Testing

Sr. Nisha C D
Assistant Professor, Dept. of Computer Science
Little Flower College, Guruvayoor

Outline

- Fundamentals of Software Testing
  - Testing techniques
    - White-box testing
      - Control-flow-based testing
      - Data-flow-based testing
    - Black-box testing
      - Equivalence partitioning

Testing Objective

- **Testing:** a process of executing software with the intent of finding errors
- **Good testing:** a high probability of finding as-yet-undiscovered errors
- **Successful testing:** discovers unknown errors

Basic Definitions

- **Test case:** specifies
  - Inputs + pre-test state of the software
  - Expected results (outputs an state)
- **Black-box testing:** ignores the internal logic of the software, and looks at what happens at the interface (e.g., given this inputs, was the produced output correct?)
- **White-box testing:** uses knowledge of the internal structure of the software
  - E.g., write tests to "cover" internal paths

Testing Approaches

- Will look at a sample of approaches for testing
- White-box testing
  - Control-flow-based testing
  - Data-flow-based testing
- Black-box testing
  - Equivalence partitioning

Control-flow-based Testing

- A traditional form of white-box testing
- Step 1: From the source, extract a CFG
- Step 2: Design test cases to cover certain elements of this graph
  - Nodes, edges, paths

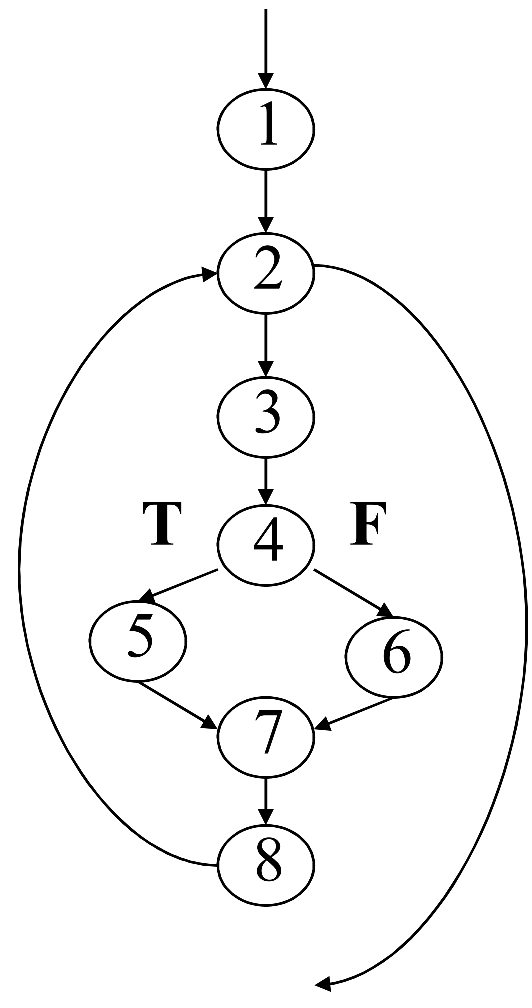
- Basic idea: given the CFG, define a coverage target and write test cases to achieve it

Statement Coverage

- Traditional target: statement coverage
  - Need to write test cases that cover all nodes in the control flow graph
- Intuition: code that has never been executed during testing may contain errors
  - Often this is the "low-probability" code

Example

- Suppose that we write and execute two test cases
- Test case #1: follows path 1-2-exit (e.g., we never take the loop)
- Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit (loop twice, and both times take the true branch)
- Problems?



Branch Coverage

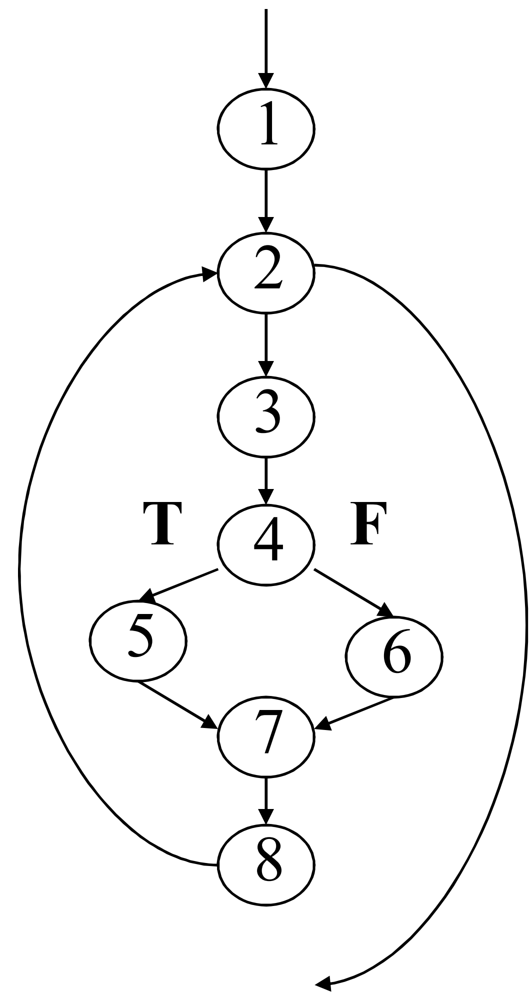
- Target: write test cases that cover all branches of predicate nodes
  - True and false branches of each IF
  - The two branches corresponding to the condition of a loop
  - All alternatives in a SWITCH statement
- In modern languages, branch coverage <u>implies</u> statement coverage

Branch Coverage

- Statement coverage does not imply branch coverage
- Can you think of an example?
- Motivation for branch coverage: experience shows that many errors occur in "decision making" (i.e., branching)
  - Plus, it subsumes statement coverage.

Example

- Same example as before
- Test case #1: follows path 1-2-exit
- Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit
- Problem?



Achieving Branch Coverage

- For decades, branch coverage has been considered a necessary testing minimum
- To achieve it: pick a set of start-to-end paths in the CFG, that cover all branches
  - Consider the current set of chosen paths
  - Try to add a new path that covers at least one edge that is not covered by the current paths
- Then write test cases to execute these paths

Some Observations

- It may be impossible to execute some of the chosen paths from start-to-end
  - Why? Can you think of an example?
  - Thus, branches should be executed as part of other chosen paths
- There are many possible sets of paths that achieve branch coverage

Example

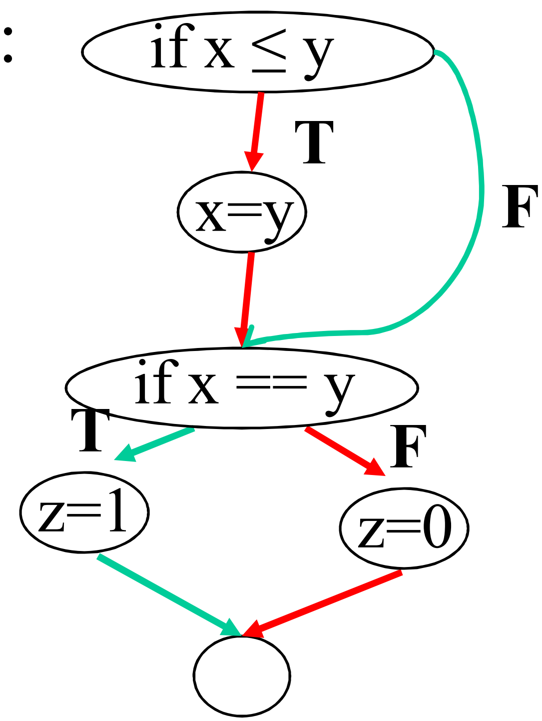
Candidate start-to-end paths:

(1) green path

(2) red path

% branch coverage?

Problem?



Data-flow-based Testing

- Basic idea: test the connections between variable definitions ("write") and variable uses ("read")
- Starting point: variation of the control flow graph
  - Statement nodes represent **one** statement
- Set Def(n) contains variables that are defined at node *n* (i.e., they are written)
  - The definitions at node *n*
- Set Use(n): variables that are read
  - The uses at node *n*

Example

Assume **y** is an input variable

```
1 s:= 0;
2 x:= 0;
3 while (x<y) {
4     x:=x+3;
5     y:=y+2;
6     if (x+y<10)
7         s:=s+x+y;
      else
8         s:=s+x-y;
```

Def(1) := {s}, Use(1) := $\emptyset$

Def(2) := {x}, Use(2) := $\emptyset$

Def(3) := $\emptyset$, Use(3) := {x,y}

Def(4) := {x}, Use(4) := {x}

Def(5) := {y}, Use(5) := {y}

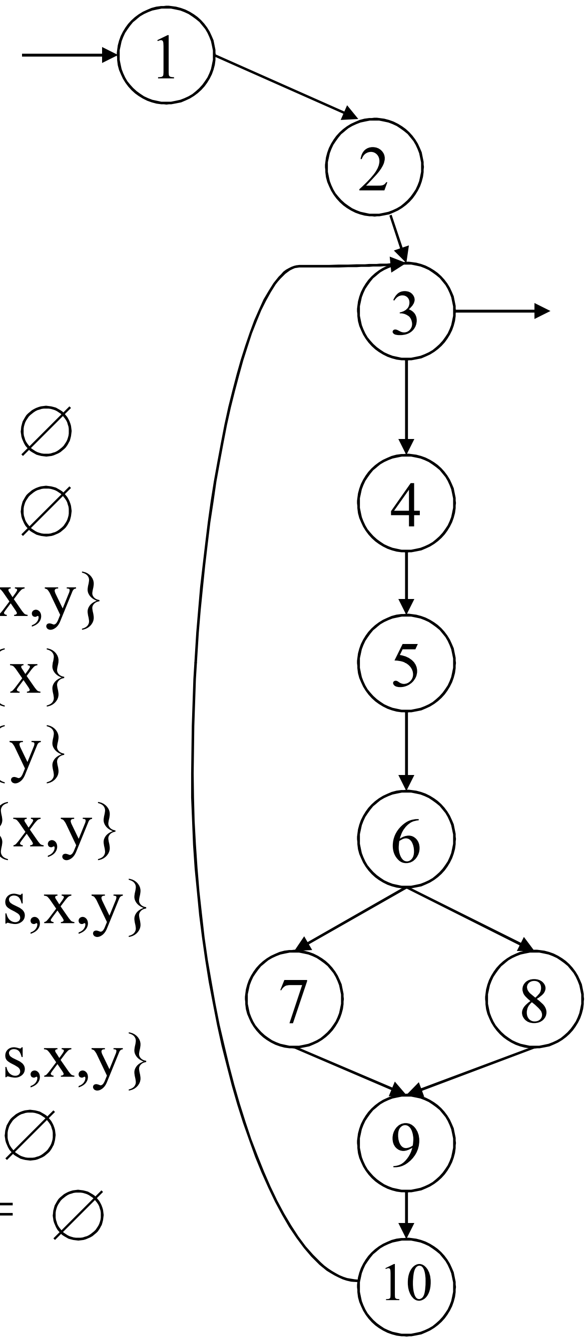
Def(6) := $\emptyset$, Use(6) := {x,y}

Def(7) := {s}, Use(7) := {s,x,y}

Def(8) := {s}, Use(8) := {s,x,y}

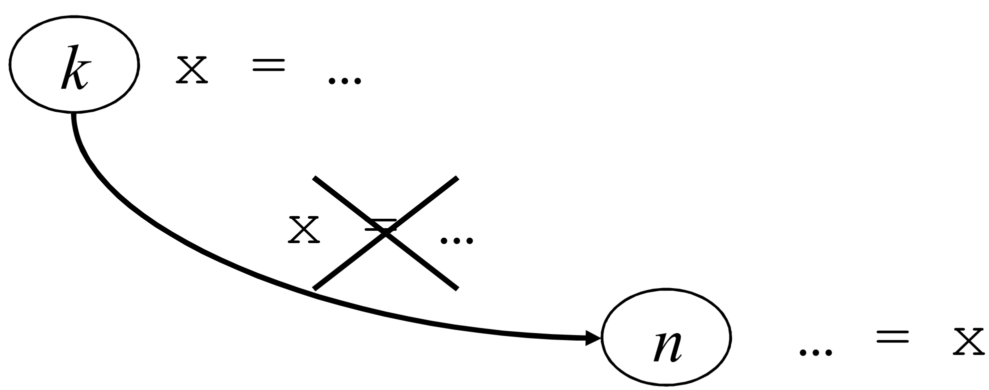
Def(9) := $\emptyset$, Use(9) := $\emptyset$

Def(10) := $\emptyset$, Use(10) := $\emptyset$



Remember Reaching Definitions

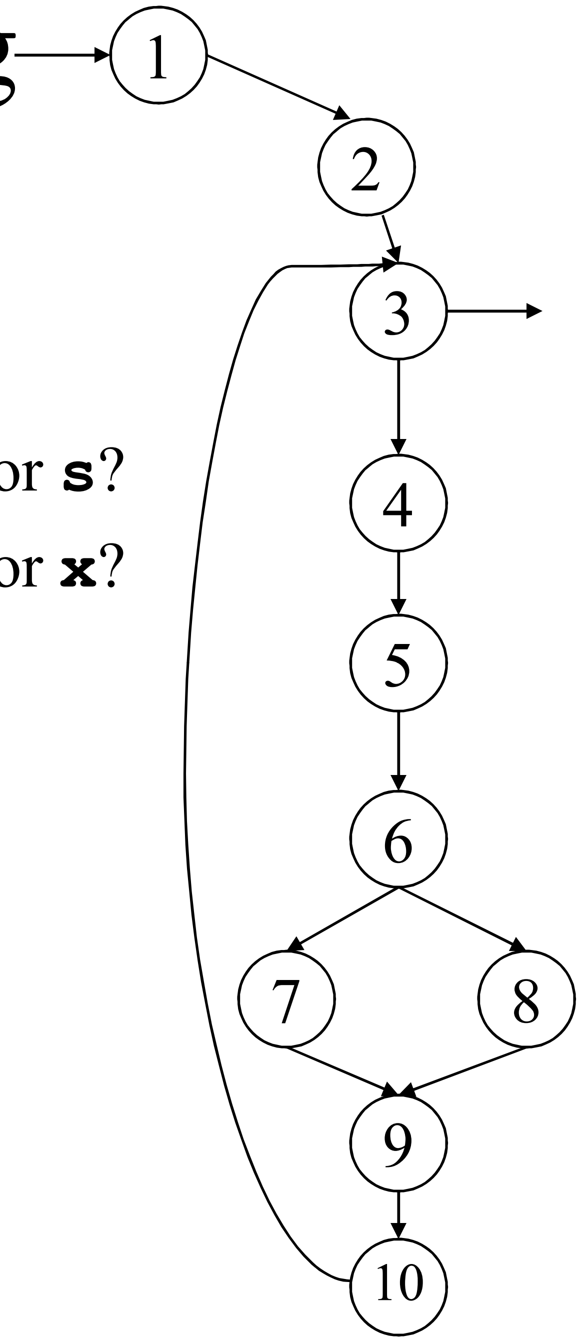
- ***Definition*** A statement that may change the value of a variable (e.g., **x = i+5**)
- A definition of a variable **x** at node *k* ***reaches*** node *n* if there is a path from *k* to *n*, clear of a definition of **x**.



Def-use Pairs

- A def-use pair (DU pair) for variable **`x`** is a pair of nodes (n1,n2) such that
  - **`x`** is in Def(n1)
  - The definition of **`x`** at n1 *reaches* n2
  - **`x`** is in Use(n2)
- In other words, the value that is assigned to **`x`** at n1 is used at n2
  - Since the definition *reaches* n2, the value is not "killed" along some path n1...n2.

Examples of Reaching Definitions



Assume **y** is an input variable

```
1 s:= 0;
2 x:= 0;
3 while (x<y) {
4     x:=x+3;
5     y:=y+2;
6     if (x+y<10)
7        s:=s+x+y;
      else
8        s:=s+x-y;
```

What are the def-use pairs for `s`?

What are the def-use pairs for `x`?

So, how do we compute def-use pairs?

Data-flow-based Testing

- Data-flow-based coverage target: DU pair coverage
  - Compute all DU pairs, and construct test cases that cover these pairs. HOW DO WE COMPUTE DU PAIRS?
- Several coverage targets (criteria), with different relative strength

- **Motivation for data-flow-based testing coverage:** see the effects of using the values produced by computations
  - Focuses on the data, while control-flow-based testing focuses on the control

Finally, the targets (criteria): All-defs criterion

- If variable **`x`** is in Def(n1), the **all-defs** criterion requires the test data to exercise at least one path free of definition of **`x`** which goes from *n1* to some node *n2* such that (n1,n2) is a DU pair for **`x`**.
  - Remember, **`x`** is defined at *n1*,
  - The definition of **`x`** at *n1* reaches *n2*, and
  - **`x`** is used at *n2*

All-uses criterion

- If variable **`x`** is in Def(n1), the **all-uses** criterion requires the test data to exercise <u>at least one path free of definition of **`x`**</u> which goes from *n1* to <u>each node</u> *n2* such that (n1,n2) is a DU pair for **`x`**.

All-DU-paths criterion

- If variable **x** is in Def(n1), the **all-DU-paths** criterion requires the test data to exercise <u>each path free of definition of **x**</u> which goes from *n1* to <u>each node</u> *n2* such that (n1,n2) is a DU pair for **x**.

- So what is the relative strength of the three criteria: All-defs, All-uses, All-DU-paths?

All-defs, all-uses, all-du-paths

Assume **y** is input

```
1   s:= 0;
2   x:= 0;
3 while (x<y) {
4       x:=x+3;
5       y:=y+2;
6       if (x+y<10)
7         s:=s+x+y;
        else
8         s:=s+x-y;
}
```

1. Design test cases that cover all-uses

Black-box Testing

- Unlike white-box testing, no knowledge about the internals of the code
- Test cases are designed based on specifications
  - Example: search for a value in an array
    - Postcondition: return value is the index of some occurrence of the value, or -1 if the value does not occur in the array
    - We design test cases based on this spec

Equivalence Partitioning

- Basic idea: consider input/output domains and partition them into equiv. classes
  - For different values from the same class, the software should behave equivalently
- Use test values from each class
  - Example: if the range for input **x** is 2..5, there are three classes: "<2", "between 2..5", "5<"
  - Testing with values from different classes is more likely to uncover errors than testing with values from the same class

Equivalence Classes

- Examples of equivalence classes
  - Input x in a certain range [a..b]: this defines three classes “x<a”, “a<=x<=b”, “b<x”
  - Input x is boolean: classes “true” and “false”
  - Some classes may represent invalid input
- Choosing test values
  - Choose a **typical** value in the middle of the class(es) that represent valid input
  - Also choose values at the **boundaries** of all classes: e.g., if the range is [a..b], use a-1,a, a+1, b-1,b,b+1

Example

- Suppose our spec says that the code accepts between 4 and 24 inputs, and each one is a 3-digit positive integer
- One dimension: partition the number of inputs
  - Classes are “x<4”, “4<=x<=24”, “24<x”
  - Chosen values: 3,4,5, 14, 23,24,25
- Another dimension: partition the integer values
  - Classes are “x<100”, “100<=x<=999”, “999<x”
  - Chosen values: 99,100,101, 500, 998,999,1000

Another Example

- Similar approach can be used for the output: exercise boundary values
- Suppose that the spec says "the output is between 3 and 6 integers, each one in the range 1000-2500
- Try to design input that produces
  - 3 outputs with value 1000
  - 3 outputs with value 2500
  - 6 outputs with value 1000
  - 6 outputs with value 2500

Example: Searching

- Search for a value in an array
  - Return value is the index of some occurrence of the value, or -1 if the value does not occur in the array
- One partition: size of the array
  - Since people often make errors for arrays of size 1, we decide to create a separate equivalence class
  - Classes are "empty arrays", array with one element", "array with many elements"

Example: Searching

- Another partition: location of the value
  - Four classes: "first element", "last element", "middle element", "not found"

Array	Value	Output
Empty	5	-1
[7]	7	0
[7]	2	-1
[1,6,4,7,2]	1	0
[1,6,4,7,2]	4	2
[1,6,4,7,2]	2	4
[1,6,4,7,2]	3	-1

Testing Strategies

- We talked about testing techniques (white-box, black-box)
- Many unanswered questions
  - E.g., who does the testing? Which techniques should we use and when? And more…
- There are no universal strategies, just principles that have been useful in practice
  - E.g., the notions of **unit testing** and **integration testing**

Some Basic Principles

- Testing starts at the component level and works "outwards"
  - Unit testing, integration testing, system testing
- Different testing techniques are appropriate at different scopes
- Testing is conducted by developers and/or by a specialized group of testers
- Testing is different from debugging
  - Debugging follows successful testing

Scope and Focus

- Unit testing: scope = individual component
  - Focus: component correctness
  - Black-box and white-box techniques
- Integration testing: scope = set of interacting components
  - Focus: correctness of component interactions
  - Mostly black-box, some white-box techniques
- System testing: scope = entire system
  - Focus: overall system correctness
  - Only black-box techniques

Test-First Programming

- Modern practices emphasize the importance of testing during development
- Example: test-first programming
  - Basic idea: before you start writing any code, first write the tests for this code
  - Write a little test code, write the corresponding unit code, make sure it passes the tests, and then repeat
  - What programming methodology uses this approach?
  - What are the advantages of test-first programming?

Advantages of Test-First Programming

- Developers do not “skip” unit testing
- Satisfying for the programmer: feeling of accomplishment when the tests pass
- Helps clarify interface and behavior before programming
  - To write tests for something, first you need to understand it well!
- Software evolution
  - After changing existing code, rerun the tests to gain confidence (regression testing)