Test Generation

Lecture 9: CPEN 400P

Karthik Pattabiraman, UBC

(Slides are based on Gary Tan’s CSE597: Topics in Software Testing at Penn State)

Outline

Goals and principles

Black-box testing

White-box testing

Automated test generation

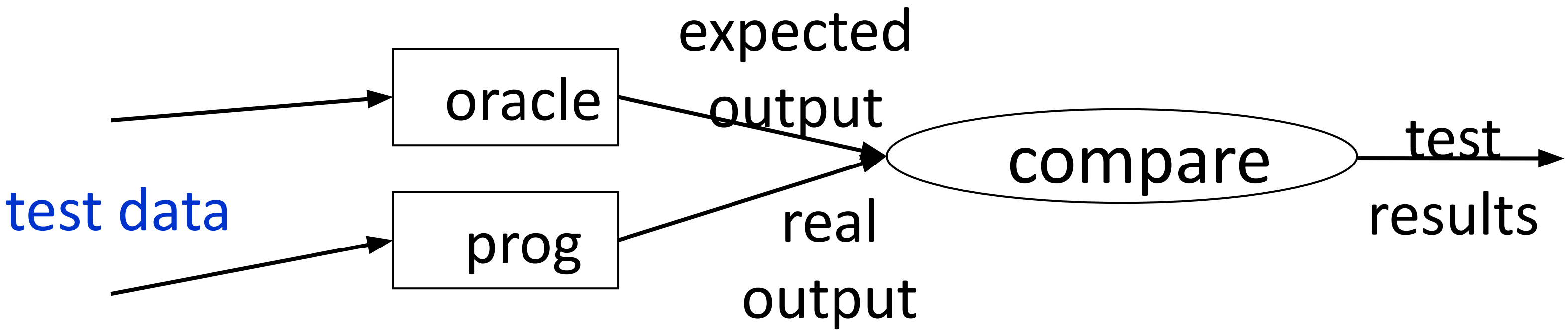
Dynamic Analysis

- Analyze the program when it is running with a specific input
- Many techniques
  - Testing (this class)
  - Dynamic invariants (next class)
  - Fuzzing (class after the midterm exam)

Program Testing

- Testing: the process of running a program on a set of test cases and comparing the actual results with expected results
  - For the implementation of a factorial function, test cases could be {0, 1, 5, 10}
- Testing cannot guarantee program correctness
  - What’s the simplest program that can fool the test cases above?
  - However, testing can catch many bugs

Testing Process



Selecting Test Data

- Testing is w.r.t. a finite test set
    - Exhaustive testing is usually not possible
    - E.g, a function takes 3 integer inputs, each ranging over 1 to 1000
        - Suppose each test takes 1 second
        - Exhaustive testing would take ~31 years
- Question: How do you design the test set?
    - Black-box testing
    - White-box testing (or, glass-box)

Outline

Goals and principles

Black-box testing

White-box testing

Automated test generation

Black-Box Testing

- Generating test cases based on specification alone
    - Without considering the implementation (internals)
- Advantage
    - Test cases are not biased toward an implementation
        - E.g., boundary conditions

Generating Black-Box Test Cases

- Example

```
static float sqrt (float x, float epsilon)
// Requires: x >= 0 && .00001 < epsilon < .001
// Effects: Returns sq such that x-epsilon <= sq*sq <= x+ epsilon
```

- The precondition can be satisfied
  - Either “x=0 and .00001 < epsilon < .001”,
  - Or “x>0 and .00001 < epsilon < .001”
- Any test data should cover these two cases
- Also test the case when x is negative and epsilon is outside the expected range

More Examples

- Test cases: cover both true and false cases; also test numbers 0, 1, 2, and 3

  static int search (int[ ] a, int x)

  // Effects: If a is null throws NullPointerException else if x is in a, returns i such that a[i]=x, else throws NotFoundException

- Test cases?

More Examples

- Test cases: cover both true and false cases; also test numbers 0, 1, 2, and 3

  static int search (int[ ] a, int x)

  // Effects: If a is null throws NullPointerException else if x is in a, returns i such that a[i]=x, else throws NotFoundException

- Test cases?
  - a=null
  - A case where a[i]=x for some i
  - A case where x is not in the array a

Boundary Conditions

- Common programming mistakes: not handling boundary cases
  - Input is zero
  - Input is negative
  - Input is null
  - …
- Test data should include these boundary cases

Class Activity: Generate blackbox tests

static void appendVector (Vector v1, Vector v2)

// Effects: If v1 or v2 is null throws NullPointerException else removes all elements of v2 and appends them in reverse order to the end of v1

- Test cases?

Class Activity: Solution

static void appendVector (Vector v1, Vector v2)

// Effects: If v1 or v2 is null throws NullPointerException else removes all elements of v2 and appends them in reverse order to the end of v1

- Test cases?
  - v1=null;
  - v2=null
  - v1 is the empty vector
  - v2 is the empty vector
  - Both are empty vectors
  - Another one: v1 and v2 refer to the same vector
    - Aliases

Outline

Goals and principles

Black-box testing

White-box testing

Automated test generation

White-Box Testing

- Looking into the internals of the program to figure out a set of test cases

```
static int maxOfThree (int x, int y, int z) {
// Effects: Return the maximum value of x, y and z
    if (x>y)
            if (x>z) return x; else return z;
    else
            if (y>z) return y; else return z;
}
```

  - The implementation is divided into four cases, so we need to cover them all
    - $x>y$ and $x>z$
    - $x>y$ and $x<=z$
    - $x<=y$, and $y>z$
    - $x<=y$, and $y<=z$

Test Coverage

- Idea: code that has not been covered by tests are likely to contain bugs
  - Divide a program into a set of elements
    - The definition of elements leads to different kinds of test coverage
  - Define the coverage of a test suite to be:

$$\frac{\text{\# of elements executed by the test suite}}{\text{\# of elements in total}}$$

Test Coverage

- Test quality is determined by the coverage of the program by the test set
- Benefits
  - Can be used as a stopping rule: stop testing if 100% of elements have been tested
  - Comparison: a test set that has a test coverage of 80% is better than one that covers 70%
  - Test case generation: look for a test which exercises some statements not covered by the tests so far

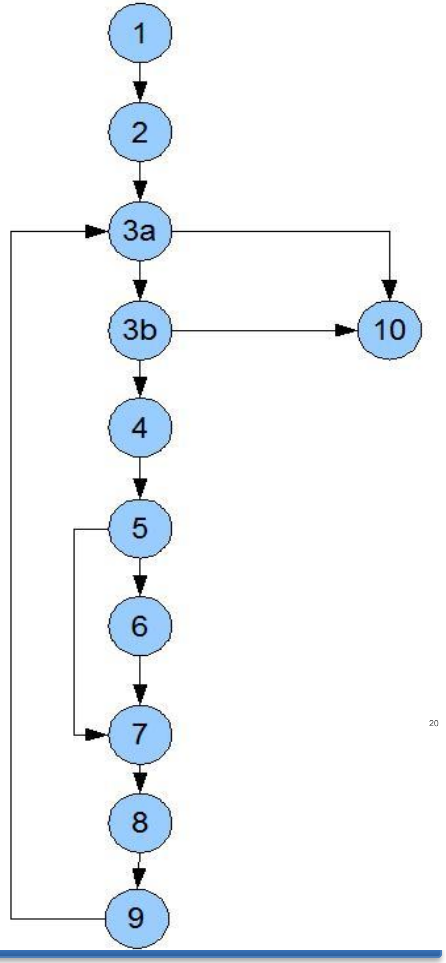
Different Coverage Criteria

- Usually based on control flow graphs (CFG)
    - Can have automated tool support
- Different types of coverage
    - Statement coverage
    - Edge coverage
        - Edges in CFGs
    - Path coverage

A Running Example

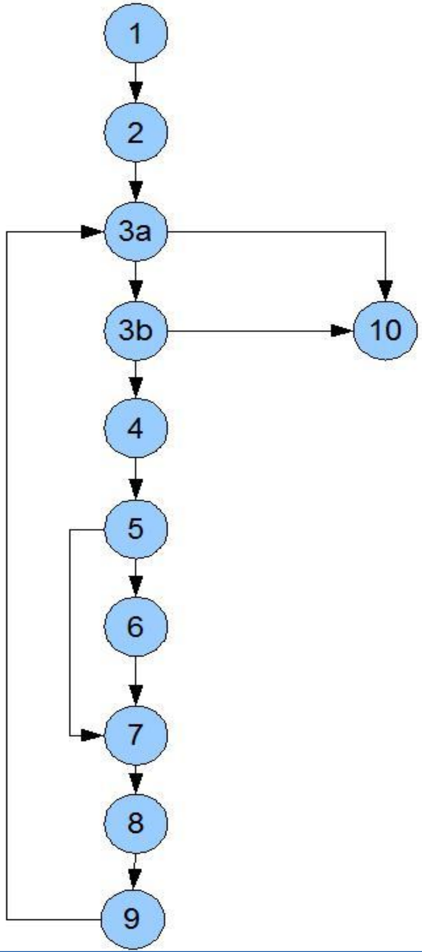
```
// Input: table is an array of numbers;
// Input: n is the size of table
// Input: element is the element to be found
// Output: found indicates whether the element
//         is in the table

1: found = false;
2: counter = 0;
3: while ((counter < n) && (!found))
4: {
5:   if (table[counter] == element)
6:     found = true;
7:
8:   counter++;
9: }
10:
```



Statement Coverage

```
1:  found = false;
2:  counter = 0;
3:  while ((counter < n) && (!found))
4:  {
5:    if (table[counter] == element)
6:      found = true;
7:
8:    counter++;
9:  }
10:
```



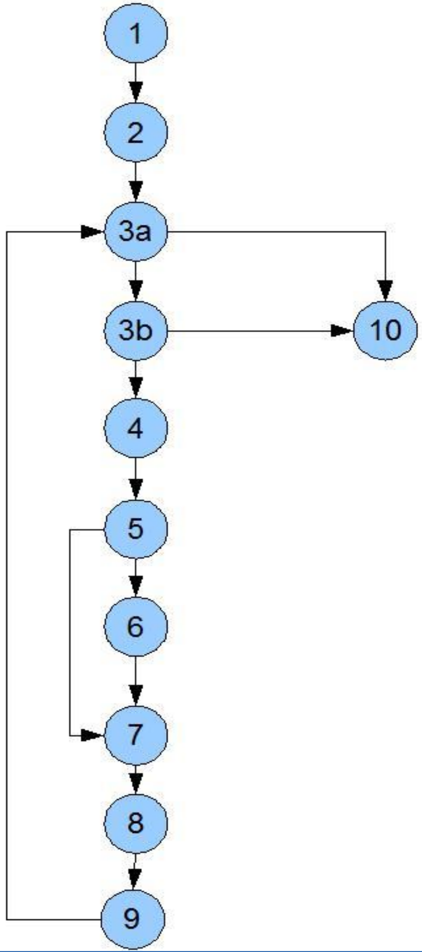
- Test data: table={3,4,5}; n=3; element=3
  - Does it cover all statements?
    - Yes
  - But does it cover all edges?
  - No, missing the edge from 3a to 10 and 5 to 7

Statement Coverage in Practice

- 100% is hard
    - Usually about 85% coverage
- Microsoft reports 80-90% statement coverage
- Safety-critical application usually requires 100% statement coverage
    - Boeing requires 100% statement coverage
    - Other metrics: Modified Condition Decision Coverage (MCDC) for safety-critical applns.

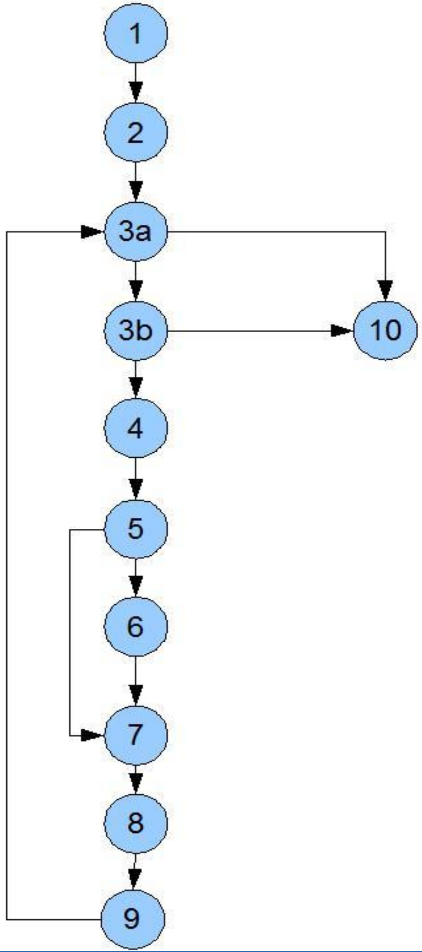
Edge Coverage

```
1: found = false;
2: counter = 0;
3: while ((counter < n) && (!found))
4: {
5:   if (table[counter] == element)
6:     found = true;
7:
8:   counter++;
9: }
```



Edge Coverage

```
1: found = false;
2: counter = 0;
3: while ((counter < n) && (!found))
4: {
5:   if (table[counter] == element)
6:     found = true;
7:
8:   counter++;
9: }
```



- Test data to cover all edges
  - table={3,4,5}; n=3; element=3
  - table={3,4,5}; n=3; element=4
  - table={3,4,5}; n=3; element=6

Path Coverage

- Path-complete test data
  - Covering every possible control flow path
- For example:

```
static int maxOfThree (int x, int y, int z) {
    if (x>y)
        if (x>z) return x; else return z;
    else
        if (y>z) return y; else return z;
}
```

  - Test data is complete as long as the following four case are covered
    - x>y and x>z
    - x>y and x<=z
    - x<=y, and y>z
    - x<=y, and y<=z

Covering All Paths

- NOTE: A program having path-complete test data doesn't mean it's correct

```
static int maxOfThree (int x, int y, int z) {
    return x;
}
```

  - "x=5, y=4, z=3" would pass the test and be path complete

- Same goes for the case of all-statement coverage, or all-edge coverage

Possibly Infinite Paths

- If there is a loop in the program, then there are possibly infinite # of paths
    - In general, impossible to cover all of them
- One Heuristic
    - Include test data that cover zero, one, and two iterations of a loop
    - Why two iterations?
        - A common programming mistake is failing to reinitialize data in the second iteration
    - This offers no guarantee, but can catch many errors

Path Coverage

```
found = false;
counter = 0;
while ((counter < n) && (!found))
{
  if (table[counter] == element)
    found = true;

  counter++;
}
```

Path Coverage

```
found = false;
counter = 0;
while ((counter < n) && (!found))
{
  if (table[counter] == element)
    found = true;

  counter++;
}
```

- Zero iteration: table={ }; n=0; element=3
- One iteration: table={3,4,5}; n=3; element=3
- Two iterations: table={3,4,5}; n=2; element=4

Combining Them All

- A good set of test data combines various testing strategies
  - Black-box testing
    - Generating test cases by specifications
    - Boundary conditions
  - White-box testing
    - Test coverage (e.g., being edge complete)

Class Activity

Generate black box and white box test cases (path coverage) for the following

```
// Effects: If s is null throws NullPointerException,   else returns true iff s is a palindrome
boolean palindrome (String s)  throws NullPointerException {
  int low=0;
  int high = s.length() -1;
  while (high>low) {
    if (s.charAt(low) != s.charAt(high))
      return false;
    low++;
    high--;
  }
  return true;
}
```

Class Activity: Solution

- Based on spec.
  - s=null
  - s="deed" (palindrome)
  - s="abc" (not a palindrome)
  - s=""   (boundary condition)
  - s="a" (boundary condition)
- Based on the program
  - Null pointer exception
  - Not executing the loop at all
  - Returning false in the first iteration
  - Returning true after the first iteration
  - Returning false in the second iteration
  - Returning true after the second iteration

Outline

Goals and principles

Black-box testing

White-box testing

Automated test generation

Automated Test Generation

- Designing tests with good coverage is hard; not as clean as the examples
  - Manually designing a good test set is a major task
  - 100% coverage almost never achieved in practice
- Q: can we automate it?
- Yes, for certain situations
  - **Pre-condition based test generation**
  - Fuzzing can be viewed as a way of automated test generation
  - Symbolic execution can also be used for test generation

Pre- and Post-Conditions

- A pre-condition is a predicate
  - Assumed to hold before a function executes
- A post-condition is a predicate
  - Expected to hold after a function executes, whenever the pre-condition holds
- Example

```
static float sqrt (float x, float epsilon)
// pre: x >= 0 && .00001 < epsilon < .001
// post: Returns sq such that            x-epsilon
   <= sq*sq <= x+ epsilon
```

Pre- and Post-Conditions

- Most useful if they are executable
  - Written in the programming language itself
  - A special case of assertions
- Need not be complete
  - Useful even if they do not cover every situation

Test Generation Using Preconditions

- A simple algorithm

```
while (true) {
    test = randomlyGenerate();
    if (precondition(test)) runTest(test);
}
```

Won’t work for large test spaces, and very dependent on the random number generator - need cleverer approach

Cleverer way to generate tests

Main insight: Sufficient to consider inputs up to a small size

Hypothesis: If there's any test that causes the program to fail, there's a small such test (i.e., the large test can be reduced to the small test)

- If a list function works for small lists, it probably works for lists of any length
- Not always true if there're integer overflows, memory limits etc.

Pioneered by the Korat tool - ISSTA'02

Main Idea

- Use precondition to filter the generated inputs:

User Input: User selects maximum input size k

```
tests = generateAllInputsUpto(k);
forall test in tests {
  if (precondition(test)) runTest(test);
}
```

Strengths and Weaknesses

- Strong when we can enumerate all possibilities
  - e.g. Four nodes, two edges per node

  => Good for:
    - Linked data structures
    - Small, easily specified procedures
    - Unit testing

- Weaker when enumeration is weak
  - Integers, Floating-point numbers, Strings

Weaknesses

Only as good as the pre- and post-conditions

```
Pre: is_member(x, list)
List remove(Element x, List list) {
   if (x == head(list))
      return tail(list);
   else
      return cons(head(list),
                  remove(x, tail(list)));
}
Post: !is_member(x, list')
```

Weaknesses

Only as good as the pre- and post-conditions

```
Pre: !is_empty(list)
List remove(Element x, List list) {
   if (x == head(list))
      return tail(list);
   else
      return cons(head(list),
                  remove(x, tail(list)));
}
Post: is_list(list’)
```

Class Activity

Write the weakest possible pre-condition that prevents any in-built exceptions from being thrown in the following code

```
Pre:
```

```
int foo(int[] A, int[] B) {
    int r = 0;
    for (int i = 0; i < A.length; i++) {
        r += A[i] * B[i];
    }
    return r;
}
```

Class Activity: Solution

Write the weakest possible pre-condition that prevents any in-built exceptions from being thrown in the following code

Pre:

```
A != null && B != null && A.length <= B.length
```

```
int foo(int[] A, int[] B) {
    int r = 0;
    for (int i = 0; i < A.length; i++) {
        r += A[i] * B[i];
    }
    return r;
}
```

Outline

Goals and principles

Black-box testing

White-box testing

Automated test generation