Automated Test Generation

Mayur Naik
CIS 573 – Fall 2019
Outline

• Previously: Random testing (Fuzzing)
  – Security, mobile apps, concurrency

• Systematic testing: Korat
  – Linked data structures

• Feedback-directed random testing: Randoop
  – Classes, libraries
Korat

• A test-generation research project

• Idea
  – Leverage pre-conditions and post-conditions to generate tests automatically

• But how?
The Problem

• There are **infinitely** many tests
  – Which finite subset should we choose?

• And even **finite** subsets can be huge

• Need a subset which is:
  – **concise**: Avoids **illegal** and **redundant** tests
  – **diverse**: Gives **good coverage**
An Insight

• Often can do a good job by systematically testing all inputs up to a small size

• **Small Test Case Hypothesis:**
  – If there is any test that causes the program to fail, there is a small such test

• If a list function works for lists of length 0 through 3, probably works for all lists
  – E.g., because the function is oblivious to the length
How Do We Generate Test Inputs?

• Use the types

```java
class BinaryTree {
    Node root;
    class Node {
        Node left;
        Node right;
    }
}
```

• The class declaration shows what values (or null) can fill each field

• Simply enumerate all possible shapes with a fixed set of Nodes
Scheme for Representing Shapes

- Order all possible values of each field
- Order all fields into a vector
- Each shape == vector of field values

e.g.: BinaryTree of up to 3 Nodes:

```java
class BinaryTree {
    Node root;
    class Node {
        Node left;
        Node right;
    }
}
```
QUIZ: Representing Shapes

Fill in the field values in each vector to represent the depicted shape:

<table>
<thead>
<tr>
<th>root</th>
<th>left</th>
<th>right</th>
<th>left</th>
<th>right</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>N0</td>
<td></td>
<td></td>
<td>N1</td>
<td></td>
<td>N2</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

```
   N0
   /  \
N1   N2
```

```
   N0
   /  \
N1   N2
```

```
   N0
   /  \
N1   N2
```
QUIZ: Representing Shapes

Fill in the field values in each vector to represent the depicted shape:

\[
\begin{array}{c|c|c|c|c|c|c}
\text{root} & N0 & \text{left} & \text{right} & N1 & \text{left} & \text{right} & N2 & \text{left} & \text{right} \\
\hline
\text{null} & \text{N0} & \text{null} & \text{null} & \text{null} & \text{null} & \text{null} & \text{null} & \text{null} & \text{null} \\
\end{array}
\]
A Simple Algorithm

- User selects maximum input size $k$
- Generate all possible inputs up to size $k$
- Discard inputs where pre-condition is false
- Run program on remaining inputs
- Check results using post-condition
QUIZ: Enumerating Shapes

Korat represents each input shape as a vector of the following form:

root     N0         N1         N2
  left   right  left   right  left   right

What is the total number of vectors of the above form?
QUIZ: Enumerating Shapes

Korat represents each input shape as a vector of the following form:

\[
\begin{array}{cccc}
\text{root} & N0 & N1 & N2 \\
\text{left} & \text{right} & \text{left} & \text{right} & \text{left} & \text{right}
\end{array}
\]

What is the total number of vectors of the above form?

16384
The General Case for Binary Trees

• How many binary trees are there of size <= k?

• Calculation:
  – A BinaryTree object, bt
  – k Node objects, n0, n1, n2, ...
  – 2k+1 Node pointers
    • root (for bt)
    • left, right (for each Node object)
  – k+1 possible values (n0, n1, n2, ... or null) per pointer

• \((k+1)^{(2k+1)}\) possible “binary trees”
A Lot of “Trees”!

• The number of “trees” explodes rapidly
  – $k = 3$: over 16,000 “trees”
  – $k = 4$: over 1,900,000 “trees”
  – $k = 5$: over 360,000,000 “trees”

• Limits us to testing only very small input sizes

• Can we do better?
An Overestimate

• \((k+1)^{(2k+1)}\) trees is a gross overestimate!

• Many of the shapes are not even trees:

• And many are isomorphic:
How Many Trees?

Only 9 distinct binary trees with at most 3 nodes

Diagram showing the 9 distinct binary trees with at most 3 nodes.
Another Insight

• Avoid generating inputs that don’t satisfy the pre-condition in the first place

• Use the pre-condition to guide the generation of tests
The Technique

• Instrument the pre-condition
  – Add code to observe its actions
  – Record fields accessed by the pre-condition

• Observation:
  – If the pre-condition doesn’t access a field, then pre-condition doesn’t depend on the field.
The Pre-Condition for Binary Trees

• Root may be null
• If root is not null:
  – No cycles
  – Each node (except root) has one parent
  – Root has no parent

```java
class BinaryTree {
    Node root;
    class Node {
        Node left;
        Node right;
    }
}
```
public boolean repOK(BinaryTree bt) {
    if (bt.root == null) return true;
    Set visited = new HashSet();
    List workList = new LinkedList();
    visited.add(bt.root);
    workList.add(bt.root);
    while (!workList.isEmpty()) {
        Node current = workList.removeFirst();
        if (current.left != null) {
            if (!visited.add(current.left)) return false;
            workList.add(current.left);
        }
        // similarly for current.right
    }
    return true;
}
The Pre-Condition for Binary Trees

```java
public boolean repOK(BinaryTree bt) {
    if (bt.root == null) return true;
    Set visited = new HashSet();
    List workList = new LinkedList();
    visited.add(bt.root);
    workList.add(bt.root);
    while (!workList.isEmpty()) {
        Node current = workList.removeFirst();
        if (current.left != null) {
            if (!visited.add(current.left)) return false;
            workList.add(current.left);
        }
        ... // similarly for current.right
    }
    return true;
}

class BinaryTree {
    Node root;
    class Node {
        Node left;
        Node right;
    }
}
```
Example: Using the Pre-Condition

• Consider the following “tree”:

\[ \begin{array}{ccc}
\text{root} & \text{left} & \text{right} \\
\text{null} & \text{null} & \text{N1}
\end{array} \quad \begin{array}{ccc}
\text{left} & \text{right} & \\
\text{null} & \text{N2}
\end{array} \quad \begin{array}{c}
\text{left} \\
\text{null}
\end{array} \]

• The **pre-condition** accesses only the root as it is null

  => Every possible shape for other nodes yields same result

  => This single input eliminates 25% of the tests!
Enumerating Tests

• Shapes are *enumerated* by their associated vectors
  – Initial candidate vector: all fields null
  – Next shape generated by:
    • Expanding last field accessed in pre-condition
    • Backtracking if all possibilities for a field are exhausted

• **Key idea:** Never expand parts of input not examined by pre-condition

• Also: Cleverly checks for and discards shapes *isomorphic* to previously-generated shapes
Example: Enumerating Binary Trees

<table>
<thead>
<tr>
<th>root</th>
<th>N0 left</th>
<th>N0 right</th>
<th>N1 left</th>
<th>N1 right</th>
<th>N2 left</th>
<th>N2 right</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N0</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N0</td>
<td>null</td>
<td>N0</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N0</td>
<td>null</td>
<td>N1</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
QUIZ: Enumerating Binary Trees

What are the next two legal, non-isomorphic shapes Korat generates?

<table>
<thead>
<tr>
<th></th>
<th>N0</th>
<th>N1</th>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>left</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>right</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Correct shapes:
**QUIZ: Enumerating Binary Trees**

What are the next two legal, non-isomorphic shapes Korat generates?

<table>
<thead>
<tr>
<th>root</th>
<th>N0</th>
<th>left</th>
<th>right</th>
<th>N1</th>
<th>left</th>
<th>right</th>
<th>N2</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N0</td>
<td>null</td>
<td></td>
<td>N1</td>
<td>null</td>
<td></td>
<td>N2</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
QUIZ: Enumerating Binary Trees

What are the next two legal, non-isomorphic shapes Korat generates?

<table>
<thead>
<tr>
<th>root</th>
<th>N0</th>
<th>left</th>
<th>right</th>
<th>N1</th>
<th>left</th>
<th>right</th>
<th>N2</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>N0</td>
<td>null</td>
<td>N1</td>
<td>N2</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
QUIZ: Enumerating Binary Trees

What are the next two legal, non-isomorphic shapes Korat generates?

<table>
<thead>
<tr>
<th></th>
<th>root</th>
<th>N0</th>
<th>left</th>
<th>right</th>
<th>N1</th>
<th>left</th>
<th>right</th>
<th>N2</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N0</td>
<td>null</td>
<td>2</td>
<td>N1</td>
<td>3</td>
<td>N2</td>
<td>null</td>
<td>4</td>
<td>null</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- First shape: 1, 2, 3, 4, 7
- Second shape: 1, 2, 5, 4
### Experimental Results

<table>
<thead>
<tr>
<th>benchmark</th>
<th>size</th>
<th>time (sec)</th>
<th>structures generated</th>
<th>candidates considered</th>
<th>state space</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BinaryTree</strong></td>
<td>8</td>
<td>1.53</td>
<td>1430</td>
<td>54418</td>
<td>$2^{53}$</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>3.97</td>
<td>4862</td>
<td>210444</td>
<td>$2^{63}$</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>14.41</td>
<td>16796</td>
<td>815100</td>
<td>$2^{72}$</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>56.21</td>
<td>58786</td>
<td>3162018</td>
<td>$2^{82}$</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>233.59</td>
<td>208012</td>
<td>12284830</td>
<td>$2^{92}$</td>
</tr>
<tr>
<td><strong>HeapArray</strong></td>
<td>6</td>
<td>1.21</td>
<td>13139</td>
<td>64533</td>
<td>$2^{20}$</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>5.21</td>
<td>117562</td>
<td>519968</td>
<td>$2^{25}$</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>42.61</td>
<td>1005075</td>
<td>5231385</td>
<td>$2^{29}$</td>
</tr>
<tr>
<td><strong>LinkedList</strong></td>
<td>8</td>
<td>1.32</td>
<td>4140</td>
<td>5455</td>
<td>$2^{91}$</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>3.58</td>
<td>21147</td>
<td>26635</td>
<td>$2^{105}$</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>16.73</td>
<td>115975</td>
<td>142646</td>
<td>$2^{120}$</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>101.75</td>
<td>678570</td>
<td>821255</td>
<td>$2^{135}$</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>690.00</td>
<td>4213597</td>
<td>5034894</td>
<td>$2^{150}$</td>
</tr>
<tr>
<td><strong>TreeMap</strong></td>
<td>7</td>
<td>8.81</td>
<td>35</td>
<td>256763</td>
<td>$2^{92}$</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>90.93</td>
<td>64</td>
<td>2479398</td>
<td>$2^{111}$</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>2148.50</td>
<td>122</td>
<td>50209400</td>
<td>$2^{130}$</td>
</tr>
</tbody>
</table>
Strengths and Weaknesses

• Strong when we can enumerate all possibilities
  – e.g. Four nodes, two edges per node

=> Good for:
  • Linked data structures
  • Small, easily specified procedures
  • Unit testing

• Weaker when enumeration is weak
  – Integers, Floating-point numbers, Strings
Weaknesses

Only as good as the pre- and post-conditions

```plaintext
Pre: is_member(x, list)
List remove(Element x, List list) {
    if (x == head(list))
        return tail(list);
    else
        return cons(head(list),
                   remove(x, tail(list)));
}
Post: !is_member(x, list')
```
Weaknesses

Only as good as the pre- and post-conditions

Pre: !is_empty(list)
List remove(Element x, List list) {
    if (x == head(list))
        return tail(list);
    else
        return cons(head(list),
                    remove(x, tail(list)));
}
Post: is_list(list')
Feedback-Directed Random Testing

How do we generate a test like this?

```java
public static void test() {
    LinkedList l1 = new LinkedList();
    Object o1 = new Object();
    l1.addFirst(o1);
    TreeSet t1 = new TreeSet(l1);
    Set s1 = Collections.unmodifiableSet(t1);

    // This assertion fails
    assert(s1.equals(s1));
}
```
Overview

Problem with uniform random testing: Creates too many illegal or redundant tests

Idea: Randomly create new test guided by feedback from previously created tests

\[
\text{test} == \text{method sequence}
\]

Recipe:

• Build new sequences incrementally, extending past sequences
• As soon as a sequence is created, execute it
• Use execution results to guide test generation towards sequences that create new object states
Randoop: Input and Output

Input:
• classes under test
• time limit
• set of contracts
  e.g. “o.hashCode() throws no exception”
  e.g. “o.equals(o) == true”

Output:
• contract-violating test cases

```
LinkedList l1 = new LinkedList();
Object o1 = new Object();
l1.addFirst(o1);
TreeSet t1 = new TreeSet(l1);
Set s1 = Collections.unmodifiableSet(t1);
assert(s1.equals(s1));
```

No contract violated up to here  fails when executed
Randooop Algorithm

components = { int i = 0; , boolean b = false; . . . }  
// seed components

Repeat until time limit expires:
• Create a new sequence
  – Randomly pick a method call \( T_{\text{ret}} = m(T_1, . . . , T_n) \)
  – For each argument of type \( T_i \), randomly pick sequence \( S_i \) from components that constructs an object \( v_i \) of that type
  – Create \( S_{\text{new}} = S_1; . . . ; S_n; T_{\text{ret}} v_{\text{new}} = m(v_1 . . . v_n); \)

• Classify new sequence \( S_{\text{new}} \): discard / output as test / add to components
Classifying a Sequence

- execute sequence
- sequence illegal?
  - Yes: Discard sequence
  - No: contract violated?
    - Yes: Output sequence
    - No: sequence redundant?
      - Yes: Discard sequence
      - No: Add to components

Illegal Sequences

• Sequences that “crash” before contract is checked
  – E.g. throw an exception

```java
int i = -1;
Date d = new Date(2006, 2, 14);
d.setMonth(i);    // pre: argument >= 0
assert(d.equals(d));
```
Redundant Sequences

- Maintain set of all objects created in execution of each sequence
- New sequence is redundant if each object created during its execution belongs to above set (using `equals` to compare)
- Could also use more sophisticated state equivalence methods

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

```java
Set s = new HashSet();
s.add("hi");
assertTrue(s.equals(s));
```
Some Errors Found by Randoop

• JDK containers have 4 methods that violate 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

• .Net framework has at least 175 methods that throw an exception forbidden by the library specification (NPE, out-of-bounds, or illegal state exception)

• .Net framework has 8 methods that violate o.equals(o) contract
Write the smallest sequence that Randoop can possibly generate to create a valid BinaryTree.

Once generated, how does Randoop classify it?

- Discards it as illegal
- Outputs it as a bug
- Adds to components for future extension
QUIZ: Randoop Test Generation (Part 1)

Write the smallest sequence that Randoop can possibly generate to create a valid BinaryTree.

```java
BinaryTree bt = new BinaryTree(null);
```

Once generated, how does Randoop classify it?
- Discards it as illegal
- Outputs it as a bug
- Adds to components for future extension

```java
class BinaryTree {
    Node root;
    public BinaryTree(Node r) {
        root = r;
        assert(repOk(this));
    }
    public Node removeRoot() {
        assert(root != null); // ...
    }
}
class Node {
    Node left;
    Node right;
    public Node(Node l, Node r) {
        left = l;
        right = r;
    }
}
```
QUIZ: Randoop Test Generation (Part 2)

Write the smallest sequence that Randoop can possibly generate that violates the assertion in `removeRoot()`.

Once generated, how does Randoop classify it?

- Discards it as illegal
- Outputs it as a bug
- Adds to components for future extension

```java
class BinaryTree {
    Node root;
    public BinaryTree(Node r) {
        root = r;
        assert(repOk(this));
    }
    public Node removeRoot() {
        assert(root != null);
        ...
    }
}

class Node {
    Node left;
    Node right;
    public Node(Node l, Node r) {
        left = l;
        right = r;
    }
}
```
QUIZ: Randoop Test Generation (Part 2)

Write the smallest sequence that Randoop can possibly generate that violates the assertion in `removeRoot()`.

```java
BinaryTree bt = new BinaryTree(null);
bright.removeRoot();
```

Once generated, how does Randoop classify it?

- Discards it as illegal
- Outputs it as a bug
- Adds to components for future extension

```java
class BinaryTree {
    Node root;
    public BinaryTree(Node r) {
        root = r;
        assert(repOk(this));
    }
    public Node removeRoot() {
        assert(root != null);
        ...
    }
}

class Node {
    Node left;
    Node right;
    public Node(Node l, Node r) {
        left = l;
        right = r;
    }
}
```
Write the smallest sequence that Randoop can possibly generate that violates the assertion in BinaryTree’s constructor.

Can Randoop create a BinaryTree object with cycles using the given API?

- Yes
- No
Write the smallest sequence that Randoop can possibly generate that violates the assertion in BinaryTree’s constructor.

Can Randoop create a BinaryTree object with cycles using the given API?

- Yes
- No
QUIZ: Korat and Randoop

Identify which statements are true for each test generation technique:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Korat</th>
<th>Randoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses type information to guide test generation.</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Each test is generated fully independently of past tests.</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Generates tests deterministically.</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Suited to test method sequences.</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Avoids generating redundant tests.</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>
### QUIZ: Korat and Randoop

Identify which statements are true for each test generation technique:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Korat</th>
<th>Randoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses type information to guide test generation.</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Each test is generated fully independently of past tests.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generates tests deterministically.</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Suited to test method sequences.</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Avoids generating redundant tests.</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Test Generation: The Bigger Picture

• Why didn’t automatic test generation become popular decades ago?

• Belief: Weak-type systems
  – Test generation relies heavily on type information
  – C, Lisp just didn’t provide the needed types

• Contemporary languages lend themselves better to test generation
  – Java, UML
What Have We Learned?

• Automatic test generation is a good idea
  – Key: avoid generating illegal and redundant tests

• Even better, it is possible to do
  – At least for unit tests in strongly-typed languages

• Being adopted in industry
  – Likely to become widespread