Program Testing and Analysis:
Manual Testing (Part 2)

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
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
Warm-up Quiz

What does the following code print?

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

- Have value `undefined`
- Write and then read "undefined" property of x
- 23
- undefined
- false

Nothing
Outline (Manual Testing)

- Overview

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

- Data flow testing
  - DU-pair coverage

- Interpretation of coverage
Branch Coverage

Idea: Test all possible branches in the CFG

$$COV_{branch} = \frac{nb. \ of \ executed \ branches}{total \ nb. \ of \ branches}$$

- An edge $a \xrightarrow{b} c$ in the CFG is a branch iff there is another edge $a \xrightarrow{b'} c'$ with $c \neq c'$
- If code contains no branches, coverage defined as 100%
Branch Coverage: Discussion

Is full branch coverage enough?

```javascript
function reverse(a) {
    var j = a.length - 1;
    var res = [];
    for (var i = 0; i < a.length; i++) {
        res[j] = a[i];
    }
    return res;
}
```
Branch Coverage: Discussion

Is full branch coverage enough?

```javascript
function reverse(a) {
    var j = a.length - 1;
    var res = [];
    for (var i = 0; i < a.length; i++) {
        res[j] = a[i];
    }
    return res;
}
```

Must decrement `j`
function reverse(a)

entry

var j = a.length - 1;

var res = [];

var i = 0

i < a.length

true

res[j] = a[i]

false

return res;

i++

exit

Can achieve 100% branch coverage with a single test:

a = [1]

But: Miss the bug!

→ Need more thorough testing
Another 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;
}
```
Another 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;
}
```

Division by zero
function fourPaths(a, b)

entry

var x = 1;

var y = 1;

a

b

true  false

x = 0  y = 0

return 5 / x;  return 5 / y;

exit

Two test cases:
- a = true,  b = false ✓
- a = false,  b = true ✓

→ 100% branch coverage

No bug found!
Path Coverage

Idea:
Test all possible paths through the CFG

\[ \text{COV}_{\text{path}} = \frac{\text{Nb. of executed paths}}{\text{Total nb. of paths}} \]

- 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 \( 1 \leq i < k \)
function fourPaths(a, b)

Tests:
- a = true, b = false
- a = false, b = true

Quiz: Path coverage?
- 2/4 paths covered

Add more tests:
- a = true, b = true
- a = false, b = false
- 4/4 paths covered
- Bug found (twice even)
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, and more than 1 consecutive iterations

Let $n_k$ be the number of loops with exactly $k$ consecutively executed iterations

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

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

var j = a.length - 1;

var res = [];

var i = 0

i < a.length

true

false

res[i] = a[i]

return res;

Test:
- a = [1] → 1 iteration

Quiz: Loop coverage?
- 1/3 possible cases, 33% loop coverage

More tests:
- a = [] → 0 iterations
- a = [1,2] → 1 iteration

→ 100% loop coverage

3rd test finds 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.
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: \( \text{Loc} \rightarrow \{ T, F \} \)

. extend transition rules:
  assume that each command is annotated with its code location, e.g. \( k = 5 \) Line 23

\[ \langle \text{if True then } C_1 \text{ else } C_2 \cdot s, b \rangle \rightarrow \langle C_1 \cdot s, b [ C_2 \rightarrow T] \rangle \]  (if+)
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 control flow 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.
function fourPaths(a, b)

1. var x = 1;

2. var y = 1;

3. a

4. x = 0
   y = 0

5. true
   false

6. b

7. return 5 / x;
   return 5 / y;

8. exit

for variable x
definition

Definition-use pairs for x

(1, 7)
(4, 7)
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

ture false

return 5 / x;  return 5 / y;

exit

Two Tests
- $a = \text{true}, \ b = \text{false}$
- $a = \text{false}, \ b = \text{true}$

DU-pairs for $a$: (1,7), (4,7)

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

→ Covers 2/4 DU-pairs

= 50% 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