Program Testing and Analysis: Manual Testing

Prof. Dr. Michael Pradel
Software Lab, TU Darmstadt

Partly based on slides from Peter Müller, ETH Zurich
Warm-up Quiz

What does the following code print?

```javascript
var a, b;
var x = {};
x[a] = 23;
console.log(x[b]);
```

Nothing  23  undefined  false
What does the following code print?

```javascript
var a, b;
var x = {};
x[a] = 23;
console.log(x[b]);
```

Options:
- Nothing
- 23
- undefined
- false
Warm-up Quiz

What does the following code print?

```javascript
var a, b;
var x = {};
x[a] = 23;
console.log(x[b]);
```

Nothing 23 undefined false
Outline (Manual Testing)

- Overview

- Control flow testing
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Loop coverage

- Data flow testing
  - DU-pair coverage

- Interpretation of coverage
Path Coverage

Idea:
Test all possible paths through the CFG

\[
COV_{path} = \frac{Nb. \ of \ executed \ paths}{Total \ nb. \ of \ paths}
\]

- A path is a sequence \( n_1, \ldots, n_k \) where
  - \( n_1 = \text{entry node} \)
  - \( n_k = \text{exit node} \)
  - There is an edge \( n_i \rightarrow n_{i+1} \) for all \( 1 \leq i < k \)
Path Coverage: Example

```javascript
function fourPaths(a, b) {
    var x = 1;
    var y = 1;
    if (a)
        x = 0;
    else
        y = 0;
    if (b)
        return 5 / x;
    else
        return 5 / y;
}
```
Path Coverage: Discussion

Is path coverage the solution?

```javascript
function contains(a, x) {
    if (!a) return false;
    var found = false;
    for (var i = 0; i <= a.length; i++) {
        if (a[i] === x) {
            found = true;
            break;
        }
    }
    return found;
}
```
Path Coverage: Discussion (2)

- Leads to more thorough testing than both statement and branch coverage
  - Complete path coverage implies complete statement coverage and branch coverage
  - But: “at least n% path coverage” implies neither “at least n% statement coverage” nor “at least n% branch coverage”

- Complete path coverage is not feasible for loops: Unbounded number of paths
Loop Coverage

Idea: For each loop, test 0, 1, or more than 1 consecutive iterations

Let \( n_k \) be the number of loops with exactly \( k \) consecutively executed iterations

\[
COV_{\text{loop}} = \frac{n_0 + n_1 + n_{>1}}{\text{Total nb. of loops} \times 3}
\]

- Typically combined with other criteria, such as statement or branch coverage
function reverse(a)

```
var res = [];

var i = 0

i = a.length
true false
```

res[j] = a[i]
return res;

```
entry

var j = a.length - 1;
```

Test:
• a = [1] ✓

Quiz: Loop coverage ?
→ 1/3 possible cases covered
→ 33% loop coverage

More tests:
• a = [] ✓ → 0 iterations
• a = [1, 2] ✓ → >1 iterations

→ 100% loop coverage
→ 3rd test case finds the bug
Measuring Coverage

- Coverage information is collected while the test suite executes
- Typically implemented via code instrumentation or debug interface
  - Count executed basic blocks, branches taken, etc.
Measuring Coverage: Example

```javascript
function fourPaths(a, b) {
    var x = 1;
    var y = 1;
    if (a) {
        x = 0;
    } else {
        y = 0;
    }
    if (b) {
        return 5 / x;
    } else {
        return 5 / y;
    }
}
```
Measuring Coverage: Example

```javascript
function fourPaths(a, b) {
    var x = 1;
    var y = 1;
    if (a) {
        executedBranches[0]++; x = 0;
    } else {
        executedBranches[1]++; y = 0;
    }
    if (b) {
        executedBranches[2]++; return 5 / x;
    } else {
        executedBranches[3]++; return 5 / y;
    }
}
```
Coverage measurement described with small-step operational semantics

- **extend configuration**:
  \[ \langle P, s \rangle \rightarrow \langle P, s, b \rangle \]

  branch coverage map: \( \text{Loc} \rightarrow \{ T, F \} \)
  
  is initialized to \( F \) for all locations

- **extend transition rules**:
  
  assume: each command is annotated with its code location; e.g. \( x = 5 \) \( \text{line 23} \)

  \[ \langle \text{if True then } c_1 \text{ else } c_2, s, b \rangle \rightarrow \langle c_0, s, b \{ c_1 \rightarrow T \} \rangle \]
Outline

■ Overview

■ Control flow testing
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Loop coverage

■ Data flow testing
  - DU-pair coverage

■ Interpretation of coverage
Data Flow Testing

- Problem: Testing **all paths is not feasible**
  - Number grows *exponentially* in the number of branches
  - Loops

- Idea: Test those paths where a **computation** in one part of the path affects the **computation** of another part
Variable Definition and Use

■ A **variable definition** for a variable v is a basic block that assigns to v
  - v can be a local or global variable, parameter, or property

■ A **variable use** for a variable v is a basic block that reads the value of v
  - In conditions, computations, output, etc.
Definition-Clear Paths

A **definition-clear path** for a variable $v$ is a path $n_1, \ldots, n_k$ in the CFG such that

- $n_1$ is a variable definition for $v$
- $n_k$ is a variable use for $v$
- No $n_i \ (1 < i \leq k)$ is a variable definition for $v$
  - $n_k$ may be a variable definition if each assignment to $v$ occurs after a use

**Note:** Def-clear paths do **not** go from entry to exit (in contrast to our earlier definition of paths)
Definition-Use Pair

A definition-use pair (DU-pair) for a variable $v$ is a pair of nodes $(d, u)$ such that there is a definition-clear path $d, \ldots, u$ in the CFG.
DU-Pairs Coverage

Idea:
Test all paths that provide a value for a variable use

\[
cov_{DU} = \frac{\text{Nb. of executed DU-pairs}}{\text{Total nb. of DU-pairs}}
\]
function fourPaths(a, b)

entry

var x = 1;

var y = 1;

a

ture false

x = 0 y = 0

b
true false

5 / x; return 5 / y; return 5 / y;

exit

Two tests:
- a = true, b = false
- a = false, b = true

DU-pairs for x: (1, 7), (4, 7)

DU-pairs for y: (2, 8), (5, 8)

→ Cover 214 DU-pairs

= 50% DU-pairs coverage
DU-Pair Coverage: Discussion

- **Complements control flow testing**
  - Use both: Choose tests that maximize branch and DU-pair coverage

- **As with path coverage, not all DU-pairs are feasible**
  - Static analysis overapproximates data flow

- **Complete DU-pair coverage does not imply that all bugs are detected**
Outline

- Overview

- Control flow testing
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Loop coverage

- Data flow testing
  - DU-pair coverage

- Interpretation of coverage
Interpreting Coverage

- High coverage does not imply that code is well tested
- But: Low coverage means that code is not well tested
- Do not blindly increase coverage but develop test suites that are effective at detecting bugs
Empirical Evidence

- **Studies on the benefit of coverage metrics**
  

- **Approach**
  
  - Seed bugs into code
  - Develop **test suites** that satisfy various coverage criteria
  - Measure how many of the seeded bugs are found
Empirical Evidence (2)

- The higher the coverage, the more bugs are detected
- Tests written with coverage criteria in mind are more effective than random tests (for the same test suite size)
- Test suite size grows exponentially in the achieved coverage
Summary: Manual Testing

Black box testing
- Exhaustive testing
- Random testing
- Functional testing

White box testing
- Structural testing
  - Control flow-based coverage criteria: Statements, branches, paths, loops
  - Data flow-based coverage criterion: DU-pairs