CSc 179 – Graph Coverage for Source Code Data Flow Testing

Credits:

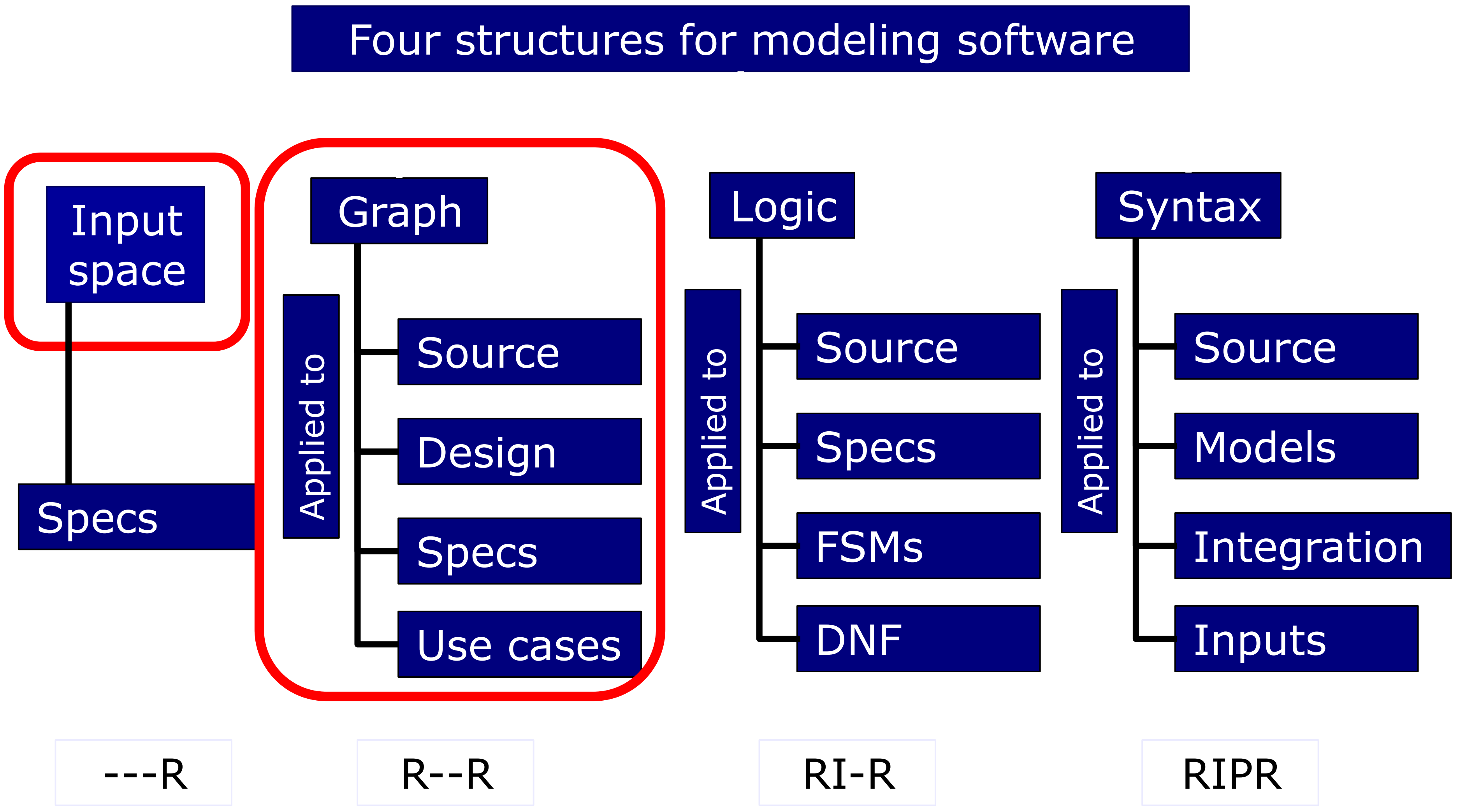
AO – Ammann and Offutt, "Introduction to Software Testing," Ch. 7

Stephen M. Thebaut, University of Florida

Naik & Tripathy, Software Testing and Quality Assurance

University of Waterloo (Arie Gurfinkel – Winter 2018)

Structures for Criteria-Based Testing



Data flow Testing

- Data flow testing focuses on the points at which variables receive values and the points at which these values are used (or referenced). It detects improper use of data values (data flow anomalies) due to coding errors.

- Rapps and Weyuker ‘s after their paper “Data flow analysis techniques for test data selection”

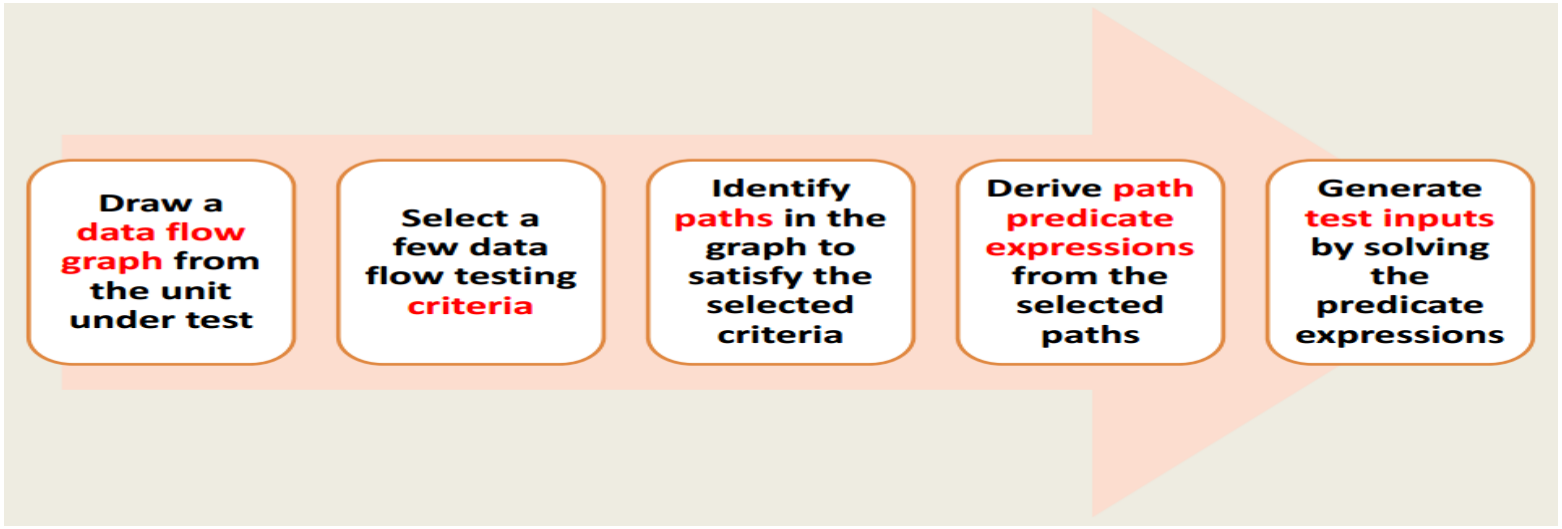
Credits: Stephen M. Thebaut, University of Florida
Naik & Tripathy, Software Testing and Quality Assurance

The General Idea

- Data flow testing can be performed at two conceptual levels.
  - Static data flow testing
  - Dynamic data flow testing
- Static data flow testing (Compile time, inspection)
  - Identify potential defects, commonly known as **data flow anomaly.**
  - Analyze source code.
  - Do not execute code.
  - 3 Types: Use before define, Define but never use, Redefined before use
- Dynamic data flow testing
  - Involves actual program execution.
  - Bears similarity with control flow testing.
    - Identify paths to execute them.
    - Paths are identified based on **data flow testing criteria**.

Overview of Dynamic Data Flow Testing

- Data flow testing is outlined as follows:
  - Draw a data flow graph from a program.
  - Select one or more data flow testing criteria.
  - Identify paths in the data flow graph satisfying the selection criteria.
  - Derive path predicate expressions from the selected paths
  - Solve the path predicate expressions to derive test inputs


Draw a data flow graph from the unit under test
Select a few data flow testing criteria
Identify paths in the graph to satisfy the selected criteria
Derive path predicate expressions from the selected paths
Generate test inputs by solving the predicate expressions

Dataflow Coverage

- Basic idea:
  - Program paths along which variables are defined and then used should be covered
- A family of path selection criteria has been defined, each providing a different degree of coverage
  - All DU pairs, All DU paths, All Defs

Variable Definition

- def : a location where a value is **stored** into memory
  - x appears on the left side of an assignment (i.e. x = 44;)
  - x is an actual parameter in a call and the method changes its value
  - x is a formal parameter of a method (implicit def when method starts)
  - x is an input to a program (i.e. Read(x))

Variable Use

- A program variable is **USED** when it appears:
  - x appears on the right side of an assignment (i.e. `y = x+17`)
  - x appears in a conditional test (i.e. `if ( x > 0 ) { … } )`
  - x is an actual parameter to a method
  - x is an output of the program
  - x is an output of a method in a return statement

Variable Use: p-use and c-use

- Use in the predicate of a branch statement is a *predicate-use* **or "p-use"**
- Any other use is a *computation-use* **or "c-use"**
- For example, in the program fragment:

```
if ( x > 0 ) {
      print(y);
}
```

there is a p-use of **x** and a c-use of **y**

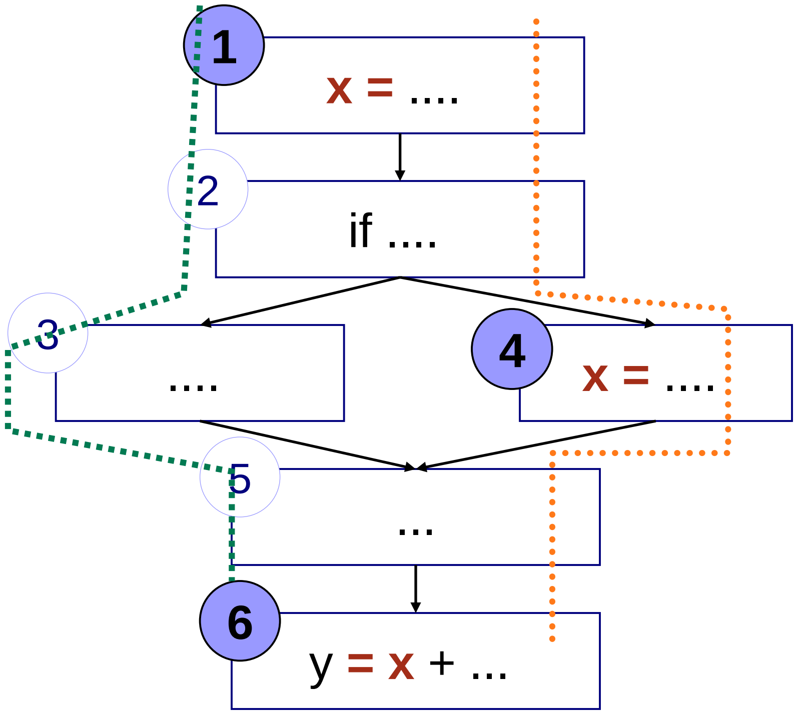
Variable Use

- A variable can also be used and then re-defined in a single statement when it appears:
  - on *both* sides of an assignment statement *eg* `y = y + x`
  - as an call-by-reference parameter in a subroutine call
    eg `increment( &y )`

More Dataflow Terms and Definitions

- A path is *definition clear* **("def-clear")** with respect to a variable **`v`** if it has no variable **re-**definition of **`v`** on the path
- A *complete path* is a path whose initial node is a start node and whose final node is an exit node

Recap: Definition-clear path



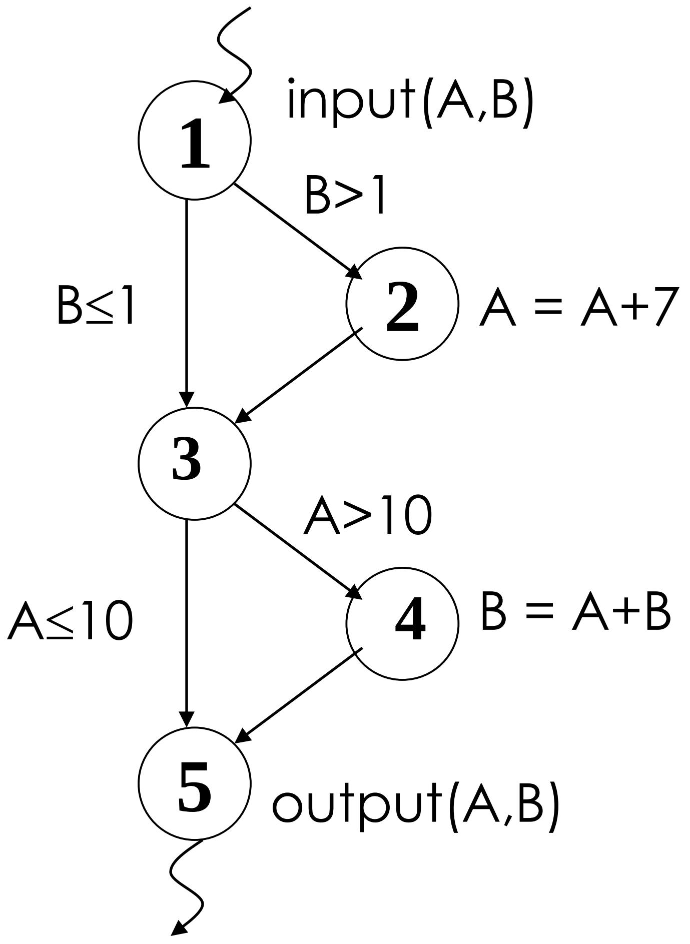
- 1,2,3,5,6 is a definition-clear path from 1 to 6
  - x is not re-assigned between 1 and 6
- 1,2,4,5,6 is not a definition-clear path from 1 to 6
  - the value of x is “killed” (reassigned) at node 4
- (1,6) is a **DU pair** because 1,2,3,5,6 is a definition-clear path

Data Flow Graph (DFG)

- A data flow graph (DFG) captures the flow of data in a program
- DFG is a directed graph constructed as follows:
  - to build a DFG, we first build a CFG and then annotate with:
  - a sequence of **definitions** and **c-uses** is associated with each **node**
  - a set of **p-uses** is associated with each edge

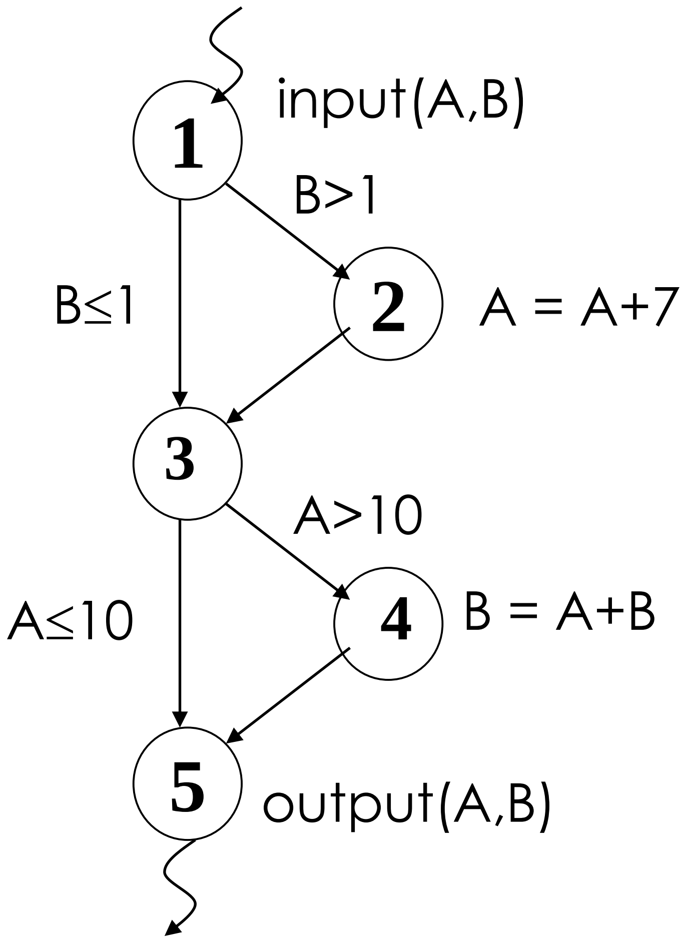
Example 1

```
1.  input(A,B)
    if (B>1) {
2.        A = A+7
    }
3.  if (A>10) {
4.        B = A+B
     }
5.  output(A,B)
```



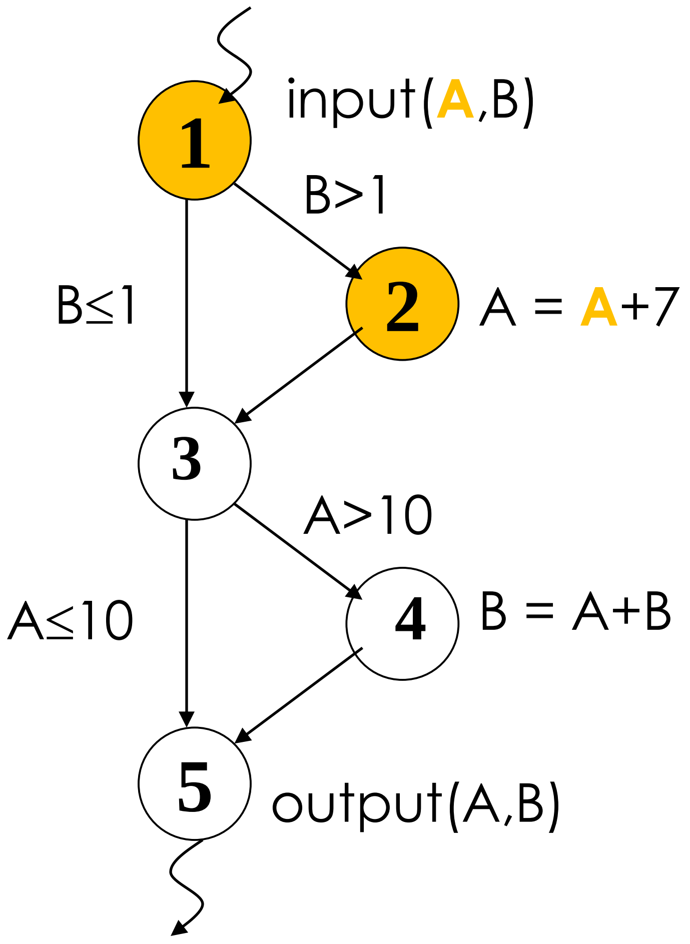
Identifying DU-Pairs – Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



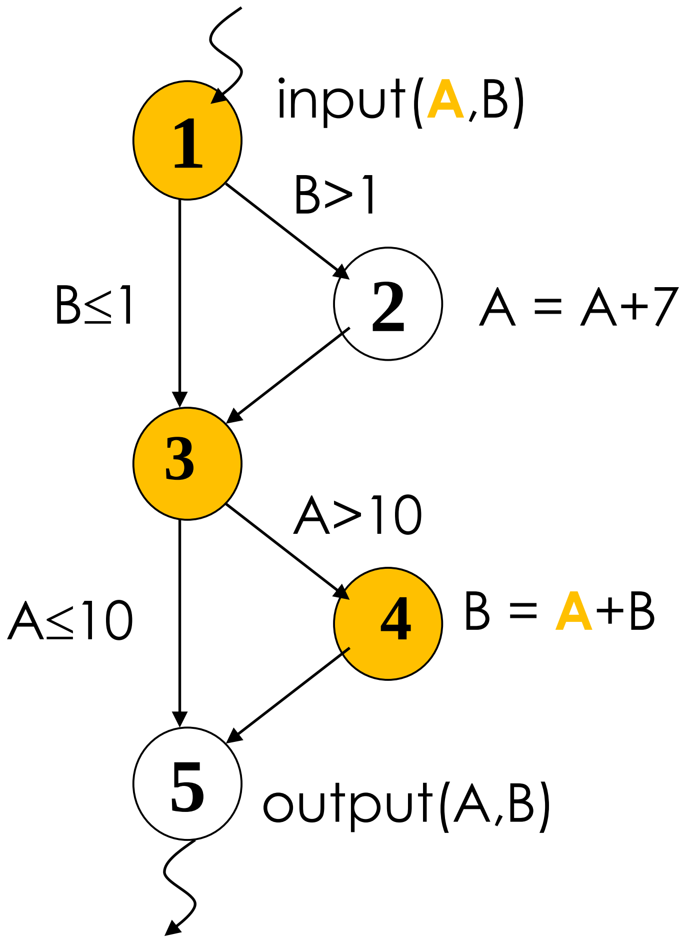
Identifying DU-Pairs – Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



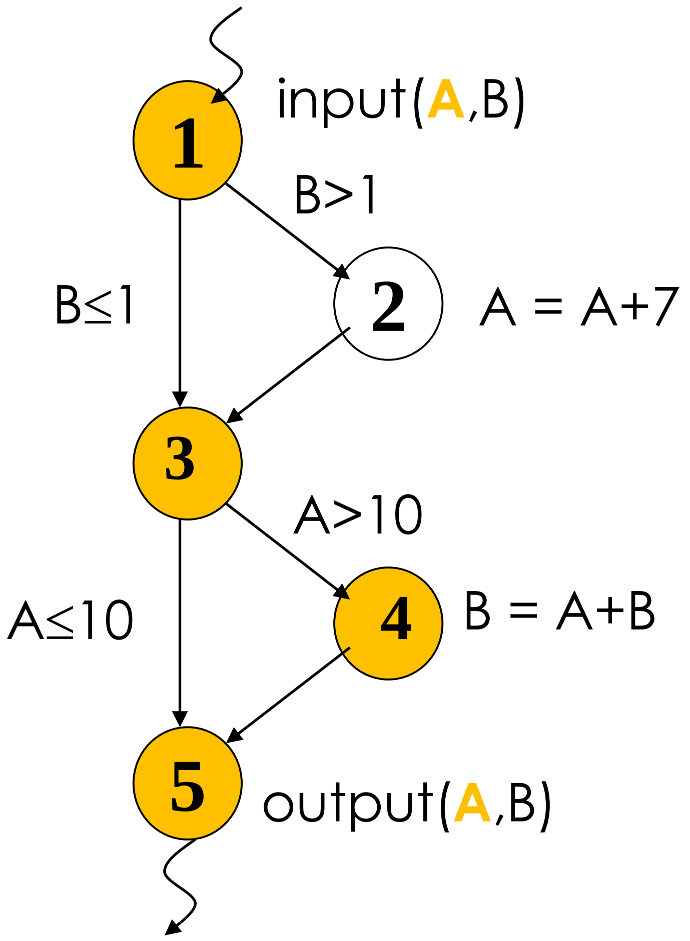
Identifying DU-Pairs – Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	**<1,3,4>**
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



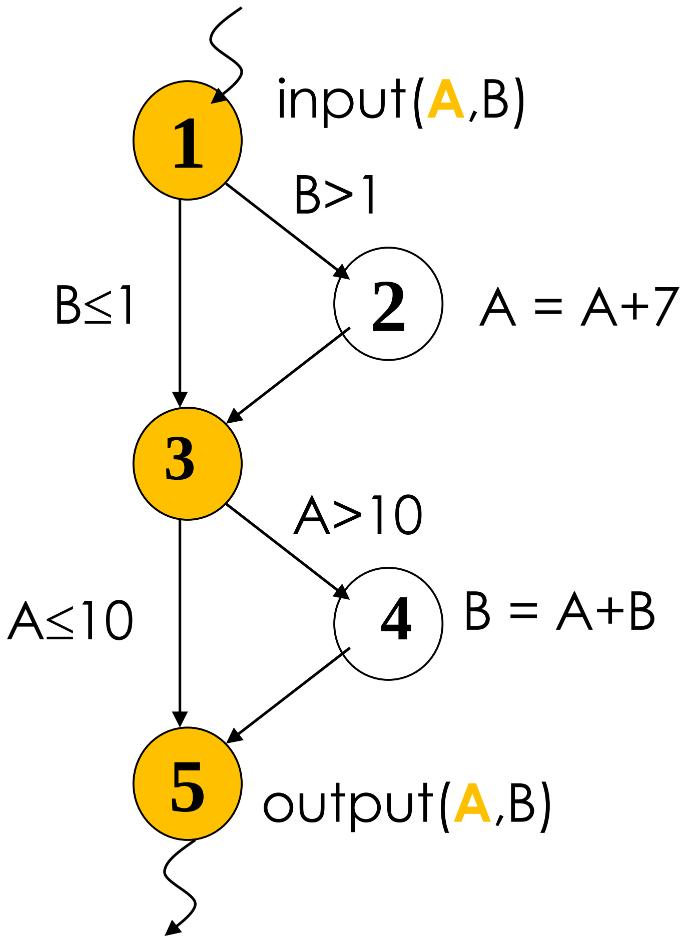
Identifying DU-Pairs – Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	**<1,3,4,5>**
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



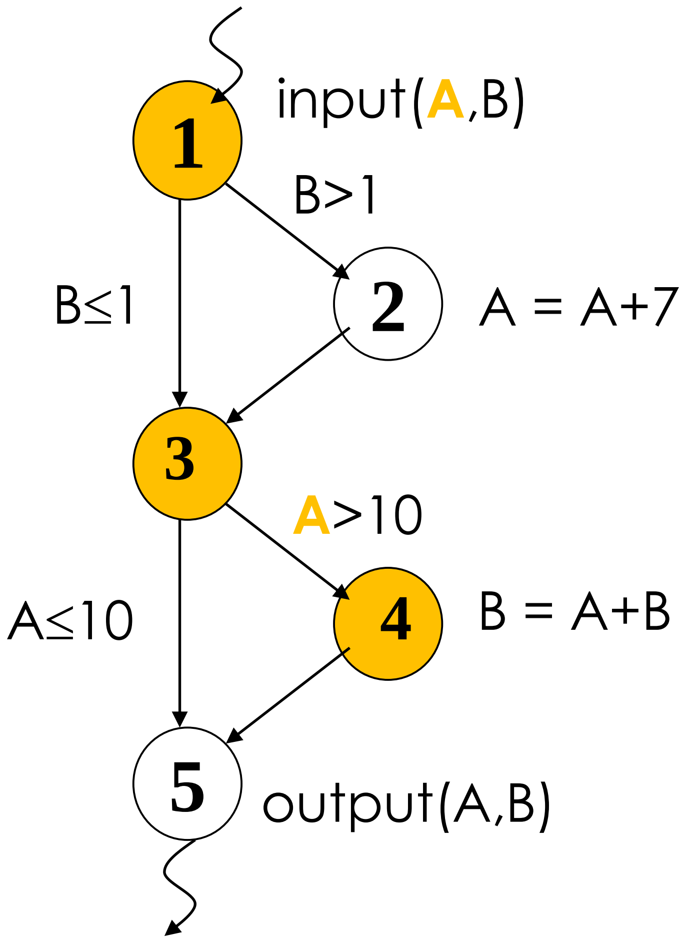
Identifying DU-Pairs – Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



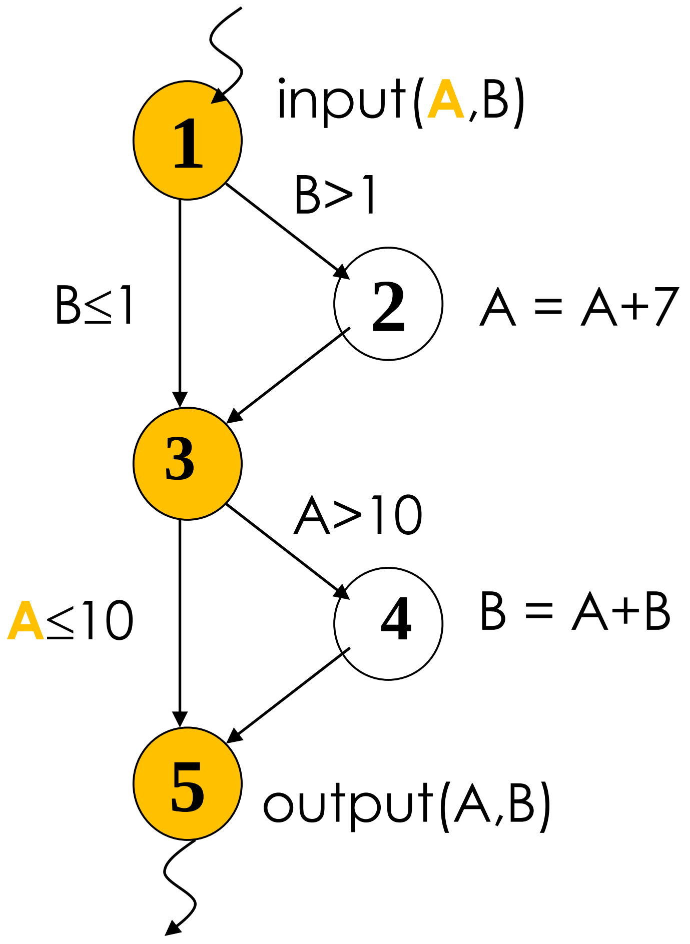
Identifying DU-Pairs – Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	**<1,3,4>**
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



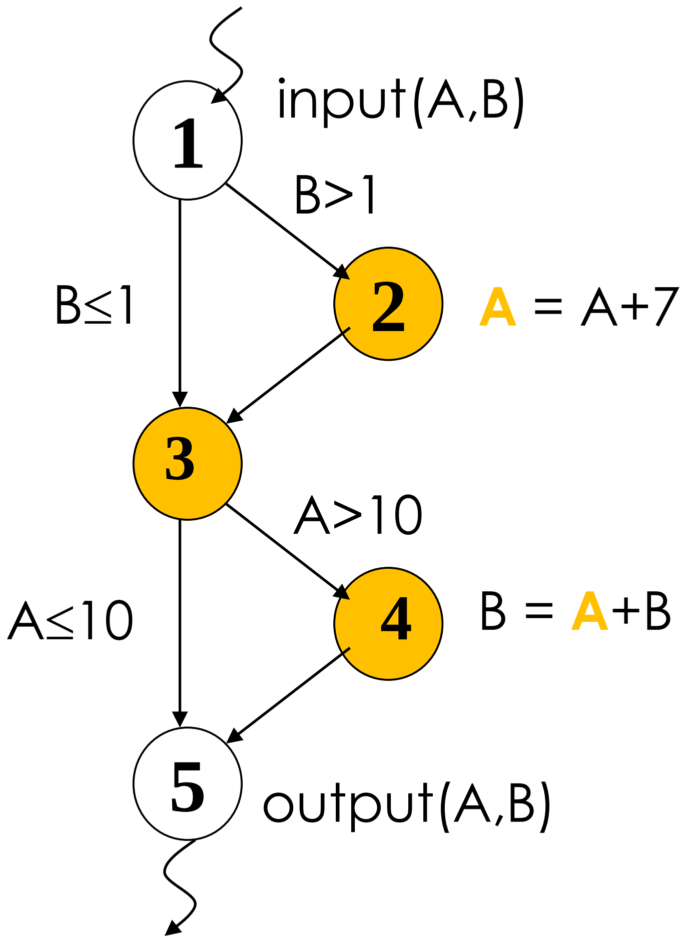
Identifying DU-Pairs – Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



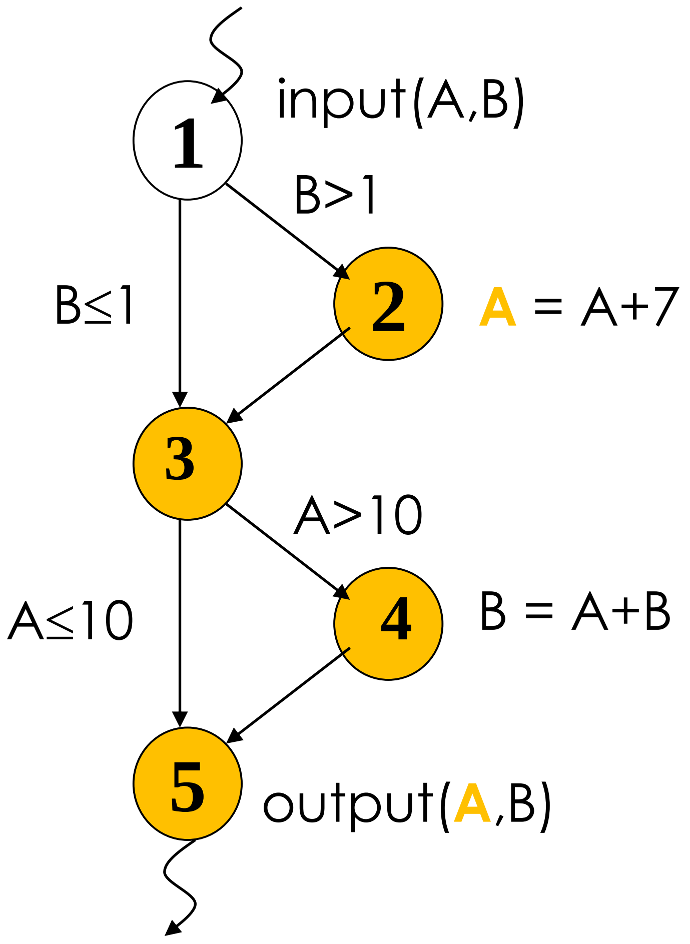
Identifying DU-Pairs – Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	**<2,3,4>**
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



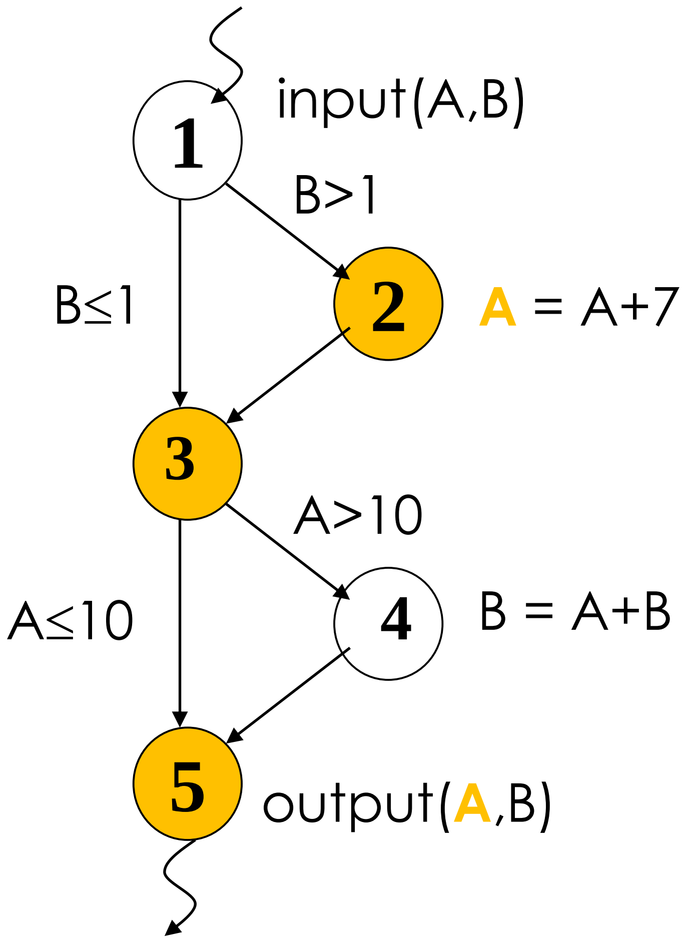
Identifying DU-Pairs – Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	**<2,3,4,5>**
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



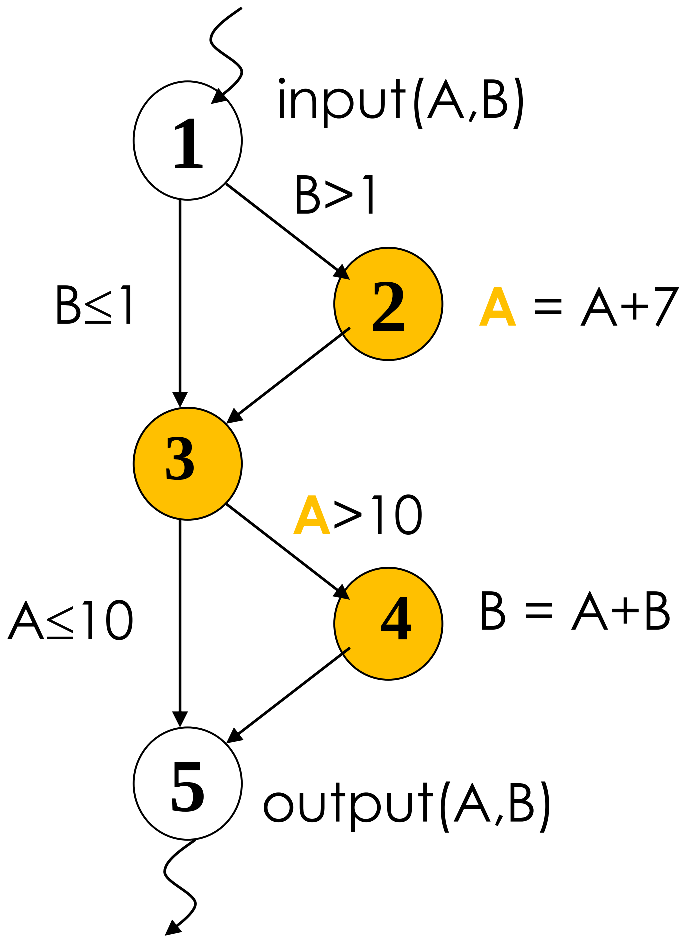
Identifying DU-Pairs – Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



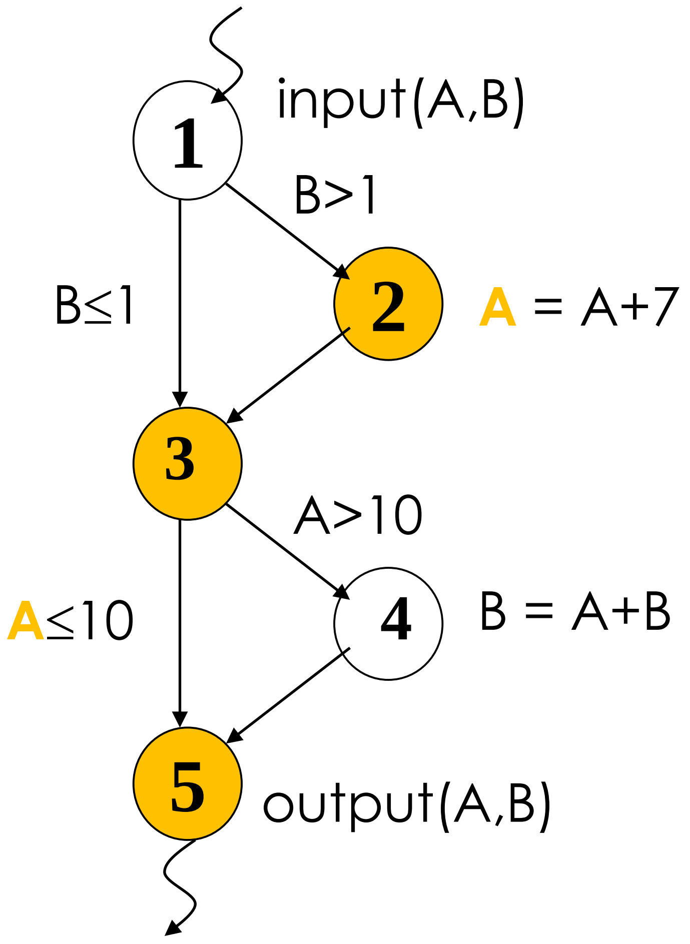
Identifying DU-Pairs – Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	**<2,3,4>**
(2,<3,5>)	<2,3,5>



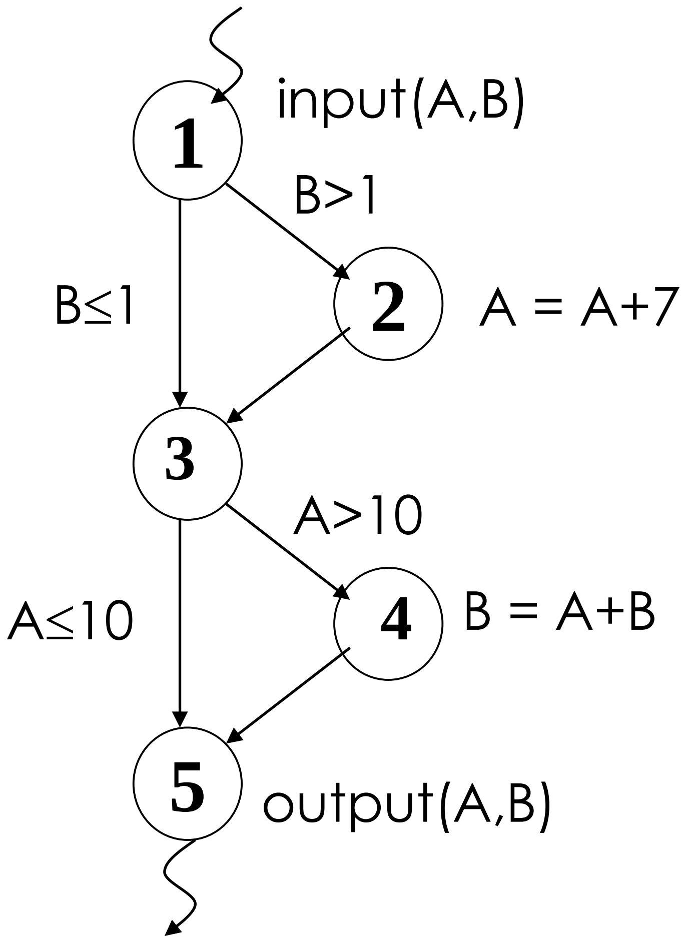
Identifying DU-Pairs – Variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



Identifying DU-Pairs – Variable B

du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,3,4>
(1,5)	<1,2,3,5>
	<1,3,5>
(1,<1,2>)	<1,2>
(1,<1,3>)	<1,3>
(4,5)	<4,5>



Dataflow Test Coverage Criteria

- ***All-Defs***
  for **every program variable `v`, at least one def-clear path** from **every definition** of **`v`** to **at least one c-use or one p-use** of **`v`** must be covered

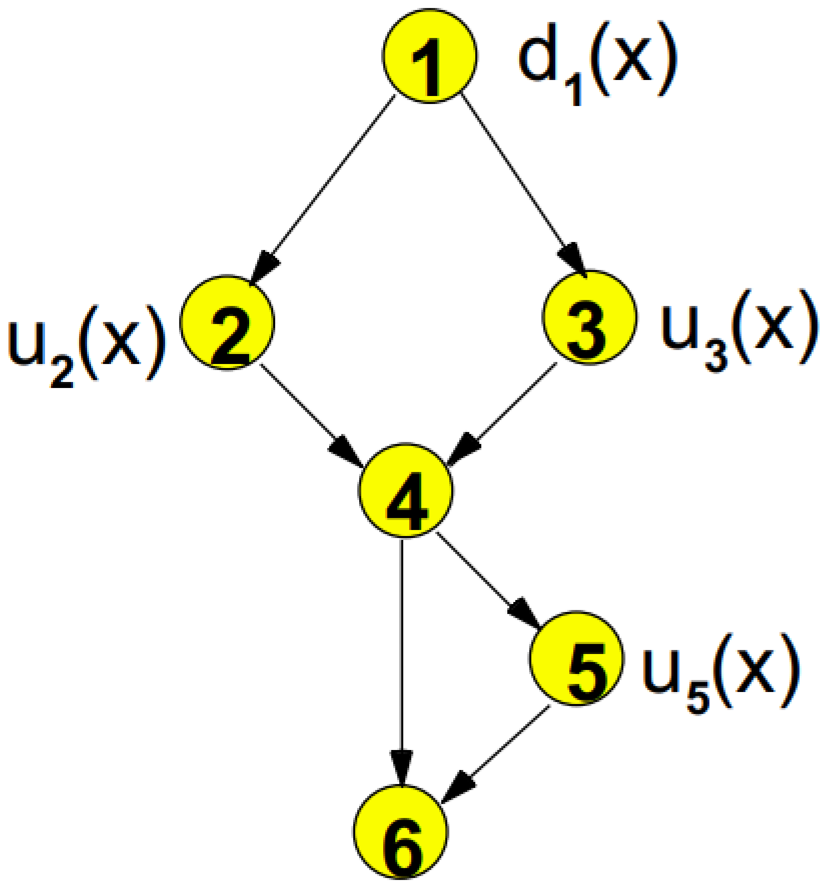
All-Defs Coverage: Example

Requires:

$d_1(x)$ to a use

Satisfactory Path:

[1, 2, 4, 6]



Credit: University of Waterloo (Arie Gurfinkel – Winter 2018)

Dataflow Test Coverage Criteria

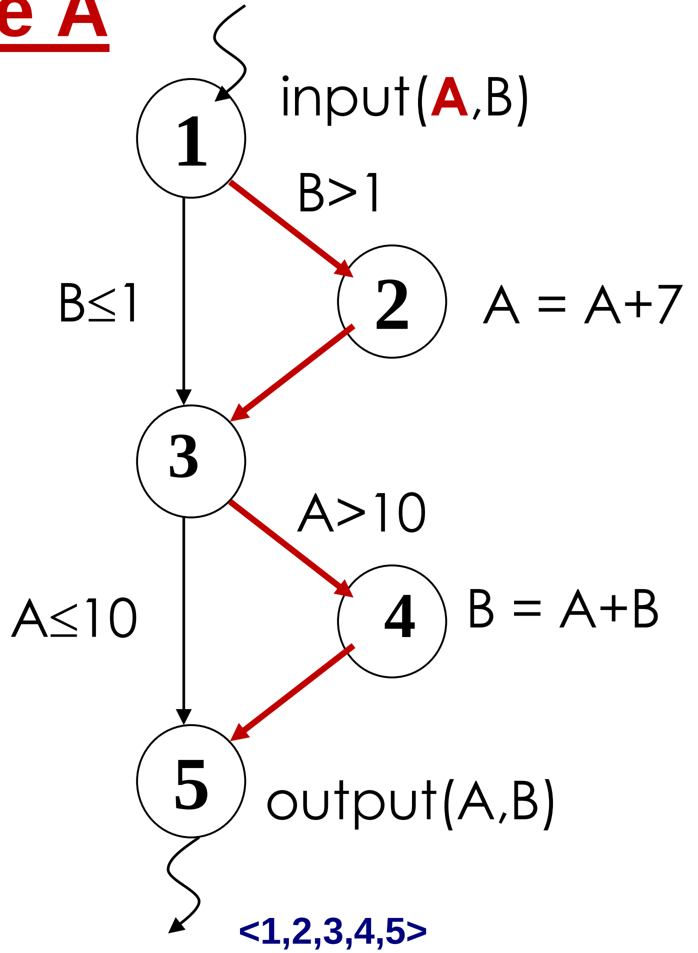
- Consider a test case executing path:

  <1,2,3,4,5>

- Identify all def-clear paths covered (*ie* subsumed) by this path for each variable
- Are all definitions for each variable associated with at least one of the subsumed def-clear paths?

Def-Clear Paths subsumed by <1,2,3,4,5> for Variable A

du-pair	**path(s)**
(1,2)	<1,2> ✓
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4> ✓
(2,5)	<2,3,4,5> ✓
	<2,3,5>
(2,<3,4>)	<2,3,4> ✓
(2,<3,5>)	<2,3,5>



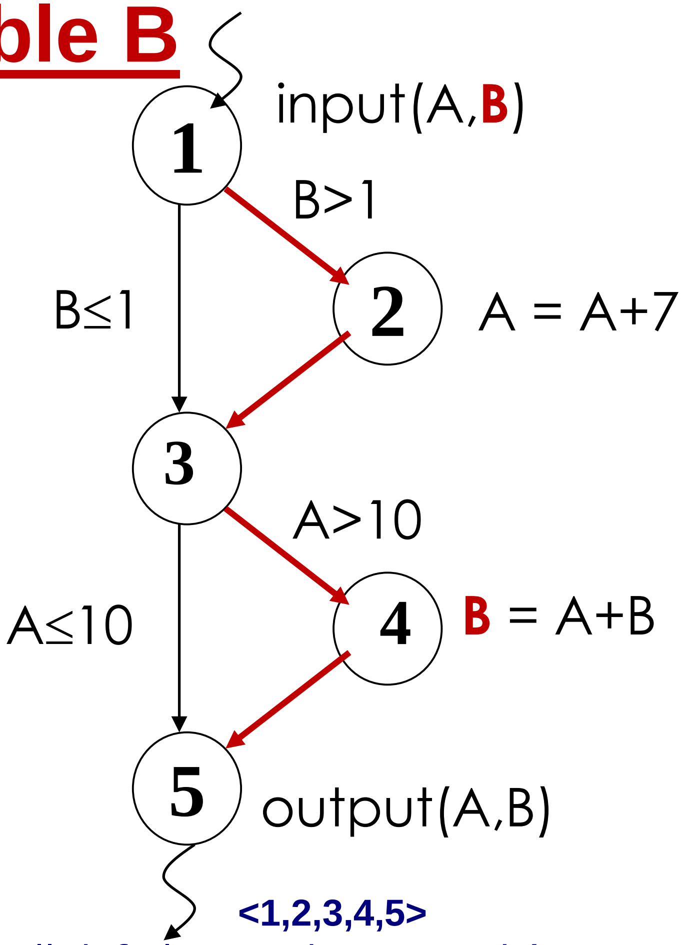
<1,2,3,4,5>

Identify all def-clear paths covered (*ie* subsumed) by this path for each variable

Are all definitions for each variable associated with at least one of the subsumed def-clear paths?

Def-Clear Paths Subsumed by <1,2,3,4,5> for Variable B

du-pair	path(s)
(1,4)	<1,2,3,4> ✓
	<1,3,4>
(1,5)	<1,2,3,5>
	<1,3,5>
(4,5)	<4,5> ✓
(1,<1,2>)	<1,2> ✓
(1,<1,3>)	<1,3>



Identify all def-clear paths covered (*ie* subsumed) by this path for each variable

Are all definitions for each variable associated with at least one of the subsumed def-clear paths?

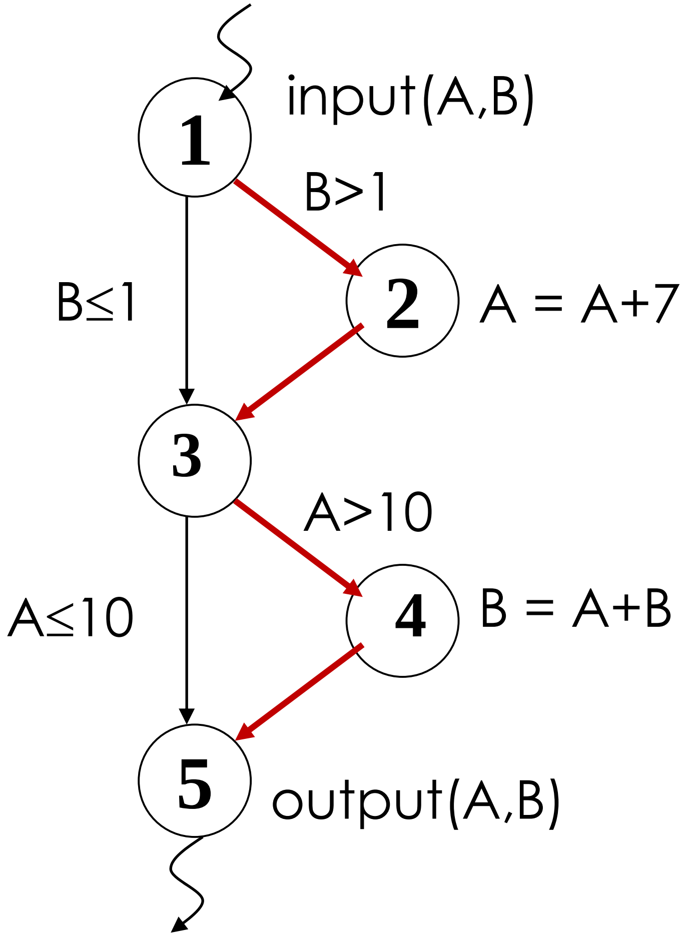
Dataflow Test Coverage Criteria

- Since **<1,2,3,4,5>** covers at least one def-clear path from every definition of A or B to at least one c-use or p-use of A or B, All-Defs coverage is achieved
- Predicate Expression:

  B > 1 & A > 10
- Test Case =

  I = { B = 4 , A = 20 }

  Expected O = { B = 31, A = 27 }



Dataflow Test Coverage Criteria

- ***All-Uses:***
  for **every program variable v, at least one def-clear path** from **every definition** of v to **every c-use and every p-use** of v must be covered

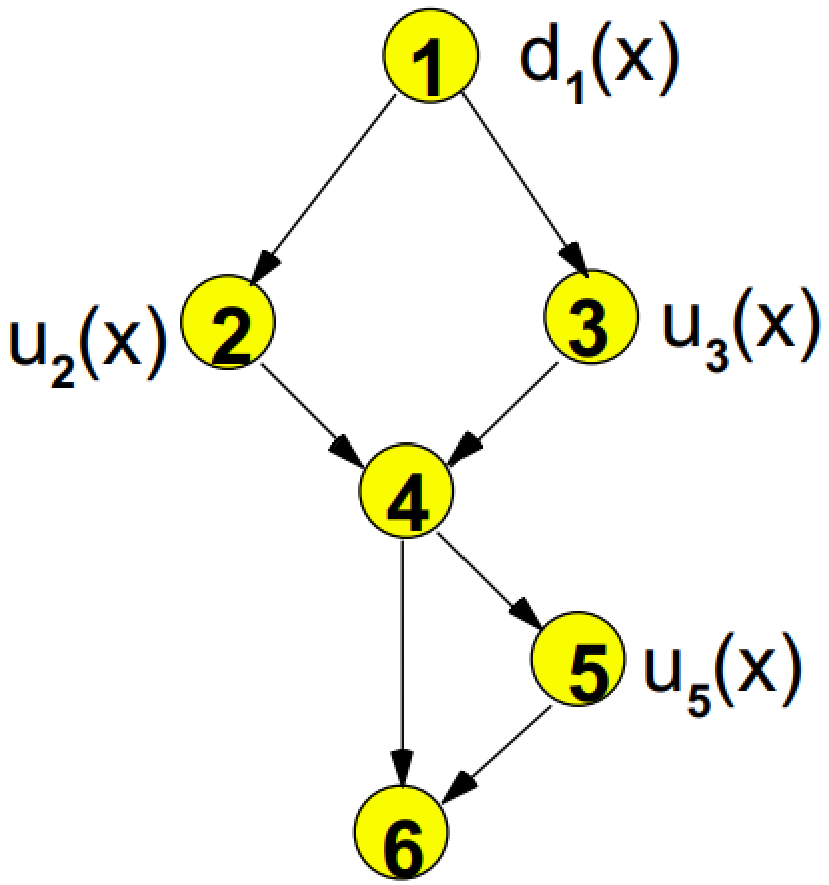
All-Uses Coverage: Example

Requires:

- d1(x) to u2(x)
- d1(x) to u3(x)
- d1(x) to u5(x)

Satisfactory Paths:

- [1, 2, 4, 5, 6]
- [1, 3, 4, 6]



Dataflow Test Coverage Criteria (Cont)

- ***All-Uses:*** for **every program variable v, at least one def-clear path** from **every definition** of v to **every c-use** and **every p-use** of v must be covered

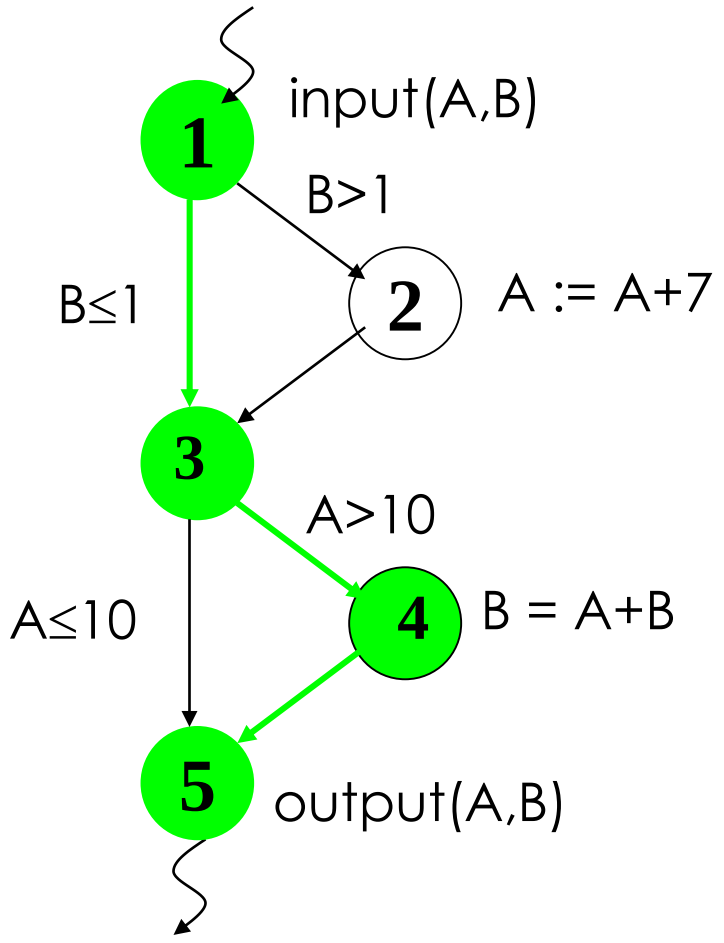
- Consider additional test cases executing paths:

  2. **<1,3,4,5>**
  3. **<1,2,3,5>**

  (note: 1. <1,2,3,4,5> execution path was shown earlier)
- Do all three test cases provide All-Uses coverage?

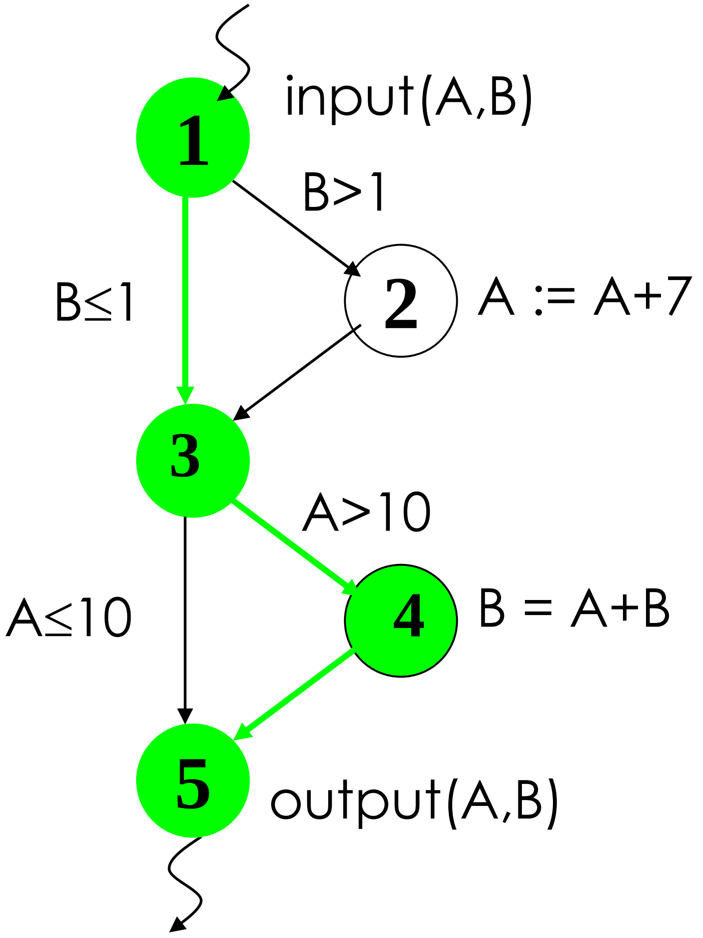
Def-Clear Paths Subsumed by <1,3,4,5> for Variable A

du-pair	path(s)
(1,2)	<1,2> ✓
(1,4)	<1,3,4> ✓
(1,5)	<1,3,4,5> ✓
	<1,3,5>
(1,<3,4>)	<1,3,4> ✓
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4> ✓
(2,5)	<2,3,4,5> ✓
	<2,3,5>
(2,<3,4>)	<2,3,4> ✓
(2,<3,5>)	<2,3,5>



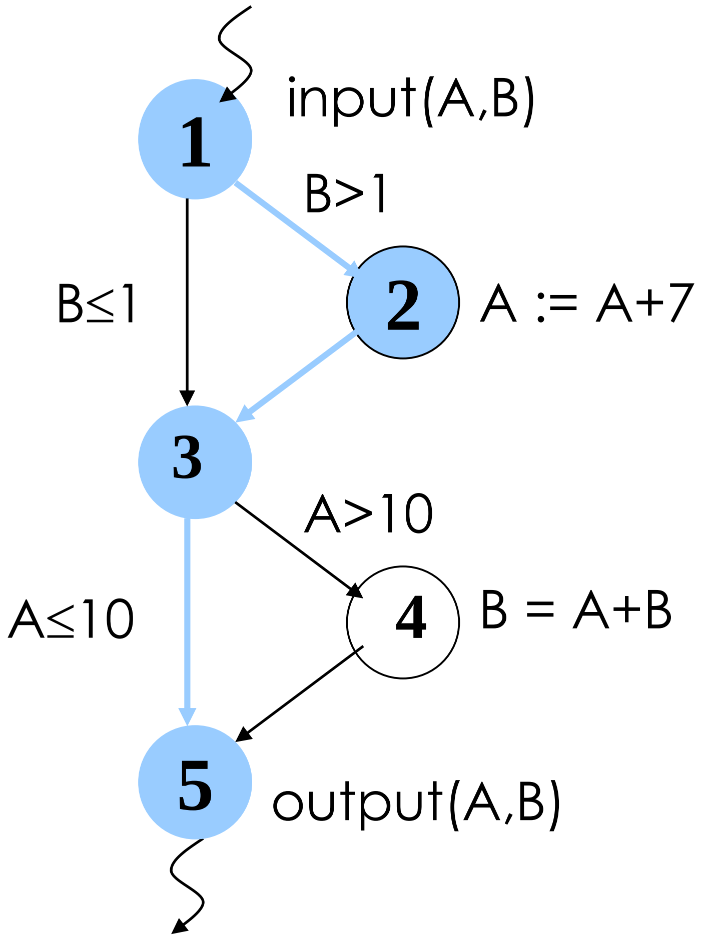
Def-Clear Paths Subsumed by <1,3,4,5> for Variable B

du-pair	path(s)
(1,4)	<1,2,3,4> ✓
	<1,3,4> ✓
(1,5)	<1,2,3,5>
	<1,3,5>
(4,5)	<4,5> ✓ ✓
(1,<1,2>)	<1,2> ✓
(1,<1,3>)	<1,3> ✓



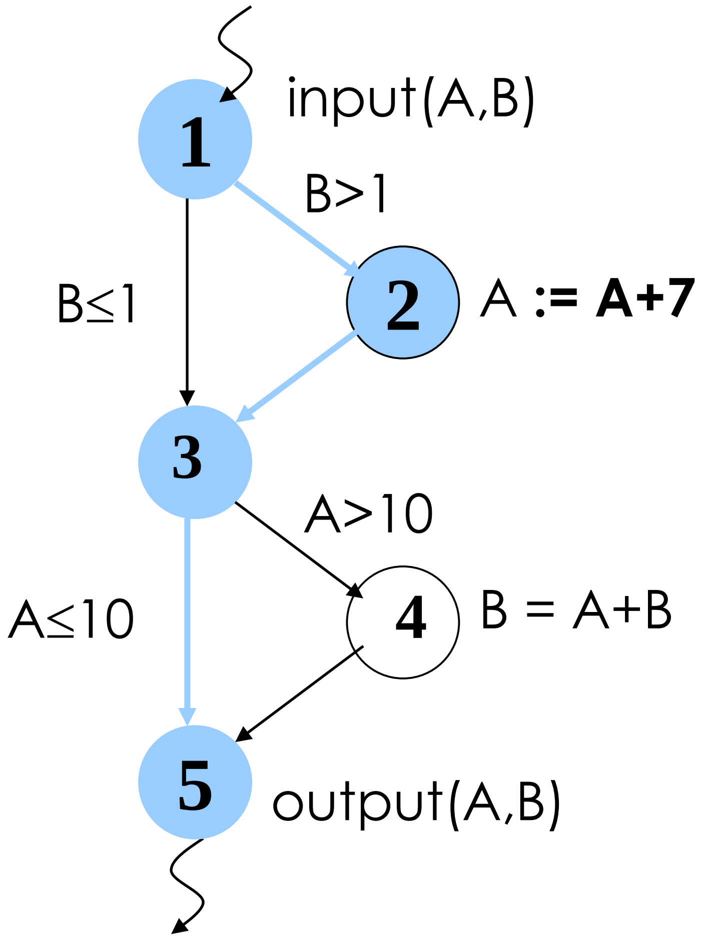
Def-Clear Paths Subsumed by <1,2,3,5> for Variable A

du-pair	path(s)
(1,2)	<1,2> ✓ ✓
(1,4)	<1,3,4> ✓
(1,5)	<1,3,4,5> ✓
	<1,3,5>
(1,<3,4>)	<1,3,4> ✓
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4> ✓
(2,5)	<2,3,4,5> ✓
	<2,3,5> ✓
(2,<3,4>)	<2,3,4> ✓
(2,<3,5>)	<2,3,5> ✓



Def-Clear Paths Subsumed by <1,2,3,5> for Variable B

du-pair	path(s)
(1,4)	<1,2,3,4> ✓
	<1,3,4> ✓
(1,5)	<1,2,3,5> ✓
	<1,3,5>
(4,5)	<4,5> ✓ ✓
(1,<1,2>)	<1,2> ✓ ✓
(1,<1,3>)	<1,3> ✓



Dataflow Test Coverage Criteria

- None of the **three test cases** covers the du-pair (1,<3,5>) for variable A.

∴ All-Uses Coverage is not achieved

More Dataflow Terms and Definitions

- A path p is **simple** if it has no repetitions of nodes other than (possibly) the first and last node.

- A **du-path** for a variable **v** is a simple path from a definition of **v** to a use of **v** that is def-clear w.r.t. **v**

Another Dataflow Test Coverage Criterion

- ***All-DU-Paths:***
  for **every program variable v, every du-path** from **every definition** of v to **every c-use** and **every p-use** of v must be covered

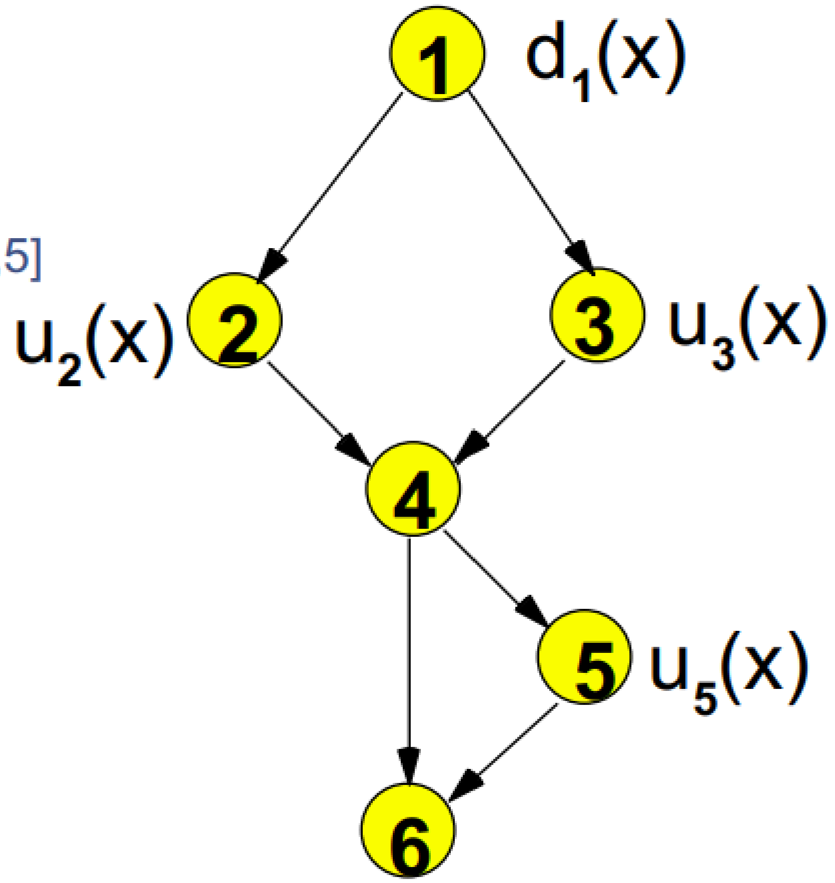
All-Du-Paths Coverage: Example

Requires:

- All d1(x) to u2(x): [1,2]
- All d1(x) to u3(x): [1,3]
- All d1(x) to u5(x): [1,2,4,5], [1,3,4,5]

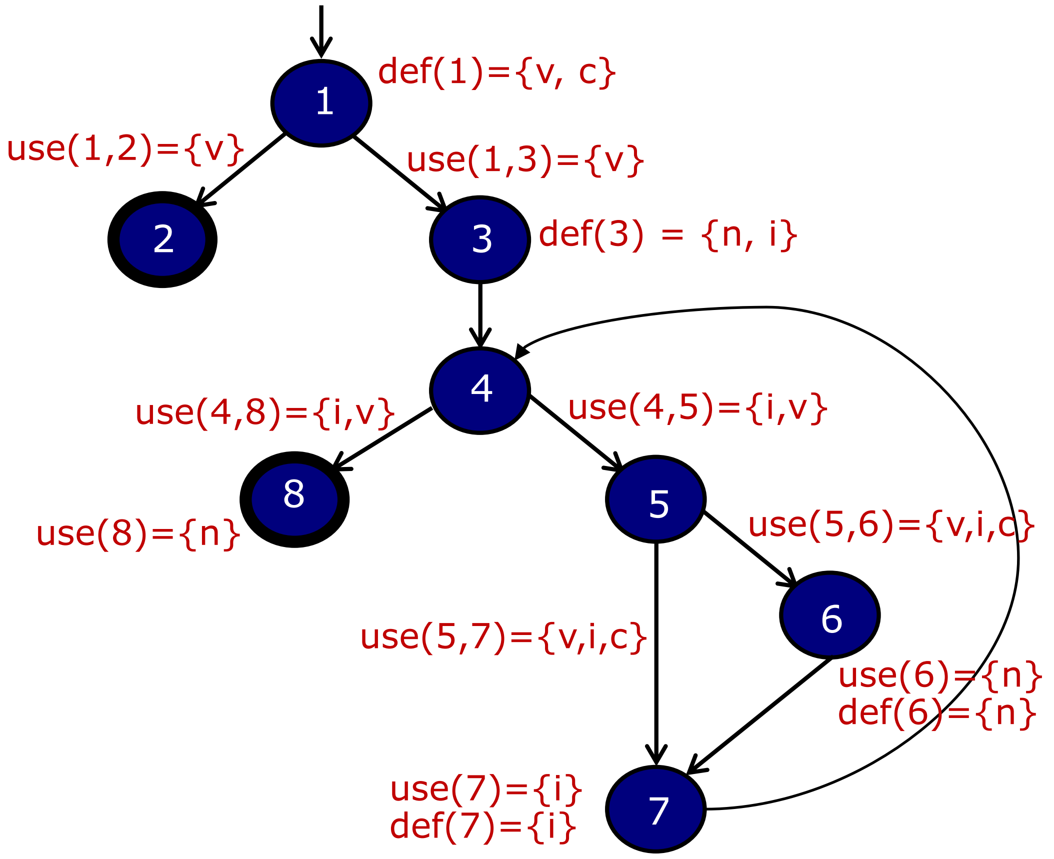
Satisfactory Paths:

- [1, 2, 4, 5, 6]
- [1, 3, 4 ,5, 6]



Applying Data Flow Coverage (Earlier Example)

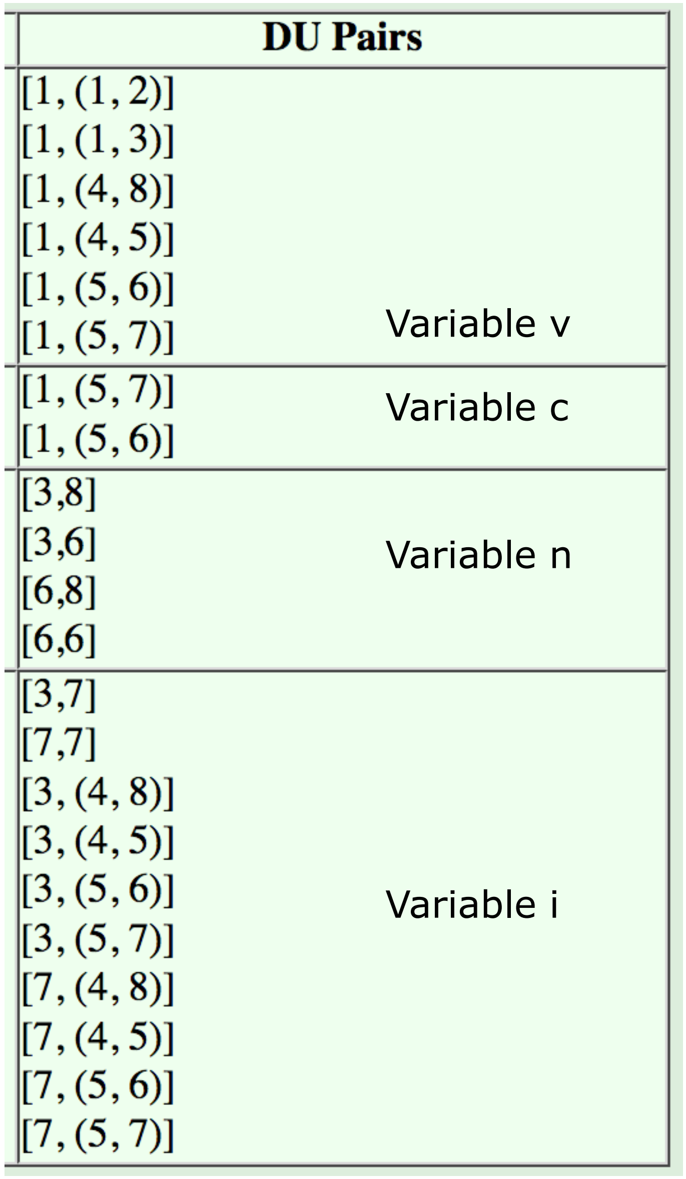
v and c are forwarded parameters



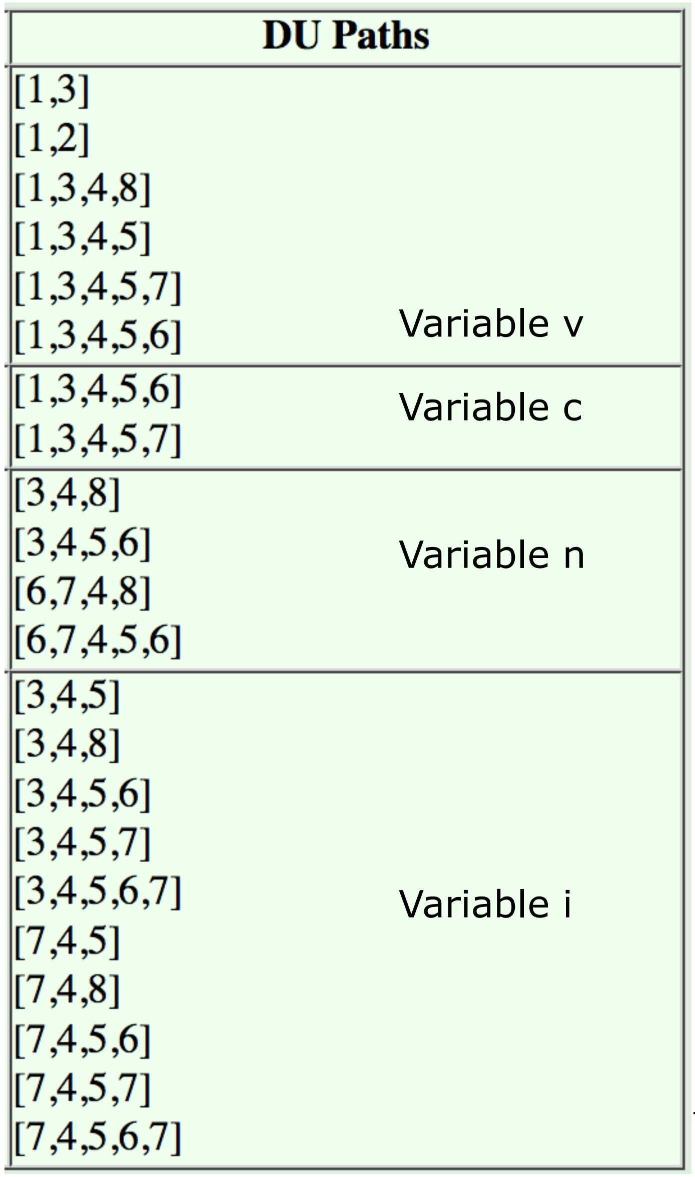
Deriving test requirements

- List all du-pairs
- Based on du-pairs, derive du-paths
- Must be def-clear paths
- All Defs Coverage (ADC)
  - For each def, at least one use must be reached
- All Uses Coverage (AUC)
  - For each def, all uses must be reached
- All DU-Paths Coverage (ADUPC)
  - For each def-use pair, all paths between defs and uses must be covered

DU-Pairs → DU-Paths

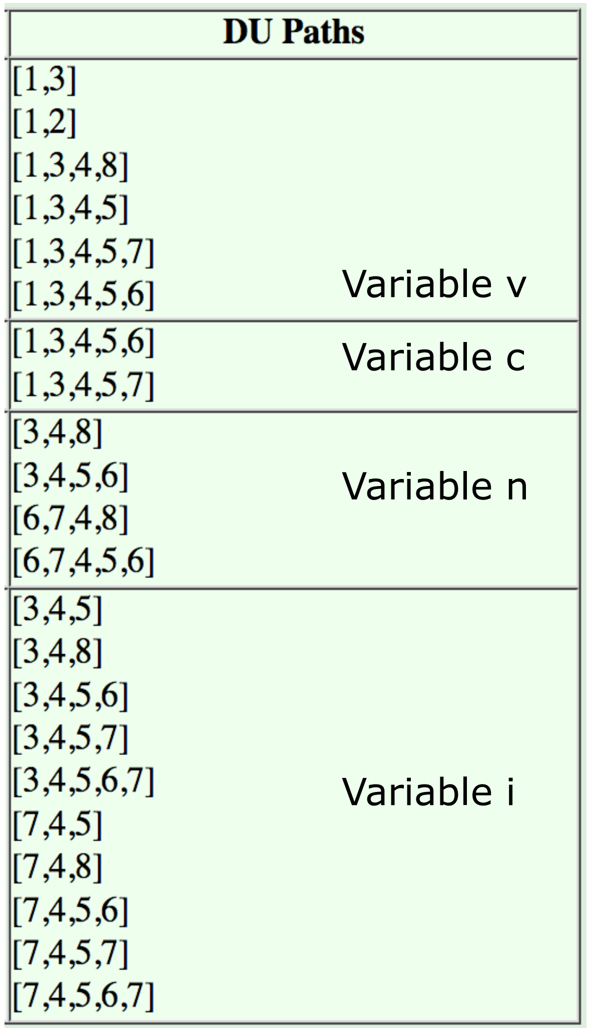


DU Pairs	
[1, (1, 2)]	
[1, (1, 3)]	
[1, (4, 8)]	
[1, (4, 5)]	
[1, (5, 6)]	
[1, (5, 7)]	Variable v
[1, (5, 7)]	Variable c
[1, (5, 6)]	
[3,8]	
[3,6]	Variable n
[6,8]	
[6,6]	
[3,7]	
[7,7]	
[3, (4, 8)]	
[3, (4, 5)]	
[3, (5, 6)]	Variable i
[3, (5, 7)]	
[7, (4, 8)]	
[7, (4, 5)]	
[7, (5, 6)]	
[7, (5, 7)]	



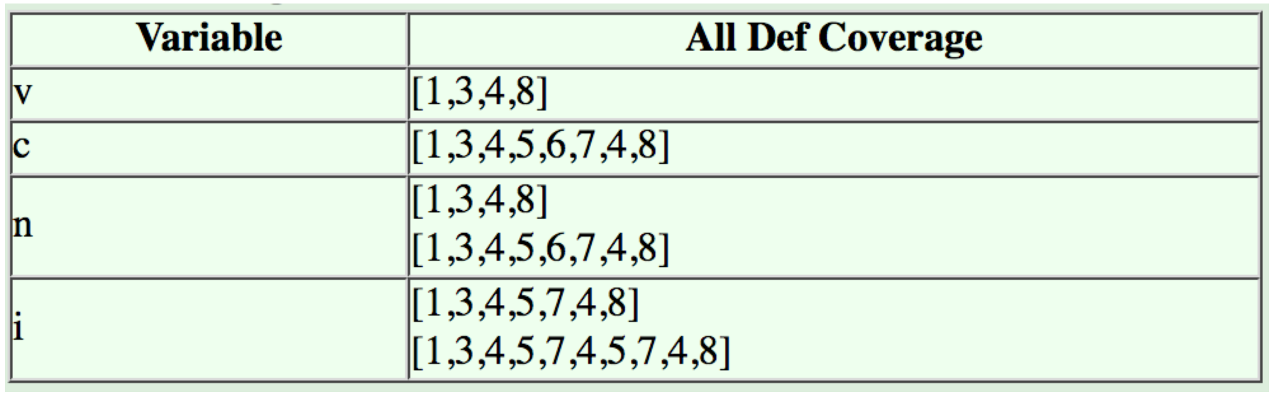
DU Paths	
[1,3]	
[1,2]	
[1,3,4,8]	
[1,3,4,5]	
[1,3,4,5,7]	
[1,3,4,5,6]	Variable v
[1,3,4,5,6]	Variable c
[1,3,4,5,7]	
[3,4,8]	
[3,4,5,6]	Variable n
[6,7,4,8]	
[6,7,4,5,6]	
[3,4,5]	
[3,4,8]	
[3,4,5,6]	
[3,4,5,7]	
[3,4,5,6,7]	Variable i
[7,4,5]	
[7,4,8]	
[7,4,5,6]	
[7,4,5,7]	
[7,4,5,6,7]	

ADC: DU-Paths → Test Paths



DU Paths	
[1,3]	
[1,2]	
[1,3,4,8]	
[1,3,4,5]	
[1,3,4,5,7]	
[1,3,4,5,6]	Variable v
[1,3,4,5,6]	Variable c
[1,3,4,5,7]	
[3,4,8]	
[3,4,5,6]	Variable n
[6,7,4,8]	
[6,7,4,5,6]	
[3,4,5]	
[3,4,8]	
[3,4,5,6]	
[3,4,5,7]	
[3,4,5,6,7]	
[7,4,5]	Variable i
[7,4,8]	
[7,4,5,6]	
[7,4,5,7]	
[7,4,5,6,7]	

Test paths that satisfy All Defs Coverage

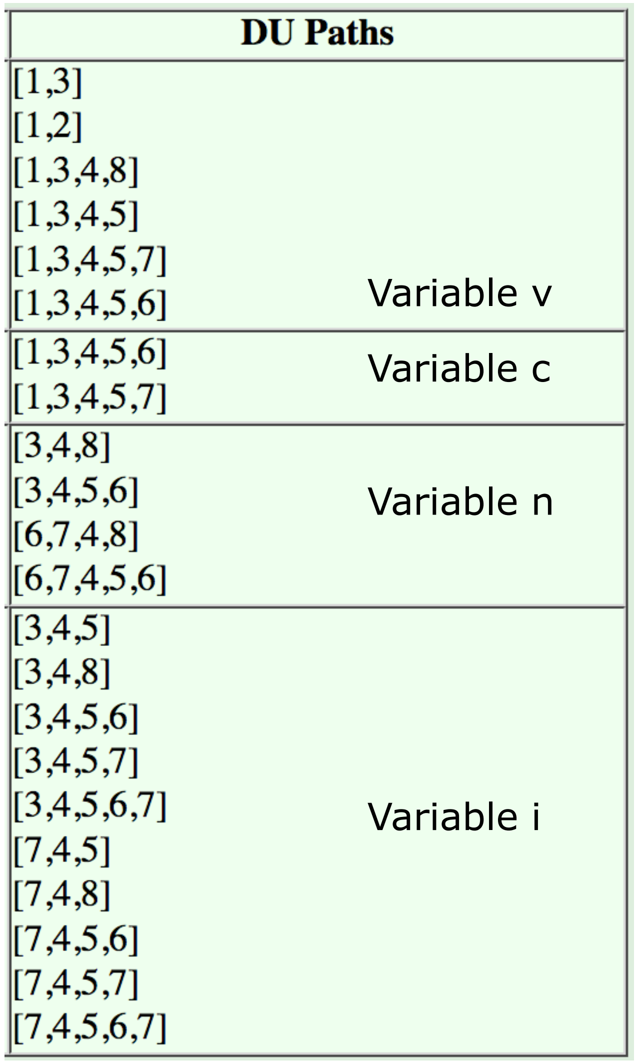


Variable	All Def Coverage
v	[1,3,4,8]
c	[1,3,4,5,6,7,4,8]
n	[1,3,4,8] [1,3,4,5,6,7,4,8]
i	[1,3,4,5,7,4,8] [1,3,4,5,7,4,5,7,4,8]

All Defs Coverage (ADC)

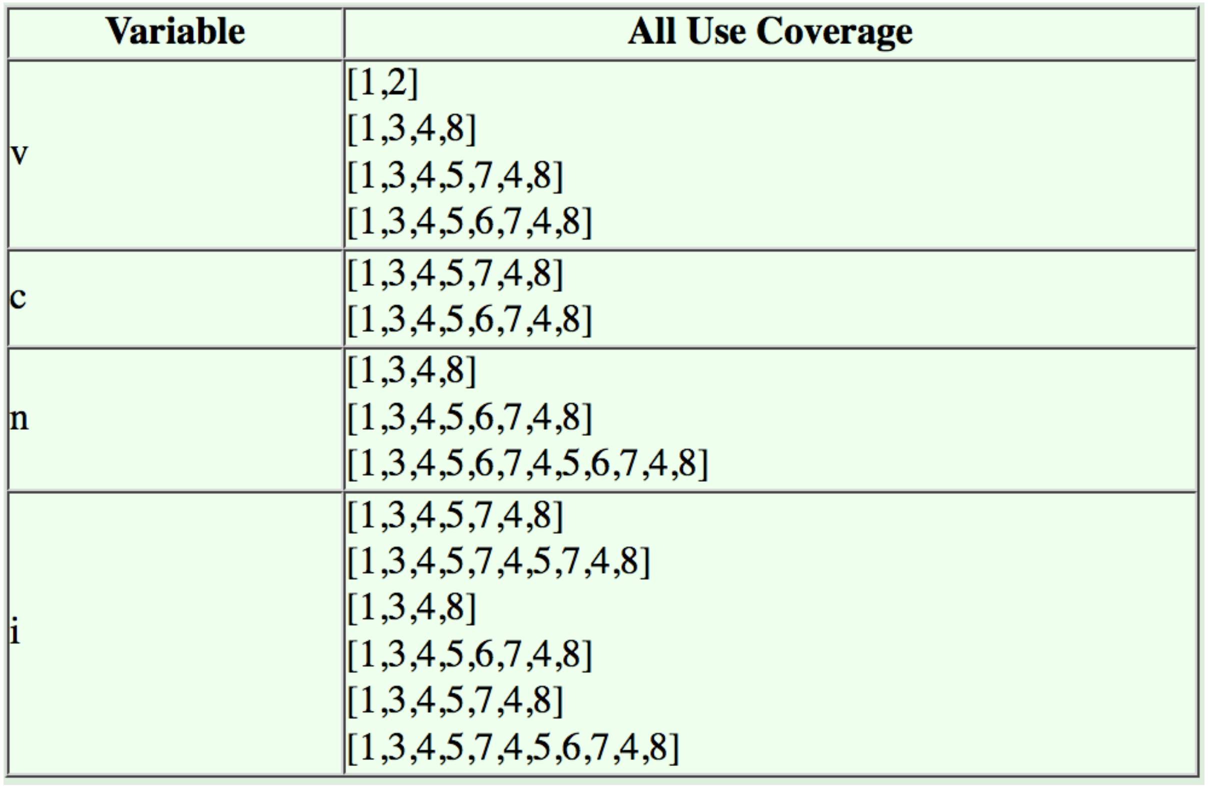
For each def, at least one use must be reached

AUC: DU-Paths → Test Paths



DU Paths	
[1,3] [1,2] [1,3,4,8] [1,3,4,5] [1,3,4,5,7] [1,3,4,5,6]	Variable v
[1,3,4,5,6] [1,3,4,5,7]	Variable c
[3,4,8] [3,4,5,6] [6,7,4,8] [6,7,4,5,6]	Variable n
[3,4,5] [3,4,8] [3,4,5,6] [3,4,5,7] [3,4,5,6,7] [7,4,5] [7,4,8] [7,4,5,6] [7,4,5,7] [7,4,5,6,7]	Variable i

Test paths that satisfy All Uses Coverage

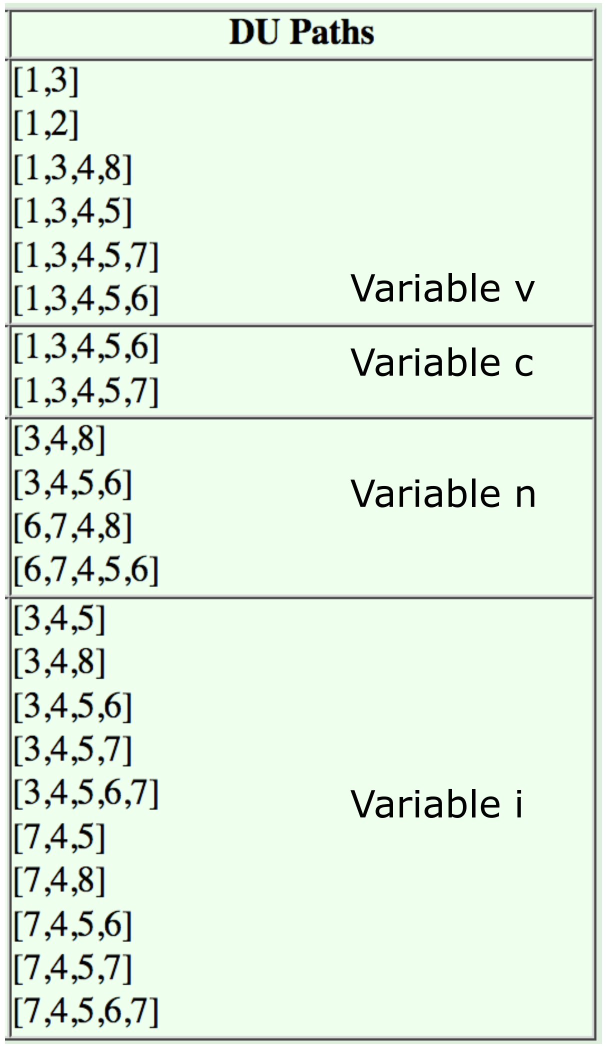


Variable	All Use Coverage
v	[1,2] [1,3,4,8] [1,3,4,5,7,4,8] [1,3,4,5,6,7,4,8]
c	[1,3,4,5,7,4,8] [1,3,4,5,6,7,4,8]
n	[1,3,4,8] [1,3,4,5,6,7,4,8] [1,3,4,5,6,7,4,5,6,7,4,8]
i	[1,3,4,5,7,4,8] [1,3,4,5,7,4,5,7,4,8] [1,3,4,8] [1,3,4,5,6,7,4,8] [1,3,4,5,7,4,8] [1,3,4,5,7,4,5,6,7,4,8]

All Uses Coverage (AUC)

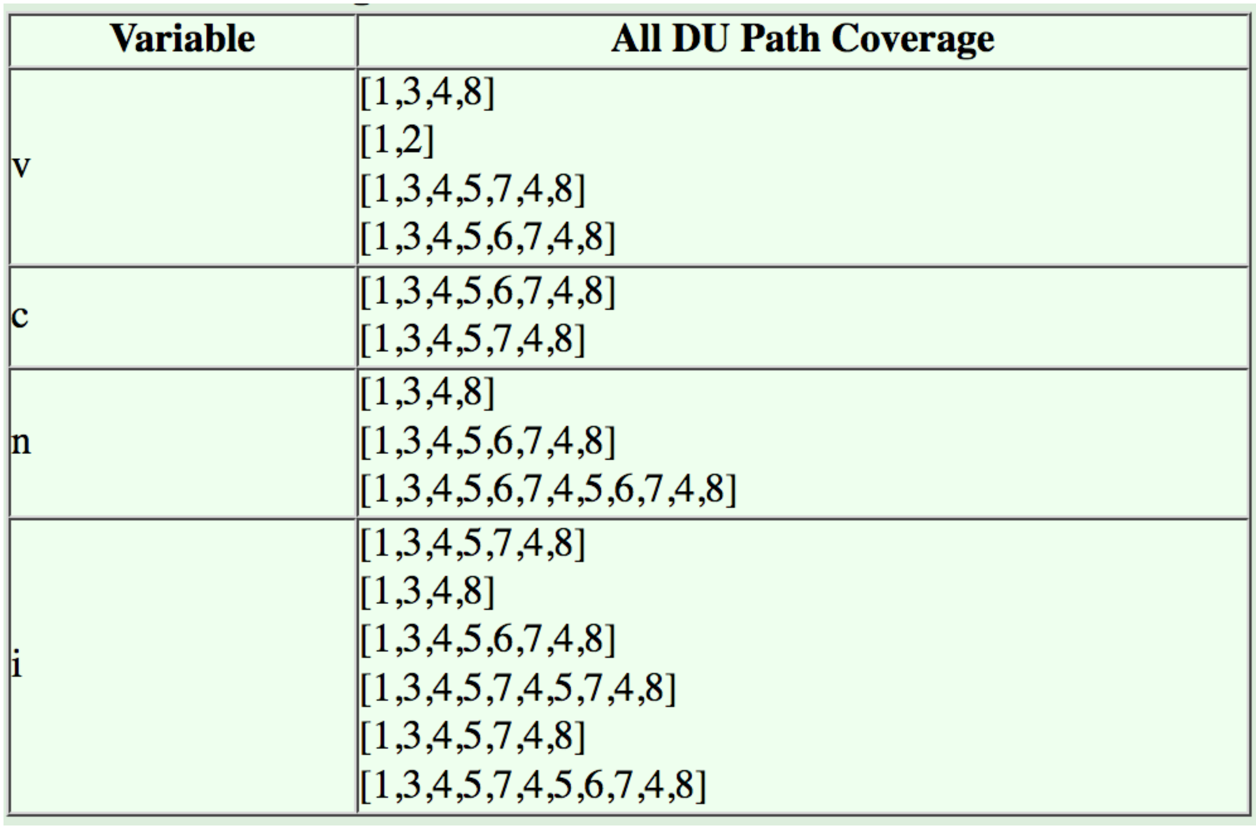
For each def, all uses must be reached

ADUPC: DU-Paths → Test Paths



DU Paths	
[1,3]	Variable v
[1,2]	
[1,3,4,8]	
[1,3,4,5]	
[1,3,4,5,7]	
[1,3,4,5,6]	
[1,3,4,5,6]	Variable c
[1,3,4,5,7]	
[3,4,8]	Variable n
[3,4,5,6]	
[6,7,4,8]	
[6,7,4,5,6]	
[3,4,5]	Variable i
[3,4,8]	
[3,4,5,6]	
[3,4,5,7]	
[3,4,5,6,7]	
[7,4,5]	
[7,4,8]	
[7,4,5,6]	
[7,4,5,7]	
[7,4,5,6,7]	

Test paths that satisfy All DU-Paths Coverage

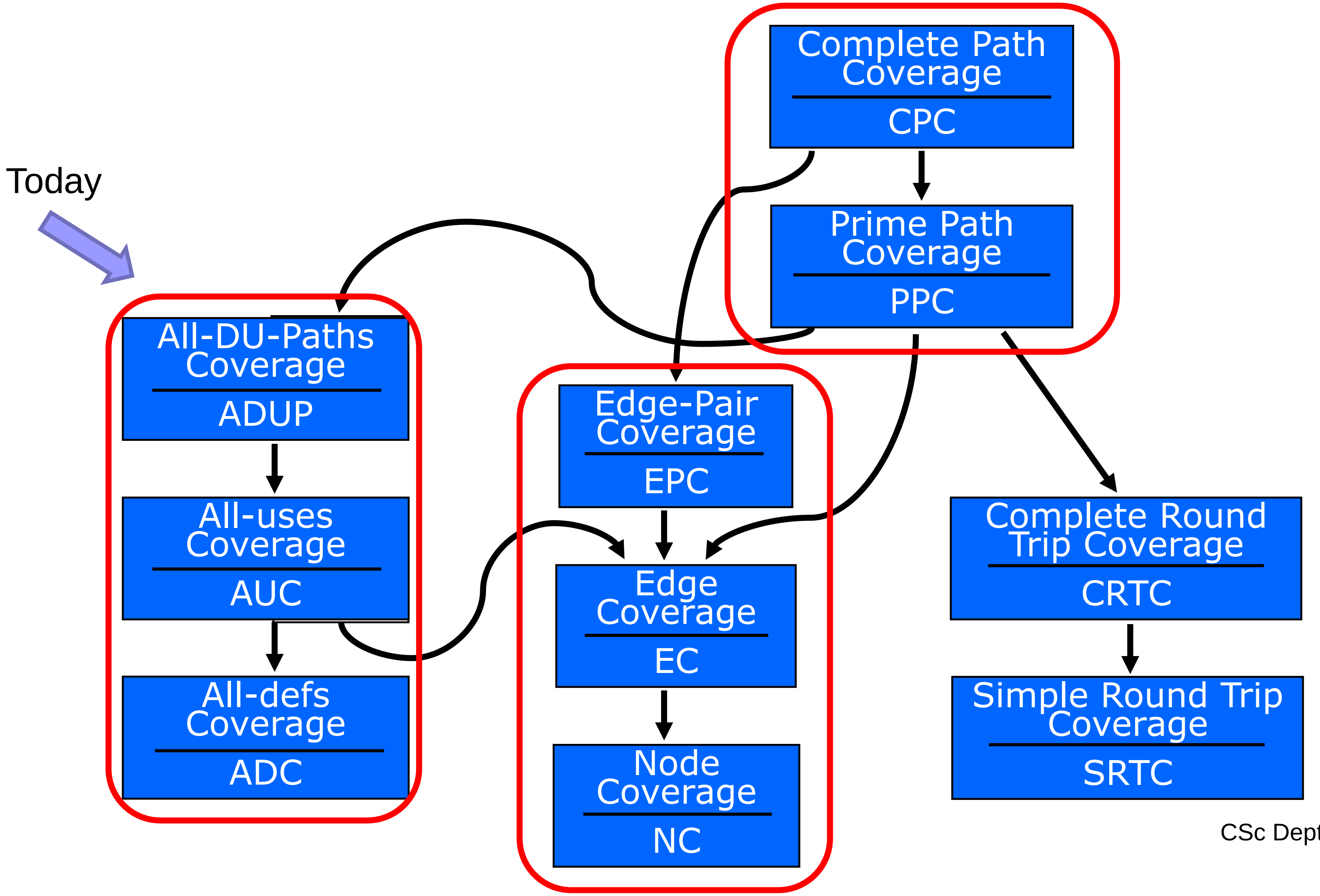


Variable	All DU Path Coverage
v	[1,3,4,8] [1,2] [1,3,4,5,7,4,8] [1,3,4,5,6,7,4,8]
c	[1,3,4,5,6,7,4,8] [1,3,4,5,7,4,8]
n	[1,3,4,8] [1,3,4,5,6,7,4,8] [1,3,4,5,6,7,4,5,6,7,4,8]
i	[1,3,4,5,7,4,8] [1,3,4,8] [1,3,4,5,6,7,4,8] [1,3,4,5,7,4,5,7,4,8] [1,3,4,5,7,4,8] [1,3,4,5,7,4,5,6,7,4,8]

All DU-Paths Coverage (ADUPC)

For each def-use pair, all paths between defs and uses must be covered

Recap:Graph Coverage Criteria Subsumption



Summary

- A common application of graph coverage criteria is to program source – control flow graph (CFG)
- Applying graph coverage criteria to control flow graphs is relatively straightforward
- Graph provides a good basis for systematic test selection.
- Control flow testing focuses on the transfer of control, while data flow testing focuses on the definitions of data and their subsequent use.
- Control flow coverage is defined in terms of nodes, edges, and paths; data flow coverage is defined in terms of def, use, and du-path.
- Path testing with Branch Coverage and Data-flow testing with AUC is a **very good combination**.

You now know …

- … data flow testing
- … All-DU-Paths Coverage
- … All-uses Coverage
- … All-defs Coverage