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
```

	Assigned	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 looses references to items in the container, memory leak**
- These 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
- In languages without garbage collection
  - A node in the program grade is a **kill** node for a variable **v** – **KILL(v, n)** – if the variable is deallocated at the statement fragment in that node.
    - **In the following slide can define additional path types**

Definitions – 2

- 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 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 – program

```
int max = 0;
int j = s.nextInt();
while (j > 0)
  if (j > max) {
    max = j;
  }
  j = s.nextInt();
}
System.out.println(max);
```

Definitions of max

A C-use of max

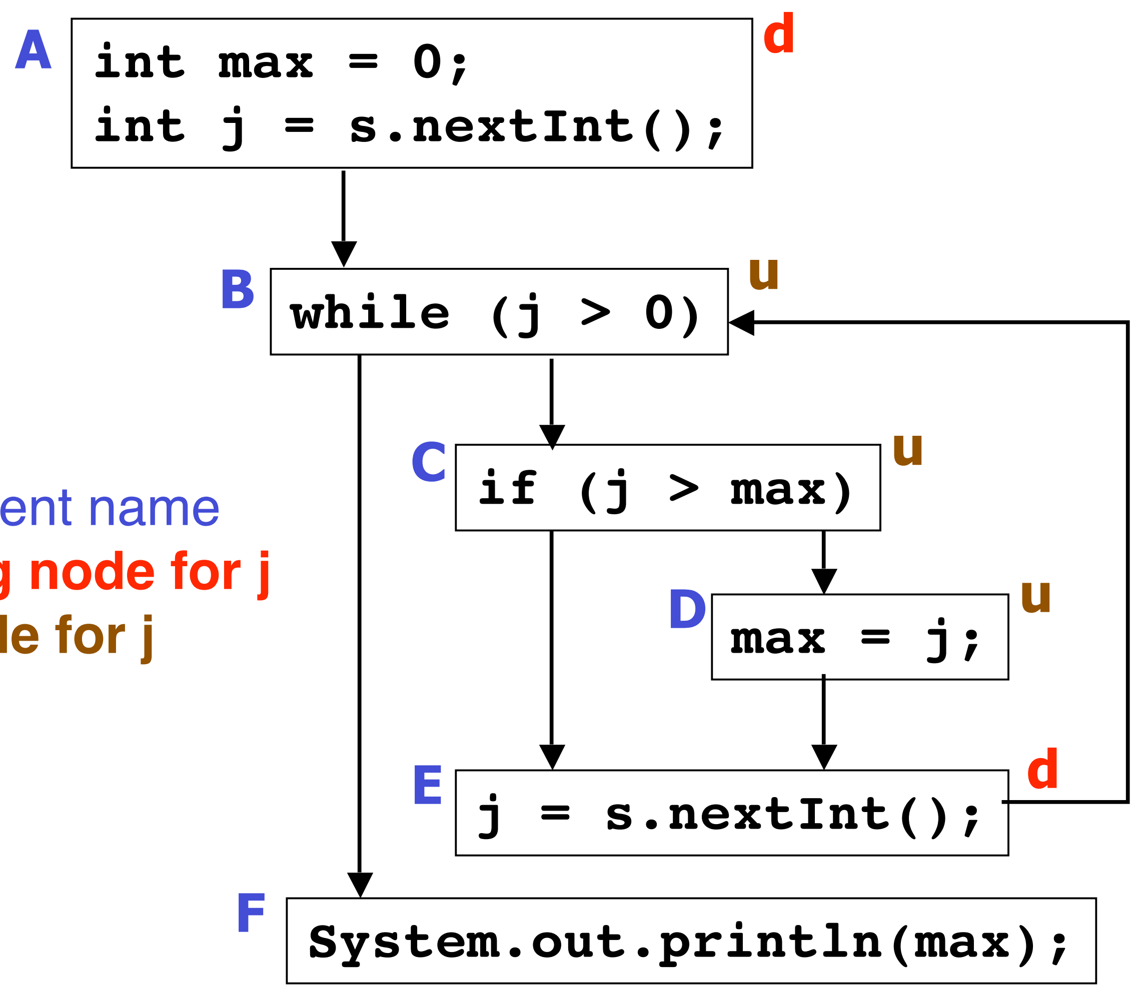
A definition of j

P-uses of j

A C-use of j

A definition of j

Example 1 – analysis



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

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

dc-paths max
A F
A C
D C
D 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 test set T satisfies the All-Def criterion iff for every variable v in the program P, T contains a dc-path from every defining node of v to a use of v
  - For every variable, T contains dc-paths from every defining node to at least one use node
    - **Not all use nodes need to be reached**

$$\forall v \in P(V), nd \in dd\_graph(P) \mid DEF(v, nd)$$

$$\bullet \exists nu \in dd\_graph(P) \mid USE(v, nu) \bullet dc\_path(nd, nu) \in T$$

All-Uses Criterion

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

$$(\forall v \in P(V), nu \in dd\_graph(P) \mid USE(v,nu)$$

$$\bullet \exists nd \in dd\_graph(P) \mid DEF(v,nd) \bullet dc\_path(nd,nu) \in T)$$

$$\wedge$$

$$all\_defs\_criterion$$

All-P-uses / Some-C-uses

- The test 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 of v; if a definition of v has no P-uses, a dc-path leads to at least C-use

$$(\forall v \in P(V), nu \in dd\_graph(P) \mid P\_use(v, nu)$$

$$\bullet \exists nd \in dd\_graph(P) \mid DEF(v, nd) \bullet dc\_path(nd, nu) \in T)$$

$$\wedge$$

$$all\_defs\_criterion$$

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 P-use

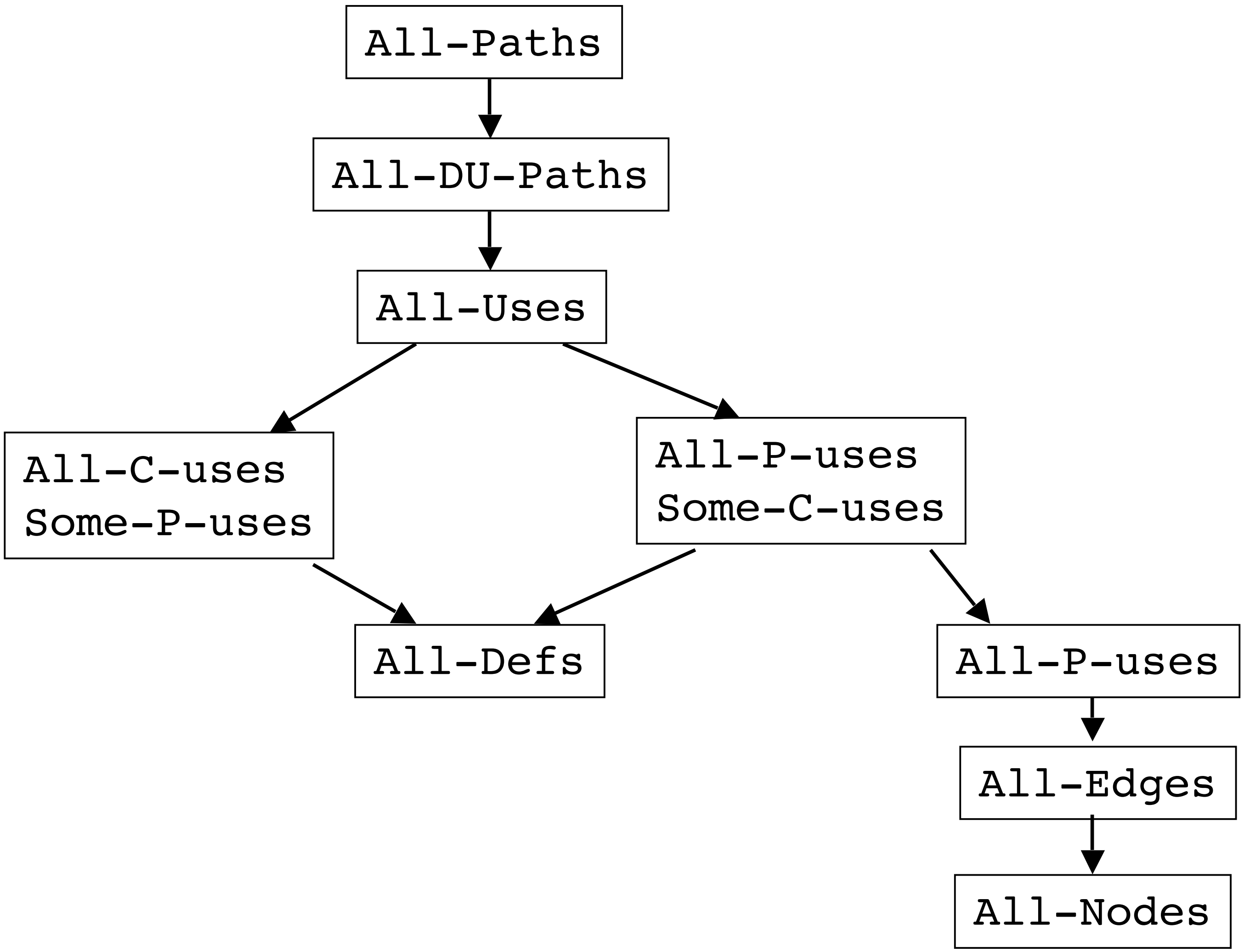
$$(\forall v \in P(V), nu \in dd\_graph(P) \mid C\_use(v,nu)$$

$$\bullet \exists nd \in dd\_graph(P) \mid DEF(v,nd) \bullet dc\_path(nd,nu) \in T)$$

$$\wedge$$

$$all\_defs\_criterion$$

Rapps-Weyuker data flow hierarchy



Data flow guidelines

- Data flow testing is good for computationally 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

Program slice

- Analyze program by focusing on parts of interest, disregarding uninteresting parts.
  - The point of slices is to separate a program into components that have a useful functional meaning
  - Ignore those parts that do not contribute to the functional meaning of interest
  - Cannot do this with du-paths, as slices are not simply sequences of statements or statement fragments
- Informally
  - A program slice is a set of program statements that contributes to or affects a value of a variable at some point in a program

Program slice – 2

- Formally
  - Given a program P and a set of variables V in P, **a slice on the variable V at statement n**, **S(V,n)**, is the set of all statements and statement fragments in P prior to the node n that contribute to the values of variables in V at node n.
    - **Usually statements and fragments correspond to numbered nodes in a program graph, so S(V,n) is a set of node numbers.**
- "Prior to" is a dynamic execution time notion
- Inclusion of node n
  - Include n if a variable in v is defined at n
  - Do not include n if no variable is defined at n; i.e. all variables are used at n

Program slide – meaning of "contributes to"

- Refine use set for a variable
  - P-use – used in a decision predicate
  - C-use – used in a computation
  - O-use – used for output
  - L-use – used for location (pointers, subscripts)
  - I-use – used for iteration (loop counters, loop indices)
  - I-def – defined by input
  - A-def – defined by assignment
- Textbook excludes all non-executable statements such as variable declarations

Program slide – meaning of "contributes to" – 2

- What to include in S(V,n)? Consider a single variable v
  - Include all I-def, A-def
  - Include any C-use, P-use of v, if excluding it would change the value of v
  - Include any P-use or C-use of another variable, if excluding it would change the value of v
  - L-use and I-use
    - **Inclusion is a judgment call, as such use does cause problems**
  - Exclude all non-executable nodes such as variable declarations – if a slice is not to be compliable
  - Exclude O-use, as does not change the value of v

Example 1 – some slices

- This not an exciting program wrt to slices
  - S(max, 9) = { 1, 4, 5, 9 }
  - S(max, 9) = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
  - S(max, 5) = { 1, 4, 5, 6, 8 }
  - S(max, 5) = { 1, 2, 3, 4, 5, 6, 7, 8 }
  - S(j, 7) = { 2, 3, 4, 5 6, 7, 8 }
  - S(j, 5) = {1, 2, 3, 4, 5, 6, 7, 8}

Slice style & technique

- Do not make a slice S(V,n) where the variables of interest are not in node n
  - Leads to slices that are too big
- Make slices on one variable
  - Sometimes slices with more variables are trivial super sets of a one variable case, then a slice on many variables is useful, as we use it and not the one variable slice
- Make slices for all A-def nodes
- Make slices for all P-def nodes – very useful in decision intensive programs

Slice style & technique – 2

- Avoid slices on C-use, they tend to be redundant
- Avoid slices on O-use, they are the union of A-def and I-def slices
- Try to make slices compliable
  - Means including declarations and compiler directives
  - Each such slice becomes executable and more easily tested
- Relative complement of slices can have diagnostic value
  - If you have difficulty at a part, divide the program into two parts
  - If the error does not lie in one part, then it must be in the relative complement

Slice style & technique – 3

- Slices and DD-paths have a many-to-many relationship
  - Nodes in one slice may be in many DD-paths, and nodes in one DD-path may be in many slices
  - Sometimes well-chosen relative complement slices can be identical to DD-paths
- Developing a lattice of slices can improve insight in potential trouble spots
- Slices contain define/reference information
  - When slices are equal, the corresponding paths are definition clear

Slices and programming practice

- Slice testing is an example where consideration of testing can lead to better program development
  - Build and test a program in slices
  - Merge/splice slices into larger programs
  - Use slice composition to re-develop difficult sections of program text