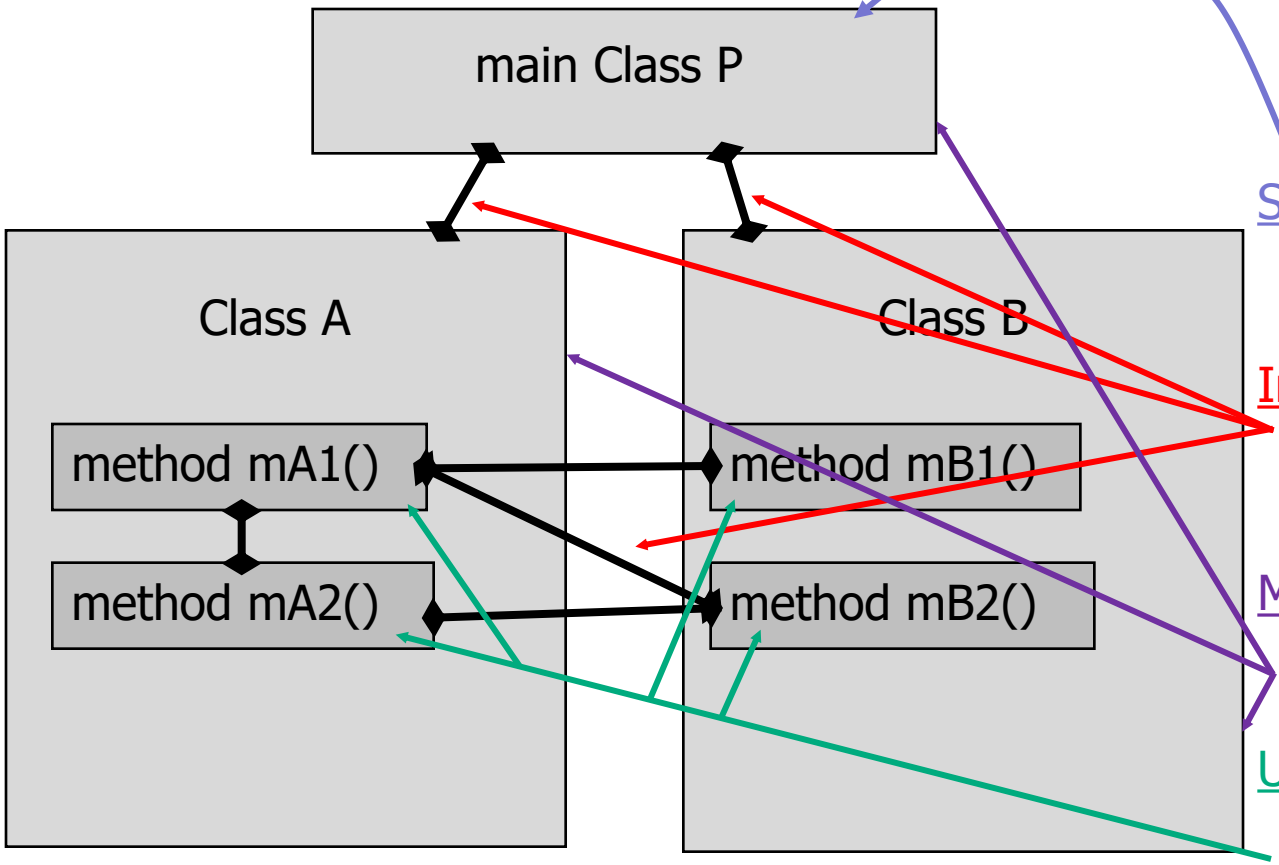
CS 575
Software Testing and Analysis

Testing Techniques

ÖZYEĞİN UNIVERSITY

(c) Slides adopted from the slides P. Amman & J. Offut (Chapter 1)

Testing at Different Levels



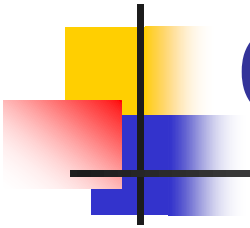
Acceptance testing: Is the software acceptable to the user?

System testing: Test the overall functionality of the system

Integration testing: Test how modules interact with each other

Module testing: Test each class, file, module or component

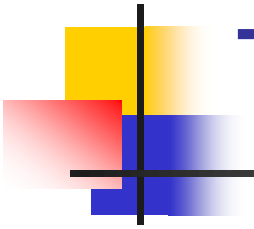
Unit testing: Test each unit (method) individually



Changing Notions of Testing

- Traditional categorization is based on **phases** of software development
  - Unit, module, integration, system, etc. …
- Another categorization can be in terms of **structures** and **criteria**
  - Graphs, logical expressions, syntax, input space, etc.
- Test design is largely the same at each phase
  - Creating the **model** is different
  - Choosing **values** and **automating** the tests is different

Test Requirements & Criteria



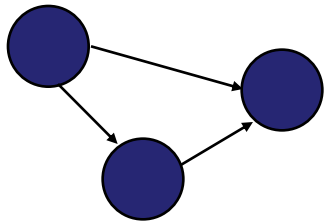
- Test Requirements: must be satisfied or covered
- Test Criterion: A collection of rules and a process that define test requirements

- The Usual Case: Define a model of the software, then find ways to cover it

Testing researchers have defined dozens of criteria, but they are all really just a few criteria on **four types of structures** …

Criteria based on Structures

- Graphs



- Logical Expressions

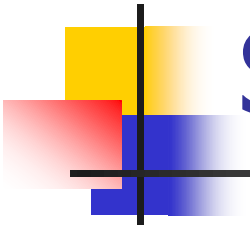
```
(not X or not Y) and A and B
```

- Input Domain Characterization

```
A: {0, 1, >1}
B: {600, 700, 800}
C: {swe, cs, isa, infs}
```

- Syntactic Structures

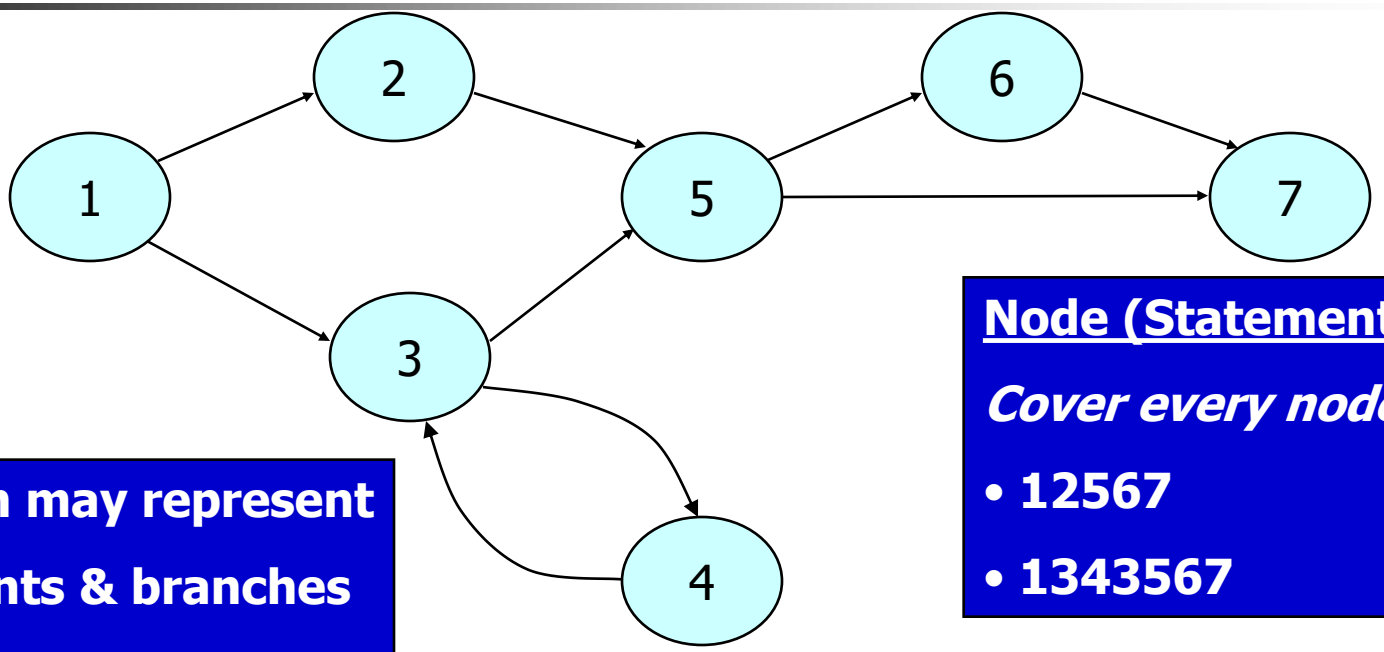
```
if (x > y)
    z = x - y;
else
   z = 2 * x;
```



Source of Structures

- These structures can be **extracted** from lots of software artifacts
  - **Graphs** can be extracted from UML use cases, finite state machines, source code, …
  - **Logical expressions** can be extracted from decisions in program source, guards on transitions, conditionals in use cases, …
- This is not the same as "***model-based testing***," which derives tests from a model that describes some aspects of the system under test
  - The model usually describes part of the **behavior**
  - The **source** is usually ***not*** considered a model

1. Graph Coverage:
Structural



This graph may represent

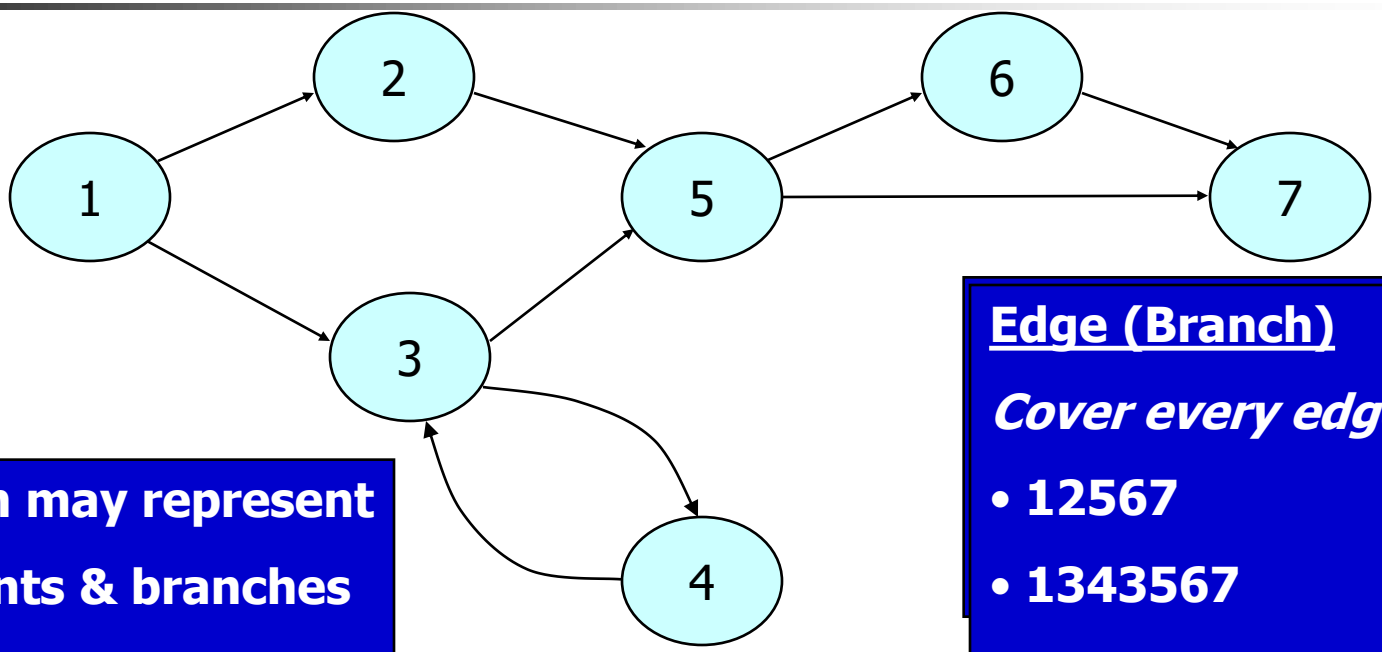
- **statements & branches**
- **methods & calls**
- **components & signals**
- **states and transitions**

Node (Statement)

Cover every node

- **12567**
- **1343567**

1. Graph Coverage:
Structural



This graph may represent

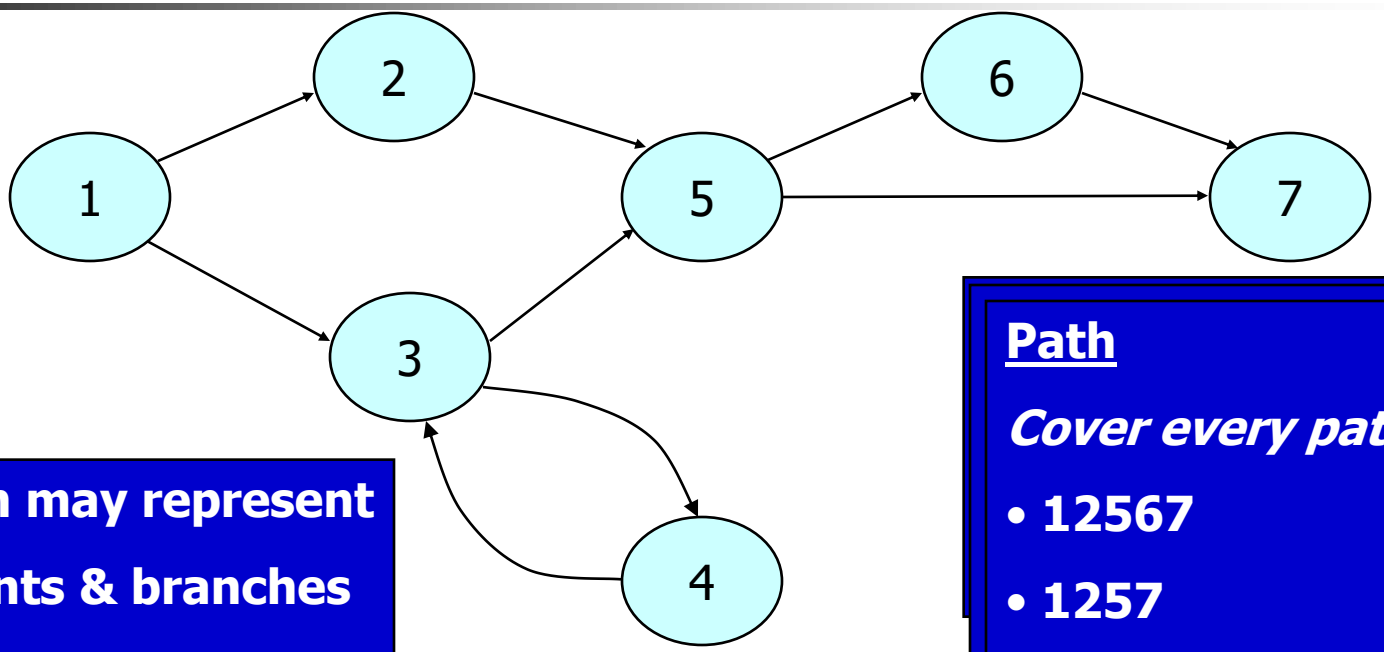
- **statements & branches**
- **methods & calls**
- **components & signals**
- **states and transitions**

Edge (Branch)

Cover every edge

- **12567**
- **1343567**
- **1357**

1. Graph Coverage: Structural



This graph may represent

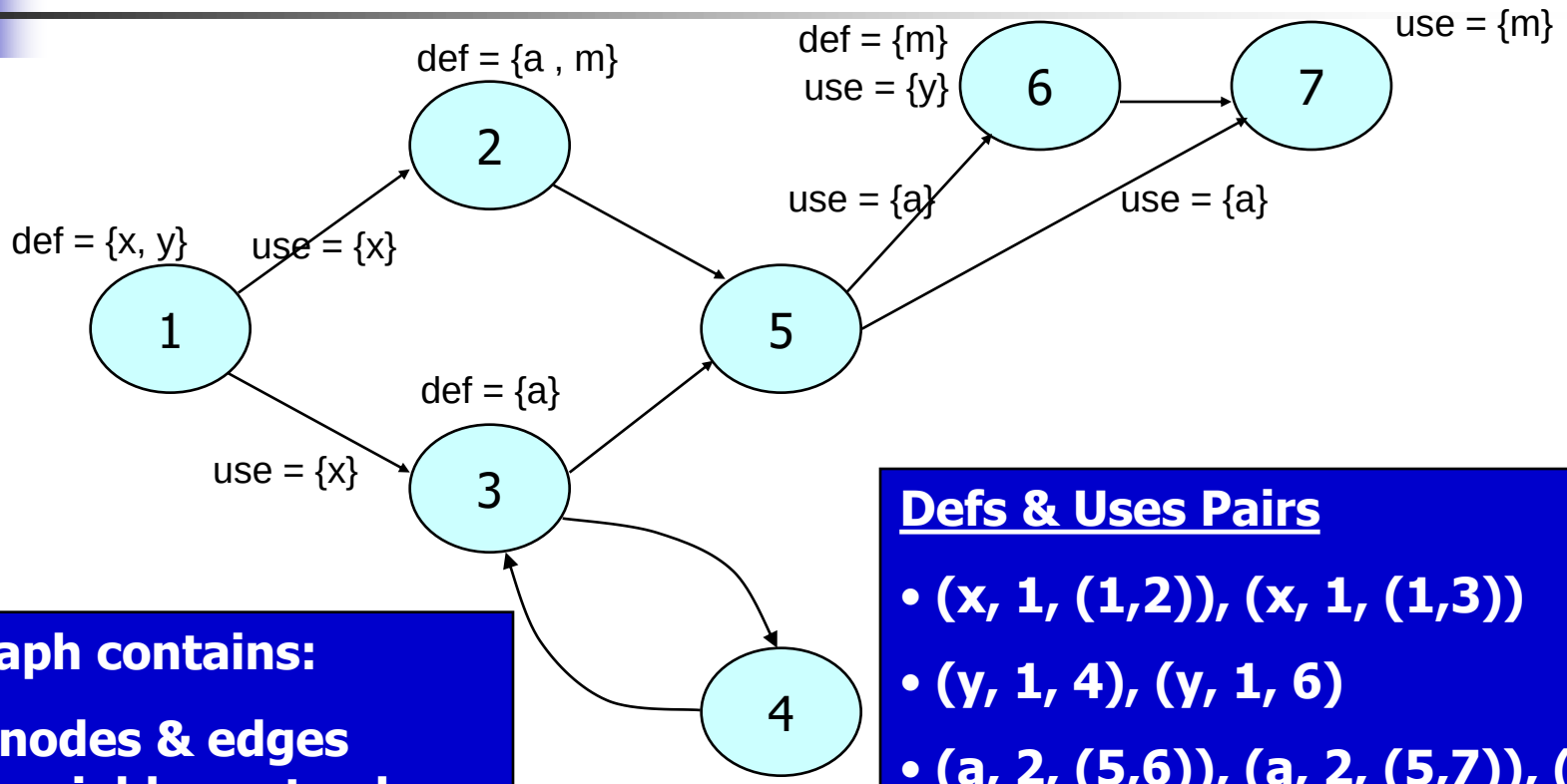
- **statements & branches**
- **methods & calls**
- **components & signals**
- **states and transitions**

Path

Cover every path

- **12567**
- **1257**
- **13567**
- **1357**
- **1343567**
- **134357 …**

1. Graph Coverage:
Data Flow



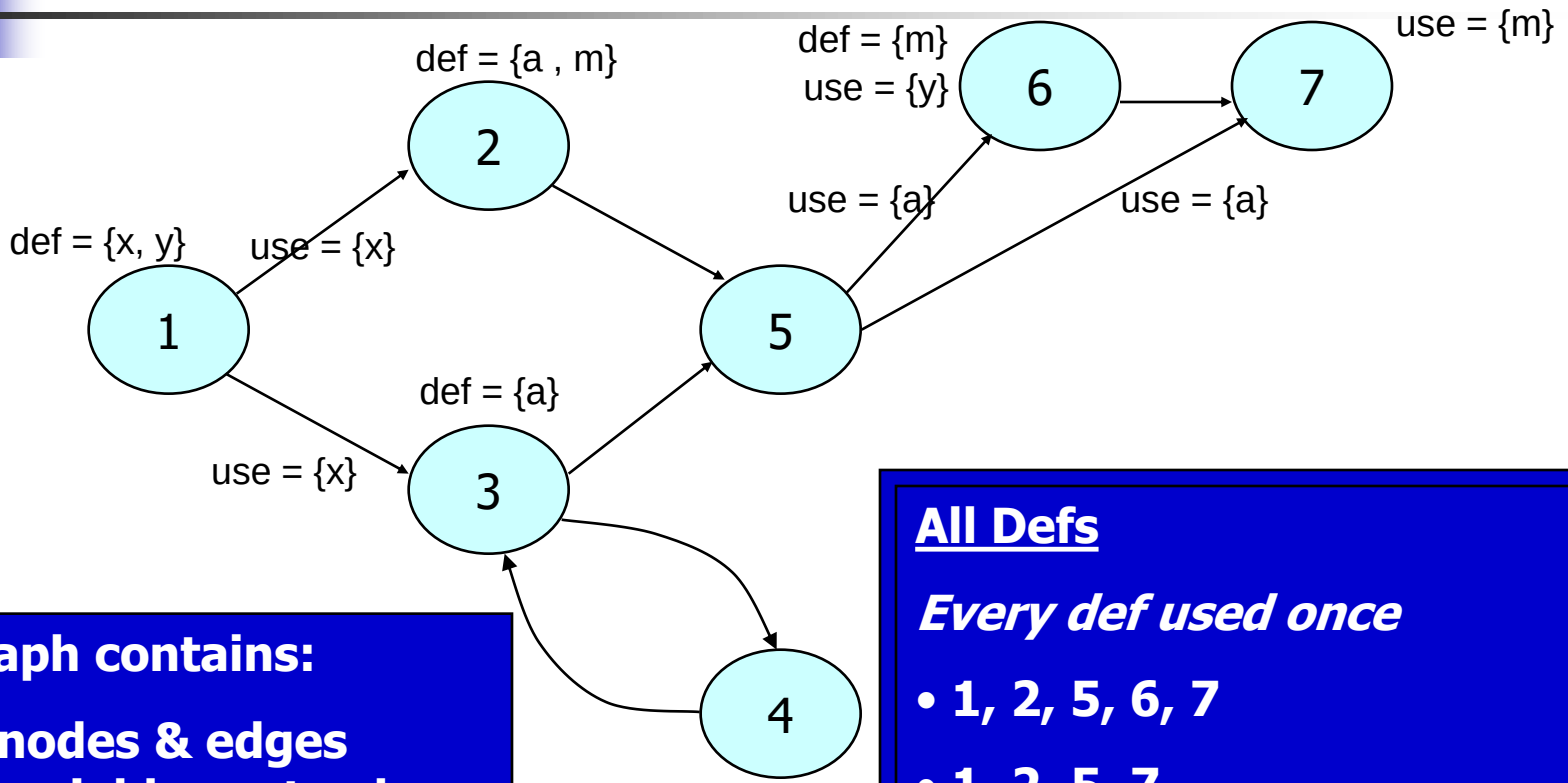
This graph contains:

- **defs: nodes & edges where variables get values**
- **uses: nodes & edges where values are accessed**

Defs & Uses Pairs

- **(x, 1, (1,2)), (x, 1, (1,3))**
- **(y, 1, 4), (y, 1, 6)**
- **(a, 2, (5,6)), (a, 2, (5,7)), (a, 3, (5,6)), (a, 3, (5,7)),**
- **(m, 2, 7), (m, 4, 7), (m, 6, 7)**

1. Graph Coverage:
Data Flow



This graph contains:

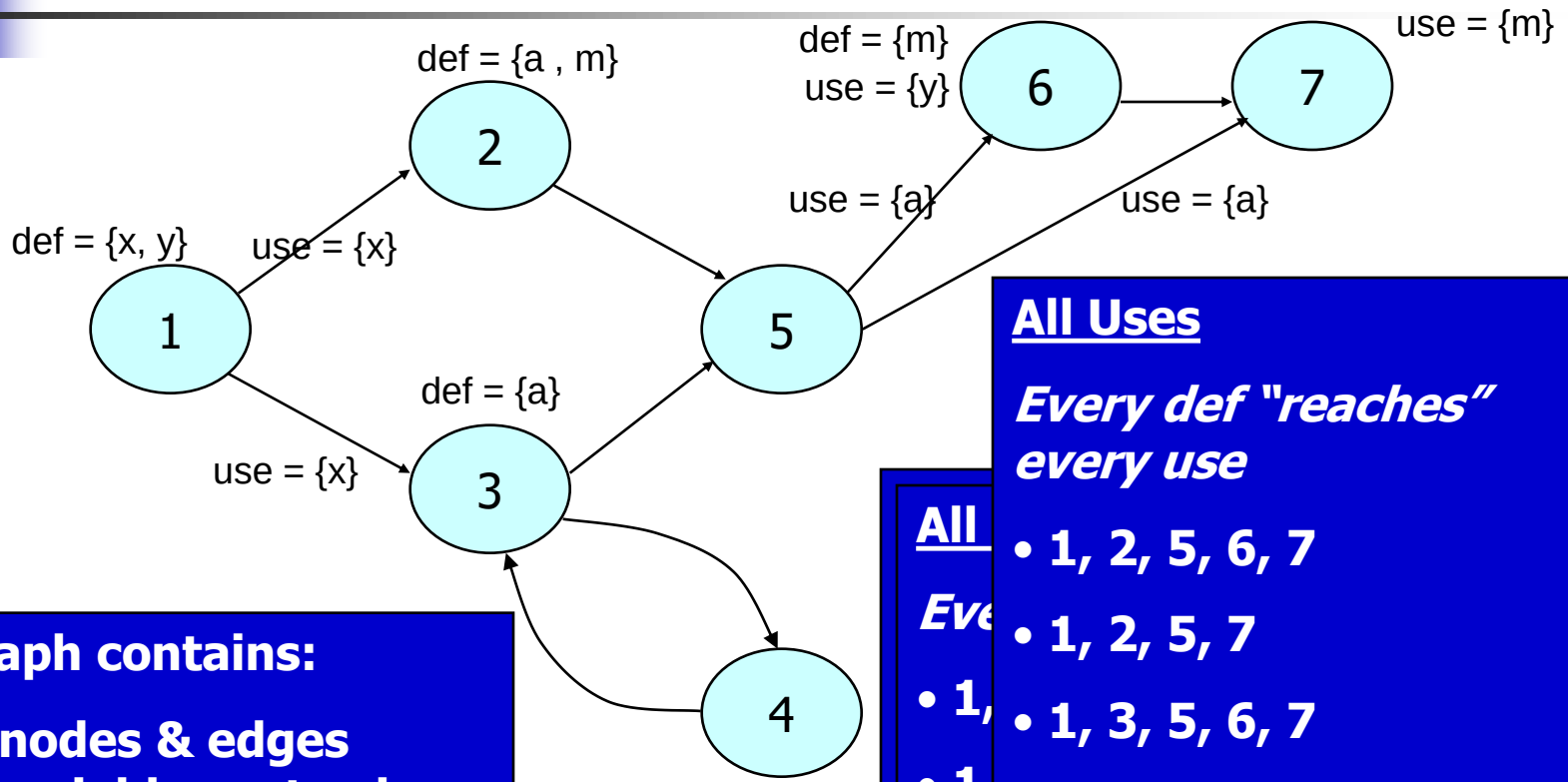
- **defs: nodes & edges where variables get values**
- **uses: nodes & edges where values are accessed**

All Defs

Every def used once

- **1, 2, 5, 6, 7**
- **1, 2, 5, 7**
- **1, 3, 4, 3, 5, 7**
- **(m, 2, 7), (m, 4, 7), (m, 6, 7)**

1. Graph Coverage:
Data Flow



This graph contains:

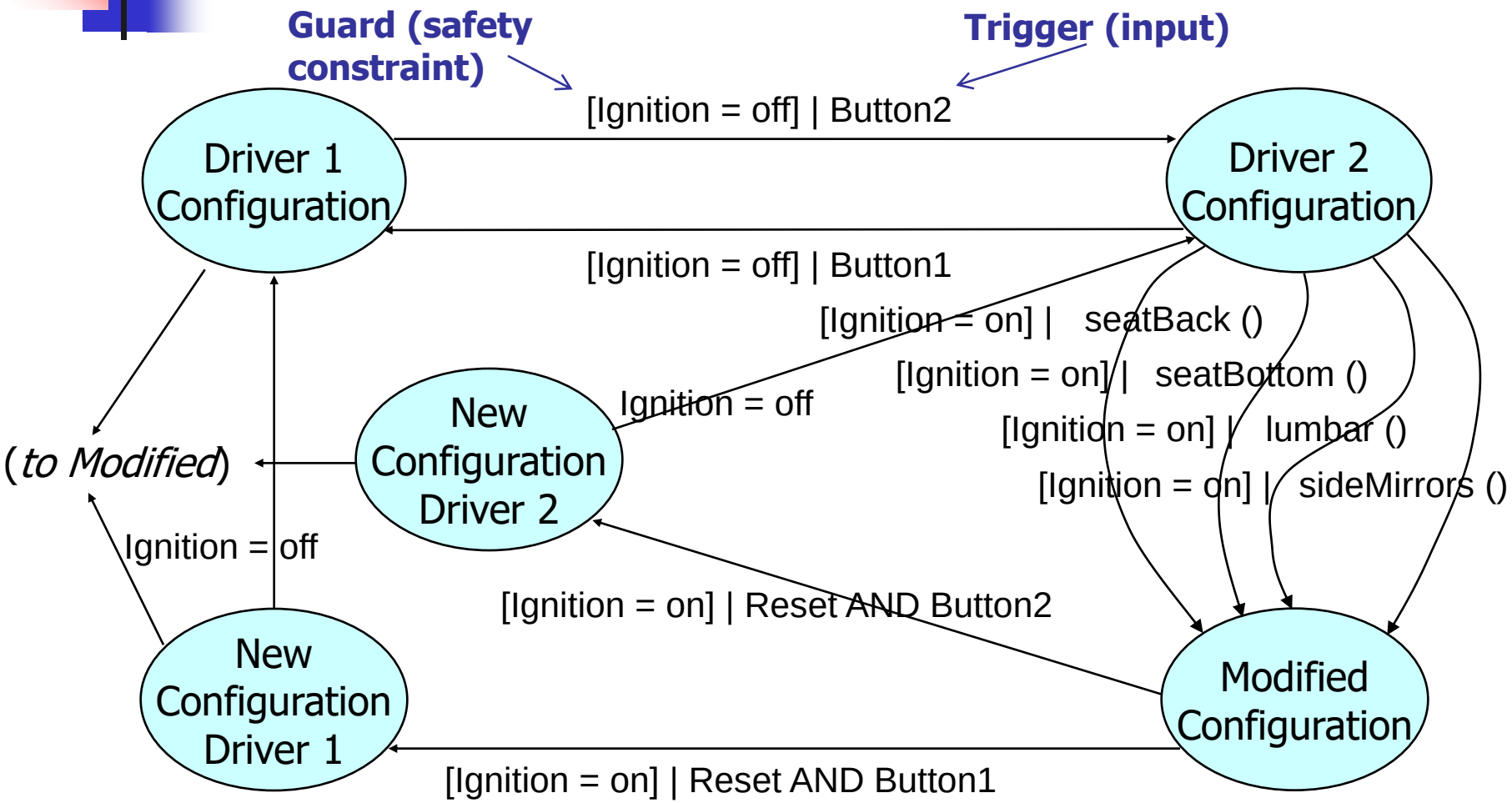
- **defs: nodes & edges where variables get values**
- **uses: nodes & edges where values are accessed**

All Uses

Every def "reaches" every use

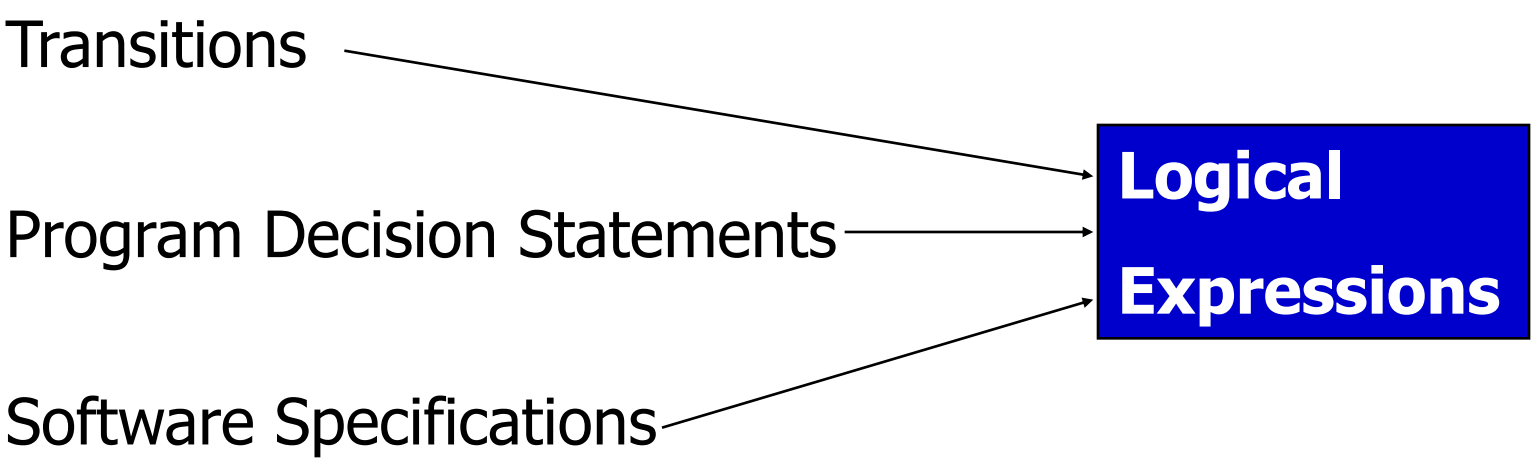
- **1, 2, 5, 6, 7**
- **1, 2, 5, 7**
- **1, 3, 5, 6, 7**
- **1, 3, 5, 7**
- **1, 3, 4, 3, 5,7**

1. Graph Coverge: Example FSM for Memory Seats in Lexus ES 300



2. Logical Expressions

$$((a > b) \text{ or } G) \text{ and } (x < y)$$



2. Logical Expressions

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

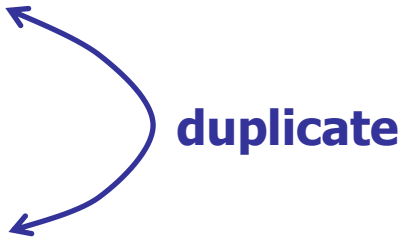
- Predicate Coverage: Each predicate must be true and false
  - *((a>b) or G) and (x < y)* = True, False
- Clause Coverage : Each clause must be true and false
  - *(a > b)* = True, False
  - *G* = True, False
  - *(x < y)* = True, False
- Combinatorial Coverage : Various combinations of clauses
  - *Active Clause Coverage*: Each clause must determine the predicate's result

2. Logical Expressions
Active Clause Coverage

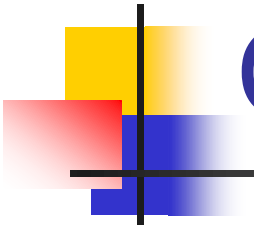
With these values for G and (x<y), (a>b) determines the value of the predicate

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

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



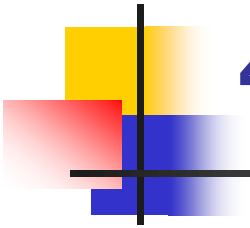
3. Input Domain Characterization



- Describe the **input domain** of the software
  - Identify inputs, parameters, or other categorization
  - Partition each input into finite sets of representative values
  - Choose combinations of values

3. Input Domain Characterization

- System level
  - Number of students { *0, 1, >1* }
  - Level of course { *600, 700, 800* }
  - Major { *swe, cs, isa, infs* }
- Unit level
  - Parameters *F (int X, int Y)*
  - Possible values *X: { <0, 0, 1, 2, >2 }, Y : { 10, 20, 30 }*
  - Tests *F (-5, 10), F (0, 20), F (1, 30), F (2, 10), F (5, 20)*



4. Syntactic Structures

- Based on a grammar, or other syntactic definition
- Primary example is mutation testing
  1. Induce **small changes** to the program: mutants
  2. Find **tests** that cause the mutant programs to fail
  3. Failure is defined as different output from the original program
  4. Check the output of useful tests on the original program

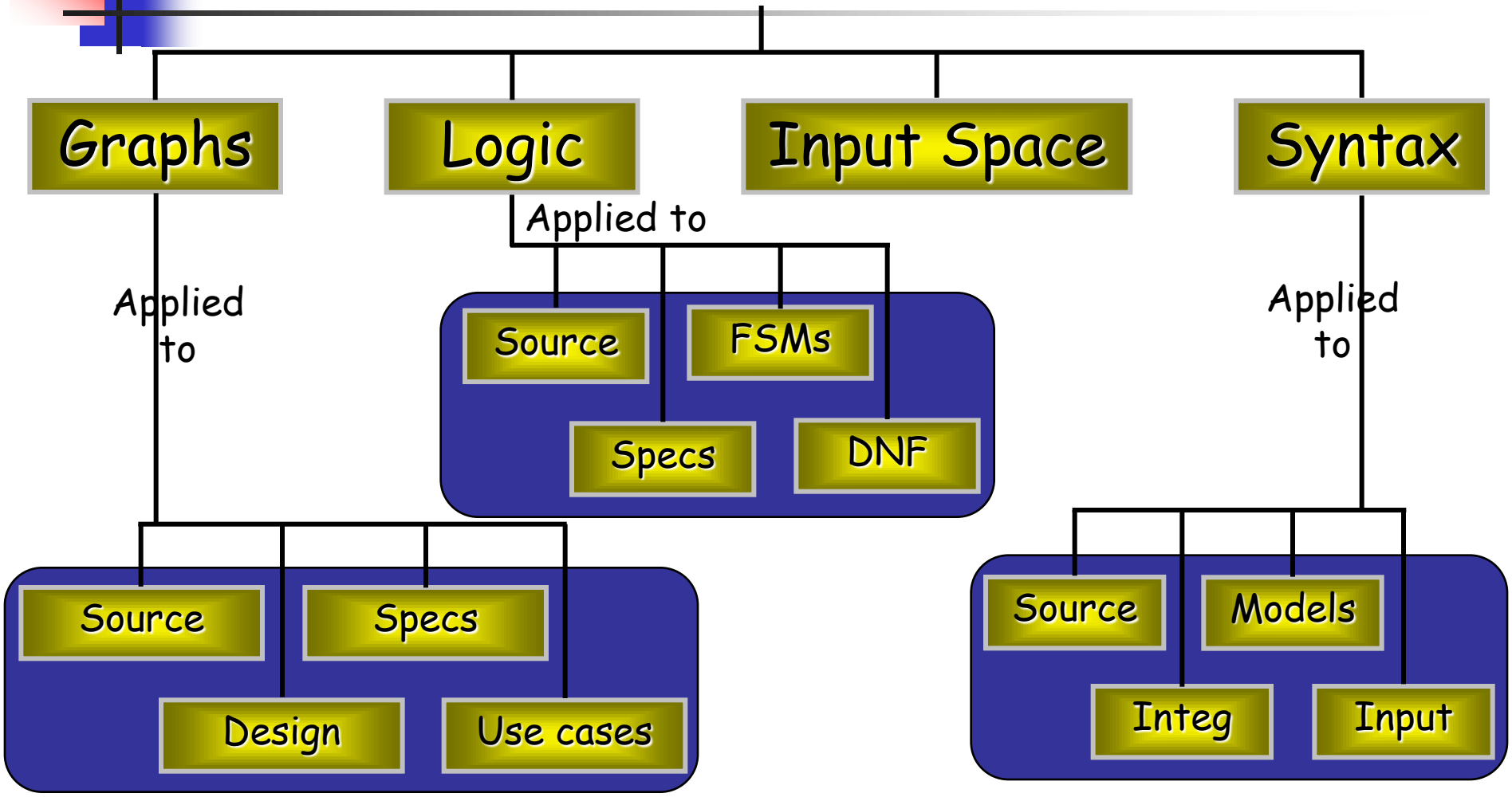
Mutation Testing

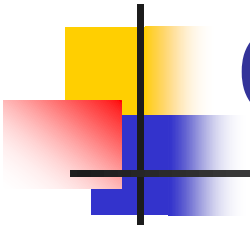
- Example program and mutants

```
if (x > y)
    z = x - y;
else
   z = 2 * x;
```

```
if (x > y)
Δif (x >= y)
    z = x - y;
    Δ z = x + y;
    Δ z = x – m;
else
     z = 2 * x;
```

4 Structures for Modeling Software





Coverage

- **Given a set of test requirements *TR* for coverage criterion *C*, a test set *T* satisfies *C* coverage if and only if for every test requirement *tr* in *TR*, there is at least one test *t* in *T* such that *t* satisfies *tr***

- 100% coverage is **impossible** in practice
  - No test case values exist that meet the test requirements
  - Dead code
  - Detection of **infeasible test requirements** is formally undecidable for most test criteria

Using Criteria (Metrics)

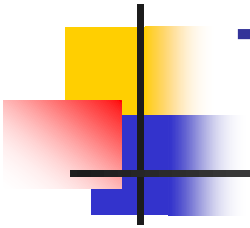
- Directly generate test values to **satisfy** the criterion
  - research focus

- Generate test values **externally** and **measure** against the criterion
  - industrial practice
  - sometimes misleading, e.g, if tests do not reach 100% coverage, what does that mean?

Comparing Criteria: Subsumption

- Criteria Subsumption : A test criterion *C1* subsumes *C2* if and only if every set of test cases that satisfies criterion *C1* also satisfies *C2*

- Must be true for every set of test cases
- *Example* : If a test set has covered every branch in a program (satisfied the branch criterion), then the test set is guaranteed to also have covered every statement



Test Coverage Criteria

- Traditional software testing is expensive and labor-intensive
- Formal coverage criteria are used to decide which test inputs to use
- More likely that the tester will find problems
- Greater assurance that the software is of high quality and reliability
- A goal or stopping rule for testing
- Criteria makes testing more efficient and effective