Program Testing and Analysis: Manual Testing (Part 2)

Dr. Michael Pradel
Software Lab, TU Darmstadt

Partly based on slides from Peter Müller, ETH Zurich
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]);
```

- 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

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
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.
function fourPaths(a, b)

for variable x:

1: var x = 1;

2: var y = 1;

3: a

4: x = 0

5: y = 0

6: b

7: return 5 / x;

8: return 5 / y;

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

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

\[ \text{COV}_{DU} = \frac{\text{Nb. of executed DU-pairs}}{\text{Total nb. of DU-pairs}} \]
Two tests:

- \( a = \text{true}, \ b = \text{false} \)
- \( a = \text{false}, \ b = \text{true} \)

DU-pairs for \( x \): \((1,7), (4,7)\)
DU-pairs for \( y \): \((2,8), (5,8)\)

\( \Rightarrow 50\% \) DU coverage

For full DU coverage: Add test:

- \( a = \text{true}, \ b = \text{true} \)
- \( a = \text{false}, \ b = \text{false} \)
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 (Manual Testing)

■ 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
Program Testing and Analysis: Random and Fuzz Testing

Dr. Michael Pradel
Software Lab, TU Darmstadt
Outline

■ Feedback-directed random test generation
  Based on *Feedback-Directed Random Test Generation*, Pacheco et al., ICSE 2007

■ Adaptive random testing
  Based on *ARTOO: Adaptive Random Testing for Object-oriented Software*, Ciupa et al., ICSE 2008

■ Fuzz testing
  Based on *Fuzzing with Code Fragments*, Holler et al., USENIX Security 2012
Motivating Examples

Two randomly generated tests:

Set s = new HashSet();
s.add("hi");
assertTrue(s.equals(s));

Set s = new HashSet();
s.add("hi");
s.isEmpty();
s.isEmpty();
assertTrue(s.equals(s));
Motivating Examples

Two randomly generated tests:

Set s = new HashSet();
s.add("hi");
assertTrue(s.equals(s));

Set s = new HashSet();
s.add("hi");
\boxed{s.isEmpty()};
assertTrue(s.equals(s));

Only difference
Motivating Examples

Two randomly generated tests:

```java
Set s = new HashSet();
s.add("hi");
assertTrue(s.equals(s));
```

```java
Set s = new HashSet();
s.add("hi");
s.isEmpty();
assertTrue(s.equals(s));
```

Redundant test
Motivating Examples (2)

Three randomly generated tests:

Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));

Date d = new Date(2006, 2, 14);
d.setMonth(-1);
assertTrue(d.equals(d));

Date d = new Date(2006, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));
Motivating Examples (2)

Three randomly generated tests:

```java
Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));

Date d = new Date(2006, 2, 14);
d.setMonth(-1);
assertTrue(d.equals(d));
```

```java
Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));
```

```java
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));
```

Violates pre-condition
Motivating Examples (2)

Three randomly generated tests:

Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));

Illegal tests

Date d = new Date(2006, 2, 14);
d.setMonth(-1);
assertTrue(d.equals(d));

Date d = new Date(2006, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));
Feedback-directed Test Generation

Idea: **Guide** randomized **creation** of new test inputs by **feedback** about execution of previous inputs

- Avoid redundant inputs
- Avoid illegal inputs

- Test input here means **sequence of method calls**
- Software under test: Classes in Java-like language
Approach

- **Build test inputs incrementally**
  - New test inputs extend previous ones
- **As soon as test input is created, execute it**
- **Use execution results to guide generation**
  - away from redundant or illegal method sequences
  - toward sequences that create new object states
Randoop: Implementation of feedback-directed random test generation

Input:
- Classes under test
- Time limit
- Set of contracts
  - Method contracts, e.g., `o.hashCode()` throws no exception
  - Object invariants, e.g.,
    ```java
    o.equals(o) == true
    ```

Output: Test cases with assertions
Example

```
HashMap h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
LinkedList l = new LinkedList();
l.addFirst(a);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
```
HashMap h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
LinkedList l = new LinkedList();
l.addFirst(a);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));

Fails when executed
Example

HashMap h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
LinkedList l = new LinkedList();
l.addFirst(a);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
Algorithm

1. Initialize seed components: \( i=0; \ b=false; \ ... \)

2. Do until time limit expires:
   - Create a new sequence
     - Randomly pick a method \( T_0.m(T_1, ..., T_k)/T_{\text{ret}} \)
     - For each \( T_i \), randomly pick a sequence \( S_i \) from the components that constructs a value \( v_i \) of type \( T_i \)
     - Create new sequence
       \( S_{\text{new}} = S_1; ...; S_k; T_{\text{ret}} \ v_{\text{new}} = m(v_1, ..., v_k); \)
     - If \( S_{\text{new}} \) was previously created (lexically), go to
   - Classify the sequence \( S_{\text{new}} \)
     - May discard, output as test case, or add to components
Classifying a Sequence

start → execute and check contracts → contract violated?

no → sequence redundant?

no → components

yes → minimize sequence

yes → discard sequence

contract-violating test case

Image source: Slides by Pacheco et al.
Redundant Sequences

- During generation, maintain a set for all objects created
- Sequence is redundant if all objects created during its execution are in the above set (using `equals()` to compare)
- Could also use more sophisticated state equivalence methods
  - E.g., heap canonicalization
Example of Random algorithm

Classes under test: java. util.*

1) Pick a method
   → No values needed
   → New sequence: `HashMap h = new HashMap();`

2) Classify sequence: no contract violated not redundant
   → add to components
3) Pick method: new HashMap

: HashMap h2 = new HashMap()

4) Classify sequence: no contract violated redundant!

→ discard sequence
5) Pick method: `HashMap.values

Need sequence that constructs value
of type HashMap

→ use sequence 2)

6) Create sequence: `HashMap h = new HashMap();
Collection c = h.values();`

→ Clarify: no contract violated
not redundant

→ add to components etc.
Test Oracles

- Testing only useful if there is an oracle

- Randoop outputs two kinds of oracles
  - Oracle for contract-violating test cases:
    ```java
    assertTrue(u.equals(u));
    ```
  - Oracle for normal-behavior test cases:
    ```java
    assertEquals(2, l.size());
    assertEquals(false, l.isEmpty());
    ```
Quiz

Which of these tests may be created by Randoop?

Test 1:
```java
LinkedList l = new LinkedList();
l.add(23);
```

Test 2:
```java
LinkedList l = new LinkedList();
l.get(-5);
```

Test 3:
```java
LinkedList l = new LinkedList();
l.add(7);
assertEquals(l.getFirst(), 7);
```
Quiz

Which of these tests may be created by Randoop?

Test 1:
```java
LinkedList l = new LinkedList();
l.add(23);  // (oracle missing)
```

Test 2:
```java
LinkedList l = new LinkedList();
l.get(-5);  // (crashes)
```

Test 3:
```java
LinkedList l = new LinkedList();
l.add(7);
assertEquals(l.getFirst(), 7);
```
Results

- Applied to data structure implementations and popular library classes
- Achieves 80-100% basic block coverage
- Finds various bugs in JDK collections, classes from the .NET framework, and Apache libraries

Read Pacheco et al.’s paper for details
Outline

- Feedback-directed random test generation
  Based on Feedback-Directed Random Test Generation, Pacheco et al., ICSE 2007

- Adaptive random testing
  Based on ARTOO: Adaptive Random Testing for Object-oriented Software, Ciupa et al., ICSE 2008

- Fuzz testing
  Based on Fuzzing with Code Fragments, Holler et al., USENIX Security 2012
Adaptive Random Testing

Idea: Testing is more effective when inputs are spread evenly over the input domain

- Generate candidate inputs randomly
- At every step, select input that is furthest away from already tested inputs
Spread Out Evenly?

- Initially proposed for **numeric values**
  - Distance between two values: Euclidean distance

- **Example:** \( f(\text{int } x) \)
  - Suppose to have tested with \( \text{Integer.MAX VALUE} \) and \( \text{Integer.MIN VALUE} \)
  - Next test: 0
Spread Out Evenly?

- Initially proposed for **numeric values**
  - Distance between two values: **Euclidean distance**

- **Example**: `f(int x)`
  - Suppose to have tested with `Integer.MAX_VALUE` and `Integer.MIN_VALUE`
  - Next test: 0

**Challenge:**

**How to compute distance of objects?**
Adaptive random testing

Space of possible inputs

O $\ldots$ already tested
Object Distance

- Measure **how different two objects are**
- Object: Primitive values, dynamic type, and non-primitive values recursively referred to

\[
dist(p, q) = \text{combination(}
\begin{align*}
&\text{elementaryDistance}(p, q), \\
&typeDistance(\text{type}(p), \text{type}(q)), \\
&\text{fieldDistance}(\{ \text{dist}(p.a, q.a) \mid a \in \text{fields}(\text{type}(p) \cap \text{fields}(\text{type}(q))) \})
\end{align*}
\]
Object Distance

- Measure **how different two objects are**
- Object: Primitive values, dynamic type, and non-primitive values recursively referred to

\[
dist(p, q) = \text{combination(}
\begin{array}{l}
\text{elementaryDistance}(p, q), \\
\text{typeDistance}\left(\text{type}(p), \text{type}(q)\right), \\
\text{fieldDistance}\left(\{\text{dist}(p.a, q.a) \mid a \in \text{fields(}\text{type}(p) \cap \text{fields(}\text{type}(q)))\}\right)
\end{array}
\)
\]

Does not require traversing the object
Object Distance

- Measure **how different two objects are**
- Object: Primitive values, dynamic type, and non-primitive values recursively referred to

\[
dist(p, q) = \text{combination}(
\text{elementaryDistance}(p, q),
\text{typeDistance}(\text{type}(p), \text{type}(q)),
\text{fieldDistance}(
\{\text{dist}(p.a, q.a) | a \in \text{fields}(\text{type}(p) \cap \text{fields}(\text{type}(q)))\})
)\]

Recursively defined
Elementary Distance

Fixed functions for each possible type:

- For numbers: \( F(|p - q|) \), where \( F \) is a monotonically non-decreasing function with \( F(0) = 0 \)
- For characters: 0 if identical, \( C \) otherwise
- For booleans: 0 if identical, \( B \) otherwise
- For strings: the Levenshtein distance
- For references: 0 if identical, \( R \) if different but none is null, \( V \) if only one of them is null

\[ C, B, R, V \in \mathbb{N} \]
Examples: Elementary Distance

- int i = 3, j = 9 → dist(13 - 91) = 6
- char c = 'a', d = 'a' → dist(c, d) = 0
- String s = "foo", t = "too" → dist(s, t) = 1
- Object o = null, p = new ArrayList() → dist(o, p) = V
Type Distance

Distance between two types

\[ typeDistance(t, u) = \lambda \times pathLength(t, u) + \nu \times \sum_{a \in \text{nonShared}(t, u)} \text{weight}_a \]

- \( pathLength(t, u) \) is the minimal distance to a common ancestor in class hierarchy
- \( \text{nonShared}(t, u) \) is the set of non-shared fields
- \( \text{weight}_a \) is the weight for a specific field

\( \lambda, \nu \in \mathbb{N} \)
Examples: Type Distance

\[ \text{dist} (B, C) = \lambda \cdot 1 + \nu \cdot (1+1) \]

\[ \text{dist} (A, B) = \lambda \cdot 0 + \nu \cdot (1) \]

\[ \text{dist} (B, D) = \lambda \cdot 1 + \nu \cdot (1+1) \]
Field Distance

Recursively compute distance of all shared fields

\[
\text{fieldDistance}(p, q) = \sum_{a} \text{weight}_a \times (\text{dist}(p.a, q.a))
\]

Arithmetic mean: Avoid giving too much weight to objects with many fields
Algorithm for Selecting Inputs

- Global sets $usedObjects$ and $candidateObjects$
- Choose object for next test:
  - Initialize $bestDistSum = 0$ and $bestObj = null$
  - for each $c \in candidateObjects$:
    - for each $u \in usedObjects$:
      - $distSum += dist(c,u)$
    - if $distSum > bestDistSum$:
      - $bestDistSum = distSum; bestObj = c$
  - Remove $bestObj$ from $candidateObjects$, add to $usedObjects$ instead, and run test with $bestObj$
Example

Method under test:

Account.transfer(Account dst, int amount)

Pool of candidates:

■ Accounts
  □ a1: owner=”A” and balance=6782832
  □ a2: owner=’B’ and balance=10
  □ a3: owner=”O” and balance=99
  □ a4: null

■ Integers:
  □ i1: 100, i2: 287391, i3: 0, i4: -50
Example: Adaptive Random Testing

First call: a3. transfer (a1, i2)

Second call: a1. transfer (a4, i4)
Results

- Implemented for Eiffel
- Use randomly generated objects as candidates
- Use Eiffel’s contracts (pre- and post-conditions, class invariants) as test oracle

Comparison with random testing:
- Find bugs with 5x fewer tests
- **But**: Takes 1.6x the time of random testing
Outline

- Feedback-directed random test generation
  Based on *Feedback-Directed Random Test Generation*, Pacheco et al., ICSE 2007

- Adaptive random testing
  Based on *ARToo: Adaptive Random Testing for Object-oriented Software*, Ciupa et al., ICSE 2008

- Fuzz testing
  Based on *Fuzzing with Code Fragments*, Holler et al., USENIX Security 2012