CS409
Software Testing

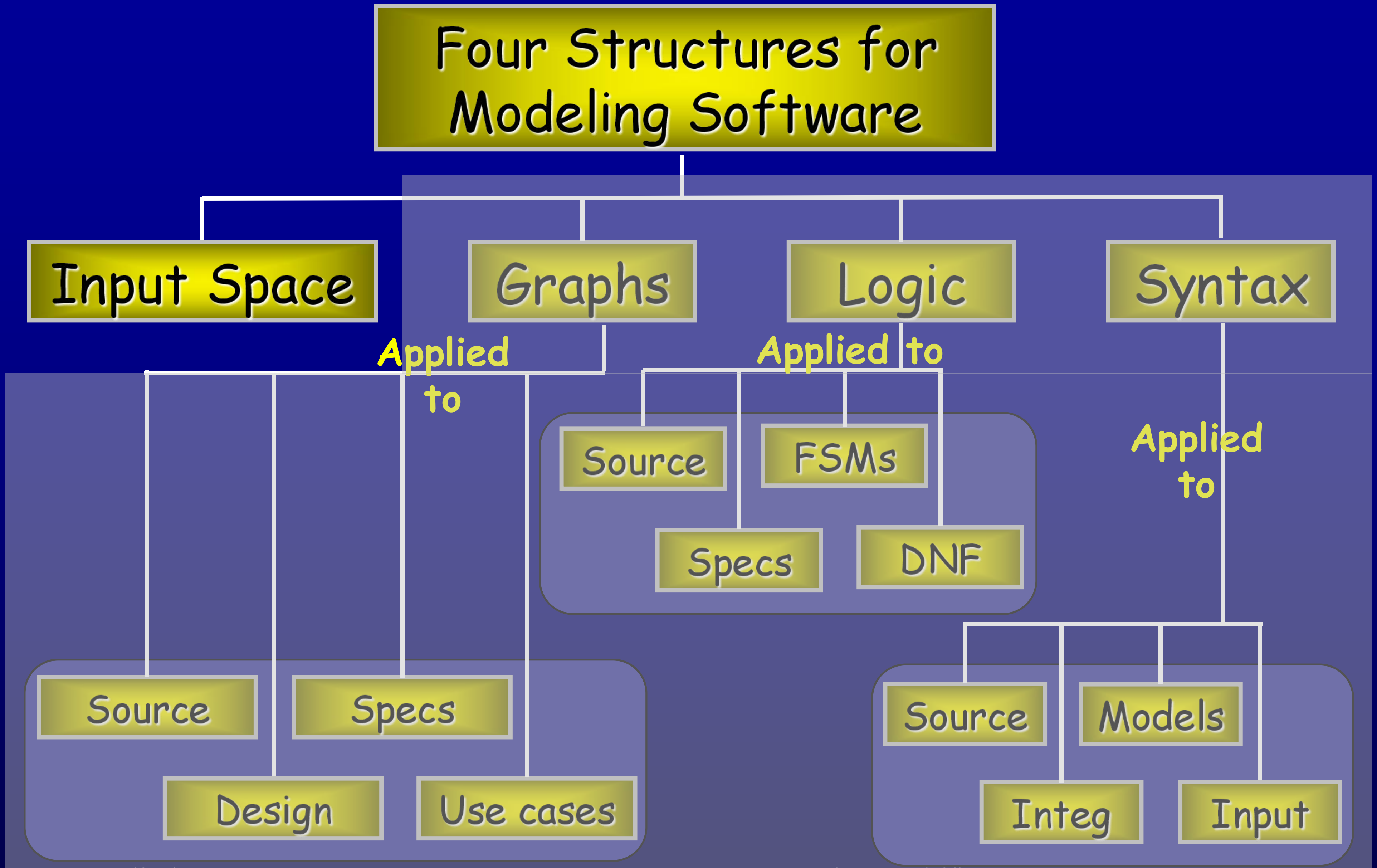
TAN, Shin Hwei
陈馨慧
Southern University of Science and Technology

Slides adapted from Introduction to Software Testing, Edition 2 (Ch 6)

Administrative Info

- MP1 was due on 10 October, 11.59pm. Late submission get 0!
- Project proposal due soon on October 16, 11.59pm!

Continue on Ch. 6 : Input Space Coverage



Recap: IDM

What two properties must be satisfied for an input domain to be properly partitioned?

Recap：IDM

Interface-based

- Develops characteristics directly from individual input parameters
- Consider syntax
- Example：relationship with zero(>0, =0,<0)

Functionality-based

- Develops characteristics from a behavioral view of the program under test
- Consider domain and semantic knowledge
- Example：Types of Triangle

Recap: Iterator example from the lab

Methods

Modifier and Type	Method and Description
boolean	hasNext() Returns true if the iteration has more elements.
E	next() Returns the next element in the iteration.
void	remove() Removes from the underlying collection the last element returned by this iterator (optional operation).

Task I: Determine Characteristics

Step 1: Identify:

- Functional units
- Parameters
- Return types and return values
- Exceptional behavior

work …

Task I: Determine Characteristics

Step 1: Identify:

- hasNext() – Returns true if more elements
- E next() – Returns next element
  - Exception: NoSuchElementException
- void remove() – Removes the most recent element returned by the iterator
  - Exception: Unsupported-OperationException
  - Exception: IllegalStateException
- parameters: state of the iterator
  - iterator state changes with next(), and remove() calls
  - modifying underlying collection also changes iterator state

Task I: Determine Characteristics

Step 2: Develop Characteristics

Table A:

Method	**Params**	**Returns**	**Values**	**Exception**	**Ch ID**	**Character -istic**	**Covered by**
hasNext	state	boolean	true, false				
next	state	E element generic	E, null				
remove	state						

work …

Task I: Determine Characteristics

Step 2: Develop Characteristics

Table A:

Method	**Params**	**Returns**	**Values**	**Exception**	**Ch ID**	**Character -istic**	**Covered by**
hasNext	state	boolean	true, false		C1	More values	
next	state	E element generic	E, null				
remove	state						

Task I: Determine Characteristics

Step 2: Develop Characteristics

Table A:

Method	Params	Returns	Values	Exception	Ch ID	Character-istic	Covered by
hasNext	state	boolean	true, false		C1	More values	
next	state	E element generic	E, null		C2	Returns non-null object	
remove	state						

Task I: Determine Characteristics

Step 2: Develop Characteristics

Table A:

Method	Params	Returns	Values	Exception	Ch ID	Character-istic	Covered by
hasNext	state	boolean	true, false		C1	More values	
next	state	E element generic	E, null		C2	Returns non-null object	
				NoSuchElement			C1
remove	state						

Task I: Determine Characteristics

Step 2: Develop Characteristics

Table A:

Method	Params	Returns	Values	Exception	Ch ID	Character-istic	Covered by
hasNext	state	boolean	true, false		C1	More values	
next	state	E element generic	E, null		C2	Returns non-null object	
				NoSuchElement			C1
remove	state			Unsupported	C3	remove() supported	

Task I: Determine Characteristics

Step 2: Develop Characteristics

Table A:

Method	Params	Returns	Values	Exception	Ch ID	Character-istic	Covered by
hasNext	state	boolean	true, false		C1	More values	
next	state	E element generic	E, null		C2	Returns non-null object	
				NoSuchEle ment			C1
remove	state			Unsupport ed	C3	remove() supported	
				IllegalState	C4	remove() constraint satisfied	

Task I: Determine Characteristics

Step 4: Design a partitioning

Which methods is each characteristic relevant for?

How can we partition each characteristic?

Table B:

ID	Characteristic	hasNext()	next()	Remove()	Partition
C1	More values				
C2	Returns non-null object				
C3	remove() supported				
C4	remove() constraint satisfied				

work …

Task I: Determine Characteristics

Step 4: Design a partitioning

Relevant characteristics for each method

Table B:

ID	Characteristic	hasNext()	next()	Remove()	Partition
C1	More values	X	X	X	
C2	Returns non-null object		X	X	
C3	remove() supported			X	
C4	remove() constraint satisfied			X	

Task I: Determine Characteristics

Step 4: Design a partitioning

Table B:

ID	Characteristic	hasNext()	next()	Remove()	Partition
C1	More values	X	X	X	{true, false}
C2	Returns non-null object		X	X	{true, false}
C3	remove() supported			X	{true, false}
C4	remove() constraint satisfied			X	{true, false}

Task II: Define Test Requirements

- Step 1: Choose coverage criterion
- Step 2: Choose base cases if needed

work …

Task II: Define Test Requirements

- Step 1: Base Choice coverage criterion (BCC)
- Step 2: Happy path (all true)
- Step 3: Test requirements …

<u>Base Choice Coverage (BCC)</u> : A base choice block is chosen for each characteristic, and **a base test** is formed by using the base choice for each characteristic. Subsequent tests are chosen by **holding all but one base choice constant** and using each non-base choice in each other characteristic.

<u>Base Choice Notes</u>

- The base test must be **feasible**
- **Happy path** tests often make good base choices

Task II: Define Test Requirements

- Step 3: Test requirements

Table C:

Method	Characteristics	Test Requirements	Infeasible TRs
hasNext	C1		
next	C1 C2		
remove	C1 C2 C3 C4		

work …

Task II: Define Test Requirements

- Step 3: Test requirements

Table C:

Method	Characteristics	Test Requirements	Infeasible TRs
hasNext	C1	{**T**, F}	
next	C1 C2	{**TT**, FT, TF}	
remove	C1 C2 C3 C4	{**TTTT**, FTTT, TFTT, TTFT, TTTF}	

ID	Characteristic
C1	More values
C2	Returns non-null object
C3	remove() supported
C4	remove() constraint satisfied

Task II: Define Test Requirements

- Step 4: Infeasible test requirements

Table C:

C1=F: has no values
C2=T: returns non-null object

Method	Characteristics	Test Requirements	Infeasible TRs
hasNext	C1	{**T**, F}	none
next	C1 C2	{**TT**, FT, TF}	FT
remove	C1 C2 C3 C4	{**TTTT**, FTTT, TFTT, TTFT, TTTF}	FTTT

Task II: Define Test Requirements

- Step 5: Revised infeasible test requirements

Table C:

Method	Characteristics	Test Requirements	Infeasible TRs	Revised TRs	# TRs
hasNext	C1	{**T**, F}	none	n/a	2
next	C1 C2	{**TT**, FT, TF}	FT	FT → F**F**	3
remove	C1 C2 C3 C4	{**TTTT**, FTTT, TFTT, TTFT, TTTF}	FTTT	FTTT → F**F**TT	5

Task III: Automate Tests

- First, we need an implementation of Iterator
  - (Iterator is just an interface)
  - ArrayList implements Iterator
- Test fixture has two variables:
  - List of strings
  - Iterator for strings
- setUp()
  - Creates a list with two strings
  - Initializes an iterator

Don’t forget to complete the ISP-lab!

Base Choice Notes

- The base test must be feasible
  - That is, all base choices must be compatible
- Base choices can be
  - Most likely from an end-use point of view
  - Simplest
  - Smallest
  - First in some ordering
- Happy path tests often make good base choices
- The base choice is a crucial design decision
  - Test designers should document why the choices were made

ISP Criteria – Multiple Base Choice

- We sometimes have more than one logical base choice

<u>Multiple Base Choice Coverage (MBCC)</u> : At least one, and possibly more, base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choice in each other characteristic.

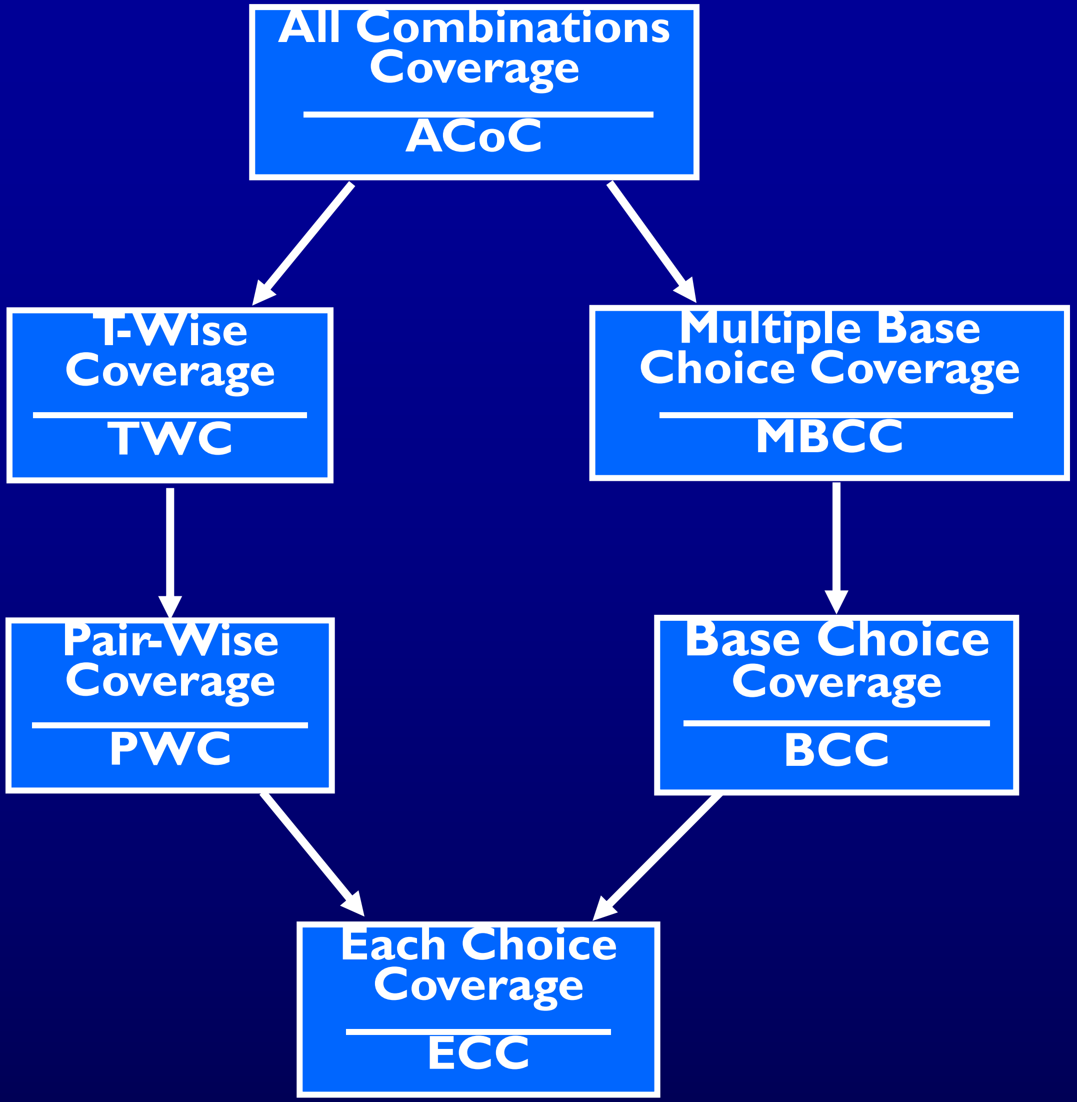
- **If *M* base tests and m_i base choices for each characteristic:**

$$M + \sum_{i=1}^{Q} (M * (B_i - m_i))$$

For *triang*() : <u>Bases</u>

A1, B1, C1	**A1, B1, C3**	**A1, B3, C1**	**A3, B1, C1**
	A1, B1, C4	**A1, B4, C1**	**A4, B1, C1**
A2, B2, C2	**A2, B2, C3**	**A2, B3, C2**	**A3, B2, C2**
	A2, B2, C4	**A2, B4, C2**	**A4, B2, C2**

ISP Coverage Criteria Subsumption



Constraints Among Characteristics

(6.3)

- Some combinations of blocks are infeasible
  - "less than zero" and "scalene" … not possible at the same time
- These are represented as constraints among blocks
- Two general types of constraints
  - A block from one characteristic cannot be combined with a specific block from another
  - A block from one characteristic can ONLY BE combined with a specific block form another characteristic
- Handling constraints depends on the criterion used
  - ACC, PWC, TWC : Drop the infeasible pairs
  - BCC, MBCC : Change a value to another non-base choice to find a feasible combination

Example Handling Constraints

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
A : length and contents	One element	More than one, unsorted	More than one, sorted	More than one, all identical
B : match	element not found	element found once	element found more than once	
Invalid combinations : (**A1**, **B3**), (**A4**, **B2**)				

element cannot be in a one-element list more than once

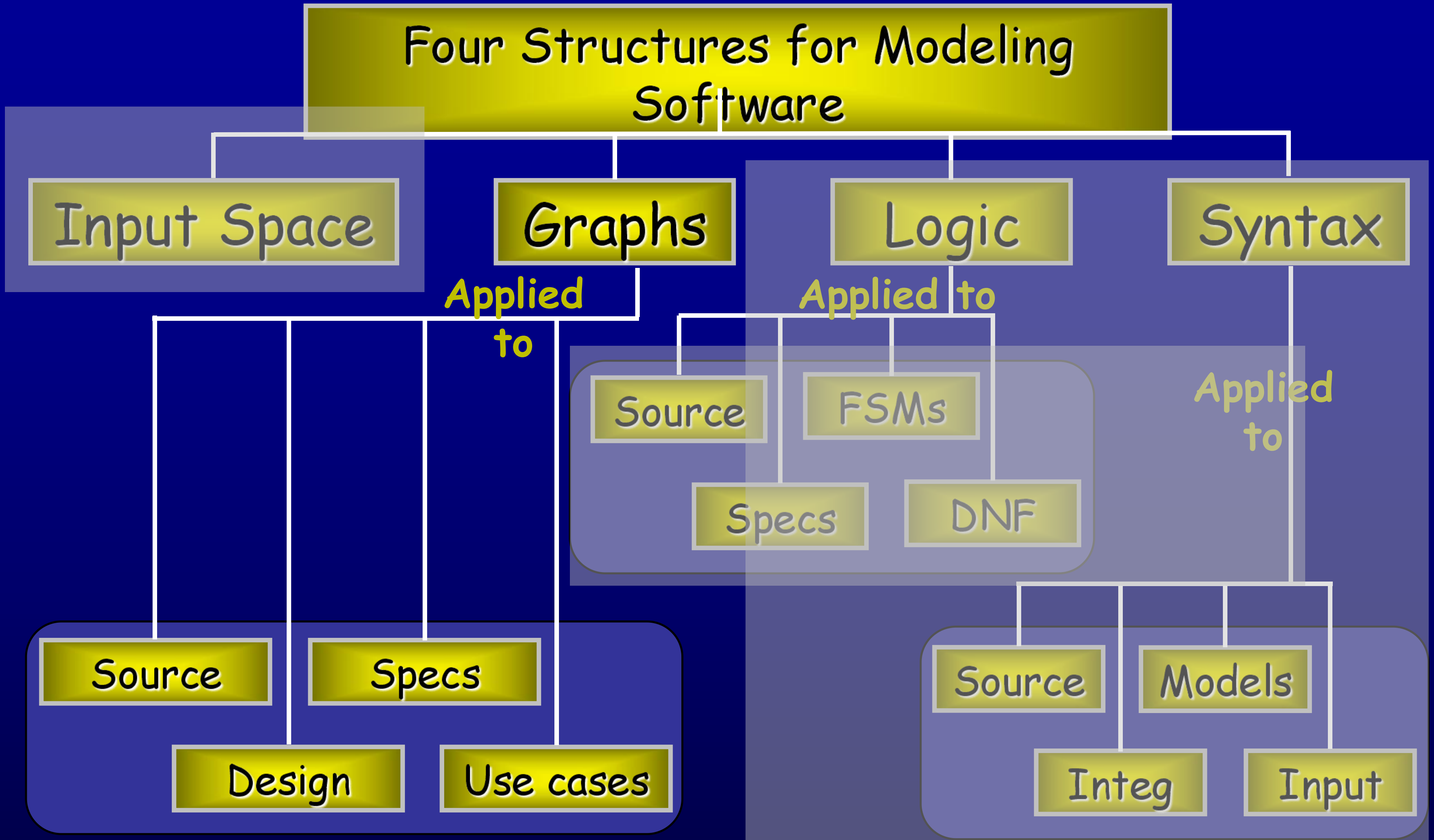
If the list only has one element, but it appears multiple times, we cannot find it just once

Input Space Partitioning Summary

- Fairly easy to apply, even with no automation
- Convenient ways to add more or less testing
- Applicable to all levels of testing – unit, class, integration, system, etc.
- Based only on the input space of the program, not the implementation

Simple, straightforward, effective, and widely used

Ch. 7 : Graph Coverage



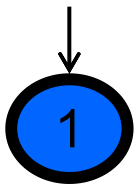
Covering Graphs (7.1)

- Graphs are the most commonly used structure for testing
- Graphs can come from many sources
  - Control flow graphs
  - Design structure
  - FSMs (Finite-state machines) and statecharts
  - Use cases
- Tests usually are intended to "cover" the graph in some way

Definition of a Graph

- A set N of nodes, N is not empty
- A set N_0 of initial nodes, N_0 is not empty
- A set N_f of final nodes, N_f is not empty
- A set E of edges, each edge from one node to another
  - (n_i, n_j), i is predecessor, j is successor

Is this a graph?

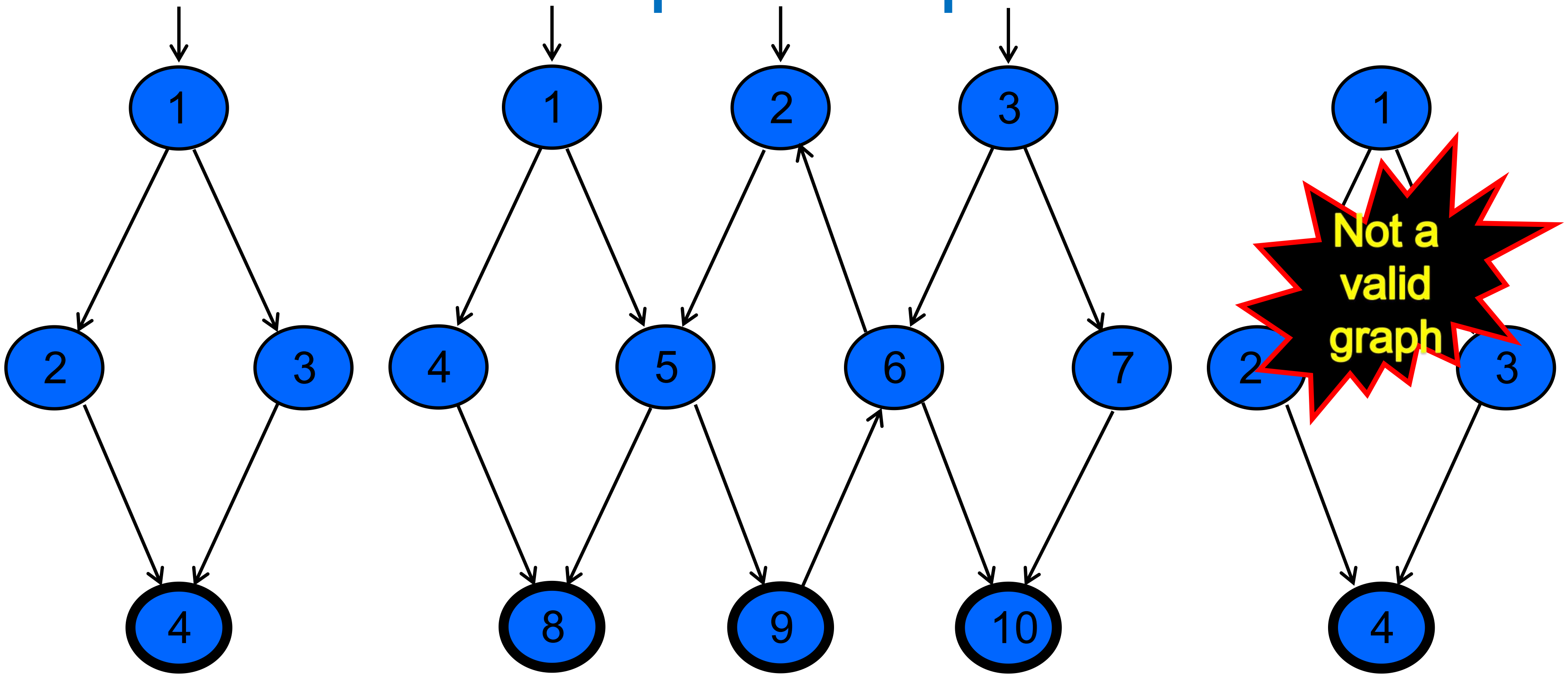


$N_0 = \{ 1 \}$

$N_f = \{ 1 \}$

$E = \{ \}$

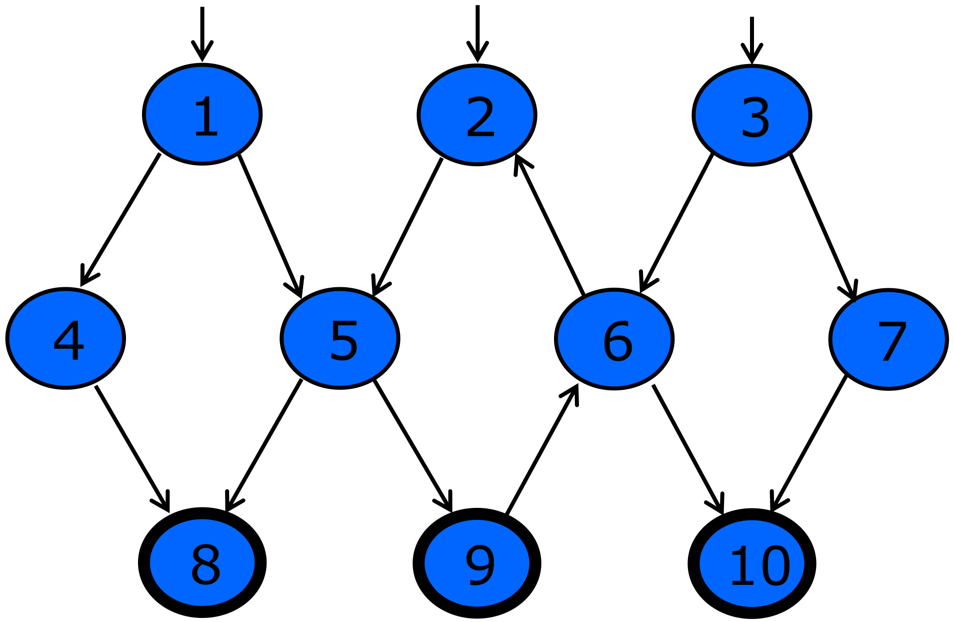
Example Graphs



$N_0 = \{ 1 \}$	$N_0 = \{ 1, 2, 3 \}$	$N_0 = \{ \}$
$N_f = \{ 4 \}$	$N_f = \{ 8, 9, 10 \}$	$N_f = \{ 4 \}$
E = { (1,2), (1,3), (2,4), (3,4) }	E = { (1,4), (1,5), (2,5), (3,6), (3, 7), (4, 8), (5,8), (5,9), (6,2), (6,10), (7,10) (9,6) }	E = { (1,2), (1,3), (2,4), (3,4) }

Paths in Graphs

- Path : A sequence of nodes – $[n_1, n_2, \ldots, n_M]$
  - Each pair of nodes is an edge
- Length : The number of edges
  - A single node is a path of length 0
- Subpath : A subsequence of nodes in *p* is a subpath of *p*
- Reach (*n*) : Subgraph that can be reached from *n*



A Few Paths

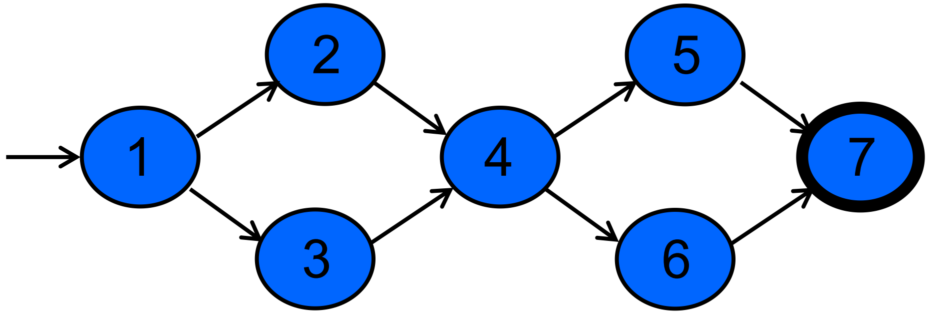
[1, 4, 8]

[2, 5, 9, 6, 2]

[3, 7, 10]

Test Paths and SESEs

- Test Path : A path that starts at an initial node and ends at a final node
- Test paths represent execution of test cases
  - Some test paths can be executed by many tests
  - Some test paths cannot be executed by any tests
- SESE graphs : All test paths start at a single node and end at another node
  - Single-entry, single-exit
  - N_0 and N_f have exactly one node



<u>Double-diamond graph</u>

Four test paths

[1, 2, 4, 5, 7]

[1, 2, 4, 6, 7]

[1, 3, 4, 5, 7]

[1, 3, 4, 6, 7]0

Visiting and Touring

- Visit : A test path *p* *visits* node *n* if *n* is in *p*
  A test path *p* *visits* edge *e* if *e* is in *p*
- Tour : A test path *p* *tours* subpath *q* if *q* is a subpath of *p*

Test path [1, 2, 4, 5, 7]

Visits nodes ? 1, 2, 4, 5, 7

Visits edges ? (1,2), (2,4), (4, 5), (5, 7)

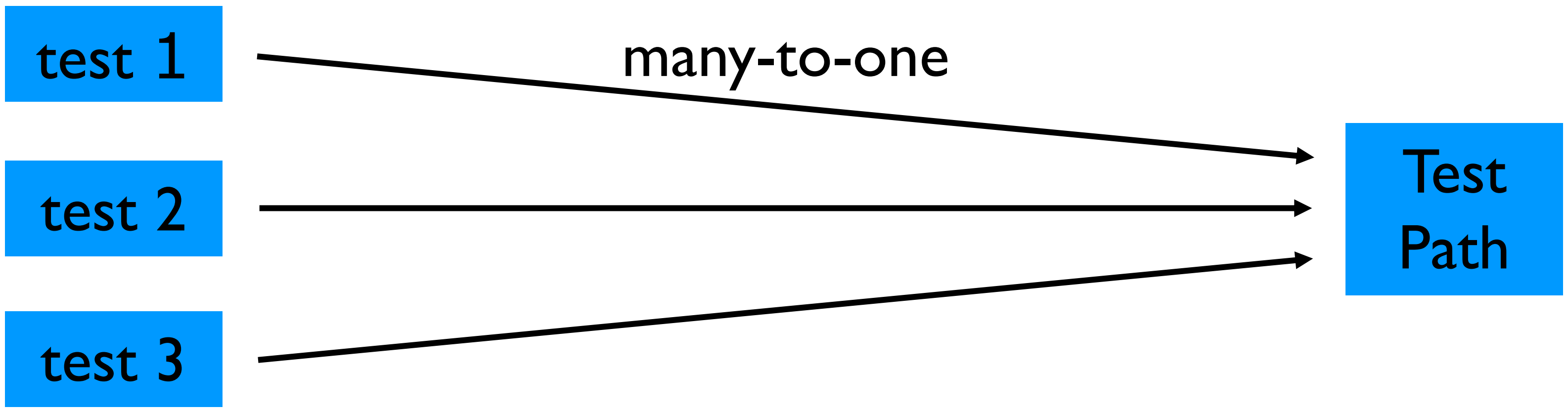
Tours subpaths ? [1,2,4], [2,4,5], [4,5,7], [1,2,4,5], [2,4,5,7], [1,2,4,5,7]

(Also, each edge is technically a subpath)

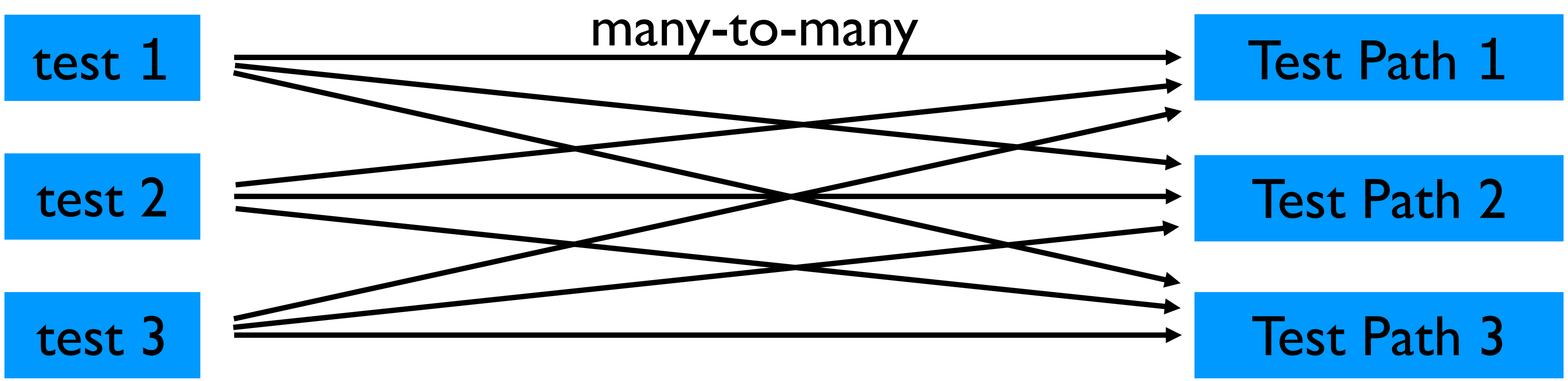
Tests and Test Paths

- path (*t*) : The test path executed by test *t*
- path (*T*) : The set of test paths executed by the set of tests *T*
- Each test executes one and only one test path
  - Complete execution from a start node to an final node
- A location in a graph (node or edge) can be reached from another location if there is a sequence of edges from the first location to the second
  - *Syntactic reach* : A subpath exists in the graph
  - *Semantic reach* : A test exists that can execute that subpath
  - This distinction becomes important in section 7.3

Tests and Test Paths



Deterministic software–test always executes the same test path



Non-deterministic software–the same test can execute different test paths

Testing and Covering Graphs (7.2)

- We use graphs in testing as follows :
  - Develop a model of the software as a graph
  - Require tests to visit or tour specific sets of nodes, edges or subpaths
- Test Requirements (TR) : Describe properties of test paths
- Test Criterion : Rules that define test requirements
- Satisfaction : *Given a set TR of test requirements for a criterion C, a set of tests T satisfies C on a graph if and only if for every test requirement in TR, there is a test path in path(T) that meets the test requirement tr*
- Structural Coverage Criteria : Defined on a graph just in terms of nodes and edges
- Data Flow Coverage Criteria : Requires a graph to be annotated with references to variables

Node and Edge Coverage

- The first (and simplest) two criteria require that each node and edge in a graph be executed

Node Coverage (NC) : Test set *T* satisfies node coverage on graph *G* iff for every syntactically reachable node *n* in *N*, there is some path *p* in *path(T)* such that *p* visits *n*.

- This statement is a bit 难懂, so we simplify it in terms of the set of test requirements

Node Coverage (NC) : TR contains each reachable node in G.

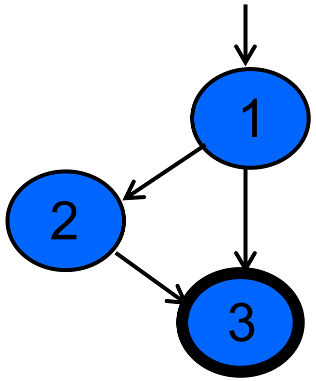
Node and Edge Coverage

- Edge coverage is slightly stronger than node coverage

> <u>Edge Coverage (EC)</u> : TR contains each reachable path of length up to 1, inclusive, in G.

- The phrase "*length up to 1*" allows for graphs with one node and no edges

- NC and EC are only different when there is an edge and another subpath between a pair of nodes (as in an "if-else" statement)

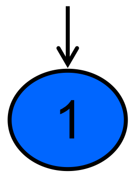


<u>Node Coverage</u> : ?   TR = { 1, 2, 3 }
Test Path = [1, 2, 3]

<u>Edge Coverage</u> : ?   TR = { (1, 2), (1, 3), (2, 3) }
Test Paths = [1, 2, 3]
[1, 3]

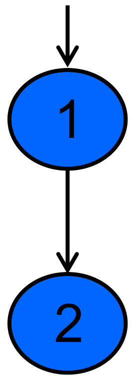
Paths of Length 1 and 0

- A graph with only one node will not have any edges



- It may seem trivial, but formally, Edge Coverage needs to require Node Coverage on this graph
- Otherwise, Edge Coverage will not subsume Node Coverage
  - So we define "length up to 1" instead of simply "length 1"

- We have the same issue with graphs that only have one edge – for Edge-Pair Coverage …

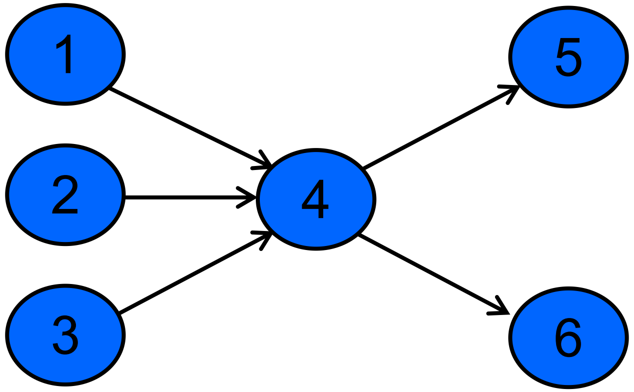


Covering Multiple Edges

- Edge-pair coverage requires pairs of edges, or subpaths of length 2

Edge-Pair Coverage (EPC) : TR contains each reachable path of length up to 2, inclusive, in G.

- The phrase "length up to 2" is used to include graphs that have less than 2 edges



Edge-Pair Coverage : ?

TR = { [1,4,5], [1,4,6], [2,4,5], [2,4,6], [3,4,5], [3,4,6] }

- The logical extension is to require all paths …

Covering Multiple Edges

Complete Path Coverage (CPC) : TR contains all paths in G.

Unfortunately, this is impossible if the graph has a loop, so a weak compromise(妥协) makes the tester decide which paths:

Specified Path Coverage (SPC) : TR contains a set S of test paths, where S is supplied as a parameter.

Coverage Summary

Node Coverage (NC) : Test set *T* satisfies node coverage on graph *G* iff for every syntactically reachable node *n* in *N*, there is some path *p* in *path(T)* such that *p* visits *n*.

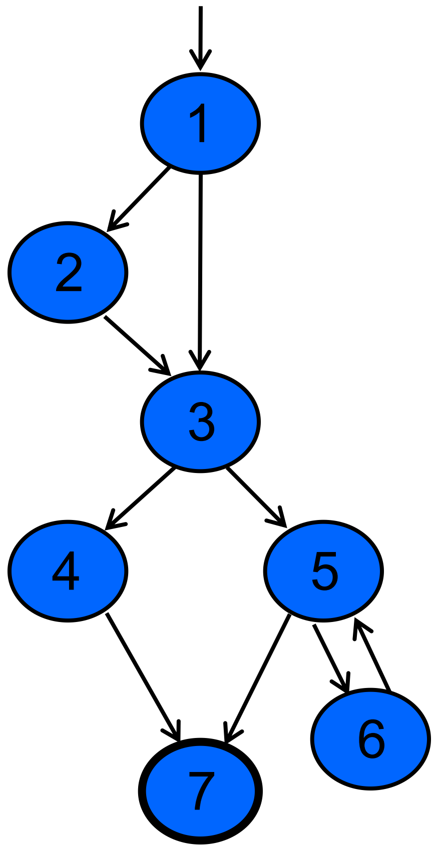
Edge Coverage (EC) : TR contains each reachable path of length up to 1, inclusive, in G.

Edge-Pair Coverage (EPC) : TR contains each reachable path of length up to 2, inclusive, in G.

Node Coverage (NC) : TR contains each reachable node in G.

Complete Path Coverage (CPC) : TR contains all paths in G.

Structural Coverage Example



Node Coverage

TR = { 1, 2, 3, 4, 5, 6, 7 }
Test Paths: [1, 2, 3, 4, 7] [1, 2, 3, 5, 6, 5, 7]

Write down the TRs and Test Paths for these criteria

Edge Coverage

TR = { (1,2), (1, 3), (2, 3), (3, 4), (3, 5), (4, 7 (6, 5) }
Test Paths: [1, 2, 3, 4, 7] [1, 3, 5, 6, 5, 7]

Edge-Pair Coverage

TR = { [1,2,3], [1,3,4], [1,3,5], [2,3,4], [2,3,5], [3,4,7], [3,5,6], [3,5,7], [5,6,5], [6,5,6], [6,5,7] }
Test Paths: [1, 2, 3, 4, 7] [1, 2, 3, 5, 7] [1, 3, 4, 7] [1, 3, 5, 6, 5, 6, 5, 7]

Complete Path Coverage

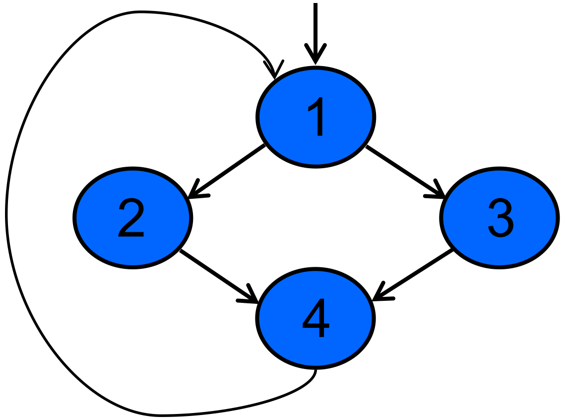
Test Paths: [1, 2, 3, 4, 7] [1, 2, 3, 5, 7] [1, 2, 3, 5, 6, 5, 7] [1, 2, 3, 5, 6, 5, 6, 5, 7] [1, 2, 3, 5, 6, 5, 6, 5, 6, 5, 7] …

Handling Loops in Graphs

- If a graph contains a loop（循环）, it has an infinite number of paths
- Thus, CPC is not feasible
- Specified Path Coverage（SPC） is not satisfactory because the results are subjective and vary with the tester
- Attempts to "deal with" loops:
  - 1970s : Execute cycles once ([4, 5, 4] in previous example, informal)
  - 1980s : Execute each loop, exactly once (formalized)
  - 1990s : Execute loops 0 times, once, more than once (informal description)
  - 2000s : Prime paths (touring, sidetrips, and detours)

Simple Paths and Prime Paths

- Simple Path : *A path from node n_i to n_j is simple if no node appears more than once, except possibly the first and last nodes are the same*
  - No internal loops
  - A loop is a simple path
- Prime Path : *A simple path that does not appear as a proper subpath of any other simple path*



Simple Paths :
[1, ,4,1,2], [2,4,1,3],
[3,4,1,2], [3,4, ,3,4], [1,2,4], [1,3,4],
[2,4,1], [3,4,1] ,2], [1,3], [2,4], [3,4],
[4,1], [1], [2], [

Write down the simple and prime paths for this graph

Prime Paths : [1,2,4,1], [1,3,4,1], [2,4,1,2], [2,4,1,3],
[3,4,1,2], [3,4,1,3], [4,1,2,4], [4,1,3,4]

Prime Path Coverage

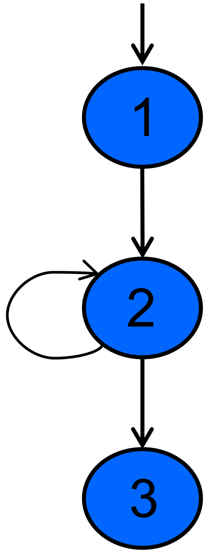
- A simple, elegant and finite criterion that requires loops to be executed as well as skipped

Prime Path Coverage (PPC) : TR contains each prime path in G.

- Will tour all paths of length 0, 1, …
- That is, it subsumes node and edge coverage
- PPC almost, but not quite, subsumes EPC …

PPC Does Not Subsume EPC

- If a node *n* has an edge to itself (*self edge*), EPC requires [*n, n, m*] and [*m, n, n*]
- [*n, n, m*] is not prime
- Neither [*n, n, m*] nor [*m, n, n*] are simple paths (not prime)



EPC Requirements : ?

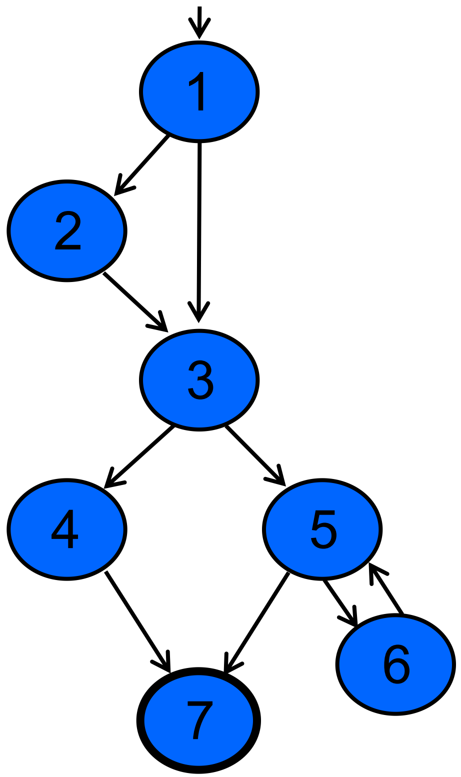
TR = { [1,2,3], [1,2,2], [2,2,3], [2,2,2] }

PPC Requirements : ?

TR = { [1,2,3], [2,2] }

Prime Path Example

- The previous example has 38 simple paths
- Only nine *prime paths*



Prime Paths
[1, 2, 3, 4, 7]

Write down all 9 prime paths

[1, 3, 5, 7]
[1, 3, 5, 6]
[6, 5, 7]
[6, 5, 6]
[5, 6, 5]

Execute loop 0 times

Execute loop once

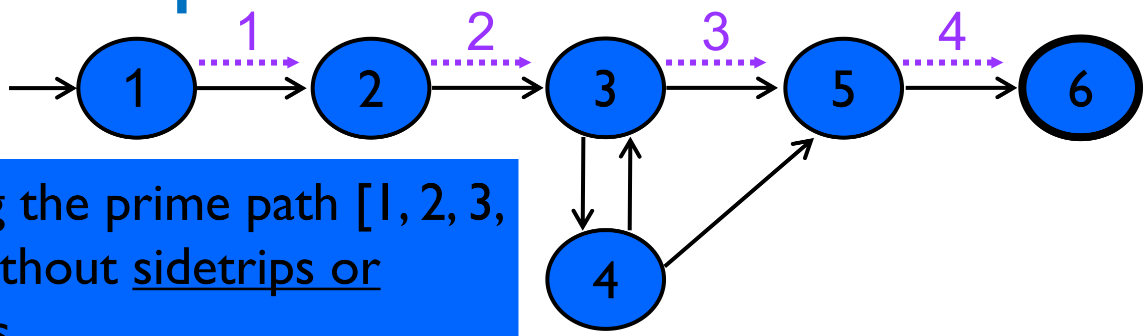
Execute loop more than once

Touring, Sidetrips, and Detours

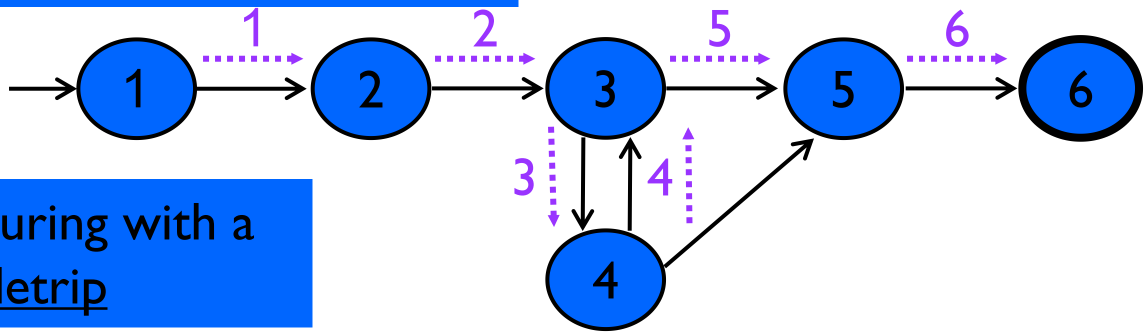
- Prime paths do not have internal loops … test paths <u>might</u>

- Tour : *A test path p tours subpath q if q is a subpath of p*
- Tour With Sidetrips : *A test path p tours subpath q with sidetrips iff every edge in q is also in p in the same order*
  - The tour can include a sidetrip, as long as it comes back to the same node
- Tour With Detours : *A test path p tours subpath q with detours iff every node in q is also in p in the same order*
  - The tour can include a detour from node n_i, as long as it comes back to the prime path at a successor of n_i

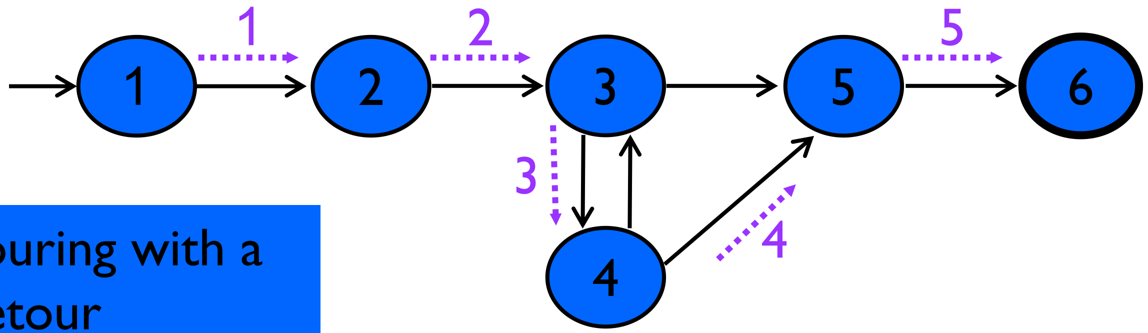
Sidetrips and Detours Example



Touring the prime path [1, 2, 3, 5, 6] without <u>sidetrips or detours</u>



Touring with a <u>sidetrip</u>



Touring with a <u>detour</u>

Infeasible Test Requirements

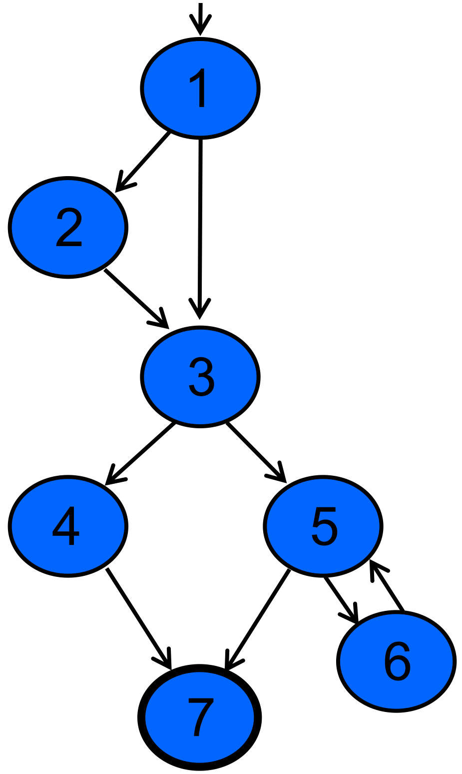
- An infeasible test requirement cannot be satisfied
  - Unreachable statement (dead code)
  - Subpath that can only be executed with a contradiction ($X > 0$ and $X < 0$)
- Most test criteria have some infeasible test requirements
- It is usually undecidable whether all test requirements are feasible
- When sidetrips are not allowed, many structural criteria have more infeasible test requirements
- However, always allowing sidetrips weakens the test criteria

Practical recommendation—Best Effort Touring

– Satisfy as many test requirements as possible without sidetrips

– Allow sidetrips to try to satisfy remaining test requirements

Simple & Prime Path Example



Simple paths

Len 0
[1]

Write paths of length 0

Len 1
[1, 2]
[1, 3]

Write paths of length 1

[5, 6]
[6, 5]

'!' means path

Len 2
[1, 2, 3]
[1, 3, 4]

Write paths of length 2

[3, 5, 6] !
[5, 6, 5] *
[6, 5, 7] !
[6, 5, 6] *

Len 3
[1, 2, 3, 4]
[1, 2, 3, 5]

Write paths of length 3

[2, 3, 5, 6] !
[2, 3, 5, 7] !

path

Len 4

Write paths of length 4

Prime Paths ?

Round Trips

- Round-Trip Path : *A prime path that starts and ends at the same node*

<u>Simple Round Trip Coverage (SRTC)</u> : TR contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.

<u>Complete Round Trip Coverage (CRTC)</u> : TR contains all round-trip paths for each reachable node in G.

- These criteria omit nodes and edges that are not in round trips
- Thus, they do not subsume edge-pair, edge, or node coverage

Data Flow Analysis

Data Flow Analysis

- Can reveal interesting bugs
  - A variable that is defined but never used
  - A variable that is used but never defined
  - A variable that is defined twice before it is used
- Paths from the definition of a variable to its use are more likely to contain bugs

Definitions

- A node in the program graph is a **defining** node for variable v if the value of v is defined at the statement fragment in that node
  - Input, LHS of assignment, procedure calls
- A node in the program graph is a **usage** node for variable v if the value of v is used at the statement fragment in that node
  - Output, RHS of assignment, conditionals

More Definitions

- A usage node is a predicate use (**P-Use**) if variable v appears in a predicate expression (e.g., x>y)
- A usage node is a computation use (**C-Use**) if variable v appears in a computation (e.g., x+y)
- A definition-use path (**du-path**) with respect to a variable v is a path whose first node is a defining node for v, and its last node is a usage node for v
- A du-path with no other defining node for v is a definition-clear path (**dc-path**)

An Example

```
int max = 0;
int i = s.nextInt();
while (i > 0)
  if (i > max) {
    max = i;
  }
  i = s.nextInt();
}
System.out.println(max);
```

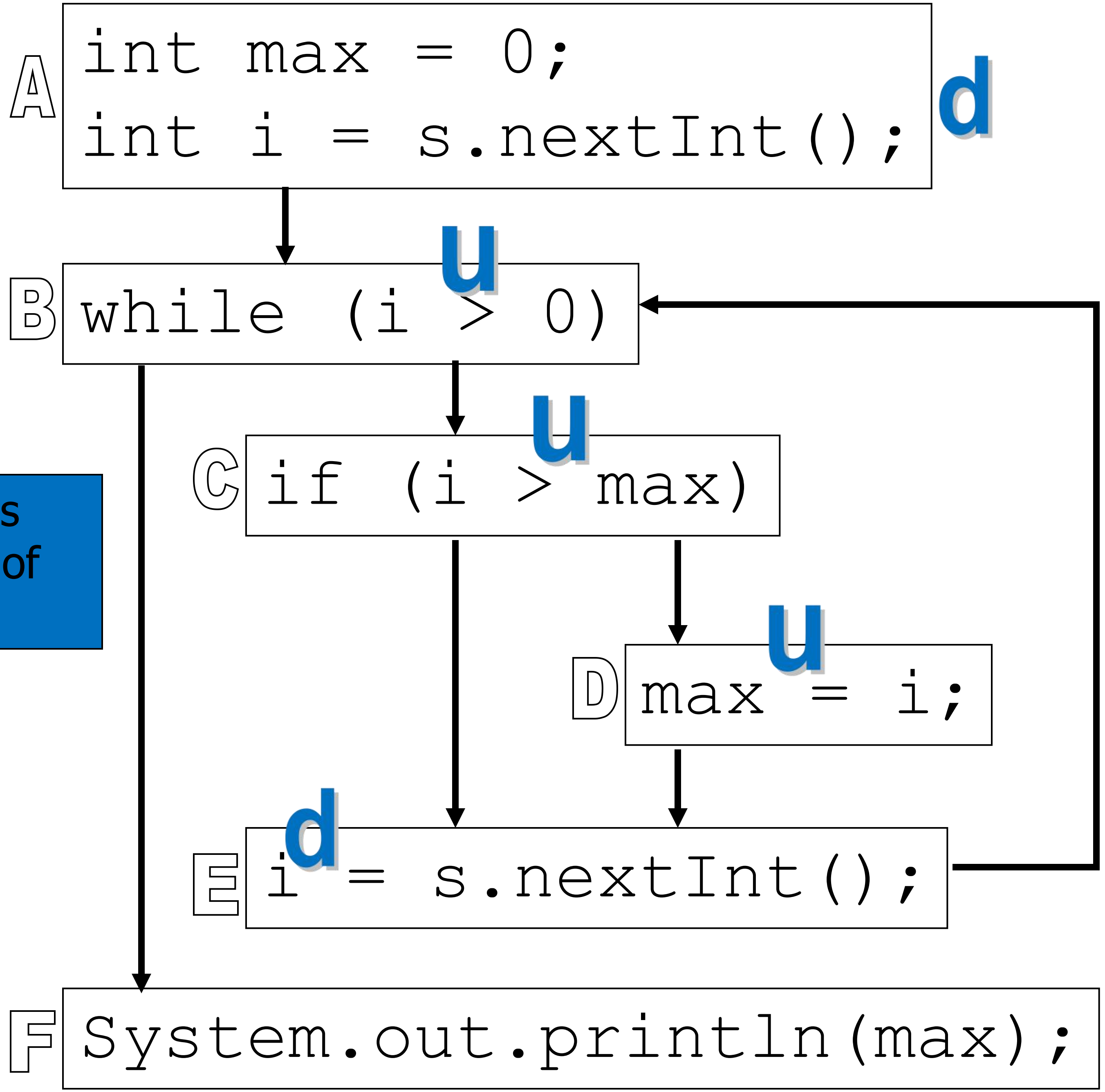
Definitions of max

A definition of i

P-uses of i

A C-use of i

Definitions and uses of variable i



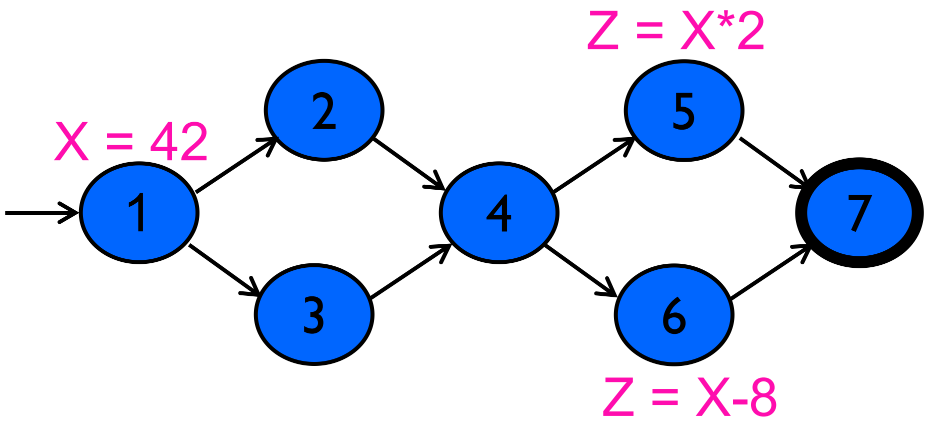
du-paths in example

- Variable i
  - AB, AC, AD, EB, EC, ED
- Variable max
  - AF, AC, DC, DF

Data Flow Criteria

Goal : Ensure that values are computed and used correctly

- Definition (def) : A location where a value for a variable is stored into memory
- Use : A location where a variable's value is accessed



Defs: def (1) = { X }

def (5) = { Z

def (6) = { Z

Uses: use (5) = { X

use (6) = { X }

Fill in these sets

The values given in defs should reach at least one, some, or all possible uses

DU Pairs and DU Paths

- def (n) or def (e) : The set of variables that are defined by node *n* or edge *e*
- use (n) or use (e) : The set of variables that are used by node *n* or edge *e*

- DU pair : A pair of locations (l_i, l_j) such that a variable *v* is defined at l_i and used at l_j

- Def-clear : A path from l_i to l_j is *def-clear* with respect to variable *v* if *v* is not given another value on any of the nodes or edges in the path
- Reach : If there is a def-clear path from l_i to l_j with respect to *v*, the def of *v* at l_i reaches the use at l_j

- du-path : A simple subpath that is def-clear with respect to *v* from a def of *v* to a use of *v*
- du (n_i, n_j, v) – the set of du-paths from n_i to n_j
- du (n_i, v) – the set of du-paths that start at n_i

Touring DU-Paths

- A test path *p* *du-tours* subpath *d* with respect to *v* if *p* tours *d* and the subpath taken is def-clear with respect to *v*
- Sidetrips can be used, just as with previous touring
- Three criteria
  - Use every def
  - Get to every use
  - Follow all du-paths

Data Flow Test Criteria

- First, we make sure every def reaches a use

All-defs coverage (ADC) : For each set of du-paths $S = du(n, v)$, TR contains at least one path d in S.

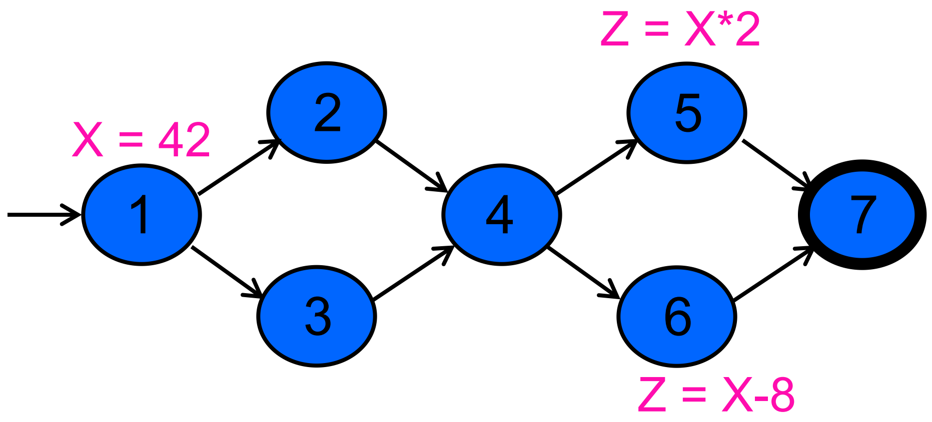
- Then we make sure that every def reaches all possible uses

All-uses coverage (AUC) : For each set of du-paths to uses $S = du(n_i, n_j, v)$, TR contains at least one path d in S.

- Finally, we cover all the paths between defs and uses

All-du-paths coverage (ADUPC) : For each set $S = du(ni, nj, v)$, TR contains every path d in S.

Data Flow Testing Example



All-defs for *X*

Write down paths to satisfy ADC

All-uses for *X*

Write down paths to satisfy AUC

All-du-paths for *X*

Write down paths to satisfy ADUPC

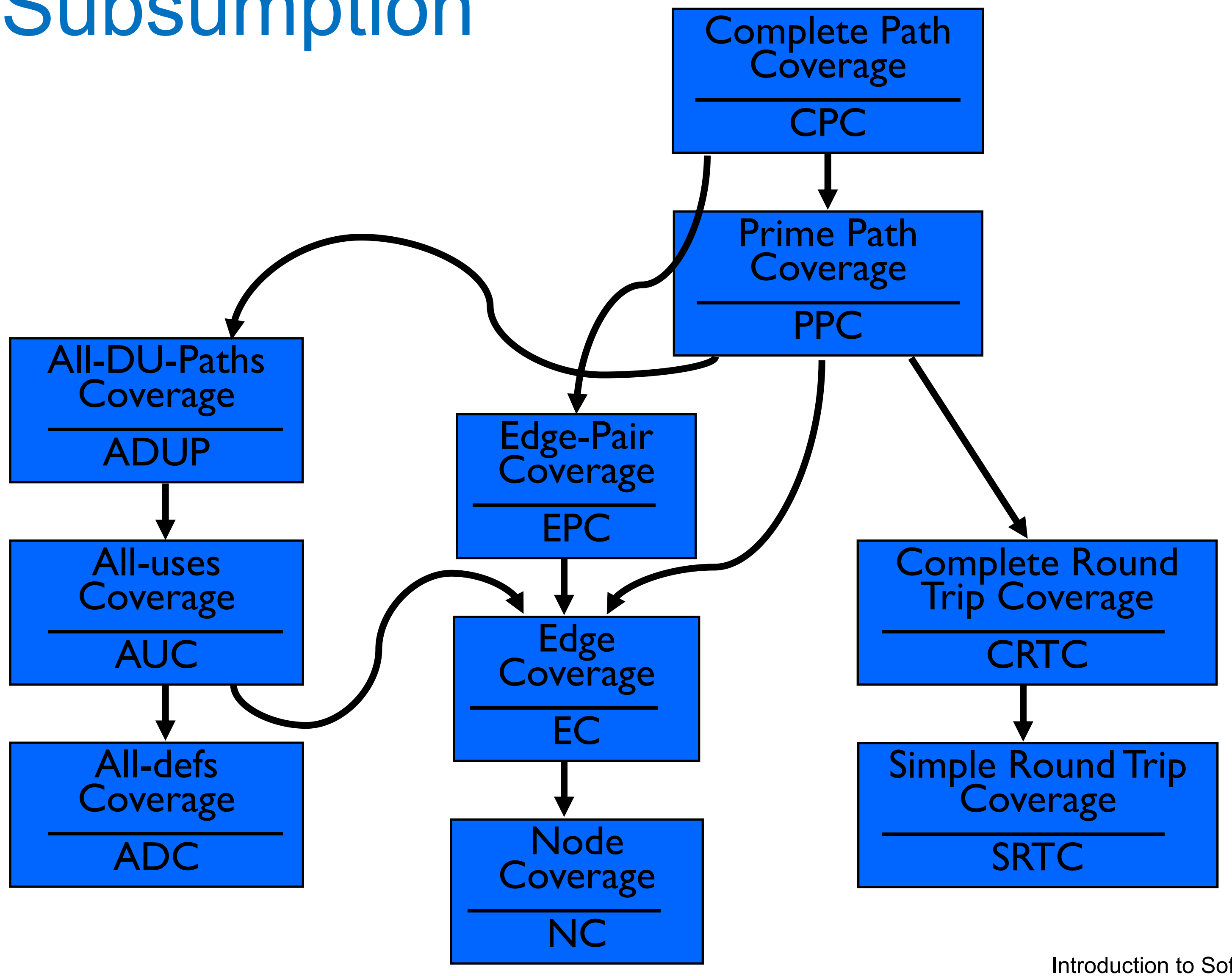
Data flow analysis issues

- Aliasing of variables () causes serious problems!
  - Aliasing: Two different names referring to same storage location

```
int[] a = new int[3];
int[] b = a;
a[2] = 42;
```

- Working things out by hand for anything but small methods is hopeless
- Compiler-based tools help in determining coverage values

Graph Coverage Criteria Subsumption



Summary 7.1-7.2

- Graphs are a very powerful abstraction for designing tests
- The various criteria allow lots of cost / benefit tradeoffs
- These two sections are entirely at the "design abstraction level" from chapter 2
- Graphs appear in many situations in software
  - As discussed in the rest of chapter 7