Advanced Software Testing and Debugging (CS598)
Spec-based Testing

Spring 2022
Lingming Zhang
Reminder

• Check your presentation schedule on course website
• Form your course project team (you can work individually!)
  • Post on Campuswire (#find-teammates) if you need to find a teammate
  • Or let me know if you need help
• Proposal presentation (Feb 17, in class)
• Proposal submission (Feb 21, 11:59pm)
  • Join your group on Campuswire before submission as this is a group assignment
  • If you are working by yourself, join any empty group on Campuswire ("People"->"Groups")
  • If you have a teammate, make sure you join the same group as your teammate
Spec-based test generation

// specification for removing from binary tree
/*@ public normal_behavior
   @ requires has(n); // precondition
   @ ensures !has(n); // postcondition @*/
This class

- Korat: Automated Testing Based on Java Predicates (ISSTA'02)
- TestEra: A Novel Framework for Automated Testing of Java Programs (ASE'01)
What specifications to use?

• Formal specifications in specifically designed languages (e.g., Z and Alloy)
  • Precise and concise
  • Hard to write (steep learning curve)

• Korat directly utilizes **Java predicates** for encoding the specifications
  • Some existing formal specifications (e.g., JML) can be automatically transformed to Java
  • **Programmers can also use the full expressiveness of the Java language to write specifications!**
public boolean repOk() {
    if (root == null) return true; // empty tree
    Set visited = new HashSet();
    visited.add(root);
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = (Node)workList.removeFirst();
        if (current.left != null) {
            if (!visited.add(current.left)) return false; // tree has no cycle
            workList.add(current.left);
        }
        if (current.right != null) {
            if (!visited.add(current.right)) return false; // tree has no cycle
            workList.add(current.right);
        }
    }
    return true; // valid non-empty tree
}
Finitization

- **Finitization**: a set of bounds that limits the size of the inputs
  - Specifies the number of objects for each used class
    - A set of objects of one class forms a **class domain**
  - Specifies the set of classes whose objects each field can point to
    - The set of values a field can take forms its **field domain**
    - Note that a field domain is a union of some class domains

```java
public static Finitization finBinaryTree(int NUM_Node) {
    Finitization f = new Finitization(BinaryTree.class);
    ObjSet nodes = f.createObject("Node", NUM_Node);
    nodes.add(null);
    f.set("root", nodes);
    f.set("Node.left", nodes);
    f.set("Node.right", nodes);
    return f;
}
```

Generated finitization description for BinaryTree
### State space

- **finBinaryTree** with **NUM_Node=3**

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

- Each field with type Node includes 4 possible choices:
  - \{null, N0, N1, N2\}

- Total number of possible tests for a tree with 3 nodes:
  - \(4 \times (4 \times 4)^3 = 2^{14} = 16,384\)

- Total number of possible tests for a tree with \(n\) nodes:
  - \((n+1) \times ((n+1) \times (n+1))^n = (n+1)^{2n+1}\)
State space: more examples

• The number of “trees” explodes rapidly!
  • n=3: over 16,000 “tests”
  • n=4: over 1,900,000 “tests”
  • n=5: over 360,000,000 “trees”

• Limit us to only very small input sizes

Are they all valid tests?
State space: examples

• finBinaryTree with NUM_Node=3

```text
BinaryTree
root
N0
left
N1
right
N2
null
null
null
null
null

BinaryTree
root
N0
left
N1
right
N1
null
null
null
null
null

N2
left
null
right
null
null
null
null
null
null
```
State space: vector representation

• To systematically explore the state space, Korat orders all the elements in every class domain and every field domain
  • The ordering in each field domain is consistent with the orderings in the class domains
  • Each candidate input is then a vector of field domain indices!

BinaryTree

N0

root

N0

left

N1

right

N2

left

null

right

null

left

null

right

null

left

N1

right

N2

Test: [ 1, 2, 3, 0, 0, 0, 0 ]

Class domain: [N0, N1, N2]
Field domain: [null, N0, N1, N2]
Search

- The search starts with the candidate vector set to all zeros
- Then, iterate through the following steps:
  - Construct the actual test based on the current vector
  - Invoke `repOK()` to check the test validity and record accessed field ordering
  - Increment the field domain index for the last field in the **recorded field ordering**
    - If the index exceeds the limit, reset it to 0 and increment the previous field in field ordering

<table>
<thead>
<tr>
<th>BinaryTree 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>N0</td>
<td>N1</td>
<td>N1</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

Current: [ 1, 2, 2, ?, ?, ?, ? ]

Next: [ 1, 2, 3, ?, ?, ?, ? ]
Search: why field ordering accessed matters

<table>
<thead>
<tr>
<th>BinaryTree</th>
<th>N0</th>
<th>N0</th>
<th>N1</th>
<th>N1</th>
<th>N1</th>
<th>N2</th>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N0</td>
<td></td>
<td></td>
<td>N1</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>N1</td>
<td>2</td>
<td>2</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>N2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Any test vectors of the form [1,2,2,?,?,?,?] are invalid!
- Keeping the accessed field ordering enables us to prune all such tests
  - $4^4$ tests pruned for this single step!
• Only the root is accessed since it is null
• Any test vectors of the form [0,?,?,?,?,?,?,?] do not need to be repeated!
• Keeping the accessed field ordering enables us to prune all such tests
  • 25% of all tests pruned by this single test input!
Can we further prune the state space?
Isomorphism

- \( \mathbf{O} \): \( O_1 \cup ... \cup O_n \), the sets of objects from \( n \) classes
- \( \mathbf{P} \): the set consisting of \text{null} and all values of primitive types that objects in \( \mathbf{O} \) can reach

- Two candidates, \( \mathbf{C} \) and \( \mathbf{C}' \), are \textit{isomorphic} iff there is a permutation \( \pi \) on \( \mathbf{O} \), mapping objects from \( O_i \) to objects from \( O_i \) for all \( 1 \leq i \leq n \), such that: \( \forall o, o' \in \mathbf{O}. \forall f \in \text{fields}(o). \forall p \in \mathbf{P}. \)
  - \( o.f==o' \) in \( \mathbf{C} \) iff \( \pi(o).f==\pi(o') \) in \( \mathbf{C}' \) AND
  - \( o.f==p \) in \( \mathbf{C} \) iff \( \pi(o).f==p \) in \( \mathbf{C}' \)

\textbf{Two data structures are isomorphic if a permutation exists between the two that preserves structure}
Isomorphism: examples

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

Test1: [ 1, 2, 3, 0, 0, 0, 0 ]

Test2: [ 1, 3, 2, 0, 0, 0, 0 ]

Test3: [ 2, 0, 0, 1, 3, 0, 0 ]

They are isomorphic!

We just need one of them...
Nonisomorphism

- **Algorithm**: only allow an index into a given class domain to exceed previous indices into that domain by 1
  - Initial prior index: -1
- **Example**: assume we are generating tests with three fields from the same class domain
Nonisomorphism: more examples

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

Test1: \[ [1, 2, 3, 0, 0, 0, 0] \]

Test2: \[ [1, \text{X}, 2, 0, 0, 0, 0] \]

Test3: \[ [\text{X}, 0, 0, 1, 3, 0, 0] \]
Korat results for BinaryTree with up to 3 nodes

- Only 9 valid tests out of $2^{14}$ possibilities!
Test generation

• Valid test cases for a method must satisfy its precondition

• Korat uses a class that represents method’s inputs:
  • One field for each parameter of the method (including the implicit this)
  • A repOk predicate that uses the precondition to check the validity of method’s inputs

• Given a finitization, Korat then generates all inputs with repOk=true

```java
class BinaryTree_remove {
    BinaryTree This; // the implicit "this"
    BinaryTree.Node n; // the Node parameter
    //@ invariant repOk();
    public boolean repOk() {
        return This.has(n);
    }
}

public static Finitization
    finBinaryTree_remove(int NUM_Node) {
    Finitization f = new Finitization(BinaryTree_remove.class);
    Finitization g = BinaryTree.finBinaryTree(NUM_Node);
    f.includeFinitization(g);
    f.set("This", g.getObjects(BinaryTree.class));
    f.set("n", /***/);
    return f;
}
```

Test generation for remove(Node n)
Test oracle

- To check partial correctness of a method, a simple test oracle could just invoke `repOk` in the post-state to check the class invariant.
- The current Korat implementation uses the JML tool-set to automatically generate test oracles from method postconditions.

```java
//@ public invariant repOk(); //class invariant
/*@ public normal_behavior
    @ requires has(n); // precondition
    @ ensures !has(n); // postcondition @*
public void remove(Node n) {
    ...
}
```

JML specification for removing from binary tree

<table>
<thead>
<tr>
<th>Testing activity</th>
<th>JUnit</th>
<th>JML+JUnit</th>
<th>Korat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generating tests</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Generating oracle</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Running tests</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Benchmark subjects

<table>
<thead>
<tr>
<th>benchmark</th>
<th>package</th>
<th>finitization parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>korat.examples</td>
<td>NUM_Node</td>
</tr>
<tr>
<td>HeapArray</td>
<td>korat.examples</td>
<td>MAX_size, MAX_length, MAX_elem</td>
</tr>
<tr>
<td>LinkedList</td>
<td>java.util</td>
<td>MIN_size, MAX_size, NUM_Entry, NUM_Object</td>
</tr>
<tr>
<td>TreeMap</td>
<td>java.util</td>
<td>MIN_size, NUM_Entry, MAX_key, MAX_value</td>
</tr>
<tr>
<td>HashSet</td>
<td>java.util</td>
<td>MAX_capacity, MAX_count, MAX_hash, loadFactor</td>
</tr>
<tr>
<td>AVTree</td>
<td>ins.namespace</td>
<td>NUM_AVPair, MAX_child, NUM_String</td>
</tr>
</tbody>
</table>
Overall 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>BinaryTree</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>HeapArray</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>LinkedList</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>TreeMap</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>
<tr>
<td>HashSet</td>
<td>7</td>
<td>3.71</td>
<td>2386</td>
<td>193200</td>
<td>$2^{119}$</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16.68</td>
<td>9355</td>
<td>908568</td>
<td>$2^{142}$</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>56.71</td>
<td>26687</td>
<td>3004597</td>
<td>$2^{166}$</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>208.86</td>
<td>79451</td>
<td>10029045</td>
<td>$2^{190}$</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>926.71</td>
<td>277387</td>
<td>39075006</td>
<td>$2^{215}$</td>
</tr>
<tr>
<td>AVTree</td>
<td>5</td>
<td>62.05</td>
<td>598358</td>
<td>1330628</td>
<td>$2^{50}$</td>
</tr>
</tbody>
</table>
Korat vs. TestEra

Table 4: Performance comparison. For each benchmark, per-

<table>
<thead>
<tr>
<th>benchmark</th>
<th>size</th>
<th>Korat</th>
<th>Alloy Analyzer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>struc. gen.</td>
<td>total time</td>
</tr>
<tr>
<td>BinaryTree</td>
<td>3</td>
<td>5</td>
<td>0.56</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>14</td>
<td>0.58</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>42</td>
<td>0.69</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>132</td>
<td>0.79</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>429</td>
<td>0.97</td>
</tr>
<tr>
<td>HeapArray</td>
<td>3</td>
<td>66</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>320</td>
<td>0.57</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1919</td>
<td>0.73</td>
</tr>
<tr>
<td>LinkedList</td>
<td>3</td>
<td>5</td>
<td>0.58</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>15</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>52</td>
<td>0.57</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>203</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>877</td>
<td>0.87</td>
</tr>
<tr>
<td>TreeMap</td>
<td>4</td>
<td>8</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>14</td>
<td>0.87</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>20</td>
<td>1.49</td>
</tr>
<tr>
<td>AVTree</td>
<td>2</td>
<td>2</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>84</td>
<td>0.65</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5923</td>
<td>1.41</td>
</tr>
</tbody>
</table>
Korat for methods

<table>
<thead>
<tr>
<th>benchmark</th>
<th>method</th>
<th>max. size</th>
<th>test cases generated</th>
<th>gen. time</th>
<th>test time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>remove</td>
<td>3</td>
<td>15</td>
<td>0.64</td>
<td>0.73</td>
</tr>
<tr>
<td>HeapArray</td>
<td>extractMax</td>
<td>6</td>
<td>13139</td>
<td>0.87</td>
<td>1.39</td>
</tr>
<tr>
<td>LinkedList</td>
<td>reverse</td>
<td>2</td>
<td>8</td>
<td>0.67</td>
<td>0.76</td>
</tr>
<tr>
<td>TreeMap</td>
<td>put</td>
<td>8</td>
<td>19912</td>
<td>136.19</td>
<td>2.70</td>
</tr>
<tr>
<td>HashSet</td>
<td>add</td>
<td>7</td>
<td>13106</td>
<td>3.90</td>
<td>1.72</td>
</tr>
<tr>
<td>AVTree</td>
<td>lookup</td>
<td>4</td>
<td>27734</td>
<td>4.33</td>
<td>14.63</td>
</tr>
</tbody>
</table>
Discussion

• Strengths
• Limitations
• Future work
This class

• Korat: Automated Testing Based on Java Predicates (ISSTA'02)
• TestEra: A Novel Framework for Automated Testing of Java Programs (ASE'01)
TestEra vs Korat

• Similarities:
  • Both target structurally complex test input generation based on specifications
  • Both automatically generate all non-isomorphic tests within a given input size

• Differences
  • TestEra uses Alloy\(^1\) as the specification language
    • Alloy is a simple declarative language based on first-order logic
  • TestEra uses Alloy and Alloy Analyzer to generate the tests and to evaluate the correctness criteria
  • TestEra produces concrete Java inputs as counterexamples to violated correctness criteria

\(^1\) https://www.csail.mit.edu/research/alloy
TestEra components

- A specification of inputs to a Java program written in Alloy
  - Class invariant and precondition
- A correctness criterion written in Alloy
  - Class invariant and post-condition
- An concretization function
  - Which maps instances of Alloy specifications to concrete Java objects
- An abstraction function
  - Which maps the concrete Java objects to instances of Alloy specifications
TestEra big picture
TestEra: example

- A recursive method for performing merge sort on acyclic singly linked lists

```java
class List {
    int elem;
    List next;
    static List mergeSort(List l) { ... }
}
```

```alloy
module list
import integer
sig List {
    elem: Integer,
    next: lone List
}
```

- Signature declaration introduces the List type with functions:
  - `elem: List → Integer`
  - `next: List → List`
Input specification

module list
import integer

sig List {
  elem: Integer,
  next: lone List }

fun Acyclic(l: List) {
  all n: l.*next | lone n.~next // at most one parent
  no l.~next } // head has no parent

one sig Input in List {}

fact GenerateInputs {
  Acyclic(Input) }

• ~: transpose (converse relation)
• *: reflexive transitive closure
• Subsignature Input is a subset of List and it has exactly one atom which is indicated by the keyword one
Correctness specification

fun Sorted(l: List) {
    all n: l.*next | some n.next => n.elem <= n.next.elem }  //?

fun Perm(l1: List, l2:List)
    all e: Integer | #(e.^elem & l1.*next) =
        #(e.^elem & l2.*next)  //?

fun MergeSortOK(i:List, o:List) {
    Acyclic(o)
    Sorted(o)
    Perm(i, o) }

one sig Output in List {}

fact OutputOK {
    MergeSortOk(Input, Output) }
Counter-examples

• If an error is inserted in the method for merging where \((l_1.\text{elem} \leq l_2.\text{elem})\) is changed to \((l_1.\text{elem} \geq l_2.\text{elem})\)

• Then TestEra generates a counter example:

Counterexample found:
Input List: 1 -> 1 -> 3 -> 2
Output List: 3 -> 2 -> 1 -> 1
TestEra: case studies

• Red-Black trees
  • Tested the implementation of Red-Black trees in java.util.TreeMap
  • Introduced some bugs and showed that they can catch them with TestEra framework

• Intentional Naming System
  • A naming architecture for resource discovery and service location in dynamic networks
  • Found some bugs

• Alloy Analyzer
  • Found some bugs in the Alloy Analyzer using TestEra framework
Discussion

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