Advanced Software Testing and Debugging (CS598)
Guided Unit Test Generation

Spring 2022
Lingming Zhang
Testing: basic concepts

- **Test case** (or, simply **test**): an execution of the software with a given test input, including:
  - Input values
  - Sometimes include execution steps
  - Expected outputs (**test oracle**)

- **Test suite**: a finite set of tests
  - Usually can be run together in sequence

- **Test adequacy**: a measurement to evaluate the test quality
  - Such as code coverage
Testing: levels

• Unit Testing
  • Test each single module in isolation

• Integration Testing
  • Test the interaction between modules

• System Testing
  • Test the system as a whole, by developers

• Acceptance Testing
  • Validate the system against user requirements, by customers with no formal test cases
Types of test generation

• **Black-box (functional) vs. white-box (structural) testing**

  - **Black-box test generation**: generates tests based on the functionality of the program
  - **White-box test generation**: generates tests based on the source-code structure of the program
White-box generation granularities

• Projects providing a number of public APIs for external use (e.g., JDK lib)
  • **Method-level test generation**: consider various method invocation sequences to expose possible faults

  [Guided unit test generation (this class)]

• Projects usually used as a whole (e.g., scientific computation software)
  • **Path-level generation**: consider all the possible execution paths to cover most program elements

  [Symbolic execution (next class)]
This class

• Feedback-directed Random Test Generation (ICSE'07)
• Whole Test Suite Generation (TSE'12)
Problem: unit test generation

Program under test:
```java
public class Math{
    public static int sum(int a, int b){
        return a+b;
    }
    ...
}
```

Example JUnit test:
```java
public class MathTest{
    @Test
    public void testSum(){
        int a=1;
        int b=1;
        int c=Math.sum(a, b);
        assertEquals(2,c);
    }
    ...
}
```

Is this an important problem?

Is this an important problem?

84,377 lines of source code
86,924 lines of unit-test code

Commons-Math
How to perform random white-box test generation?

- Need to generate a random sequence of invocations, where each requires:
  - A random method
  - Some random parameters
  - A random receiver object
  - Not required for static methods

```
public class HashSet extends Set{
    public boolean add(Object o){...}
    public boolean remove(Object o){...}
    public boolean isEmpty(){...}
    public boolean equals(Object o){...}
}
```

Program under test

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

Generated test t1

```
Set s = new HashSet();
s.add("hi");
s.remove(null);
```

Generated test t2

```
Set s = new HashSet();
s.isEmpty();
s.remove("no");
s.isEmpty();
s.add("no");
s.isEmpty();
s.isEmpty();
...
```

Generated test t3
Random method-sequence generation: limitations

• Does not have test oracles
  • E.g., an ideal test oracle for the test below: `assertEquals(1, s.size())`

• Cannot generate complex tests
  • E.g., the parameters of some method invocations can be generated by other method invocations

• Can have many redundant & illegal tests

```java
Set s = new HashSet();
s.isEmpty();
s.isEmpty();
s.remove("no");
s.isEmpty();
s.add("no");
s.isEmpty();
s.isEmpty();
```

A random test
Random method-sequence generation: redundant & illegal tests

1. Useful test:
   Set s = new HashSet();
   s.add(“hi”);

2. Redundant test:
   Set s = new HashSet();
   s.add(“hi”);
   s.isEmpty();

3. Useful test:
   Date d = new Date(2006, 2, 14);

4. Illegal test:
   Date d = new Date(2006, 2, 14);
   d.setMonth(-1);  // pre: argument >= 0

5. Illegal test:
   Date d = new Date(2006, 2, 14);
   d.setMonth(-1);  // pre: argument >= 0
   d.setDay(5);

Should not output

Should not even generate
Randoop: feedback-directed (adaptive) random test generation

• Use code contracts as test oracles
• Build test inputs incrementally
  • New test inputs extend previous ones
  • In this context, a test input is a method sequence
• As soon as a test is created, use its execution results to guide generation
  • away from redundant or illegal method sequences
  • towards sequences that create new object states
Randoop input/output

• **Input:**
  • Classes under test
  • Time limit
  • Set of contracts
    • Method contracts (e.g. “o.hashCode() throws no exception”)
    • Object invariants (e.g. “o.equals(o) == true”)

• **Output:** contract-violating test cases

```java
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 on Sun’s JDK 1.5/1.6 when executed
Randoop: algorithm

• Seed value pool for primitive types
  • pool = { 0, 1, true, false, “hi”, null ... }

• Do until time limit expires:
  • Create a new sequence
    • Randomly pick a method call $m(T_1...T_k)/T_{ret}$
    • For each input parameter of type $T_i$, randomly pick a sequence $S_i$ from the value pool that constructs an object $v_i$ of type $T_i$
    • Create new sequence $S_{new} = S_1; ... ; S_k ; T_{ret} v_{new} = m(v_1...v_k)$
    • if $S_{new}$ was previously created (lexically), go to first step
  • Classify the new sequence $S_{new}$
    • May discard, output as test case, or add to pool
Randoop: example

Program under test:
public class A{
    public A() {...}
    public B m1(A a1) {...}
}
public class B{
    public B(int i){...}
    public void m2(B b, A a) {...}
}

Value pool:
S1: B b1=new B(0);

Test1:
B b1=new B(0);
Randoop: example

Program under test:
public class A{
    public A() {...}
    public B m1(A a1) {...}
}
public class B{
    public B(int i) {...}
    public void m2(B b, A a) {...}
}

Value pool:
S1: B b1=new B(0);
S2: A a1=new A();

Test1:
B b1=new B(0);

Test2:
A a1=new A();

{0, 1, null, "hi", ...}
Randoop: example

Program under test:
```java
public class A{
    public A() {...}
    public B m1(A a1) {...}
}
public class B{
    public B(int i) {...}
    public void m2(B b, A a) {...}
}
```

Test1:
```java
B b1=new B(0);
```

Test2:
```java
A a1=new A();
```

Test3:
```java
A a1=new A(); //reused from s2
B b2=a1.m1(a1);
```

Value pool:
```java
[0, 1, null, "hi", ...]
```
Randoop: example

Program under test:
public class A{
    public A() {...}
    public B m1(A a1) {...}
}
public class B{
    public B(int i) {...}
    public void m2(B b, A a) {...}
}

Test1:
B b1=new B(0);

Test2:
A a1=new A();

Test3:
A a1=new A();
B b2=a1.m1(a1);

Test4:
B b1=new B(0); //reused from s1
A a1=new A();
B b2=a1.m1(a1); //reused from s3
b1.m2(b2, a1);

Value pool:
S1: B b1=new B(0);
S2: A a1=new A();
S3: A a1=new A();
B b2=a1.m1(a1);
{0, 1, null, “hi”,…}

S4: …
Classifying a sequence

Start → Execute and check contracts → Contract violated?

Yes → Minimize sequence

No → Sequence redundant?

Yes → Discard sequence

No → Value pool
Redundant sequences

• During generation, maintain a set of all objects created
• A sequence is redundant if all the objects created during its execution are members of the above set (using `equals` to compare)
• Could also use more sophisticated state equivalence methods
  • E.g. heap canonicalization used in model checkers
Tool support

• **Input:**
  - An assembly (for .NET) or a list of classes (for Java)
  - Generation time limit
  - Optional: a set of contracts to augment default contracts

• **Output:** a test suite (JUnit or Nunit) containing
  - Contract-violating test cases
  - Normal-behavior test cases
Randoop outputs oracles

• Oracle for contract-violating tests:

```java
Object o = new Object();
LinkedList l = new LinkedList();
l.addFirst(o);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u)); //expected to fail
```

• Oracle for normal-behavior tests (regression tests):

```java
Object o = new Object();
LinkedList l = new LinkedList();
l.addFirst(o);
l.add(o);
assertEquals(2, l.size()); //expected to pass
assertEquals(false, l.isEmpty()); //expected to pass
```
Some Randoop options

• Avoid use of null

Statically:
Object o = new Object();
LinkedList l = new LinkedList();
l.add(null);

Dynamically:
Object o = returnNull();
LinkedList l = new LinkedList();
l.add(o);

• Bias random selection
  • Favor shorter sequences
  • Favor methods that have been less covered
  • Use constants mined from source code

• Source code available:
  • https://randoop.github.io/randoop/
# Code coverage by Randoop

<table>
<thead>
<tr>
<th>Data structure programs</th>
<th>Time (s)</th>
<th>Branch cov.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded stack (30 LOC)</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>Unbounded stack (59 LOC)</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>BS Tree (91 LOC)</td>
<td>1</td>
<td>96%</td>
</tr>
<tr>
<td>Binomial heap (309 LOC)</td>
<td>1</td>
<td>84%</td>
</tr>
<tr>
<td>Linked list (253 LOC)</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>Tree map (370 LOC)</td>
<td>1</td>
<td>81%</td>
</tr>
<tr>
<td>Heap array (71 LOC)</td>
<td>1</td>
<td>100%</td>
</tr>
</tbody>
</table>
Bug detection by Randoop: subjects

<table>
<thead>
<tr>
<th>Subjects</th>
<th>LOC</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK (2 libraries) (java.util, javax.xml)</td>
<td>53K</td>
<td>272</td>
</tr>
<tr>
<td>Apache commons (6 libraries) (logging, primitives, chain, jelly, math, collections)</td>
<td>114K</td>
<td>974</td>
</tr>
<tr>
<td>.Net libraries (6 libraries)</td>
<td>615K</td>
<td>3455</td>
</tr>
</tbody>
</table>
Bug detection by Randoop: methodology

• Ran Randoop on each library
  • Used default time limit (2 minutes)

• Contracts:
  • `o.equals(o)==true`
  • `o.equals(o)` throws no exception
  • `o.hashCode()` throws no exception
  • `o.toString()` throw no exception
  • No null inputs and:
    • Java: No NPEs
    • .NET: No NPEs, out-of-bounds, of illegal state exceptions
## Bug detection by Randoop: subjects

<table>
<thead>
<tr>
<th>Subjects</th>
<th>Failed tests</th>
<th>Unique failed tests</th>
<th>Error-revealing tests</th>
<th>Distinct errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK</td>
<td>613</td>
<td>32</td>
<td>29</td>
<td>8</td>
</tr>
<tr>
<td>Apache commons</td>
<td>3,044</td>
<td>187</td>
<td>29</td>
<td>6</td>
</tr>
<tr>
<td>.Net framework</td>
<td>543</td>
<td>205</td>
<td>196</td>
<td>196</td>
</tr>
<tr>
<td>Total</td>
<td>4,200</td>
<td>424</td>
<td>254</td>
<td>210</td>
</tr>
</tbody>
</table>
Errors found: examples

- JDK Collections classes have 4 methods that create objects violating `o.equals(o)` contract
- Javax.xml creates objects that cause `hashCode` and `toString` to crash, even though objects are well-formed XML constructs
- Apache libraries have constructors that leave fields unset, leading to NPE on calls of `equals`, `hashCode` and `toString` (this only counts as one bug)
- .Net framework has at least 175 methods that throw an exception forbidden by the library specification (NPE, out-of-bounds, of illegal state exception)
- .Net framework has 8 methods that violate `o.equals(o)`
- .Net framework loops forever on a legal but unexpected input
Has Randoop been compared to existing solutions?

• Systematic testing:
  • Java PathFinder (JPF)
  • jCUTE

• Undirected Random testing:
  • Randoop-feedback
  • JCrasher
Regression testing scenario

• Randoop can create regression oracles

• Generated test cases using JDK 1.5
  • Randoop generated 41K regression test cases

• Ran resulting test cases on
  • JDK 1.6 Beta
    • 25 test cases failed
  • Sun’s implementation of the JDK
    • 73 test cases failed

• Failing test cases pointed to 12 distinct errors

• These errors were not found by the extensive compliance test suite that Sun provides to JDK developers

```java
Object o = new Object();
LinkedList l = new LinkedList();
l.addFirst(o);
l.add(o);
assertEquals(2, l.size()); // expected to pass
assertEquals(false, l.isEmpty()); // expected to pass
```
Randoop: applications
Discussion

• Strengths
• Limitations
• Future work
This class

- Feedback-directed Random Test Generation (ICSE'07)
- Whole Test Suite Generation (TSE'12)
Genetic algorithm
The eight queens problem

Perfect!
The eight queens problem

2 attacks!
The eight queens problem

3 attacks!

Easily solved via recursion or dynamic programming!
How about 800 queens problem?!
Genetic algorithm: 8/800 queens problem
Genetic algorithm: test generation (aka search-based test generation)

Set s = new HashSet();
    s.isEmpty();
    s.remove("no");
    s.isEmpty();
    s.add("no");
    s.isEmpty();
    ...

Test suite
Crossover and mutation

Fig. 3. Crossover and mutation are the basic operators for the search using a GA. Crossover is applied at test suite level; mutation is applied to test cases and test suites.
Fitness function and selection

\[ \text{fitness}(T) = |M| - |M_T| + \sum_{b \in B} \text{dist}(b, T) \]

\[ \text{dist}(b, T) = \begin{cases} 0 & \text{If the branch is covered} \\ d(b, T) & \text{If the predicate is executed at least twice} \\ 1 & \text{Otherwise} \end{cases} \]

Branch distance, \( d(b, T) \), describes how “close” \( b \) is to being covered (normalized to \([0,1]\))

If \( A = B \) \( \Rightarrow \) \( d(b, T) = | A - B | \)
Discussion

• Strengths
• Limitations
• Future work
Thanks and stay safe!