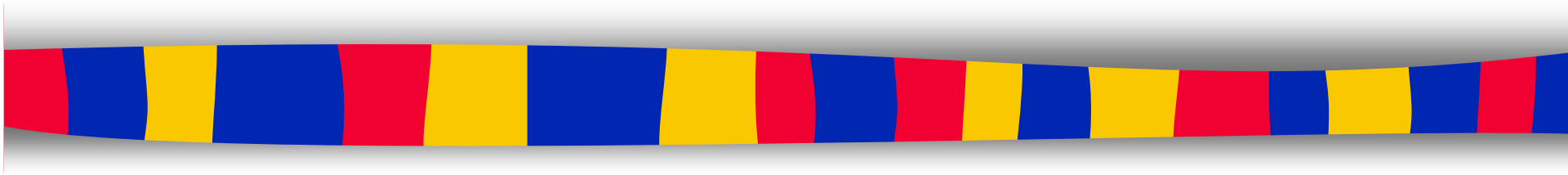
EECS 4313

Software Engineering Testing



Topic 08:

Dataflow Testing and Static Analysis

Zhen Ming (Jack) Jiang

Relevant Readings

- [Jorgensen] chapter 9
- [Ammann & Offutt] chapter 7

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
  - Goal: try to ensure that values are computed and used correctly

Dataflow Analysis

- 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
  - De-allocating a variable before it is used
    - Container problem
      - De-allocating container loses references to items in the container, memory leak

Definitions

- A node **n** in the program graph is a **defining** node for variable **v**, written as **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**, written as **USE(v, n)**, if the value of **v** is used at the statement fragment in that node
  - Output, assignment, conditionals
- 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
- A node in the program is a **kill** node for a variable **v**, written as **KILL(v, n)**, if the variable is deallocated at the statement fragment in that node

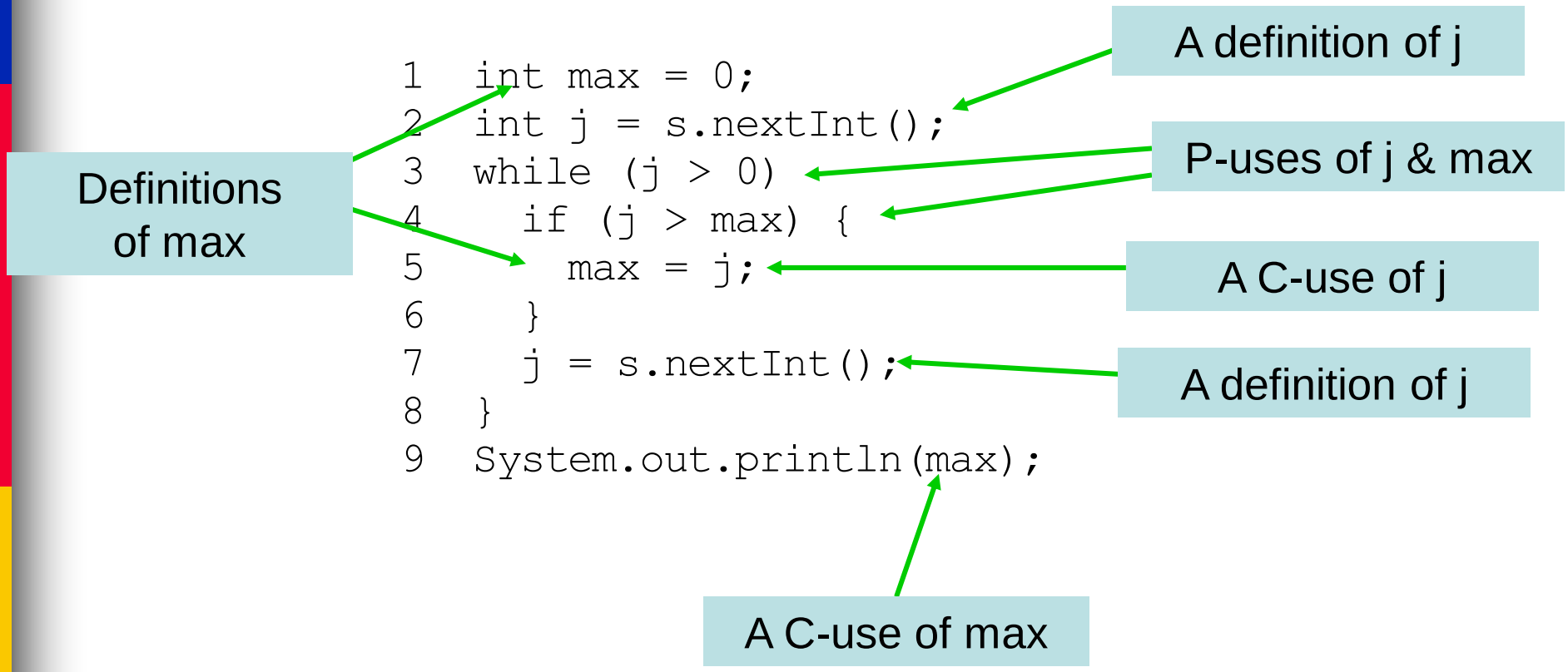
Example 2 – Billing program

```
public int calculateBill (int usage) {
  double bill = 0;

  if (usage > 0) { bill = 40; }

  if (usage > 100) {

    if (usage <= 200) { bill = bill + (usage – 100) *0.5; }
    else { bill = bill + 50 + (usage – 200) * 0.1; }

    if (bill >= 100) { bill = bill * 0.9; }
  }

  return bill;
}
```

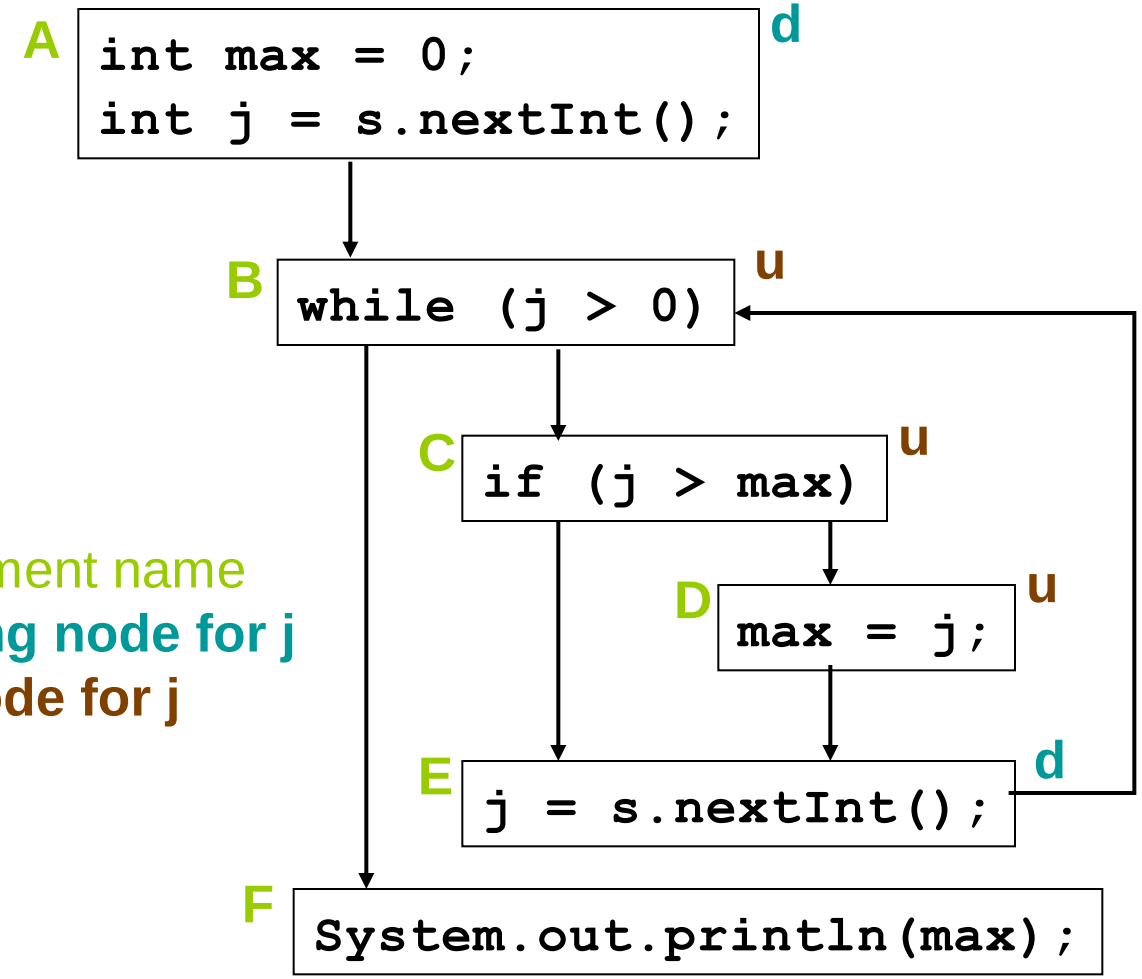
Definition-Use path

- What is a du-path (definition-use path)?
  - 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
- What is a dc-path (definition-clear path)?
  - A du-path with no other defining node for v is a definition-clear path

Example 1 – Max program



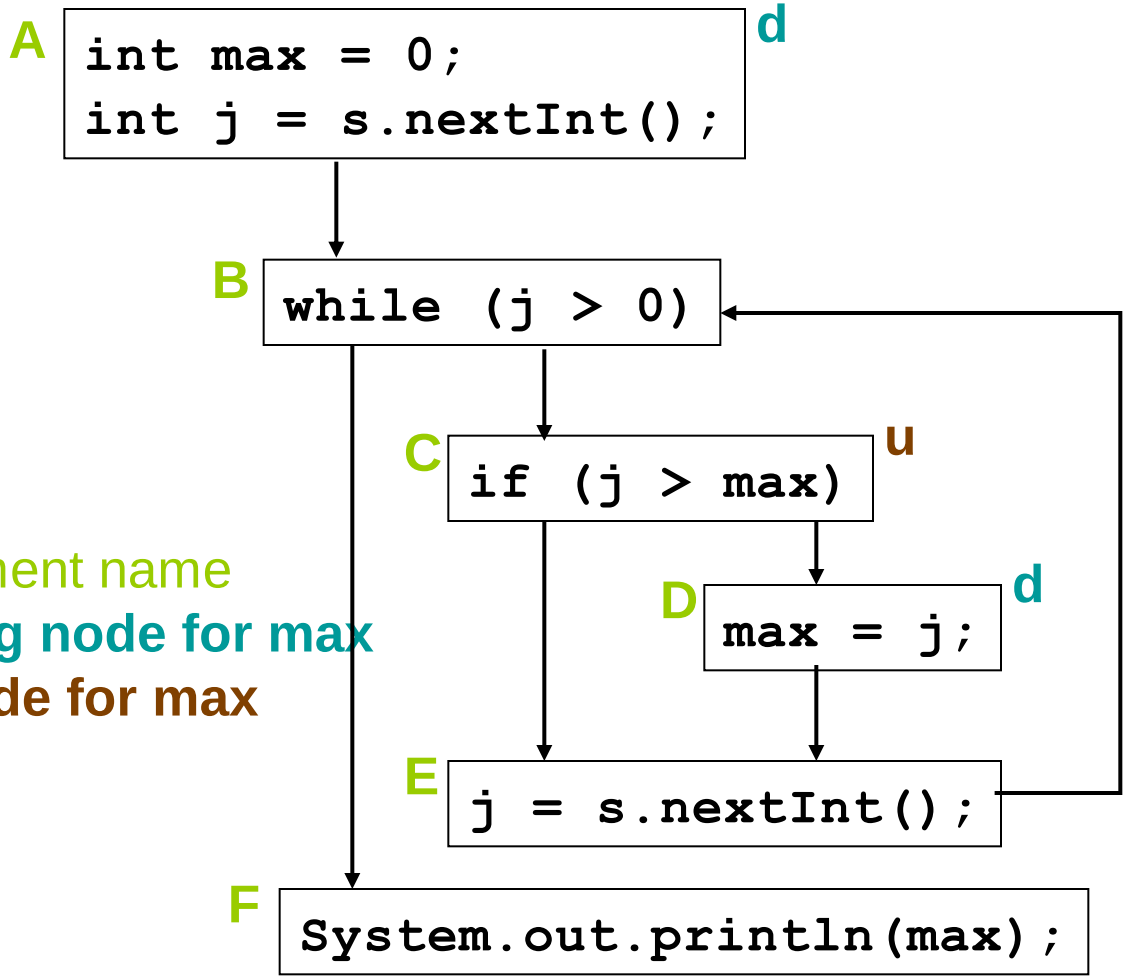
Max program – DC path analysis for j



dc-paths j
A B
A B C
A B C D
E B
E B C
E B C D

Legend
A..F Segment name
d defining node for j
u use node for j

Max program – DC path analysis for max



dc-paths max
A B F
A B C
D E B C
D E B F

Legend
A..F Segment name
d defining node for max
u use node for max

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
- Dataflow 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 iff
  - For every variable v in 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

    - T is the set of paths in the program graph
    - V is the set of variables

All-Uses Criterion

- The set T satisfies the All-Uses criterion iff
  - For every variable v in V, T contains dc-paths that start at every defining node for v, and terminate at **every** usage node for v
    - T is the set of paths in the program graph
    - V is the set of variables

- We cannot take the cross product of DEF and USE to define du-paths:
  - DEF(v, n) $\times$ USE(v, n)
    - Because it can result in infeasible paths

All-P-uses / Some-C-uses Criterion

- The set T satisfies the All-P-uses/Some-C-uses criterion iff
  - For every variable v in V for 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

    - T is the set of paths in the program graph
    - V is the set of variables

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 V for 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

    - T is the set of paths in the program graph
    - V is the set of variables

Miles-per-gallon Program

```
public void miles_per_gallon (int miles, int gallons, int price) {
  if (gallons == 0) {
    // Watch for division by zero!!
    System.out.println("You have " + gallons + "gallons of gas");
  } else if (miles/gallons > 25) {
    System.out.println( "Excellent car.  Your mpg is " + miles/gallon);
  } else {
    System.out.println( "You must be going broke.  Your mpg is "
                + miles/gallon + " cost " + gallons * price);
  }
}
```

- We want du- and dc-paths
- What do we do next then?

Mile-per-gallon (MPG) Program Segmented

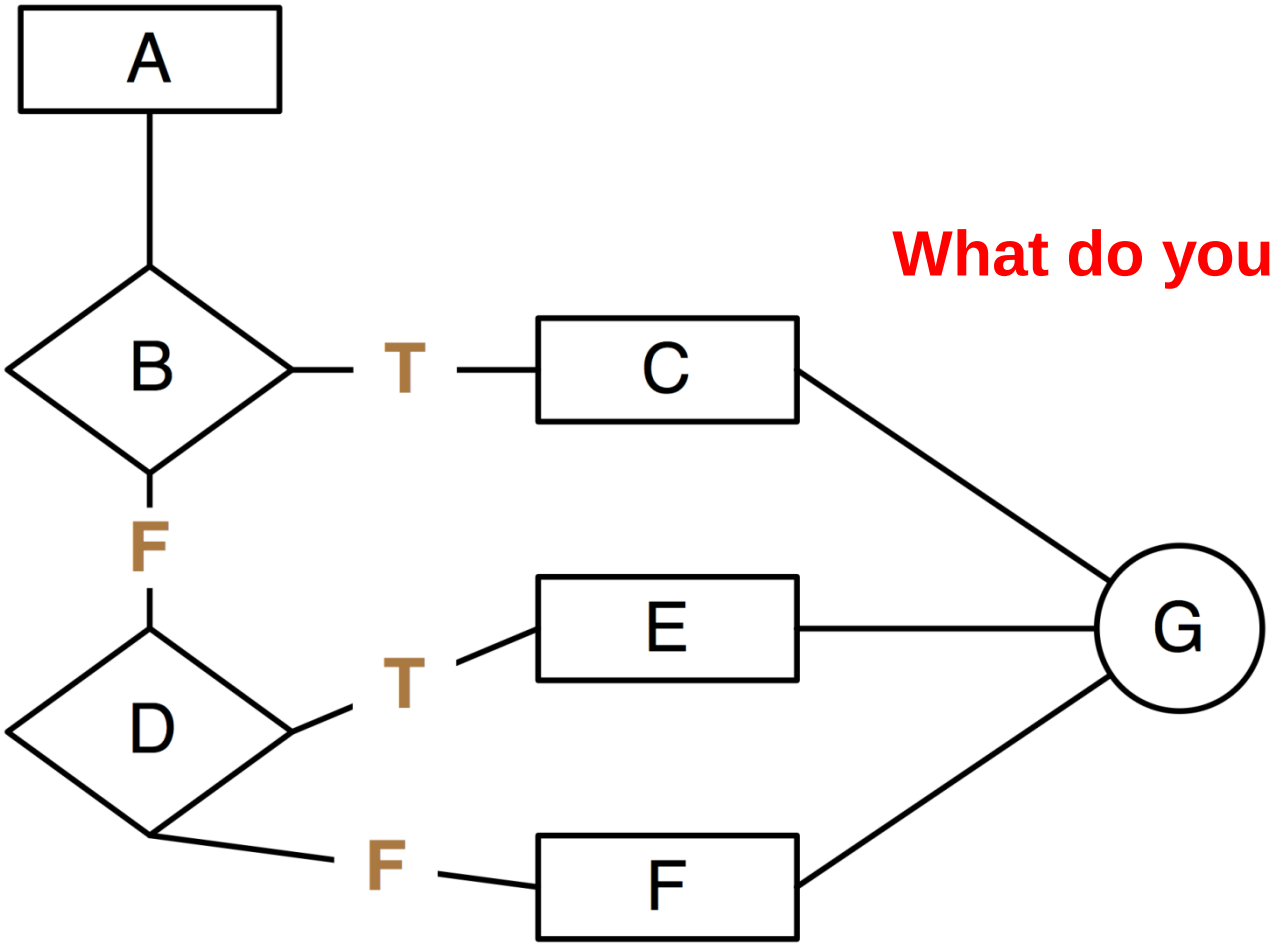
public void miles_per_gallon (int miles, int gallons, int price) {	A
if (gallons == 0) {	B

// Watch for division by zero!!
System.out.println("You have " + gallons + "gallons of gas");
	C

} else if (miles/gallons > 25) {	D
System.out.println("Excellent car. Your mpg is " + miles/gallon);	E
} else { System.out.println("You must be going broke. Your mpg is " + miles/gallon + " cost " + gallons * price);	F
} }	G

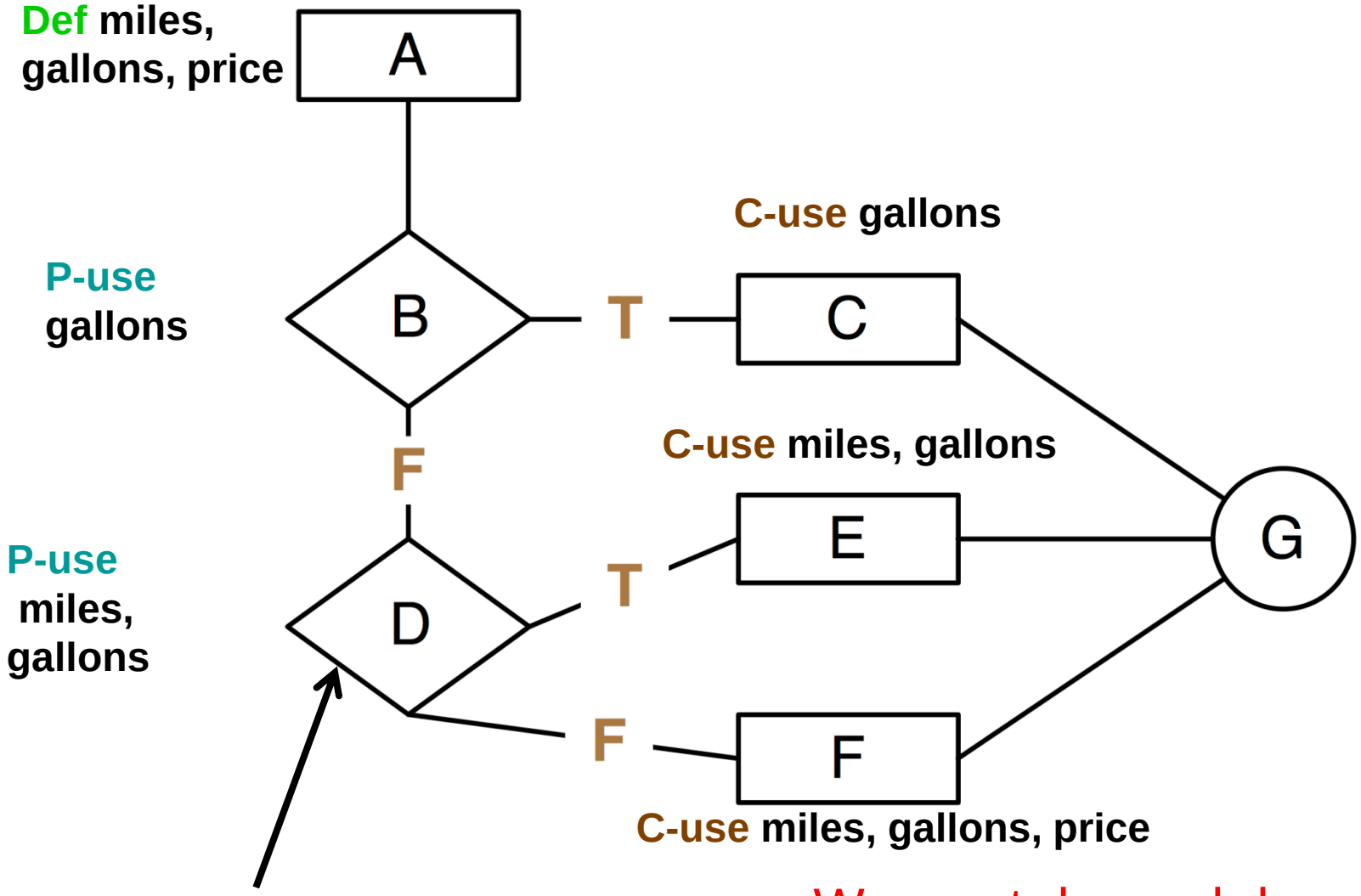
- We want du- and dc-paths
- What do we do next then?

MPG program graph



What do you do now?

MPG program graph



- We want du- and dc-paths
- What do we do next then?

Example du-paths

- For each variable in the miles_per_gallon program, create 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

MPG
du-Paths for Miles

- All-Defs
  - Each definition of each variable for at least one use of the definition
    - A B D (or ABDE, or ABDF)
- 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 last one path of each variable to each p-use of the definition
    - A B D
- 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
- 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 by p-use, then 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

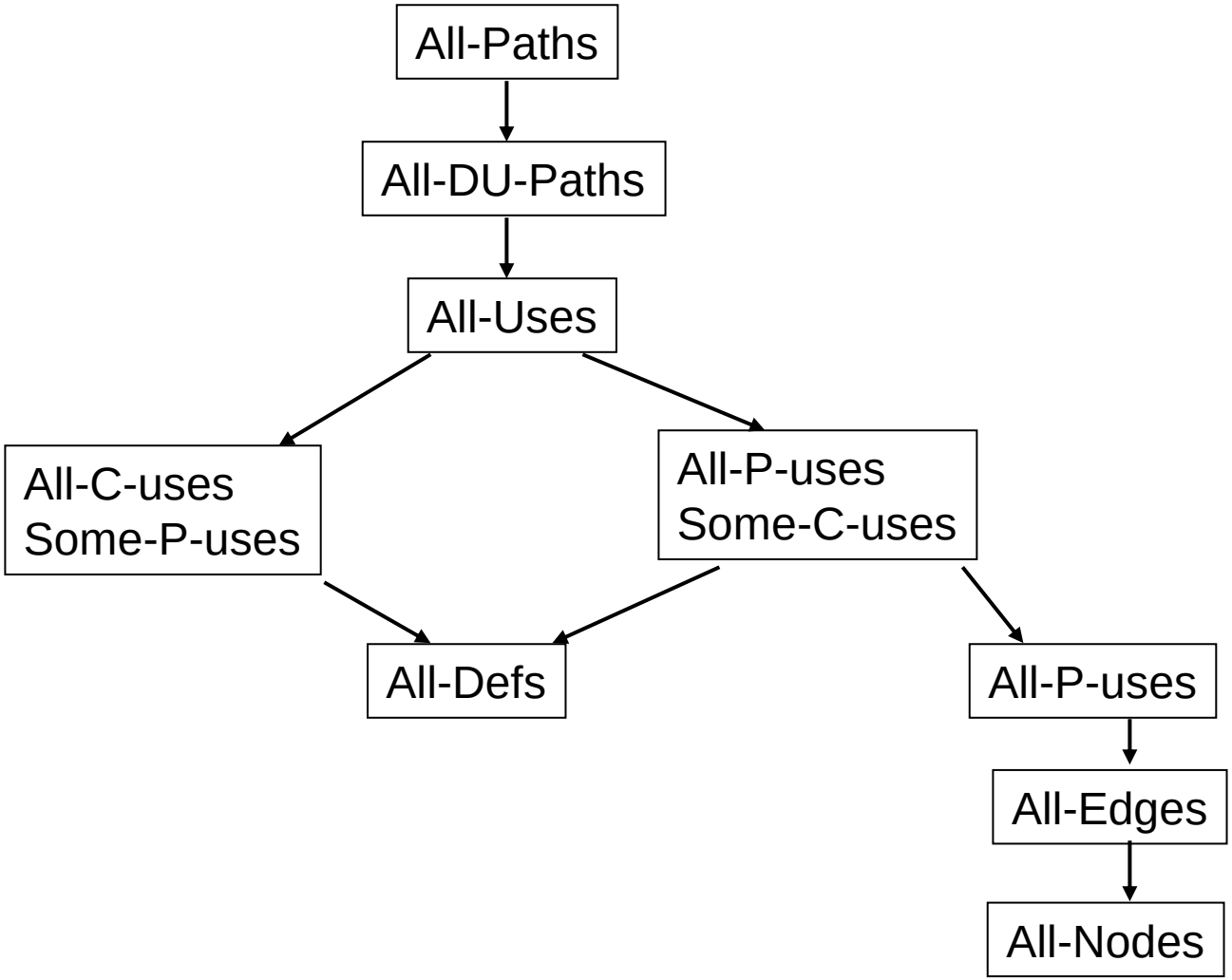
MPG
du-Paths for Gallons

- All-Defs
  - Each definition of each variable for at least one use of the definition
    - A B (or ABD, or ABC, or ABDE, or ABDF)
- 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
- 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 by c-use, then 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
- 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 hierarchy of data flow coverage metrics



Data flow guidelines

- When is dataflow analysis good to use?
  - 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.
- 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

Potential Anomalies – static analysis questions

Data flow node combinations for a variable

Allowed? – Potential Bug? – Serious defect?

Anomalies		Explanation
~ d	first define	???
du	define-use	???
dk	define-kill	???
~ u	first use	???
ud	use-define	???
uk	use-kill	???
~ k	first kill	???
ku	kill-use	???

Potential Anomalies – static analysis questions (continued)

Data flow node combinations for a variable

Allowed? – Potential Bug? – Serious defect?

Anomalies		Explanation
kd	kill-define	???
dd	define-define	???
uu	use-use	???
kk	kill-kill	???
d ~	define last	???
u ~	use last	???
k ~	kill last	???

Potential Anomalies – static analysis

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

Potential Anomalies – static analysis (continued)

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

A Brief Introduction on Static Analysis techniques

Information adapted from slides by Prof. Alex Orso and
http://examples.javacodegeeks.com/core-java/findbugs-eclipse-example/

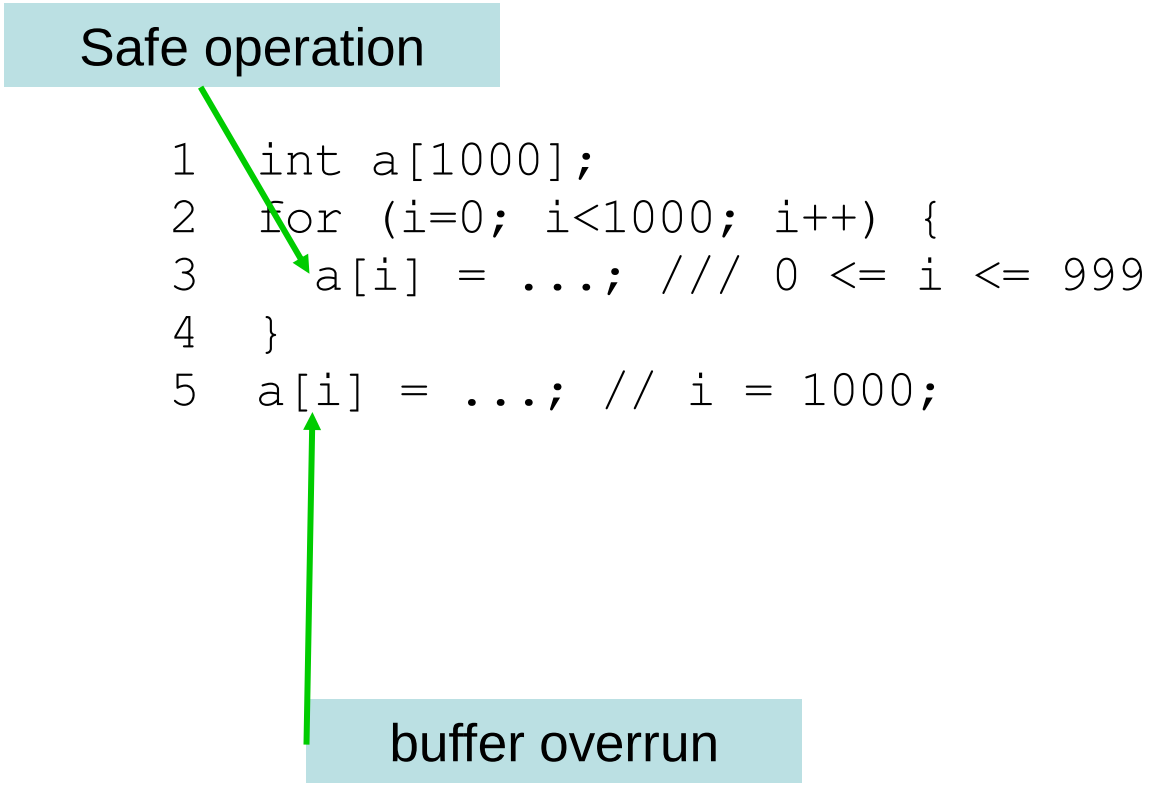
Static and dynamic verification

- Dynamic verification:
  - Concerned with exercising and observing software behaviour
  - The system is executed with test data and its operational behaviour is observed
  - Typically, testing
- Static verification:
  - Concerned with analysis of a static system representation
  - Various degrees of sophistication
  - Examples:
    - Inspections/reviews/walkthroughs
    - Static program analysis
- Different trade-offs between the static and dynamic verification
  - precision vs. recall
  - precision vs. cost
  - …

Automated Static Analysis

- Static analyses look at the program code and try to discover potentially erroneous conditions
- Can be very effective
- Typically complementary to testing
- Static verification checks that every operation of a program will never cause an error (e.g., division by zero, buffer overrun, deadlock, etc.)

Static verification example



Types of static analyses

- Control flow analysis
  - Finds unreachable code, compute complexity, etc.
- Data-flow analysis
  - Detects uninitialized variables, variables declared but never used, etc.
- Type analysis
  - Checks the program is type safe
- Interface analysis
  - Checks the consistency of routine and procedure declarations and their use
- Many of the above analysis can be performed by compliers nowadays

Data-flow Analysis

- Based on the identification of defs, uses, and data-flow anomalies
- Possible to define general rules
  - Variables should be defined before used
  - A variable should be used before redefined
  - A variable should be used after being defined
- Note: the violation of a rule does not necessarily indicate a fault
- Possible to extend flow analysis to other resources (e.g., file)
  - Opening (o), closing (c), reading (r), writing (w)
    - r must be preceded by o
    - w must be preceded by o
    - c must be preceded by o
    - …
  - In general, this type of flow analysis can be extended to all cases in which a program execution can be looked at as a sequence of actions that must occur according to a protocol

Use of static analysis

- Main advantage: exhaustive
- Main drawback: false positive
- Static analysis tools
  - FindBugs (http://findbugs.sourceforge.net/)
  - PMD (https://pmd.github.io/)
  - Coverity (http://www.coverity.com/)