Module 3

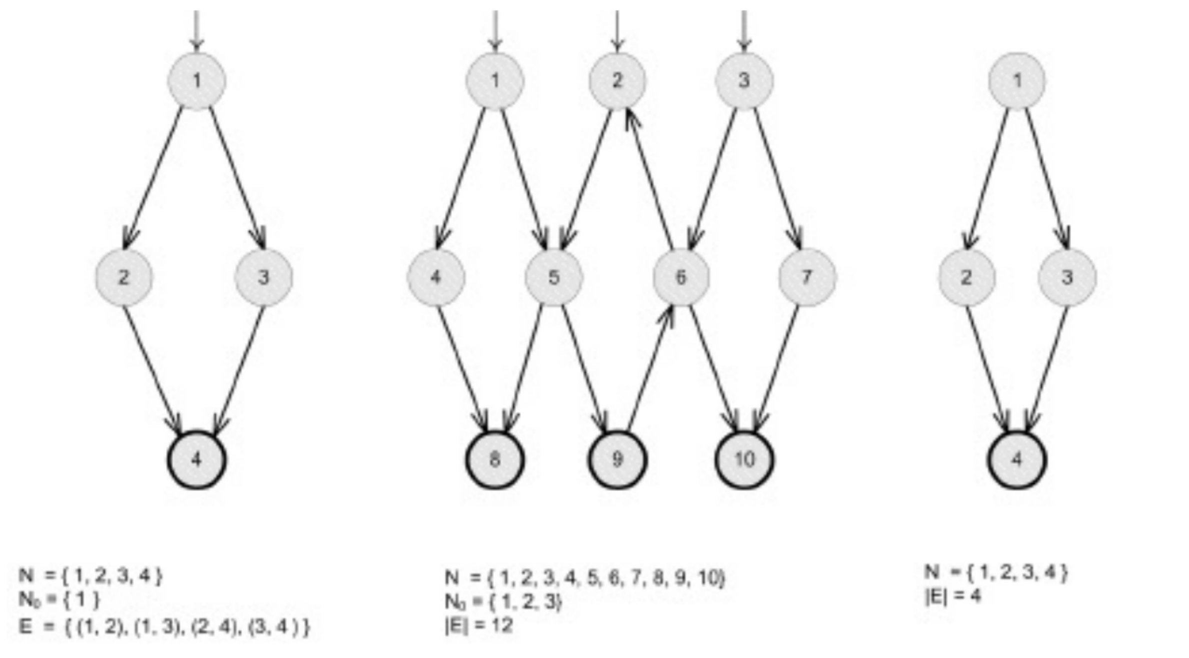
farisbashatm@gmail.com

Graph Coverage

- **Introduction**: The chapter discusses test coverage criteria using graphs to design and evaluate tests, which is a step towards the RIPR model ensuring tests "reach" specific locations in a graph model.
- **Basic Theory**:
    - You're describing a directed graph $G = (N, E)$, where:
        - N is the set of nodes.
        - E is the set of edges, which is a subset of $N \times N$, meaning each edge is a pair of nodes.
        - N_0 is the set of initial nodes, a subset of N.
        - N_f is the set of final nodes, also a subset of N.

  So, formally:
    - $G = (N, E)$
    - $N_0 \subseteq N$
    - $N_f \subseteq N$
- **Application**: Graphs are derived from software artifacts like control flow graphs, design structures, finite state machines, statecharts, and use cases. The abstraction of an artifact into a graph simplifies test case generation and evaluation.
- **Graph Criteria**: Testers must "cover" specific parts of the graph by traversing certain paths. A test path starts at an initial node N_0 and ends at a final node N_f.
- **Syntactic and Semantic Reachability**: Nodes or edges are syntactically reachable if a path exists from an initial node to them. Semantic reachability considers if such a path can be executed with some input.
- **Paths and Cycles**: Paths are sequences of connected nodes; cycles start and end at the same node. Testing often requires ensuring paths cover necessary sequences in the graph.
- **SESE Graphs**: Single-Entry Single-Exit (SESE) graphs have exactly one initial and one final node, ensuring every test starts at the initial node and ends at the final node.
- **Graph Terminology**:
    - Nodes (or vertices) and edges (or arcs) represent the structural elements of the graph. Testing criteria often demand visiting or touring nodes, edges, or subpaths.
    - To ensure a graph is useful for test generation:
        - N, N_0, and N_f must each contain at least one node.

        - It's beneficial to focus on specific parts of the graph at times.
        - N0 may consist of multiple initial nodes, forming a set.
        - Multiple initial nodes are necessary for certain software artifacts like classes with multiple entry points.
        - In some cases, the graph may be constrained to have only one initial node.
        - Edges are represented as (n_i, n_j), where n_i is the predecessor and n_j is the successor node.
- **Test Path**:
    - A test path represents the execution of a set of test cases, starting from an initial node and ending at a final node.
    - A path p, possibly of length zero, that starts at some node in N_0 and ends at some node in N_f.
    - In testing paths, we borrow terminology from traveling:
        - A test path "visits" a node if that node is on the path.
        - Similarly, a test path "visits" an edge if that edge is traversed.
        - For subpaths, we use the term **tour**. A test path tours a subpath if the subpath is part of the main path.
- **Minimal Test Paths**: A minimal set of test paths is the smallest set that still satisfies the testing criteria, ensuring cost-effective testing.

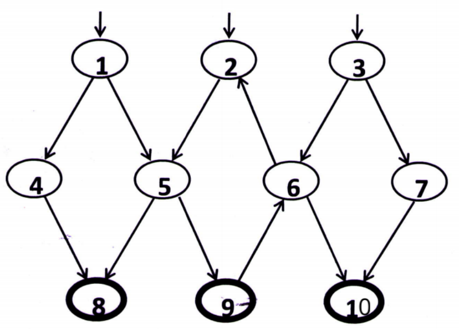


(a) A graph with a single initial node (b) A graph with mutiple initial nodes (c) A graph with no initial node

This chapter aims to build a solid foundation for understanding graph-based test coverage criteria, preparing for practical applications in software testing.

Example Question

15 a) 8



1.Give the sets N_0 ,N_f, N and E for the above graph.

2.Give a path that is not a test path.

3.List all test paths in the above graph.

4.From the above graph, find test case inputs such that the corresponding test path visits edge (n1, n3).

Let's address each part of the question based on the given graph.

1. Give the sets N_0, N_f, N, and E for the above graph.

- N: Set of all nodes
  $N = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- N_0: Set of initial nodes (nodes with incoming arrows from outside the graph)
  $N_0 = \{1, 2, 3\}$
- N_f: Set of final nodes (nodes with double circles)
  $N_f = \{8, 9, 10\}$
- E: Set of edges (pairs of connected nodes)
  $E = \{(1, 4), (1, 5), (2, 5), (2, 6), (3, 6), (3, 7), (4, 8), (5, 8), (5, 9), (6, 9), (6, 10), (7, 10)\}$

2. Give a path that is not a test path.

A path that is not a test path would be any path that does not start from an initial node N_0 or does not end at a final node N_f. For example:
$(5, 9)$

This is not a test path because it neither starts at an initial node nor ends at a final node.

3. List all test paths in the above graph.

A test path starts at an initial node N_0 and ends at a final node N_f. The test paths are:

$$
\begin{gathered}
(1, 4, 8) \\
(1, 5, 8) \\
(1, 5, 9) \\
(2, 5, 8) \\
(2, 5, 9) \\
(3, 6, 10) \\
(3, 7, 10)
\end{gathered}
$$

4. From the above graph, find test case inputs such that the corresponding test path visits edge $(1, 3)$.

The graph does not contain the edge $(1, 3)$. Therefore, there is no test path that visits this edge.

===

Graph Coverage Criteria

> **Graph Coverage**: Given a set TR of test requirements for a graph criterion C, a test set T satisfies C on graph G if and only if for every test requirement tr in TR, there is at least one test path p in path(T) such that p meets tr.

Graph coverage criteria help define and measure the adequacy of tests on software represented as graphs. These criteria are divided into:

1. **Structural Graph Coverage Criteria**: Focus on the control flow within the software.
2. **Data Flow Coverage Criteria**: Focus on the flow of data through the software.

Test requirements are defined in terms of paths, nodes, and edges in the graph. A test set satisfies a criterion if it meets all test requirements.

Structural Coverage Criteria

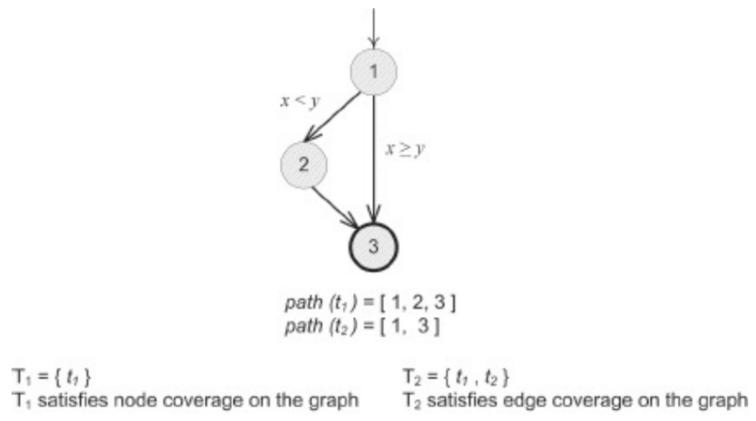
These criteria specify test requirements (TR) to visit nodes and edges in a graph.

Node Coverage (NC) (Statement Coverage)

- **Definition**: Every node in the graph must be visited by at least one test path.
- **Technical Definition**: TR contains each reachable node in G. Formally, for each reachable node $n \in N$, TR contains the predicate “visit n.”

Edge Coverage (EC) (Branch Coverage)

- **Definition**: Every edge in the graph must be visited by at least one test path.
- **Technical Definition**: TR contains each reachable path of length up to 1, inclusive, in G.


1
$x < y$
2
$x \geq y$
3
path (t_1) = [1, 2, 3]
path (t_2) = [1, 3]
T_1 = { t_1 }
T_1 satisfies node coverage on the graph
T_2 = { t_1 , t_2 }
T_2 satisfies edge coverage on the graph

(a) Node Coverage (b) Edge Coverage

Extended Coverage Criteria

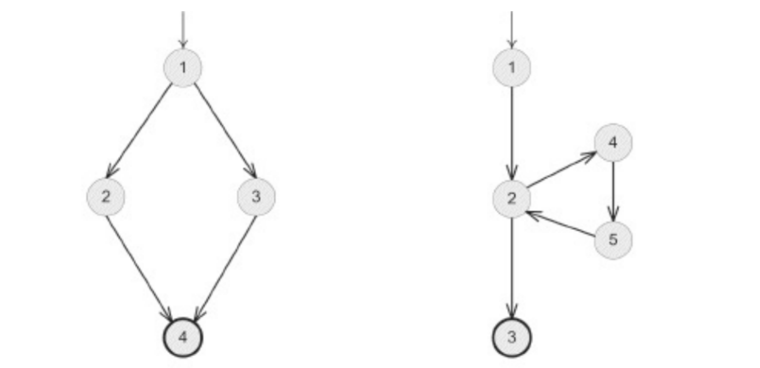
These criteria further extend the basic node and edge coverage.

Edge-Pair Coverage (EPC)

- **Definition**: Each reachable path of length up to 2 must be toured.
- **Technical Definition**: TR contains each reachable path of length up to 2, inclusive, in G.

Prime Path Coverage (PPC)

- **Definition**: Each prime path (a simple path not part of any longer simple path) must be toured.
- **Technical Definition**: TR contains each prime path in G. A path from n_i to n_j is a prime path if it is a simple path and it does not appear as a proper subpath of any other simple path.



Prime Paths = { [1, 2, 4], [1, 3, 4] }
path (t_1) = [1, 2, 4]
path (t_2) = [1, 3, 4]
$T_1 = \{t_1, t_2\}$
T_1 satisfies prime path coverage on the graph

(a) Prime Path Coverage on a Graph With No Loops

Prime Paths = { [1, 2, 3], [1, 2, 4, 5], [2, 4, 5, 2], [4, 5, 2, 4], [5, 2, 4, 5], [4, 5, 2, 3] }
path (t_3) = [1, 2, 3]
path (t_4) = [1, 2, 4, 5, 2, 4, 5, 2, 3]
$T_2 = \{t_3, t_4\}$
T_2 satisfies prime path coverage on the graph

(b) Prime Path Coverage on a Graph With Loops

Round Trip Coverage

- **Simple Round Trip Coverage (SRTC)**
  - **Definition**: At least one round-trip path for each node must be taken.
  - **Technical Definition**: TR contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.
- **Complete Round Trip Coverage (CRTC)**
  - **Definition**: All round-trip paths for each node must be taken.
  - **Technical Definition**: TR contains all round-trip paths for each reachable node in G.

Complete Path Coverage (CPC)

- **Definition**: All possible paths in the graph must be toured.
- **Technical Definition**: TR contains all paths in G. Not feasible for graphs with cycles due to infinite paths.

Specified Path Coverage (SPC)

- **Definition**: A specified set of paths must be toured.
- **Technical Definition**: TR contains a set S of test paths, where S is supplied as a parameter.
- **Example**: Paths provided by a customer or usage scenarios.

Touring, Sidetrips, and Detours

In the context of graph coverage criteria, the concepts of touring, sidetrips, and detours are essential for understanding how to handle paths with loops and infeasible test requirements.

Example Figures (Figure 7.8 and Figure 7.9)

- **Figure 7.8**: Illustrates a graph with a loop from node 3 to node 4 and back to node 3.
- **Figure 7.9(a)**: Shows a tour with a sidetrip where the sequence of edges executed includes a sidetrip through the loop.
- **Figure 7.9(b)**: Shows a tour with a detour where the sequence of edges executed includes a detour that bypasses an edge in the subpath.

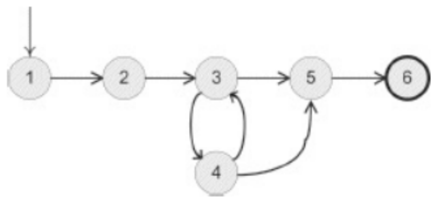


Figure 7.8. Graph with a loop.

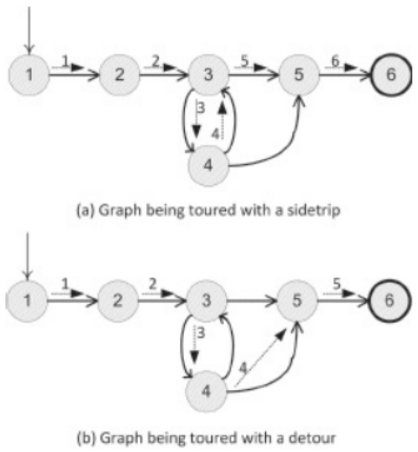

1
2
3
5
6
4
(a) Graph being toured with a sidetrip
(b) Graph being toured with a detour

Figure 7.9. Tours, sidetrips, and detours in graph coverage.

Key Definitions

Tour

- **Definition**: A test path p is said to tour subpath q if and only if q is a subpath of p.
- **Example**: For the graph in Figure 7.8, if $q = [2, 3, 5]$, then any path p that contains the subpath $[2, 3, 5]$ exactly as it appears tours q.

Tour With Sidetrips

- **Definition**: A test path p is said to tour subpath q with sidetrips if and only if every edge in q is also in p in the same order, but p may temporarily leave the path from a node and then return to the same node.
- **Example**: In Figure 7.9(a), if $p = [1, 2, 3, 4, 3, 5, 6]$ and $q = [2, 3, 5]$, p tours q with sidetrips because the subpath $[2, 3, 5]$ is visited in the correct order, although p includes a sidetrip through node 4.

Tour With Detours

- **Definition**: A test path p is said to tour subpath q with detours if and only if every node in q is also in p in the same order, but p may leave the path from a node and then return to the next node on the path (skipping an edge).
- **Example**: In Figure 7.9(b), if $p = [1, 2, 3, 4, 5, 6]$ and $q = [2, 3, 5]$, p tours q with detours because the nodes $[2, 3, 5]$ are visited in the correct order, although p includes a detour from node 3 to node 5 via node 4, bypassing the direct edge (3, 5).

Data Flow Criteria in Testing

Data flow testing focuses on the paths that data values take through a program. The key idea is to ensure that values are correctly defined (created) and used (accessed) within the program. Here's a simple overview:

Key Concepts:

1. **Definition (def)**: The point where a value is assigned to a variable (e.g., through assignment or input).
2. **Use**: The point where a variable's value is accessed.
3. **du-pair**: A pair consisting of a definition and a use of the same variable. It stands for definition-use pair.
4. **du-path**: A simple path that is "def-clear" for a variable from a definition to a use, meaning the variable is not redefined along the path.
5. **Def-clear path**: A path where the variable is not redefined.

Simple Example:

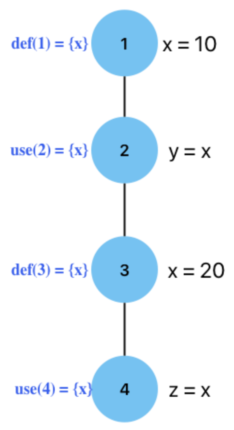
Consider a simple program segment with a variable `x`:

```
1: x = 10   # Definition of x
2: y = x    # Use of x
3: x = 20   # Definition of x
4: z = x    # Use of x
```

In this example:

- The du-pair (1, 2) indicates that the definition of `x` at line 1 is used at line 2.
- The du-pair (3, 4) indicates that the definition of `x` at line 3 is used at line 4.

Example with Graph:



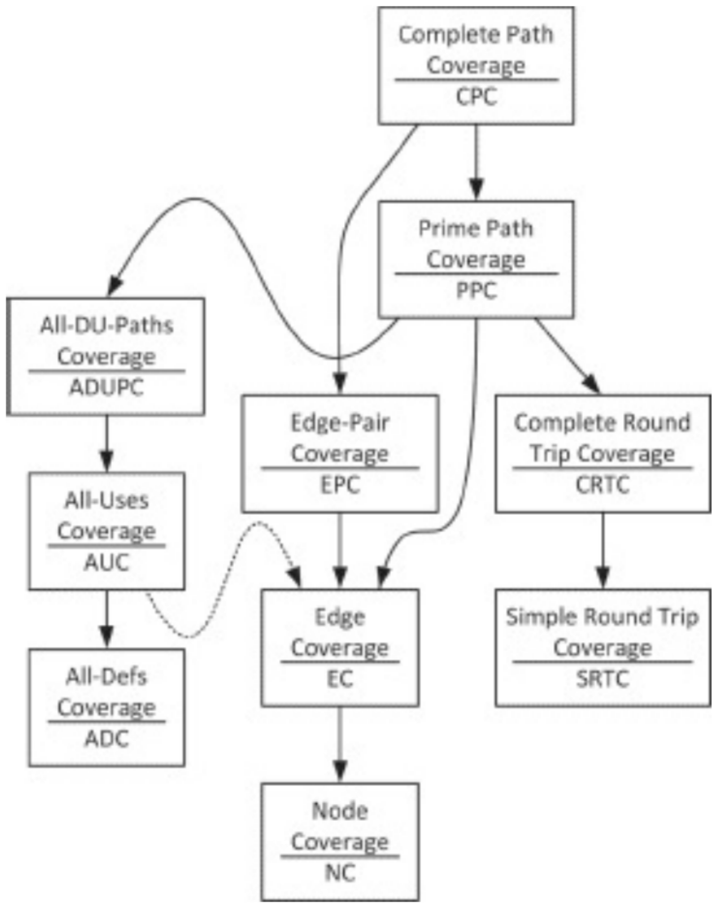
- **def(1) = {x}**: `x` is defined at node 1.
- **use(2) = {x}**: `x` is used at node 2.
- **def(3) = {x}**: `x` is defined again at node 3.
- **use(4) = {x}**: `x` is used at node 4.

Coverage Criteria

1. **All-Defs Coverage (ADC)**: Ensure that for each definition of a variable, at least one path to a use of that variable is executed.
   - **Example**: Ensure that the path from line 1 to line 2 is tested, as well as the path from line 3 to line 4.
2. **All-Uses Coverage (AUC)**: Ensure that for each definition-use pair, at least one path is executed.
   - **Example**: Ensure that the paths (1, 2) and (3, 4) are both tested.
3. **All-du-Paths Coverage (ADUPC)**: Ensure that every possible path from each definition to each use is executed.
   - **Example**: In a more complex graph, this would mean testing all possible paths from each definition of a variable to each use.

Subsumption Relationships Among Graph Coverage Criteria

Definition:
Subsumption relationships among graph coverage criteria describe how satisfying one coverage criterion often ensures that other, less stringent criteria are also satisfied.



- **Edge Coverage subsumes Node Coverage**
  - Traversing every edge ensures visiting every node.
  - Exception: Nodes with no edges.
- **Node Coverage does not subsume Edge Coverage**
  - Node Coverage can be satisfied without covering all edges.
- **Prime Path Coverage generally subsumes Edge-Pair Coverage**
  - Exception: Nodes with self-loops.
- **Best-Effort Touring**
  - Ensures subsumption even if some test requirements are infeasible.
- **Data Flow Criteria Assumptions**
  - Every use is preceded by a def.
  - Every def reaches at least one use.
  - Same variables used on each outgoing edge.
- **All-Uses Coverage subsumes All-Defs Coverage**
  - Ensures every def is used.
- **All-du-Paths Coverage subsumes All-Uses Coverage**

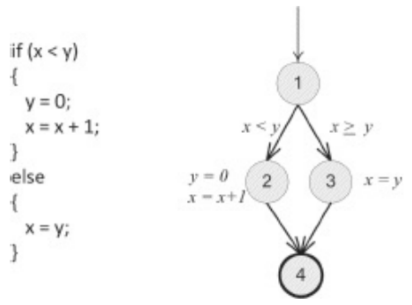
  - Ensures every def reaches every possible use.
- **All-Uses Coverage subsumes Edge Coverage**
  - Ensures every edge is executed at least once.
- **Prime Path Coverage subsumes All-du-Paths Coverage**
  - Prime paths are simpler to compute than data flow relationships.

Graph Coverage for Source Code

Definition:
Graph coverage criteria for source code measure how thoroughly program structures are tested by generating control flow graphs (CFGs) that represent the paths through the code.

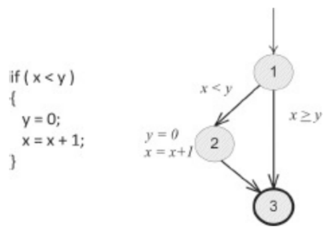
Structural Graph Coverage for Source Code

- **Control Flow Graph (CFG):**
  - Nodes represent basic blocks (maximal sequences of statements executed together).
  - Edges represent control flow between blocks.
- **If-Else Structure**
  - Creates two basic blocks.
  - Conditional test creates a decision node.
  - See Figure 7.16.



```
if (x < y)
{
  y = 0;
  x = x + 1;
}
else
{
  x = y;
}
```

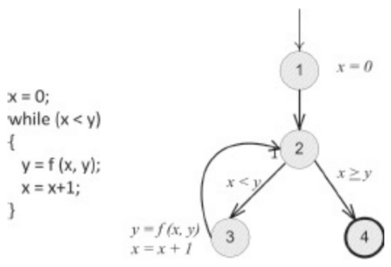
  - **Figure 7.16.** CFG fragment for the *if-else* structure.
- **If Without Else**
  - Creates three nodes.
  - Traverses all nodes but not all edges.
  - See Figure 7.17.



```
if ( x < y )
{
  y = 0;
  x = x + 1;
}
```

  - **Figure 7.17.** CFG fragment for the *if* structure without an *else*

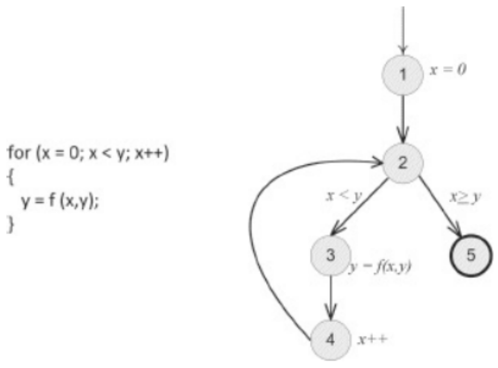
- **While Loop**
    - Includes a dummy node for loop iterations.
    - Decision node for the test.
    - See Figure 7.19.



```
x = 0;
while (x < y)
{
  y = f (x, y);
  x = x+1;
}
```

- **Figure 7.19.** CFG fragment for the *while* loop structure.

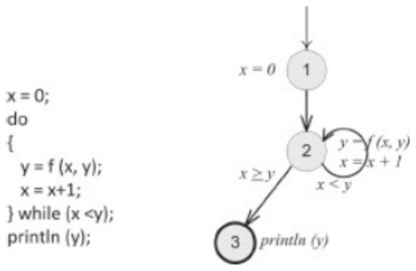
- **For Loop**
    - Initialization, test, and increment in different nodes.
    - Slightly different from the while loop.
    - See Figure 7.20.



```
for (x = 0; x < y; x++)
{
  y = f (x,y);
}
```

- **Figure 7.20.** CFG fragment for the *for* loop structure.

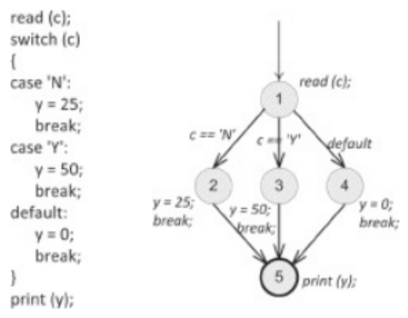
- **Do-While Loop**
    - Loop body executed at least once.
    - See Figure 7.21.



```
x = 0;
do
{
  y = f (x, y);
  x = x+1;
} while (x <y);
println (y);
```

- **Figure 7.21.** CFG fragment for the *do-while* structure.

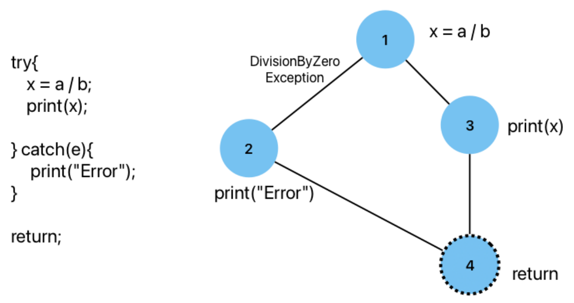
- **Switch/Case Statement**
    - Multi-way branching.
    - Omitting a break reflects fall-through behavior.
    - See Figure 7.23.



```
read (c);
switch (c)
{
case 'N':
    y = 25;
    break;
case 'Y':
    y = 50;
    break;
default:
    y = 0;
    break;
}
print (y);
```

Figure 7.23. CFG fragment for the *case* structure.

- **Exception Handling (try-catch)**
  - Represents control flow with exceptions.



```
try{
   x = a / b;
   print(x);

} catch(e){
    print("Error");
}

return;
```

Path Selection Criteria

Path selection criteria are essential for determining which paths in a control flow graph (CFG) to test in order to effectively detect defects in the code. Here are four key path selection criteria:

```python
def check_number_sign(number):
    if number > 0:
        return "Positive"
    elif number < 0:
        return "Negative"
    else:
        return "Zero"
```

1. All-Path Coverage Criterion

- **Definition:** Tests all possible paths in a CFG.
- **Example:** In a simple program with a loop and conditionals, this criterion would require testing every combination of paths through the loop and conditionals.
- **Challenge:** Impractical for large or complex programs with numerous or infinite paths.
- **Example in Program:**
- Path 1: The condition `number > 0` is true.
- Path 2: The condition `number < 0` is true.
- Path 3: Both conditions are false (number is zero).

2. Statement Coverage Criterion

- **Definition:** Ensures every individual statement in the code is executed at least once.
- **Application:** Simple and ensures basic code functionality, but may miss logic errors.
- **Example in Program:**
  - Execute paths where `number > 0`, `number < 0`, and `number == 0`.

3. Branch Coverage Criterion

- **Definition:** Ensures each branch (i.e., true/false outcomes of conditionals) is executed at least once.
- **Application:** More thorough than statement coverage, detecting errors in conditional logic.
- **Example in Program:**
  - Test with a number greater than 0.
  - Test with a number less than 0.
  - Test with 0.

4. Predicate Coverage Criterion

- **Definition:** Ensures all possible combinations of Boolean conditions in decision points are tested.
- **Example:** In a program with a complex conditional like `if (A && (B || C))`, predicate coverage would require testing combinations where A, B, and C are true and false to cover all logical paths.
- **Application:** Detects subtle bugs by thoroughly testing complex conditionals.
- **Example in Program:**
  - The conditions `number > 0` and `number < 0` are tested for both true and false cases.

GRAPH COVERAGE FOR DESIGN ELEMENTS

Graph coverage for design elements involves analyzing and testing the connections between different components of a software design using graphical representations, ensuring thorough testing of all design parts.

Modularity and Reuse:
The rise of data abstraction and object-oriented software has increased the emphasis on modularity and reuse, making the testing of software design elements more crucial, particularly during integration testing.

Structural Graph Coverage:
Testing often begins with creating graphs based on couplings between software components, as shown in Figure 7.26.

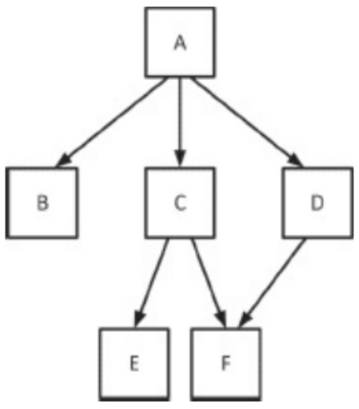


Figure 7.26. A simple call graph.

Call Graphs:
The most common graph used for structural design coverage is the call graph. In a call graph, the nodes represent methods (or units) and the edges represent method calls

- **Node Coverage (Method Coverage)**: Each method must be called at least once.
- **Edge Coverage (Call Coverage)**: Each call must be executed at least once, requiring some methods to be called multiple times, e.g., Method F in Figure 7.26 must be called from both C and D.

Application to Modules:
For modules (e.g., Java classes), not all units may call each other, leading to disconnected call graphs or no calls at all, making module testing with this technique inappropriate. Sequence-based techniques may be required.

Inheritance and Polymorphism:
OO features like inheritance and polymorphism introduce testing challenges.

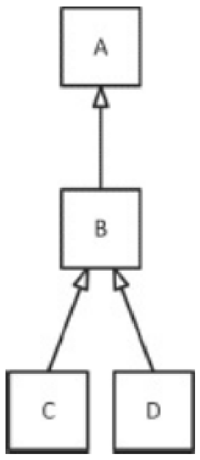


Fig 7.27

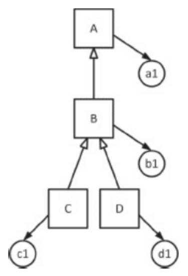


Figure 7.28. An inheritance hierarchy with objects instantiated.

- **Inheritance Graph**: Represented in Figure 7.27, where classes inherit from one another.
- **Object Instantiation**: For coverage, objects need to be instantiated (Figure 7.28).
- **OO Call Coverage Criterion**: Requires Call Coverage for at least one object of each class.
- **All Object Call Criterion**: Extends the criterion to all instantiated objects of each class.

Data Flow Graph Coverage for Design Elements

Control connections among design elements are straightforward, making them less effective for fault detection compared to data flow connections, which are complex and harder to analyze. This complexity suggests they are rich sources of software faults. Key concepts include:

- **Caller**: Unit invoking another unit (callee).
- **Call site**: Statement making the call.
- **Actual parameter**: In the caller, assigned to a formal parameter in the callee.
- **Call interface**: Mapping of actual to formal parameters.

The data flow testing criteria focus on ensuring that variables defined in the caller are used correctly in the callee. This involves examining the last definitions of variables before calls and returns and the first uses of variables after calls and returns (Figure 7.29).

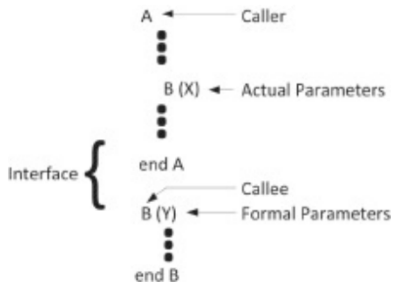


Figure 7.29. An example of parameter coupling.

Types of Data Flow Coupling:

1. **Parameter coupling**: Parameters passed in calls.

2. **Shared data coupling**: Accessing the same global or non-local variable.
3. **External device coupling**: Accessing the same external medium, like a file.

Key Definitions:

- **Last-def**: Nodes defining a variable with a def-clear path through the call site to a use in the other unit.
- **First-use**: Nodes with uses of a variable having a def-clear and use-clear path from the entry point (callee) or call site (caller).

Inter-procedural DU pairs:
Definition-use pairs that span across different program units or procedures. They involve:

- **Definition (def)**: Where a variable is assigned a value.
- **Use (use)**: Where the variable's value is utilized.
  These pairs ensure proper data flow and detect faults between a caller unit and a callee unit during integration testing. Figures 7.30 is Inter-procedural du pair coupling

Coupling du-pair:
A **coupling du-pair** consists of a variable's definition (def) in one unit and its use (use) in another unit. This is key in identifying potential faults in data flow between units. Figures 7.30 and 7.31 illustrate examples of parameter coupling and the concepts of last-defs and first-uses.

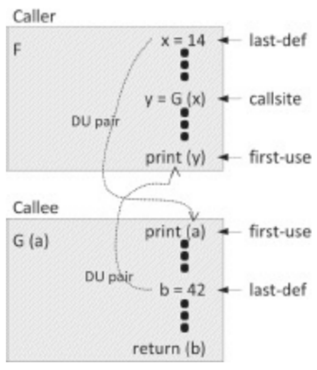


Figure 7.30. Coupling du-pairs.

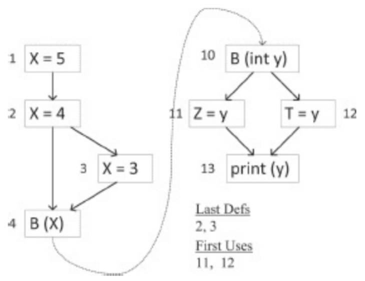


Figure 7.31. Last-defs and first-uses.

Coupling Coverage Criteria:

- **All-Coupling-Def coverage**: Path executed from every last-def to at least one first-use.
- **All-Coupling-Use coverage**: Path executed from every last-def to every first-use.

- **All-Coupling-du-Paths coverage**: Every simple path from every last-def to every first-use is executed.

Example for Coupling DU Pairs

Caller (main program):

```python
def main():
    x = 10  # Definition of x at line 2
    result = square(x)  # Call to callee with x as the actual parameter at line 3
    print(result)  # Use of result at line 4

def square(n):  # Callee function
    return n * n  # Use of parameter n at line 8

main()
```

Coupling DU Pairs

1. **(main, x, 2) -> (square, n, 8)**:
   - Definition of `x` at line 2 in `main` is used as `n` in `square` at line 8.
2. **(square, n * n, 8) -> (main, result, 4)**:
   - Result of `n * n` in `square` at line 8 is used as `result` in `main` at line 4.