Korat
Automated Testing Based on Java Predicates

Sarfraz Khurshid, MIT
[joint work with: Chandrasekhar Boyapati and Darko Marinov]

ISSTA 2002
Rome, Italy

motivation

- test (full) conformance of Java code
  - generate test inputs automatically
  - evaluate correctness automatically
  - check exhaustively (up to *given* input size)
- discover bugs
  - generate concrete counterexamples
  - do not generate false alarms
- do not require a different specification language

Korat

- automates specification-based testing
  - uses Java Modeling Language (JML) specifications
  - generates test inputs using precondition
    - builds a Java predicate
    - uses finitization (that defines input space)
    - systematically explores input space
    - prunes input space using field accesses
    - provides isomorph-free generation
  - checks correctness using postcondition
    - JML/JUnit toolset
- generates complex structures
  - Java Collections Framework (JCF)

talk outline

- motivation
- example
- test input generation
- checking correctness
- experiments
- conclusions

binary tree

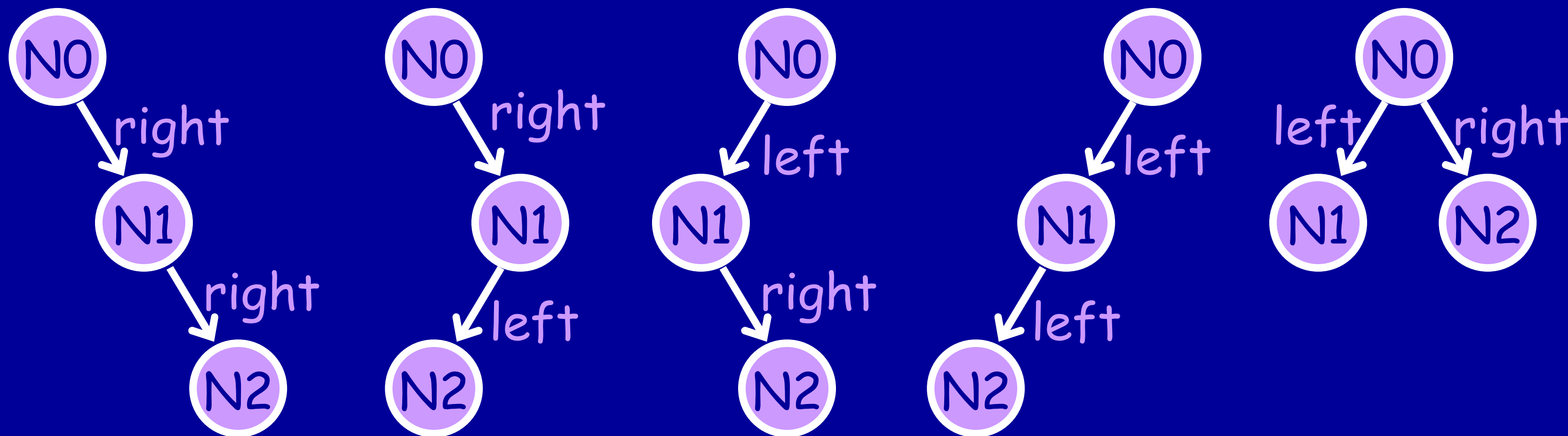
```
class BinaryTree {
  //@ invariant                    // class invariant for BinaryTree
  //@       repOk();
  Node root;
  int size;

  static class Node {
    Node left;
    Node right;
  }

  /*@ normal_behavior              // specification for remove
    @   requires has(n);           // precondition
    @   ensures !has(n);           // postcondition
    @*/
  void remove(Node n) { ... }
}
```

binary tree (class invariant)

```
boolean repOk() {
  if (root == null) return size == 0;                    // empty tree has size 0
  Set visited = new HashSet(); visited.add(root);
  List 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; // acyclicity
      workList.add(current.left);
    }
    if (current.right != null) {
      if (!visited.add(current.right)) return false; // acyclicity
      workList.add(current.right);
    }
  }
  if (visited.size() != size) return false;          // consistency of size
  return true;
}
```

binary tree (Korat's generation)

- Korat generates a finitization
- 3 nodes



- 7 nodes
  - Korat generates 429 trees in less than 1 sec
  - 2^{45} candidate structures

talk outline

- motivation
- example
- test input generation
- checking correctness
- experiments
- conclusions

test input generation

- given predicate p and finitization f, Korat generates all inputs for which p returns "true"
- finitization
- state space
- search
- nonisomorphism
- instrumentation
- generating test inputs

finitization

- set of bounds that limits the size of inputs
  - specifies number of objects for each class
- class domain
  - set of objects from a class
  - eg, for class “Node”: { N0, N1, N2 }
- field domain
  - set of values a field can take
  - union of some class domains
  - eg, for field “left”: { null, N0, N1, N2 }
- Korat automatically generates a skeleton
  - programmers can specialize/generalize it

finitization (binary tree)

- Korat generates

```
Finitization finBinaryTree(int n, int min, int max) {
  Finitization f = new Finitization(BinaryTree.class);
  ObjSet nodes = f.createObjects(“Node”, n);   // #Node = n
  nodes.add(null);
  f.set(“root”, nodes);                        // root in null + Node
  f.set(“size”, new IntSet(min, max));         // min <= size <= max
  f.set(“Node.left”, nodes);                   // Node.left in null + Node
  f.set(“Node.right”, nodes);                  // Node.right in null + Node
  return f;
}
```

- a specialization

```
Finitization finBinaryTree(int n) {
  return finBinaryTree(n, n, n);
}
```

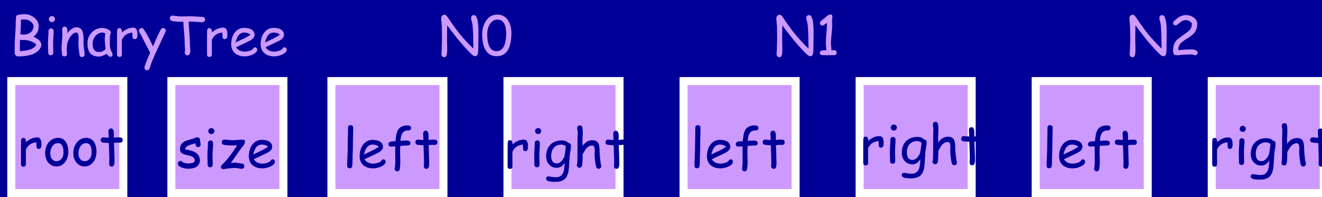
- finBinaryTree(3) generates trees with 3 nodes

state space

- given a finitization, Korat
  - allocates given number of objects
  - constructs a vector of object fields
    - fields of objects have unique indexes in the vector
    - a valuation of the vector is a candidate input
    - state space is all possible valuations

state space (binary tree)

- for finBinaryTree(3)
  - 1 BinaryTree object, 3 Node objects
  - vector has 8 fields



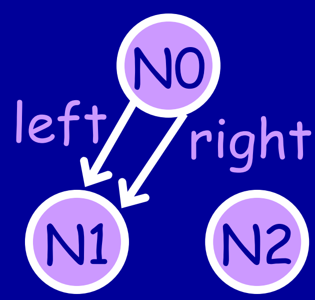
- state space has $4 * 1 * (4 * 4)^3 = 2^{14}$ candidates
- for finBinaryTree(n) state space has $(n + 1)^{2n+1}$ candidates

search

- Korat orders elements in class/field domains
- candidate is a vector of field domain indices
- for each candidate vector (initially 0), Korat
  - creates corresponding structure
  - invokes repOk and monitors the execution
  - builds field ordering, ie, list of fields ordered by time of first access
  - if repOk returns "true", outputs structure(s)
  - if repOk returns "false", backtracks on the last field accessed using field ordering

search (binary tree [1])

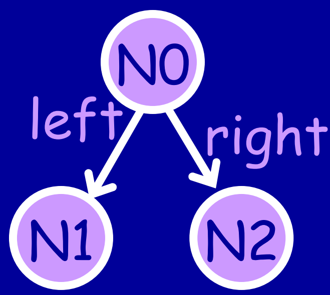
- class domain for "Node": [N0, N1, N2]
- field domain
  - "root", "left", "right": [null, N0, N1, N2]
  - "size": [3]
- candidate [N0, 3, N1, N1, null, null, null, null] encodes



  - repOk returns "false"; field ordering:
    - [0, 2, 3] or [root, N0.left, N0.right]

search (binary tree [2])

- backtracking on N0.right
  - gives next candidate
    [N0, 3, N1, N2, null, null, null, null]



  - prunes from search all 4^4 candidates of the form [N0, _, N1, N1, _, _, _, _]
  - completeness: guaranteed because repOk returned "false" without accessing the other fields

nonisomorphism

- candidates C and C' rooted at r are isomorphic
  $\exists \pi \,.\, \forall o, o' \in O_{C,r} \,.\, \forall f \in fields(o) \,.\, \forall p \in P \,.$
  o.f == o' in C <=> π(o).f == π(o') in C' and
  o.f == p in C <=> π(o).f == p in C'
- Korat generates only the lexicographically smallest candidate from each partition
  - increments field domain indices by more than 1, eg. resetting to 0 before hitting max
- optimality: Korat generates exactly one structure from each partition

instrumentation

- to monitor repOk's executions and build field ordering, Korat
  - uses observer pattern
  - performs source to source translation
  - replaces field accesses with *get* and *set* methods to notify observer
  - adds special constructors
    - initializes all objects in finitization with an observer

generating test inputs (1)

- to generate test inputs for method m, Korat
  - builds a class that represents m's inputs
  - builds repOk that checks m's precondition
  - generates all inputs i s.t. "i.repOk()"
- recall "remove" method for "BinaryTree"

```
class BinaryTree {
  //@ invariant repOk();
  ...
  //@ requires has(n);
  void remove(Node n) { ... }
}
```

```
class BinaryTree_remove {
  //@ invariant repOk();
  BinaryTree This;
  BinaryTree.Node n;
  boolean repOk() {
    return This.repOk() && This.has(n);
  }
}
```

generating test inputs (2)

- an alternative approach [JML+JUnit]
  - (manually) compute all possibilities for each parameter
  - take cross product to get space of inputs
  - filter using precondition
  - Korat improves on this by monitoring repOk executions and breaking isomorphisms

talk outline

- motivation
- example
- test input generation
- checking correctness
- experiments
- conclusions

checking correctness

- to test method m, Korat invokes m on each test input and checks each output with a test oracle
- current Korat implementation
  - uses JML toolset for generating oracles
  - JUnit for executing test and error reporting

	testing framework		
testing activity	JUnit	JML+JUnit	Korat
generating test inputs			✔
generating test oracles		✔	✔
running tests	✔	✔	✔

talk outline

- motivation
- example
- test input generation
- checking correctness
- experiments
- conclusions

performance (generation)

benchmark	size	structures generated	time (sec)	state space
BinaryTree	8	1430	2	2^{53}
	12	208012	234	2^{92}
HeapArray	6	13139	2	2^{20}
	8	1005075	43	2^{29}
java.util.LinkedList	8	4140	2	2^{91}
	12	4213597	690	2^{150}
java.util.TreeMap	7	35	9	2^{92}
	9	122	2149	2^{130}
java.util.HashSet	7	2386	4	2^{119}
	11	277387	927	2^{215}
AVTree (INS)	5	598358	63	2^{50}

performance (checking)

benchmark	method	size	test inputs generated	gen time	test time
BinaryTree	remove	3	15	1	1
HeapArray	extractMax	6	13139	1	2
LinkedList	reverse	2	8	1	1
TreeMap	put	8	19912	137	3
HashSet	add	7	13106	4	2
AVTree	lookup	4	27734	5	15

- methods checked for all inputs up to given size
- complete statement coverage achieved for these inputs

talk outline

- motivation
- example
- test input generation
- checking correctness
- experiments
- conclusions

related work

- specification-based testing
  - using Z specifications [Horcher'95]
  - using UML statecharts [Offutt & Abdurazik'99]
  - TestEra [Marinov & Khurshid'01]
  - JML+JUnit [Cheon & Leavens'01]
- static analysis
  - ESC [Detlefs et al'98]
  - TVLA [Sagiv et al'98]
  - Roles [Kuncak et al'02]
- software model checking
  - VeriSoft [Godefroid'97]
  - JPF [Visser et al'00]

conclusions

- Korat automates specification-based testing
  - uses method precondition to generate all nonisomorphic test inputs
    - prunes search space using field accesses
  - invokes the method on each input and uses method postcondition as a test oracle
- Korat prototype uses JML specifications
- Korat efficiently generates complex data structures including some from JCF

questions/comments ?

khurshid@lcs.mit.edu
http://mulsaw.lcs.mit.edu