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

JML Example

```
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) { ... }

...

}
```

Source:Wikipedia

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.

Data structure example

```
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;
}
```

Data structure example

```
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

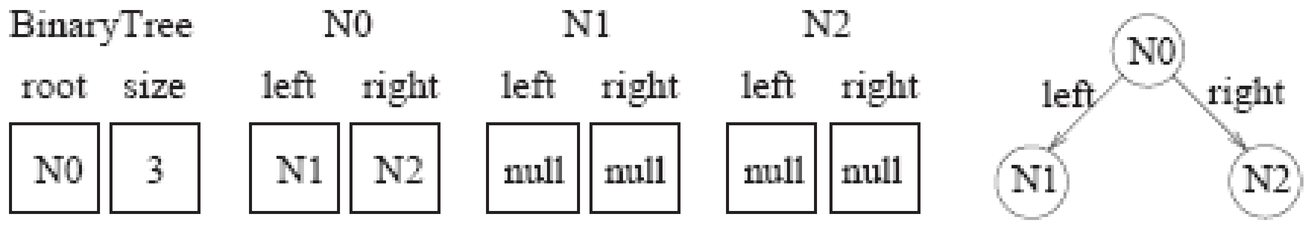


Figure 11: Candidate input that is a valid `BinaryTree`.

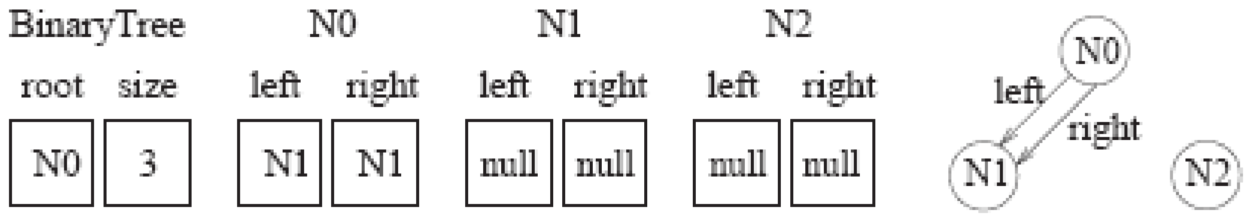


Figure 12: Candidate input that is not a valid `BinaryTree`.

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

```
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;
```

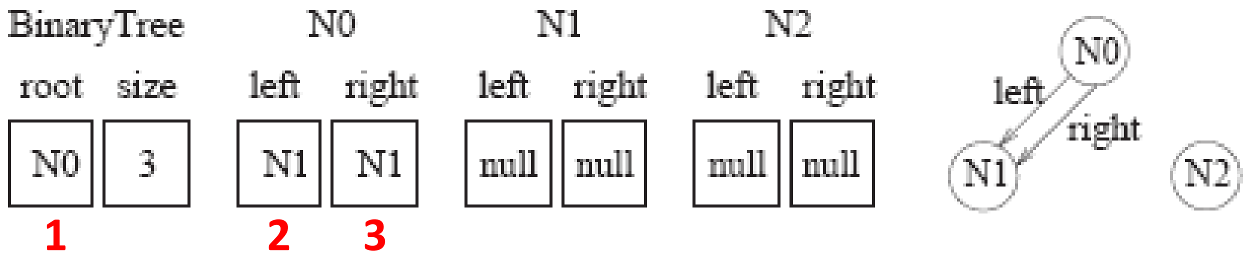


Figure 12: Candidate input that is not a valid BinaryTree.

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 $\cancel{3}$ 2
- $\cancel{2}$ 1 3
- $\cancel{2}$ 3 1
- $\cancel{3}$ 1 2
- $\cancel{3}$ 2 1

Nonisomorphism

- 1 1 1
- 1 1 2
- 1 1 $\xcancel{3}$
- 1 2 1
- 1 2 2
- 1 2 3
- 1 $\xcancel{3}$ 1
- 1 $\xcancel{3}$ 2
- 1 $\xcancel{3}$ 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.

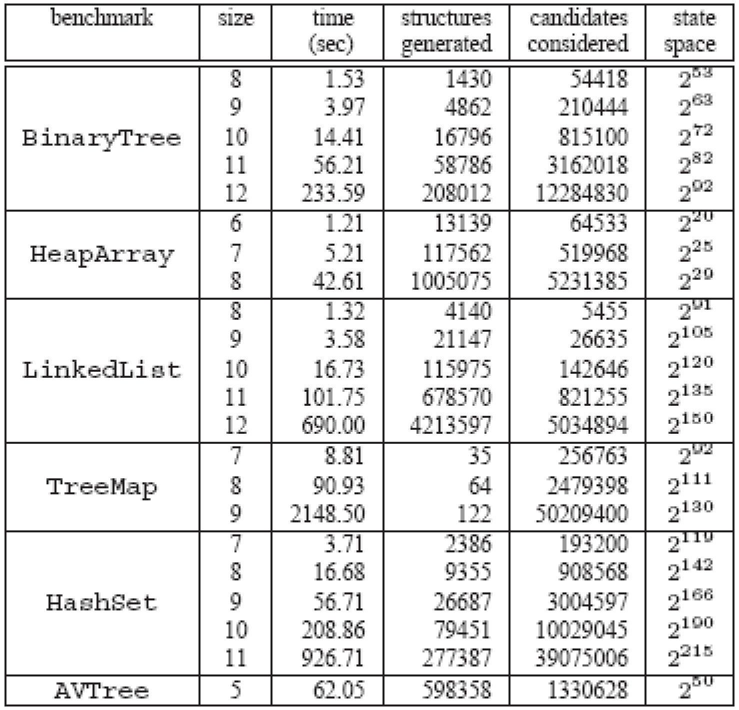
Testing methods

- 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.

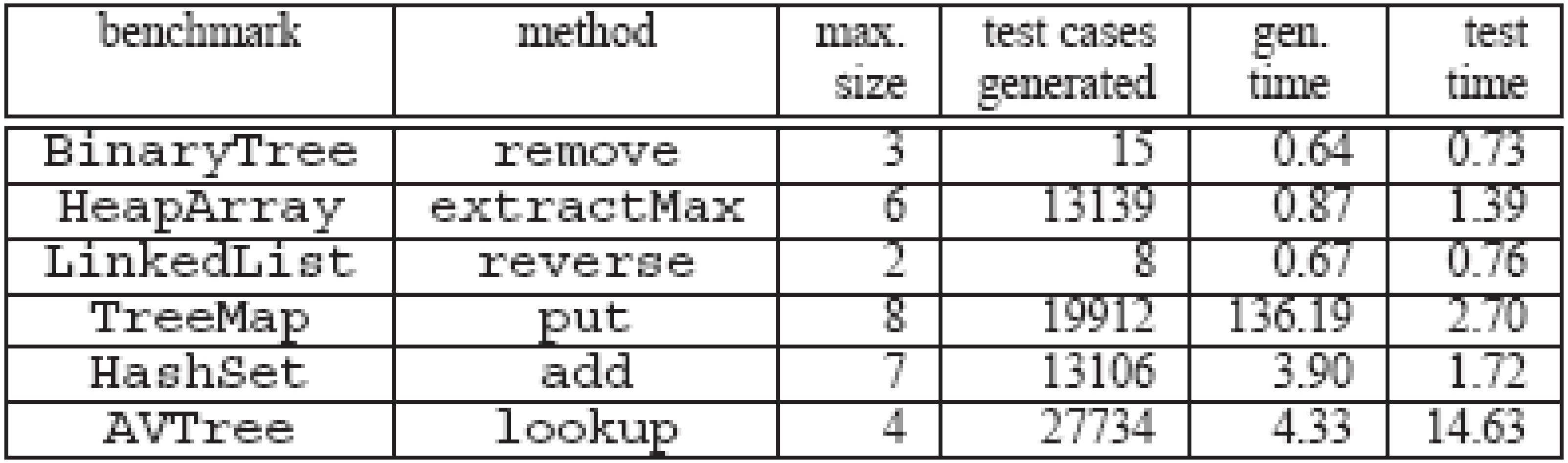
Structures verified



benchmark	size	time (sec)	structures generated	candidates considered	state space
`BinaryTree`	8	1.53	1430	54418	2^{53}
	9	3.97	4862	210444	2^{63}
	10	14.41	16796	815100	2^{72}
	11	56.21	58786	3162018	2^{82}
	12	233.59	208012	12284830	2^{92}
`HeapArray`	6	1.21	13139	64533	2^{20}
	7	5.21	117562	519968	2^{25}
	8	42.61	1005075	5231385	2^{29}
`LinkedList`	8	1.32	4140	5455	2^{91}
	9	3.58	21147	26635	2^{105}
	10	16.73	115975	142646	2^{120}
	11	101.75	678570	821255	2^{135}
	12	690.00	4213597	5034894	2^{150}
`TreeMap`	7	8.81	35	256763	2^{92}
	8	90.93	64	2479398	2^{111}
	9	2148.50	122	50209400	2^{130}
`HashSet`	7	3.71	2386	193200	2^{119}
	8	16.68	9355	908568	2^{142}
	9	56.71	26687	3004597	2^{166}
	10	208.86	79451	10029045	2^{190}
	11	926.71	277387	39075006	2^{215}
`AVTree`	5	62.05	598358	1330628	2^{50}

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



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

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.”

Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the small scope hypothesis. submitted for publication. http://citeseer.ist.psu.edu/623993.html

The small-scope hypothesis

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

Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the small scope hypothesis. submitted for publication.
http://citeseer.ist.psu.edu/623993.html

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

$Array[X]$

$$\begin{array}{l} array : \mathrm{seq}\, X \\ max\_size : \mathbb{N} \\ bot, top : \mathbb{N} \\ size : \mathbb{N} \\ \hline bot \in 1 \mathrel{..} max\_size \\ top \in 1 \mathrel{..} max\_size \\ size \in 0 \mathrel{..} max\_size \\ \#array = max\_size \\ size \bmod max\_size = (top - bot + 1) \bmod max\_size \end{array}$$

$ArrayIn_0[X]$

$$\begin{array}{l} UpdateArray[X] \\ x? : X \\ \hline size < max\_size \\ size' = size + 1 \\ bot' = bot \\ top' = (top \bmod max\_size) + 1 \\ array' = array \oplus \{top' \mapsto x?\} \end{array}$$

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() {
```

Source: http://kind.ucd.ie/products/opensource/ESCJava2/ESCTools/docs/ESCJAVA-UsersManual.html

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