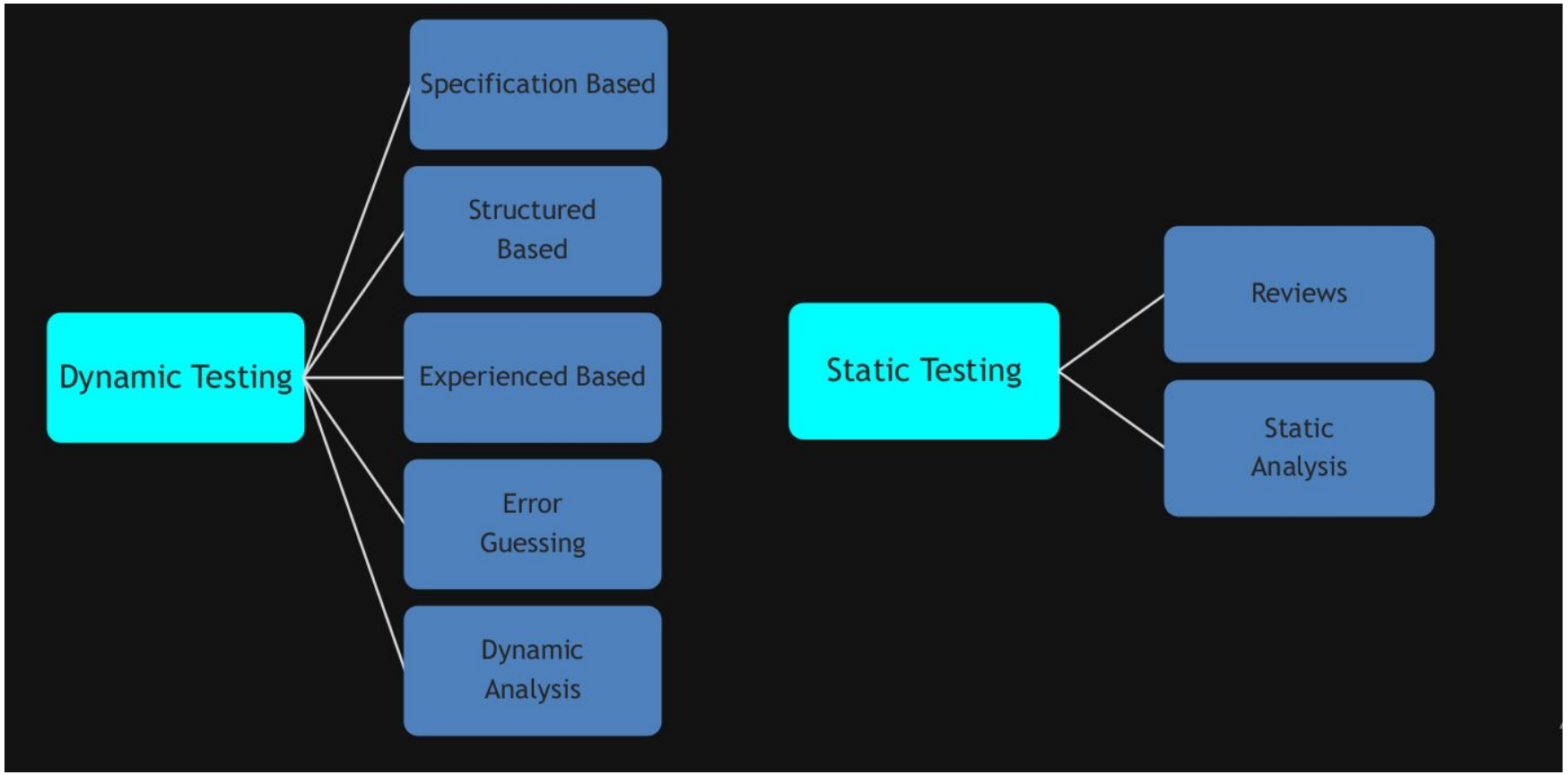
SOFT 3416
Software Verification and Validation
Week IV

Ahmet Feyzi Ateş

Işık University Faculty of Engineering and Natural Sciences
Department of Computer Science Engineering
2022-2023 Spring Semester

Static and Dynamic Test Techniques – An Overview



Categories of Dynamic Testing Techniques

White-box (structural) Test Techniques:

- Based on control flow:
  - Statement coverage
  - Condition coverage
  - Decision coverage
  - Branch coverage
  - Condition/Decision coverage
  - Modified coverage
  - Condition/Decision coverage
- Based on data flow:
  - All du-paths strategy
  - All uses strategy
  - All predicate uses strategy
  - All computational uses strategy
  - All p-uses/some c-uses strategy
  - All c-uses/some p-uses strategy
  - All definitions strategy

Categories of Dynamic Testing Techniques

Black-box (specification-based) Test Techniques:

- Equivalence Partitioning
- Boundary Value Analysis
- Decision Table Testing
- State Transition Testing
- Use Case Testing

Experience-based Test Techniques:

- Error Guessing
- Exploratory Testing
- Checklist-based Testing

Code Structure of Software

- Structure based methods are also called ***white-box*** testing techniques.
- There are many different kinds of structural measures for software;
  - each of which tells us something about the effort required to write the code in the first place,
  - to understand the code when making a change, or
  - to test the code using particular tools or techniques.

- There are several aspects of code structure to consider:
  - control flow structure;
  - data flow structure;
  - data structure.

Static Data Flow Analysis

- Data flow analysis in software is a technique used to analyze how data moves through a program.
- It involves examining how data is defined, used, and modified as it flows through different parts of the program's code.
- This analysis helps developers understand how data is processed and propagated within the software, enabling them to identify potential issues, optimize performance, and improve code quality.
- Data flow analysis can be performed statically or dynamically.

Static Data Flow Analysis (cont'd)

- In static analysis, the code is analyzed without actually executing it.
- Static data flow analysis examines the source code or intermediate representations of the program to infer how data flows through the program's variables, functions, and modules.
- Static analysis can help detect issues such as uninitialized variables, unreachable code, unused variables, and potential security vulnerabilities.

Data Flow Testing

- A family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects.

- For example, pick enough paths to assure that:
  - every data object has been initialized prior to use, or that
  - all defined objects have been used for something.

- Based on the idea that:
  - one should not feel confident about a program without having seen the effect of using the value produced by *each and every* computation.
  - just like one would not feel confident about a program without executing *every* statement in it as part of some test (path testing).

Data Flow Testing (cont'd)

- Data flow testing is a powerful tool to detect improper use of data values due to coding errors.
- Data Flow Testing is used to find the following issues:
  - To find a variable that is used but never defined,
  - To find a variable that is defined but never used,
  - To find a variable that is defined multiple times before it is use,
  - Deallocating a variable before it is used.
- Data Flow Testing utilizes what is known as the ***Data Flow Graph,*** a directed graph that it shows the processing flow through a module of code.
- In addition, a ***Data Flow Graph*** details the definition, use and destruction of each module's variables.

Data Flow Testing (cont'd)

- **Data Flow Testing** is a type of structural (white-box) testing.
- It is a method that is used to find the test paths of a program according to the locations of definitions and uses of variables in the program.

- It is concerned with:
  - Statements where variables receive values,
  - Statements where these values are used or referenced.

- Data Flow Testing uses the data flow graph to find the situations that can interrupt the flow of the program.
  - When something has gone wrong with the software so that data objects are not available when they should be, or silly things are being done to data objects.

Data Flow Testing (cont'd)

- Variables that contain data values have a defined life cycle.
  - They are defined (created), used and killed.
- Variables can be used in:
  - computations *(e.g., a=b+1) (computation use),* or
  - in conditionals (e.g., *if (a>5)) (predicate use)*.
- In both usages, it is important that the variable has been assigned a value before it is used.

Example

```
1. read x, y;
2. if (x>y) then
3.   a = x+1
   else
4.   a = y-1
5. print a;
```

✧ **Define/use of variables of above example:**

Variable	Defined at node	Used at node
x	1	2, 3
y	1	2, 4
a	3, 4	5

Definitions

- A node in a program's control flow graph is a ***defining node*** *of some variable v*, if and only if the value of the variable *v* is defined at the statement fragment corresponding to that node.

- Input statements, assignment statements, loop control statements, and procedure calls are all examples of statements that can be defining nodes.

- When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables are changed.

Defining Nodes

- An object is defined *explicitly* when it appears in a data declaration.
- Or *implicitly* when it appears on the left hand side of the assignment.
- A dynamically allocated object has been allocated.
- Something is pushed onto the stack.
- A record written to a file or database.

Definitions (cont'd)

- A node in a program's control flow graph is a ***usage node*** *of the variable v*, if and only if the value of the variable *v* is used at the statement fragment corresponding to that node.

- Output statements, assignment statements, conditional statements, loop control statements, and procedure calls are all examples of statements that are usage nodes.

- When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables remain unchanged.

Definitions (cont'd)

- A usage node is a:
  - ***predicate use*** if and only if the statement corresponding to the node is a predicate (conditional) statement;
  - otherwise, it is a ***computation use***.

- A variable is used for:
  - *computation (c)* when it appears on the right hand side of an assignment statement.
  - It is used in a *Predicate (p)* when it appears directly in a predicate.

Definitions (cont'd)

- A node in a program's control flow graph is a is a ***kill node*** for a variable *v* if the variable is deallocated at the statement fragment corresponding to that node.

- An object is killed or undefined when;
  - it is released or otherwise made unavailable (by explicitly deleting the variable in languages without garbage collection).
  - When the code corresponding to such statements executes, there are no longer any memory location(s) associated with the variables.

Kill Nodes

- An object is killed or undefined when;
  - When its contents are no longer known with absolute certainty/perfectness.
  - Release of dynamically allocated objects back to the availability pool.
  - The old top of the stack after it is popped.
  - An assignment statement can kill and redefine immediately. For example;
    - if *A* had been previously defined and a new assignment has been made such as ***A = 17***; we have killed A's previous value and redefined A.

Definitions (cont'd)

- A definition-use path with respect to a variable *v* (denoted as ***du-path***) is a path such that, there are define and usage nodes in that path such that one *define node is the initial node*, and one *usage node is the final node* of the path.

- A definition-clear path with respect to a variable *v* (denoted ***dc-path***) is a definition-use path with some initial and final nodes, such that *no other node* in the path other than the initial node is a defining node of *v*.

Data Flow Testing (the Technique)

- The above given definitions capture the essence of computing with stored data values.
- ***du-paths*** and ***dc-paths*** describe the flow of data across source statements from points at which the values are defined to points at which the values are used.
- ***du-paths*** that are not definition-clear, i.e., that are not ***dc-paths***, are potential trouble spots.

Data Flow Anomalies

Time-sequenced Pairs of (d), (u), and (k)

dd Defined and defined again *(not invalid but suspicious, could be a programming error)*

du Defined and used *(correct)*

dk Defined and killed *(not invalid but suspicious, could be a programming error)*

ud Used and defined *(acceptable)*

uu Used and used again *(acceptable)*

uk Used and killed *(acceptable)*

kd Killed and defined *(acceptable)*

ku Killed and used *(a serious defect)*

kk Killed and killed *(could be a programming error)*

Data Flow Anomalies (cont'd)

- In addition to the two letter situations given above, there are six single letter situations for the first and last occurrences of a variable through a program path:
  - **-d** the variable does not exist yet (indicated by **-**), then it is defined ***(correct***).
  - **-u** variable does not exist (indicated by **-**), then it is used ***(incorrect)***.
  - **-k** variable does not exist (indicated by **-**), then it is killed ***(probably incorrect).***
  - **k-** :- ***not anomalous.*** The last thing done on this path was to kill the variable.
  - **d-** :- ***possibly anomalous***. The variable was defined and not used on this path. But this could be a global definition.
  - **u-** :- ***not anomalous.*** The variable was used but not killed on this path. Although this sequence is not anomalous, it signals a frequent kind of bug. If d and k mean dynamic storage allocation and deallocation, respectively, this could be an instance in which a dynamically allocated object was not returned to the pool after use.

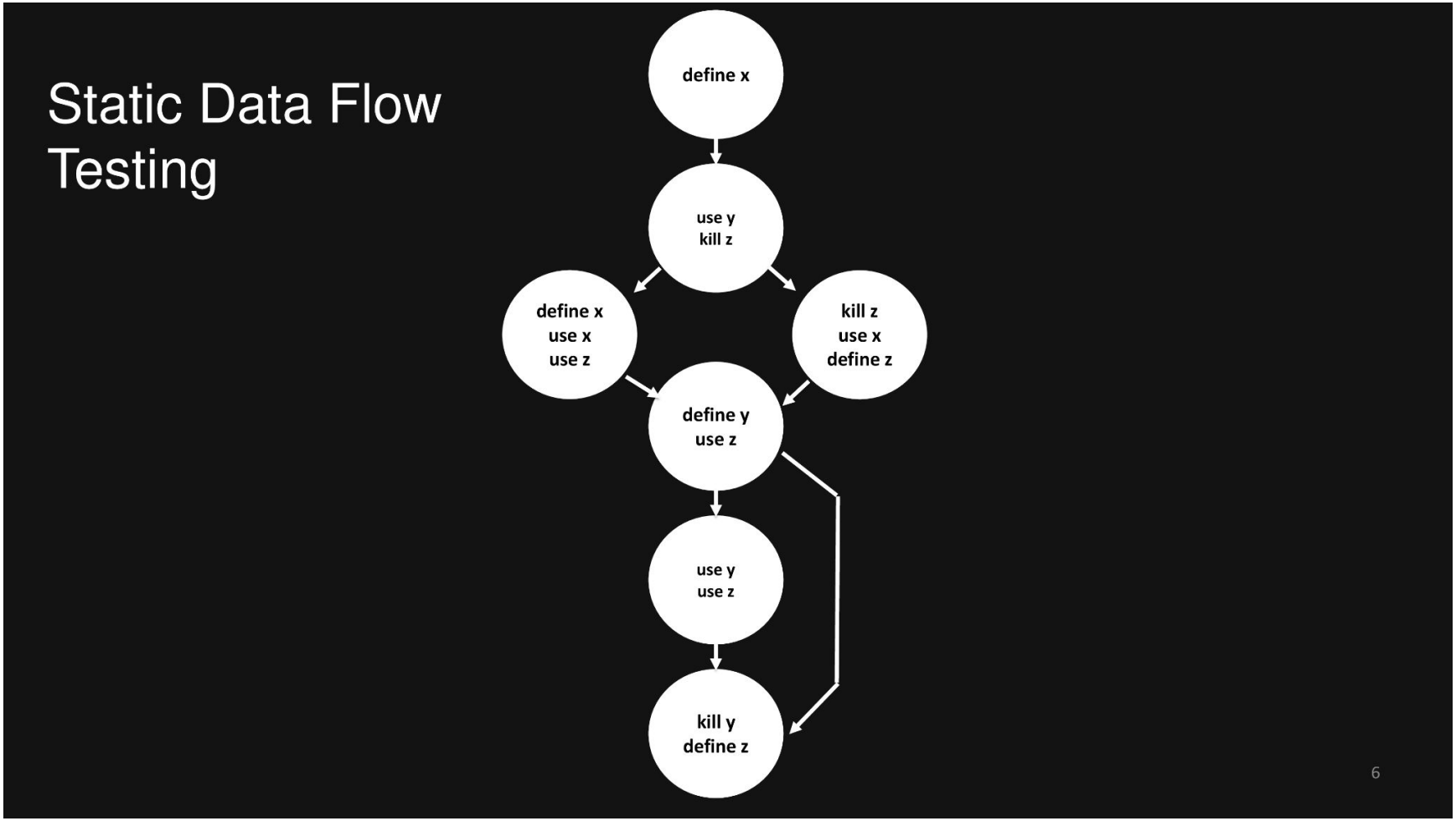
Static Data Flow Testing

- Static analysis is analysis done on source code without actually executing it. For example: source code syntax error detection is the result of static analysis.

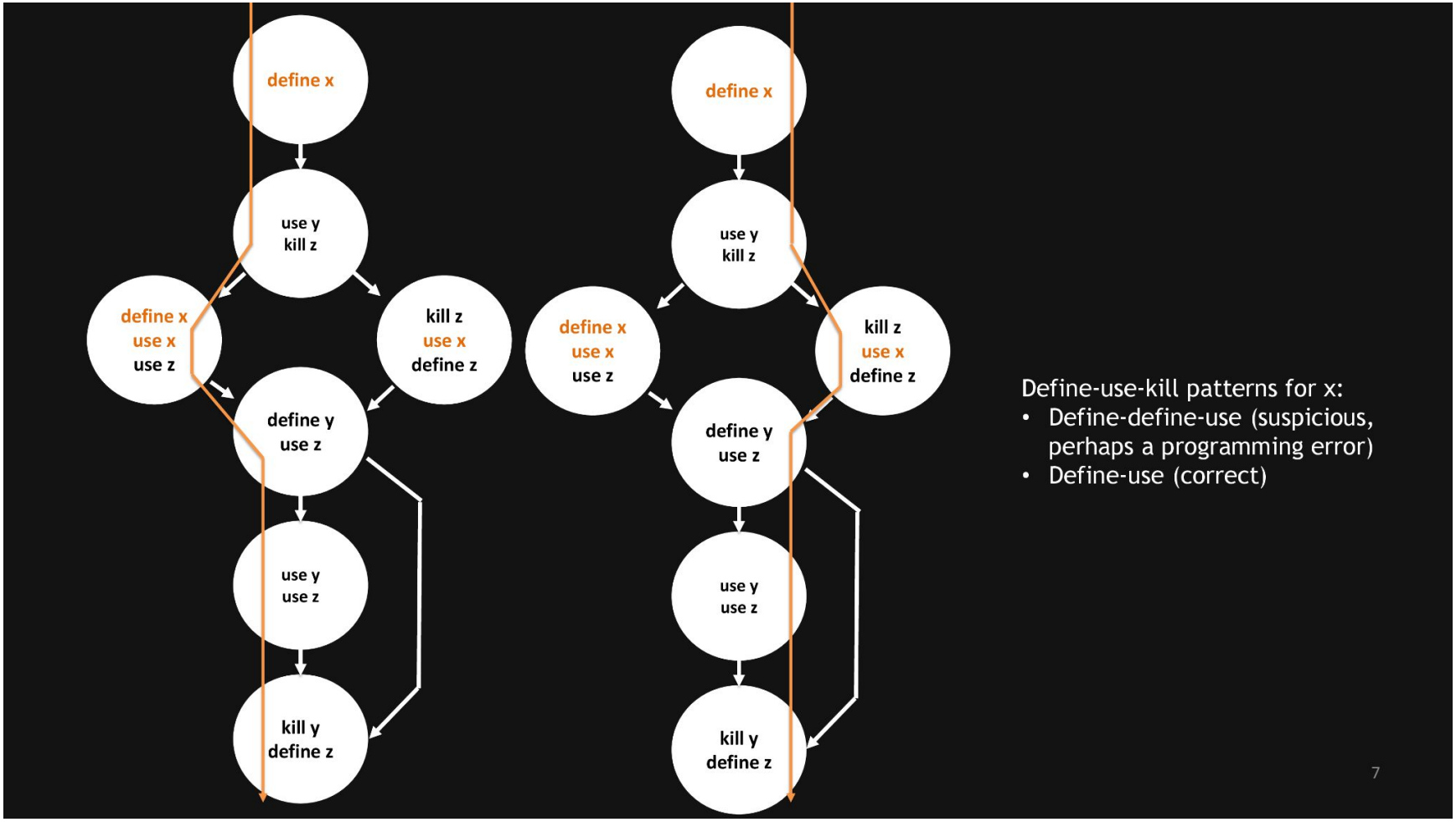
- If a data flow anomaly can be detected by static analysis methods, then it is found by the language processor (compiler/interpreter/linter).

- For example, compilers for languages which force variable declarations can detect (-u) and (ku) anomalies.

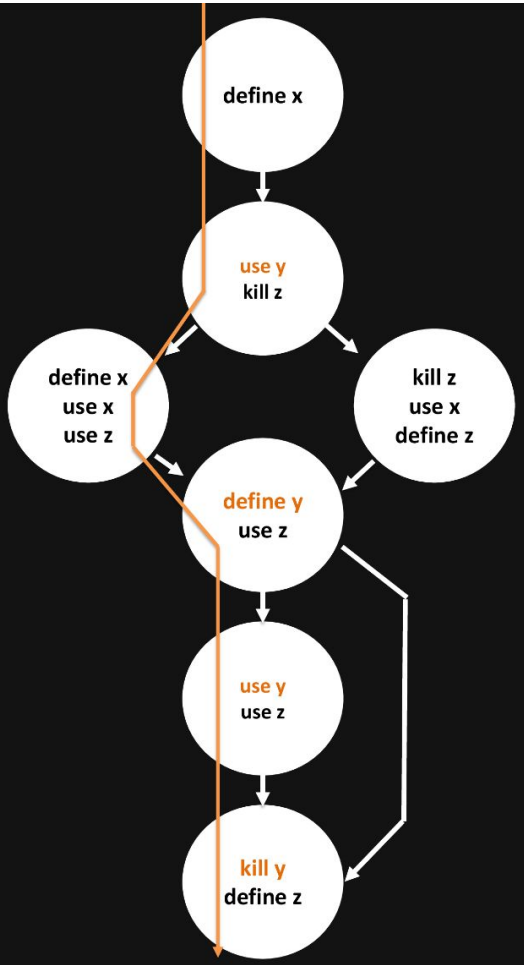
Static Data Flow Testing



Static Data Flow Testing



Static Data Flow Testing

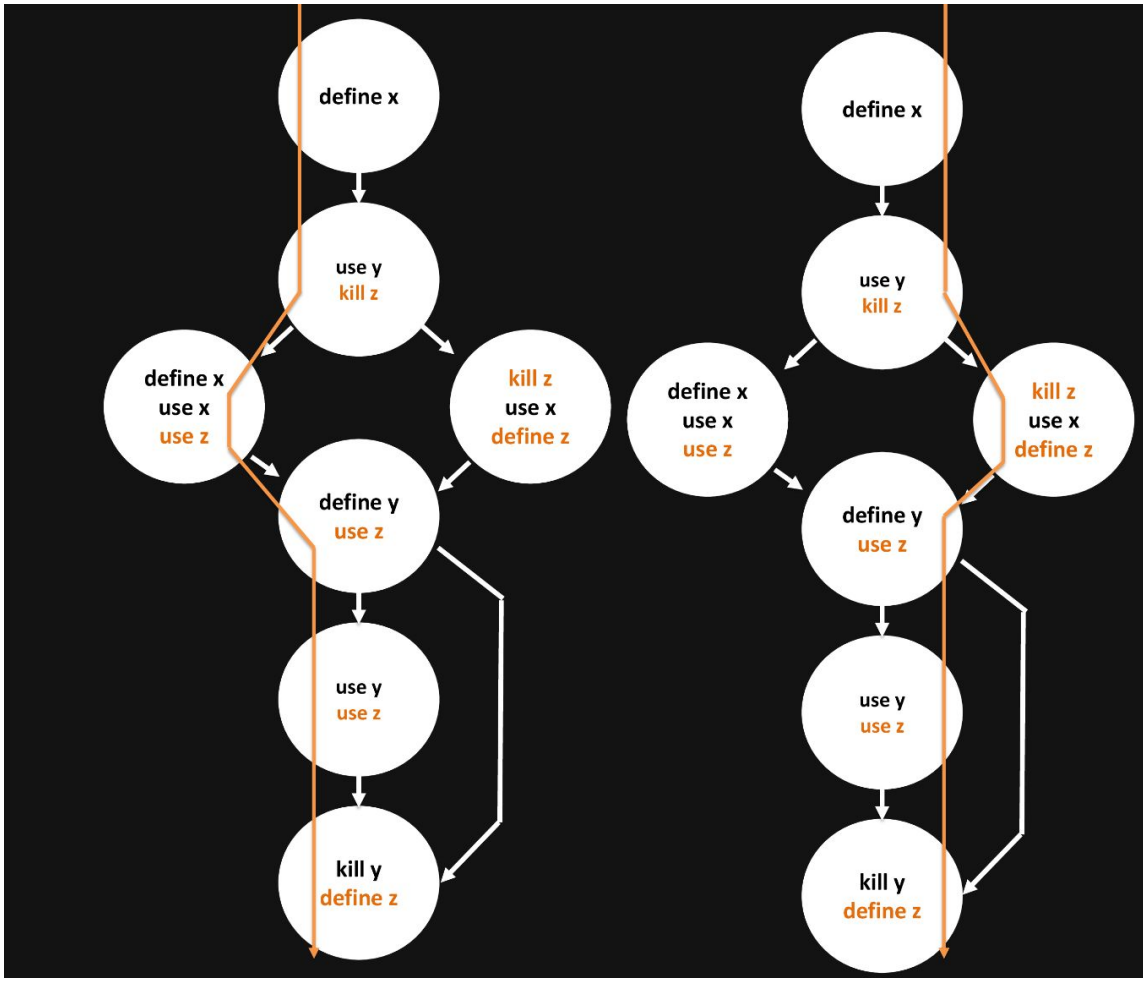


Define-use-kill patterns for y:

- -use (major blunder)
- Use-define (acceptable)
- Define-use (correct)
- Use-kill (acceptable)
- Define-kill (probably programming error)

8

Static Data Flow Testing



Define-use-kill patterns for z:

- -kill (programming error)
- Kill-use (major blunder)
- Use-use (correct)
- Use-define (acceptable)
- Kill-kill probably a programming error)
- Kill-define (acceptable)
- Define-use (correct)

Following problems have been discovered:

- X define-define
- Y -use
- Y define-kill
- Z -kill
- Z kill-use
- Z kill-kill

9

Static Analysis vs Dynamic Analysis

- Static analysis identifies defects *before* you run a program, i.e., between coding and unit testing phases.
- Static code analysis is used for a specific purpose in a specific phase of development.
- But there are some limitations of a static code analysis tool.
- **No Understanding of Developer Intent**

```
int calculateArea(int length, int width)
{
    return (length + width);
}
```

  - A static analysis tool may detect a possible overflow in the above calculation.
  - But it can't determine that function fundamentally *does not* do what is expected!
- **Possible Defects Lead to False Positives and False Negatives**
  - In some situations, a tool can only report that there is a possible defect.

```
int divide(void)
{
    int x;
    if (foo())
    {
        x = 0;
    }
    else
    {
        x = 5;
    }
    return (10/x);
}
```

  - If we know nothing about *foo()*, we do not know what value x will have.

Dynamic Data Flow Testing

- There are many things for which current notions of static analysis are inadequate. They are:
  - **Dead Variables (variables declared but never used):** Although it is often possible to prove that a variable is dead or alive at a given point in the program, the general problem is unsolvable.
  - **Arrays and Pointers:** Arrays are problematic in that the array is defined or killed as a single object, but reference is to specific locations within the array is possible. Array pointers are usually dynamically calculated, so there's no way to do a static analysis to validate the pointer value.
  - In many languages, dynamically allocated arrays contain garbage unless explicitly initialized and therefore, -u anomalies are possible.
  - **Records and Pointers:** The array problem and the difficulty with pointers is a special case of multipart data structures. We have the same problem with records (structs in C) and the pointers to them. Also, in many applications we create files and their names dynamically and there's no way to determine, without execution, whether such objects are in the proper state on a given path or, for that matter, whether they exist at all.
  - **Dynamic Subroutine and Function Names in a Call:** when subroutine or function name is a dynamic variable in a call. What is passed to the function is constructed on a specific execution path. There's no way, without executing the path, to determine whether the call is correct or not.

Data Flow Testing Strategies

- The data flow test strategies given below are differ in the extent to which predicate uses and/or computational uses of variables are included in the test set.
- Various types of data flow testing strategies in decreasing order of their effectiveness are:
  - **All - du Paths (ADUP):** The all-du-paths (ADUP) strategy is the strongest data-flow testing strategy. It requires that every du-path from every definition of every variable to every use of that definition be exercised under some test.
  - **All Uses Strategy (AU):**The all uses strategy is that at least one dc-path from every definition of every variable to every use of that definition be exercised under some test.
  - **All p-uses/some c-uses strategy (APU+C) :** For every variable and every definition of that variable, include at least one dc-path from the definition to every predicate use; if there are definitions of the variables that are not covered by the above prescription, then add computational use test cases as required to cover every definition.
  - **All c-uses/some p-uses strategy (ACU+P) :** To first ensure coverage by computational use cases and if any definition is not covered by the previously selected paths, add such predicate use cases as are needed to assure that every definition is included in some test.
  - **All Definitions Strategy (AD) :** The all definitions strategy asks only every definition of every variable be covered by at least one use of that variable, be that use a computational use or a predicate use.
  - **All Predicate Uses (APU), All Computational Uses (ACU) Strategies :** The all predicate uses strategy is derived from APU+C strategy by dropping the requirement that we include a c-use for the variable if there are no p-uses for the variable. The all computational uses strategy is derived from ACU+P strategy by dropping the requirement that we include a p-use for the variable if there are no c-uses for the variable.

Control Flow Analysis

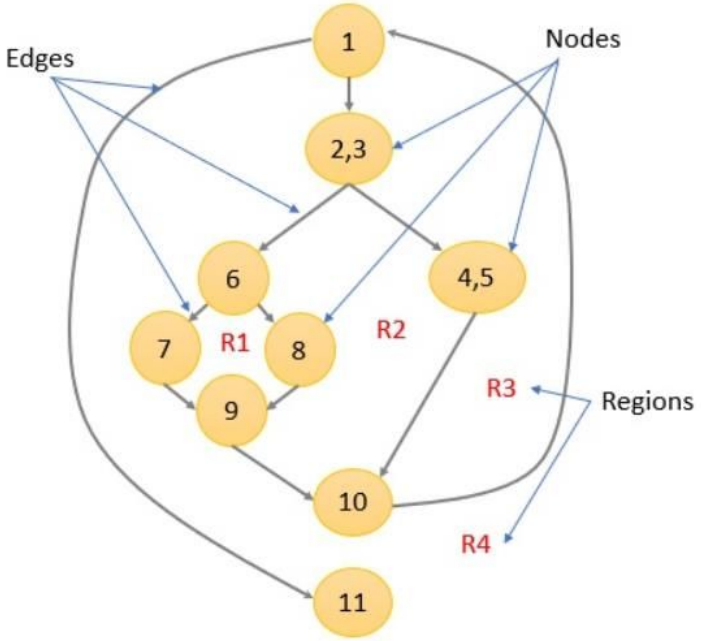
- Control flow analysis in software refers to the process of examining the order in which instructions or statements are executed within a program.
- It involves analyzing the flow of control, which determines the sequence of execution of different parts of the program.
- Control flow analysis helps developers understand how the program's logic is structured and how control is transferred between different program components, such as functions, loops, and conditional statements.
- Control Flow Graphs are the foundation of control flow testing.
- Modules of code are converted to graphs, the paths through the graphs are analyzed, and test cases created

Control Flow Structure

- The control flow structure addresses the sequence in which the instructions are executed.
- This aspect of structure reflects the iterations and loops in a program's design.
- Control flow analysis can also be used to identify unreachable (dead) code.
- It is often assumed that a large module takes longer to specify, design, code and test than a smaller one.
- If only the size of a program is measured, no information is provided on how often an instruction is executed as it is run.
- In fact many of the code metrics relate to the control flow structure, e.g. number of nested levels or cyclomatic complexity.

Control Flow/Program Graphs

- For a program written in an imperative programming language:
  - Program graph is a directed graph
  - Nodes are statement fragments
  - Edges represent flow of control



Control Flow/Program Graphs

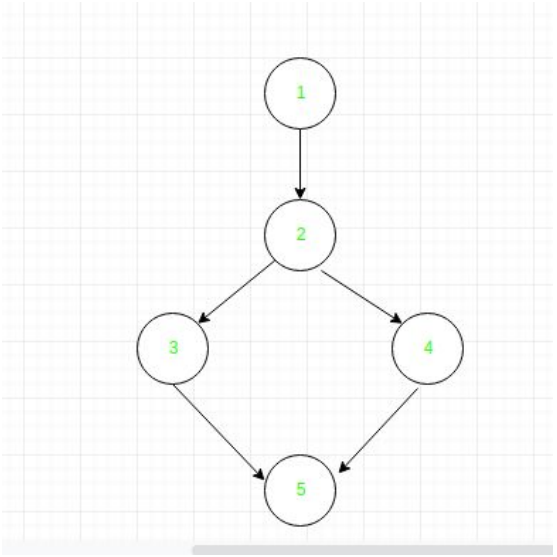
- Control Flow Graphs are the foundation of control flow analysis and testing.
- Modules of code are converted to graphs, the paths
- through the graphs are analyzed, and test cases created.
- Elements of Control Flow Graphs are:

  - Process Blocks
  - Decision Point
  - Junction Point

Example

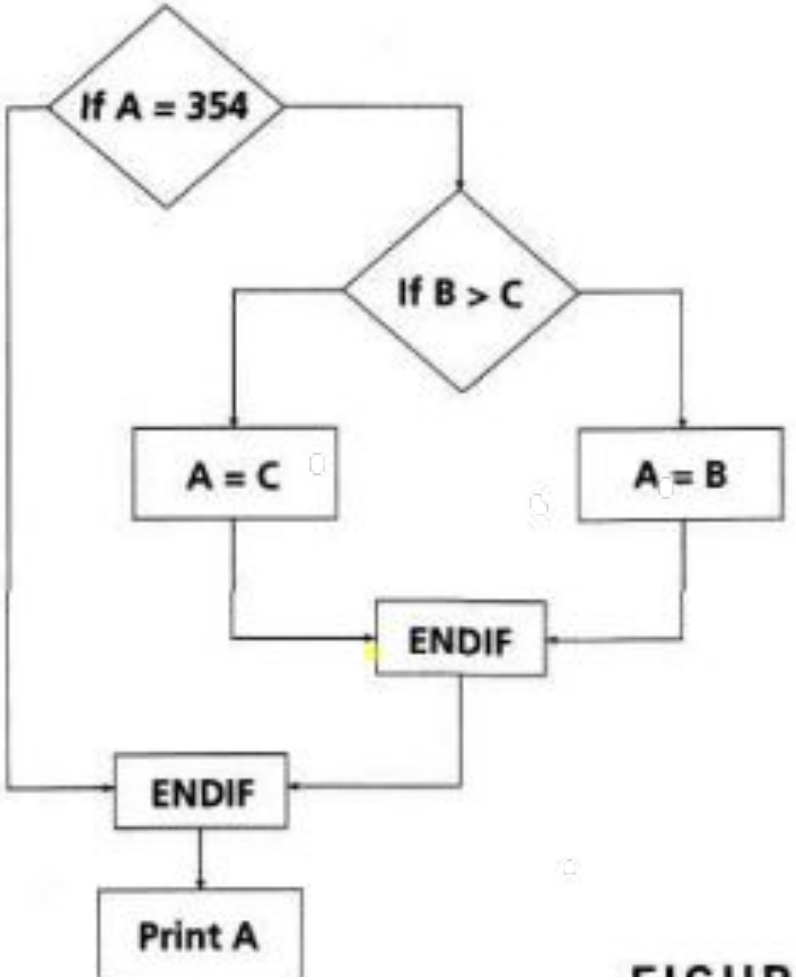
```
1. read x, y;
2. if (x>y) then
3.   a = x+1
   else
4.   a = y-1
5. print a;
```

Control flow graph of the above example:



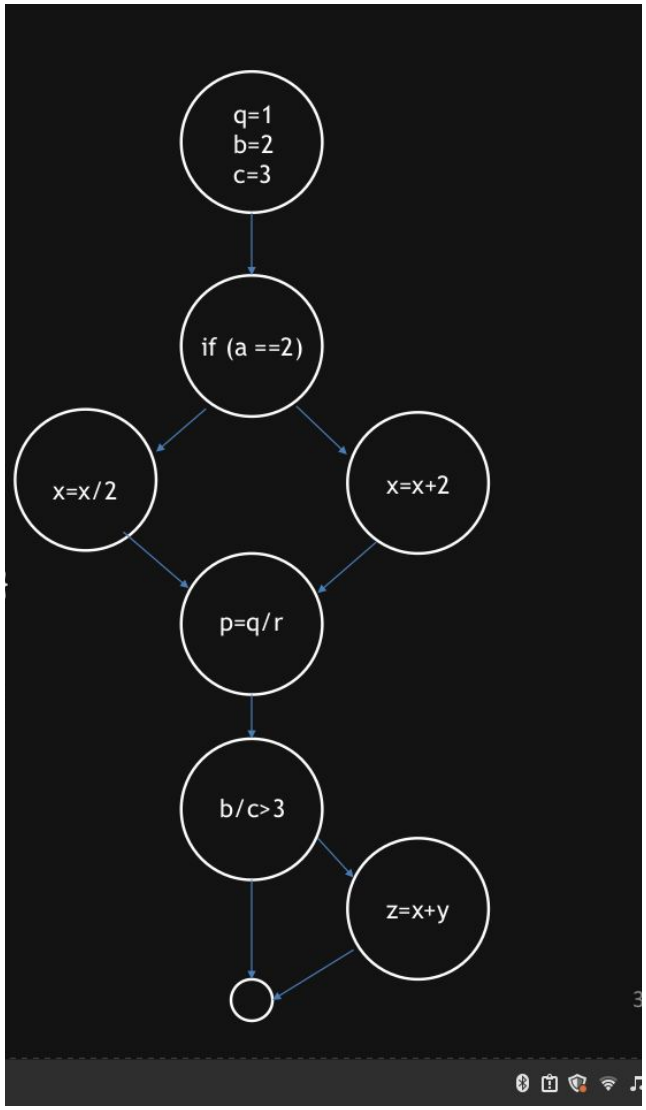
Control Flow Graph

```
if (A == 354)
    if (B > C)
        A = B;
    else
        A = C;
printf("%d", A);
```



Control Flow Graph

```
q=1
b=2
c=3
if (a==2)
   {x=x+2;}
else
   {x=x/2;}
p=q/r;
if (b/c>3)
   {z=x+y;}
```



Control Flow Testing

- In Control Flow testing, different levels of test coverage are defined:
  - **Level 1** Statement Coverage means that every statement within the module is executed at least once.
  - **Level 2** Decision or Branch Coverage means each decision that has a TRUE and FALSE outcome is evaluated at least once.
  - **Level 3** Condition Coverage; atomic (partial) condition in the test adopt both a TRUE and a FALSE value.
  - **Level 4** Decision/Condition Coverage, test cases are created for every decision and every condition.
  - **Level 5** Multiple Condition Coverage, requires that all TRUE-FALSE combinations of the atomic partial conditions be exercised.
  - **Level 6**, when a module has loops in the code paths, a reduction can be made by limiting loop execution.
  - **Level 7**, Path Coverage, test cases are created for each path.

Structural Testing/Basis Path Testing

- Structural testing is based on using specific knowledge of the program source text to define the test cases.
  - Contrast with black-box testing where the program text is not seen but only hypothesized.

- Structural testing methods are amenable to:
  - Rigorous definitions
  - Control flow, data flow, coverage criteria
  - Mathematical analysis
  - Graphs, path analysis
  - Precise measurement
  - Metrics, coverage analysis.

Structural Testing/Basis Path Testing

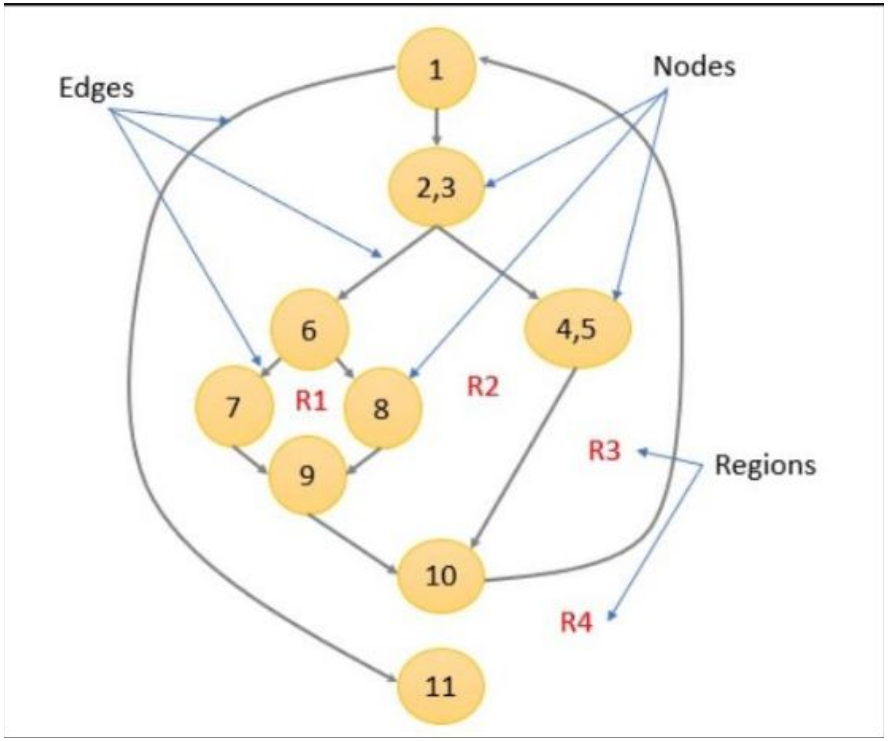
- Derive the control flow graph from the software module.
- Compute the graph's Cyclomatic Complexity (C).
  - Cyclomatic Complexity (C) of a graph is computed as:

    C = Edges – Nodes + 2

    P : the number of connected components in the Control Flow Graph.
  - Alternative calculation of Cyclomatic Complexity (C) is given by the formula:

    C = Number of Regions
- Identify the set of C ***basis paths***.
  - Basis paths are linearly independent paths through the control flow graph.
  - A basis path is a unique path through the program that exercises a different combination of decision outcomes
- Create a test case for each basis path.

Structural Testing/Basis Path Testing

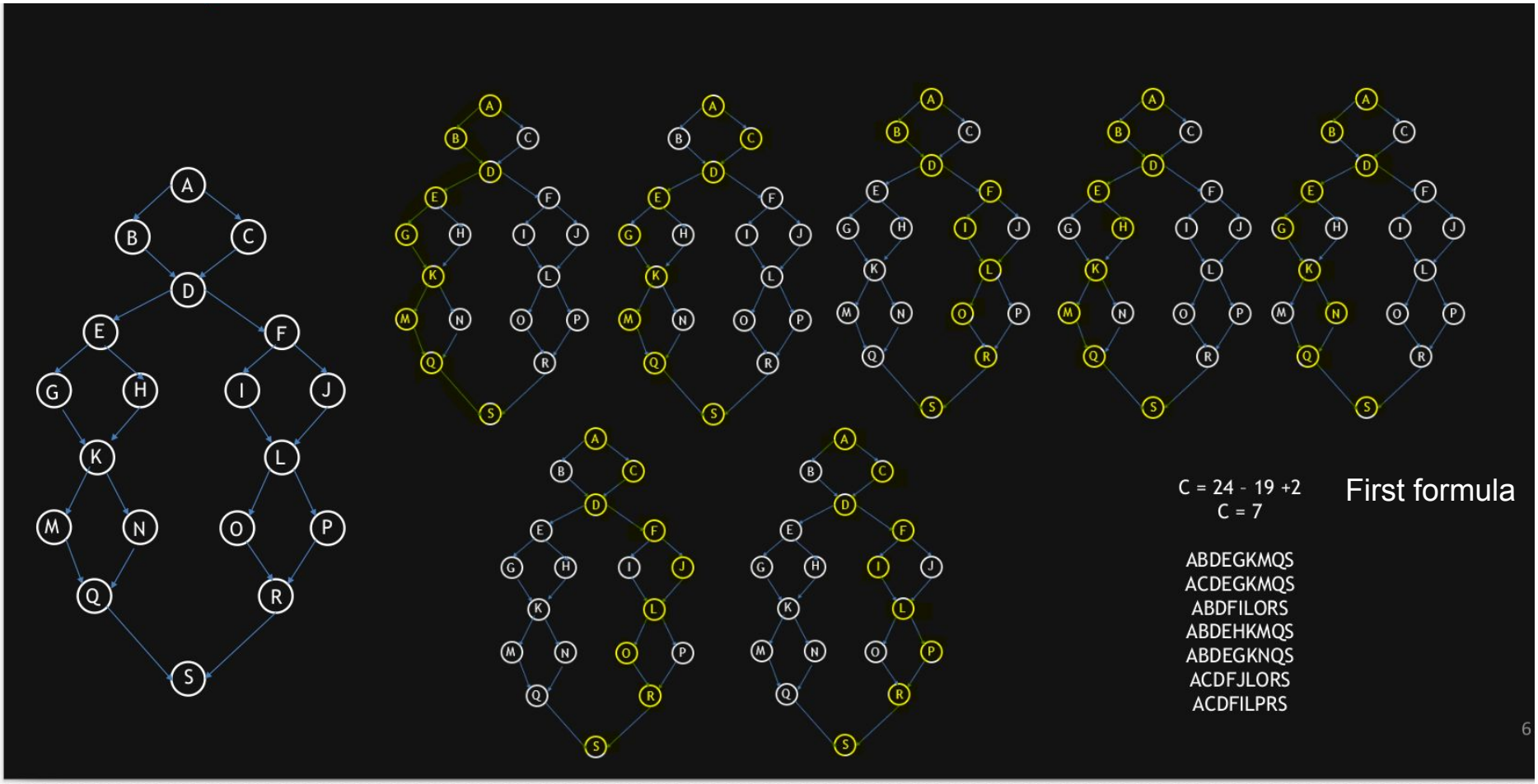


A connected component is a subset of vertices within the graph where each vertex is reachable from every other vertex in the subset via edges.
A Region is an area bounded by edges and nodes of the graph (area outside the graph is also counted as a region).

The above graph has 1 connected component, and 4 regions.
Hence :

- C = 11 - 9 + 2 = **4** (via first formula) or
- C = **4 (**via second formula**)**

Structural Testing/Basis Path Testing



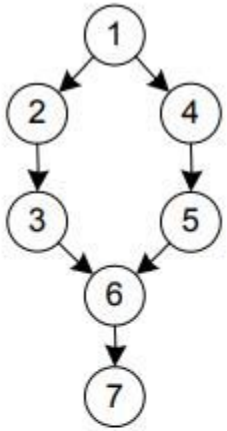
Structural Testing/Basis Path Testing

- Paths derived from the control flow graph construct.
- When a test case executes, it traverses a path.
- Huge number of paths implies some simplification is needed.
- *Problem*: infeasible number of paths.
- By itself, path testing can lead to a false sense of security.

Program Graphs of Structured Programming Constructs

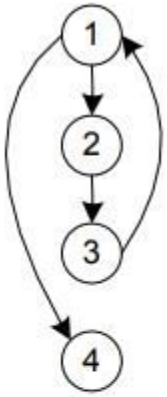
If-Then-Else

```
1  If <condition>
2     Then
3        <then statements>
4     Else
5        <else statements>
6  End If
7  <next statement>
```



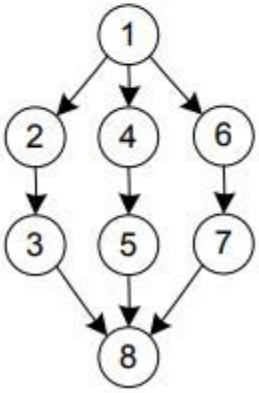
Pre-test Loop

```
1  While <condition>
2     <repeated body>
3  End While
4  <next statement>
```



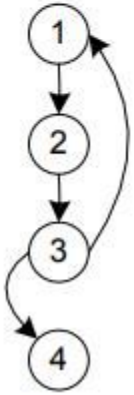
Case/Switch

```
1  Case  n Of 3
2     n=1:
3        <case 1  statements>
4     n=2:
5        <case 2  statements>
6     n=3:
7        <case 3  statements>
8  End Case
```



Post-test Loop

```
1  Do
2     <repeated body>
3  Until <condition>
4  <next statement>
```



An Example: Program Text and Graph

```
output ("Enter 3 integers")
input(a,b,c)
output("Side a b c:", a, b, c)
if (a < b) and (b < a+c) and (c < a+b) then
    isTriangle <--- true
else
    isTriangele <--- false
if  isTriangle then
    if (a = b) and (b = c) then
        output ("equilateral")
    else if (a != b) and (a != c) and (b != c) then
        output ("scalene")
    else
        output ("isosceles")
else
    output("not a triangle")
```