Software Testing and Quality Assurance

Lecture – 3

Agenda

- Dataflow based testing
- Requirements based testing

Data-Flow Testing

- **Data-flow testing** uses the control flow graph to explore the unreasonable things that can happen to data (*i.e.,* anomalies).
- Consideration of data-flow anomalies leads to test path selection strategies that fill the gaps between complete path testing and branch or statement testing.

Data-Flow Testing (Cont’d)

- **Data-flow testing** is the name given to a family of test strategies based on selecting paths through the program’s control flow in order to explore sequences of events related to the status of data objects.
- *E.g.*, Pick enough paths to assure that:
  - Every data object has been initialized prior to its use.
  - All defined objects have been used at least once.

Data Object Categories

- (d) Defined, Created, Initialized
- (k) Killed, Undefined, Released
- (u) Used:
  - (c) Used in a calculation
  - (p) Used in a predicate

(d), (u) and (k)

- An object (*e.g.*, variable) is **defined** (d) when it:
  - appears in a data declaration
  - is assigned a new value
  - is a file that has been opened
  - is dynamically allocated, etc.
- An object is **killed** when it is:
  - released (*e.g.*, free) or otherwise made unavailable (*e.g.*, out of scope)
  - a loop control variable when the loop exits
  - a file that has been closed
- An object is **used** when it is part of a computation or a predicate.
- 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.

Example: Definition and Uses

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

```
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);
```

Example: Definition and Uses

```
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);
```

Def	*C-use*	*P-use*
x, y		
z	x	
		z, y
w	x	
y	y	
	x, y, w, z	

Data-Flow Anomalies

- A **data-flow anomaly** is denoted by a two character sequence of actions. *E.g.,*
  - **ku**: Means that an object is killed and then used.
  - **dd**: Means that an object is defined twice without an intervening usage.

Two Letter Combinations for **d k u**

- **dd**: Probably harmless, but suspicious.
- **dk**: Probably a bug.
- **du**: Normal situation.
- **kd**: Normal situation.
- **kk**: Harmless, but probably a bug.
- **ku**: Definitely a bug.
- **ud**: Normal situation (reassignment).
- **uk**: Normal situation.
- **uu**: Normal situation.

Single Letter Situations

- A leading dash means that nothing of interest (**d**, **k**, **u**) occurs prior to the action noted along the entry-exit path of interest.
- A trailing dash means that nothing of interest happens after the point of action until the exit.

Single Letter Situations

- **-k**: Possibly anomalous:
  - Killing a variable that does not exist.
  - Killing a variable that is global.
- **-d**: Normal situation.
- **-u**: Possibly anomalous, unless variable is global.
- **k-**: Normal situation.
- **d**-: Possibly anomalous, unless variable is global.
- **u**-: Normal situation.

Dataflow anomalies

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

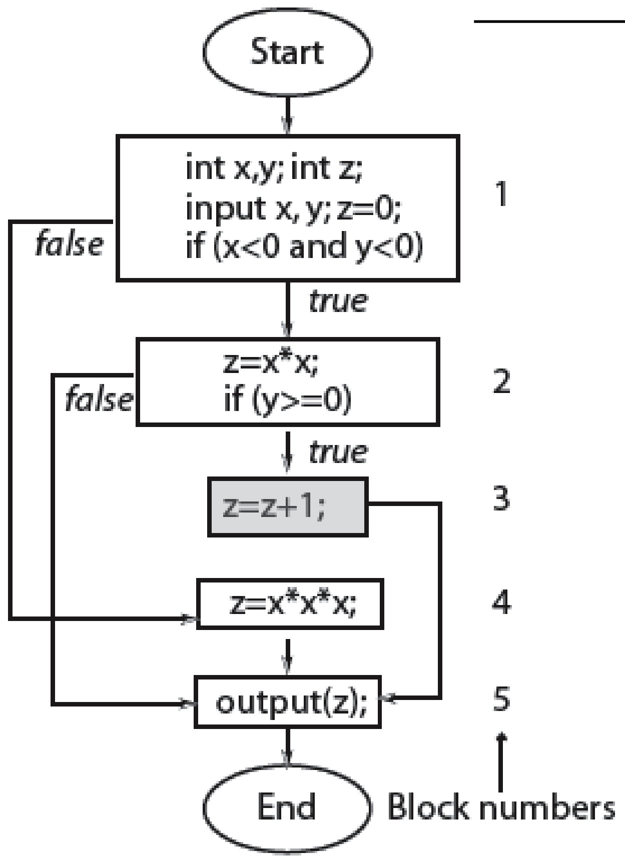
```
x = f1(y);
x = f2(z);
```

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

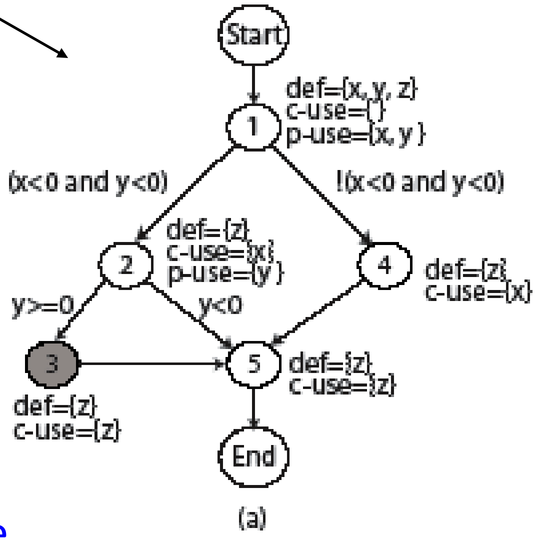
Dataflow anomalies

- Type 1: Defined and then defined again (Example 1 above)
  - Four interpretations of Example 1
    - The first statement is redundant.
    - The 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 = 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.

Data flow graph: Example



Node (or Block)	def	c-use	p-use
1	{x, y, z}	{ }	{x, y}
2	{z}	{x}	{y}
3	{z}	{z}	{ }
4	{z}	{x}	{ }
5	{ }	{z}	{ }



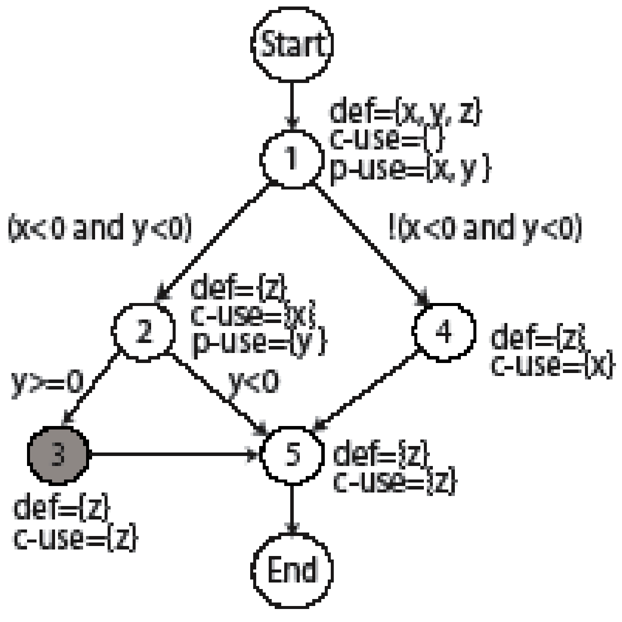
Unreachable node

Def-clear path

Any path starting from a node at which variable x is defined and ending at a node at which x is used, without redefining x anywhere else along the path, is a def-clear path for x.

Path 2-5 is def-clear for variable z defined at node 2 and used at node 5. Path 1-2-5 is NOT def-clear for variable z defined at node 1 and used at node 5.

Thus definition of z at node 2 is live at node 5 while that at node 1 is not live at node 5.



(a)

Def-use pairs

Def of a variable at line l_1 and its use at line l_2 constitute a def-use pair. l_1 and l_2 can be the same.

dcu (di(x)) denotes the set of all nodes where di(x)) is live and used.

dpu (di(x)) denotes the set of all edges (k, l) such that there is a def-clear path from node i to edge (k, l) and x is used at node k.

We say that a def-use pair $(d_i(x), u_j(x))$ is covered when a def-clear path that includes nodes i to node j is executed. If $u_j(x))$ is a p-use then all edges of the kind (j, k) must also be taken during some executions.

Static vs Dynamic Anomaly Detection

- **Static Analysis** is analysis done on source code without actually executing it.
- *E.g.*, Syntax errors are caught by static analysis.

Static vs Dynamic Anomaly Detection (Cont’d)

- **Dynamic Analysis** is analysis done as a program is executing and is based on intermediate values that result from the program’s execution.
- *E.g.,* A division by 0 error is caught by dynamic analysis.
- If a data-flow anomaly can be detected by static analysis then the anomaly does not concern testing. (Should be handled by the compiler.)

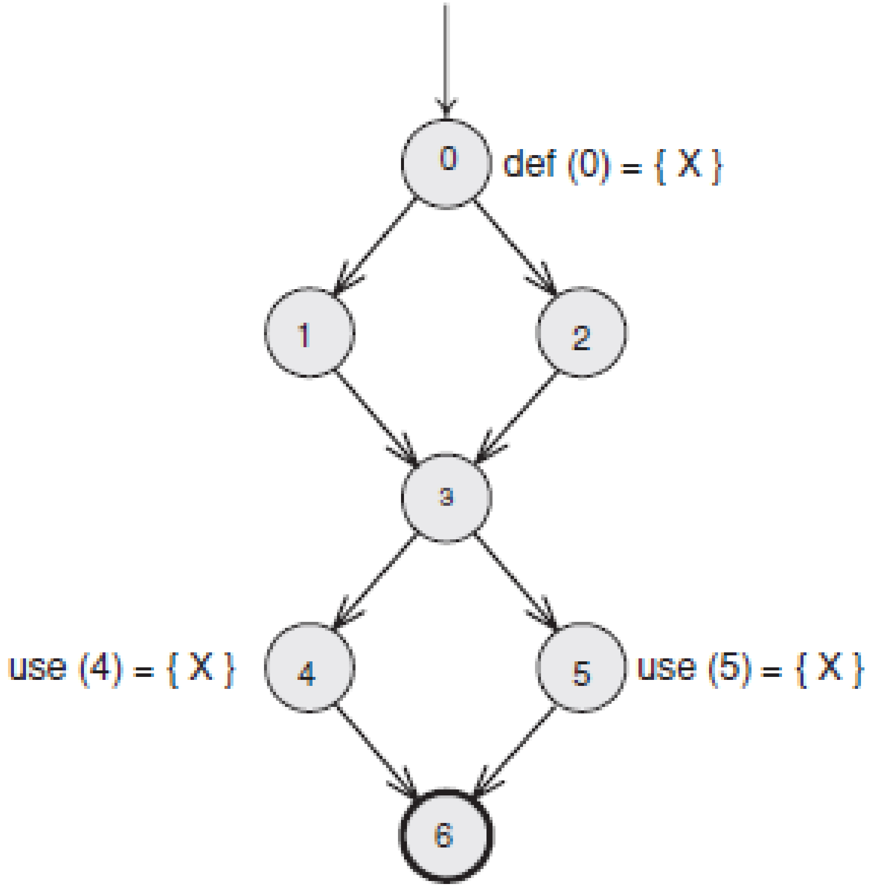
Anomaly Detection Using Compilers

- Compilers are able to detect several data-flow anomalies using static analysis.
- *E.g.*, By forcing declaration before use, a compiler can detect anomalies such as:
  - **-u**
  - **ku**
- Optimizing compilers are able to detect some dead variables.

Simple Path Segments

- A **Simple Path Segment** is a path segment in which at most one node is visited twice.
  - *E.g.*, (7,4,5,6,7) is simple.
- Therefore, a simple path may or may not be loop-free.

Paths - Explained



All-defs
0-1-3-4

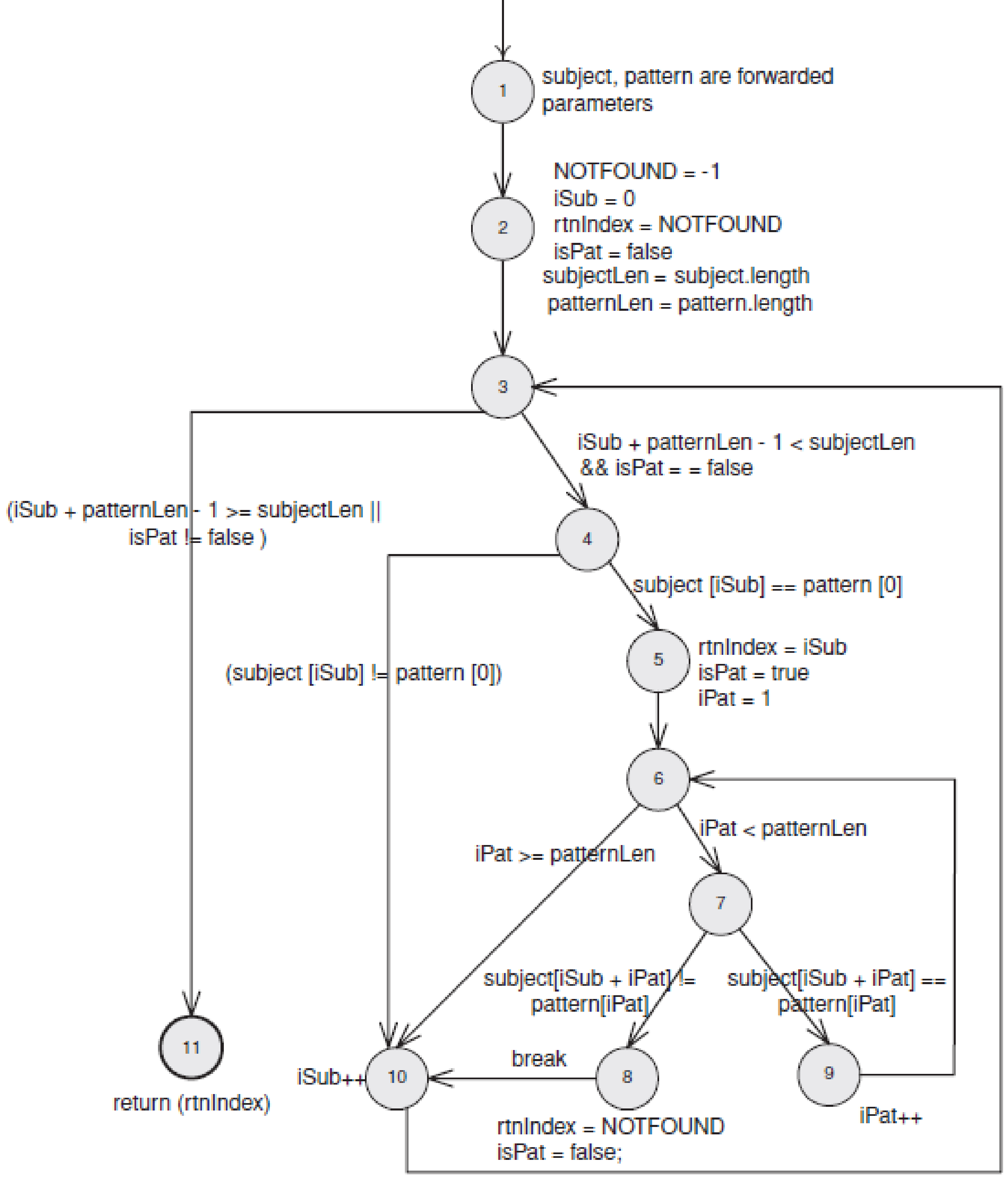
All-uses
0-1-3-4
0-1-3-5

All-du-paths
0-1-3-4
0-1-3-5
0-2-3-4
0-2-3-5

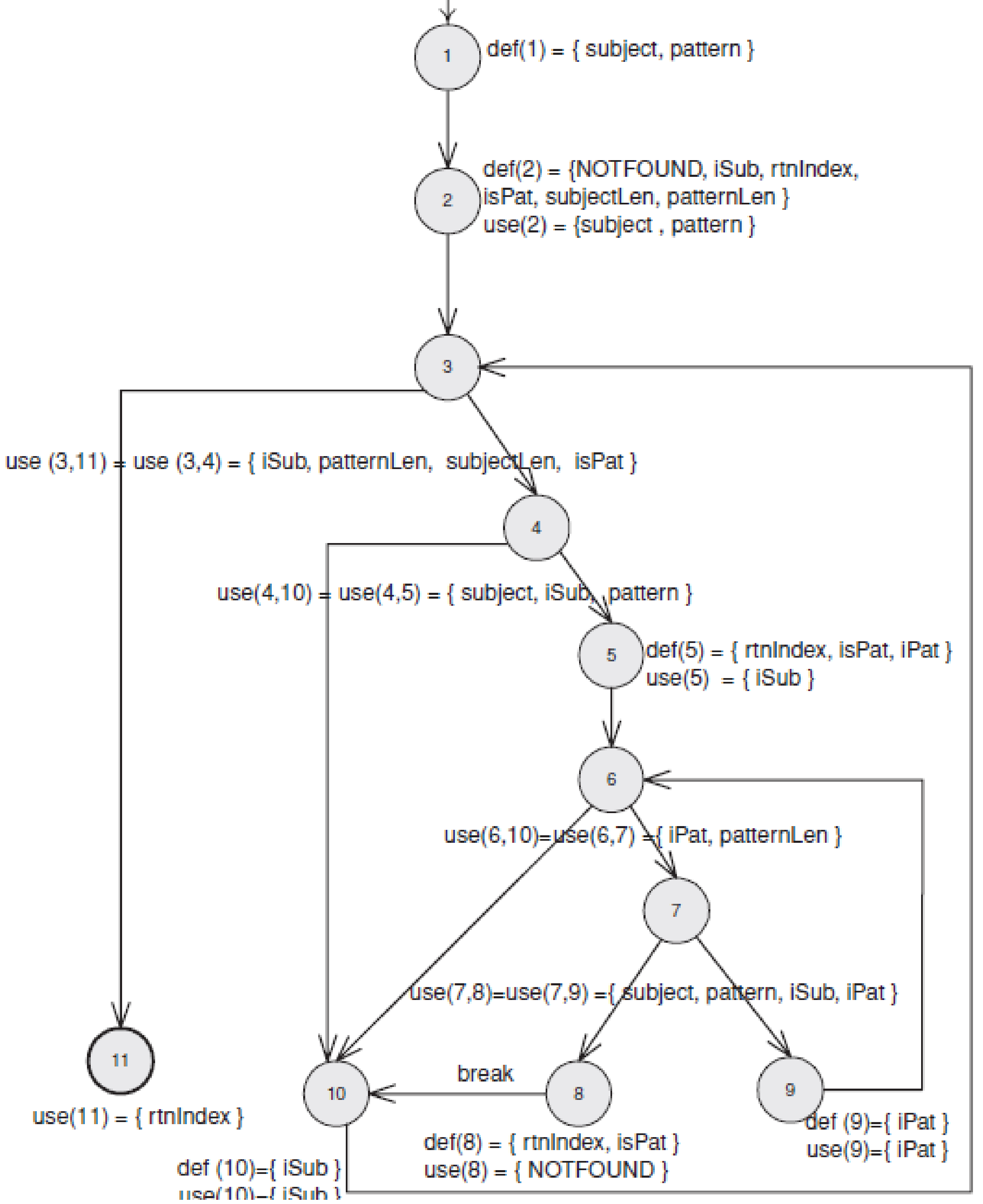
Data-Flow Testing Strategies

- All **du** Paths (ADUP)
- All **Uses** (AU)
- All **p-uses**/some **c-uses** (APU+C)
- All **c-uses**/some **p-uses** (ACU+P)
- All **Definitions** (AD)
- All **p-uses** (APU)
- All **c-uses** (ACU)

Example



1
subject, pattern are forwarded parameters
2
NOTFOUND = -1
iSub = 0
rtnIndex = NOTFOUND
isPat = false
subjectLen = subject.length
patternLen = pattern.length
3
iSub + patternLen - 1 < subjectLen && isPat = = false
(iSub + patternLen - 1 >= subjectLen || isPat != false)
4
subject [iSub] == pattern [0]
(subject [iSub] != pattern [0])
5
rtnIndex = iSub
isPat = true
iPat = 1
6
iPat < patternLen
iPat >= patternLen
7
subject[iSub + iPat] != pattern[iPat]
subject[iSub + iPat] == pattern[iPat]
11
return (rtnIndex)
iSub++
10
break
8
rtnIndex = NOTFOUND
isPat = false;
9
iPat++


1
def(1) = { subject, pattern }
2
def(2) = {NOTFOUND, iSub, rtnIndex, isPat, subjectLen, patternLen }
use(2) = {subject , pattern }
3
use (3,11) = use (3,4) = { iSub, patternLen, subjectLen, isPat }
4
use(4,10) = use(4,5) = { subject, iSub, pattern }
5
def(5) = { rtnIndex, isPat, iPat }
use(5) = { iSub }
6
use(6,10)=use(6,7) ={ iPat, patternLen }
7
use(7,8)=use(7,9) ={ subject, pattern, iSub, iPat }
8
9
10
11
break
use(11) = { rtnIndex }
def(8) = { rtnIndex, isPat }
use(8) = { NOTFOUND }
def (9)={ iPat }
use(9)={ iPat }
def (10)={ iSub }
use(10)={ iSub }

Paths

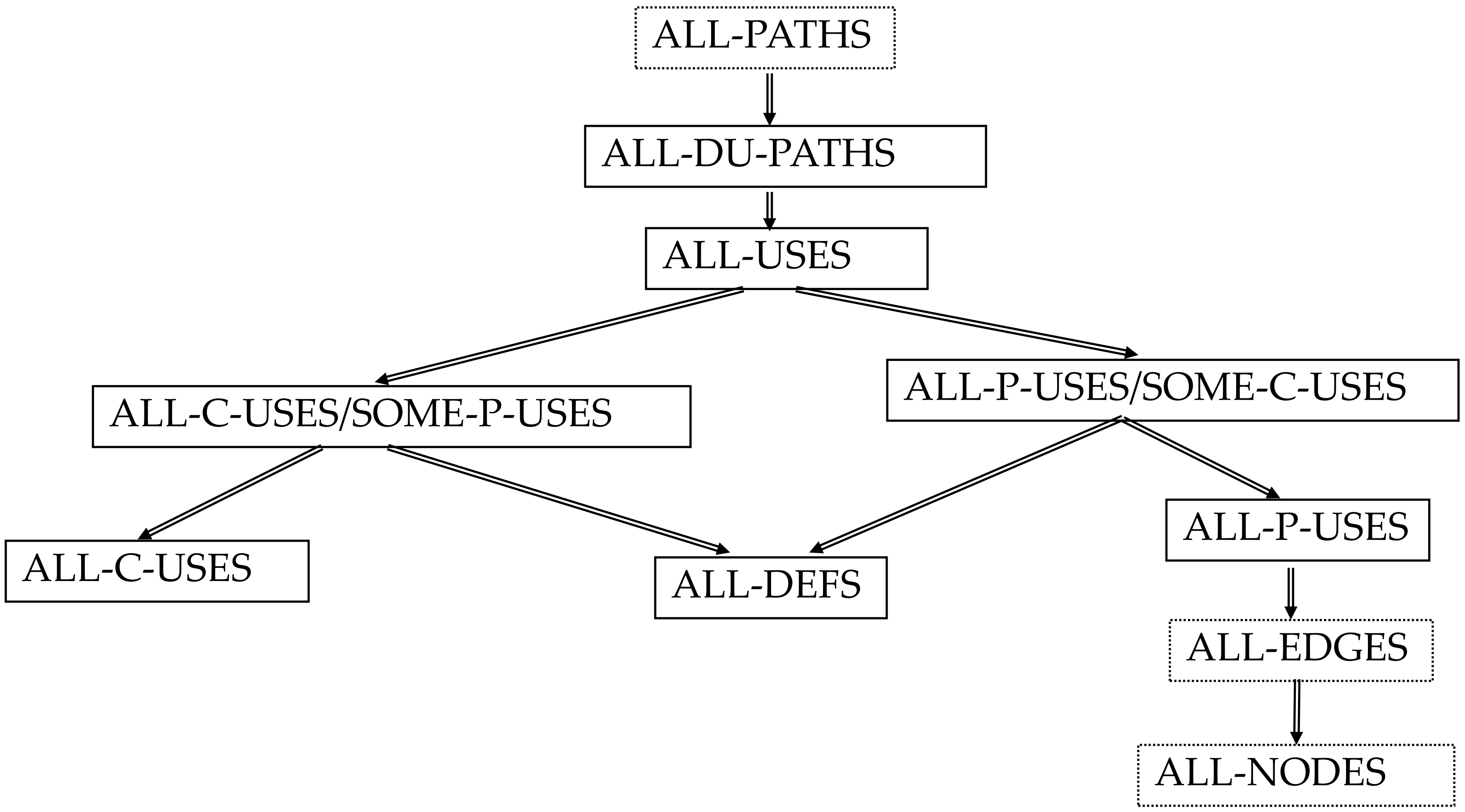
The def-path set for the use of $isub$ at node 10 is:

$$\begin{aligned} du(10, iSub) = \{&[10, 3, 4], [10, 3, 4, 5], [10, 3, 4, 5, 6, 7, 8], [10, 3, 4, 5, 6, 7, 9], \\ &[10, 3, 4, 5, 6, 10], [10, 3, 4, 5, 6, 7, 8, 10], [10, 3, 4, 10], \\ &[10, 3, 11]\} \end{aligned}$$

This def-path set can be broken up into the following def-pair sets:

$$\begin{aligned} du(10, 4, iSub) &= is\{[10, 3, 4]\} \\ du(10, 5, iSub) &= \{[10, 3, 4, 5]\} \\ du(10, 8, iSub) &= \{[10, 3, 4, 5, 6, 7, 8]\} \\ du(10, 9, iSub) &= \{[10, 3, 4, 5, 6, 7, 9]\} \\ du(10, 10, iSub) &= \{[10, 3, 4, 5, 6, 10], [10, 3, 4, 5, 6, 7, 8, 10], [10, 3, 4, 10]\} \\ du(10, 11, iSub) &= \{[10, 3, 11]\} \end{aligned}$$

Relationship among DF criteria



Coverage-based testing of requirements

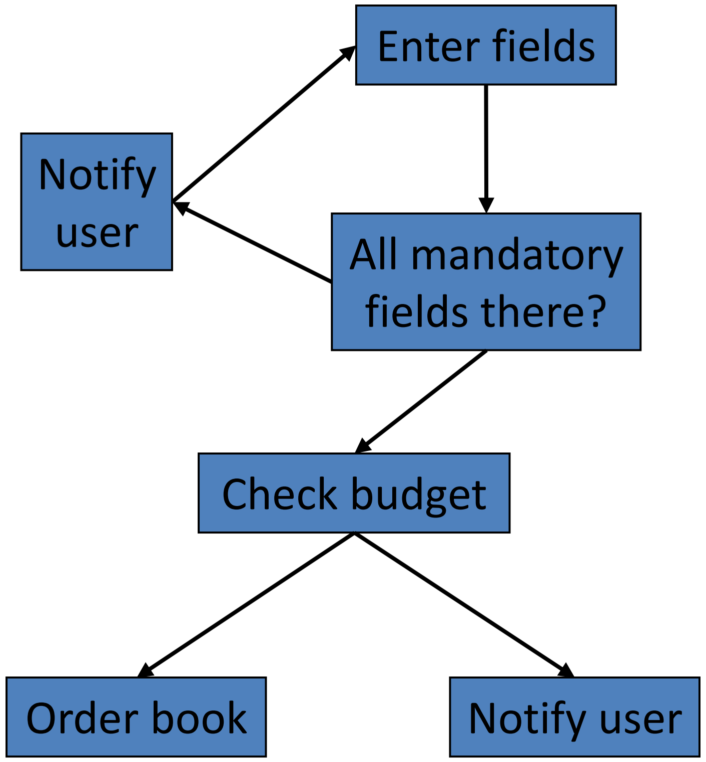
- Requirements may be represented as graphs, where the nodes represent elementary requirements, and the edges represent relations (like yes/no) between requirements.

- And next we may apply the earlier coverage criteria to this graph

- Discussed in detail by Boris Beizer in his book

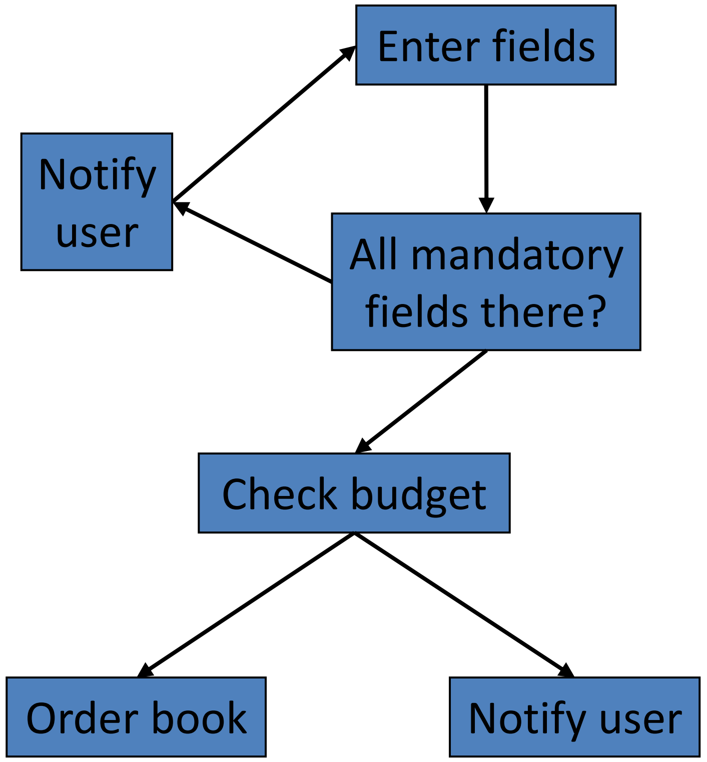
Example translation of requirements to a graph

A user may order new books. He is shown a screen with fields to fill in. Certain fields are mandatory. One field is used to check whether the department's budget is large enough. If so, the book is ordered and the budget reduced accordingly.



Similarity with Use Case success scenario

1. User fills form
2. Book info checked
3. Dept budget checked
4. Order placed
5. User is informed



Summary

- Data are as important as code.
- Define what you consider to be a data-flow anomaly.
- Data-flow testing strategies span the gap between **all paths** and **branch** testing.