UT ECE | THE UNIVERSITY OF TEXAS AT AUSTIN
DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

EE360T/382V Software Testing

khurshid@ece.utexas.edu

April 16, 2018

Overview

Today

- Systematic test input generation
- Non-deterministic choice
  - Java PathFinder (JPF) model checker

Next time

- Symbolic execution

Unit testing object-oriented (OO) code

Specifications for OO code have three key components:

- Class invariants (e.g., *repOk* methods)
- Method preconditions and postconditions

Testing a method requires creating its inputs – pre-state

- For instance method: receiver object and arguments

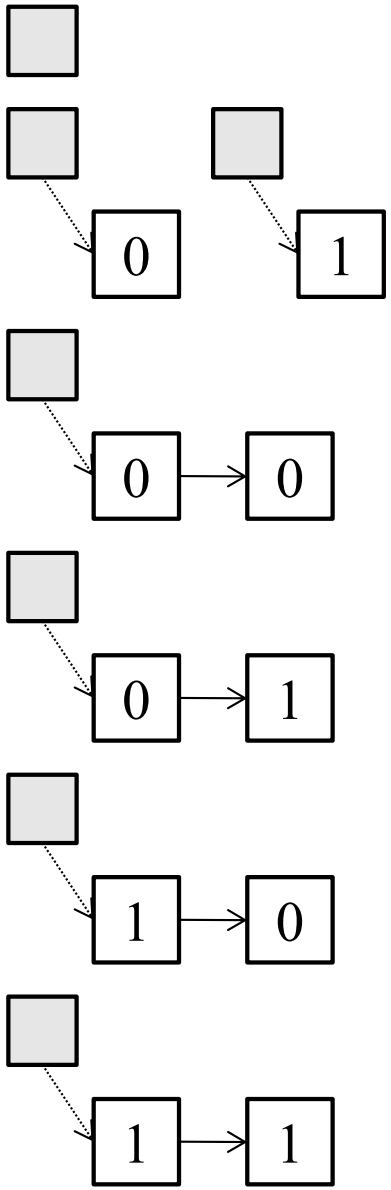
Two basic approaches to input generation

- Abstract level—use method sequences
- Representation level—allocate objects and set fields
  - Can combine the two approaches

Example: Singly-linked acyclic list

```
public class LinkedList {
    Node header;
    int size;

    static class Node {
        int elem;
        Node next;
    }
}
```



Example: Class invariant

```
boolean repOk() {
    if (header == null) return size == 0;
    Set<Node> visited = new HashSet<Node>();
    Node current = header;
    while (current != null) {
        if (!visited.add(current)) return false;
        current = current.next;
    }
    return size == visited.size();
}
```

Example: Methods

```
void add(int x) {
    // precondition: repOk()
    // postcondition: repOk() && x is added at head
    Node n = new Node();
    n.elem = x;
    n.next = header;
    header = n;
    size++;
}

void remove(int x) { ... }
```

Two JUnit tests for add

```
@Test public void abst() {
   // create receiver object state
   LinkedList l = new LinkedList();
   l.add(0);



   l.add(1); // execute method to test

   assertTrue(l.repOk()); // check output
}
```

```
@Test public void conc() {
   // create receiver object state
   LinkedList l = new LinkedList();
   Node n0 = new Node();
   l.header = n0; l.size = 1;
   n0.elem = 0; n0.next = null;


   l.add(1); // execute method to test

   assertTrue(l.repOk()); // check output
}
```

Example: abstract level generation

21 sequences using ≤2 method invocations with integers 0 and 1 (starting with an empty list)

```
ε
add(0)
add(1)
remove(0)
remove(1)
```

```
add(0); add(0)
add(0); add(1)
add(0); remove(0)
add(0); remove(1)
add(1); add(0)
add(1); add(1)
add(1); remove(0)
add(1); remove(1)
```

```
remove(0); add(0)
remove(0); add(1)
remove(0); remove(0)
remove(0); remove(1)
remove(1); add(0)
remove(1); add(1)
remove(1); remove(0)
remove(1); remove(1)
```

Background: Java PathFinder (JPF)

https://github.com/javapathfinder

Is a stateful model checker for Java programs

- Focus: multi-threaded programs

Implements its *own* Java virtual machine

Is extensible and open-source

Performs various optimizations

- E.g., state compression, partial order reduction, symmetry reduction, slicing, and abstraction

Has been applied with success to find bugs

- Real time avionics operating system
- Model of a spacecraft controller

Background: Non-deterministic choice

gov.nasa.jpf.jvm.Verify class provides methods for non-deterministic choice

- `getInt(min, max)`
- `getBoolean()`

E.g., "int x = `getInt(0, 2);`" non-deterministically initializes "x" to values 0, 1, 2

- JPF JVM executes the program for each of these three values

Example: Non-deterministic choice

```
static void m() {
  int x = Verify.getInt(0, 2);
  boolean b = Verify.getBoolean();
  System.out.println(x + ", " + b);
}
```

Running this program on JPF JVM outputs:

```
0, false
0, true
1, false
1, true
2, false
2, true
```

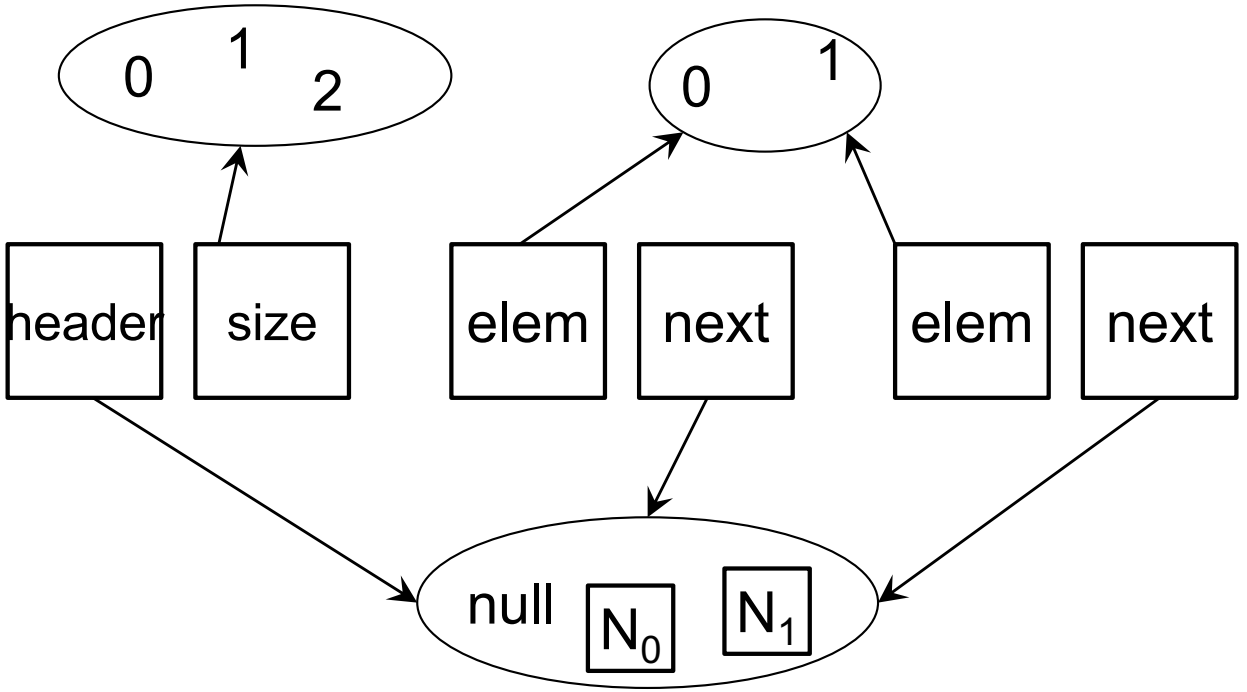
Example: Abstract level testing using JPF

```
public static void main(String[] a) {
        final int SEQ_LENGTH = Verify.getInt(0, 2);
        LinkedList l = new LinkedList();
        assert l.repOk();
        for (int i = 0; i < SEQ_LENGTH; i++) {
                if (Verify.getBoolean()) {
                        l.remove(Verify.getInt(0, 1));
                        assert l.repOk();
                } else {
                        l.add(Verify.getInt(0, 1));
                        assert l.repOk();
                }
        }
}
```

Example: Representation level generation

Consider generating lists with ≤2 nodes with 0 and 1

Any list with up to 2 nodes can be represented using a candidate vector with 6 elements



In total: $(3\times3)\times(2\times3)^2 = 324$ possible *candidate structures*

Example: Representation level generation using JPF

```
// allocate objects
LinkedList l = new LinkedList();
LinkedList.Node n0 = new LinkedList.Node();
LinkedList.Node n1 = new LinkedList.Node();

//initialize field domains
LinkedList.Node[] nodes = new LinkedList.Node[]{ null, n0, n1 };
int[] elems = new int[]{ 0, 1 };

// set fields
l.header = nodes[Verify.getInt(0, nodes.length - 1)];
l.size = Verify.getInt(0, 2);
n0.next = nodes[Verify.getInt(0, nodes.length - 1)];
n0.elem = elems[Verify.getInt(0, elems.length - 1)];
n1.next = nodes[Verify.getInt(0, nodes.length - 1)];
n1.elem = elems[Verify.getInt(0, elems.length - 1)];

// check if structure is valid using repOk as a filter
if (l.repOk()) ...
```

Problem with using *repOk* as a filter

Search must explore all possible candidates

A large number of invalid or redundant candidates are first generated and then discarded

E.g., for singly-linked list of size up to 2

- 324 candidates are generated
- 68 valid lists are generated
- but there are only 7 non-equivalent lists

Idea: Use *repOk* to prune search

Execute *repOk* to identify what makes a candidate invalid

- What fields did *repOk* access before returning false?

Backtrack on last accessed field

- Doing so prunes all candidates with the same values for the accessed fields as the invalid candidate

Example pruning with a boolean formula

Consider finding a solution to the following formula:
(a) ∧ (!b) ∧ (c)

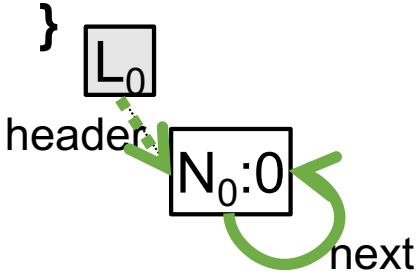
Exhaustive enumeration considers $2^3 = 8$ candidates:
000, 001, 010, 011, 100, 101, 110, 111

However, when a = 0, the formula is false for any b, c

- "Executing" the formula on 000 and "returning" false as soon as we determine the "output", we prune from search all inputs of the form 0xx
  - A pruning search explores 000, 100, 101 to find the solution

Example: Pruning with *repOk*

```
boolean repOk() {
    if (header == null) return size == 0;
    Set<Node> visited = new HashSet<Node>();
    Node current = header;
    while (current != null) {
        if (!visited.add(current)) return false;
        current = current.next;
    }
    return size == visited.size();
}
```

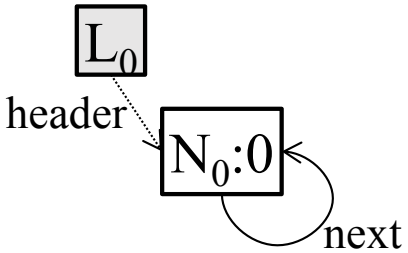


[L_0.header, N_0.next]

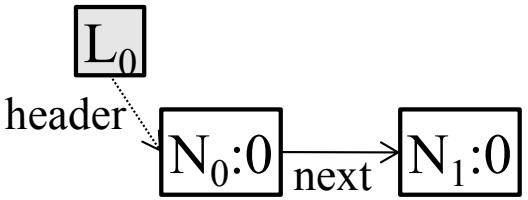
Example: Backtracking

Backtrack on last field accessed, i.e., N_0.next

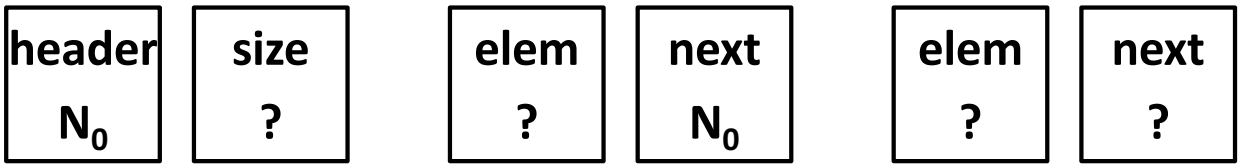
Last candidate



Next candidate



Prunes candidates of the form



Pruning using *repOk*

Pruning using *repOk* explores fewer candidates than filtering using *repOk*

- E.g., for singly-linked list of size up to 2
  - 59 candidates are generated (versus 324)
  - 13 valid lists are generated (versus 68)
  - but there are only 7 non-equivalent lists

But pruning may still explore redundant candidates

- Equivalent candidates are generated

Pruning can be optimized to remove redundant candidates from search

- E.g., optimized pruning generates 31 candidates, and exactly 7 valid lists

Implementation

Bytecode instrumentation allows monitoring field accesses

- Introduce boolean shadow fields
- Replace field accesses with method invocations to track accesses
- Define field domains
- Implement methods that make non-deterministic field assignments

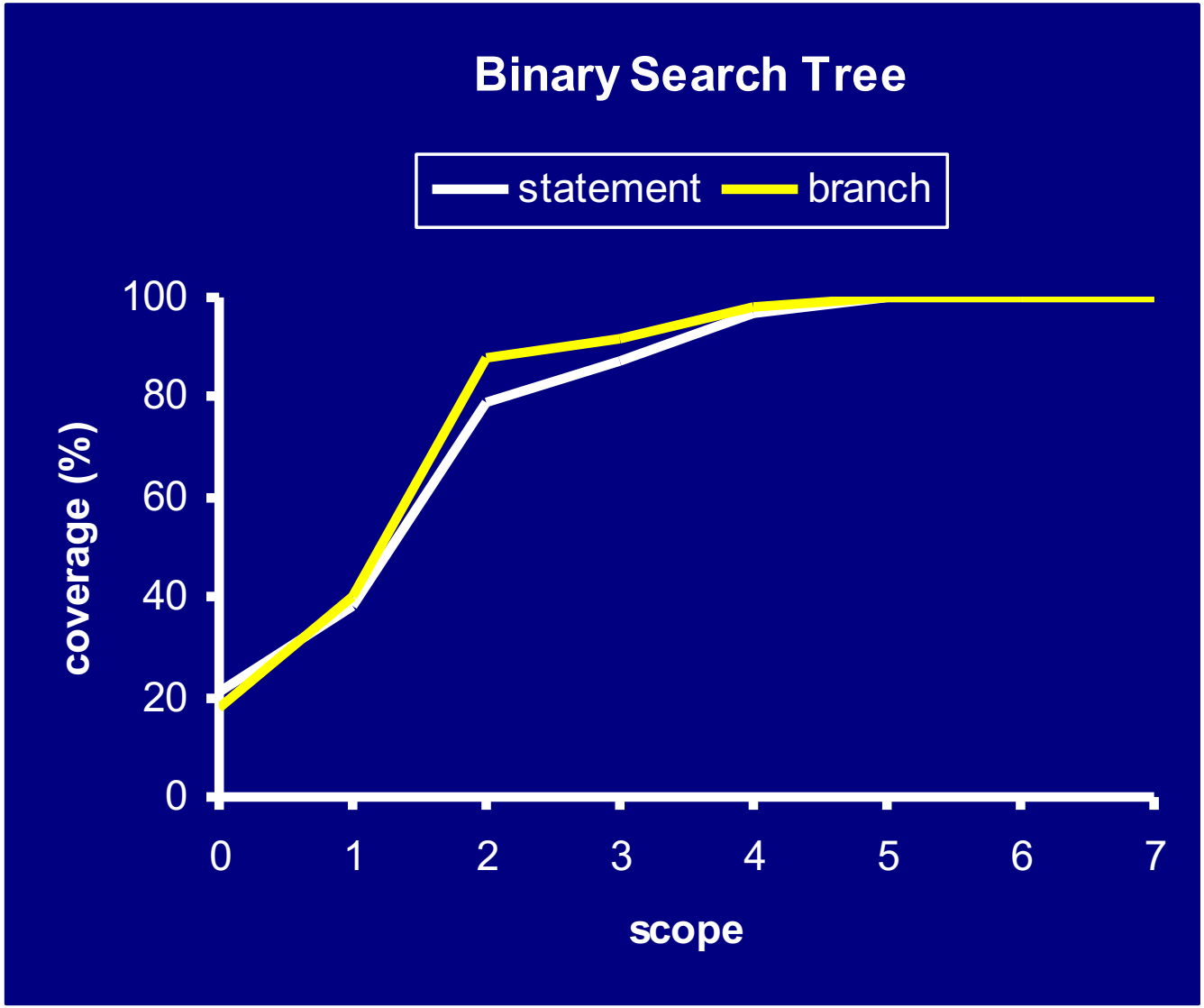
Korat tool introduced the idea of using *repOk* for pruning [ISSTA 2002]

- http://korat.sourceforge.net/

Experiments [Korat, ISSTA'02]

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}

Code coverage



?/!