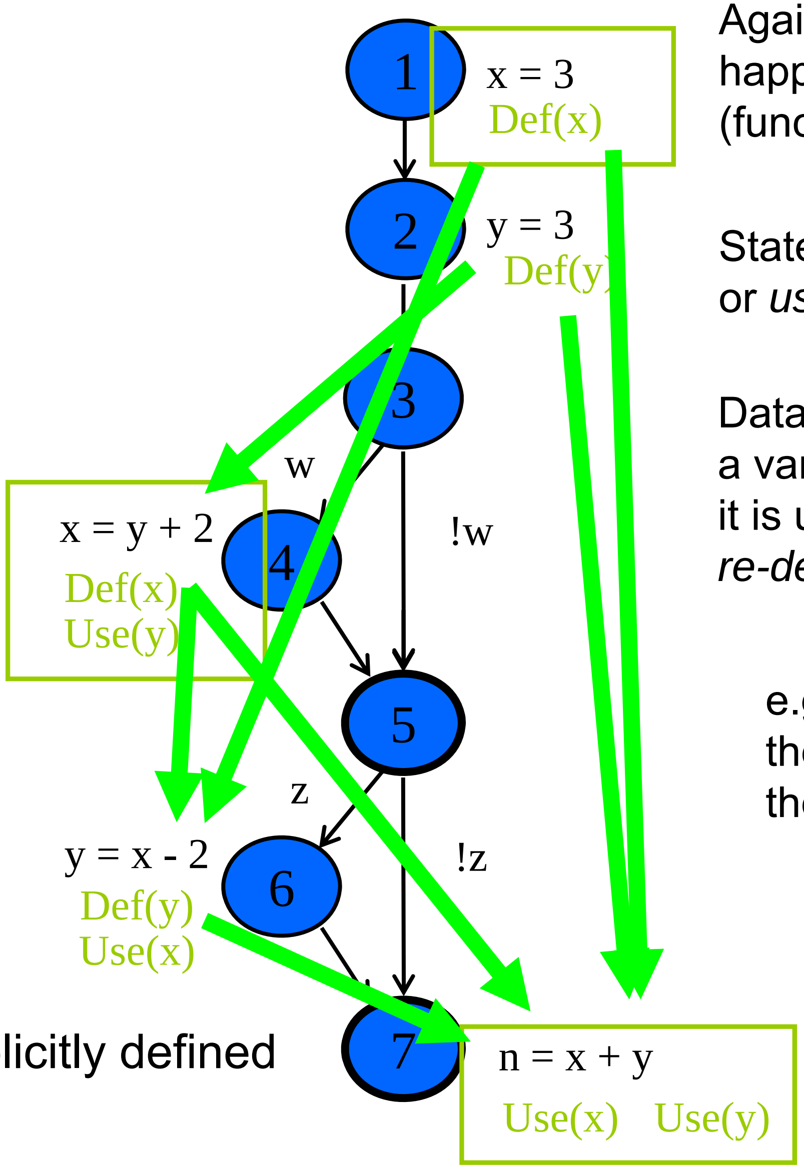
What is *dataflow*?

```
x = 3;
y = 3;

if (w) {
   x = y + 2;
}

if (z) {
  y = x – 2;
}

n = x + y
```



Again, we only care about what happens in one single code unit (function or method).

Statements can *defined* or *used* variables

Data flow occurs from where a variable is *defined* to where it is *used*, without any intervening *re-definitions*

e.g., in this path: 1 2 3 5 6 7, the definition of x at 1 reaches the use of x at 7.

In this path: 1 2 3 4 5 6 7 the definition of x at 1 fail to reach the use of x at 7 because of the redefinition at 4

The dataflow is implicitly defined by the CFG

Operations on Variables

- ***Definition***: A value is written into a variable
  - ***Example***: **`x=0;`** `def(x)`
  - ***Example***: **`*y=1;`** `def(*y)`
  - ***Example***: **`scanf("%d %d", &x, &y);`** `def(x), def(y)`
  - ***Example***: **`int x[10];`** `def(x), def(x[0]), …def(x[9])`
- ***Use***: The value of the variable is read
  - ***Example***: **`y=x+1;`** `use(x)`
  - ***Example***: **`*y=1;`** `use(y)`
  - ***Example***: **`k=*y+1;`** `use(y), use(*y)`
- ***Undefinition***: A variable is not longer accessible
  - ***Example***: **`free(p);`** `undef(*p)`
- A statement can involve more than one operations

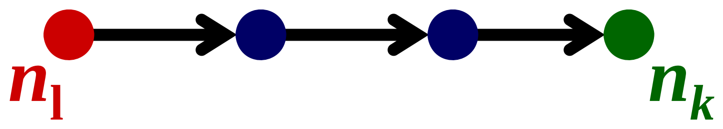
Two Types of use

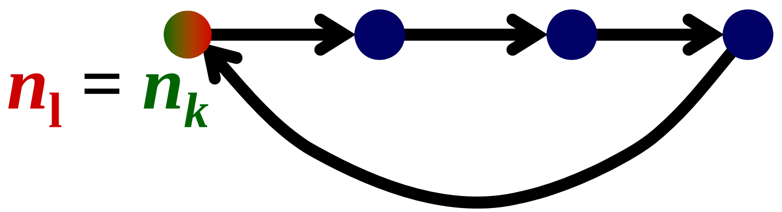
- ***Computation use* (*c-use*):** The value of the variable directly affects a computation or is part of an output
  - *Example 1:* `y = x + 1;` c-use(x)
  - *Example 2:* `printf(“%d”,x);` c-use(x)

- ***Predicate use* (*p-use*):** The value of the variable directly affects a control flow (and may indirectly affect a computation)
  - *Example 1:* `if (x == 0) { y = 1; }` p-use(x)
  - *Example 2:* `switch(c) {case 1:… default:…}` p-use(c)
  - *Example 3*: `if(A[i+1]>0)return;` p-use(A), p-use(i), p-use(A[i+1])

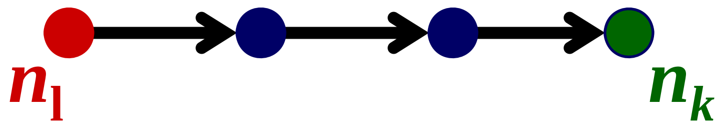
Du-Path

- A path ($n_1, n_2, ..., n_k$) is a ***du-path for variable x*** if n_1 define x and
  - **either** n_k has a **c-use** of x and ($n_1, n_2, ..., n_k$) is a def-clear simple path with respect to x





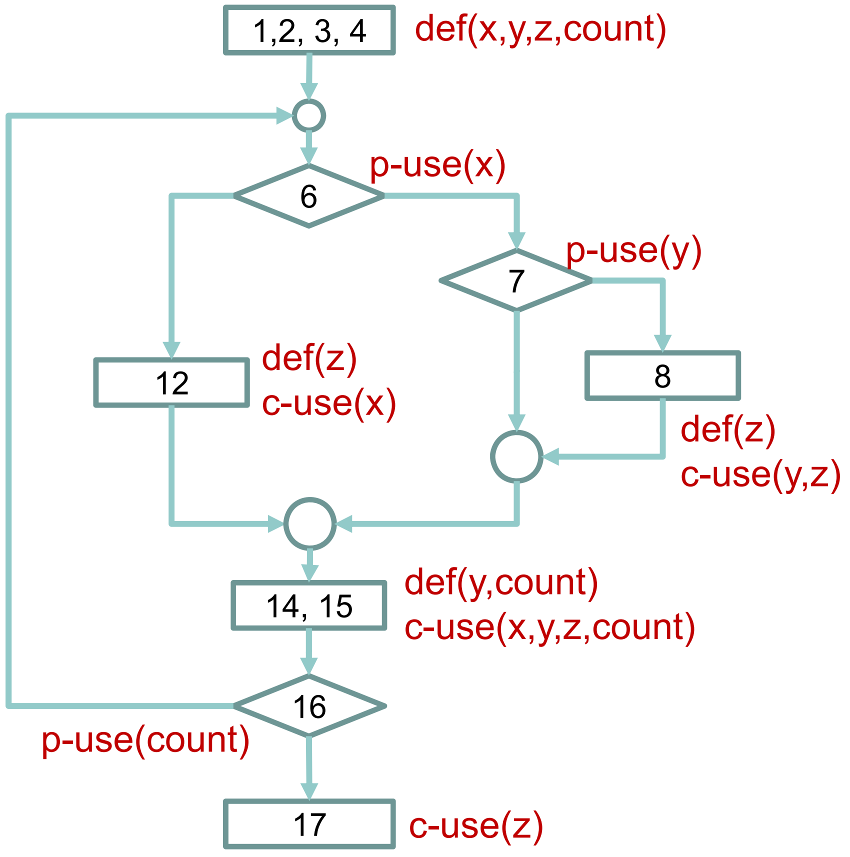
  - **or** n_k has a p-use of x and ($n_1, n_2, ..., n_k$) is a def-clear loop-free path with respect to x.



Any CFG can have only finite number of du-paths.

Quiz 6: c-use, p-use, def, and du-path

```
1    begin
2      float x, y, z=0.0;
3      int count;
4      input (x, y, count);
5      do {
6        if (x≤0) {
7          if (y≥0) {
8            z=y*z+1;
9          }
10       }
11       else{
12         z=1/x;
13       }
14       y=x*y+z
15       count=count-1
16     while (count>0)
17     output (z);
18   end
```



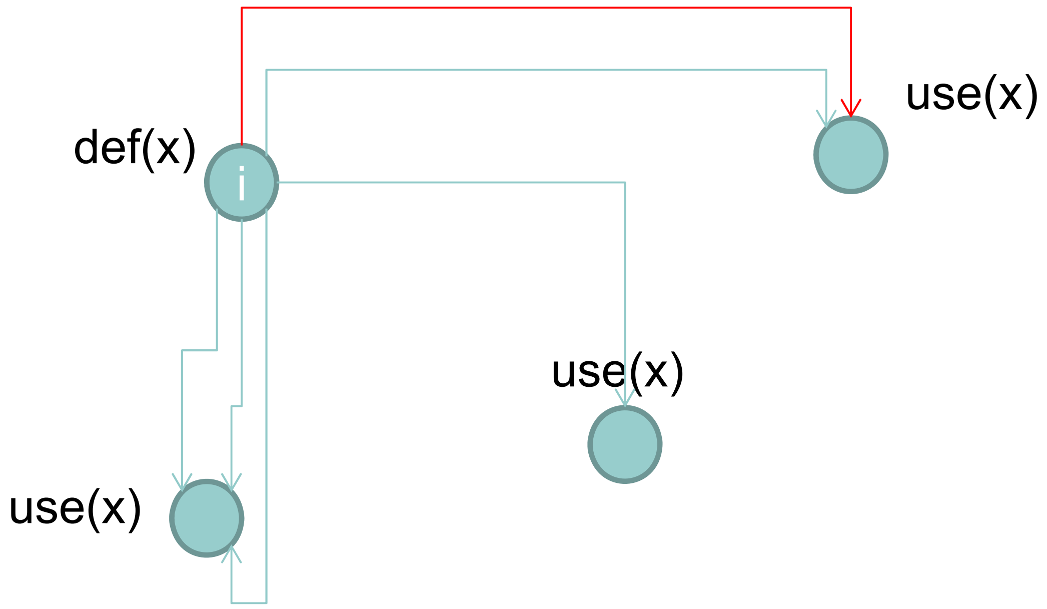
du-path for x:

1,2,3,4,6
1,2,3,4,6,12
1,2,3,4,6,12,14
1,2,3,4,6,7,14
1,2,3,4,6,7,8,14
1,2,3,4,6,12,14,15,16,17
1,2,3,4,6,7,14,15,16,17
1,2,3,4,6,7,8,14,15,16,17

Mark c-use, p-use, and def on nodes in the CFG
*Then find **all** du-paths for x.*

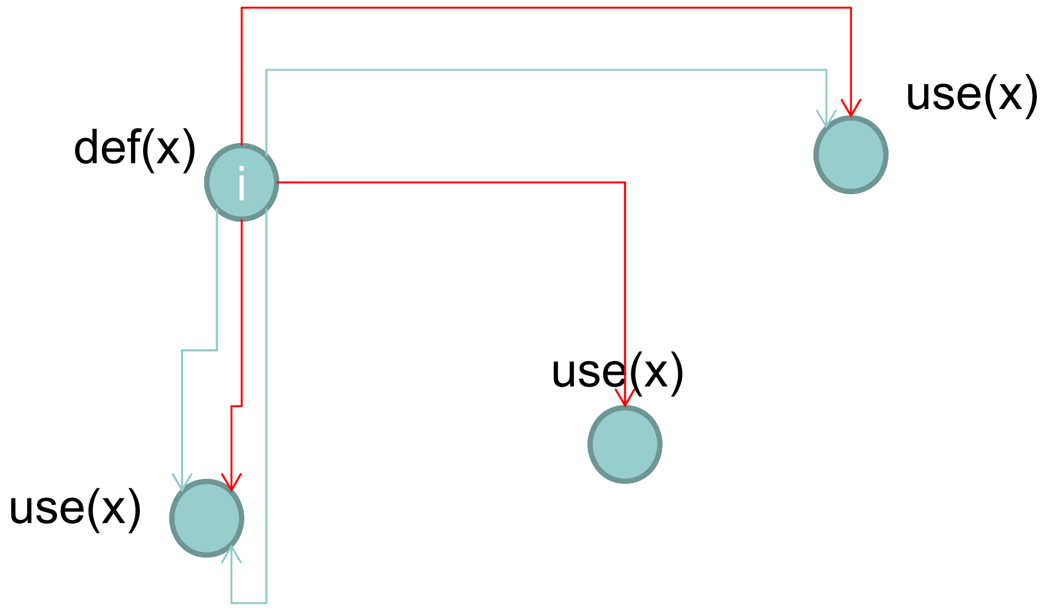
All-Defs Coverage

- For each node i that contains a definition to variable x, the ***all-defs coverage*** requires that the test suite covers at least one feasible du-path of x that starts from i.
  - If none of such du-paths for x are feasible or there is no du-paths for x at all (e.g. define but never use), we don't need to cover anything.



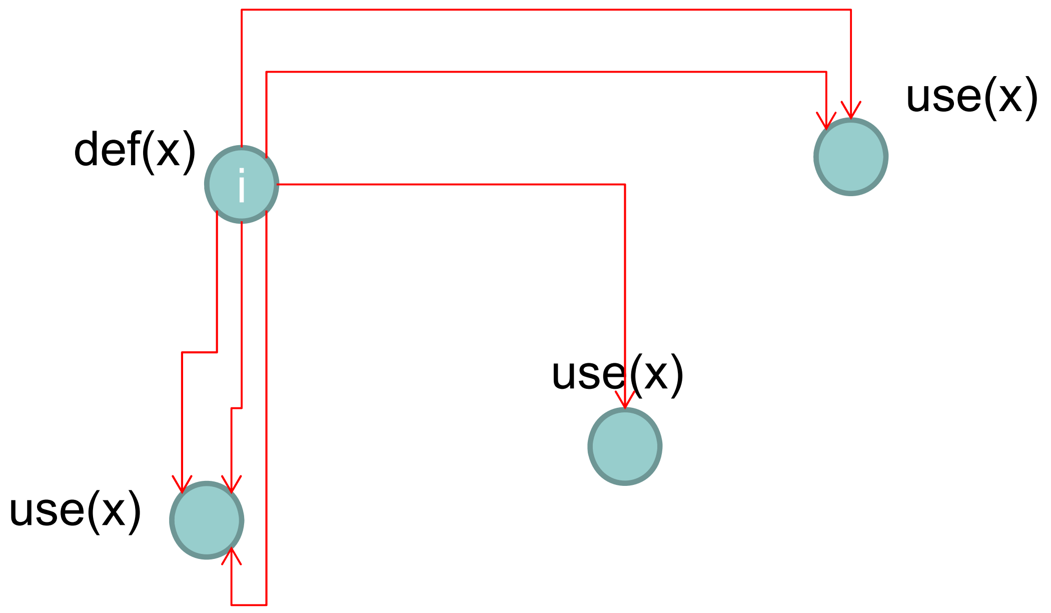
All-Uses Coverage

- For each node i in the CFG that contains a definition to variable x, ***all-uses coverage*** requires that for every use of x in the CFG, the test suite covers at least one feasible du-path of x that starts from i and ends at that use of x.
  - Again, if none of the du-paths of x that starts from i to that use of x are feasible, we don't need to cover anything.



All-Du-Paths Coverage

- For each node i in the CFG that contains a definition to variable x, the ***all-du-paths coverage*** requires that for every use of x in the CFG, the test suite covers all feasible du-path of x that starts from i and ends at that use of x.

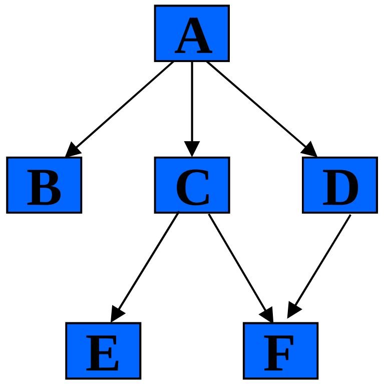


Subsumption Relationships

$$\begin{tikzcd}
 & & \textbf{All path coverage} \arrow[d, Rightarrow] & & \\
 & & \text{All-du-paths} \arrow[d, Rightarrow] & & \\
 & & \text{All-uses} \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & & \\
 & \text{All-c-uses/some-p-uses} \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & & \text{All-p-uses/some-c-uses} \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
\text{All-c-uses} & & \text{All-defs} \arrow[d, Rightarrow] & & \text{All-p-uses} \\
 & & \textbf{All-edges (branch coverage)} \arrow[d, Rightarrow] & & \\
 & & \textbf{All-nodes (statement coverage)} & &
\end{tikzcd}$$

Function/Call Coverage

- *Call Graph*
  - Nodes : Functions (or methods)
  - Edges : Calls to functions (*call-site*)
  - Compare it with control-flow graph (CFG)



Example call graph

Function coverage: cover every function at least once (every node in the call graph)

Call coverage: cover every call-site at least once (every edge in the call graph)

Mutant Coverage

- **Mutant**: A program with a seeded fault
- **Mutation coverage = D/N**

  D = Number of killed mutants

  N = Total number of mutants
- **Mutation testing**: use mutant coverage as the test adequacy criteria to design the test suite.

Program P:
```
cin>>y>>z;
x=y+z;
cout<<x;
```

Mutant 1:
```
cin>>y>>z;
x=y*z;
cout<<x;
```

Mutant 2:
```
cin>>y>>z;
x=y-z;
cout<<x;
```

Mutant 3:
```
cin>>y>>z;
x=y+y;
cout<<x;
```

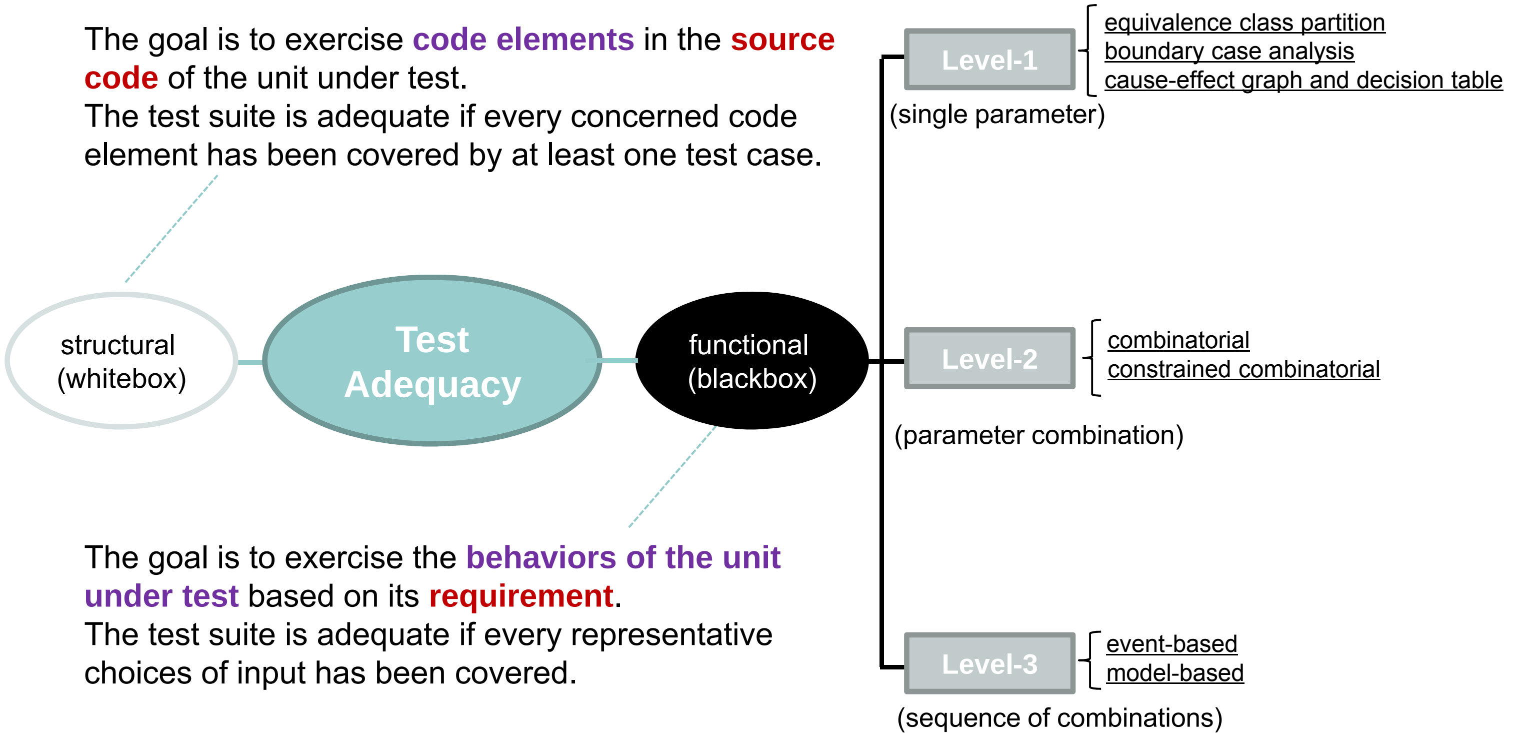
test suite:
1: y=2, z=2
2: y=3, z=3

kill mutant 1
kill mutant 2
does not kill mutant 3

mutant coverage: 66.7%

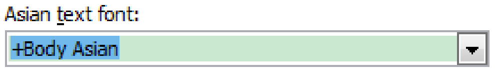
Knowledge Point #6

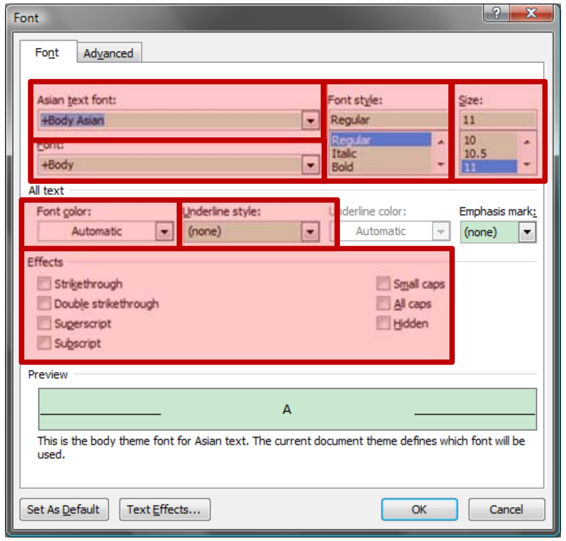
- Black-box Test adequacy


The goal is to exercise **code elements** in the **source code** of the unit under test.
The test suite is adequate if every concerned code element has been covered by at least one test case.
structural (whitebox)
Test Adequacy
functional (blackbox)
The goal is to exercise the **behaviors of the unit under test** based on its **requirement**.
The test suite is adequate if every representative choices of input has been covered.
Level-1
equivalence class partition
boundary case analysis
cause-effect graph and decision table
(single parameter)
Level-2
combinatorial
constrained combinatorial
(parameter combination)
Level-3
event-based
model-based
(sequence of combinations)

Three-levels of Test Adequacy

- Level 1: ***Single*** input parameter
  - Cover the representative choices for **one single input parameter**.
  - e.g. the font type. (宋体 or 黑体 or Arial or ….)
- Level 2: ***Combination*** of input parameters
  - Cover the representative **combinations of multiple parameters**.
  - e.g. settings in the whole font dialog. (宋体-加粗-11号-红色 or 宋体-普通-小四-黑色 or 黑体-加粗-12号-黄色 or …)
- Level 3: ***Sequence*** of parameter combinations
  - Cover the representative **sequences of parameter combinations**.
  - e.g. font settings for multiple paragraphs. (e.g. 为不同的段落设置不同的字体)。


Asian text font: +Body Asian


Font

Font | Advanced

Asian text font: +Body Asian
Font: +Body
Font style: Regular (Regular, Italic, Bold)
Size: 11 (10, 10.5, 11)

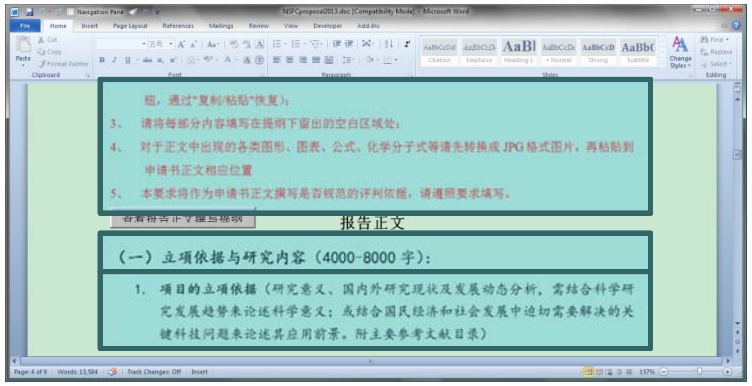
All text
Font color: Automatic
Underline style: (none)
Underline color: Automatic
Emphasis mark: (none)

Effects
- Strikethrough
- Double strikethrough
- Superscript
- Subscript
- Small caps
- All caps
- Hidden

Preview
A

This is the body theme font for Asian text. The current document theme defines which font will be used.

Set As Default | Text Effects... | OK | Cancel


NSFCproposal2013.doc [Compatibility Mode] - Microsoft Word

钮，通过"复制/粘贴"恢复)；
3、请将每部分内容填写在提纲下留出的空白区域处；
4、对于正文中出现的各类图形、图表、公式、化学分子式等请先转换成 JPG 格式图片，再粘贴到申请书正文相应位置
5、本要求将作为申请书正文撰写是否规范的评判依据，请遵照要求填写。

报告正文

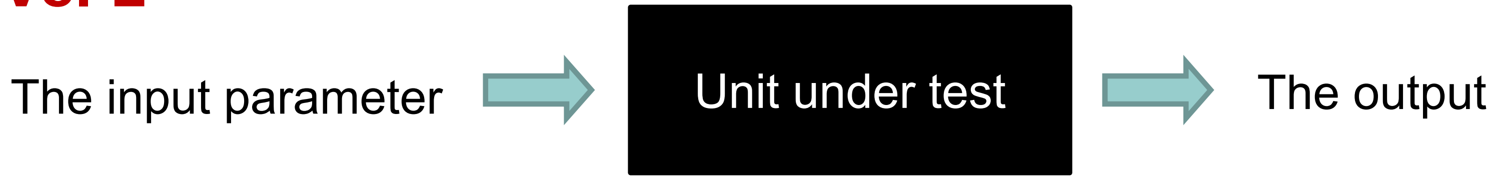
（一）立项依据与研究内容（4000-8000 字）：

1、项目的立项依据（研究意义、国内外研究现状及发展动态分析，需结合科学研究发展趋势来论述科学意义；或结合国民经济和社会发展中迫切需要解决的关键科技问题来论述其应用前景。附主要参考文献目录）

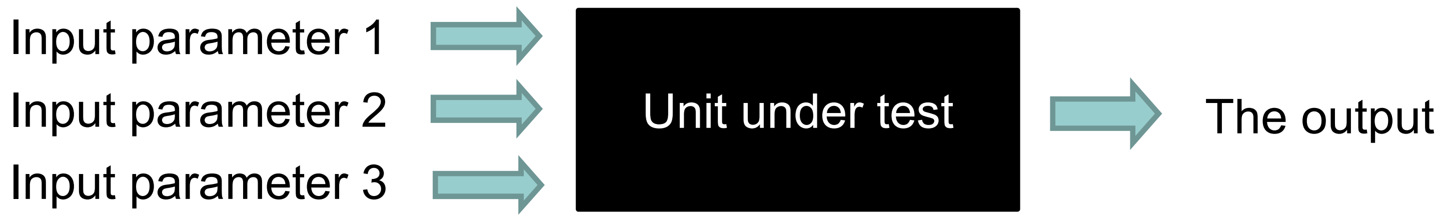
Words: 13,584 | Track Changes: Off | Insert | 157%

Illustration of The Three Levels

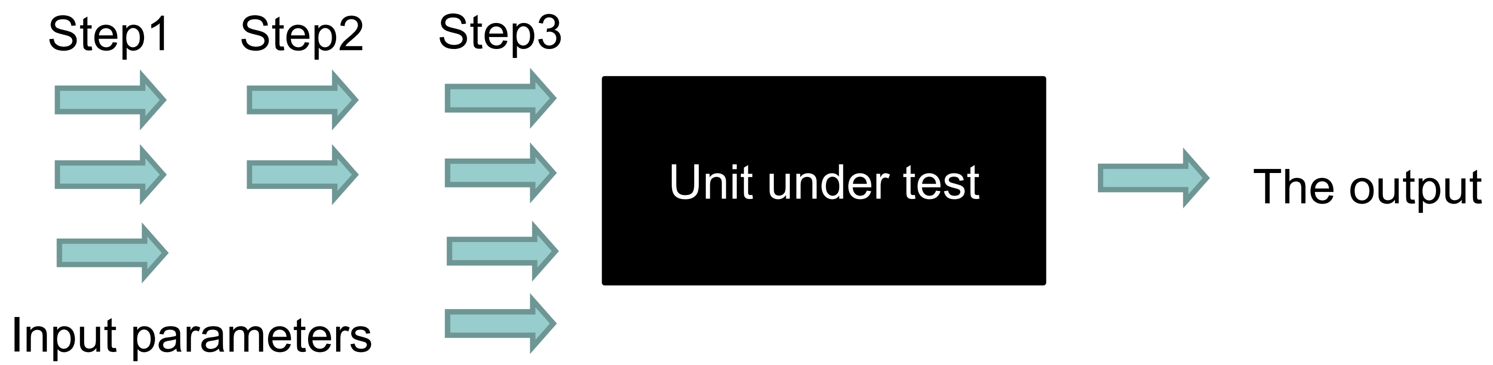
Level 1



Level 2



Level 3



Level-1 Techniques

- **Equivalence class partitioning （ECP, 等价类划分）**
    - Partition the input domain of the parameter into equivalence classes
    - Adequacy criterion: cover each partition at least once

- **Boundary value analysis （BVA, 边界值分析）**
    - Analyze the boundary cases for each equivalence class in ECP.
    - Adequacy criterion: cover each boundary case at least once.

- **Cause-effect graph and decision table（因果图和决策表）**
    - Analyze the causal relation between input and output as *edges*.
    - Adequacy criterion: cover each edge at least once.

Cause-Effect Graph

- Cause-effect graph models dependency between program input conditions (known as *causes*) and output condition (known as *effects*).
  - A cause is any condition in the requirements that may affect the program output.
  - An effect is the response of the program to some combination of input conditions.
- Example: the input parameter is a list of integers
  - Display message A if input contains 1 and 2.
  - Display message B if input contains 2 and 3.
  - Display message C if input contains 1 or 3.
  - Display message D if input contains 1, 2, and 4.
  - One and only one of {3, 4} must be in the input.

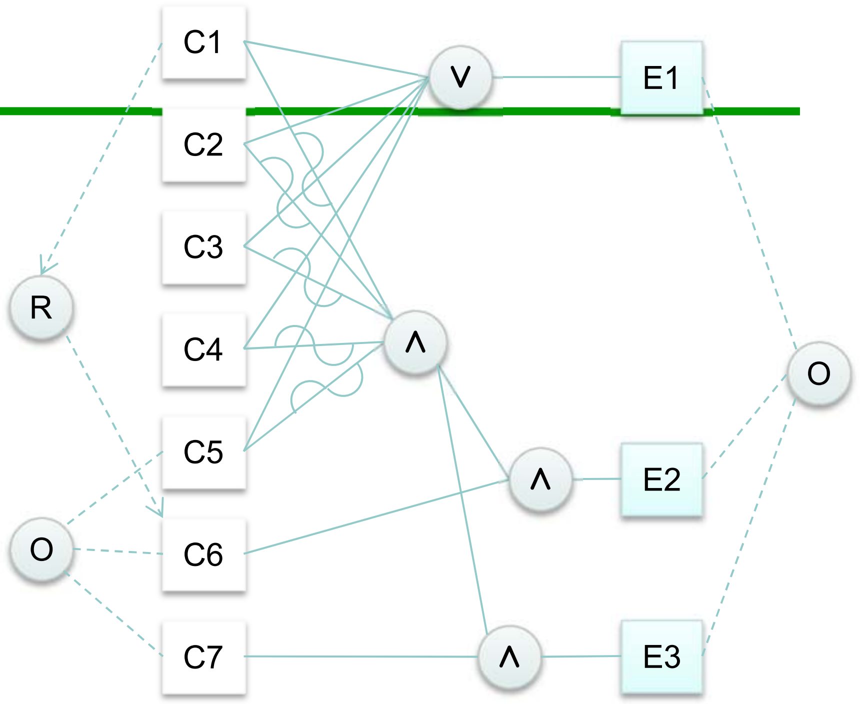
Causes:
input conditions

Effects:
output conditions

Constraint between causes

Example

- Movement of horse in Chinese chess（象棋中马的移动）
  - Input Parameter: the place where you want to move your horse.
- The effects:
  - E1: Illegal move
  - E2: Legal move into an empty space
  - E3: Legal move into an enemy’s piece
- The causes:
  - C1: The place is outside the board.
  - C2: Does not follow the rule （非’日’字）.
  - C3: Another piece prevents the movement （绊马腿）.
  - C4: The move results in the king being checkmated.
  - C5: The place contains one’s own piece.
  - C6: The place is empty.
  - C7: The place contains enemy’s piece.



C1	**T**	F	F	F	F	F	F	F	**T**	F	F	F
C2	F	**T**	F	F	F	F	F	F	F	F	F	**T**
C3	F	F	**T**	F	F	F	F	F	F	F	F	F
C4	F	F	F	**T**	F	F	F	F	F	F	F	F
C5	F	F	F	F	**T**	F	F	F	F	F	F	F
C6	**T**	**T**	**T**	**T**	F	**T**	**T**	F	**T**	F	**T**	F
C7	F	F	F	F	F	F	F	**T**	F	**T**	F	**T**
E1	•	•	•	•	•				•			•
E2						•	•				•	
E3								•		•		

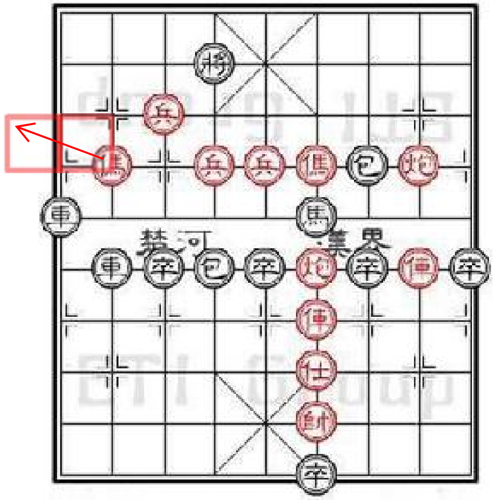
Example

- Movement of horse in Chinese chess（象棋中马的移动）
  - Input Parameter: the place where you want to move your horse.
- The effects:
  - E1: Illegal move
  - E2: Legal move into an empty space
  - E3: Legal move into an enemy's piece
- The causes:
  - C1: The place is outside the board.
  - C2: Does not follow the rule （非'日'字）.
  - C3: Another piece prevents the movement （绊马腿）.
  - C4: The move results in the king being checkmated.
  - C5: The place contains one's own piece.
  - C6: The place is empty.
  - C7: The place contains enemy's piece.

Reduce redundant test cases

C1	**T**	F	F	F	F	F	F	F	**T**	F	F	F
C2	F	**T**	F	F	F	F	F	F	F	F	F	**T**
C3	F	F	**T**	F	F	F	F	F	F	F	F	F
C4	F	F	F	**T**	F	F	F	F	F	F	F	F
C5	F	F	F	F	**T**	F	F	F	F	F	F	F
C6	**T**	**T**	**T**	**T**	F	**T**	**T**	F	**T**	F	**T**	F
C7	F	F	F	F	F	F	F	**T**	F	**T**	F	**T**
E1	•	•	•	•	•				•			•
E2						•	~~•~~				~~•~~	
E3								•		•		

Generate one test case for each column
Example: column1 move from (1,3) to (-1,2)



Level-2 Techniques

- **Combinatorial Coverage** （组合覆盖）
  - Adequacy criterion: cover each t-way interaction at least once
- **Constrained Combinatorial Coverage**（带约束的组合覆盖）
  - Adequacy criterion: combinatorial coverage subject to additional constraints between parameters

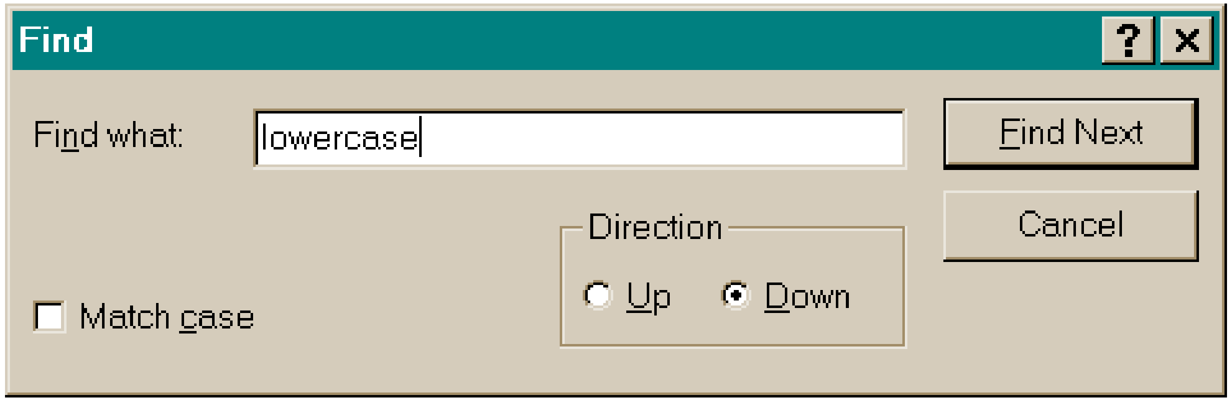
Combinatorial Coverage

- Let p be the number of parameters:
  - Parameters are indexed $1,\ldots, p$.
- For each parameter i, suppose that there are n_i concerned values
  - These values are from the n_i equivalence classes & boundary cases of this parameter.

- **t-way interactions** between parameters $i, i+1,\ldots i+t-1$: the $n_i * n_{i+1} * \ldots * n_{i+t-1}$ value combinations.
  - **Assumption**: parameters are **independent**. That is, the choice of values for any parameter does not affect the choice of values for any other parameter.
  - We will remove this restriction later in *constrained combinatorial coverage*.

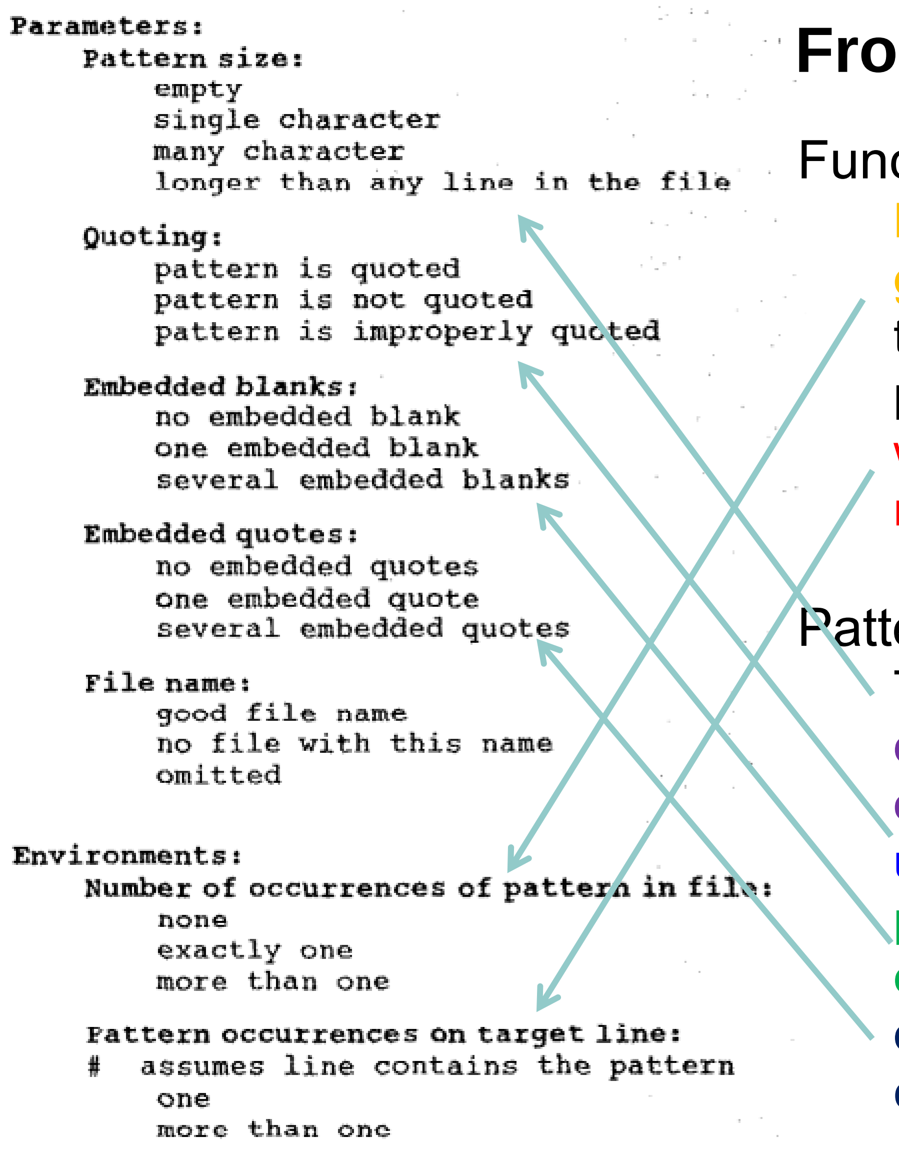
- **t-wise (combinatorial) coverage**: cover all the t-way interactions between any set of *t* parameters.
  - Special case: pairwise (combinatorial) coverage -- covering all two-way interactions

Pairwise Coverage



- Test a simple Find dialog. It takes three inputs:
  - `Find what:` a text string.
  - `Match case:` yes or no
  - `Direction:` up or down
- For the text string, consider three values: “lowercase” and “Mixed Cases” and “CAPITALS”.
- Total combinations: 2*2*3=12
- What are the combinations that satisfy pairwise coverage?
  - LYD, MYU, CYD, LNU, MND, CNU

Category Partition Testing



```
Parameters:
    Pattern size:
        empty
        single character
        many character
        longer than any line in the file

    Quoting:
        pattern is quoted
        pattern is not quoted
        pattern is improperly quoted

    Embedded blanks:
        no embedded blank
        one embedded blank
        several embedded blanks

    Embedded quotes:
        no embedded quotes
        one embedded quote
        several embedded quotes

    File name:
        good file name
        no file with this name
        omitted


Environments:
    Number of occurrences of pattern in file:
        none
        exactly one
        more than one

    Pattern occurrences on target line:
    #  assumes line contains the pattern
        one
        more than one
```

From the requirement

Function:

Locate one or more instances of a given pattern in a text file. All lines in the file that contains the pattern are printed. A line containing the pattern is written only once, regardless of the number of occurrence.

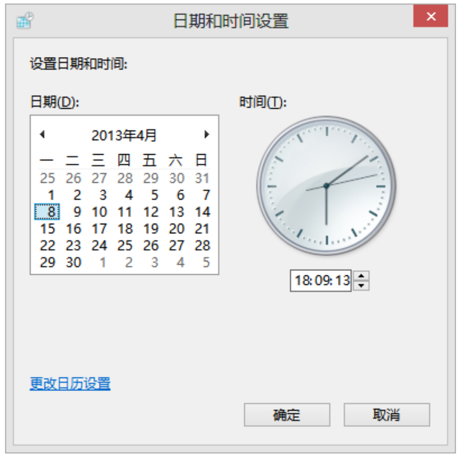
Pattern:

The pattern is any string whose length does not exceed the maximum length of a line in the file. It can be quoted or unquoted. To include a blank in the pattern, the entire pattern must be enclosed in quotes “”. To include the quotation mark in the pattern, two consecutive quotes must be used.

Constrained Combinatorial Coverage

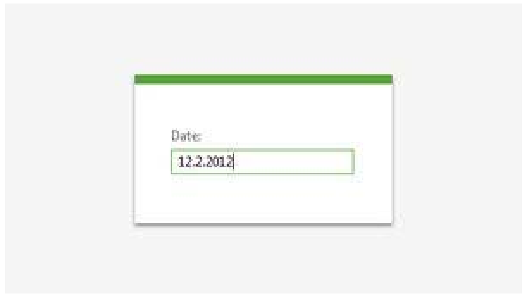
- **Reason 1**: rule out *infeasible* (不可能) combinations
  - They are impossible to input through user interface.
  - Need to differentiate *infeasible* (不可能) from *invalid* (非法).
    - e.g. the combination year=2013, month=2, day=30.

infeasible



vs.

invalid



- Rule out *infeasible combinations*, not *invalid combinations*.
  - On the contrary, we shall deliberately insert invalid combination into the pairwise-coverage test suite, it they have not been covered.

Constrained Combinatorial Coverage

- **Reason 2**: rule out *redundant* (冗余) combinations
  - Some values of one parameter might trigger the system to bypass whatever values the other parameters take.

Example:

```
public static void main (String args[]) {
   if (args.length > 2) {
      Quit.now("Usage: java STS attributes.txt [s|st|ts|t]");
   }
   ...
```

We only need one parameter combination with args.length>2. Others are all redundant, as they exercise exactly the same behavior of the program.

- Constraints can be used to rule out redundant combinations and reduce the size of test suite.

Level-3 Coverage Criteria

- **Event–based Coverage** （基于事件的覆盖）
  - **GUI program = a set of events**
- **State-based Coverage**（基于状态的覆盖）
  - **GUI program = a set of features**

GUI from tester's perspective

- Three different points of view on a GUI program from a tester's perspective
  - GUI program = a set of code units
  - GUI program = a set of events
  - GUI program = a set of features
- The first one: use previous whitebox coverage criteria
  - statements, basic block, basis path, …
- The latter two lead to two new sets of blackbox coverage criteria dedicated to GUI testing.

Event-based

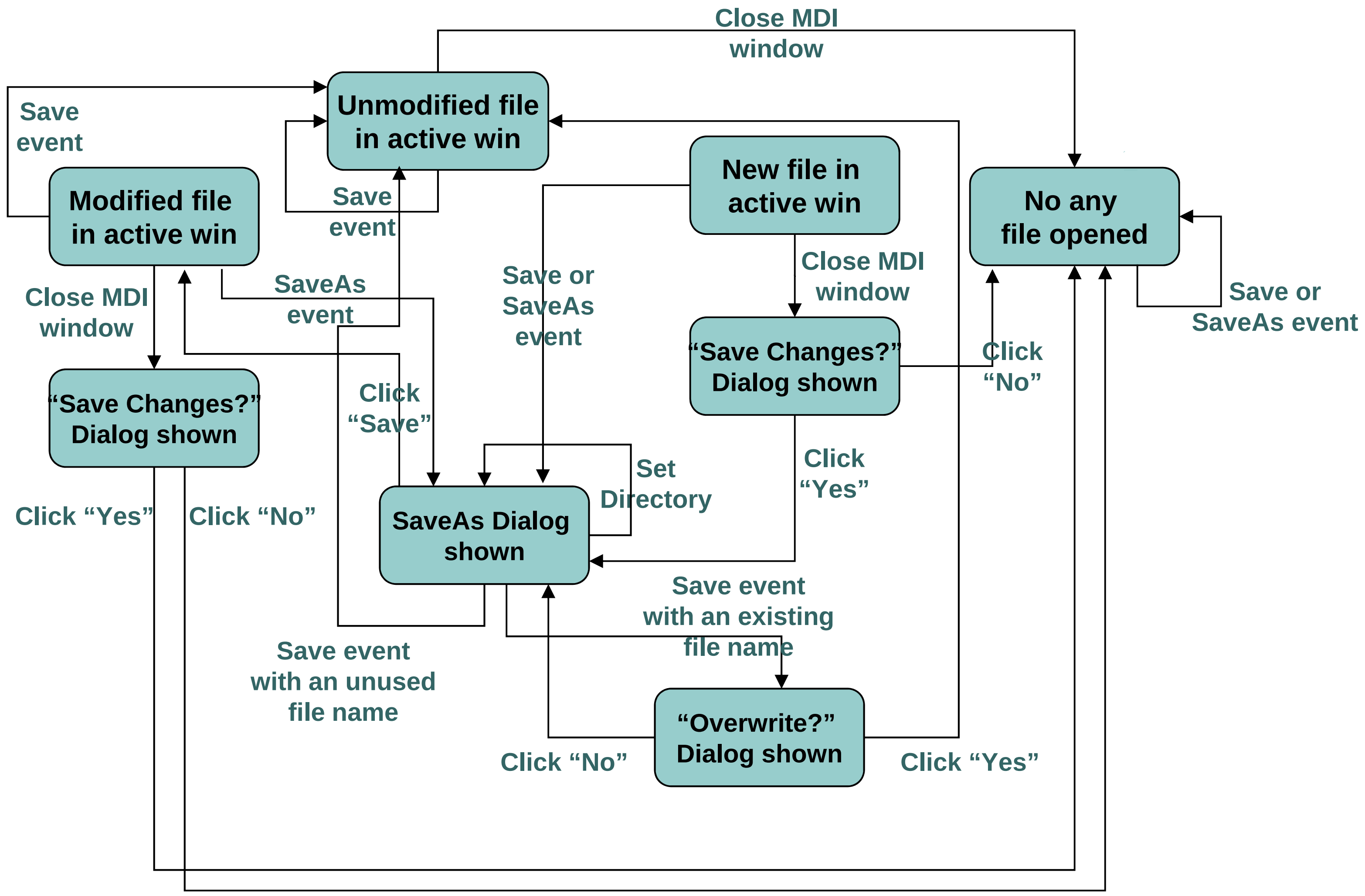
- How to design test suite that achieve event-based test adequacy for a GUI program?
  - **Step 1**: enumerate all windows of the program and build the window hierarchy.
  - **Step 2**: for each window, identify the events that can occur and describe them with operators.
  - **Step 3**: design test cases to cover every representative events.
  - **Step 4:** identify the "follow" relation between events and build the event flow graph.
  - **Step 5**: design test cases to cover intra-window event interactions.
  - **Step 6**: design test cases to cover inter-window event interactions.

Model-based

- Features are first identified by reading the documents.
  - Each feature involves multiple scenarios

- A finite state machine is constructed for each feature to capture all scenarios of this feature.
  - **States** are intermediate states of the GUI windows
  - **Transitions** are events between states that users do to achieve the feature.

FSM for the “open” Feature



Knowledge Point #7

- Test generation
  - Whitebox (based on source code)
    - Symbolic test generation（基于符号执行）
    - Concolic test generation （基于协同执行）
  - Blackbox (based on specification)
    - Random test generation（随机生成）
    - Evolutionary test generation（遗传算法）
    - Grammar-based test generation（基于语法)

Symbolic test generation

- **Goal**: generate test data to cover a specific path
- Generate a set of **path constraints** *C* for a given path *p* such that any test data that satisfy *C* will cover the path *p*.

```
void foo(int x, int y){
1:      int result=0;
2:      if (x>10)
3:          result = x;
4:      if (y>x){
5:          result ++;
}
```

Path to cover: 1→2→3→4→5

Path constraints for this path: (x>10) && (y>x)

Solution to this constraint: (x=11, y=12)

- Apply a *constraint solver* on C to get the actual data
  - Or you can solve the constraint by hand.

Random Test Generation

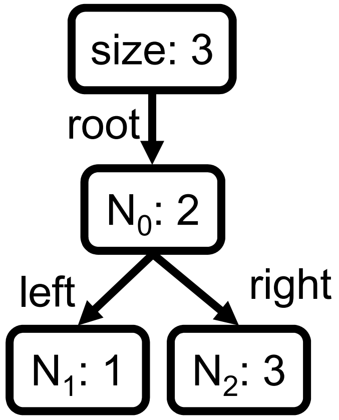
```
public BinarySearchTree {
    Node root;
    int size;

    static class Node {
        Node left, right;
        int value;
    }

    public int remove(int i) {
       .......
    }
}
```

Method under test:
b.remove();

How to randomly generate a binary tree b?

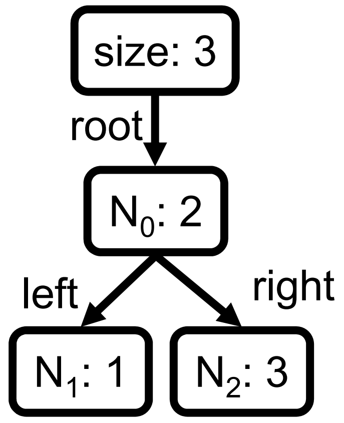


Dumb random test generation will arbitrarily set the value of root and size → most of the results are useless.

Systematic random test generation will try to generate valid binary tree only → randomize the structure of the tree in a valid way.

The key Issue

- The key issue of systematic random test generation is **how to enumerate the valid input space**.
  - This can be difficult when the input is subject to complex validity constraints.



What are the possible choices of valid binary trees?

Validity constraints:

1) All nodes are reachable from root
2) No loop
3) Parent smaller than left child, greater or equal to right child.
4) size is equal to the total number of nodes.

Two Approaches to this Problem

- **Approach 1**: first generate all possible input without considering validity constraints, then rule out invalid ones.
  - Example: first randomly generate all possible object graphs, then rule out those that do not satisfy the validity constraints of binary tree.

- **Approach 2**: start from a valid input, then apply all possible sequence of valid operations
  - An operation is valid if it will only transform one valid input to another valid input.
  - Example: start from an empty binary tree, then iteratively apply different sequence of the 'insert' operation.

Grammar-based test generation

$$
\begin{array}{lll}
\boldsymbol{S} & ::= & \langle Brace\rangle \mid \langle Curly\rangle \mid \varepsilon \\
Brace & ::= & (\ \langle S\rangle\ )\ \langle S\rangle \\
Curly & ::= & \{\ \langle S\rangle\ \}\ \langle S\rangle
\end{array}
$$

A ***derivation*** *is a series of expansion of the grammar that result in a sequence of terminal symbols. It follows that the sequence is a valid sentence of the grammar. We can use this to generate valid sentences.* Example :

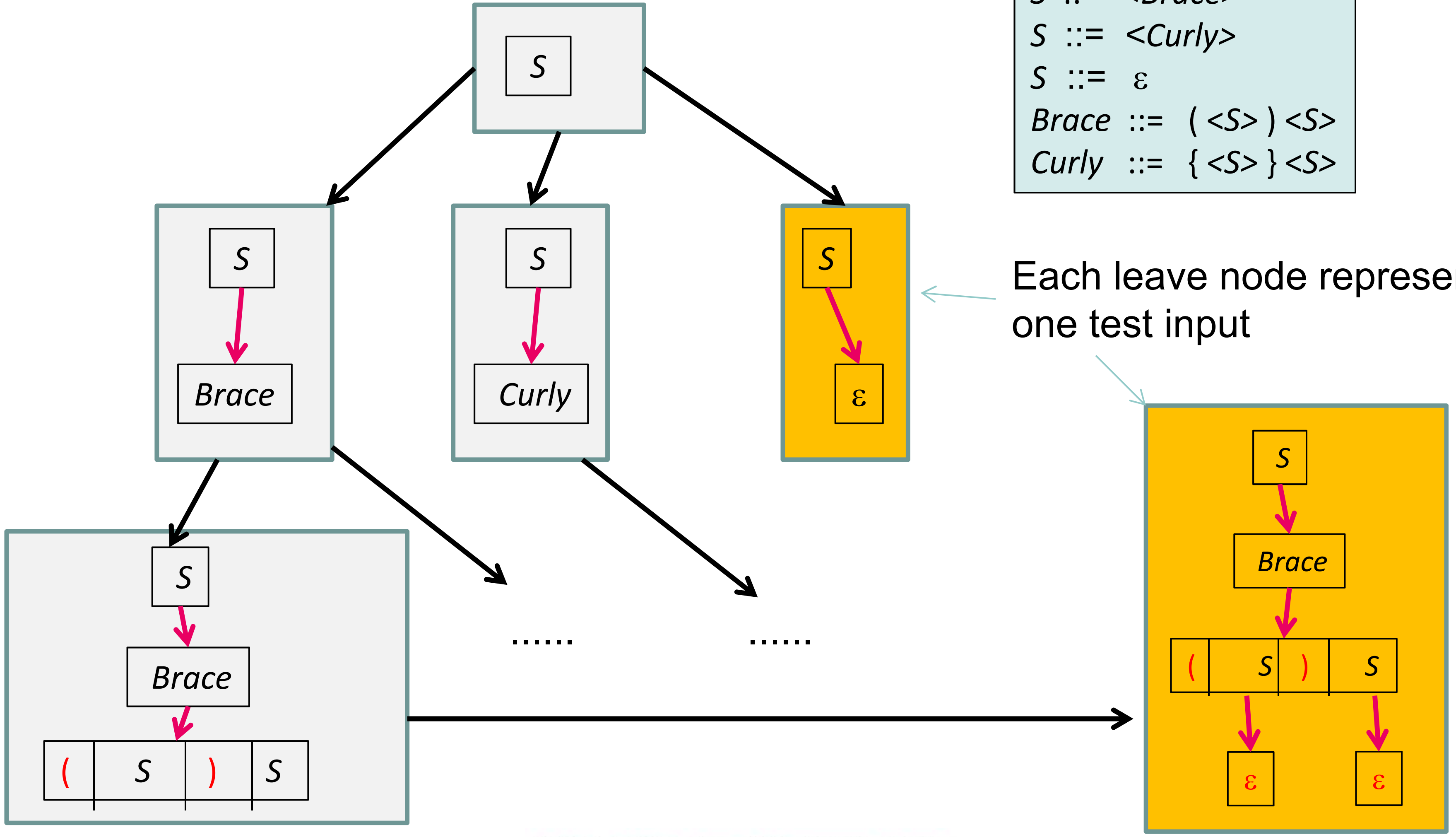
$$
\begin{array}{ll}
S & \rightarrow\ Brace \\
 & \rightarrow\ (\ S\ )\ S \\
 & \rightarrow\ (\ \varepsilon\ )\ S \\
 & \rightarrow\ (\ \varepsilon\ )\ \varepsilon
\end{array}
$$

Depth-first Traversal

- Depth-first traverse of a graph in which node is derivation tree, and edge is one-step derivation relation

```
S      ::=  <Brace>
S      ::=  <Curly>
S      ::=  ε
Brace  ::=  ( <S> ) <S>
Curly  ::=  { <S> } <S>
```

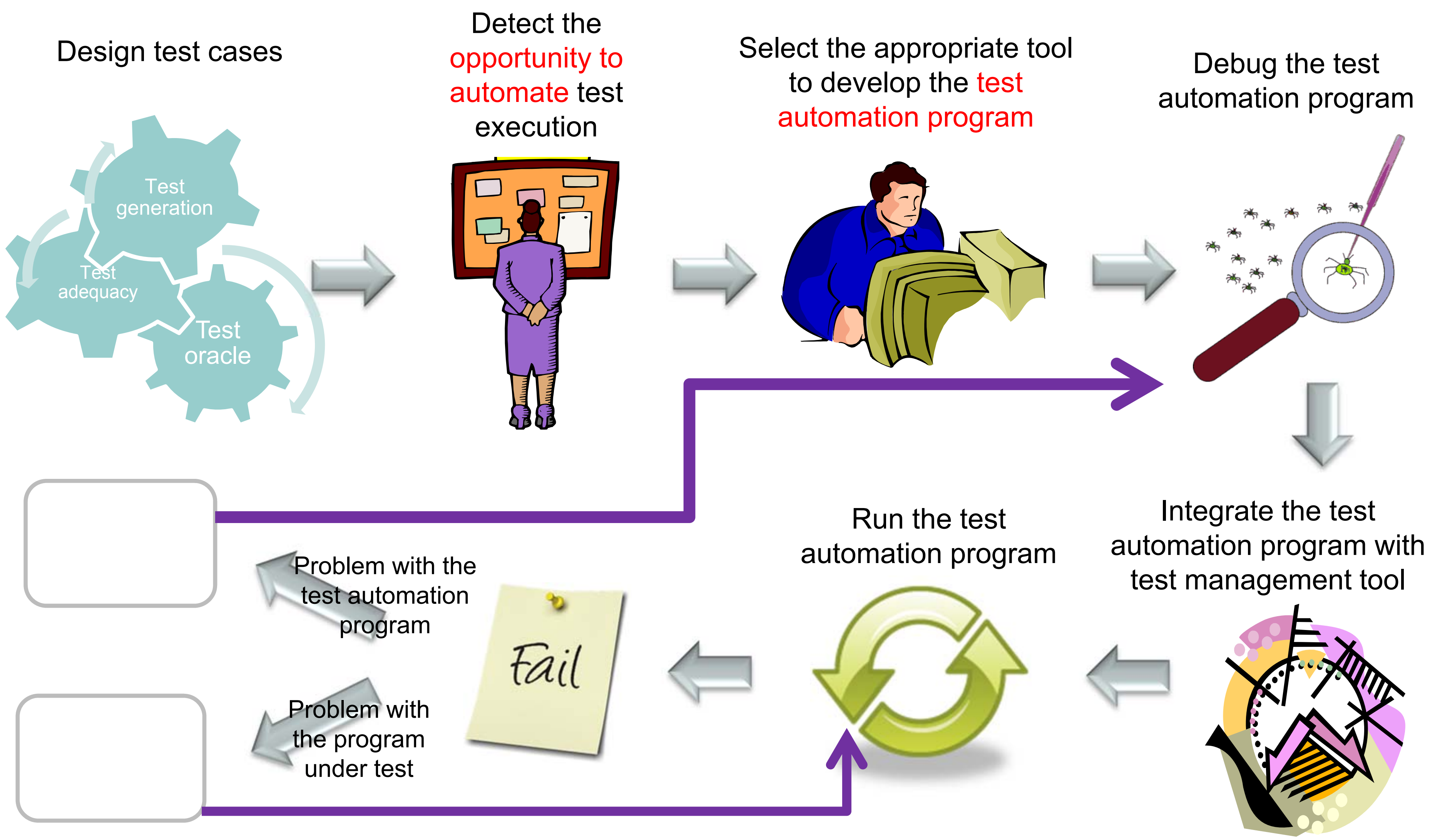
Each leave node represents one test input



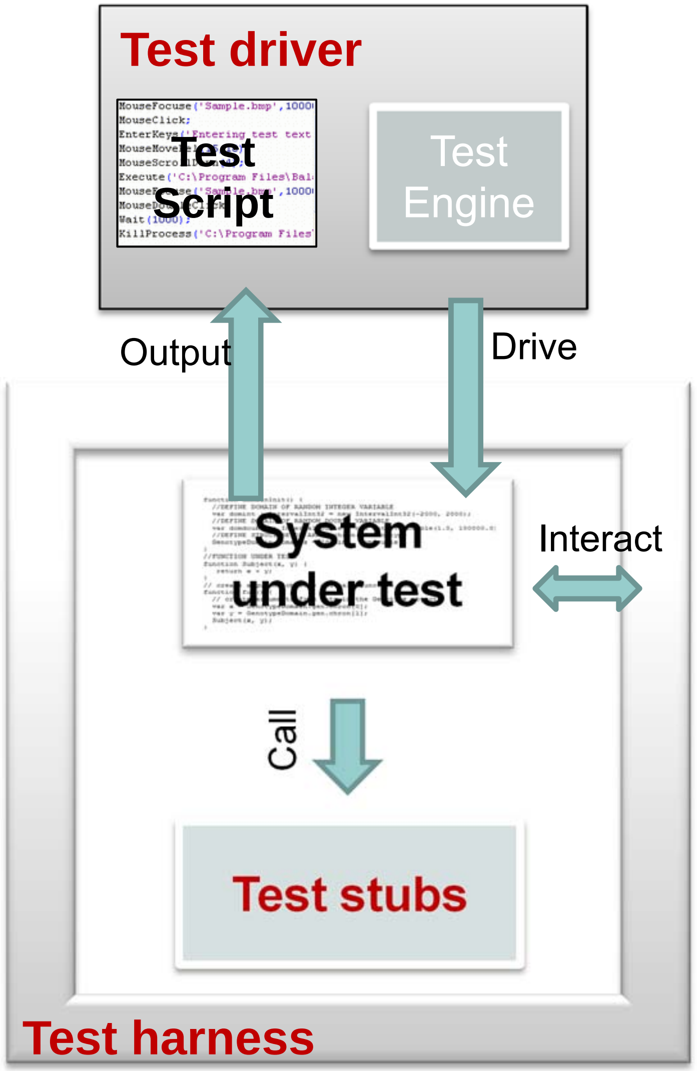
Knowledge Point #8

- Test automation
  - Designing test automation program
  - Capture/replay
  - Structural Test Scripts Development
  - Data-driven test automation
  - Keyword-driven test automation

Test Automation Process


Design test cases
Test generation
Test adequacy
Test oracle
Detect the opportunity to automate test execution
Select the appropriate tool to develop the test automation program
Debug the test automation program
Integrate the test automation program with test management tool
Run the test automation program
Fail
Problem with the test automation program
Problem with the program under test

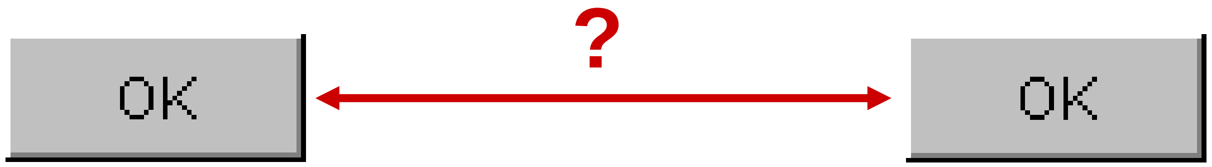
Test Automation Program



- Include three parts:
  - Test driver （测试驱动）
    - Drive test execution and implement test oracle
    - Consist of test engine and test scripts when using test automation tool (e.g. RFT).
  - Test stub （测试桩）
    - Substitute for functions/methods/objects/components/library that the code under test depends on .
  - Test harness （测试套）
    - Substitutes for other parts of the deployed environment (e.g. software simulation of a hardware device)

Key Issues in Test Scripts

- Identification of GUI objects
  - If the tool finds a button during script replay, how does it know that this is the same button it found when recording this script?



- Test oracle
  - How to check expected output?

GUI objects Identification

- Simple way: ***find-by-attribute***
  - Finding objects by matching the values of their properties (e.g. caption, background color, etc.)
  - Put the attribute names and values directly into the test script.
  - Many open source test automation tools only support this way, such as MarathonLTE.
- More general way: ***object map***
  - An intermediate layer that maps each GUI object identifier to a *descriptor*.
    - Weighted multiple properties
    - Indexing
    - Patterns
    - Virtual objects

Checkpoint

- A **checkpoint** is a point in a script that you insert to check the state of the system
  - Called *checkpoint* in QTP and TestComplete
  - Called *verification point* in RFT and Marathon

- A checkpoint serves as your eyes: record a verification point wherever you need to verify expected results
  - The insertion of checkpoint depends on GUI object identification: first specify a GUI object, then specify its expected value.

Structural Test Scripts Development

- Test script development with structured programming constructs.
  - Control statement: conditional/switch statement, loop statement
  - Modular design: functional call, library
  - Exception handling
- Recall: test automation is software development
  - Encapsulation and Reuse

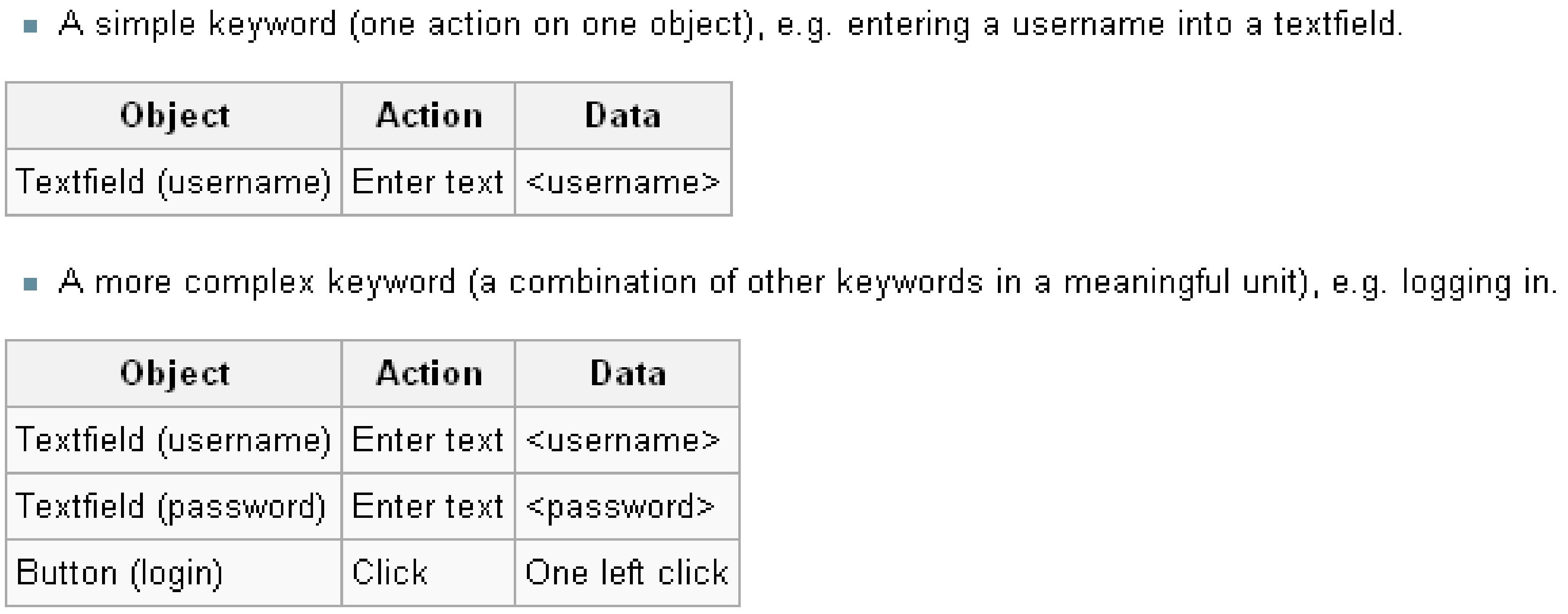
Data-driven test automation

- Data-focused automation. Data is defined in external data source and de-coupled from test script.
- Users define the data sets, while the test engineer develop the test scripts to load the data sets.
- Good fit for testing that features multi-environment, big datasets, and rarely changing test scripts.
  - Test the boundaries of a quantity field in an online retail application
  - Test variable length data in a text input field
  - Test the order-total functionality of a retail cash register

Keyword-driven test automation

- It is also known as action-word testing
- Center around the concept of **keywords**
    - Keywords are implemented by ***test engineers***.
    - Test scripts are composed by ***domain experts*** using keywords.

Example keywords for a web-questionnaire application:



- A simple keyword (one action on one object), e.g. entering a username into a textfield.

Object	Action	Data
Textfield (username)	Enter text	<username>

- A more complex keyword (a combination of other keywords in a meaningful unit), e.g. logging in.

Object	Action	Data
Textfield (username)	Enter text	<username>
Textfield (password)	Enter text	<password>
Button (login)	Click	One left click

Knowledge Point #9

- Test management
  - Test process
  - Test activities
  - Test plan
  - Test documentation
  - IEEE-829

Exam format

- 20选择题（60分）+2简答（10分）+1大题（30分）
- 总分=作业（20%）+考试（30%）+课程项目（50%）
- 两次作业各10%
- 课程项目1：10%
- 课程项目2：20%
- 课程项目3：20%

大题类型

- Whitebox
  - Draw a control flow chart for the code. List all its basis paths.
  - Design test cases to achieve statement, branch coverage, loop coverage, condition coverage, short-circuiting condition coverage.
  - List defs and uses. Design test cases to cover def-use pairs.
- Blackbox
  - Equivalence class partitioning
  - Boundary value analysis
  - Pairwise coverage with constraints

Thank you!