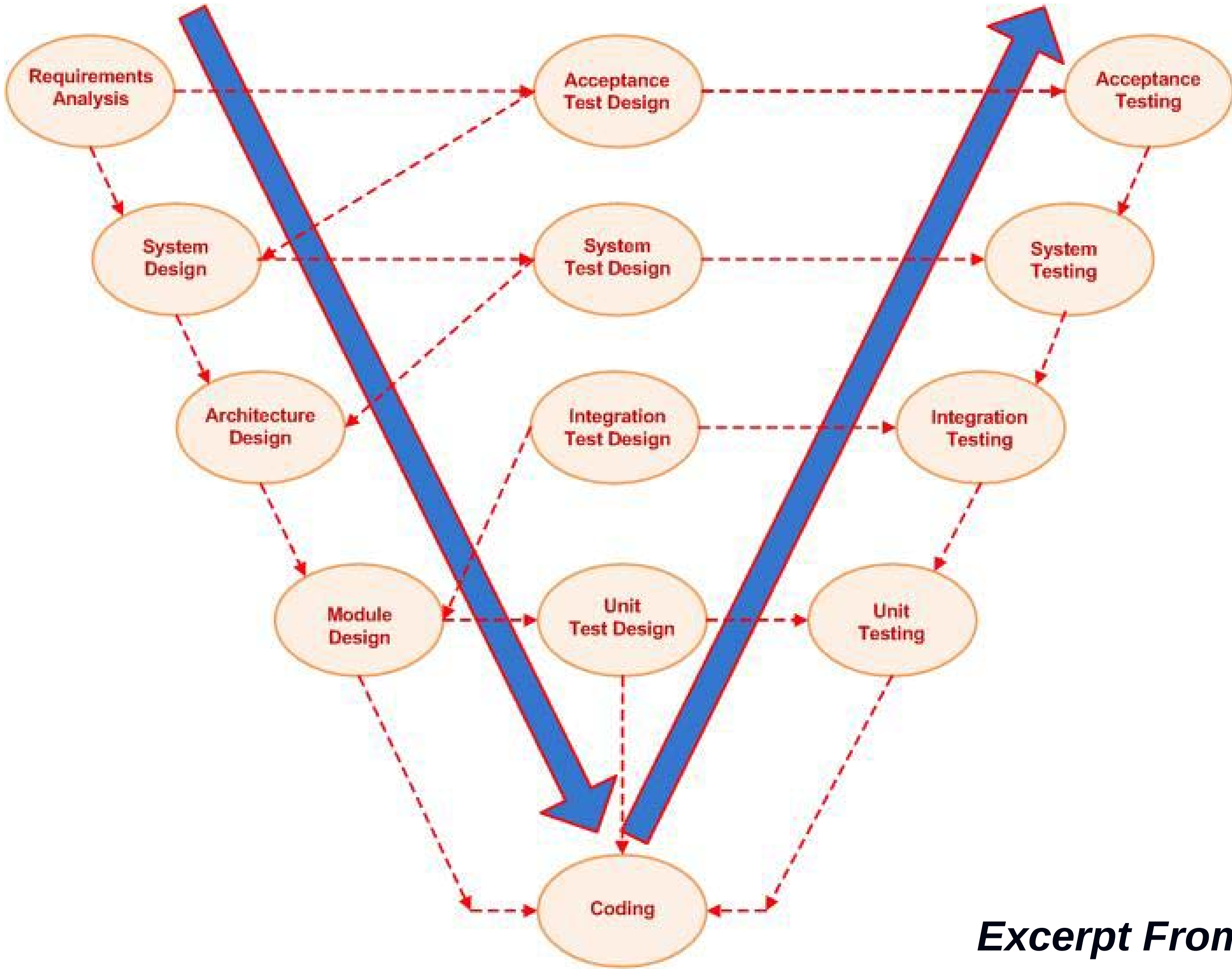
Chapter 14
Testing Tactics

Moonzoo Kim
CS Dept. KAIST

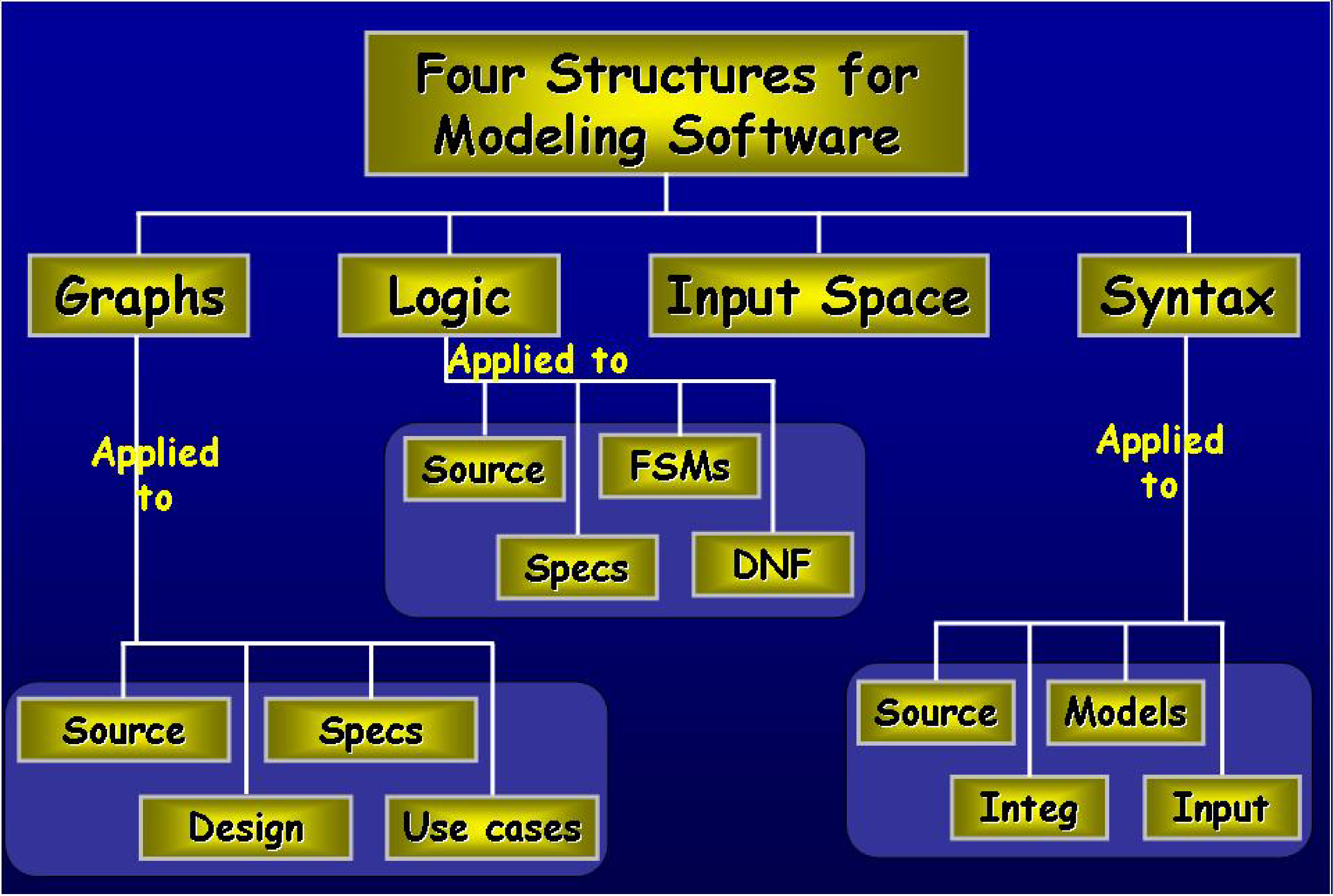
Overview of Ch14. Testing Tactics

- 14.1 Software Testing Fundamentals
- 14.2 Blackbox and White-Box Testing
- 14.3 White-Box Testing
- 14.4 Basis Path Testing
  - Glow Graph Notation
  - Independent Program Paths
  - Deriving Test Cases
  - Graph Matrices
- 14.5 Control Structure Testing
  - Condition Testing
  - Data Flow Testing
  - Loop Testing

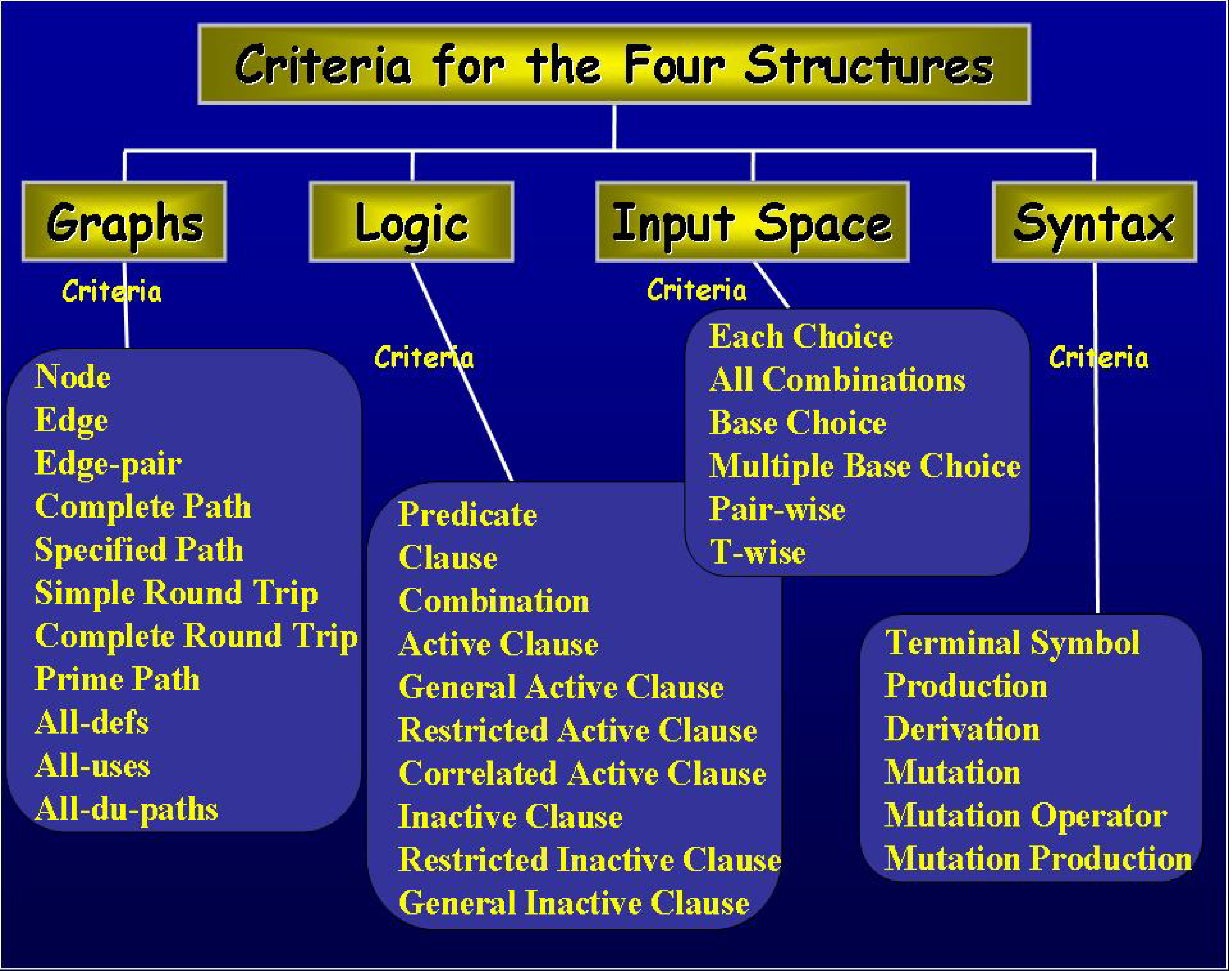
"V" Model



Excerpt From Wikipedia



Quoted from "Intro. To Software Testing" by P.Ammann and J.Offutt



Quoted from "Intro. To Software Testing" by P.Ammann and J.Offutt

Testability

- Operability
  - it operates cleanly
- Observability
  - the results of each test case are readily observed
- Controllability
  - the degree to which testing can be automated and optimized
- Decomposability
  - testing can be targeted
- Simplicity
  - reduce complex architecture and logic to simplify tests
- Stability
  - few changes are requested during testing
- Understandability
  - of the design

- Modular design provides good testability
- Let's think about embedded SW
  - mobile phone software
  - Linux kernel

KAIST

What is a "Good" Test?

- A good test has a high probability of finding an error
- A good test is not redundant.
- A good test should be "best of breed"
- A good test should be neither too simple nor too complex

Designing Unique Tests (pg423)

- **The scene:**
  - Vinod's cubical.
- **The players:**
  - Vinod, Ed

    members of the *SafeHome* software engineering team.
- **The conversation:**
- **Vinod:** So these are the test cases you intend to run for the *password validation* operation.
- **Ed:** Yeah, they should cover pretty much all possibilities for the kinds of passwords a user might enter.
- **Vinod:** So let's see ... you note that the correct password will be 8080, right?
- **Ed:** Uh huh.
- **Vinod:** And you specify passwords 1234 and 6789 to test for errors in recognizing invalid passwords?
- **Ed:** Right, and I also test passwords that are close to the correct password, see ... 8081 and 8180.
- **Vinod:** Those are okay, but I don't see much point in running both the 1234 and 6789 inputs. They're redundant . . . test the same thing, don't they?

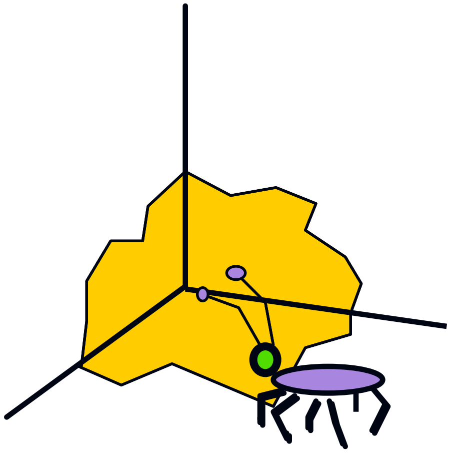
KAIST

- **Ed:** Well, they're different values.
- **Vinod:** That's true, but if 1234 doesn't uncover an error ... in other words ... the *password validation* operation notes that it's an invalid password, it is not likely that 6789 will show us anything new.
- **Ed:** I see what you mean.
- **Vinod:** I'm not trying to be picky here ... it's just that we have limited time to do testing, so it's a good idea to run tests that have a high likelihood of finding new errors.
- **Ed:** Not a problem ... I'll give this a bit more thought.

Test Case Design

"Bugs lurk in corners
and congregate at
boundaries ..."

Boris Beizer



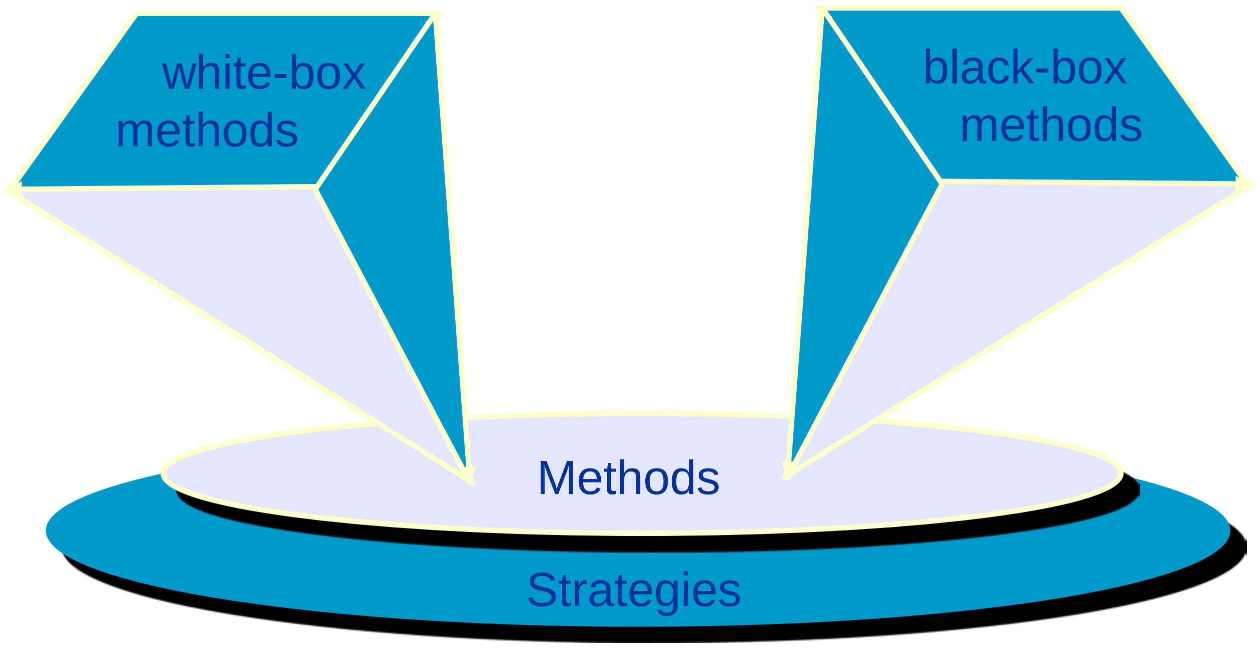
OBJECTIVE to uncover errors

CRITERIA in a complete manner

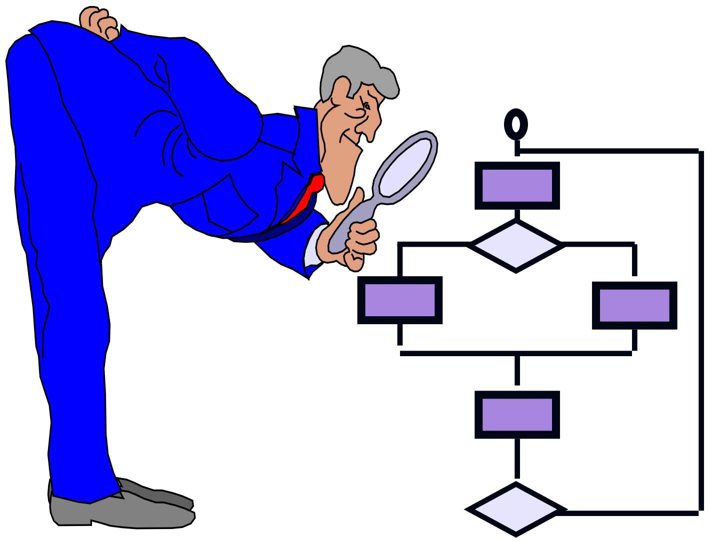
CONSTRAINT with a minimum of effort and time

KAIST

Software Testing



White-Box Testing



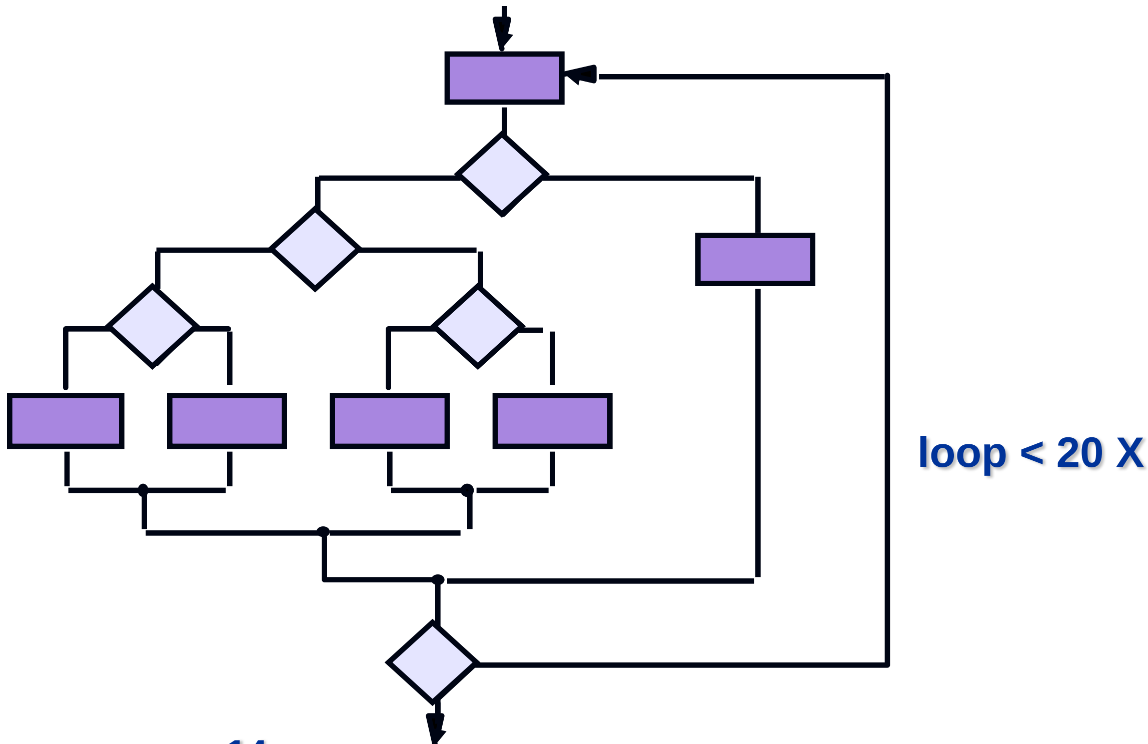
... our goal is to ensure that all statements and conditions have been executed at least once ...

(statement coverage, branch coverage, path coverage, etc)

Why Statement/Branch/Path Coverage?

- logic errors and incorrect assumptions are inversely proportional to a path's execution probability
- we often <u>believe</u> that a path is not likely to be executed; in fact, reality is often counter intuitive
- typographical errors are random; it's likely that untested paths will contain some

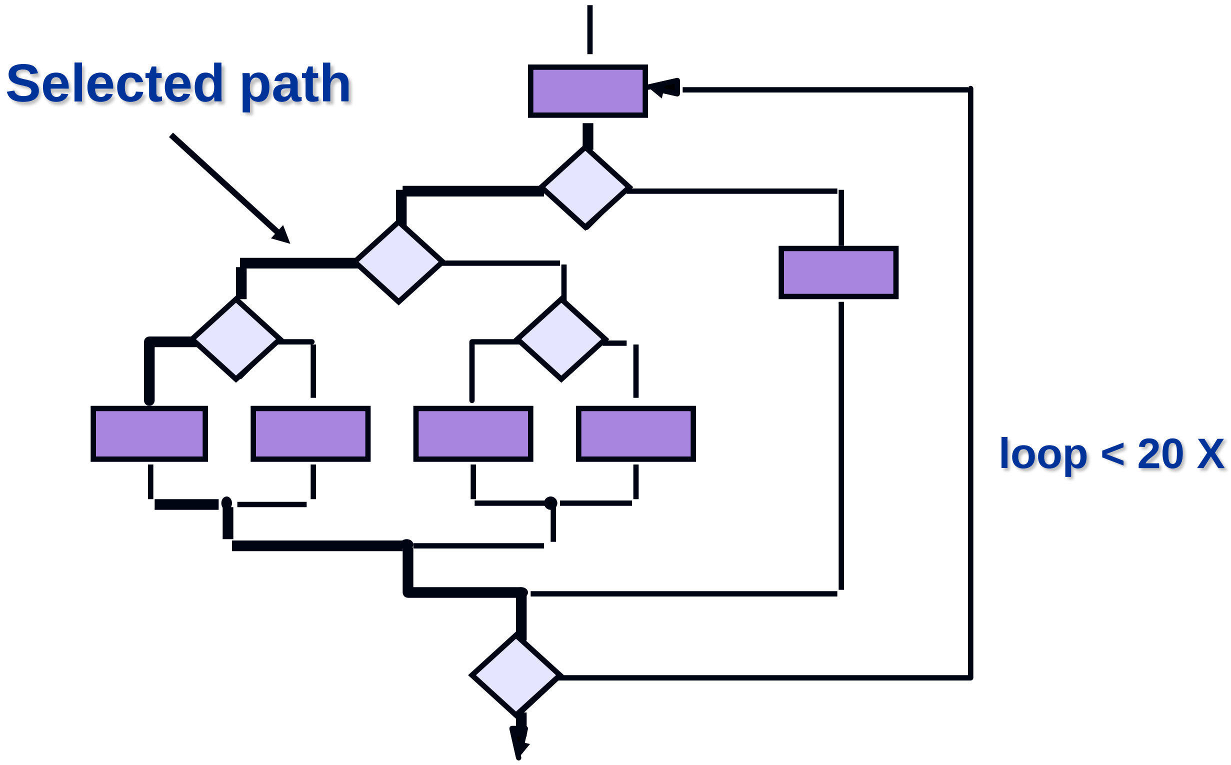
Exhaustive Path Testing



There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

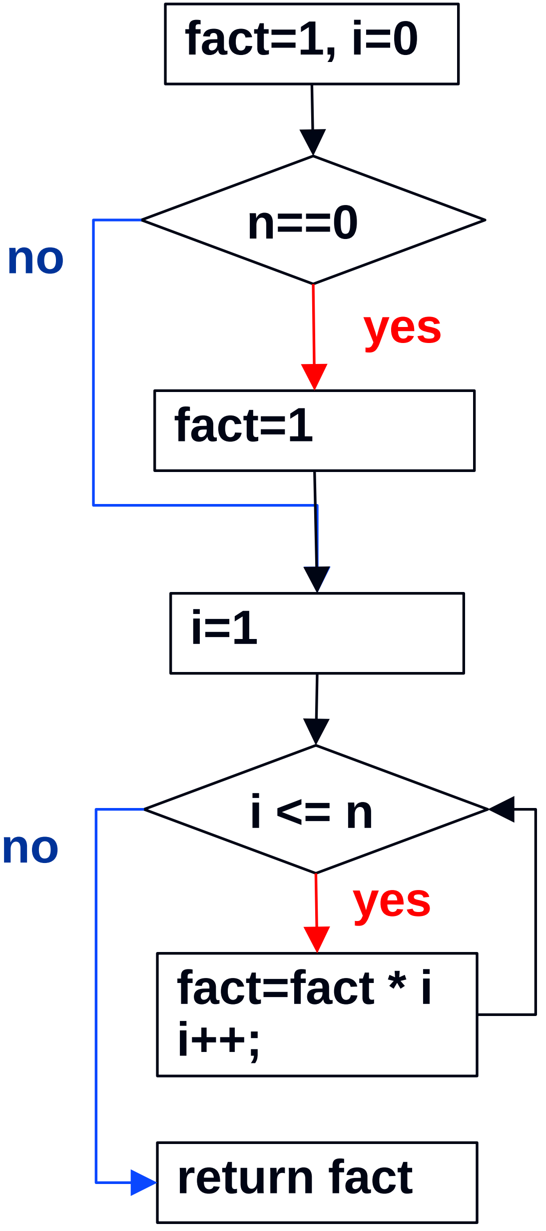
However, model checking techniques can analyze more than 10^{14} test scenarios systematically in a modest time.

Selective Path Testing



Example

```
int factorial( unsigned char n) {
     unsigned char fact=1,i=0;
    if( n == 0) fact=1; // 0!=1
    for(i=1; i <= n; i++)
          fact = fact * i;
    return fact;
}
```



Statement Coverage <= Branch coverage <= Path coverage

Why More than Path Coverage?

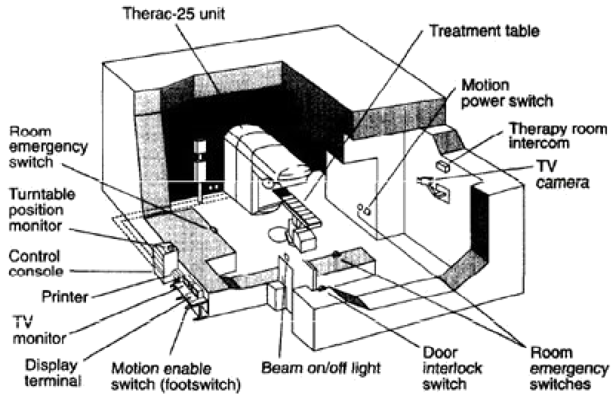
- A flow graph does not reflect a real imperative program
  - A state of a real imperative program consists of values of variables while graph theory considers a node as a simple entity

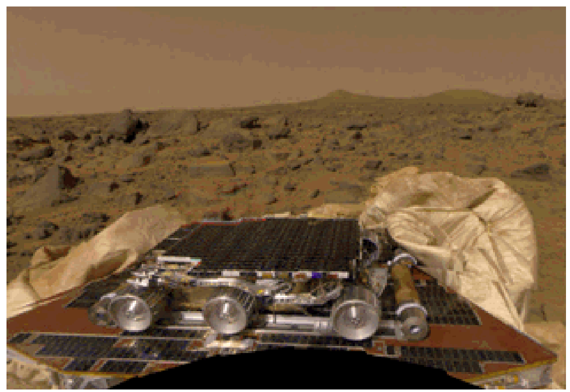
```
// Only one path exists
// Suppose we use a test case of x=0, and y=0
 int adder(int x, int y) {  return 0;}
```

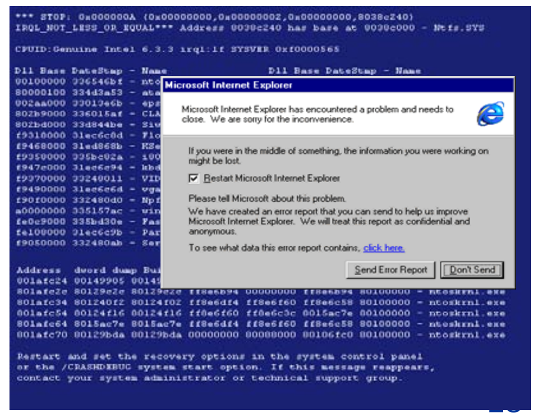
- Most complicated error is caused from loop construct
  - Coverage test does not consider loop
- Therefore, statement/branch/path coverage testing should not be considered as complete test
  - Dijkstra said that testing cannot show the absence of a bug, but a presence of a bug in this sense

Tragic Accidents due to Software Bugs

We need more rigorous and complete analysis methods than testing!!!

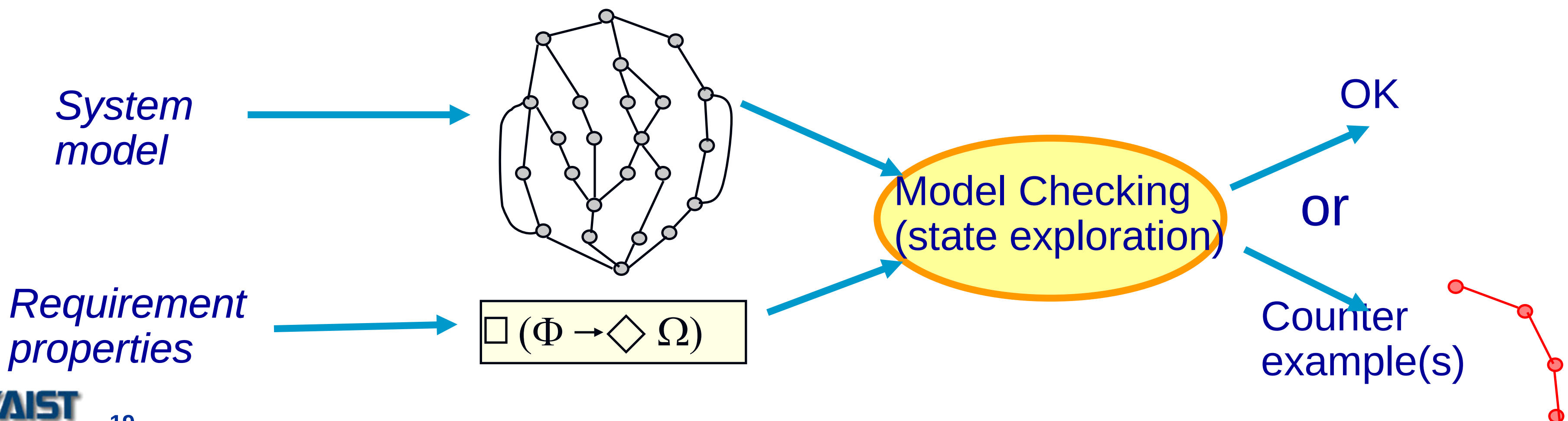






Model Checking Basics

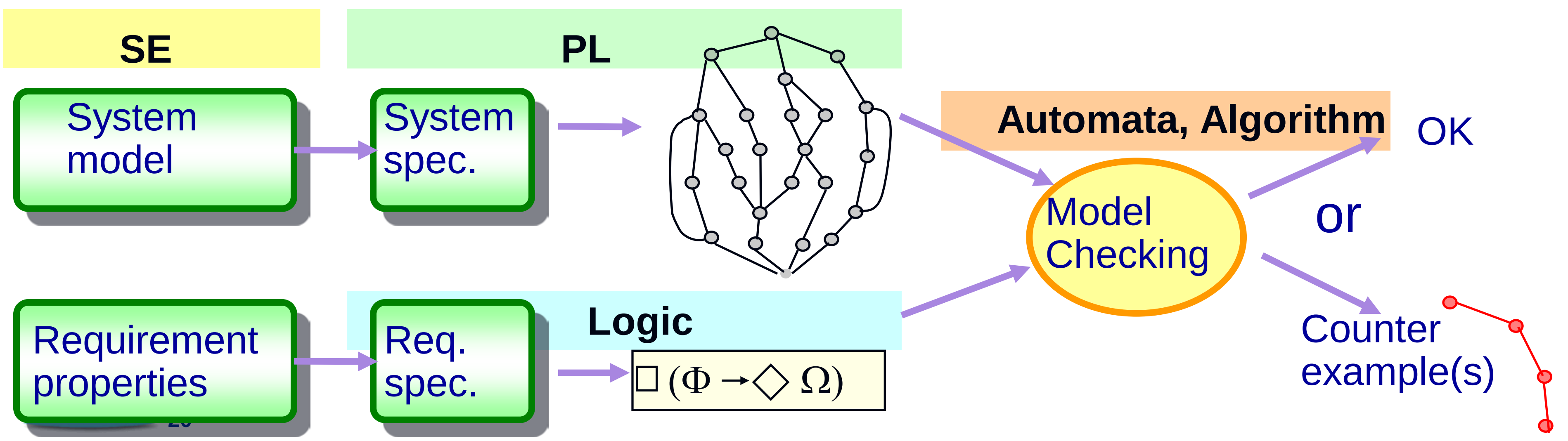
- Specify requirement properties and build a system model
  - Similar to a test oracle and a target software under testing (SUT) in testing
- Generate all possible states (containing values of variables) from the model and then check whether given requirement properties are satisfied within the state space



Model Checking Basics (cont.)

- Undergraduate foundational CS classes contribute this area
  - CS204 Discrete mathematics
  - CS300 Algorithm
  - CS320 Programming language
  - CS322 Automata and formal language
  - CS350 Introduction to software engineering
  - CS402 Introduction to computational logic

Model checking techniques can help analyze more than 10^{1000} test scenarios systematically



An Example of Model Checking ½
(checking every possible values of variables)

System Spec.

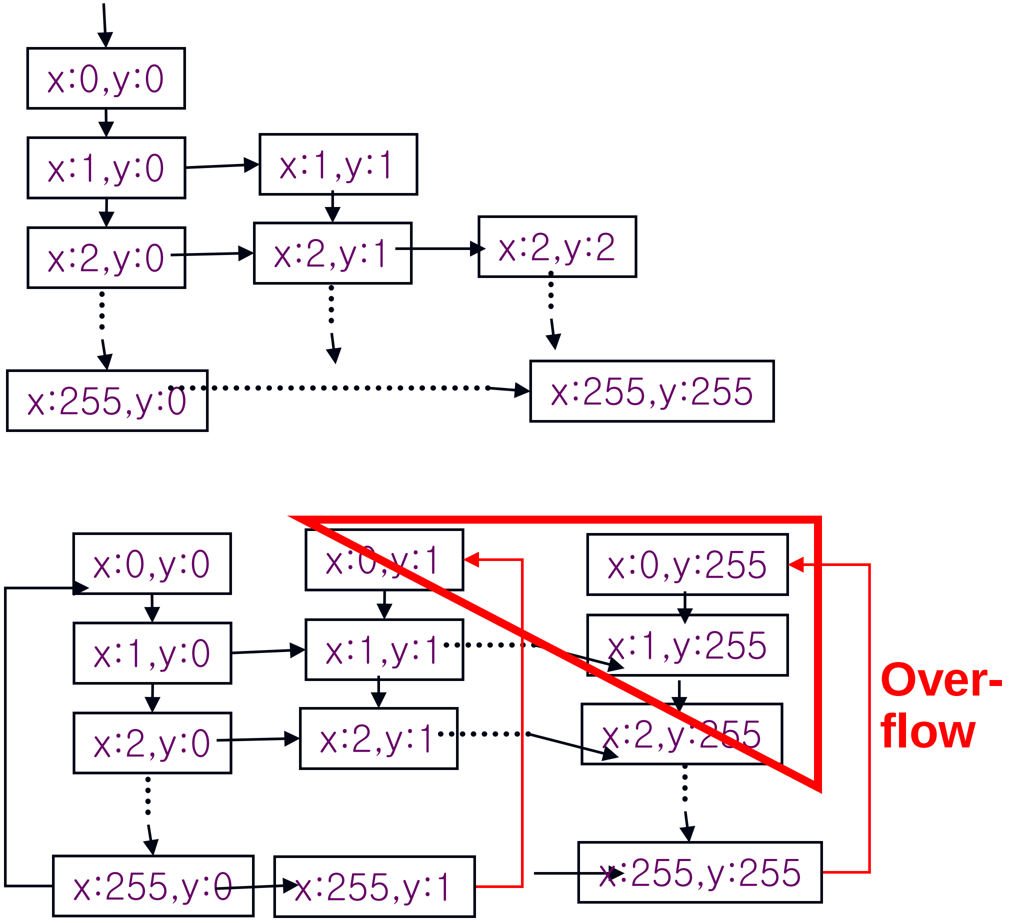
```
unsigned char x=0;
unsigned char y=0;

void proc_A()  {// Thread 1
  while(1)
    x++;
}

void proc_B() {Thread 2
  while(1)
    if (x>y)
      y++;
}
```

Req. Spec

always (x >= y)



An Example of Model Checking 2/2
(*checking **every possible** thread scheduling*)

```
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 | *Process 1*

Process 1:
x = 2
y==0 || y ==2
z = 2
x==2

Process 0:
x = 1
y==0 || y == 1

Process 1:
y=2
(z==2)
cnt++

Process 0:
z = 1
x==1
y = 1
z == 1
cnt++

Violation detected !!!

Counter Example

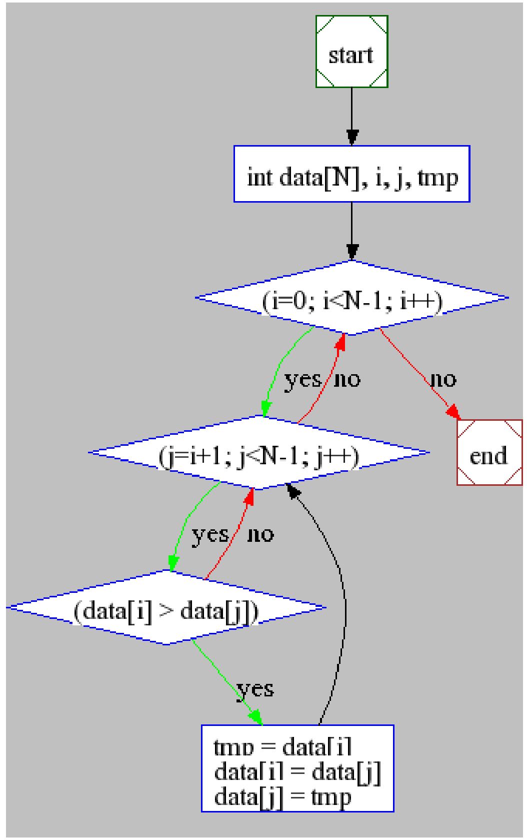
Model Checking History

1981	Clarke / Emerson: CTL Model Checking Sifakis / Quielle	10^5
1982	EMC: Explicit Model Checker Clarke, Emerson, Sistla	
1990	Symbolic Model Checking Burch, Clarke, Dill, McMillan	10^{100}
1992	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	

Model Checking Example: Bubble Sort

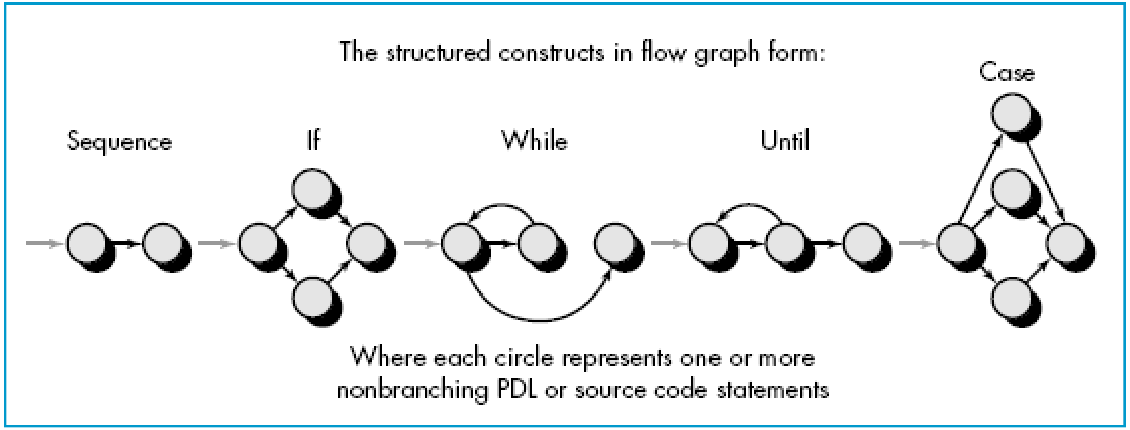
```
#include <stdio.h>
#define N 4
int main(){
    int data[N], i, j, tmp;

    /* It misses the last element,
       i.e., data[N-1]*/
1:      for (i=0; i<N-1; i++) {
2:         for (j=i+1; j<N-1; j++) {
3:              if (data[i] > data[j])  {
4:                   tmp = data[i];
                     data[i] = data[j];
                     data[j] = tmp;
                }
            }
        }
5: /* Check the array is sorted */
}
```

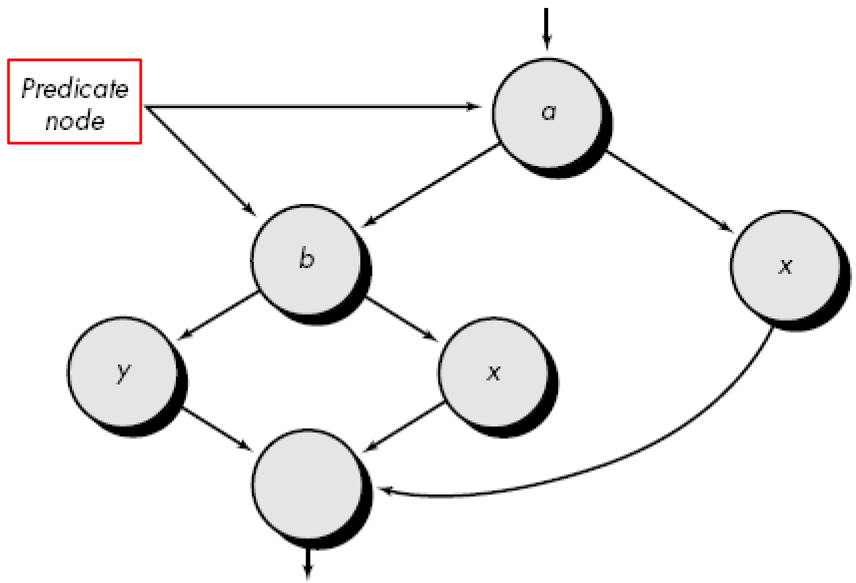


- There exist at most 8 (2x2x2) simple paths
    - However, the following test cases fail to detect the bug
      (0,1,2,3),
      (0,2,1,3),
      (1,0,2,3),
      (1,2,0,3),
      (2,0,1,3),
      (2,1,0,3)
- A number of possible states is $(2^{32})^4 = 3.4\times10^{38}$
    - Suppose that 1 test takes1 microsecond total testing takes 3.4×10^{32} seconds
    - However, SAT based model checking completes the analysis in 2 seconds
-

Basis Path Testing: Flow Graph Notation



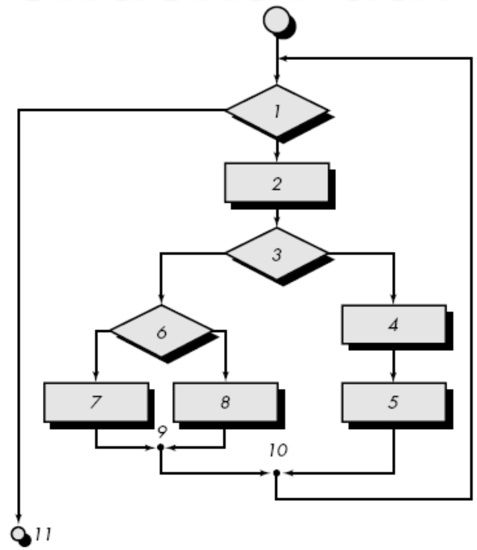

```
.
.
.
IF a OR b
then procedure   x
else procedure   y
ENDIF
```



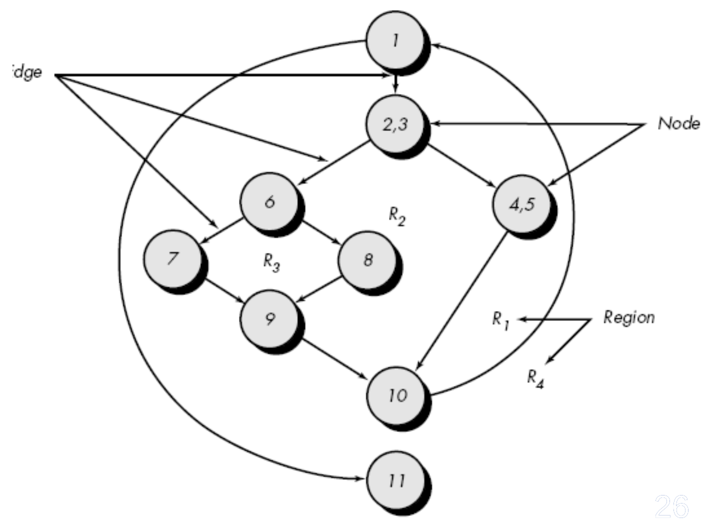
KAIST

Basis Path Testing: an Independent Path

- An independent path is any path through the program that introduces at least one new <u>statement</u> or a new <u>condition</u>.
- Equivalently, an independent path must move along at least one edge that has not been traversed before the path is defined.
- Ex. A set of independent paths
  - Path 1: 1-11
  - Path 2: 1-(2,3)-(4,5)-10-1-11
  - Path 3: 1-(2,3)-6-8-9-10-1-11
  - Path 4: 1-(2,3)-6-7-9-10-1-11
- But the following path is not
  - 1-(2,3)-(4,5)-10-1-2-3-6-8-9-10-1-11
- Paths 1,2,3, and 4 constitute a basis set
  - If tests can be designed to exercise a basis set, the followings can be guaranteed.
    - Every statement will be executed at least once
    - Every condition will be executed on its true and false sides

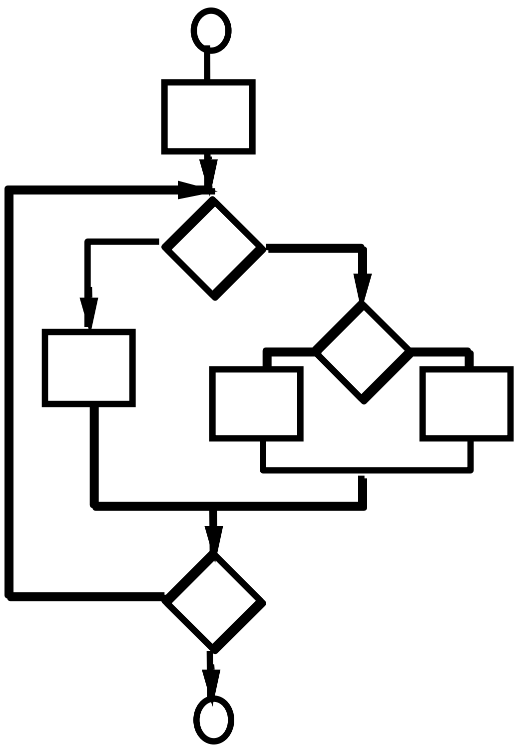


(A)



(B)

Basis Path Testing: How Many Paths?



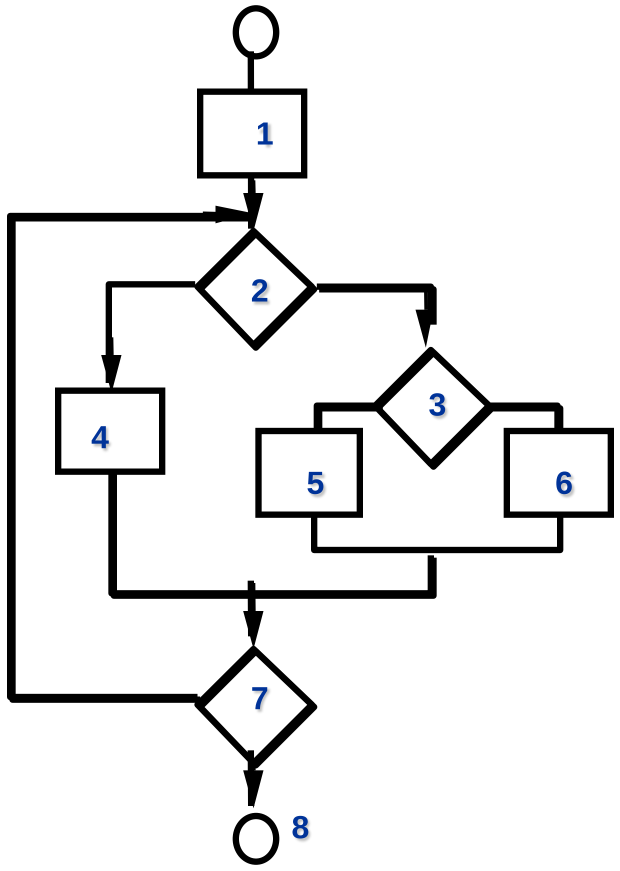
- First, we compute the cyclomatic complexity, which is a quantitative measure of the logical complexity

- Cyclomatic complexity defines the # of independent paths to test for complete statement/branch coverage
  - number of simple decisions + 1
  - number of edge – number of node +2
  - number of enclosed areas + 1
  - In this case, V(G) = 4

V(G) is the upper bound for the # of independent paths for complete coverage

KAIST

Basis Path Testing



Next, we derive the independent paths:
(paths containing a new edge)

Since V(G) = 4,
there are four paths

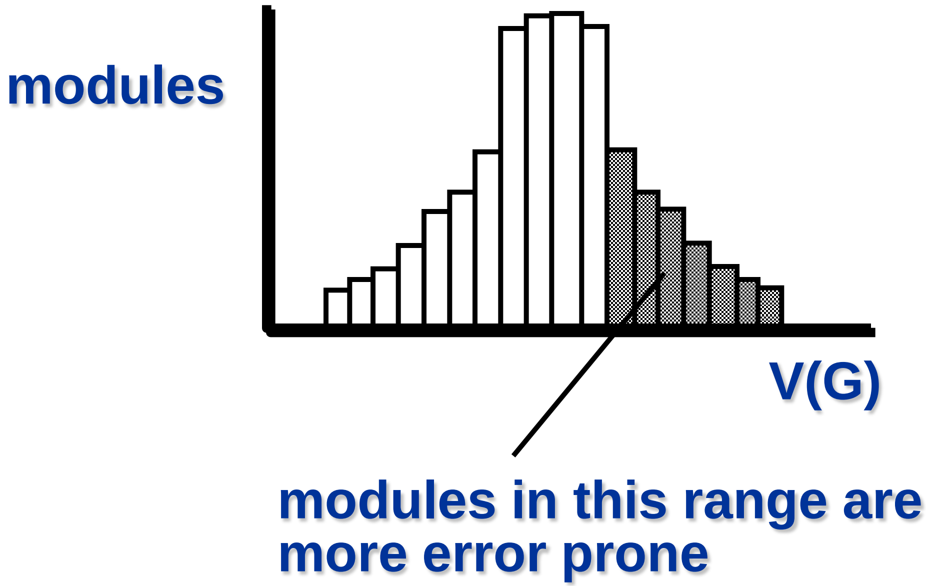
Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8
Path 3: 1,2,4,7,8
Path 4: 1,2,4,7,2,4,7,8

Finally, we should derive test cases to exercise these paths

KAIST

Cyclomatic Complexity

A number of industry studies have indicated that the higher V(G), the higher the probability or errors.



Using Cyclomatic Complexity (pg428)

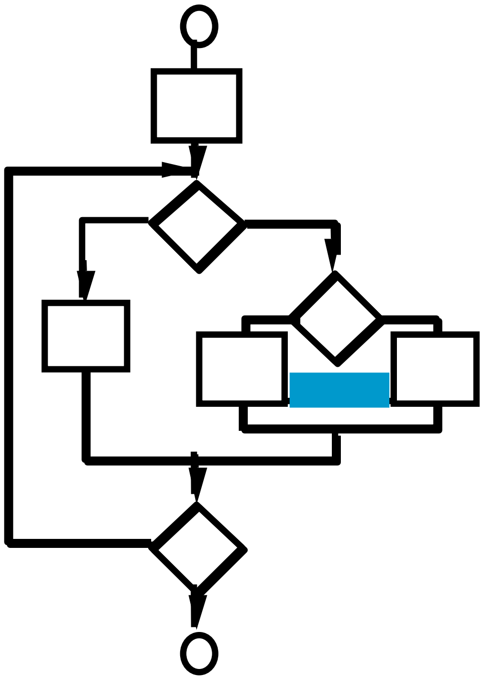
- **The scene:**
  - Shakira's cubicle.
- **The players:**
  - Vinod,Shakira

    members of the *SafeHome* software engineering team who are working on test planning for the security function.
- **The conversation:**
- **Shakira:** Look ... I know that we should unit test al! the components for the security function, but there are a lot of 'em and if you consider the number of operations that have to be exercised, I don't know ... maybe we should forget white-box testing, integrate everything, and start running black-box tests.
- **Vinod:** You figure we don't have enough time to do component tests, exercise the operations, and then integrate?
- **Shakira:** The deadline for the first increment is getting closer than I'd like ... yeah, I'm concerned.
- **Vinod:** Why don't you at least run white-box tests on the operations that are likely to be the most error prone?

KAIST

- **Shakira (exasperated):** And exactly how do I know which are likely to be the most error prone?
- **Vinod:** V of *G.*
- **Shakira:** Huh?
- **Vinod:** Cyclomatic complexity--V of *G*. Just compute *V(G)* for each of the operations within each of the components and see which have the highest values for V(G). They're the ones that are most likely to be error prone.
- **Shakira:** And how do I compute *V* of *G?*
- **Vinod:** It's really easy. Here's a book that describes how to do it.
- **Shakira (leafing through the pages):** Okay, it doesn't look hard. I'll give it a try. The ops with the highest *V(G)* will be the candidates for white-box tests.
- **Vinod:** Just remember that there are no guarantees. A component with a low *V(G)* can still be error prone.
- **Shakira:** Alright. But at least this'll help me to narrow down the number of components that have to undergo white-box testing.

Basis Path Testing Notes



- you don't need a flow chart, but the picture will help when you trace program paths
- count each simple logical test, compound tests count as 2 or more
- basis path testing should be applied to critical modules

Graph Matrices

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing

Control Structure Testing

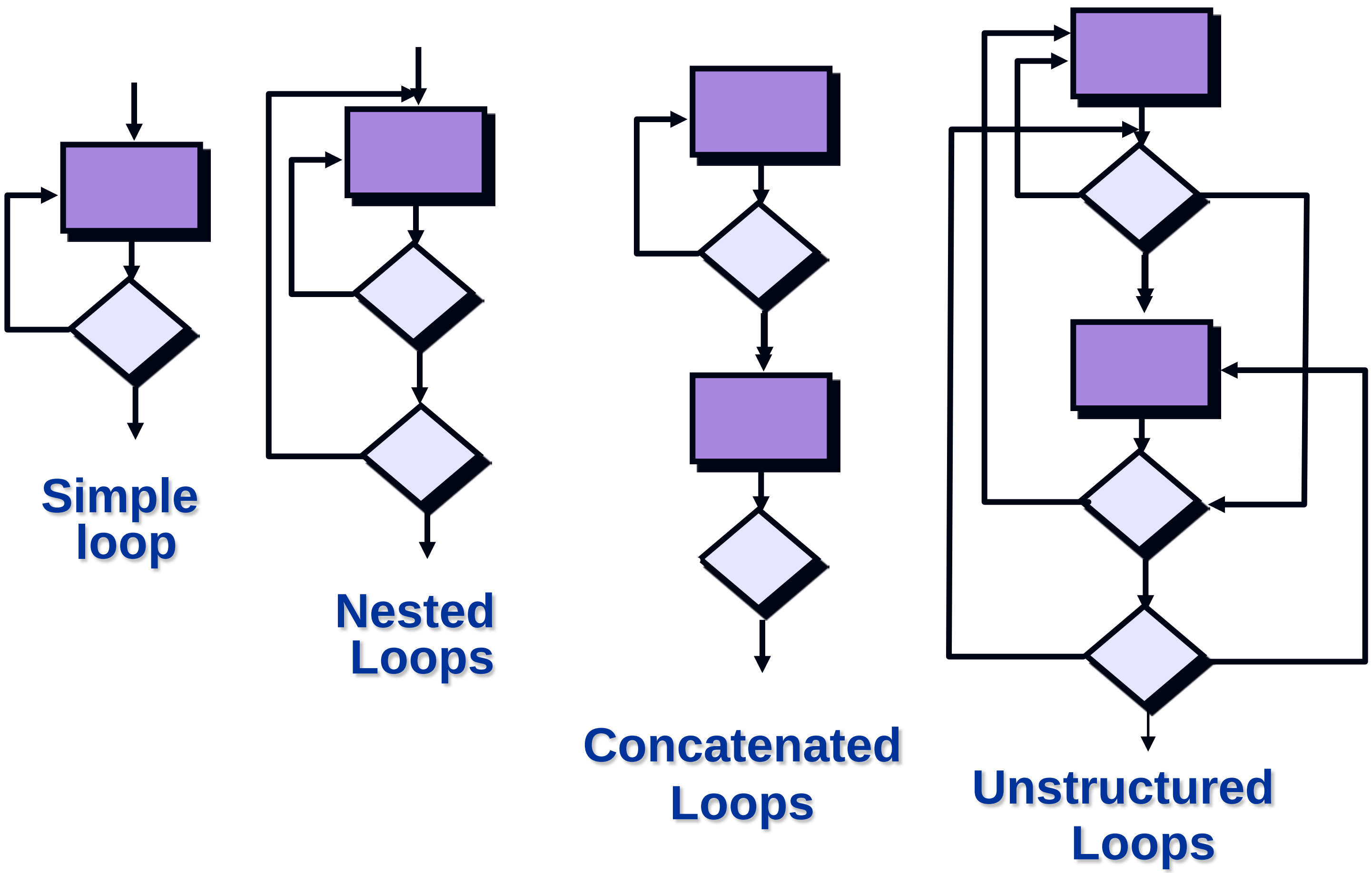
- Condition testing
  - a test case design method that exercises the logical conditions contained in a program module
- Data flow testing
  - selects test paths of a program according to the locations of definitions and uses of variables in the program

Data Flow Testing

- For a statement S
  - DEF(S) = {X| statement S contains a definition of X}
  - USE(S) = {X| statement S contains a use of X}
- A definition-use (DU) chain of variable X is of the form [X,S,S'] where S and S' are statement, X is in DEF(S) and USE(S')
  - [x,s1,s3] is a DU chain
  - [y,s1,s3] is NOT a DU chain
- A branch is not guaranteed to be covered by DU testing

```
void f() {
s1:   int x = 10, y;
s2:   if ( …) {
          …
s3:       y = x + 1;
      }
```

Loop Testing



Loop Testing: Simple Loops

Minimum conditions—Simple Loops

1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop m < n
5. (n-1), n, and (n+1) passes through the loop

where n is the maximum number of allowable passes

KAIST

Loop Testing: Nested Loops

Nested Loops

Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.

Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.

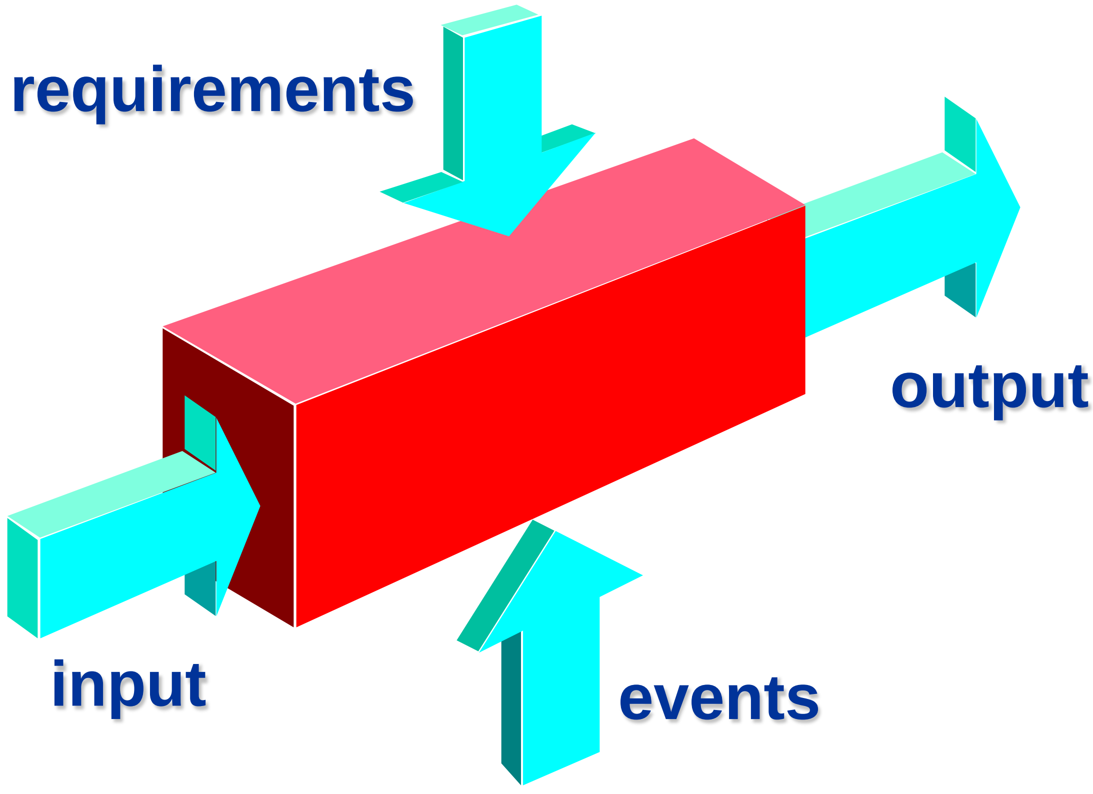
Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

Concatenated Loops

If the loops are independent of one another
then treat each as a simple loop
else* treat as nested loops
endif*

for example, the final loop counter value of loop 1 is used to initialize loop 2.

Black-Box Testing



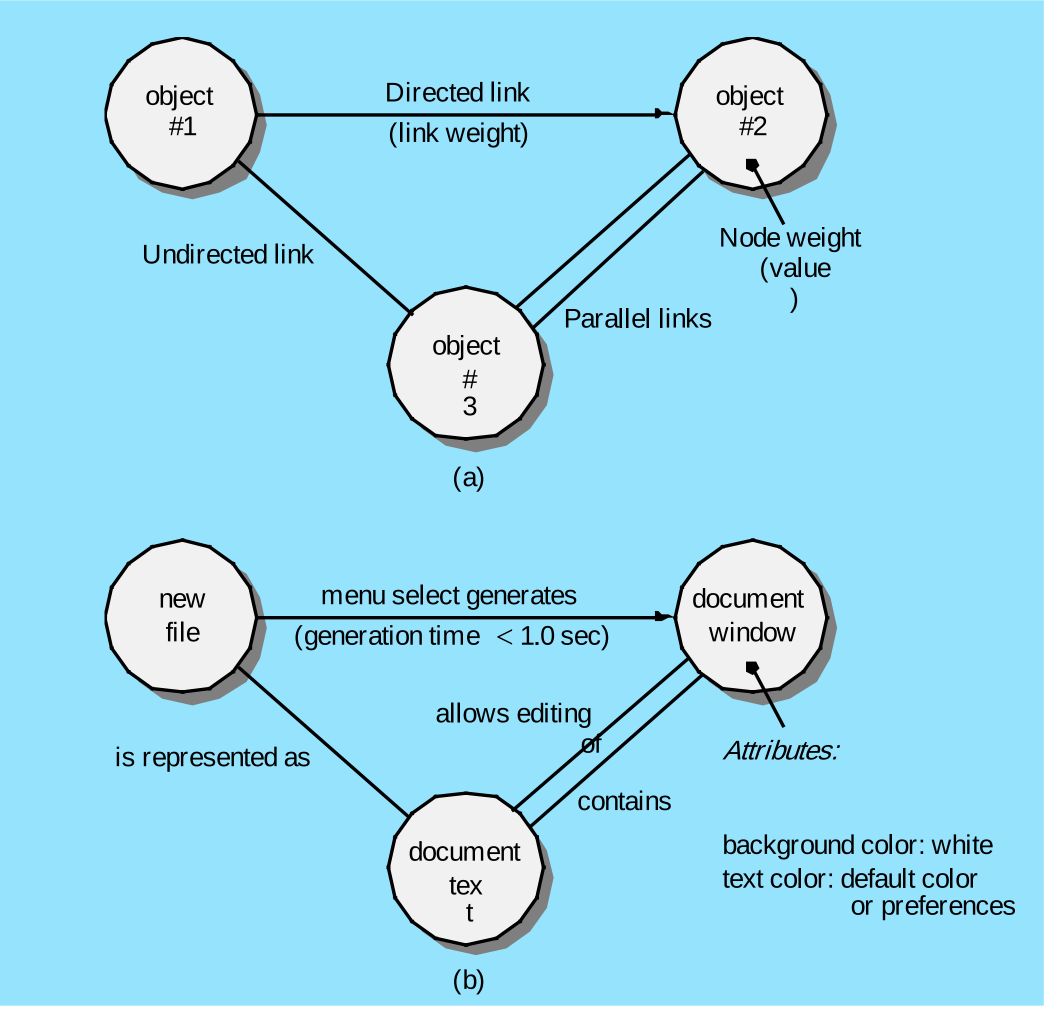
Black-Box Testing

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

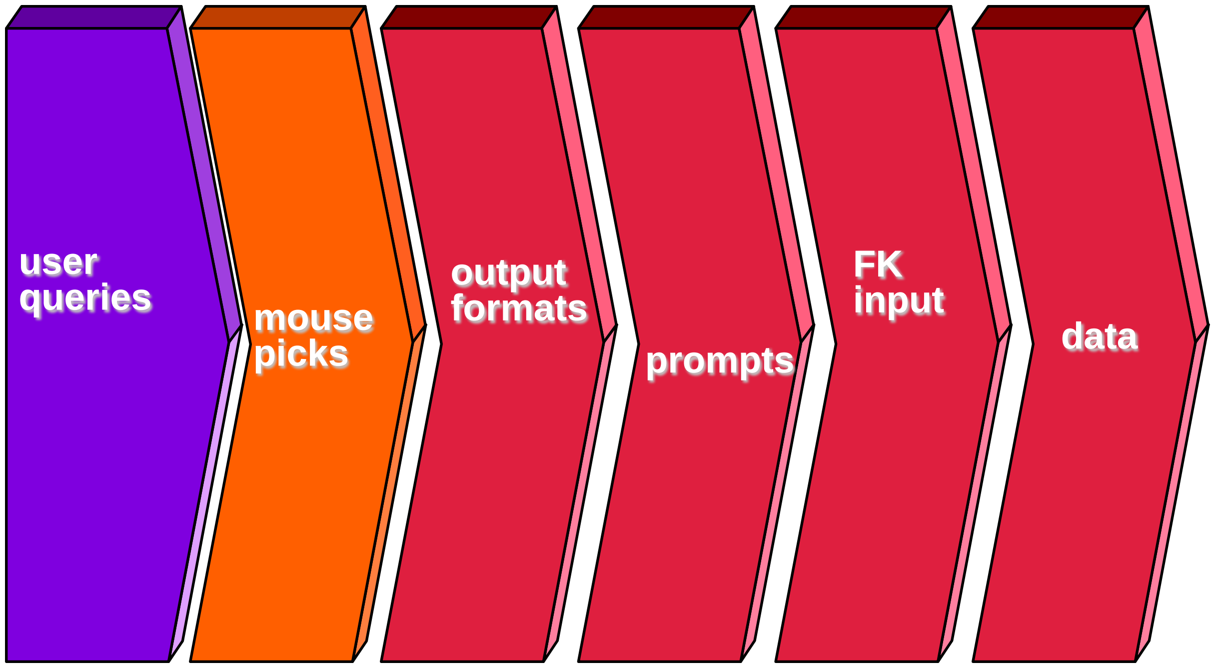
Graph-Based Methods

To understand the objects that are modeled in software and the relationships that connect these objects

In this context, we consider the term "objects" in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.



Equivalence Partitioning



Sample Equivalence Classes

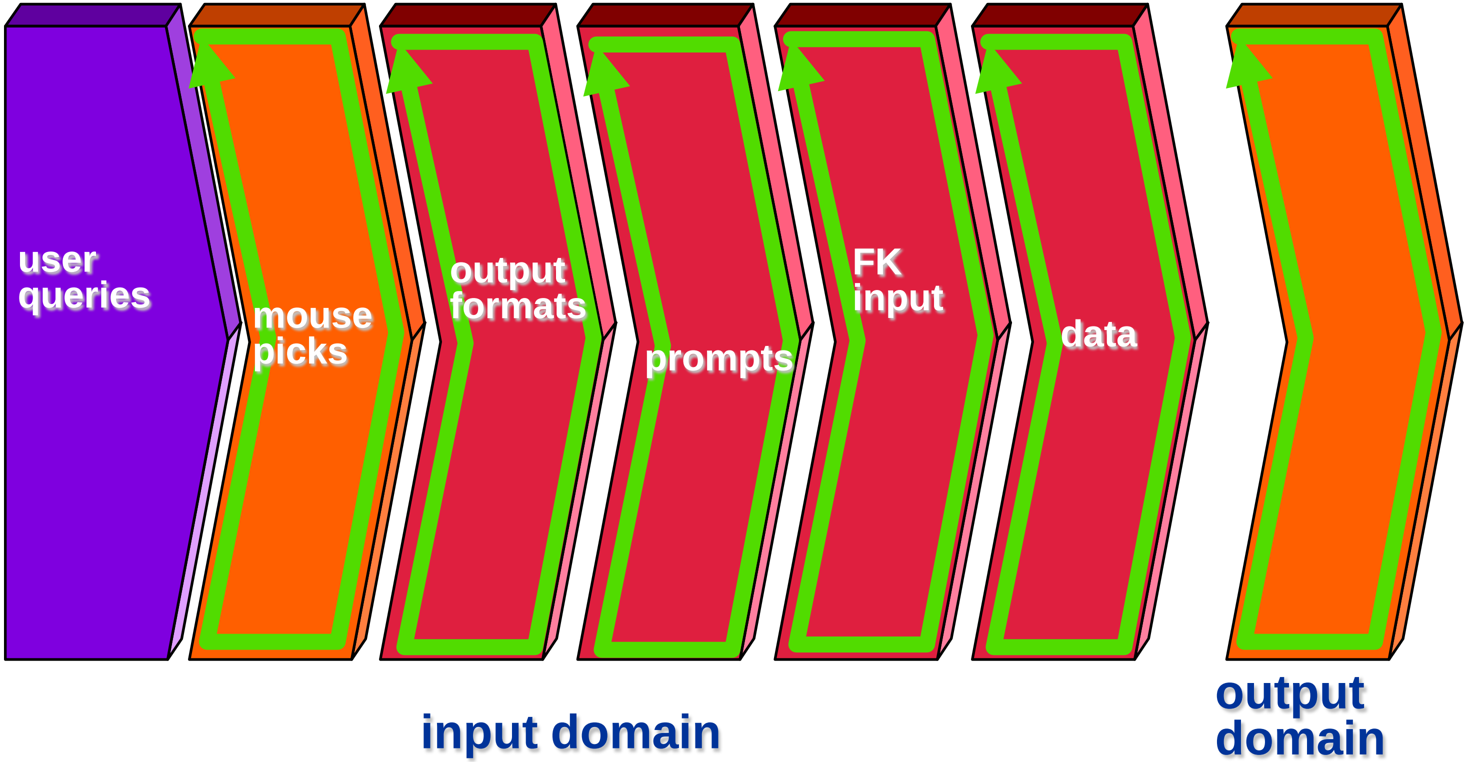
Valid data

- user supplied commands
- responses to system prompts
- file names
- computational data
  - physical parameters
  - bounding values
  - initiation values
- output data formatting
- responses to error messages
- graphical data (e.g., mouse picks)

Invalid data

- data outside bounds of the program
- physically impossible data
- proper value supplied in wrong place

Boundary Value Analysis

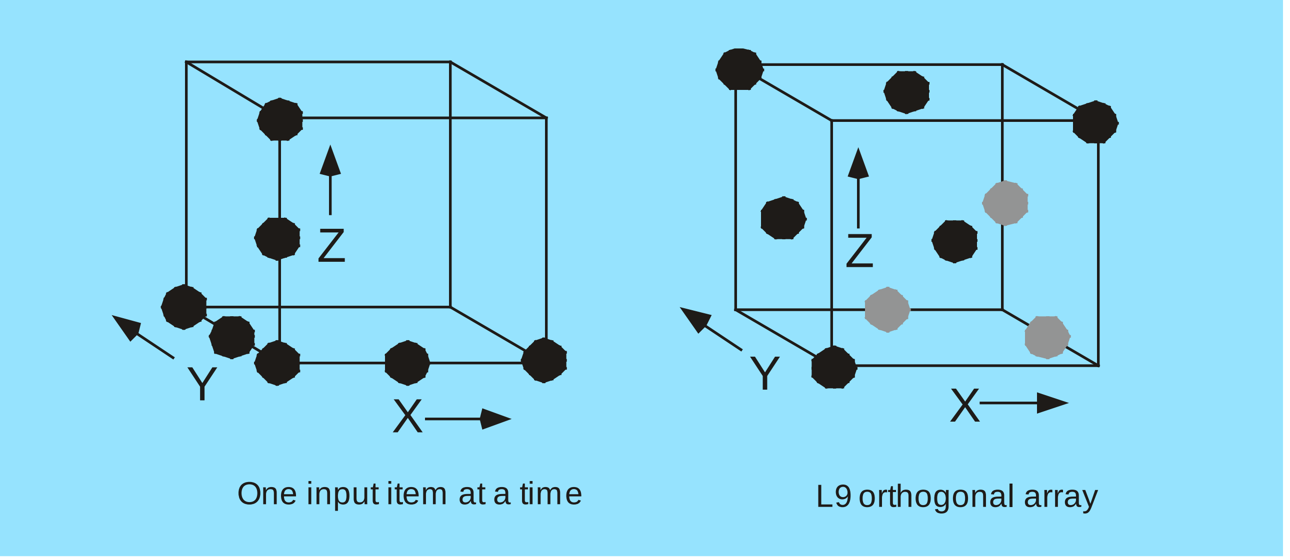


Comparison Testing

- Used only in situations in which the reliability of software is absolutely critical (e.g., human-rated systems)
  - Separate software engineering teams develop independent versions of an application using the same specification
  - Each version can be tested with the same test data to ensure that all provide identical output
  - Then all versions are executed in parallel with real-time comparison of results to ensure consistency

Orthogonal Array Testing

- Used when the number of input parameters is small and the values that each of the parameters may take are clearly bounded



Testing Methods

- Fault-based testing
  - The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.
- Class Testing and the Class Hierarchy
  - Inheritance does not obviate the need for thorough testing of all derived classes. In fact, it can actually complicate the testing process.
- Scenario-Based Test Design
  - Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests.

OOT Methods: Random Testing

- Random testing
  - identify operations applicable to a class
  - define constraints on their use
  - identify a miminum test sequence
    - an operation sequence that defines the minimum life history of the class (object)
  - generate a variety of random (but valid) test sequences
    - exercise other (more complex) class instance life histories

OOT Methods: Partition Testing

- Partition Testing
  - reduces the number of test cases required to test a class in much the same way as equivalence partitioning for conventional software
  - state-based partitioning
    - categorize and test operations based on their ability to change the state of a class
  - attribute-based partitioning
    - categorize and test operations based on the attributes that they use
  - category-based partitioning
    - categorize and test operations based on the generic function each performs

OOT Methods: Inter-Class Testing

- Inter-class testing
  - For each client class, use the list of class operators to generate a series of random test sequences. The operators will send messages to other server classes.
  - For each message that is generated, determine the collaborator class and the corresponding operator in the server object.
  - For each operator in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
  - For each of the messages, determine the next level of operators that are invoked and incorporate these into the test sequence

OOT Methods: Behavior Testing

The tests to be designed should achieve all state coverage [KIR94]. That is, the operation sequences should cause the Account class to make transition through all allowable states

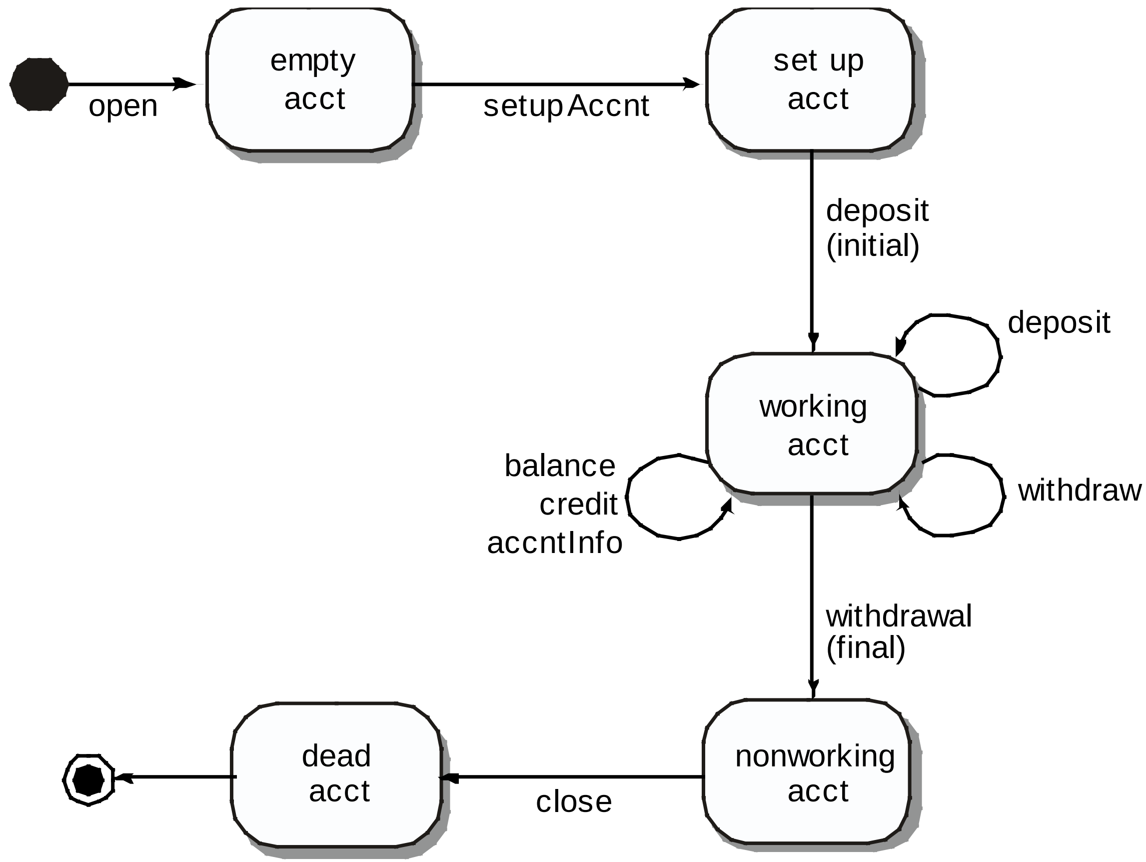


Figure 14.3 State diagram for Account class (adapted from [KIR94])

Testing Patterns

Pattern name: pair testing

Abstract: **A process-oriented pattern, pair testing describes a technique that is analogous to pair programming (Chapter 4) in which two testers work together to design and execute a series of tests that can be applied to unit, integration or validation testing activities.**

Pattern name: separate test interface

Abstract: **There is a need to test every class in an object-oriented system, including "internal classes" (i.e., classes that do not expose any interface outside of the component that used them). The separate test interface pattern describes how to create "a test interface that can be used to describe specific tests on classes that are visible only internally to a component." [LAN01]**

Pattern name: scenario testing

Abstract: **Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The scenario testing pattern describes a technique for exercising the software from the user's point of view. A failure at this level indicates that the software has failed to meet a user visible requirement. [KAN01]**