Software Testing

Jeffrey Carver
University of Alabama

April 7, 2016

Warm-up Exercise

- Software testing
- Graphs
- Control-flow testing
- Data-flow testing
- Input space testing
- Test-driven development

Introduction

Goal of Testing

- Cannot ensure code is ‘defect-free’
- Can increase confidence in correctness
- Often neglected because of ‘lack of time’ or ‘lack of funding’ – dangerous situation
- Various ways to approach testing and define test cases

What to Test

- Functional (Black Box)
  - External behavior
  - User-observable behavior
  - Focus: reducing the chances of a target user encountering a functional problem
- Structural (White Box)
  - Internal structure
  - Correct implementation
  - Focus: reducing internal faults so the software is less likely to fail in an unknown situation

When to Stop Testing

- Can use coverage criteria
  - Assumption: higher coverage → fewer remaining defects
  - Functional or Structural
- Reliability Goals
  - Can be more objective
  - Measures what users are likely to encounter
  - Can be tailored for anticipated user groups

Definitions

- Human Error
  - Mistake in the human mental process that leads to a problem in the software or software artifacts
- Software Fault
  - A static defect in the software
- Software Error
  - An incorrect internal state that is the manifestation of some fault
- Software Failure
  - External, incorrect behavior with respect to the requirements or other description of the expected behavior

A Concrete Example

Human Error: Developer misunderstood language syntax

Fault: Should start searching at 0, not 1

```
public static int numZero (int[] arr)
{ //Effects: if arr is null throw NullPointerException
  // else return the number of occurrences of 0 in arr
  int count = 0;
  for (int i = 1; i < arr.length; i++)
   if(arr[i] == 0)
     count++;
   return count;
}
```

Error: i is 1, not 0, on first iteration
Failure: none

Test 1
[2, 7, 0]

Error: i is 1, not 0, on first iteration
Error propagates to variable *count*
Failure: *count* is 0 at the return statement

Test 2
[0, 2, 7]

Testing and Debugging

- Testing: Evaluating software by observing its execution
- Test Failure: Execution of a test that results in a software failure
- Debugging: The process of finding a fault, given a failure

Not all inputs will “trigger” a fault into causing a failure

Conditions Necessary for Failure

- Reachability – The location or locations in the program that contain the fault must be reached

- Infection – The state of the program must be incorrect

- Propagation – The infected state must cause some output or final state of the program to be incorrect

Test Requirements and Criteria

- Test Criterion: A collection of rules and a process that define test requirements
  - Cover every statement
  - Cover every functional requirement
- Test Requirement: Specific things that must be satisfied or covered during testing
  - Each statement is a test requirement
  - Each functional requirement is a test requirement
- All criteria based on four types of structures
  - Graphs
  - Logical Expressions
  - Input Domains
  - Syntax Descriptions

Test Design

- Criteria-Based
  - Design test values to satisfy coverage criteria or other engineering goal
- Human-Based
  - Design test values based on domain knowledge of the program and human knowledge of testing

Changing Notion of Testing

- Old approach
  - Black-box -- White-box
  - Testing each phase differently
- New approach
  - Based on structures and criteria
  - Define a model of the software – find ways to cover it
  - Test design is largely the same at each phase
    - Model is different
    - Choosing the values is different

Covering Graphs

Graphs

- Most commonly used structure for testing
- Many sources
  - Control flow
  - Design structures
  - FSM / statecharts
  - Use Cases
- Tests usually intended to cover the graph in some way

Graphs:
Definition

- A non-empty set *N* of nodes
- A non-empty set of N_o of initial nodes
- A non-empty set N_f of final nodes
- A set *E* of edges from one node to another (ni,nj)
  - i is predecessor
  - j is successor

Graphs: Paths

- Path: A sequence of nodes – [n_1, n_2, …n_x]
- Length: Number of edges
- Subpath: A subsequence of nodes
- Reach(n): Subgraph that can be reached from *n*

Graphs:
Visiting and Touring

- **Visit**: A test path *p* visits node *n* if *n* is in *p*
  A test path *p* visits edge *e* if e is in *p*

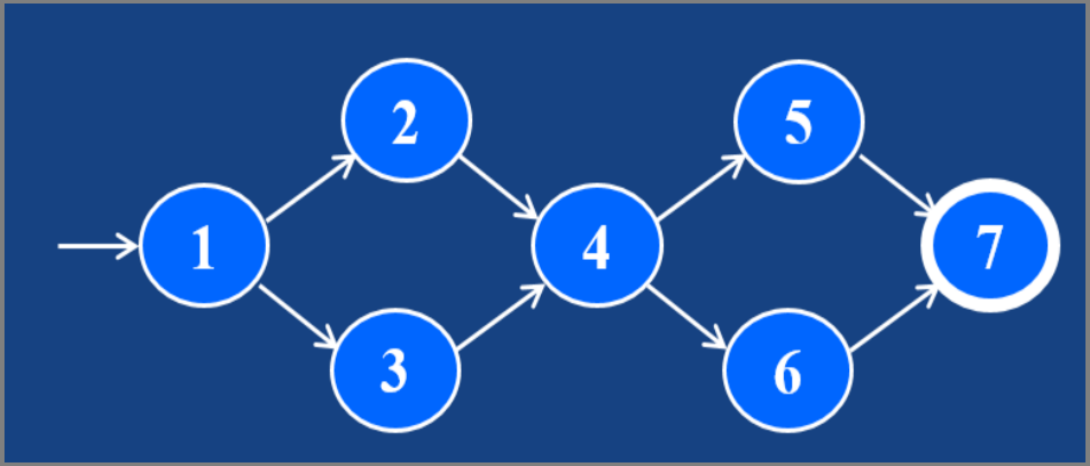
- **Tour**: A test path *p* tours subpath *q* if *q* is a subpath of *p*

Path: 1, 2, 4, 5, 7

Visits nodes?

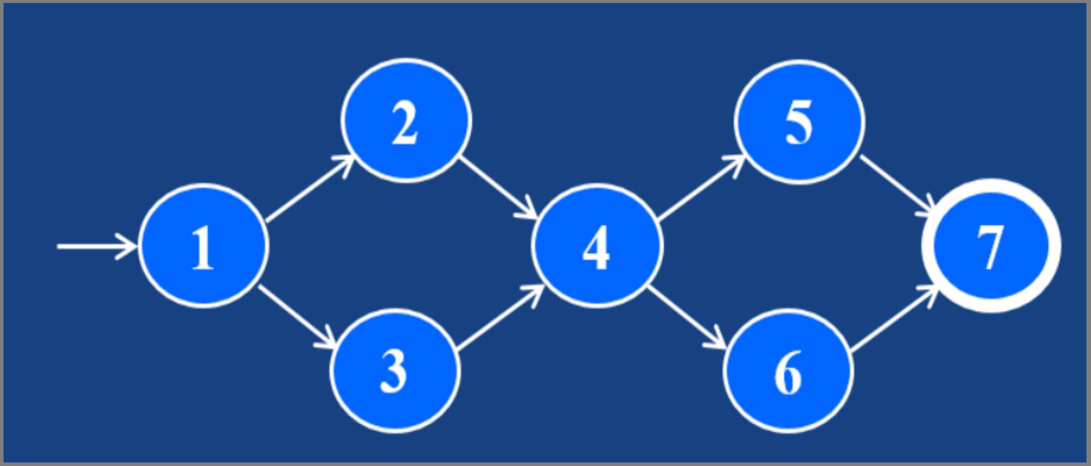
Visits edges?

Tours subpaths?



Graphs: Test Paths

- Starts at an initial node and ends at a final node
- Represents test case execution
  - Some can be executed by many tests
  - Some cannot be executed by any tests
- SESE graphs: All test paths start at single node and end at another node
  - Single-entry, single-exit



Graphs:
Testing and Covering

- Using graphs for testing
  - Develop graph of software
  - Require tests to visit or tour specific nodes, edges, or subpaths

- Test Requirements (TR): Describe properties of test paths, i.e. a set of test requirements (tr)

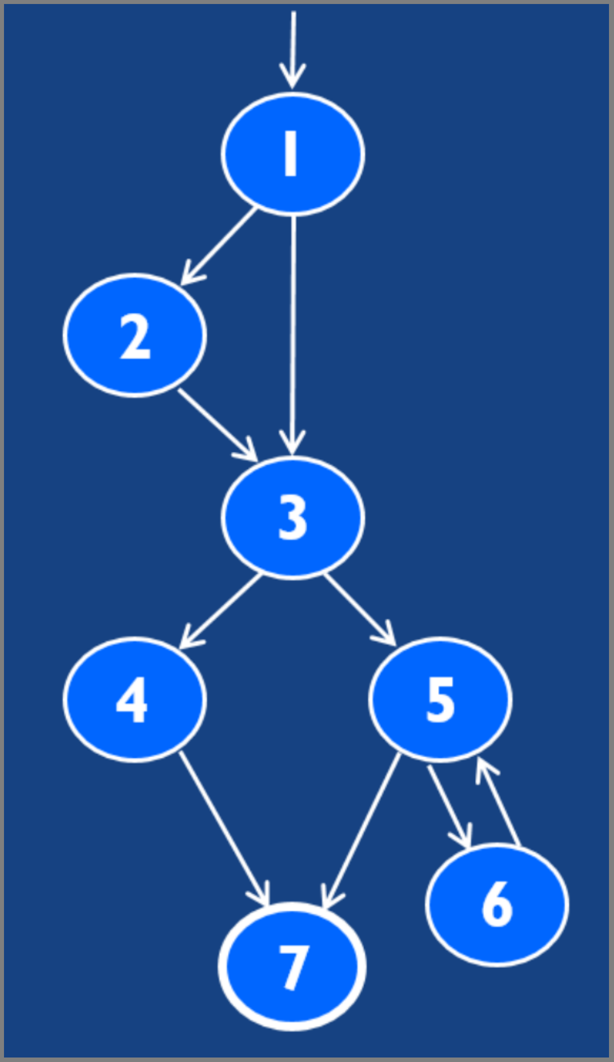
- Test Criterion (C): Rules that define TR

- Satisfaction: Given ***TR*** for ***C***, a set of tests ***T*** satisfies ***C*** iff for every ***tr*** in ***TR***, there is a ***path(T)*** that meets ***tr***

Graphs:
Types of Coverage

- Node Coverage (NC): TR contains each reachable node in G
- Edge Coverage (EC): TR contains each reachable path of length 1 in G
- Edge-Pair Coverage (EPC): TR contains each reachable path of up to 2 in G
- Complete Path Coverage (CPC): TR contains all paths in G
- Specified Path Coverage (SPC): TR contains a set S of test paths, where S is supplied as a parameter

Example



- Node Coverage
- Edge Coverage
- Edge-Pair Coverage
- Complete Path Coverage

Coverage Challenges

- Loops
  - All loops should be executed
  - All loops should be skipped
- Sidetrips
  - Leave a path and return to the same node
- Detours
  - Leave a path and return to the successor node
- Infeasible Test Requirements

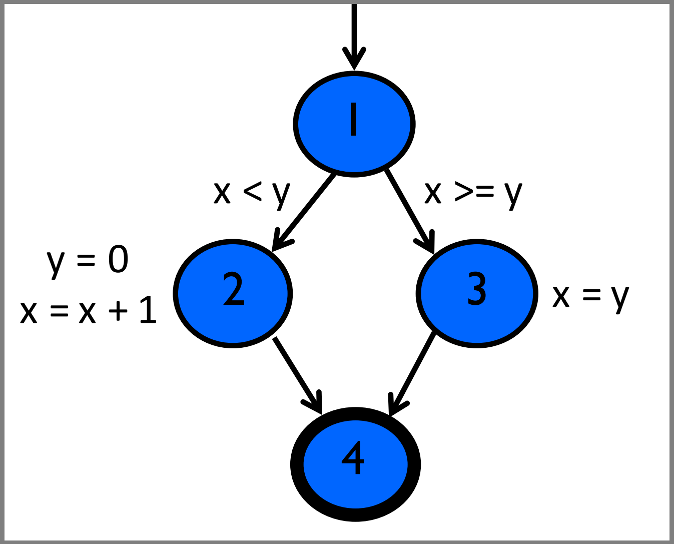
Control-Flow Testing

Control-Flow Graph

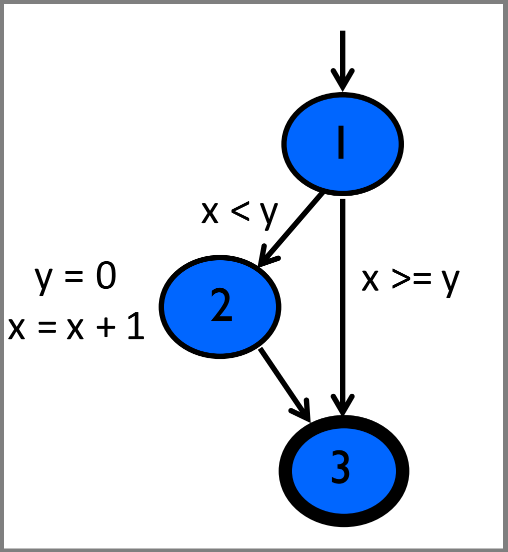
- Nodes: statements or sequences of statements
- Edges: transfers of control
- Basic blocks: sequence of statements without branches
- Control structures: if, while, for, ...

If Statement

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```

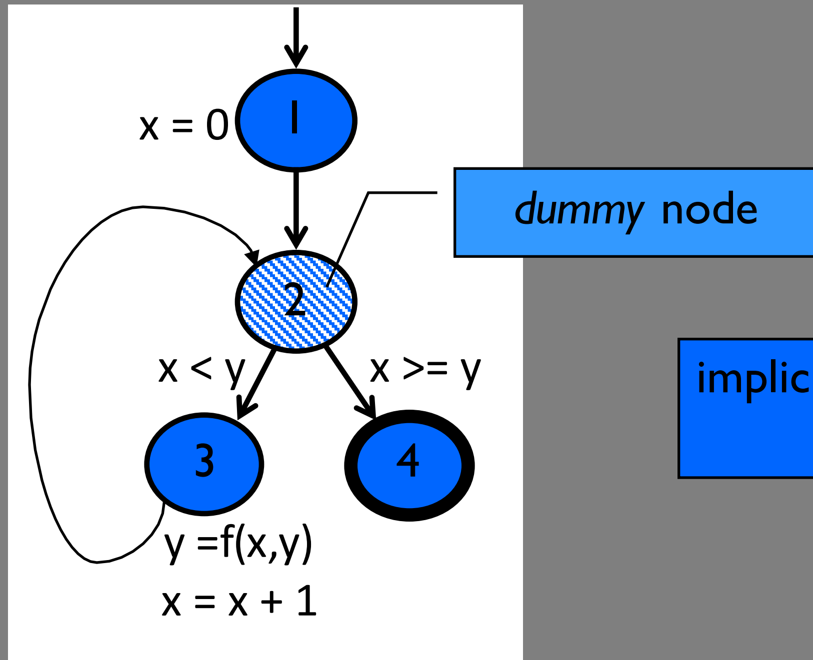



```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```

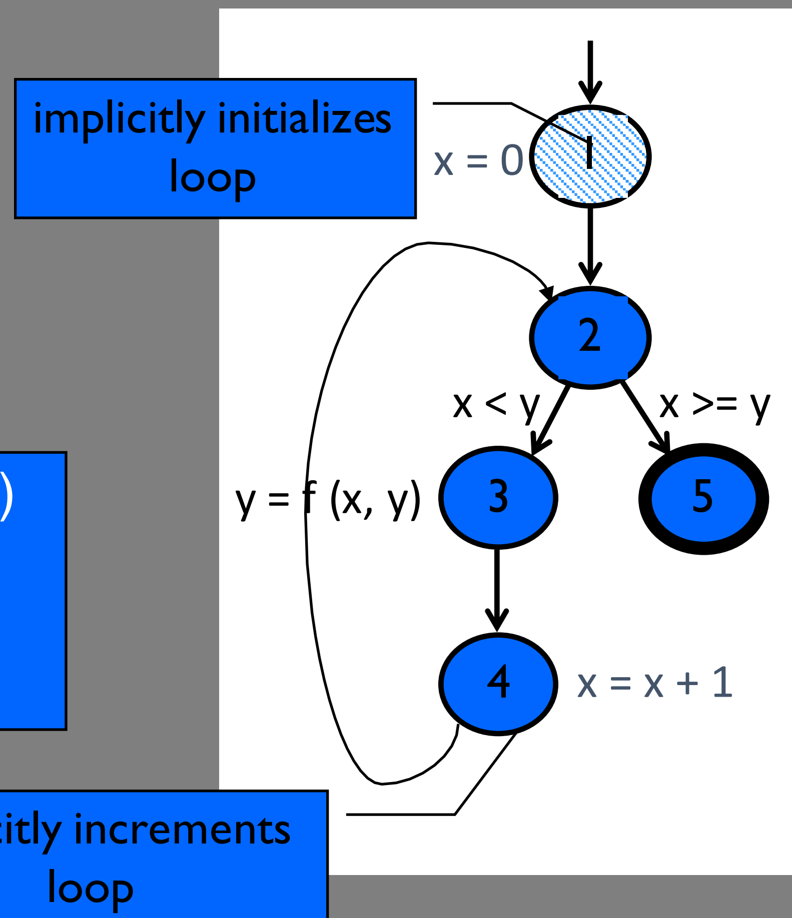


Loops

```
x = 0;
while (x < y)
{
   y = f (x, y);
   x = x + 1;
}
```




```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
```

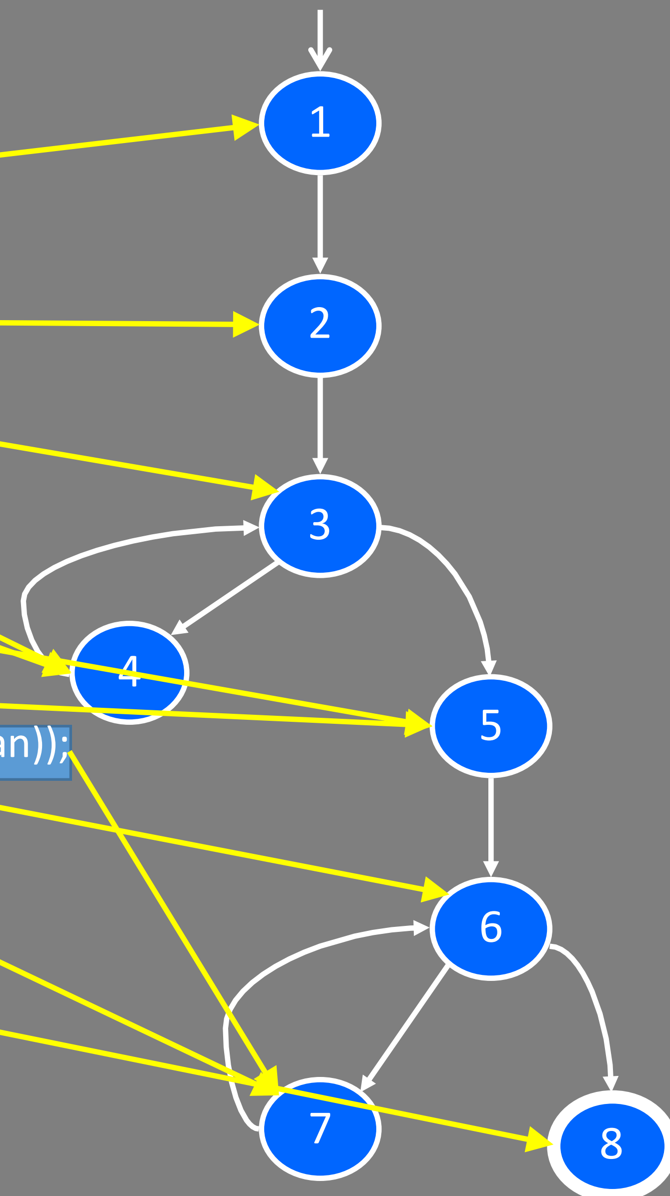


Example

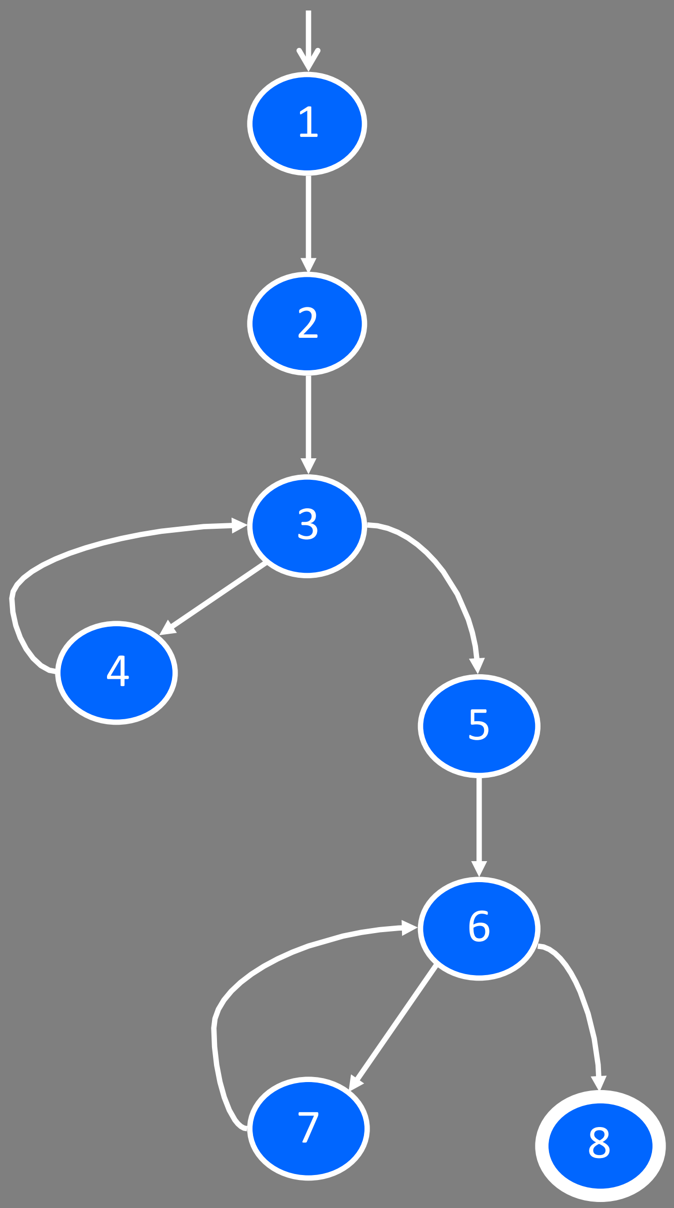
- Using the code on the handout
- Draw a control-flow graph

Example

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med    = numbers [ length / 2];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum + ((numbers[I]- mean) * (numbers[I] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                    " + length);
    System.out.println ("mean:                      " + mean);
    System.out.println ("median:                    " + med);
    System.out.println ("variance:                  " + var);
    System.out.println ("standard deviation: " + sd);
}
```



Coverage



- Node?
- Edge?

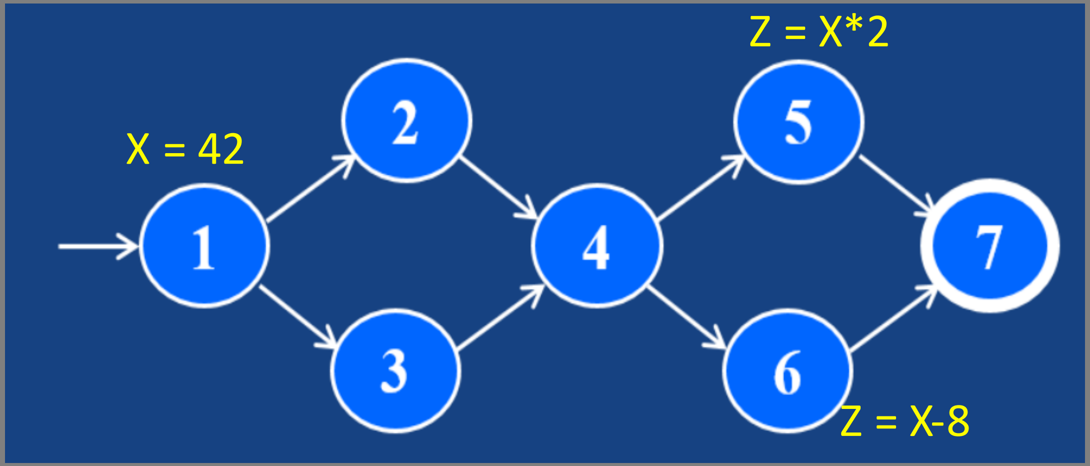
Extensions

- Design elements
  - Nodes are units/methods
  - Edges are calls to units
- Specifications
  - Finite State Machines
  - Models of behavior

Data-Flow Testing

Data Flow Criteria

- **Goal**: Ensure that values are computed and used correctly
- **Definition** (def): value for variable stored in memory
- **Use**: variable's value is accessed



Defs:	def(1) = {X}
	def(5) = {Z}
	def(6) = {Z}
Uses	use(5) = {X}
	use(6) = {X}

The value given in *defs* should reach at least one, some, or all possible *uses*

DU Pairs and DU Paths

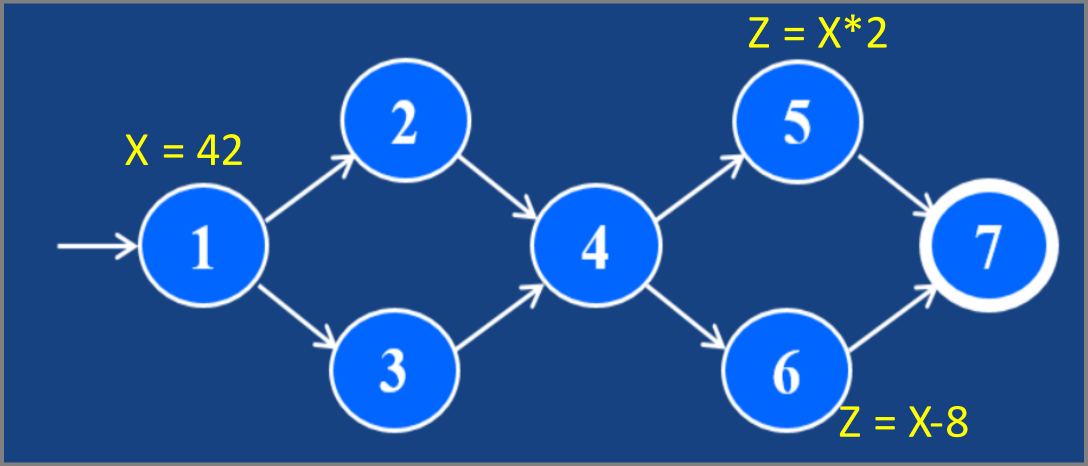
- def(n): set of variables defined by node n
- use(n): set of variables used by node n
- DU Pair: pair of locations (l1, l2) such that variable *v* is defined at *l1* and used at *l2*
- Def-clear: path from *l1* to *l2* is def-clear w/r/t *v*, if *v* does not receive another value on any node or edge on the path
- Reach: if path from *l1* to *l2* is def-clear w/r/t *v*, then the def of *v* at *l1* reaches the use at *l2*
- du-path: simple subpath that is def-clear w/r/t *v* from def *v* to use of *v*
- du(n1,n2,v): set of du-paths from *n1* to *n2*
- du(n1,v): set of du-paths that start at *n1*

Defs and Uses

- Def: location where value is stored
  - LHS of assignment statement
  - Actual parameter in a method call that changes its value
  - Formal parameter of a method (implicit def when method starts)
  - Input to program
- Use: location where value is accessed
  - RHS of assignment statement
  - In a conditional test
  - Actual parameter to a method
  - Output of program
  - Output of a method in a return statement

Touring DU-Paths

- Test path p **du-tours** subpath *d* w/r/t *v* if *p* tours *d* and *d* is def-clear w/r/t v
- Three criteria
  - All-defs coverage: Every def reaches a use
  - All-uses coverage: Every def reache all uses
  - All-du-path-coverage: All paths between def and uses



All-defs (x):	[1,2,4,5]
All-uses (x):	[1,2,4,5], [1,2,4,6]
All du-paths:	[1,2,4,5], [1,2,4,6]
	[1,3,4,5], [1,3,4,6]

Example

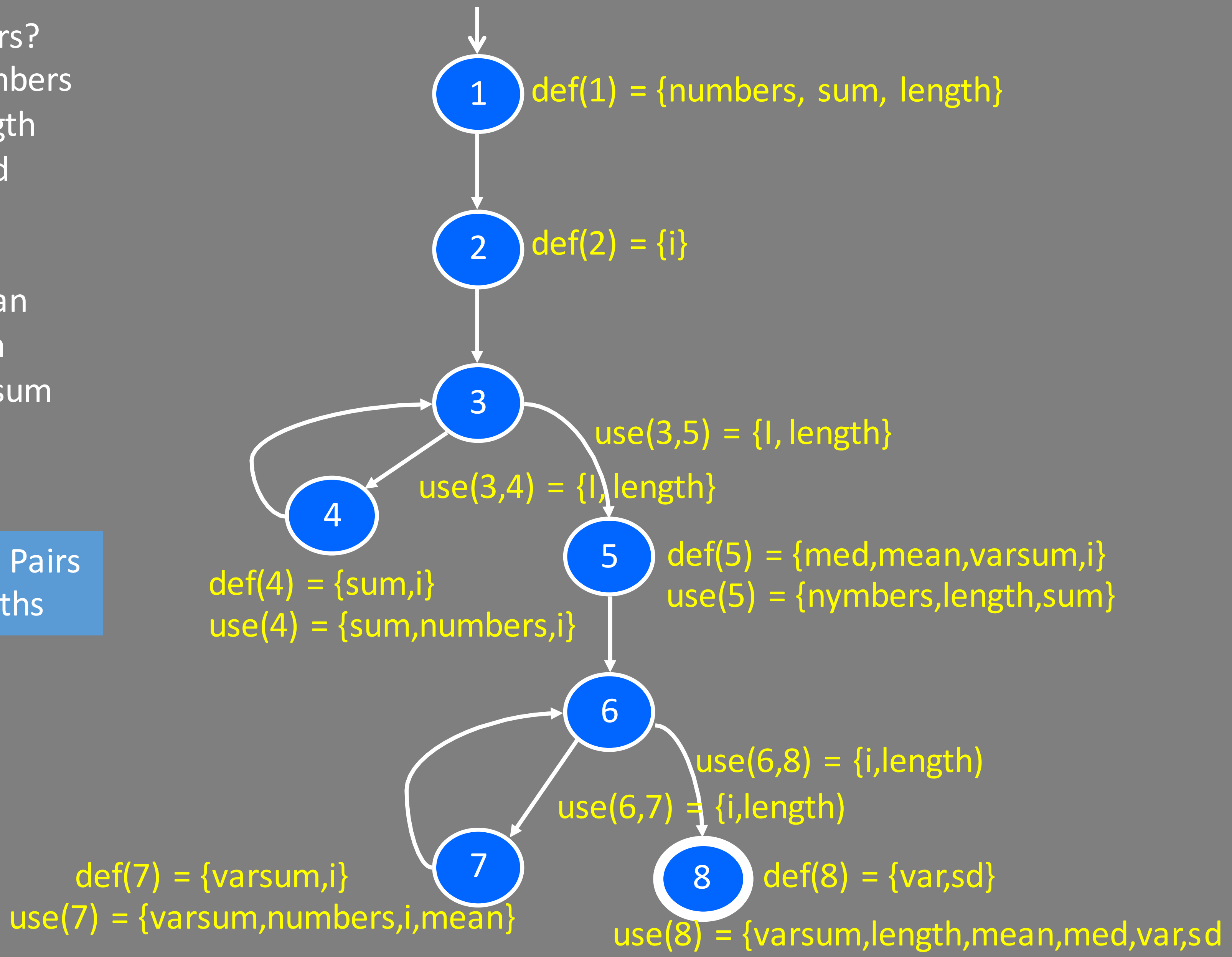
- Use code and graph from before
- Label each node in the graph with the defs and uses

Example

DU Pairs?

- numbers
- length
- med
- var
- sd
- mean
- sum
- varsum
- i

Convert DU Pairs into test paths



Extensions

- Similar to Control Flow
- Design
  - Def-use might be in different methods
  - Interested last-def and first-use pairs

Input Space Testing

Overview

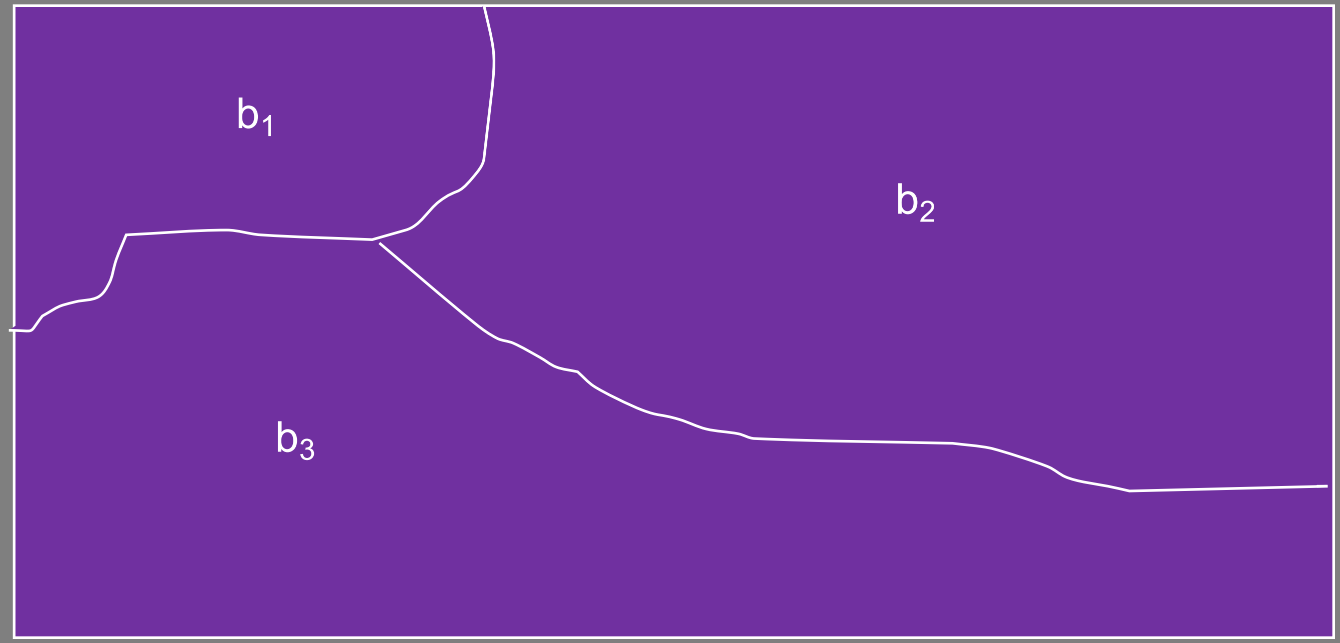
- Input domain – all possible inputs
  - Maybe infinite
  - Testing is about choosing finite set
- Input parameters – define scope of input domain
  - Parameters to a method
  - Data from a file
  - Global variables
  - User inputs
- Input Space Partitioning
  - Partition domain for each input parameter
  - Choose at least one value from each region

Benefits

- Can be applied at several levels
  - Unit
  - Integration
  - System
- Relatively easy to apply with no automation
- Easy to adjust to choose more or fewer test cases
- Requires no implementation knowledge, only knowledge of input space

Partitioning Domains

- Partition – divide domain into blocks
  - Disjointness – blocks must not overlap
  - Completeness – blocks must cover space



Using Partitions

- Each value assumed to be equally useful for testing
- Testing
  - Find a characteristic in data
  - Partition each characteristic
  - Choose tests by combining values from characteristics
- Example characteristics
  - Input X is null
  - Order of input file F (sorted, not sorted, …)
  - Min separation between two aircraft
  - Input devide (DVD, CD, computer, …)

Choosing Partitions

- May seem easy, but easy to get wrong
- Consider characteristic: Order of file F
  - Choose
    - b_1 = sorted in ascending order
    - b_2 = sorted in descending order
    - b_3 = not sorted
  - Is there a problem?
    - What about a file size of 1?
    - Fits all 3 partitions
  - Solution?
    - Each characteristic should address only 1 property
    - Characteristic 1 = File is sorted ascending
      - b_1 = true
      - b_2 = false
    - Characteristic 2 = File is sorted descending
      - b_1 = true
      - b_2 = false

Input Domain Modeling: Steps

- Step 1: Identify testable functions
- Step 2: Find all parameters
- Step 3: Model input domain
- Step 4: Apply test criteria
- Step 5: Choose test inputs

Input Domain Modeling: Approaches

- Interface-based
  - Simpler approach
  - Syntactic view of program
  - Characteristics correspond to individual input parameters in isolation
  - Partially automatable
  - Ignores relationships among parameters
- Functionality-based
  - More difficult – requires design effort
  - Behavioral view of program
  - Based on requirements rather than syntax
  - May result in (fewer) better tests
  - Characteristics correspond to functionality
  - Can incorporate relationship among parameters

Input Domain Modeling: Steps 1 & 2

- Identify testable functions & find all parameters
- Candidates for characteristics
  - Preconditions/postconditions
  - Variable relationships
    - Each other
    - Special values (e.g. 0, null, …)
- Does not use program source
- Better to have more characteristics with fewer blocks

Input Domain Modeling: Steps 1 & 2

```
public boolean findElement(List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

<u>Interface-Based Approach</u>

Two parameters: list, element

Characteristics:
- list is null (b_1 = T, b_2 = F)
- list is empty (b_1 = T, b_2 = F)

<u>Functionality-Based Approach</u>

Two parameters: list, element

Characteristics:
- number of occurrences of element in list
  (0, 1, >1)
- element occurs first in the list
  (T, F)
- element occurs last in the list
  (T, F)

Input Domain Modeling: Step 3 – Model Input Domain

- Partitions flow directly from Steps 1 & 2
- Creative design activity to decide balance between #characteristics and #blocks
- Strategies for identifying values
  - Valid/invalid/special values
  - Sub-partition some blocks
  - Domain boundaries
  - “normal use”
  - Try to balance the number of blocks/characteristic
  - Check for completeness/disjointness

Input Domain Modeling:
Step 3 – Model Input Domain

Using Trityp code on handout – Method Triag

<u>Interface-based</u>

1 Testable function, 3 integer inputs

Max of 3*3*3 = 27 tests
Some triangles are invalid
Refine

Characteristic	b_1	b_2	b_3
q_1 = "Relation of Side 1 to 0"	Greater than 0	Equal to 0	Less than 0
q_2 = "Relation of Side 2 to 0"	Greater than 0	Equal to 0	Less than 0
q_3 = "Relation of Side 3 to 0"	Greater than 0	Equal to 0	Less than 0

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Refinement of q_1"	Greater than 1	Equal to 1	Equal to 0	Less than 0
q_2 = "Refinement of q_2"	Greater than 1	Equal to 1	Equal to 0	Less than 0
q_3 = "Refinement of q_3"	Greater than 1	Equal to 1	Equal to 0	Less than 0

Input Domain Modeling:
Step 3 – Model Input Domain

<u>Functionality-Based</u>

Behavior is about identifying valid triangles

Characteristic	b_1	b_2	b_3	b_4
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

Another approach:
Break geometric characterization into 4 separate characteristics

Characteristic	b_1	b_2
q_1 = “Scalene”	True	False
q_2 = “Isosceles”	True	False
q_3 = “Equilateral”	True	False
q_4 = “Valid”	True	False

Input Domain Modeling: Steps 4 & 5 – Choosing Combinations of Values

- Criteria
  - All Combinations – all combinations, all blocks
  - Each choice – one value from each block
  - Pair-wise – each block/each characteristic with every block/every other characteristic
  - t-wise – each block for each group of t characteristics
  - Base choice – choose a ‘base’ block for each characteristic; combine all bases; hold all but one base constant
- Most obvious is All Combinations
- Some combinations are not possible

References

- Much of the material in the slides has been adapted from:

Introduction to Software Testing, Amman and Offutt

Software Testing

Jeffrey Carver
University of Alabama
carver@cs.ua.edu

April 7, 2016