Korat: Automated Testing Based on Java Predicates

Jeff Stuckman
Korat

• Korat is a specification-based automated testing tool ideal for testing data structures
• Korat generates synthetic test data based on the method preconditions and assertions embedded into the data structure
• Korat then runs the test cases and compares the results against the postconditions
Korat

• Korat uses optimizations to speed up the generation of test data
  – Isomorphic data structures are not generated
  – Data structures smaller than a given size are not generated
  – Data structures which are invalid or which violate the preconditions are predicted and skipped

• Korat can take advantage of existing JML annotations

• Korat uses Java *predicates* to define correctness
JML

• Specification language to annotate Java programs
• Method specifications
  – Preconditions
  – Postconditions
  – Member variables modified
  – Assertions
• Class specifications
  – Class invariants
  – Inheritance of specifications
public class BankingExample {

    public static final int MAX_BALANCE = 1000;
    private int balance;
    private boolean isLocked = false;

    //@ invariant balance >= 0 && balance <= MAX_BALANCE;

    //@ assignable balance;
    //@ ensures balance == 0;
    public BankingExample() { ... }

    //@ requires amount > 0;
    //@ ensures balance = old(balance) + amount;
    //@ assignable balance;
    public void credit(int amount) { ... }

    ...

}

Generating synthetic test data

• Valid test data is searched by using a state space search with backtracking.
• Search space of data structures may be very large, but many structures may not be valid.
• A *finitization* is used to limit the size of the data structures.
• The predicate is instrumented to predict invalid test inputs.
• Isomorphic test cases are pruned.
Finitization

- The size of inputs must be limited to make the state space finite.
- A finitization supplies the legal values for each attribute in a data structure.
- A finitization will specify the set of objects that can appear in a data structure, along with their attributes.
class BinaryTree {
    private Node root; // root node
    private int size; // number of nodes in the tree
    static class Node {
        private Node left; // left child
        private Node right; // right child
    }
    ...
    public static Finitization finBinaryTree(int NUM_Node) {
        Finitization f = new Finitization(BinaryTree.class);
        ObjSet nodes = f.createObjects("Node", NUM_Node);
        // #Node = NUM_Node
        nodes.add(null);
        f.set("root", nodes); // root in null + Node
        f.set("size", NUM_Node); // size = NUM_Node
        f.set("Node.left", nodes); // Node.left in null + Node
        f.set("Node.right", nodes); // Node.right in null+ Node
        return f;
    }
}
public class HeapArray {
    private int size; // number of elements in the heap
    private Comparable[] array; // heap elements
...

    public static Finitization finHeapArray(int MAX_size,
                                            int MAX_length,
                                            int MAX_elem) {
        Finitization f = new Finitization(HeapArray.class);
        // size in [0..MAX_size]
        f.set("size", new IntSet(0, MAX_size));
        f.set("array",
            // array.length in [0..MAX_length]
            new IntSet(0, MAX_length),
            // array[] in null + Integer([0..MAX_elem])
            new IntegerSet(0, MAX_elem).add(null));
        return f;
    }
}
State space

- **Class domain** – a set of objects from one class (represented by the finitization)
- **Field domain** – the legal values for a field (represented by the union of class domains)
- **Candidate vector** – a vector of field domain indices for the root object of a data structure and every object in every class domain
State space example

Figure 11: Candidate input that is a valid BinaryTree.

Figure 12: Candidate input that is not a valid BinaryTree.
Searching for valid inputs

• Checking every possibility would take too long
• The order of field accesses is monitored to prune the state space
Searching for valid inputs

```java
public boolean repOk() {
    // checks that empty tree has size zero
    if (root == null) return size == 0;

    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) {
            // checks that tree has no cycle
            if (!visited.add(current.left)) return false;
            workList.add(current.left);
        }
        if (current.right != null) {
            // checks that tree has no cycle
            if (!visited.add(current.right)) return false;
        }
    }
    return true;
}
```

Figure 12: Candidate input that is not a valid BinaryTree.
Searching for valid inputs

• Process is similar to depth-first-search
• Optimization depends on order that repOK() accesses fields
• Efficient repOK() functions will be analyzed more efficiently.
Nonisomorphism

• Testing multiple isomorphic data structures is not beneficial.

• Two data structures are *isomorphic* if a permutation exists between the two that preserves structure.

• If multiple isomorphic data structures are possible, the data with the lexicographically smallest candidate vector is accepted.
Nonisomorphism

- Algorithm: Only allow an index into a given class domain to exceed previous indices into that domain by 1.
  - 1 2 3
  - 1 2 2
  - 1 1 3
  - 2 3 1
  - 2 3 1
  - 3 1 2
  - 3 2 1
Nonisomorphism

• 1 1 1
• 1 1 2
• 1 1 3
• 1 2 1
• 1 2 2
• 1 2 3
• 1 3 1
• 1 3 2
• 1 3 3
Instrumentation

• Java code is instrumented so field accesses can be checked when verifying preconditions.
• Instrumentation is added to each object that will become part of the synthetic test data.
• Source code is modified before compilation.
• Fields are converted to properties.
Testing methods

• Methods being tested contain an implicit *this* parameter to represent the object being invoked.

• Method parameters are combined into one helper class, to allow for interdependency between parameters

• Methods may have multiple behaviors with their own preconditions.
Testing methods

- Korat generates test cases such that the class invariant and one of the preconditions’ behaviors is satisfied.
- Method postconditions are checked after running the test case.
- The class invariant on the implicit `this` parameter is also checked.
- Korat uses the JML toolkit to translate JML constructs to Java predicates.
Structures verified

<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>$2^{58}$</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>117652</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>$2^{20}$</td>
</tr>
<tr>
<td>LinkedList</td>
<td>8</td>
<td>1.32</td>
<td>4140</td>
<td>5435</td>
<td>$2^{20}$</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>$2^{102}$</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>1952200</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>

Table 3: Korat’s performance on several benchmarks. All finitization parameters are set to the size value. Time is the elapsed real time in seconds for the entire generation. State size is rounded to the nearest smaller exponent of two.
Test performance

<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>AVTTree</td>
<td>lookup</td>
<td>4</td>
<td>27734</td>
<td>4.33</td>
<td>14.63</td>
</tr>
</tbody>
</table>
The small-scope hypothesis

“The ‘small scope hypothesis’ argues that a high proportion of bugs can be found by testing the program for all test inputs within some small scope. In object-oriented programs, a test input is constructed from objects of different classes; a test input is within a scope of $s$ if at most $s$ objects of any given class appear in it. If the hypothesis holds, it follows that it is more effective to do systematic testing within a small scope than to generate fewer test inputs of a larger scope.”

The small-scope hypothesis

- Experiments have shown that “exhaustive testing” in a small scope can achieve near-complete statement and branch coverage.

Related work

• Previous tools have performed automated testing based on Z specifications, but they did not construct complex data structures.
• Alloy Analyzer (also from MIT) was similar to Korat, but it was slow and required the use of a special modeling language.
• JML+Junit is another automated testing tool that reads JML annotations. It requires the developer to supply possible values for each parameter.
The Z specification language

```
Array[X]
array : seq X
max_size : \mathbb{N}
bot, top : \mathbb{N}
size : \mathbb{N}

bot \in 1 .. max_size
\top \in 1 .. max_size
size \in 0 .. max_size
#array = max_size
size mod max_size = (top - bot + 1) mod max_size
```

```
ArrayIn_0[X]
UpdateArray[X]
x? : X

size < max_size
size' = size + 1
bot' = bot
\top' = (top mod max_size) + 1
array' = array \oplus \{top' \mapsto x\}
```
ESC/Java2

• Static checker for JML specification violations and common programming errors
• JML specifications can improve the detection of common programming errors

Bag.java:21: Warning: Possible negative array index (IndexNegative)
    a[mindex] = a[n];
    ^

....

12:     //@ requires n >= 1;
13:     int extractMin() {

Java Pathfinder

• Unique combination of testing and static analysis
• Runs methods with generated inputs and tries to violate assertions
• Tries various thread schedules to uncover deadlocks
• Developed and used by NASA
KeY

• A static analysis tool using JML
• Any JML specification becomes a “proof obligation”
• KeY attempts to prove the correctness of proof obligations and supplies proofs if successful
• Will not work in many situations