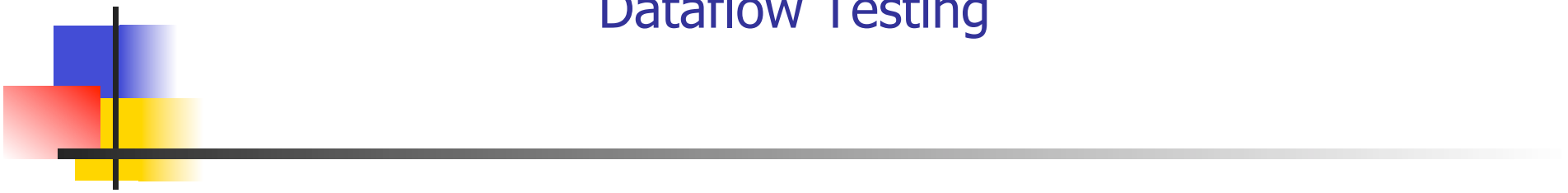
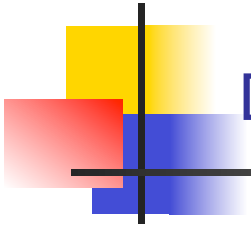


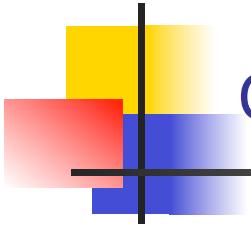
Dataflow Testing

Chapter 10



Dataflow Testing

- Testing All-Nodes and All-Edges in a control flow graph may miss significant test cases
- Testing All-Paths in a control flow graph is often too time-consuming
- Can we select a subset of these paths that will reveal the most faults?
- Dataflow Testing focuses on the points at which variables receive values and the points at which these values are used

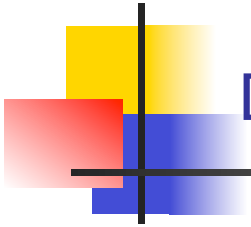


Concordances

- Data flow analysis is in part based concordance analysis such as that shown below – the result is a variable cross-reference table

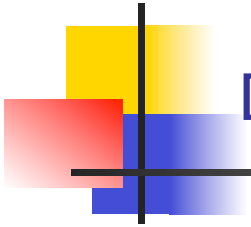
```
 18   beta ← 2
 25   alpha ← 3 × gamma + 1
 51   gamma ← gamma + alpha - beta
123   beta ← beta + 2 × alpha
124   beta ← gamma + beta + 1
```

	Defined	Used
alpha	25	51, 123
beta	18, 123, 124	51, 123, 124
gamma	51	25, 51, 124



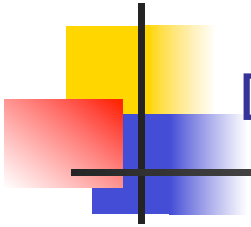
Dataflow Analysis

- 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
  - Sending a modifier message to an object more than once between accesses
  - Deallocating a variable before it used
    - **Container problem – deallocating container loses references to items in the container, memory leak**



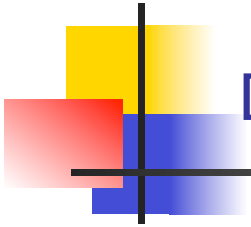
Dataflow Analysis – 2

- The bugs can be found from a cross-reference table using **static analysis**
- Paths from the definition of a variable to its use are more likely to contain bugs



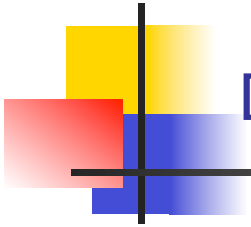
Definitions

- A node **n** in the program graph is a **defining** node for variable **v** – **DEF(v, n)** – if the value of **v** is defined at the statement fragment in that node
  - Input, assignment, procedure calls
- A node in the program graph is a **usage** node for variable **v** – **USE(v, n)** – if the value of **v** is used at the statement fragment in that node
  - Output, assignment, conditionals



Definitions – 2

- In languages without garbage collection
  - A node in the program is a **kill** node for a variable **v** – **KILL(v, n)** – if the variable is deallocated at the statement fragment in that node.
- A usage node is a predicate use, **P-use**, if variable v appears in a predicate expression
  - Always in nodes with outdegree ≥ 2
- A usage node is a computation use, **C-use**, if variable v appears in a computation
  - Always in nodes with outdegree ≤ 1

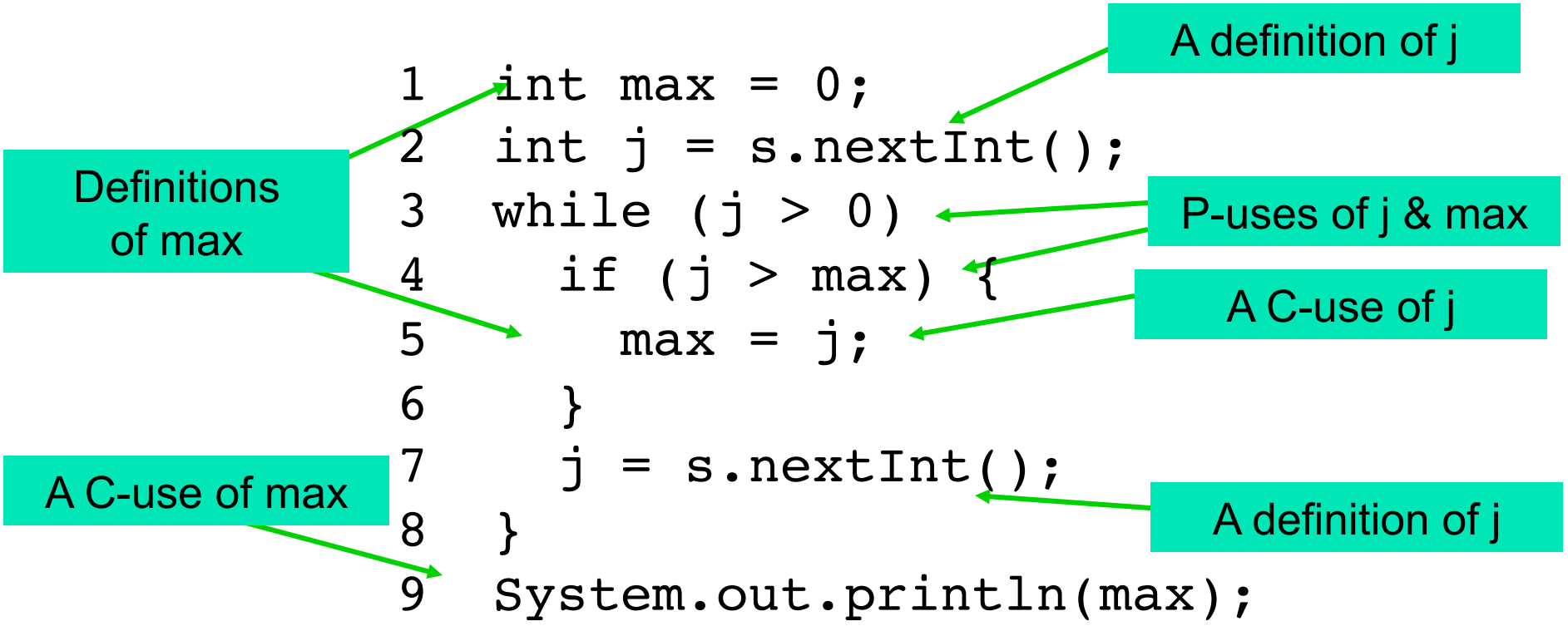


Definitions – 3

- 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**

Example 1 – Max program



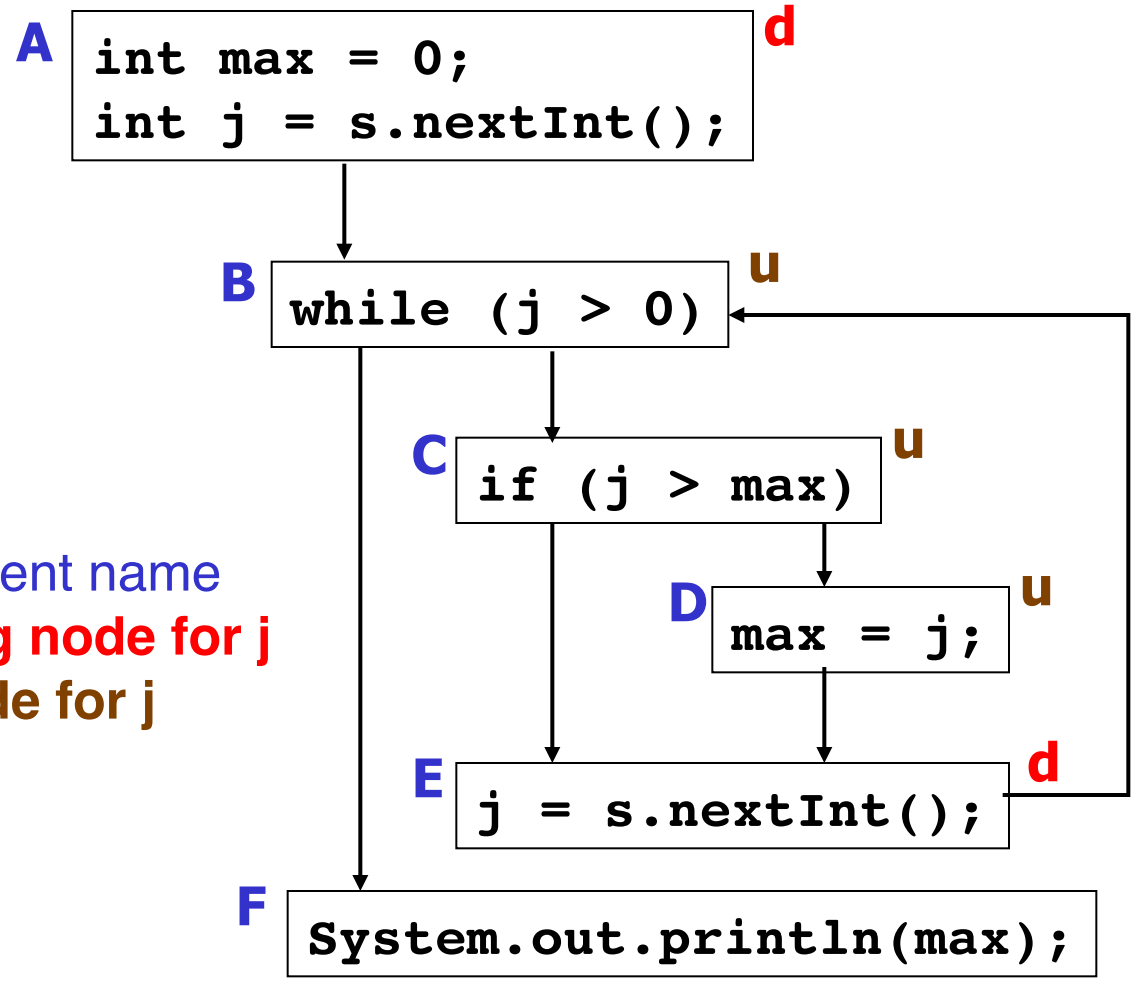

```
1   int max = 0;
2   int j = s.nextInt();
3   while (j > 0)
4     if (j > max) {
5       max = j;
6     }
7     j = s.nextInt();
8   }
9   System.out.println(max);
```

Example 2 – Billing program

```
calculateBill (usage : INTEGER) : INTEGER
double bill = 0;

if usage > 0 then bill = 40 fi
if usage > 100
then if usage ≤ 200
        then bill = bill + (usage – 100) *0.5
        else bill = bill + 50 + (usage – 200) * 0.1
             if bill ≥ 100 then bill = bill * 0.9 fi
     fi
fi
return bill
end
```

Max program – analysis



Legend

A..F Segment name

d defining node for j

u use node for j

dc-paths j

A B

A B C

A B C D

E B

E B C

E B C D

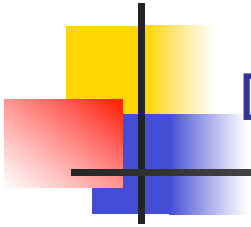
dc-paths max

A B F

A B C

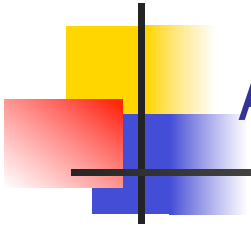
D E B C

D E B F



Dataflow Coverage Metrics

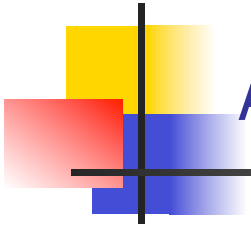
- Based on these definitions we can define a set of coverage metrics for a set of test cases
- We have already seen
  - All-Nodes
  - All-Edges
  - All-Paths
- Data flow has additional test metrics for a set T of paths in a program graph
  - All assume that all paths in T are feasible



All-Defs Criterion

- The set T satisfies the All-Def criterion
  - For every variable v, T contains a dc-path from every defining node for v to at least one usage node for v
    - **Not all use nodes need to be reached**

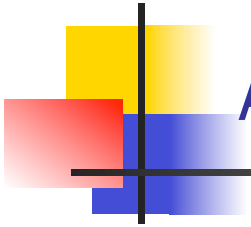
$$\forall v \in V(P), nd \in prog\_graph(P) \mid DEF(v, nd)$$
$$\bullet \exists nu \in prog\_graph(P) \mid USE(v, nu) \bullet dc\_path(nd, nu) \in T$$



All-Uses Criterion

- The set T satisfies the All-Uses criterion iff
  - For every variable v, T contains dc-paths that start at every defining node for v, and terminate at every usage node for v
    - **Not DEF(v, n) × USE(v, n) – not possible to have a dc-path from every defining node to every usage node**

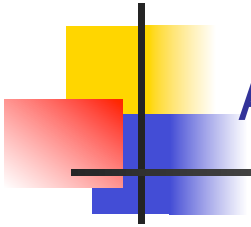
$$(\forall v \in V(P), nu \in prog\_graph(P) \mid USE(v, nu)$$
$$\bullet \exists nd \in prog\_graph(P) \mid DEF(v, nd) \bullet dc\_path(nd, nu) \in T)$$
$$\wedge$$
$$all\_defs\_criterion$$



All-P-uses / Some-C-uses

- The set T satisfies the All-P-uses/Some-C-uses criterion iff
  - For every variable v in the program P, T contains a dc-path from every defining node of v to every P-use node for v
    - **If a definition of v has no P-uses, a dc-path leads to at least one C-use node for v**

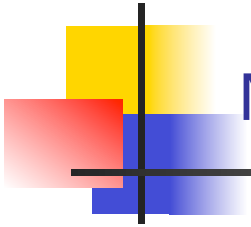
$$(\forall v \in V(P), nu \in prog\_graph(P) \mid P\_use(v, nu)$$
$$\bullet \exists nd \in prog\_graph(P) \mid DEF(v, nd) \bullet dc\_path(nd, nu) \in T)$$
$$\wedge$$
$$all\_defs\_criterion$$



All-C-uses / Some-P-uses

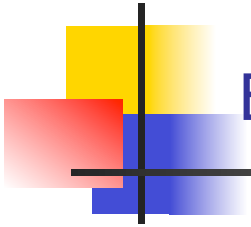
- The test set T satisfies the All-C-uses/Some-P-uses criterion iff
  - For every variable v in the program P, T contains a dc-path from every defining node of v to every C-use of v
    - **If a definition of v has no C-uses, a dc-path leads to at least one P-use**

$$(\forall v \in V(P), nu \in prog\_graph(P) \mid C\_use(v, nu)$$
$$\bullet \exists nd \in prog\_graph(P) \mid DEF(v, nd) \bullet dc\_path(nd, nu) \in T)$$
$$\wedge$$
$$all\_defs\_criterion$$



Miles-per-gallon Program

```
miles_per_gallon ( miles, gallons, price : INTEGER )
if gallons = 0 then
   // Watch for division by zero!!
   Print("You have " + gallons + "gallons of gas")
else if miles/gallons > 25
   then print( "Excellent car.  Your mpg is "
                + miles/gallon)
   else print( "You must be going broke.  Your mpg is "
               + miles/gallon + " cost " + gallons * price)
fi
end
```



Example du-paths

- For each variable in the miles_per_gallon program see the test paths for the following dataflow path sets
  - All-Defs (AD)
  - All-C-uses (ACU)
  - All-P-uses (APU)
  - All-C-uses/Some-P-uses (ACU+P)
  - All-P-uses/Some-C-uses (APU+C)
  - All-uses

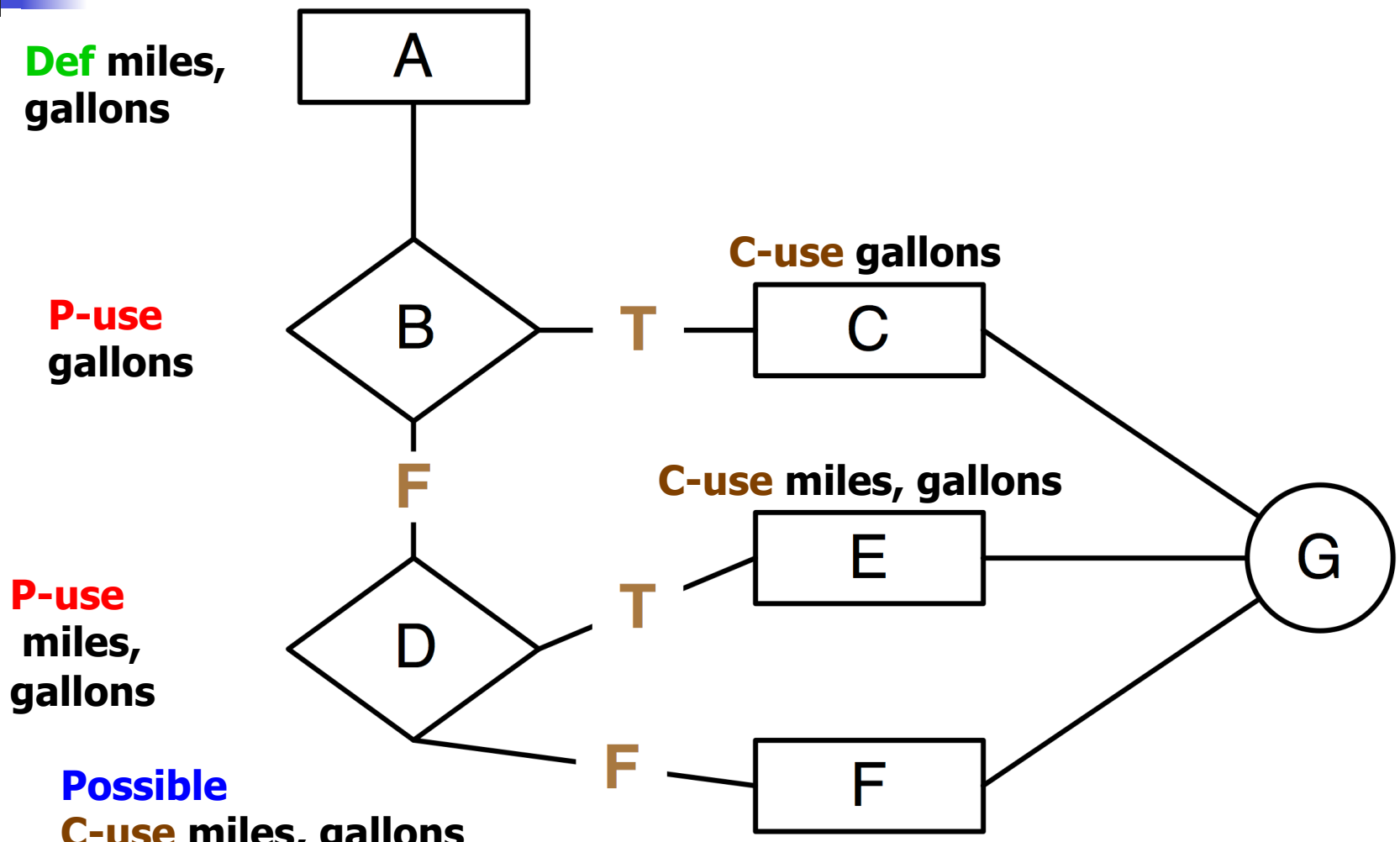
Mile-per-gallon Program – Segmented

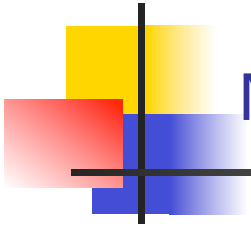
gasguzzler (miles, gallons, price : INTEGER)	A
if gallons = 0 then	B

// Watch for division by zero!!
Print("You have " + gallons + "gallons of gas")
	C

else if miles/gallons > 25	D
then print("Excellent car. Your mpg is " + miles/gallon)	E
else print("You must be going broke. Your mpg is " + miles/gallon + " cost " + gallons * price)	F
fi end	G

MPG program graph



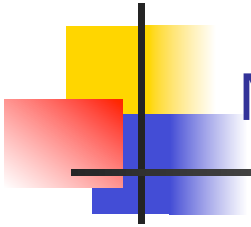


MPG – DU-Paths for Miles

- All-Defs
  - Each definition of each variable for at least one use of the definition
    - **A B D**
- All-C-uses
  - At least one path of each variable to each c-use of the definition
    - **A B D E** **A B D F** **A B D**
- All-P-uses
  - At least one path of each variable definition to each p-use of the definition
    - **A B D**

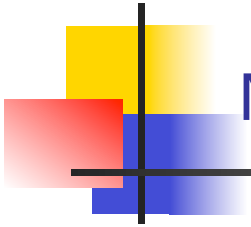
MPG – DU-Paths for Miles – 2

- All-C-uses/Some-P-uses
  - At least one path of each variable definition to each c-use of the variable. If any variable definitions are not covered use p-use
    - **A B D E** **A B D F** **A B D**
- All-P-uses/Some-C-uses
  - At least one path of each variable definition to each p-use of the variable. If any variable definitions are not covered use c-use
    - **A B D**
- All-uses
  - At least one path of each variable definition to each p-use and each c-use of the definition
    - **A B D** **A B D E** **A B D F**



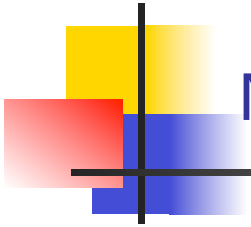
MPG – DU-Paths for Gallons

- All-Defs
  - Each definition of each variable for at least one use of the definition
    - **A B**
- All-C-uses
  - At least one path of each variable to each c-use of the definition
    - **A B C A B D E A B D F A B D**
- All-P-uses
  - At least one path of each variable definition to each p-use of the definition
    - **A B A B D**



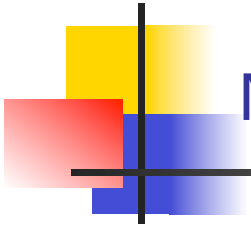
MPG – DU-Paths for Gallons – 2

- All-C-uses/Some-P-uses
  - At least one path of each variable definition to each c-use of the variable. If any variable definitions are not covered use p-use
    - **A B C A B D E A B D F A B D**
- All-P-uses/Some-C-uses
  - At least one path of each variable definition to each p-use of the variable. If any variable definitions are not covered use c-use
    - **A B A B D**
- All-uses
  - At least one path of each variable definition to each p-use and each c-use of the definition
    - **A B A B C A B D A B D E A B D F**



MPG – DU-Paths for Price

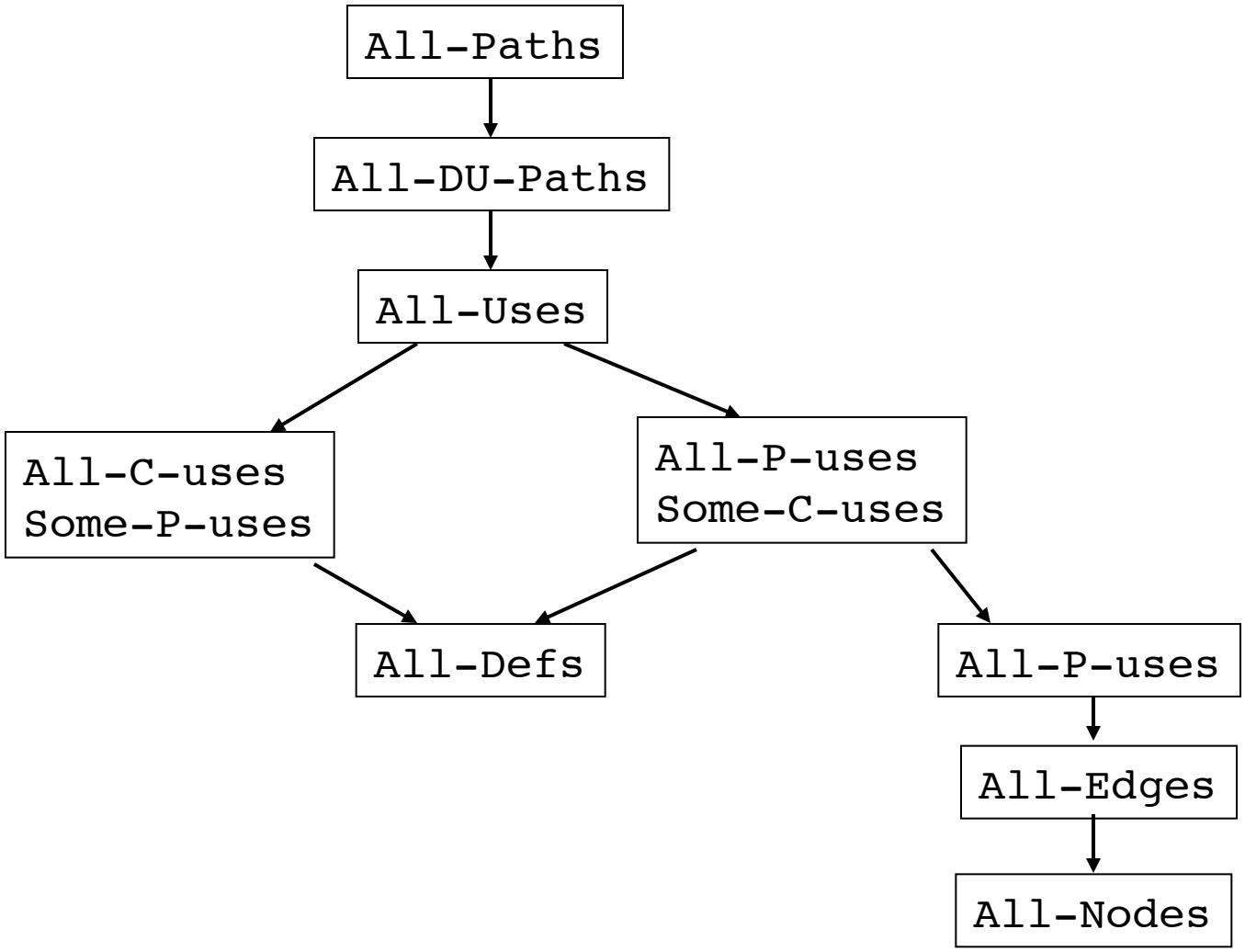
- All-Defs
  - Each definition of each variable for at least one use of the definition
    - **A B D F**
- All-C-uses
  - At least one path of each variable to each c-use of the definition
    - **A B D F**
- All-P-uses
  - At least one path of each variable definition to each p-use of the definition
    - **none**

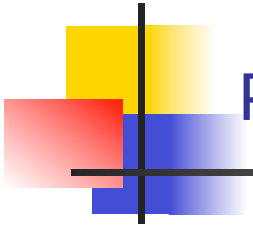


MPG – DU-Paths for Price – 2

- All-C-uses/Some-P-uses
  - At least one path of each variable definition to each c-use of the variable. If any variable definitions are not covered use p-use
    - **A B D F**
- All-P-uses/Some-C-uses
  - At least one path of each variable definition to each p-use of the variable. If any variable definitions are not covered use c-use
    - **A B D F**
- All-uses
  - At least one path of each variable definition to each p-use and each c-use of the definition
    - **A B D F**

Rapps-Weyuker data flow hierarchy

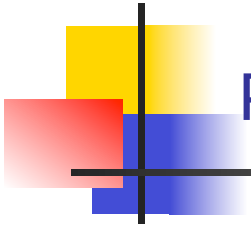




Potential Anomalies

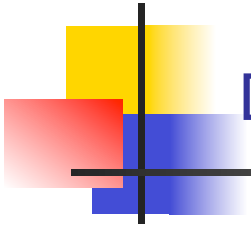
Data flow node combinations for a variable

Anomalies		Explanation
~ d	first define	Allowed
du	define-use	Allowed - normal case
dk	define-kill	Potential bug
~ u	first use	Potential bug
ud	use-define	Allowed - redefined
uk	use-kill	Allowed
~ k	first kill	Potential bug
ku	kill-use	Serious defect



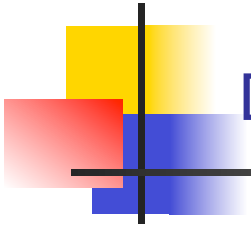
Potential Anomalies – 2

Anomalies		Explanation
kd	kill-define	Allowed - redefined
dd	define-define	Potential bug
uu	use-use	Allowed - normal case
kk	kill-kill	Potential bug
d ~	define last	Potential bug
u ~	use last	Allowed
k ~	kill last	Allowed - normal case



Data flow guidelines

- Data flow testing is good for computationally/control intensive programs
  - If P-use of variables are computed, then P-use data flow testing is good
- Define/use testing provides a rigorous, systematic way to examine points at which faults may occur.



Data flow guidelines – 2

- Aliasing of variables causes serious problems!
- Working things out by hand for anything but small methods is hopeless
- Compiler-based tools help in determining coverage values