Data Flow Testing

© João Pereira

Learning objectives

1. Understand why data flow criteria have been designed and used

2. Recognize and distinguish basic DF criteria
   – All DU pairs, all DU paths, all definitions, …

3. Understand how the infeasibility problem impacts data flow testing

Motivation

- Huge gap between **path** and **branch** coverage models
  - Path model is too strong
    - It is exhaustive but it still may miss significant test cases
    - It is often too time consuming
    - Path-based criteria require impractical number of test cases
      - And only a few paths uncover additional faults, anyway
  - Branch model is weak
    - Not exhaustive
    - May miss significant test cases

- Need another approach to distinguish “important” paths

Data flow testing

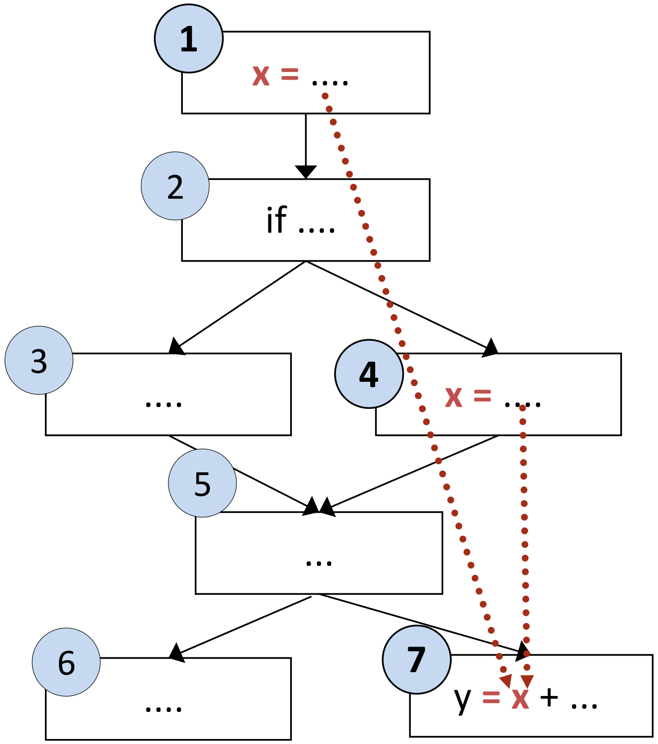
- Instead of focus on control flow of a program
- Focus on data flow of a program
  - Program reads variables, assigns new values to variables and performs computations
  - One can visualize the flow of data values from one statement to another

- Data Flow Testing focuses on
  - the points at which variables change value
  - and the points at which variables are read

- Intuition:
  - Statements interact through *data flow*
  - Value computed in one statement, used in another
  - Bad value computation revealed only when it is used

Data flow testing - 2

- Value of x at 7 could be computed at 1 or at 4
- Bad computation at 1 or 4 could be revealed only if they are used at 7



Data operation categories

- (**d**) Defined, Created, Initialized
  - It assigned a value to the variable
  - A variable is **defined** when it:
    - appears in a data declaration
    - is assigned a new value
    - is a file that has been opened
    - is dynamically allocated
    - ...
- (**k**) Killed, Undefined, Released
  - Variable is deallocated at the statement fragment
  - The value and the location of the variable become unbound
- (**u**) Used:
  - The value of the variable is used in an expression

(u) Use operation

- A variable **v** is **used** in a statement **s** when its value is applied in an expression belonging to that statement
- Two types of **use**, depending on the type of the expression:
  - **predicate use (p-use): v** appears in a predicate expression of **s**
    - **if (x > 5) …**
  - **computational use (c-use): v** appears in a *computation* expression of **s**
    - **y = 5 * x;**
      - c-use of x and def of y
- A variable is used for a computation **(c)** when it appears on the RHS (sometimes even the LHS in case of array indices) of an assignment statement.
- A variable is used in a predicate **(p)** when it appears directly in that predicate.

Use and definition - 1

- Variables are defined by assigning values to them and are used in expressions:
  - *x = y + z*
    - defines variable *x* and uses (**c-use)** variables *y* and *z*
  - scanf ("%d %d", &*x*, &*y*)
    - defines variables ***x*** and ***y***
  - printf ("Output: %d \n", *x* + *y*)
    - uses variables (**c-use**) ***x*** and ***y***
  - if (x > 0)
    - Uses variable (**p-use**) ***x***
- A parameter ***x*** passed to a function
  - *call-by-value*, is considered as *a use (c-use)* of *x*
  - *call-by-reference*, is considered as *a definition* and *use (c-use)* of *x*

Use and definition - 2

- Variables can be used and re-defined in the same statement:
  - On both sides of an assignment
    - **x** = **x** + 5;
    - **x** *= 5;
  - As a call by reference parameter in function call
    - increment(&**y**);

Use and definition – **Arrays**

- Arrays are tricky
- Example:

```
int a[10];
a[i] = x + y;
```

- Two approaches for second statement
  1. The second statement defines *a* and uses *i*, *x*, and *y*
  2. Or *second statement defines a[i]* and not the entire array *a*
  - The choice of whether to consider the entire array a as defined or the specific element *depends upon how stringent the requirement for coverage analysis is.*

Example: Use and definition

```
1.  read (x, y);
2.   z = x + 2;
3.   if (z < y)
4   w = x + 1;
     else
5.  y = y + 1;
6.   print (x, y, w, z);
```

What are the *definitions* and *uses* for this program?

Lines	*Def*	*C-use*	*P-use*
1	x, y		
2	z	x	
3			z, y
4	w	x	
5	y	y	
6		x, y, w, z	

Data flow testing: Two approaches

- Data flow testing can be performed at two conceptual levels.
  – Static data flow testing
  – Dynamic data flow testing
- Static data flow testing
  – Analyze source code
  – Do not execute code
  – Identify potential defects, commonly known as **data flow anomaly**
- 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**

Data flow anomaly

- Anomaly: It is an abnormal way of doing something.
  - Example 1: The second definition of x overrides the first.

```
x = f1(y);
x = f2(z);
```

- Three types of abnormal situations with using variable.
  - Type 1: Defined and then defined again
    - Action sequence **dd**
  - Type 2: Undefined but referenced
    - Action sequence **-u**
  - Type 3: Defined but not referenced
    - Action sequence **dk**

Data flow anomaly

- Type 1 - Defined and then defined again

```
x = f1(y);        →        x = f1(y);
x = f2(z);                 v = f3(x);
                           x = f2(z);
```

  – Four interpretations of example
    - First statement is redundant.
    - First statement has a fault -- the intended one might be: w = f1(y).
    - The second statement has a fault – the intended one might be: v = f2(z).
    - There is a missing statement in between the two: v = f3(x).
  – Note: It is for the programmer to make the desired interpretation.
- Type 2 - Undefined but referenced
  – Example:
    - x = y – w; // w has not been defined by the programmer
  – Two interpretations
    - The programmer made a mistake in using w
    - The programmer wants to use the compiler assigned value of w
- Type 3 - Defined but not referenced
  – Example: Consider x = f(x, y). If x is not used subsequently, we have a Type 3 anomaly.

Dynamic data flow testing

- A family of coverage models
- Select paths by analyzing the program's data flow in order to explore sequences of events related to the status of data objects
- *E.g.,* Pick enough paths to assure that:
  - Every definition of a variable was used at least once
  - All uses of a definition have been exercised
- This family leads to test path selection strategies that fill the **gap** between **path** and **branch** coverage models

Overview of dynamic data flow testing

- A program manipulates/uses variables in several ways
  - Initialization, assignment of variables
  - And then used in a computation and/or condition
- Motivation for data flow testing?
  - One should not feel confident that a variable has been **assigned the correct value**, if no test causes the execution of a **path** from the point of assignment to a point where the value is **used**.
- The above motivation indicates that **certain kinds of paths** should be executed in data flow testing

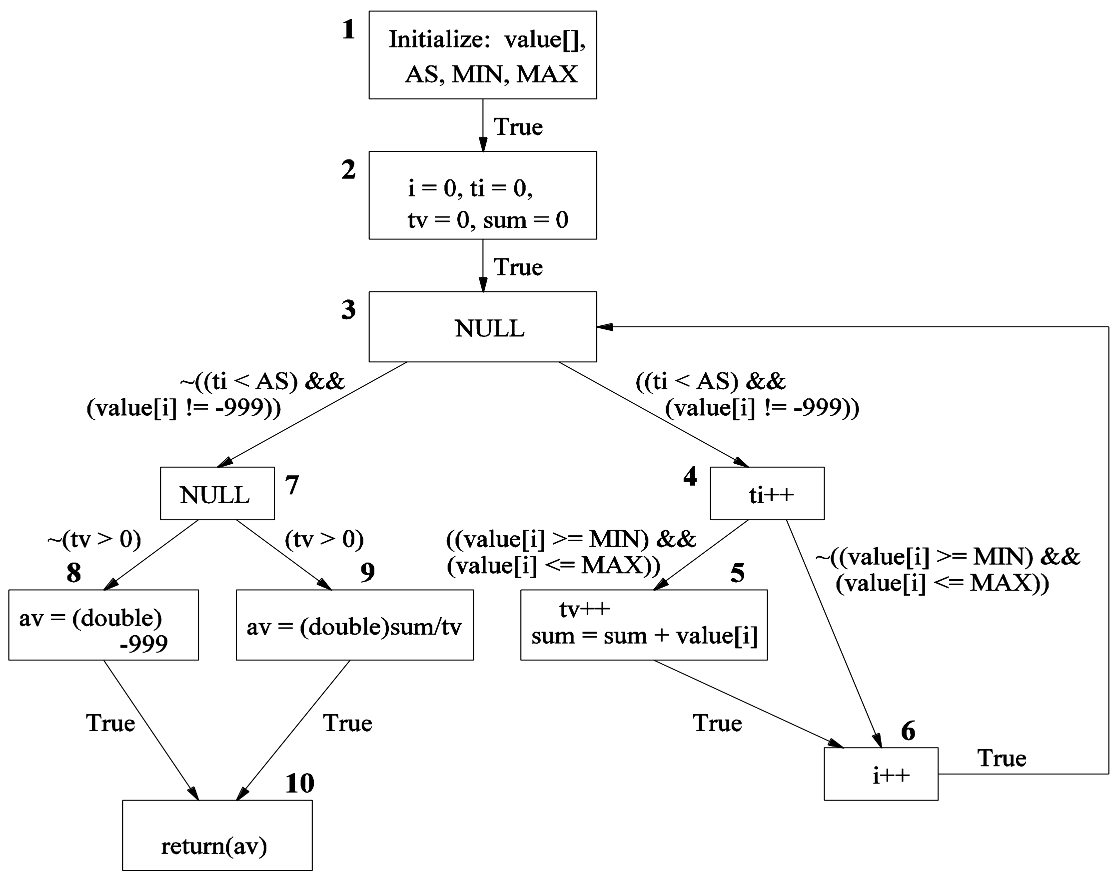
Overview of dynamic data flow testing

- Data flow testing is outlined as follows:

1. Draw a data flow graph from program **P**
   - Similar to control flow graph of **P**
     - All nodes, edges and paths of CGF are preserved
     - Can also consider one statement per node (makes analysis simpler)
2. For each variable, classify each node as *defining* or *usage node*
3. Select one or more data flow testing criteria
   - All-uses, all-defs, …
4. Identify paths in the data flow graph satisfying the testing criteria
5. Compute input values for each path
   - Derive path predicate expressions from the selected paths
   - Solve the path predicate expressions to derive test inputs

Definition clear path

- **Definition clear path (dc-path)**: A path ($i - n_1 - \dots n_m - j$), $m \geq 0$, is called a definition clear path (def-clear path) with respect to variable **v** if v has been neither defined nor undefined in nodes $n_1 - \dots n_m$
  - def-clear path w.r.t. tv (node 2)
    - (2 – 3 – 4 – 5)
    - (2 – 3 – 4 – 6)
    - (2 – 3 – 4 – 6 – 3 – 4 – 6 – 3 – 4 – 5)
    - (2 – 3 – 4 – 5 – 6 – 3 – 4 – 5)
  - def-clear path w.r.t. tv (node 5)
    - (5 – 6 – 3 – 4 – 5)



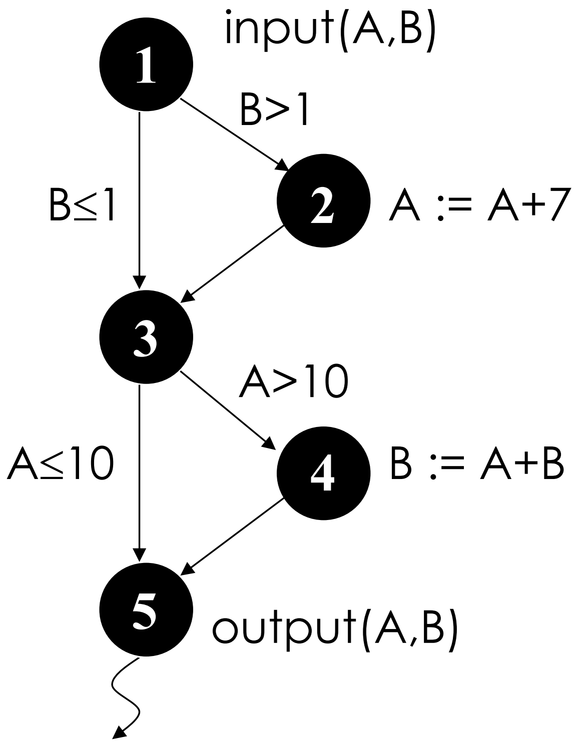
More dataflow terms and definitions

- A *definition-use pair* **(“du-pair”)** with respect to a variable **v** is a pair **(d,u)** such that
  - **d** is a node in the program’s flow graph at which **v** is defined,
  - **u** is a node or edge at which **v** is used *and*
  - there is at least one def-clear path *with respect to **v*** from **d** to **u**
  - In other words, there is at least one path (d, …, u) such that the value that is assigned to **v** at **d** is used at **u**
- DU-pair : A pair of nodes **(i, j)** (or **(I, <j, k)>)**) such that a variable **v** is defined at **i** and that value is used at **j** (or **<j, k>**)
- Note that the definition of a du-pair does not require the existence of a **feasible** def-clear path from d to u

Example 1

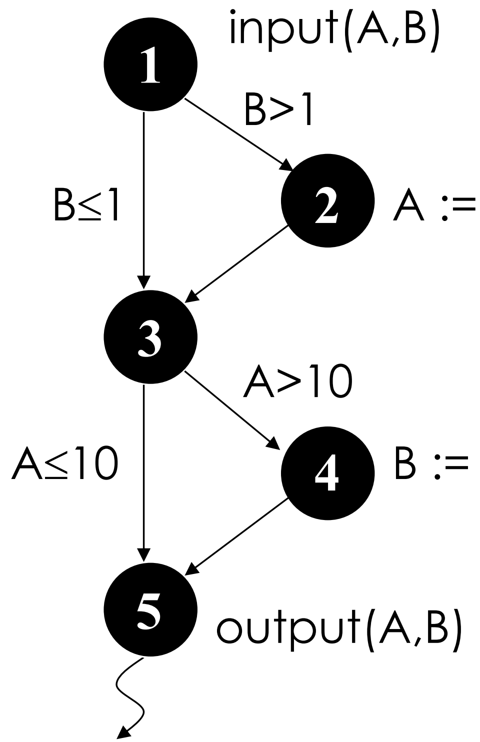
```
1. input(A,B)
    if (B > 1)
2.      A = A + 7
3. if (A > 10)
4.      B = A + B
5. output(A, B)
```

- **Data flow testing**
  - Compute the du-pairs and dc-paths for each variable



Identifying DU-Pairs – Variable A

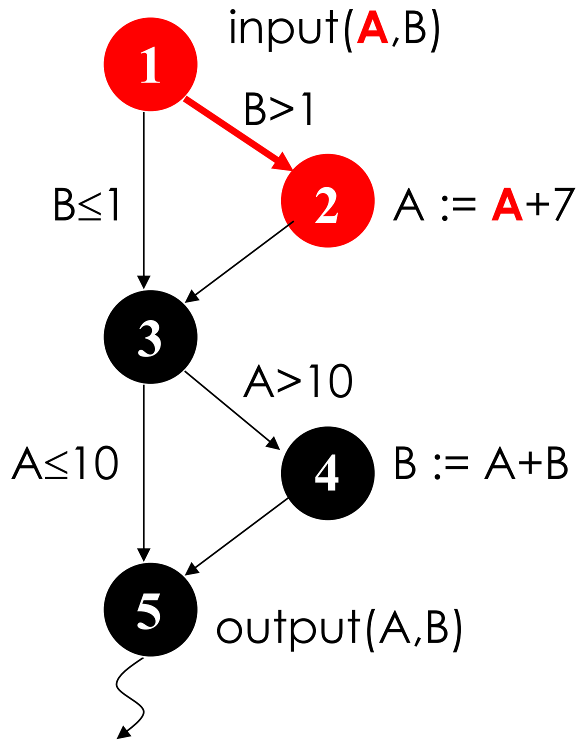
du-pair	dc-path
(1,2)	
(1,4)	
(1,5)	
(1,<3,4>)	
(1,<3,5>)	
(2,4)	
(2,5)	
(2,<3,4>)	
(2,<3,5>)	



- Nodes defining A?
  - Def(A) = {1, 2}
- Nodes using A?
  - Use{A} = {2, 4, 5, <3,4>, <3,5>}

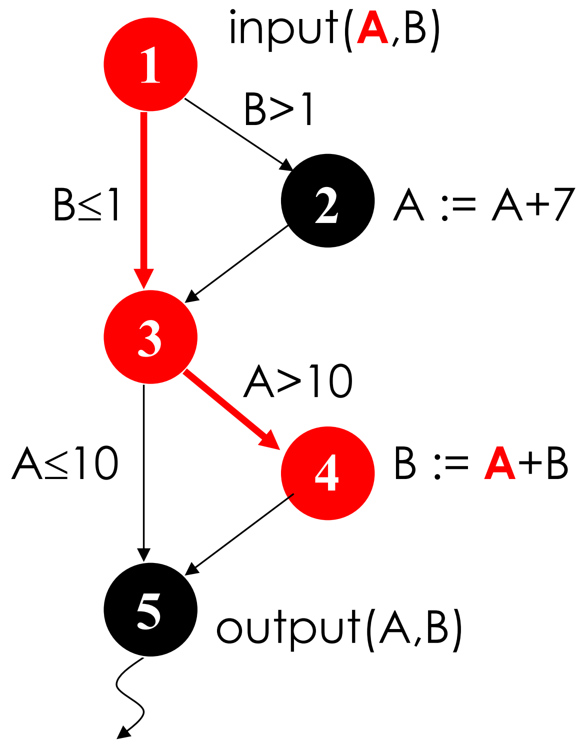
Identifying DU-Pairs – Variable A

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



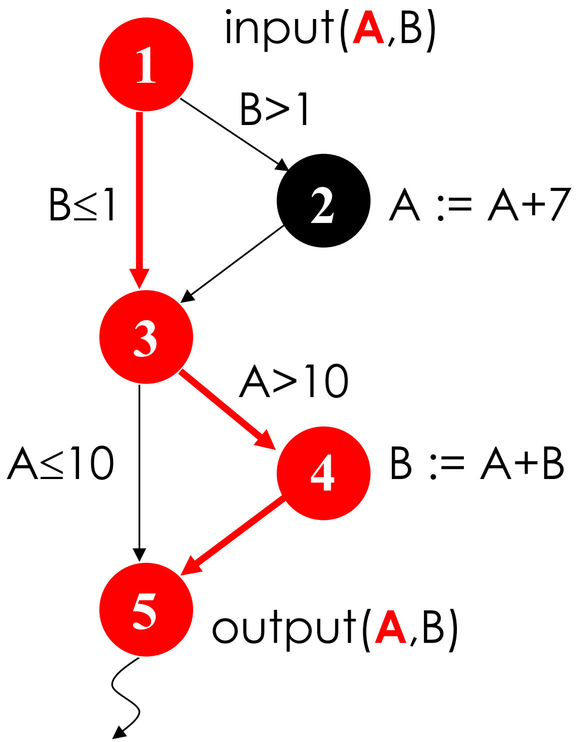
Identifying DU-Pairs – Variable A

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



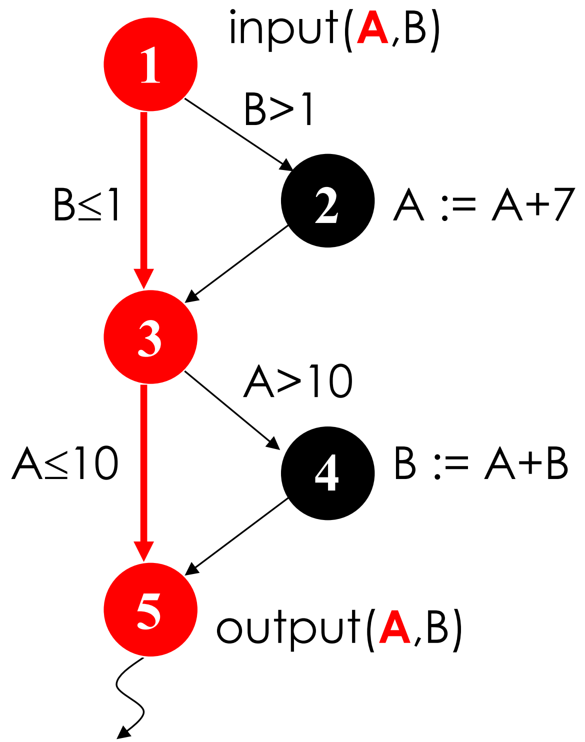
Identifying DU-Pairs – Variable A

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



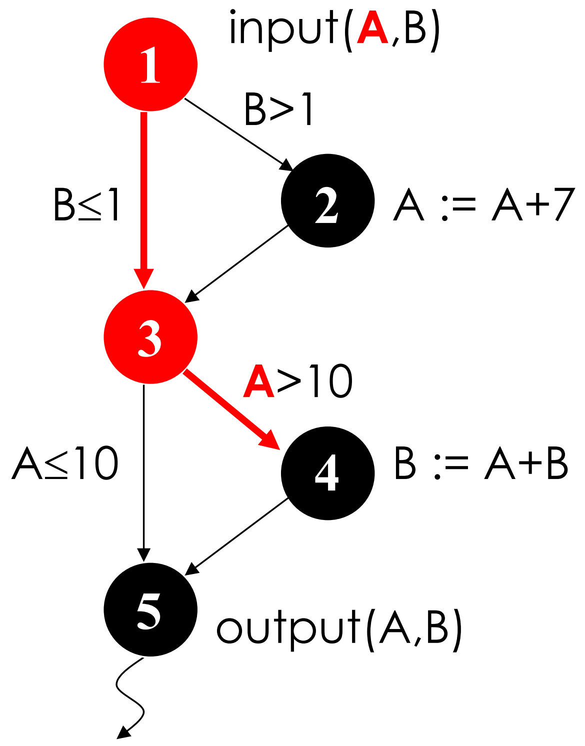
Identifying DU-Pairs – Variable A

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



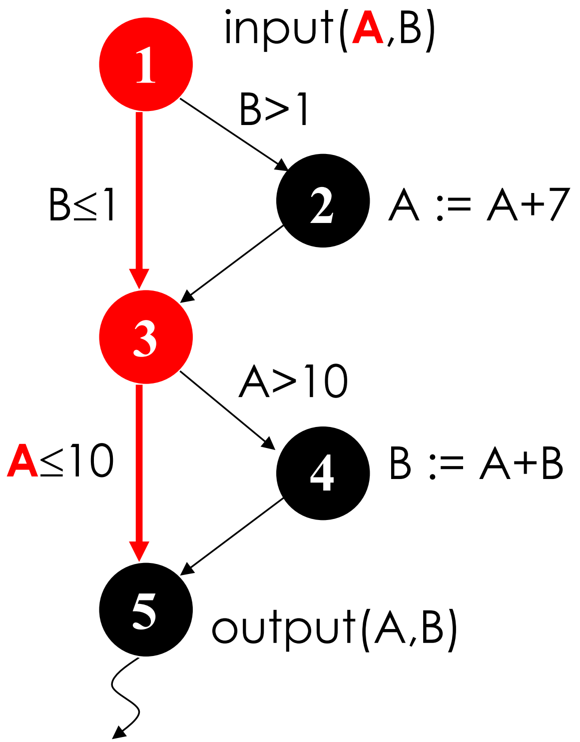
Identifying DU-Pairs – Variable A

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



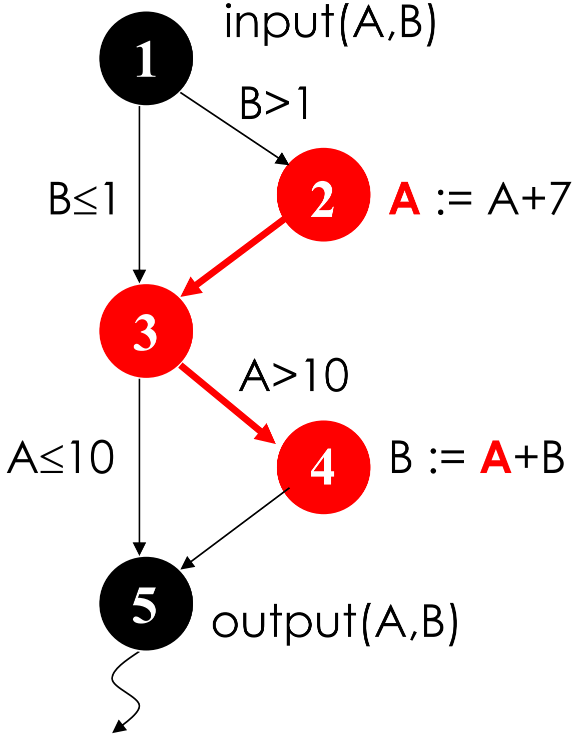
Identifying DU-Pairs – Variable A

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)	
(2,5)	
(2,<3,4>)	
(2,<3,5>)	



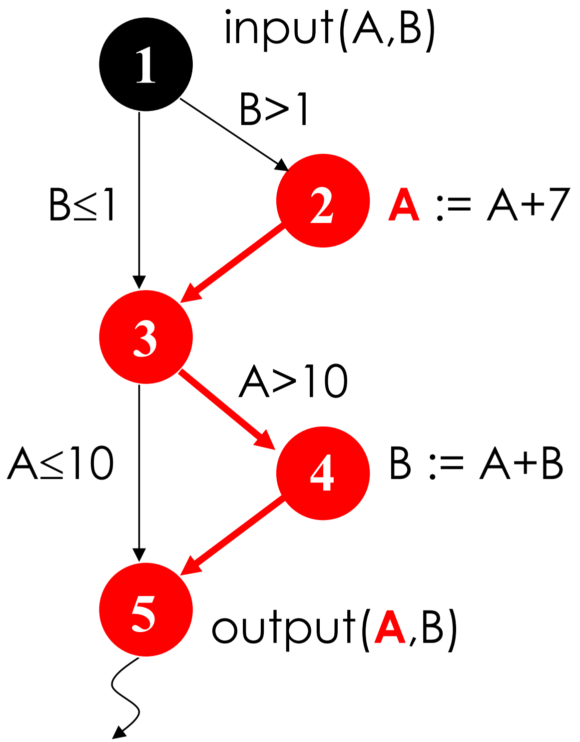
Identifying DU-Pairs – Variable A

du-pair	dc-path
(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>)	
(2,<3,5>)	



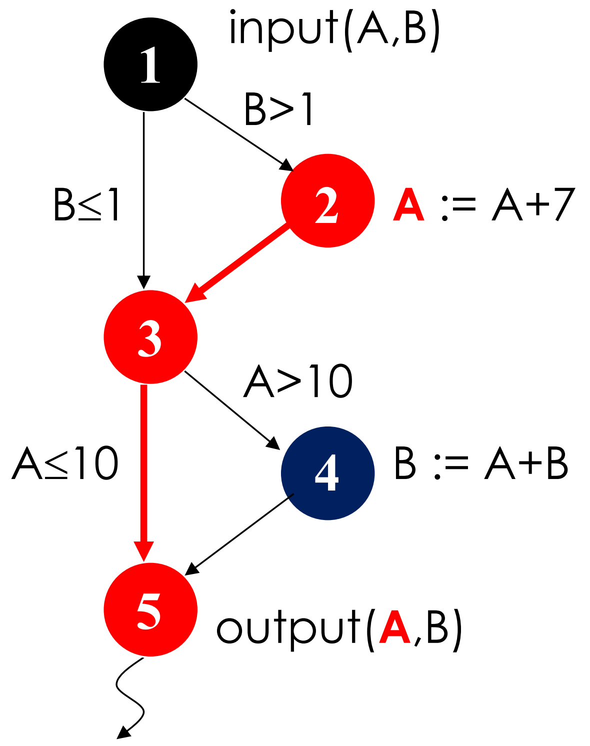
Identifying DU-Pairs – Variable A

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



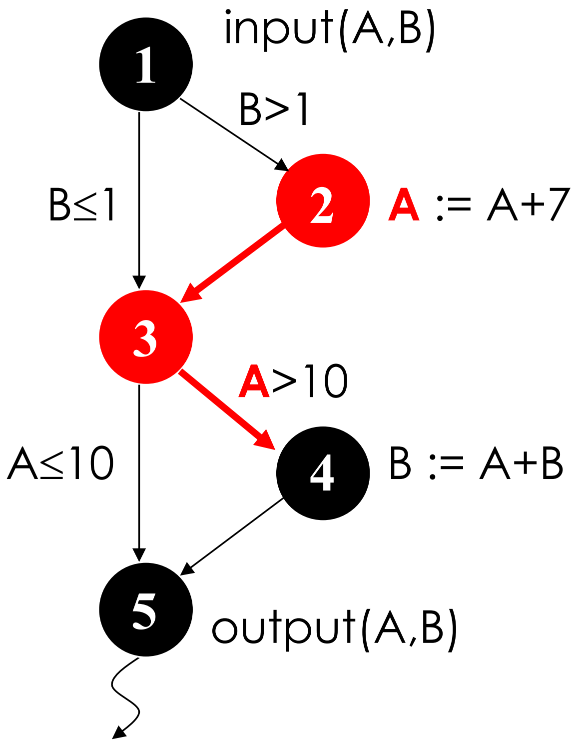
Identifying DU-Pairs – Variable A

du-pair	dc-path
(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,<3,5>)	



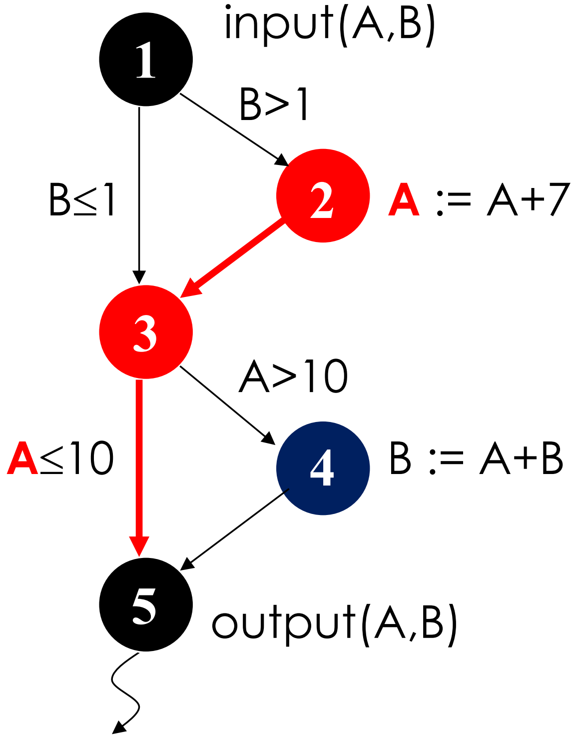
Identifying DU-Pairs – Variable A

du-pair	dc-path
(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>)	



Identifying DU-Pairs – Variable A

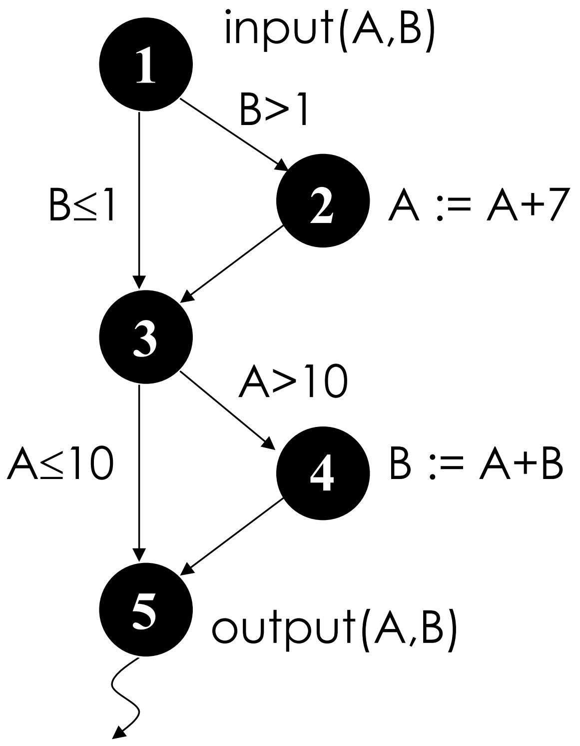
du-pair	dc-path
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	**<2,3,5>**



Identifying DU-Pairs – Variable B

- Nodes defining B?
  - Def(B) = {1, 4}
- Nodes using B?
  - Use{B} = {4, 5, <1,2>, <1,3>}

du-pair	dc-path
(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>



Frankl and Weyuker’s Data flow coverage criteria

- Seven data flow testing criteria
  - **All-defs**
  - **All-uses**
  - All-c-uses
  - All-p-uses
  - All-p-uses/some-c-uses
  - All-c-uses/some-p-uses
  - **All-du-paths**
- The family of data flow criteria requires that the test data execute definition-clear paths from each node containing a definition of a variable to specified nodes containing *c-use* and edges containing *p-use* of that variable.

All-Defs coverage criterion

- ***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

➢Meaning: All definitions get used at least once

All-Defs coverage criterion - Example

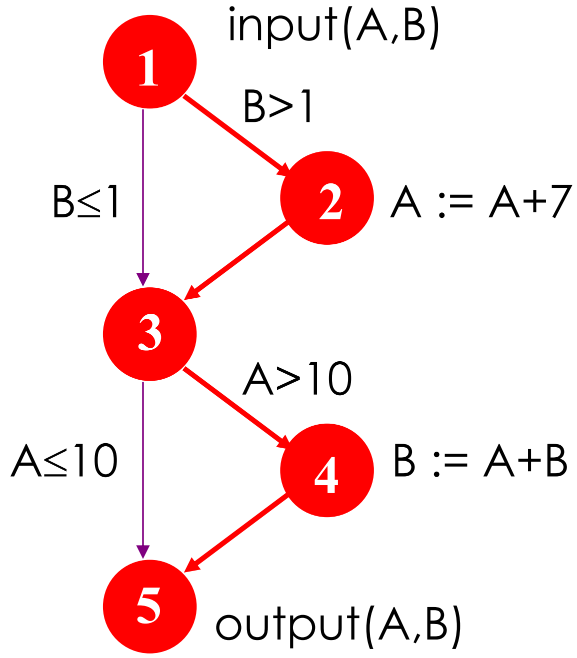
- Consider a test case executing complete path <1,2,3,4,5>
  - **Complete path**: A complete path is a path from the entry node to the exit node
  - Corresponds to the entry-exit node of CF models

- 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 and B

du-pair for A	dc-path
(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>

du-pair for B	dc-path
(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>



- 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 with a single test case

All-Uses coverage criterion

- All-Uses coverage criterion:

  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

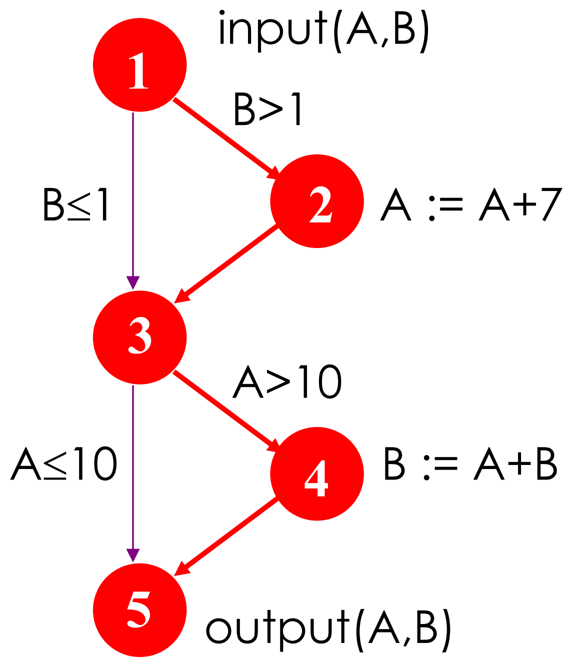
- Requires all du-pairs are exercised at least once

- Meaning: Every computation and branch directly affected by a definition is exercised

Does <1,2,3,4,5> achieves All-Uses for variables A and B?

du-pair for A	dc-path
(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>

du-pair for B	dc-path
(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>

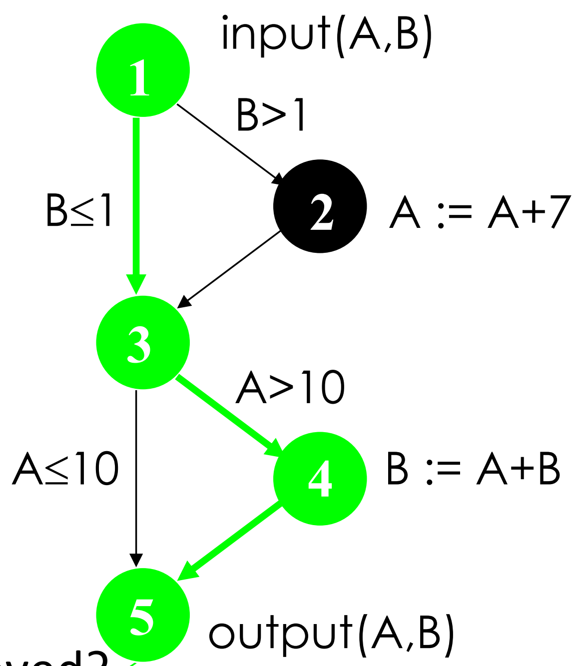


- No! For example (1,4) and (1,5) for A are not covered
- Consider additional test cases executing paths:
  - **<1,3,4,5>**
  - **<1,2,3,5>**

Def-Clear paths subsumed by <1,3,4,5>

du-pair for A	dc-path
(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>

du-pair for B	dc-path
(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> ✓

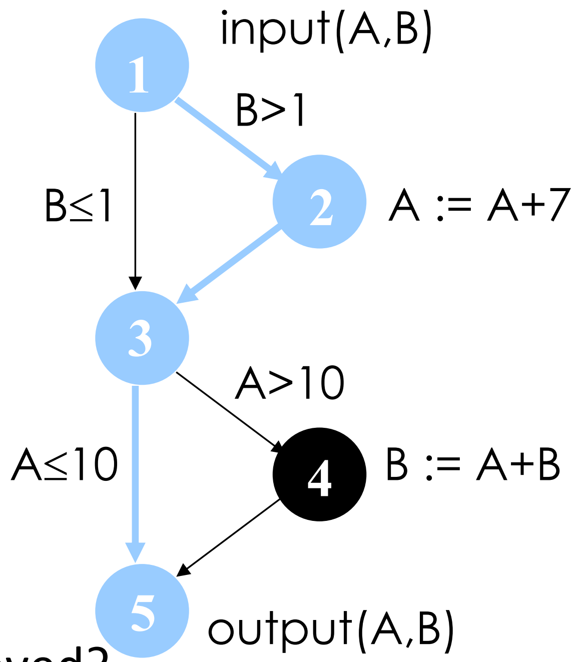


- Is the All-Uses coverage achieved?
- No! Du-pairs not exercised yet:
- Variable A : du-pair (1,<3,5>) and (2,<3,5>)
- Variable B: du-pair (1, 5)

Def-Clear paths subsumed by <1,2,3,5>

du-pair for A	dc-path
(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> ✔

du-pair for B	dc-path
(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> ✔



- Is the All-Uses coverage achieved?
- None of the three test cases covers the du-pair (1,<3,5>) for variable **A**
  - All-Uses Coverage is not achieved
  - Need additional test case

More data flow coverage criteria

- Given the set **V** of variables of program **P**
- **All-P-Uses**:
  - For every variable v in V, at least one dc-path from every definition of v to every P-use of v must be covered
- **All-C-Uses:**
  - For every variable v in V, at least one dc-path from every definition of v to every C-use of v must be covered

More data flow coverage criteria - 2

- Given the set **V** of variables of program **P**
- **All-P-Uses/Some-C-Uses**:
  - For every variable v in V, at least one dc-path from every definition of v to every P-use of v must be covered
  - If a definition of v has no P-uses, at least one dc-path from the definition of v to a C-use of v must be covered
- **All-C-Uses/Some-P-Uses:**
  - For every variable v in V, at least one dc-path from every definition of v to every C-use of v must be covered
  - If a definition of v has no C-uses, at least one dc-path from the definition of v to a P-use of v must be covered

All-DU-Paths coverage criterion

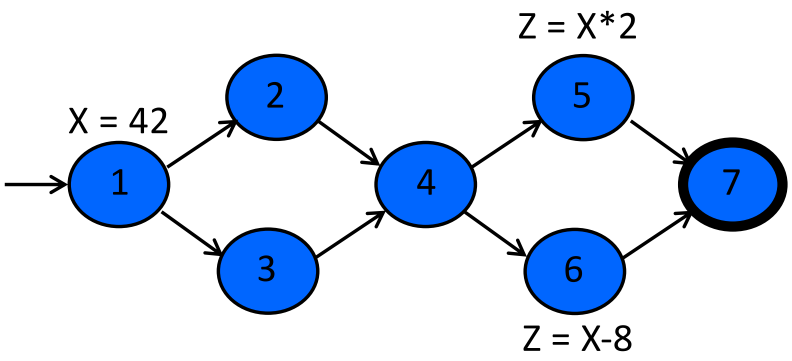
- **Simple path**: A simple path is a path in which all nodes, except possibly the first and the last, are distinct
  - 1-2-3-4-5-9 and 1-3-4-5-2-1 are simple paths
  - 1-2-3-4-5-3-7 and 1-2-3-4-1-6-5-8 are not simples paths
- A path $\langle \mathbf{n_1, n_2, \ldots, n_j, n_k} \rangle$ is a **du-path** wrt a variable **v** if
  1. **v** is defined at node $\mathbf{n_1}$ and
  2. there is a **c-use** of **v** at node $\mathbf{n_k}$ or a **p-use** of **v** at edge $\langle \mathbf{n_j, n_k} \rangle$ and
  3. $\langle \mathbf{n_1, n_2, \ldots, n_j, n_k} \rangle$ is a def-clear **simple** path
- ***All DU-Paths criterion:***

  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

Data flow testing example



- Nodes defining X?
  - Def(X) = {1}
- Nodes using X?
  - Use{X} = {5, 6}

du-pair	dc-path
(1, 5)	< 1, 2, 4, 5 >
	< 1, 3, 4, 5 >
(1,6)	< 1, 2, 4, 6 >
	< 1, 3, 4, 6 >

Determine a minimal test suite that achives100% coverage for:

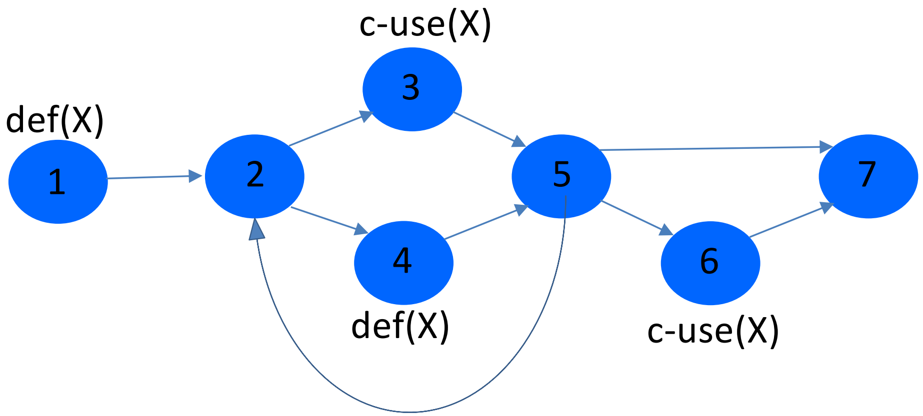
- All-defs
- All-uses
- All-du-paths

All-defs for *X*
[1, 2, 4, 5, 7]

All-uses for *X*
[1, 2, 4, 5, 7]
[1, 2, 4, 6, 7]

All-du-paths for *X*
[1, 2, 4, 5, 7]
[1, 2, 4, 6, 7]
[1, 3, 4, 5, 7]
[1, 3, 4, 6, 7]

Data Flow Testing Example - 2



du-pair	dc-path	Du-path
(1, 3)	< 1, 2, 3 >	✓
(1, 6)	< 1, 2, 3, 5, 6 >	✓
	< 1, (2, 3, 5)+ , 6 >	✗
(4, 3)	< 4, 5, 2, 3 >	✓
(4, 6)	< 4, 5, 2, 3 , 5, 6 >	✗
	< 4, (5,2,3)+,5, 6 >	✗
	< 4, 5 , 6 >	✓

Determine a minimal test suite that achives 100% coverage for:

- All-defs
- All-uses
- All-du-paths

All-defs for *X*

[1, 2, 3, 5, 2, 4, 5, 6 , 7]

All-uses for *X*

[1, 2, 3, 5, 6 ,7]

[1, 2, 4, 5, 2, 3, 5, 6, 7]

~~**All-du-paths for *X***~~

~~[1, 2, 3, 5, 6, 7]~~

~~[1, 2, 4, 5, 2, 3, 5, 6, 7]~~

~~[1, 2, 4, 5 ,6, 7]~~

All-du-paths for *X*

[1, 2, 3, 5, 6, 7]

[1, 2, 4, 5, 2, 3, 5, 2, 4, 5, 6, 7]

Relationship between strategies