# Control and Data Flow

Shin Yoo

**Some examples are borrowed/inherited from Prof. Mark Harman, Dr. Kiran Lakhotia, and Dr. Gregory Gay :)**

# Overview

- Control Flow and Control Flow Graph (CFG)
- Data-Flow Analysis

# Control Flow

- The order in which the individual structural elements of program are executed or evaluated

- Structural element: statements, instructions, function calls, etc…

- We all construct a control flow when we try to execute a source code in our mind

# Control Flow Statements

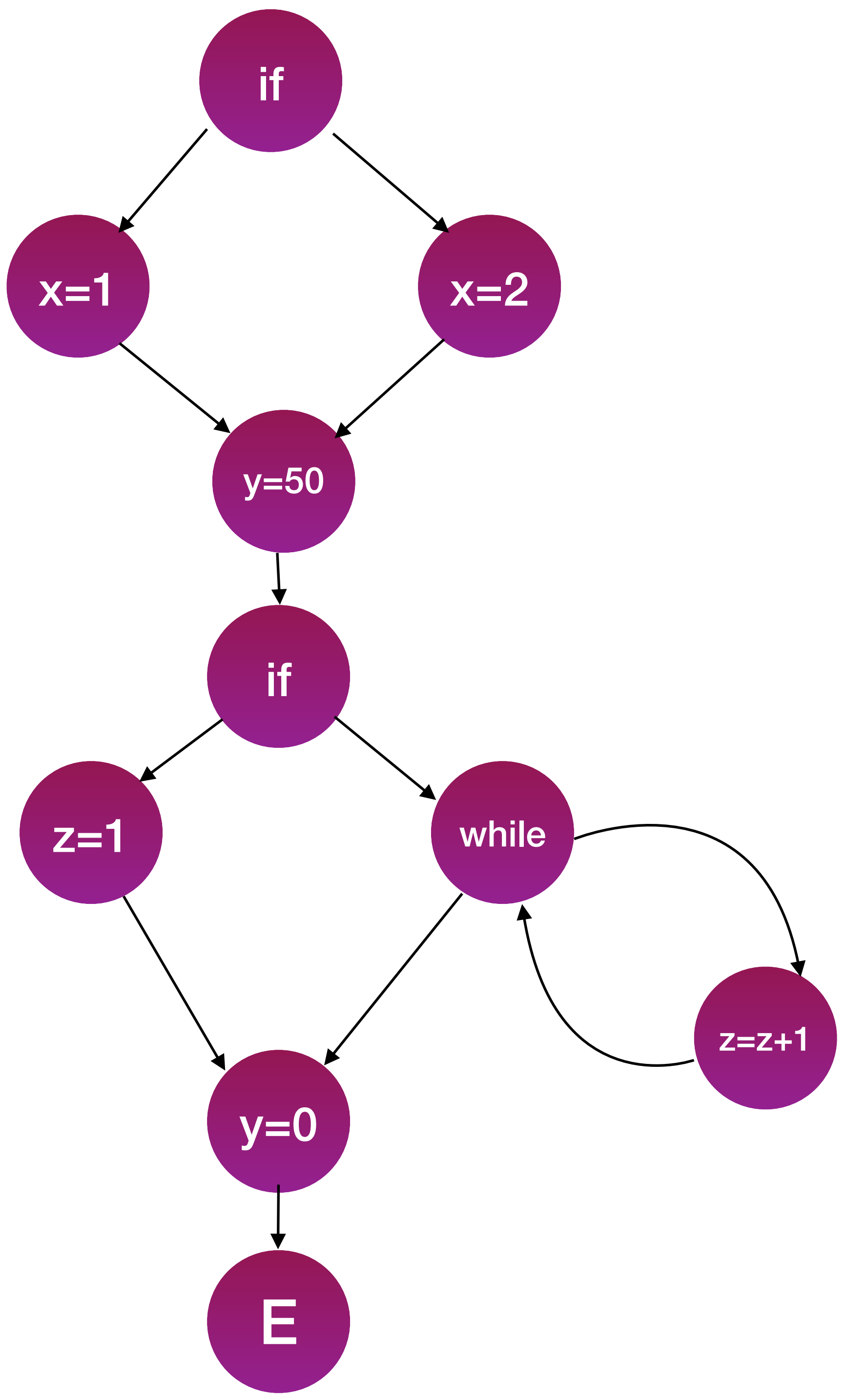
- Statements whose execution results in a choice between more than one execution paths
  - Continuation in a different location (unconditional branching)
  - Executing a set of statements only when certain conditions are met (conditional branching)
  - Executing a set of statements zero or more times until certain conditions are met (looping)
  - Executing a set of remote statements, then return the flow of control to the current position (function calls)
  - Stopping the program (unconditional halting)

# Control Flow Graph

- Graph representation of programs
  - Nodes are statements
  - Edges are all possible flow of execution
  - We assume an explicit end node
- You are expected to be able to draw one from code

```
if(...) x=1;
else x=2;
y = 50;
if(...) z = 1;
else
{
  while(...)
    z = z + 1;
}
y = 0;
```

```
if(...) x=1;
else x=2;
y = 50;
if(...) z = 1;
else
{
  while(...)
    z = z + 1;
}
y = 0;
```



# A million dollar S/W testing question

- When should (can) I stop testing?
- A perfect, yet infeasible answer: when you have executed all test cases.
- An equally perfect and infeasible answer: when you have caught all faults.

# Reformulated Question

- If we cannot know when to end, can we at least know the relative, surrogate benefit of each additional test execution?
    - Surrogate, because we can never precisely measure the actual fault detection capability *a priori*.
- A reliable surrogate measure would be very useful:
    - We can compare two test cases to decide which one to use.
    - Depending on the nature of the measure, we can mark a (incomplete yet practically necessary) stopping point.

What is it that you can measure from testing and is correlated with fault detection capability?

# Structural Code Coverage

- A necessary, but not sufficient condition for fault detection
  - With testing, you cannot detect faults in a line that you haven’t executed during testing (testing is dynamic).
- REMEMBER: coverage DOES NOT guarantee anything.

> I expect a high level of coverage. Sometimes managers require one. There's a subtle difference.

**“What’s a good code coverage to have?” Harm Pauw**
**https://www.scrum.org/resources/blog/whats-good-code-coverage-have**

# Weyuker Hypothesis

"The adequacy of a coverage criterion can only be intuitively defined."

# Dangers of Coverage

- Coverage can become a goal in its own: writing test only to increase coverage.
- Achieving 100% coverage can still detect no fault whatsoever.
- Coverage metric can tell you what is not being tested, but cannot precisely tell you what is actually being tested.

# Benefits of Coverage

- An accurate measure of what is not being tested.
- Testing everything a little bit is better than not testing most of the program.
- Not all coverage criteria are the same: a stricter coverage criterion leads to more tests, and more tests are more likely to lead to fault detection.

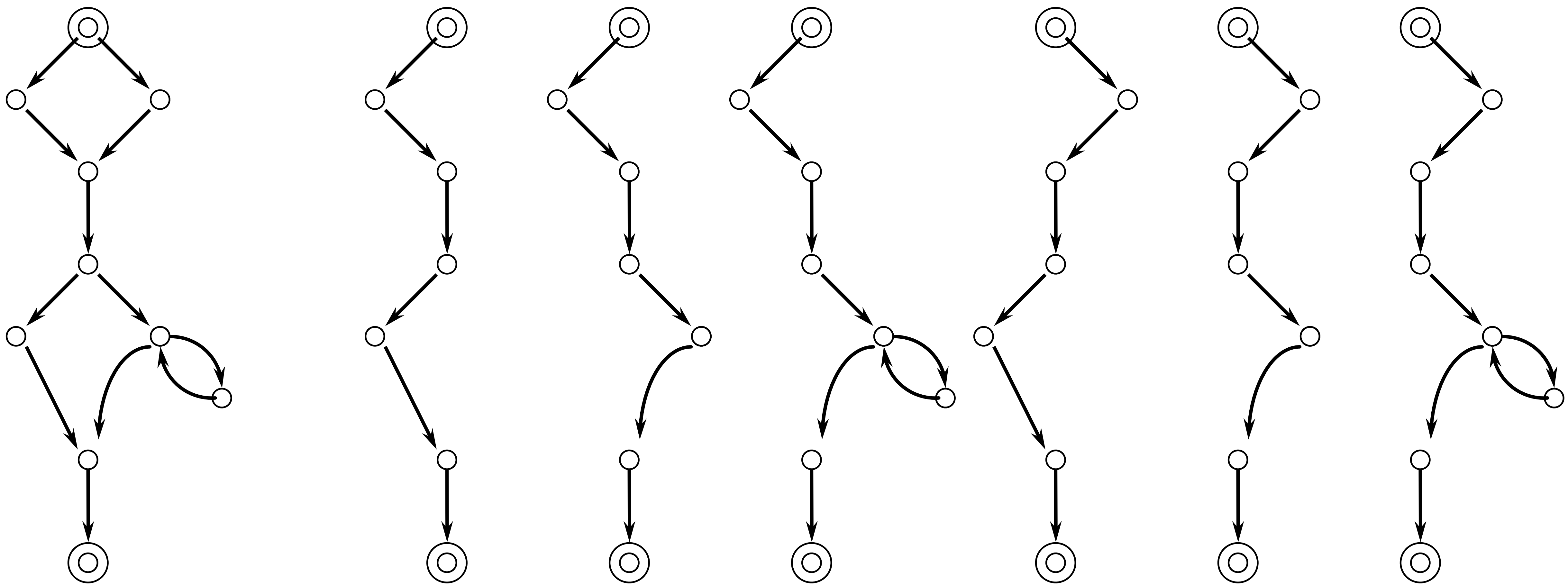
# The most widely used: Statement/Branch Coverage

- Statement coverage: % of nodes in CFG that are executed by your testing
- Branch coverage: % of branching edges in CFG that are executed by your testing
- 100% may not always be possible

```
if a>b then
   if b>c then
      if a>c then S1
             else S2
```

# Simple Path

- A simple path in a CFG is one in which no edge is traversed more than once

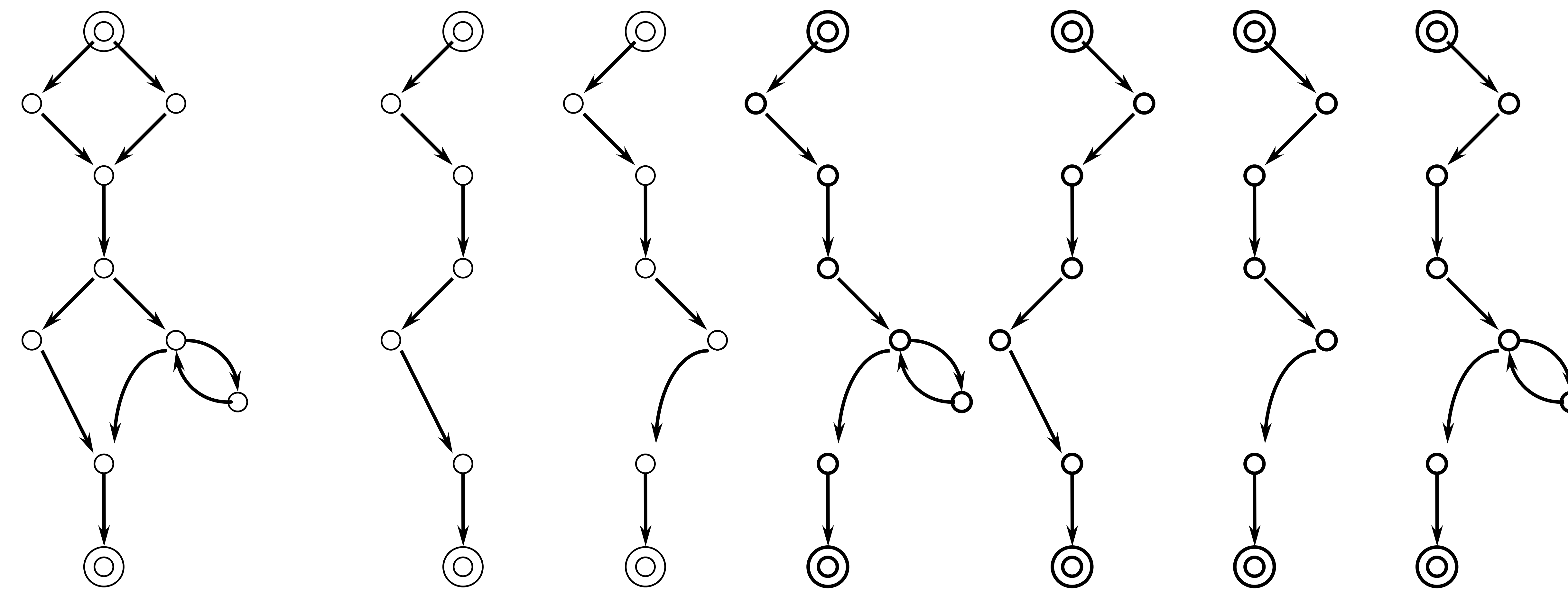


# All Paths Testing

- Execute all possible paths in code
- In general, you get unbounded number of tests because of:
  - Loops! In general, loops makes everything about program analysis more complicated and annoying.
- If you set the maximum number of iterations for each loop to k, you can bound the number of tests
  - For example, All Paths with k=2 requires you to achieve all paths coverage, repeating loops 0, 1, and 2 times
  - Setting k=1 results in simple paths

# All Paths Testing (k = 1)

- All Paths for our example code requires 6 tests:



**Why happens for k=2? How many test cases?**

# All Paths Testing

- Loop bound still needs to be relatively low - why?

# How many paths (k=20)?

```
int flipSome(int A[], int N, int X)
{
    int i=0;
    while (i<N and A[i] <X)
    {
       if (A[i]<0)
           A[i] = - A[i];
       i++;
    }
    return(1);
}
```

**Loop combined with branches will result in exponential number of paths. In this case, how many? :)**

$$2^0 + 2^1 + \ldots + 2^{20} = 2^{21} - 1$$

# All Paths Testing

- Loop bound still needs to be relatively low - why?
- What is the number of paths you get out of n consecutive loops with bound k?
    - $(k+1)^n$: it blows up exponentially, and gets worse with nested loops.

# Decision Coverage

```
if(x && (y || z))…
```

| x | y | z |
|---|---|---|
| TRUE | TRUE | TRUE |
| FALSE | TRUE | TRUE |

**The entire predicate `(x && (y || z))` should be evaluated to both true and false. The above test suite is decision adequate.**

# Other Types of Coverage

- Function Coverage: Has every function been called?
- Entry/Exit Coverage: Has every possible call and return of functions been executed?
- Decision Coverage: Entry/Exit + Branch Coverage
- Condition Coverage: Has each Boolean subexpression been evaluated to be both true and false?
- Condition/Decision Coverage: Entry/Exit + Branch + Condition Coverage
- Modified Condition/Decision Coverage: Condition/Decision Coverage plus “does each boolean subexpression actually affect the outcome of the decision?”

# Condition Coverage

```
if(x && (y || z))…
```

| x | y | z |
|---|---|---|
| TRUE | TRUE | TRUE |
| FALSE | TRUE | TRUE |

**(X)**

**Condition coverage requires each Boolean subexpression to be evaluated both true and false. Previous test suite is NOT condition adequate.**

| x | y | z |
|---|---|---|
| TRUE | TRUE | TRUE |
| FALSE | FALSE | FALSE |

**(O)**

**This is condition adequate.**

# Modified Condition/ Decision Coverage

```
if(x && (y || z))…
```

| No. | x | y | z |
|---|---|---|---|
| 1 | TRUE | FALSE | FALSE |
| 2 | TRUE | FALSE | TRUE |
| 3 | FALSE | FALSE | TRUE |
| 4 | TRUE | TRUE | FALSE |

**MC/DC requires each Boolean subexpression to be both true and false, and this to affect the final decision.**

- **All x, y, and z have been assigned both true and false.**
- **Between 1 and 4, we see that y can affect the final decision.**
- **Between 1 and 2, we see that z can affect the final decision.**
- **Between 2 and 3, we see that x can affect the final decision.**

# Condition/Decision vs. MC/DC

- MC/DC is used in:

  - Avionics Software Development Guideline: DO-178B and DO-178C, *de facto* standard set by FAA for Level A systems (those that either provide or prevent failures in safe flight and landing).

  - General electrical devices: SIL (Safety Integrity Level) 4 in IEC 61508-3 Standards

  - Automotive Testing Standard: highly recommended for ASIL (Automotive Safety Integrity Level) D in ISO 26262 Standards.

# What about data usage?

- Detecting specific values that may lead us to failures would be hard: it requires careful analysis of both the expected semantic and the implementation.

- Structural coverage is mostly about control flow (CFG).

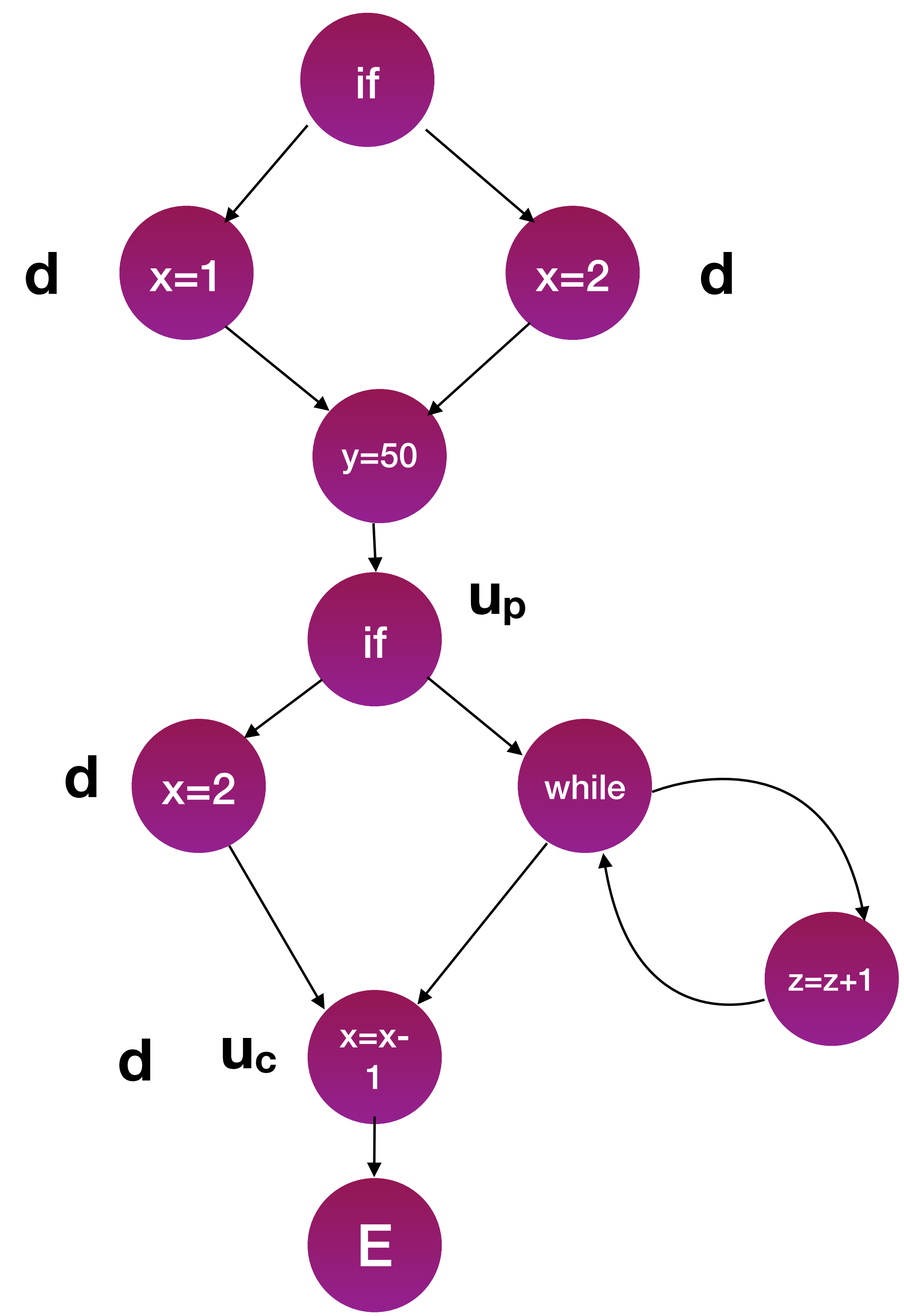
- Dataflow analysis is about the usage of variable values.

# Data Flow Analysis

- CFGs do not take how variables are used into consideration
- Data-flow based testing analyses the definition and use of data during execution
- We use CFG as a starting point, but annotate it with respect to usage of a specific variable
    - d: the value of the variable is defined
    - $u_p$: the variable is used in a predicate
    - $u_c$: the variable is used for calculation
    - k: killed (undefined or memory released)

**For variable x:**

```
if(...) x=1;
else x=2;
y = 50;
if(x%2 == 0) x = 2;
else
{
  while(...)
    z = z + 1;
}
x = x - 1;
```



# Data Flow Patterns

- There is no fixed rule that always works, but for example:
- dd : harmless but suspicious
- dk : harmless but suspicious
- du : normal
- kd : potentially suspicious
- kk : suspicious
- ku : a bug
- ud : potentially suspicious if u happens before d
- uk : normal
- uu : normal

# Some Data Flow Strategies

- All DU paths
- All Use paths
- APU+C: All predicate uses + some computations
- ACU+P: All computational uses + some predicates
- All definitions
- All predicate uses
- All computational uses

# DU paths

- A path from node x to node y is *definition clear* for a variable v iff for all nodes apart from x and y on the path, there is no assignment to v.

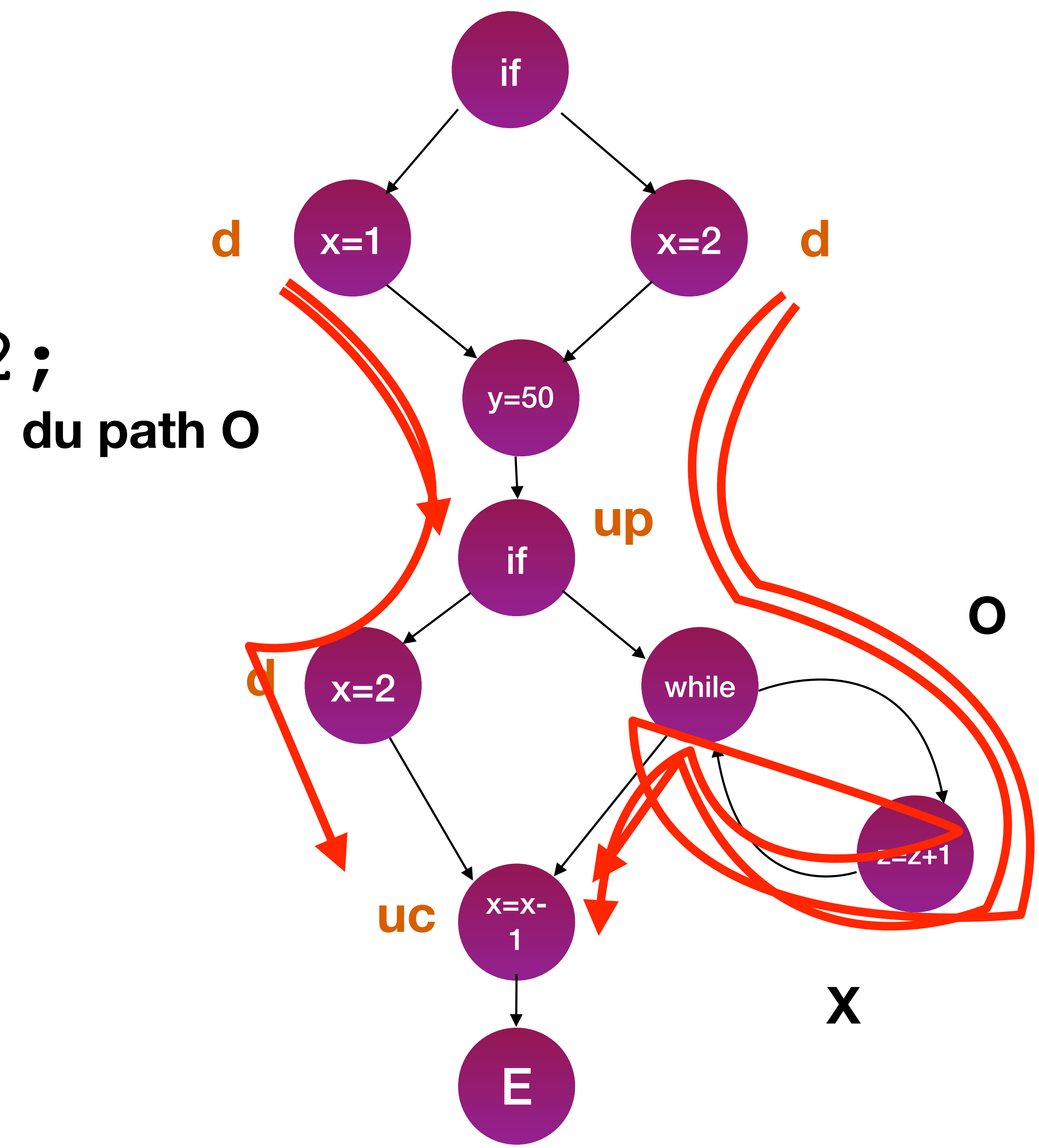
- A **du-path** from node x to node y for a variable v is a simple path from x to y which is definition clear for v and which assigns to v at x and uses v at y

  - Definition clear means we don't redefine the variable along the way

  - Simple path means no edge is traversed more than once

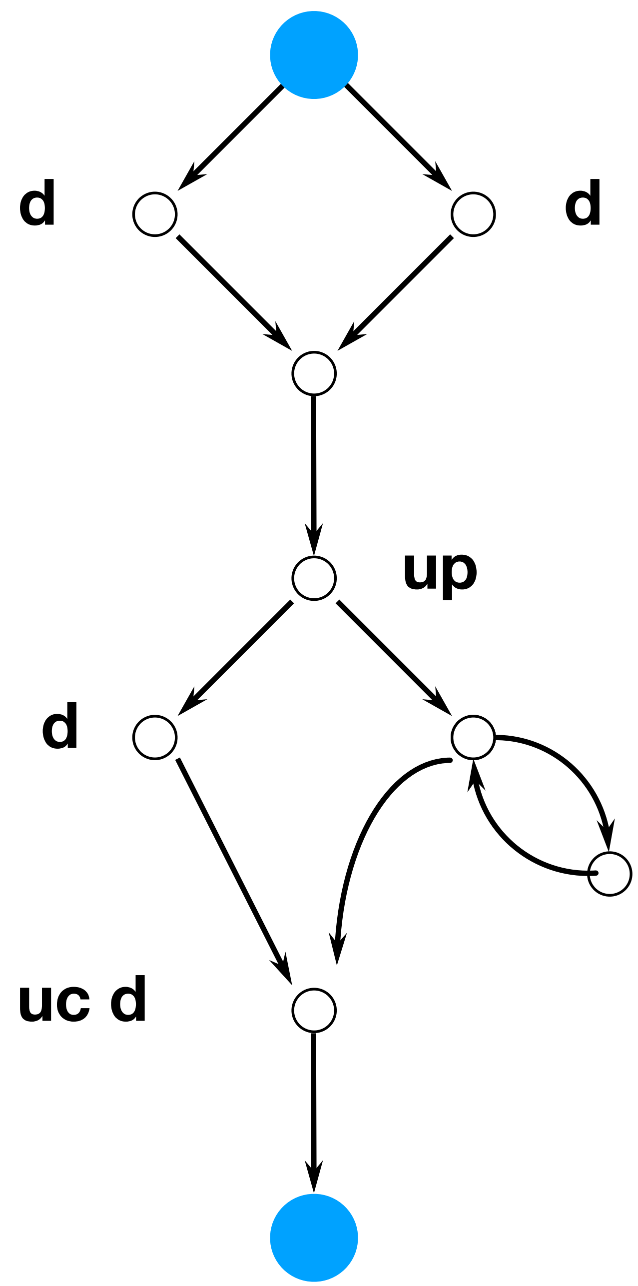
For variable x:

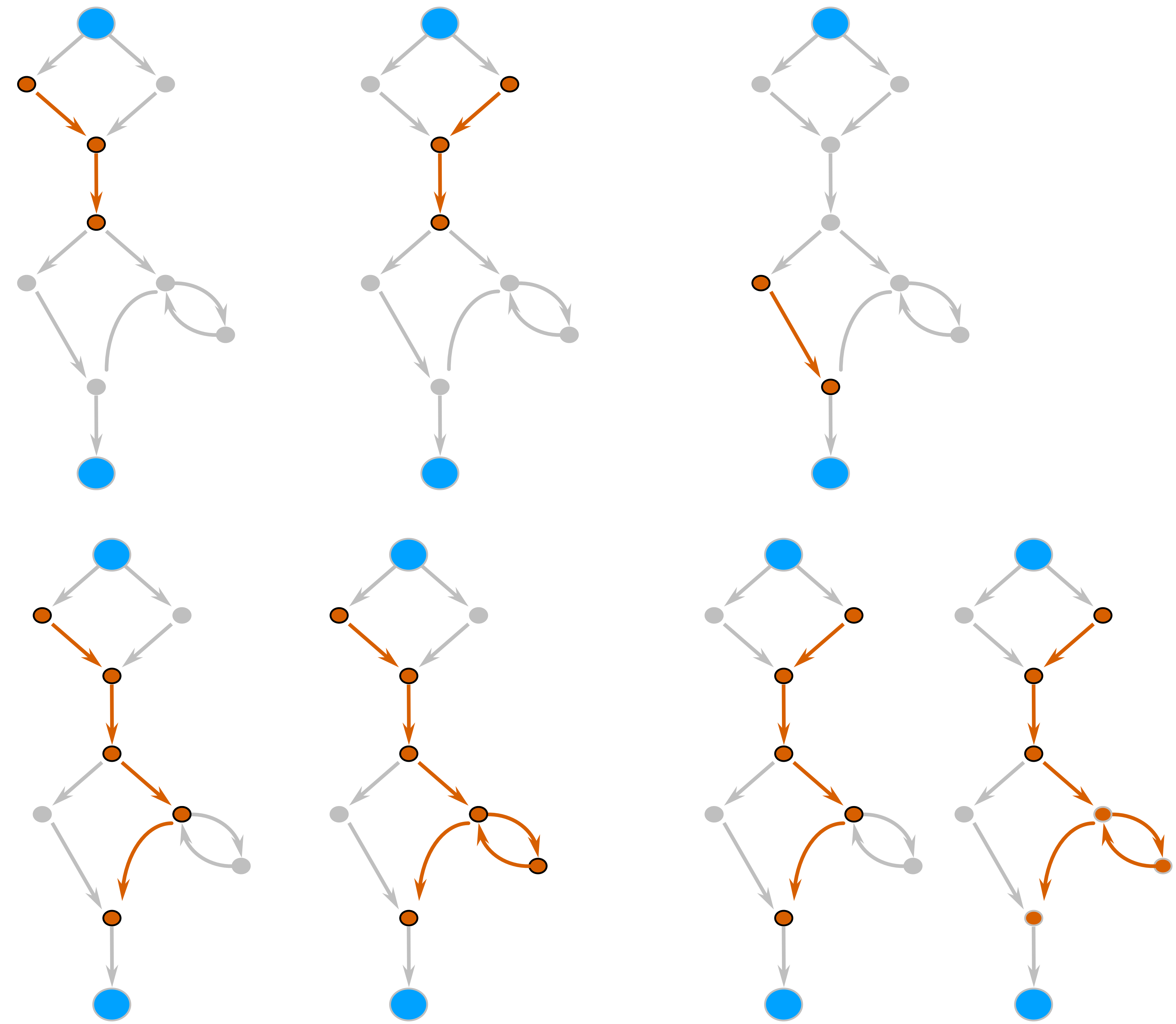
```
if(...) x=1;
else x=2;
y = 50;
if(x%2 == 0) x = 2;
else
{
  while(...)
    z = z + 1;
}
x = x - 1;
```



# All DU Path Testing

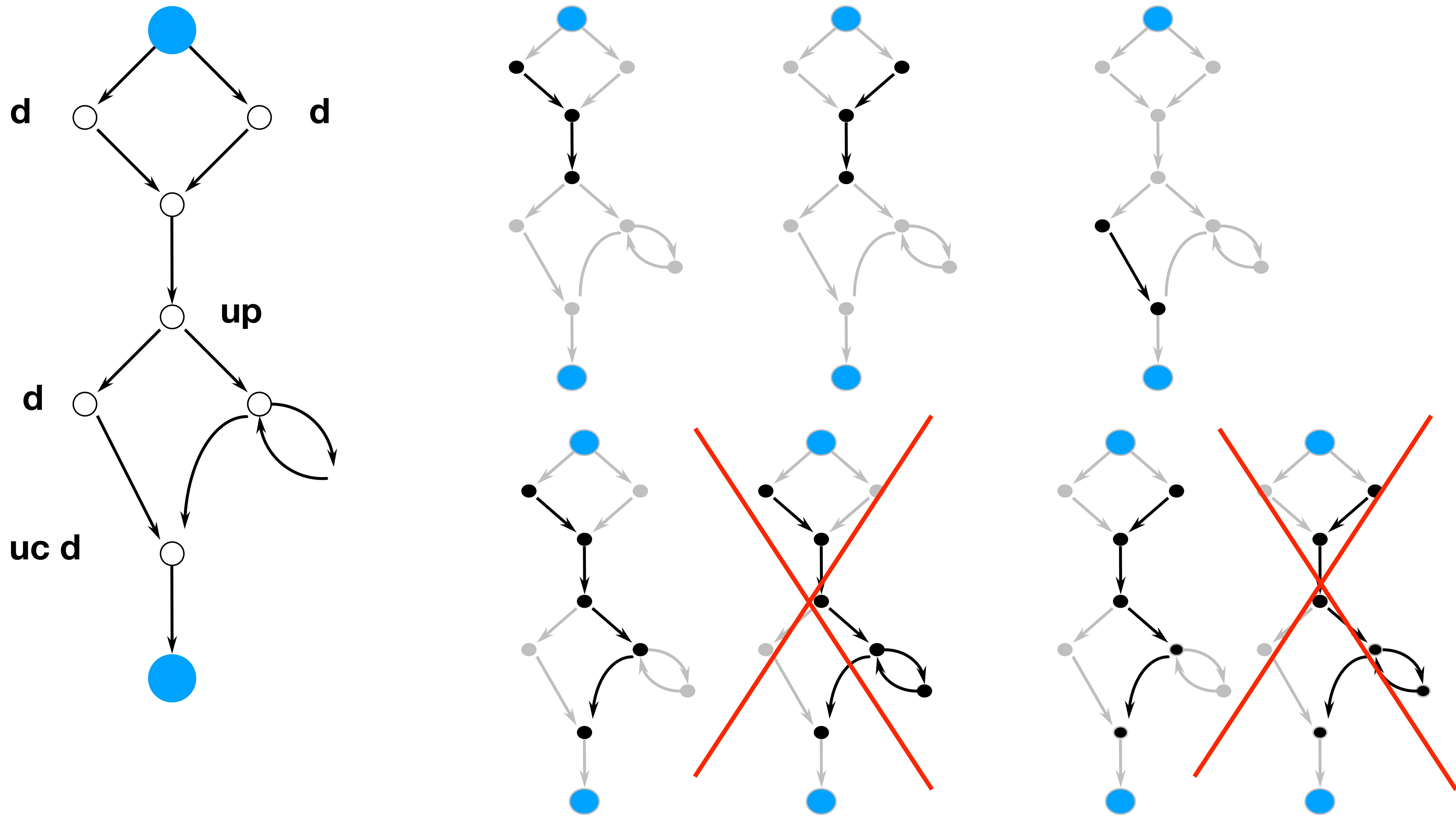
- For every variable v,
  - For every definition d of v,
    - for every use u of d,
      - for every du path between d and u, there is a test that executes the du path


d
d
up
d
uc d

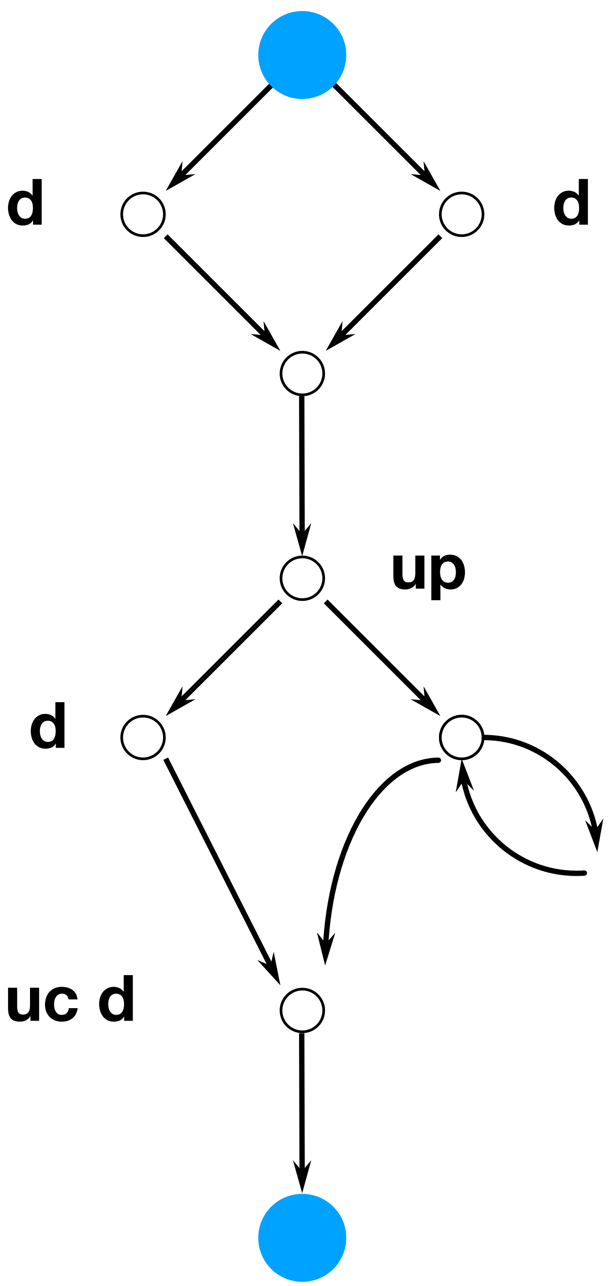


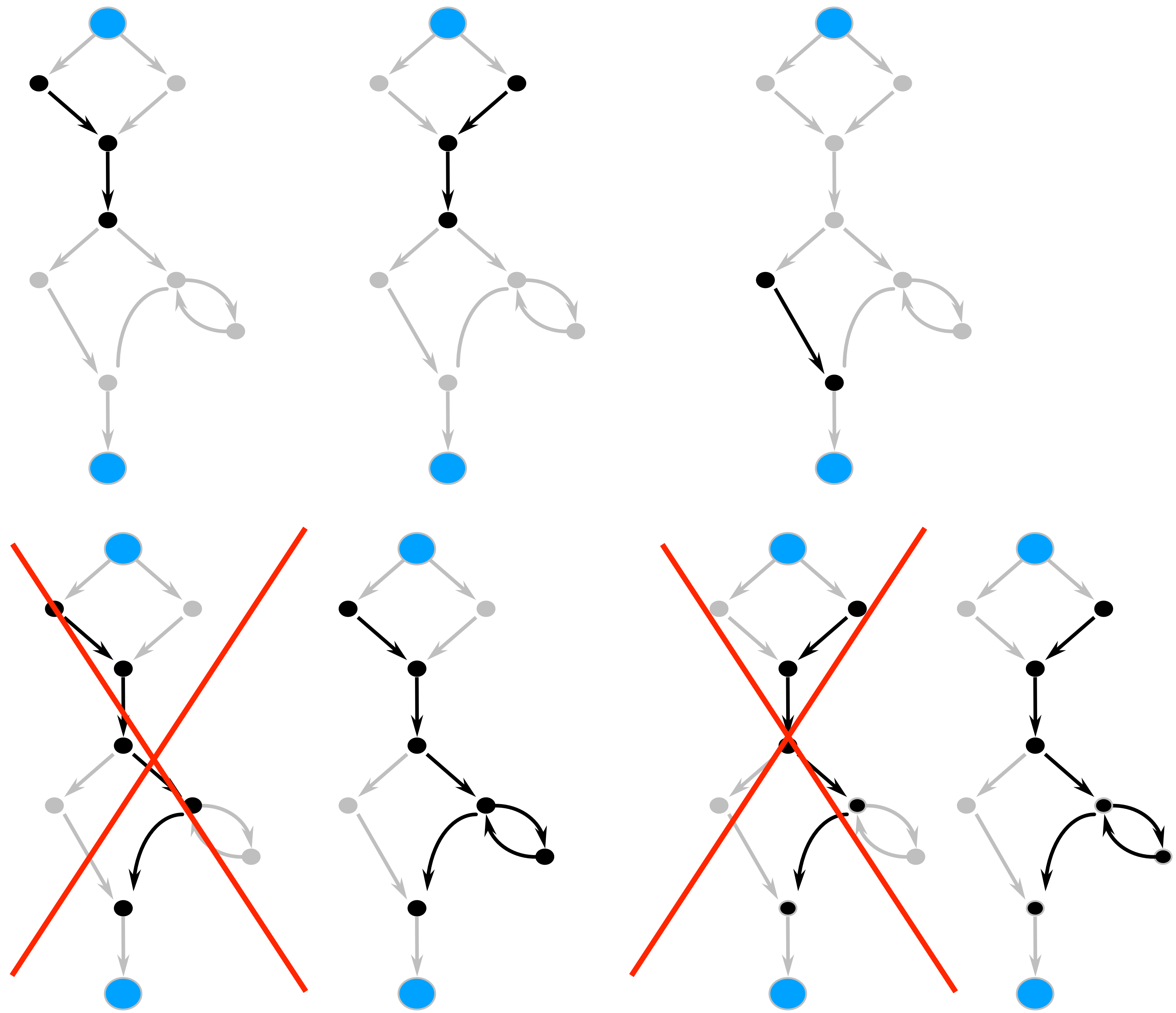
# All Use strategy - AU

- Same as all du paths except we only require at least one path from each definition to each use
- For every variable and
- for every definition, d, of that variable and
- for every use, u, of d and
- for at least one du-path from d to u
- there is test which exercises that path.

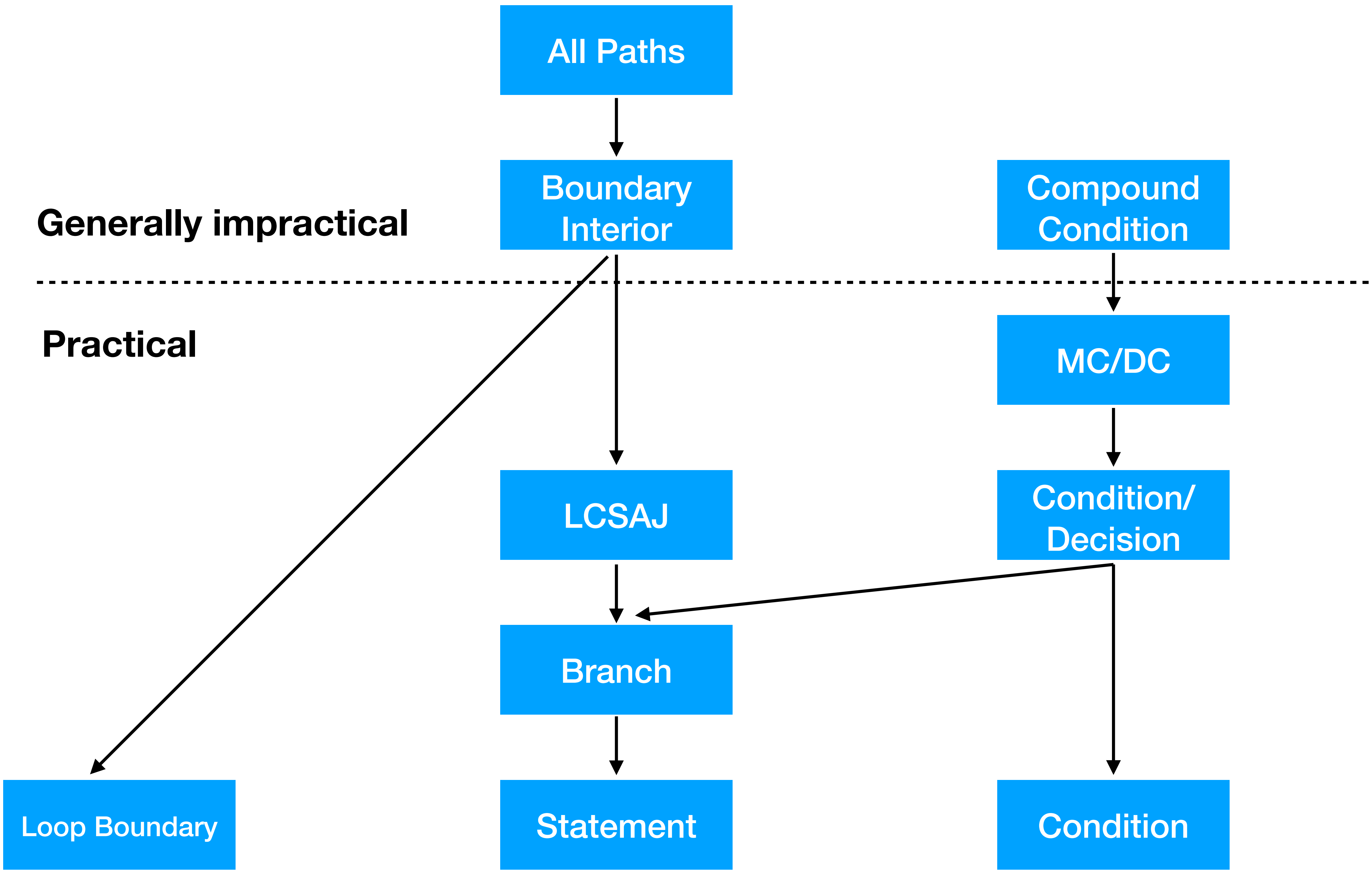

d
d
up
d
uc d

## Note that we have a choice



# Coverage Hierarchy



# Measuring Coverage

- Coverage Instrumentation: inserting additional code into the target program so that, when executed, you can collect information about which parts were reached.
    - Usually done at binary or byte code level.
- Or use one of the existing tools.

# Coverage Tools

- C: GNU gcov profiler (of the kcov fame) (https://gcc.gnu.org/onlinedocs/gcc/Gcov.html)
- Java: Jacoco (http://www.eclemma.org/jacoco/) and Cobertura (http://cobertura.github.io/cobertura/) are both popular
- Python: coverage.py (https://coverage.readthedocs.io/en/coverage-4.5.1/)
- JavaScript: JSCover (https://tntim96.github.io/JSCover/)

# Summary

- Control Flow is the order of things being evaluated/ executed
    - It can be reprinted as a CFG, a directed graph
- Data Flow tracks where values are assigned and where they are used subsequently
    - Data flow information can be annotated over CFG