Program Testing and Analysis:
Manual Testing

Dr. Michael Pradel
Software Lab, TU Darmstadt

Partly based on slides from Peter Müller, ETH Zurich
What does the following code print?

```javascript
function Person(name) { this.name = name; }
Person.prototype = {
    isJoe: function() {
        return this.name === "Joe";
    }
};
var p1 = new Person("Peter");
var p2 = p1;
p1.name = "Joe";
console.log(p2.isJoe());
```

false true Something else
What does the following code print?

```javascript
function Person(name) { this.name = name; }
Person.prototype = {
  isJoe: function() {
    return this.name === "Joe";
  }
};
var p1 = new Person("Peter");
var p2 = p1;
p1.name = "Joe";
console.log(p2.isJoe());
```

false  true  Something else
Warm-up Quiz

What does the following code print?

```javascript
function Person(name) { this.name = name; }
Person.prototype = {
    isJoe: function() {
        return this.name === "Joe";
    }
};
var p1 = new Person("Peter");
var p2 = p1;
p1.name = "Joe";
console.log(p2.isJoe());
```

false  true  Something else
What does the following code print?

```javascript
function Person(name) { this.name = name; }
Person.prototype = {
  isJoe: function() {
    return this.name === "Joe";
  }
};
var p1 = new Person("Peter");
var p2 = p1;
p1.name = "Joe";
console.log(p2.isJoe());
```

Two references to same object

false  true  Something else
Outline: Manual Testing

- Overview

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

- Data flow testing
  - DU-pair coverage

- Interpretation of coverage
Goal of Testing

- **Error**: Deviation of observed behavior from required/desired behavior
  - Functional requirements (what the system is supposed to accomplish)
  - Non-functional requirements (performance, usability, etc.)

- **Testing**: Process of **executing a program** to find errors

- **Successful test**: Test that finds errors
Limitations of Testing

Testing can only show the presence of bugs, never their absence.

E. W. Dijkstra

Impossible to completely test any non-trivial module or system

- Theoretical limitations: Termination (halting problem)
- Practical limitations: Prohibitive in time and cost
$P \ni P(i) \ldots \text{ behavior}$

- Bug found by testing
- Bug missed by testing
Testing Approaches

Exhaustive testing
- Goal: Cover all input
- Not feasible for real programs

Functional testing
- Goal: Cover all requirements
- Black-box testing
- Suitable for all stages

Random testing
- Goal: Automation
- Black-box testing
- Suitable for all stages

Structural testing
- Goal: Cover all code
- White-box testing
- Suitable for unit testing
```javascript
function roots(a, b, c /* doubles */) {
  var q = b * b - 4 * a * c;
  if (q > 0 && a != 0) {
    numRoots = 2;
    var r = Math.sqrt(q);
    x1 = (-b + r) / (2 * a);
    x2 = (-b - r) / (2 * a);
  } else if (q == 0) {
    numRoots = 1;
    x1 = -b / (2 * a);
  } else {
    numRoots = 0;
  }
}
```

Example: Quadratic Equation

\[ ax^2 + bx + c = 0 \]

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]
function roots(a, b, c /*doubles*/) {
    var q = b * b - 4 * a * c;
    if (q > 0 && a != 0) {
        numRoots = 2;
        var r = Math.sqrt(q);
        x1 = (-b + r) / (2 * a);
        x2 = (-b - r) / (2 * a);
    } else if (q == 0) {
        numRoots = 1;
        x1 = -b / (2 * a);
    } else {
        numRoots = 0;
    }
}

Fails if a==0 and b*b-4*a*c==0 (division by zero)

\[ ax^2 + bx + c = 0 \]
\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]
Approach 1: Exhaustive Testing

- Check function for all possible inputs → Not feasible, even for trivial programs

```c
function roots(a, b, c /* doubles */) {
    ...
}
```

- Assuming that doubles are 64-bit values, we get
  \[ (2^{64})^3 \approx 10^{58} \] possible values

- Programs with heap structures:
  Much larger state space
Approach 2: Random Testing

Naive approach:  
Select test data uniformly

```c
function roots(a, b, c /* doubles */) {
    ...
}
```

- Fails if \( a == 0 \) and \( b^2 - 4ac == 0 \)  
  (division by zero)
- Probability of selecting \( a == 0 \) and \( b == 0 \) is \( \frac{1}{10^{38}} \)
Random Testing: Discussion

- **Main advantage:** Generates test data **fully automatically**
  - Avoids programmer/tester bias
  - Tests **robustness**, especially handling of invalid and unusual input

- If all inputs considered equally valuable:
  - Small chance to hit a particular bug

- We’ll see more clever approaches later
Approach 3: Functional Testing

Use specification:

Given three values $a, b, c$, compute all solutions of $ax^2 + bx + c = 0$.

Two solutions:
- $a \neq 0$ and $b^2 - 4ac > 0$

One solution:
- $a = 0$ and $b \neq 0$, or
- $a \neq 0$ and $b^2 - 4ac = 0$

No solution:
- $a = 0$, $b = 0$, and $c \neq 0$, or
- $a \neq 0$ and $b^2 - 4ac < 0$

Construct test case for each case of the specification
Functional Testing: Discussion

■ Focuses in *input/output behavior*

■ *Attempts to find*
  - incorrect or missing functionalities
  - interface errors

■ *Limitations:*
  - Not effective for finding coding errors, e.g., buffer overflow
  - Does not reveal errors in specification, e.g., missing use case
Approach 4: Structural Testing

Use knowledge about code:

```javascript
function roots(a, b, c /* doubles */) {
    var q = b * b - 4 * a * c;
    if (q > 0 && a != 0) {
        ...
    } else if (q == 0) {
        ...
    } else {
        ...
    }
}
```

Test this case
Test this case
Test this case
Approach 4: Structural Testing

Use knowledge about code:

```javascript
function roots(a, b, c /* doubles */) {
    var q = b * b - 4 * a * c;
    if (q > 0 && a != 0) {
        ...
    } else if (q == 0) {
        ...
    } else {
        ...
    }
}
```

Test this case
Test this case
Test this case

May still miss error, e.g., when testing this case with a==1, b==2, c==1.
Structural Testing: Discussion

■ Goal: **Cover all code**

■ **Not well suited for system testing**
  - Does not detect missing functionality
  - Requires knowledge about code, which external testers and clients may not have (and do not care about)
  - Covering all code leads to highly redundant tests
## Testing Approaches: Summary

<table>
<thead>
<tr>
<th>Testing Approach</th>
<th>Goal</th>
<th>Testing Method</th>
<th>Suitable for</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaustive testing</td>
<td>Cover all input</td>
<td>Black-box testing</td>
<td>Real-world programs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random testing</td>
<td>Automation</td>
<td>Black-box testing</td>
<td>All levels</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional testing</td>
<td>Cover all requirements</td>
<td>Black-box testing</td>
<td>All levels</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structural testing</td>
<td>Cover all code</td>
<td>White-box testing</td>
<td>Unit testing</td>
</tr>
</tbody>
</table>
Testing Approaches: Summary

Exhaustive testing
- Goal: Cover all input
- Not feasible for real-world programs

Random testing
- Goal: Automation
- Black-box testing
- Suitable for all levels

Focus of rest of today’s lecture

Functional testing
- Goal: Cover all requirements
- Black-box testing
- Suitable for all levels

Structural testing
- Goal: Cover all code
- White-box testing
- Suitable for unit testing
Outline

■ Overview

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

■ Data flow testing
  □ DU-pair coverage

■ Interpretation of coverage
Motivating Example

Given an array of numbers, sort the array in-place in ascending order.

```javascript
function sort(a) {
    if (!a || a.length < 2) {
        return; // array is trivially sorted
    }
    for (var i = 0; i < a.length - 1; i++) {
        if (a[i] < a[i+1]) {
            break;
        }
    }
    if (i >= a.length - 1) {
        return;
    // ... (use quicksort to sort array)
    }
}
```
Motivating Example

Given an array of numbers, sort the array in-place in ascending order.

```javascript
function sort(a) {
    if (!a || a.length < 2)
        return; // array is trivially sorted
    for (var i = 0; i < a.length - 1; i++) {
        if (a[i] < a[i+1]) {
            break; // Error: Check for sortedness should use >
        }
    }
    if (i >= a.length - 1)
        return;
    // ... (use quicksort to sort array)
}
```
Example Test Suite

sort(null);

sort(undefined);

sort([]);

sort([1]);

sort([1, 2, 3]);

Test invalid input

Test valid input
Example Test Suite

sort(null);
Test invalid input

sort(undefined);

sort([]);
Test valid input

sort([1]);

sort([1,2,3]);

How effective is this test suite?
Example Test Suite

sort(null);
sort(undefined);
sort([]);
sort([1]);
sort([1,2,3]);

Test invalid input
Test valid input

How effective is this test suite?

Coverage: Measure for degree to which code is tested by a test suite
Control Flow Testing

- Idea: **Cover** as many different flows of **control** as possible

- Based on **control flow graph**
  - Nodes: Basic blocks (sequence of statements that are always executed together) or statements
  - Edges: Flow of control

- Idea: More parts of CFG executed gives higher chance to uncover bug
function sort(a) {
    if (!a || a.length < 2)
        return; // array is trivially sorted
    for (var i = 0; i < a.length - 1; i++) {
        if (a[i] < a[i+1]) {
            break;
        }
    }
    if (i >= a.length - 1)
        return
    qsort(a) // use quicksort to sort array
}
Statement Coverage

Assess effectiveness of test suite by measuring **how many statements** of the tested program it **executes**

\[
cov_{stmt} = \frac{\text{nb. of executed statements}}{\text{total nb. of statements}}
\]

- **Intuition**: Can detect bug in a statement only by executing the statement
- **Can also be defined on basic blocks**
Input = [3, 7, 5]

Statement coverage:
6 statements covered
3 statements not covered
→ 67% statement coverage
Is full statement coverage enough?

```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;
}
```
Is full statement coverage enough?

```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;
}
```

Should be `<`, otherwise goes out of bounds.
Two tests:
- $a = \text{undefined}, x = 23$
- $a = [1, 23], x = 2$

→ 100% statement coverage

By not found!
Branch Coverage

Idea: Test all possible branches in the CFG

\[ COV_{\text{branch}} = \frac{\text{nb. of executed branches}}{\text{total nb. of branches}} \]

- An edge \((a, b, c)\) in the CFG is a branch iff there is another edge \((a, b', c')\) with \(b \neq b'\)
- If code contains no branches, coverage defined as 100%
Branch coverage example:

Tests:
- $a = \text{undefined}, \ x = 23$
- $a = [1,2], \ x = 2$

Branch coverage:
- execute 5/6 branches
- 83% coverage

Add 3rd test:
- $a = [1], \ x = \text{undefined}$
- 100% coverage
Branch Coverage: Discussion

- Leads to more thorough testing than statement coverage
  - Complete branch coverage implies complete statement coverage
  - But: ”at least n% branch coverage” does not imply ”at least n% statement coverage”

- Most widely used criterion in industry
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;
}
```
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]
i++

false
return res;

exit

A single test can achieve
100% branch coverage:

a = [1] ✓

But: Buy missed!
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;  // Division by zero
    else
        return 5 / y;
}
```
Path Coverage

Idea:
Test all possible paths through the CFG

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

- 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 \( (n_i, n_{i+1}, c) \) for all \( 1 \leq i < k \)
function fourPaths(a, b)

x = 0
y = 0

Two tests:
1. a = true, b = false  
2. a = false, b = true

\[ 100\% \text{ branch coverage} \]

Buy missed!

Path coverage?
\[ 2/4 = 50\% \]

More tests:
1. a = true, b = true
2. a = false, b = false

\[ 100\% \text{ coverage} \]
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;
}
```
function contains(a, x)

entry

let

true, false

var found = false;

return false;

var i = 0

i <= a.length

true

a[i] == x

false, true

found = true

return found;

exit
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

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

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

entry

var j = a.length - 1;

var res = [];

var i = 0

i < a.length

true

res[j] = a[i]

false

return res;

i++

exit

Test:

a = [1]

Loop coverage?

→ 1/3 possible cases

33% loop coverage

More tests:

a = []

a = [1, 2]

→ 100% loop coverage
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;
    }
}
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 operational semantics

- extend configuration: \(<P, s> \rightarrow <P, s, b>\)

  branch cov. map: \(Loc \rightarrow \{T,F\}\)

- extend transition rules:
  assume that each command is annotated with its code location: e.g., \(\langle C_1, s_1, b \rightarrow s_2 \text{ line } 23\rangle\)

\(<\text{if True then } C_1\text{ else } C_2 \rightarrow C_1, s_1, b \rightarrow C_2, s_2, T\rangle\) (ifr)
Outline

- Overview

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

- Data flow testing
  - DU-pair coverage

- Interpretation of coverage