

---

# Data-flow Testing

Laurie Williams  
North Carolina State University  
williams@csc.ncsu.edu

---

# Data-flow Testing

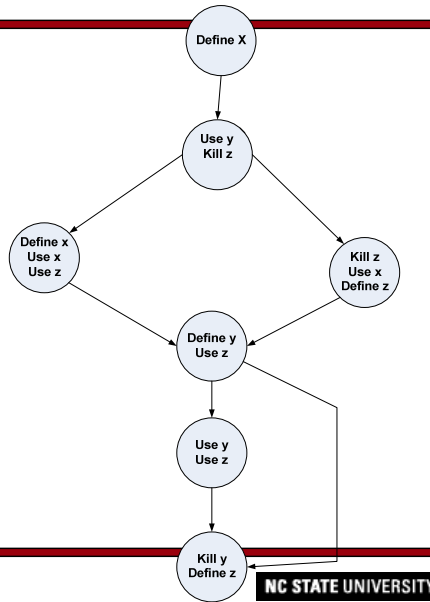
- Most failures involve\*:
  - Execution of an incorrect definition
    - Incorrect assignment or input statement
    - Definition is missing (use of null definition)
    - Predicate is faulty (incorrect path is taken which leads to incorrect definition)
- At least half of the source code consists of data declaration or definition statements
- Need to focus some testing effort on these as well (not a focus of coverage-based testing)
  - Explore the sequences of events related to the data state and the unreasonable things that can happen to data.
  - Explore the effect of using the value produced by each and every computation

## Data flow graph

### Annotations/"Link Weights"

- d – defined, created, initialized  
Data declaration; on left hand side of computation
- k – killed, undefined, released
- u – used for something (=c and p)
- c - Right hand side of computation, pointer (calculation)
- p - Used in a predicate (or as control variable of loop)

Control flowgraph's structure is the same for every variable; the weights change for each variable.



## Control flow statement classification I

- $v = \text{expression}$ 
  - c-use of all variables in expression
  - definition of  $v$
- $\text{read}(v_1, v_2, \dots, v_n)$ 
  - definitions of  $v_1 \dots v_n$
- $\text{write}(v_1, v_2, \dots, v_n)$ 
  - c-uses of  $v_1 \dots v_n$
- method call:  $P(c_1, \dots, c_n)$ 
  - definition of each formal parameter
- while: while B do S
  - p-use of each variable in boolean expression (B)

## Control flow statement classification II

---

- for statement: for (v=e1 to e2)
    - c-use of each variable in e1 . . . e2
      - definition of v
      - p-use of v
  - if-then-else: if B then S1; if B then S1 else S2
    - p-use of each variable in boolean expression (B)
      - S1 and S2 classified depending upon their composition
  - case: case e1
    - S1:
    - S2:
    - p-use of each variable in expression e1
    - S1 and S2 classified depending upon their composition
- 

## Time-sequence pairs of d, k, and u

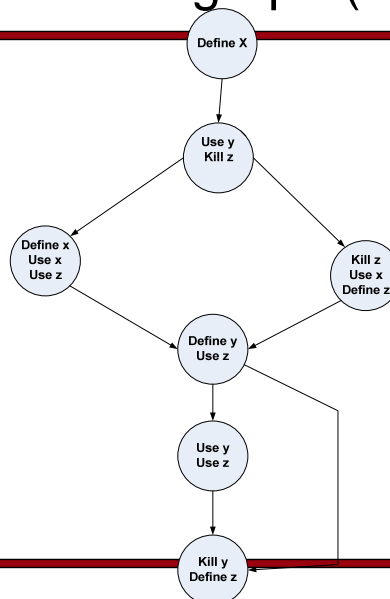
---

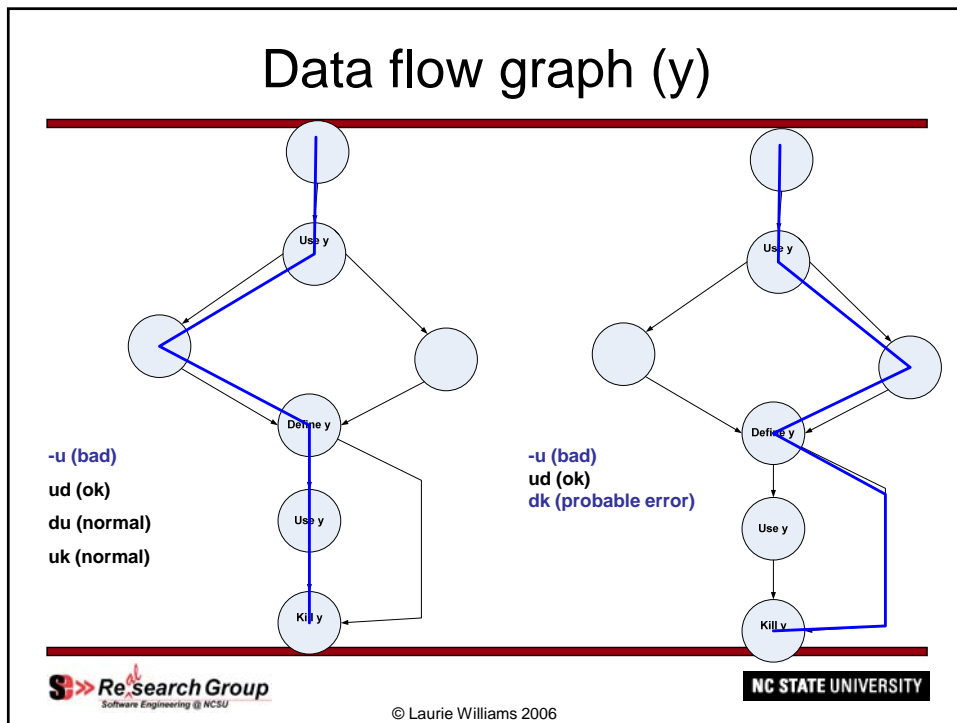
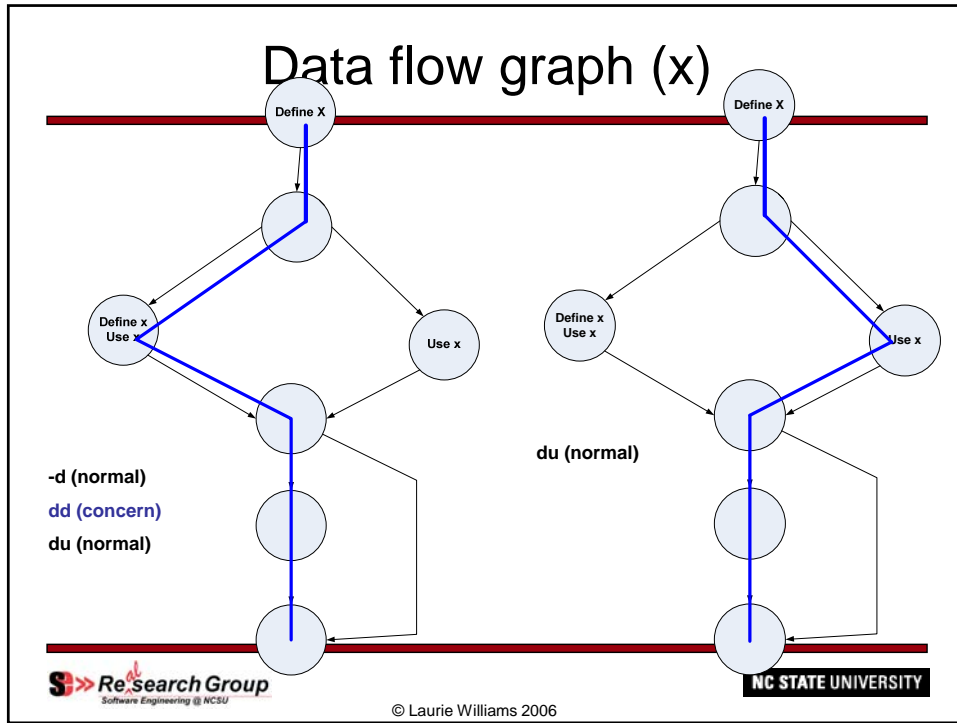
- dd – why define twice without use?
  - dk -- why define and kill without use?
  - ku – a bug; killed then used
  - kk – harmless but probably a bug
  - du – normal (a “du pair”)
  - kd – normal (kill then redefine)
  - ud – usually normal, reassignment after use
  - uk – normal
  - uu – normal
- Often these anomalies are caught by compilers or static analyzers today – depending upon the language
-

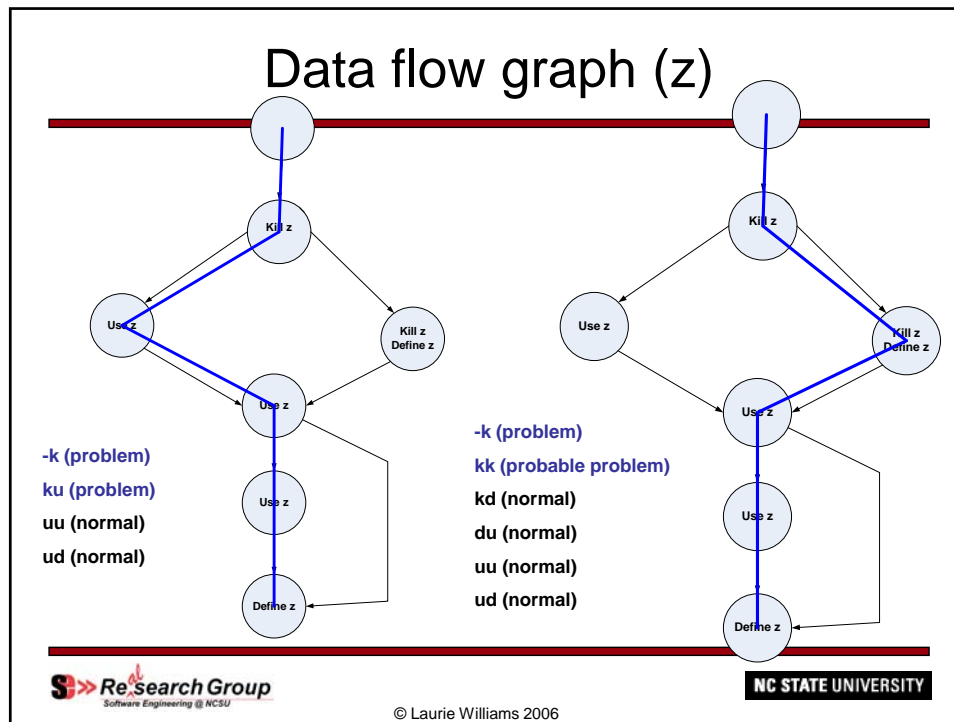
## Data-flow Anomalies

- Where dash means that nothing of interest (d, k, u) happens
- -k – variable not defined but killed
- -d – OK, first definition in path
- -u – used before defined
- k- – normal; last thing done is to kill variable
- d- – defined but never used
- These are often integration testing issues.

## Data flow graph (all)



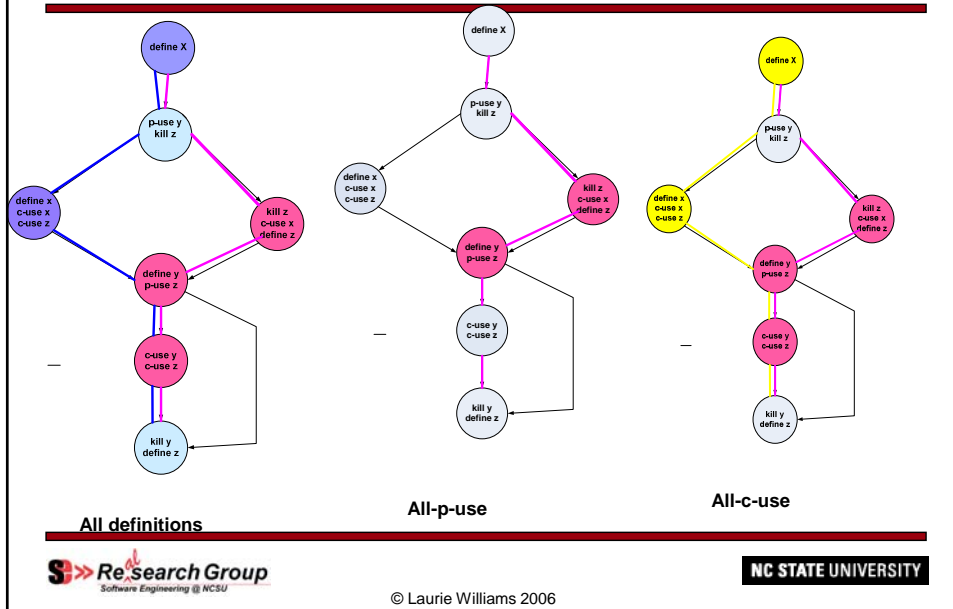




## Strategies - 1

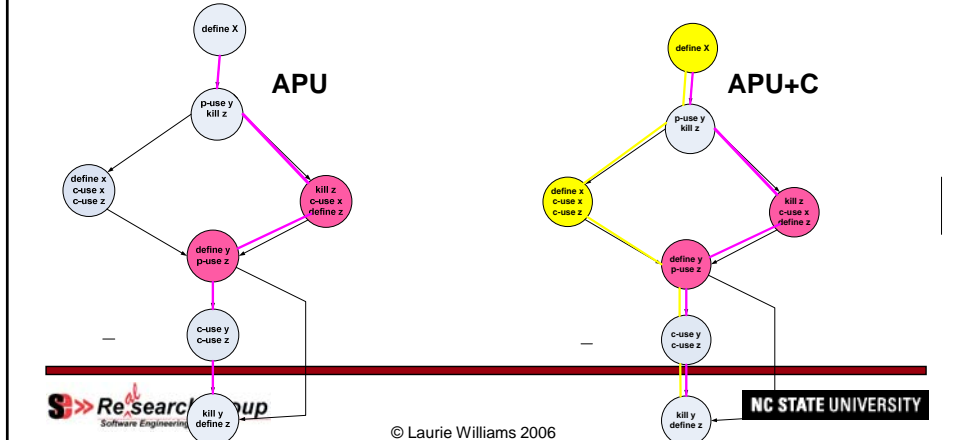
- **All-definitions (AD)**: Test cases are generated to cover each definition of each variable for at least one use of the variable.
- **All predicate-uses (all p-use, APU)**: Test cases are generated so that there is at least one path of each variable definition to each p-use of the variable.
- **All-computational-uses (all c-use, ACU)**: Test cases are generated so that there is at least one path of each variable definition to each c-use of the variable.

# Strategies - 1



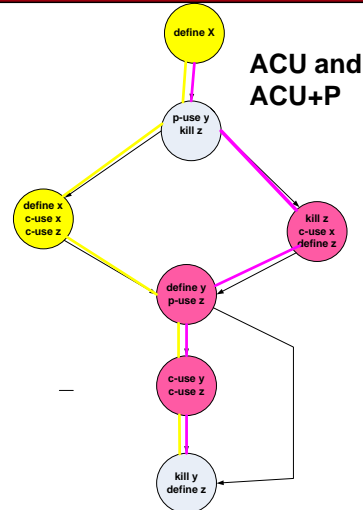
# Strategies - 2

**All-p-uses/some-c-uses (APU+C):** Test cases are generated so that there is at least one path of each variable definition to each p-use of the variable. If there are any variable definitions that are not covered, use c-uses.



## Strategies - 3

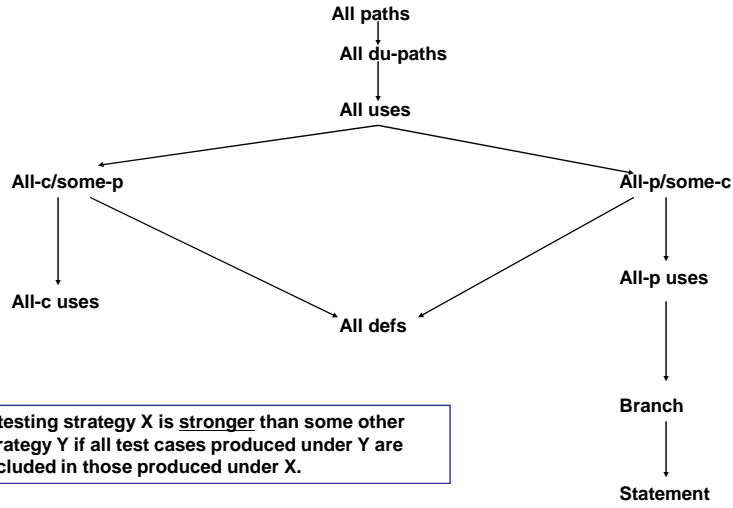
All-c-uses/some-p-uses (ACU+P): Test cases are generated so that there is at least one path of each variable definition to each c-use of the variable. If there are any variable definitions that are not covered, use p-uses.



## Strategies - 4

- All-uses (AU): Test cases are generated so that there is at least one path of each variable definition to each p-use and each c-use use of the definition.
- All-du paths (ADUP): Test cases are generated which cause the traversal of every simple subpath from each variable definition to every p-use and every c-use of that definition.
  - the strongest data-flow testing strategy.

# Relative strength of white box test strategies



A testing strategy X is **stronger** than some other strategy Y if all test cases produced under Y are included in those produced under X.