Automated Test Generation

Introduction

1. Writing and maintaining tests is a tedious and error-prone process
    - Cover techniques for automated test generation
        - Specifically for unit testing
        - More directed than random testing
            - Find bugs more efficiently and create a concise test suite that can be used for regression testing

Outline

1. Previously: Random testing (fuzzing)
    - Security, mobile apps, concurrency
2. Systematic testing: Korat
    - Linked data structures (linked lists, trees)
3. Feedback-directed random test: Randoop
    - Classes, libraries

Korat

1. Korat is a deterministic test generator which originated as a research project by a team of three graduate students at MIT
    - Leverage pre-conditions and post-conditions to generate tests automatically

The Problem

1. There are infinitely many tests
    - Which finite subset should we choose
    - And even finite subsets can be huge
    - Need a subset which is:
        - Concise: Avoids illegal and redundant tests
        - Diverse: Gives good coverage

An Insight

1. Often can do a good job by systematically testing all inputs up to a small size
2. Small test case hypothesis:
    - If there is any test that causes the program to fail, there is a small such test
    - If a list function works for lists of length 0 through 3, it probably works for all lists
        - E.g., because the function is oblivious to the length

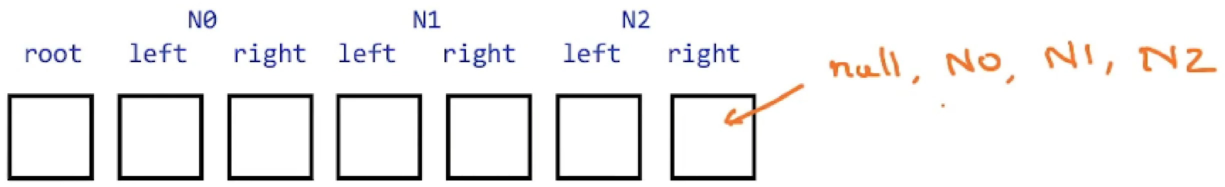
How Do We Generate Test Inputs?

1. Korat uses the types to generate test inputs
    - White-box testing method
    - The class declaration shows what values (or null) can fill each field
    - Simply enumerate all possible shapes with a fixed set of nodes

Scheme for Representing Shapes

1. Representing shapes
    - Order all possible values of each field
    - Order all fields into a vector

- Each shape == vector of field values



Representing Shapes

Representing Shapes

1. Fill in the field values in each vector to represent the adjacent shape
   - Root?
     - N0
   - N0 left?
     - N1
   - N0 right?
     - N2
   - N1 left?
     - null
   - N1 right?
     - null
   - N2 left?
     - null
   - N2 right?
     - null
2. Fill in the field values in each vector to represent the adjacent shape
   - Root?
     - N0
   - N0 left?
     - N1
   - N0 right?
     - null
   - N1 left?
     - null
   - N1 right?
     - N2
   - N2 left?
     - null
   - N2 right?
     - null

A Simple Algorithm

1. User selects a maximum input size k
2. Generate all possible inputs up to size k
3. Discard inputs where pre-condition is false
4. Run program on remaining inputs
5. Check results using post-condition

Enumerating Shapes

1. Korat represents each input shape as a vector of the following form:
    - Root, N0 left, N0 right, N1 left, N1 right, N2 left, N2 right
2. What is the total number of vectors of the above form?
    - Each field can be null, N0, N1, or N2
    - 7 fields, so 4 ˆ 7 = 16384

The General Case for Binary Trees

1. How many binary trees are there of size <= k?
    - A BinaryTree object, bt
    - k Node objects, n0, n1, n2, n3, . . .
    - 2k + 1 Node points
        - root (for bt)
        - left, right (for each Node object)
    - k + 1 possible values (n0, n1, n2, . . . or null) per pointer
    - (k + 1) ˆ (2k + 1)

A Lot of Trees!

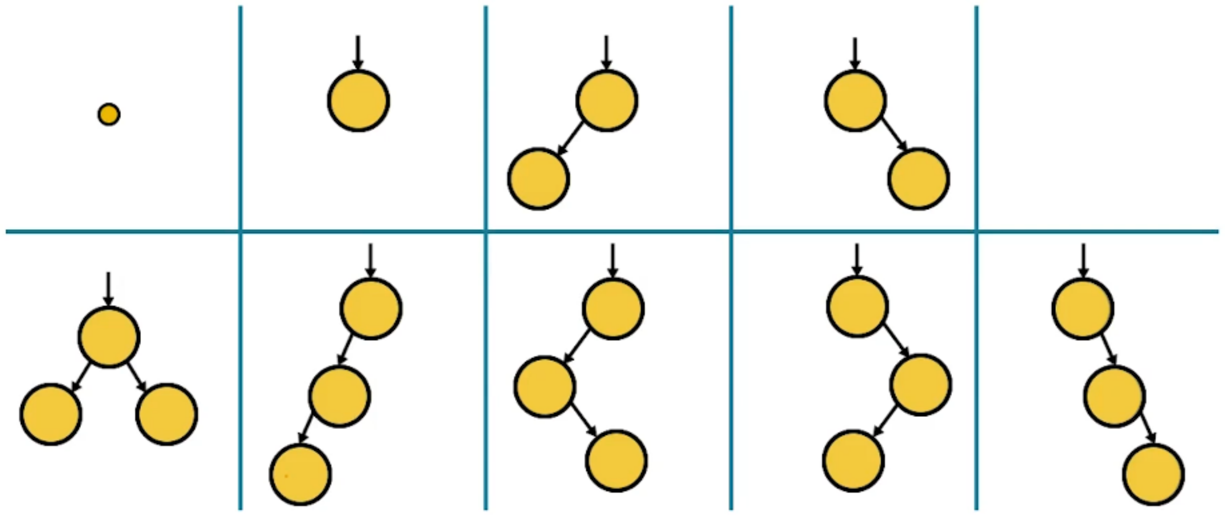
1. The number of trees explodes rapidly
    - k = 3: Over 16000 trees
    - k = 4: Over 1900000 trees
    - k = 5: Over 360000000 trees

An Overestimate

1. (k + 1) ˆ (2k + 1) is a gross overestimate
    - Many of the shapes are not even trees
        - Cycles, dangling nodes
    - Many are isomorphic (same shape)

How Many Trees?

1. Only 9 distinct binary trees with at most 3 nodes
    - How do we avoid generating illegal inputs?
    - How do we avoid generating redundant inputs?



Nine Distinct Trees

Another Insight

1. Avoid generating inputs that don't satisfy the pre-condition in the first place
    - Use the pre-condition to guide the generation of tests

The Technique

1. Instrument the pre-condition
    - Add code to observe its actions
    - Record fields accessed by the pre-conditoin
2. Observation:
    - If the pre-condition doesn't access a field, then pre-condition doesn't depend on the field

The Pre-Condition for Binary Trees 1

1. Root may be null
    - If root is not null:
        - No cycles
        - Each node (except root) has one parent
        - Root has no parent

The Pre-Condition for Binary Trees 2

```
public boolean repOK(BinaryTree bt) {
    if (bt.root == null) return true;
    Set visited = new HashSet();
    List workList = new LinkedList();
    visited.add(bt.root);
    workList.add(bt.root);
    while (!workList.isEmpty()) {
        Node current = workList.removeFirst();
        if (current.left != null) {
            if (!visited.add(current.left)) return false;
            workList.add(current.left);
        }
    }
    // similarly for current.right
    return true;
}
```

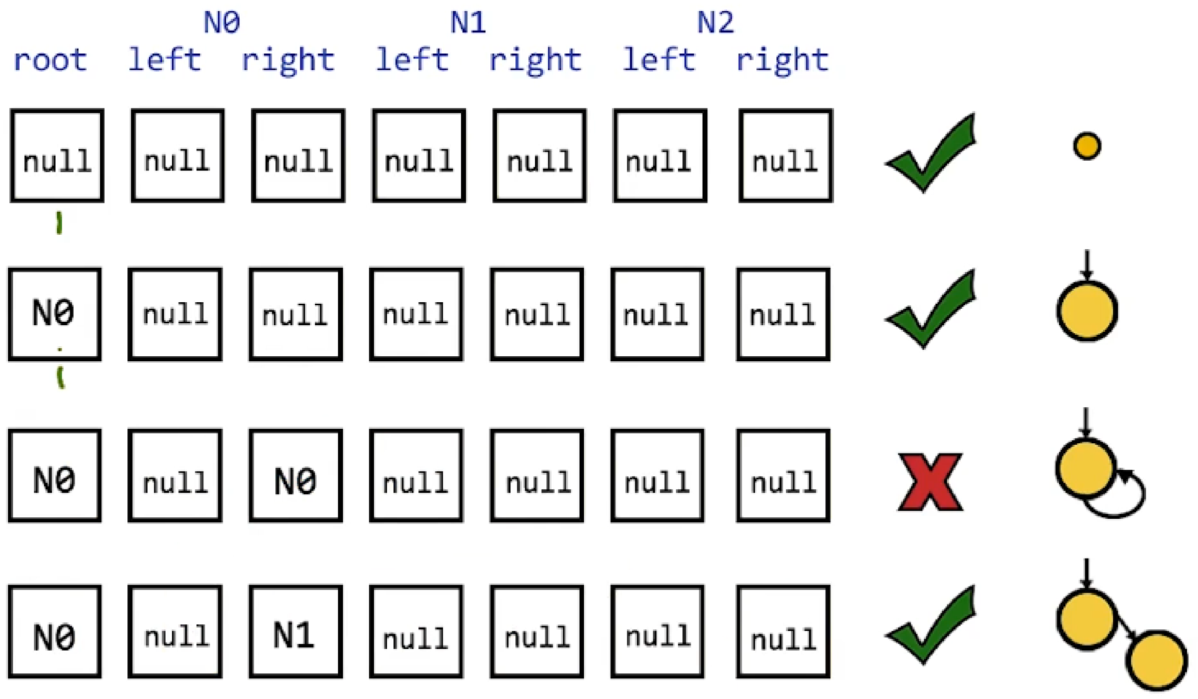
Example: Using the Pre-Condition

1. Assume the root of the tree is null
   - The pre-condition accesses only the root as it is null
     - Every possible shape for the other nodes would yield the same result
     - This single input eliminates 25% of the tests

Enumerating Tests

1. Shapes are enumerated by their associated vectors
   - Initial candidate vector: All fields null
   - Next shape generated by:
     - Expanding last field accessed in pre-condition
     - Backtracking if all possibilities for a field are exhausted
   - Key idea: Never expand parts of input not examined by pre-condition
   - Also: Cleverly checks for and discards shapes isomorphic to previously- generated shapes

Example: Enumerating Binary Trees



Enumerating Binary Trees

Enumerating Binary Trees 1

1. Assume the following shape of the tree:

- Root: N0
- N0 left: null
- N0 right: N1
- N1 left: null
- N1 right: null
- N2 left: null
- N2 right: null

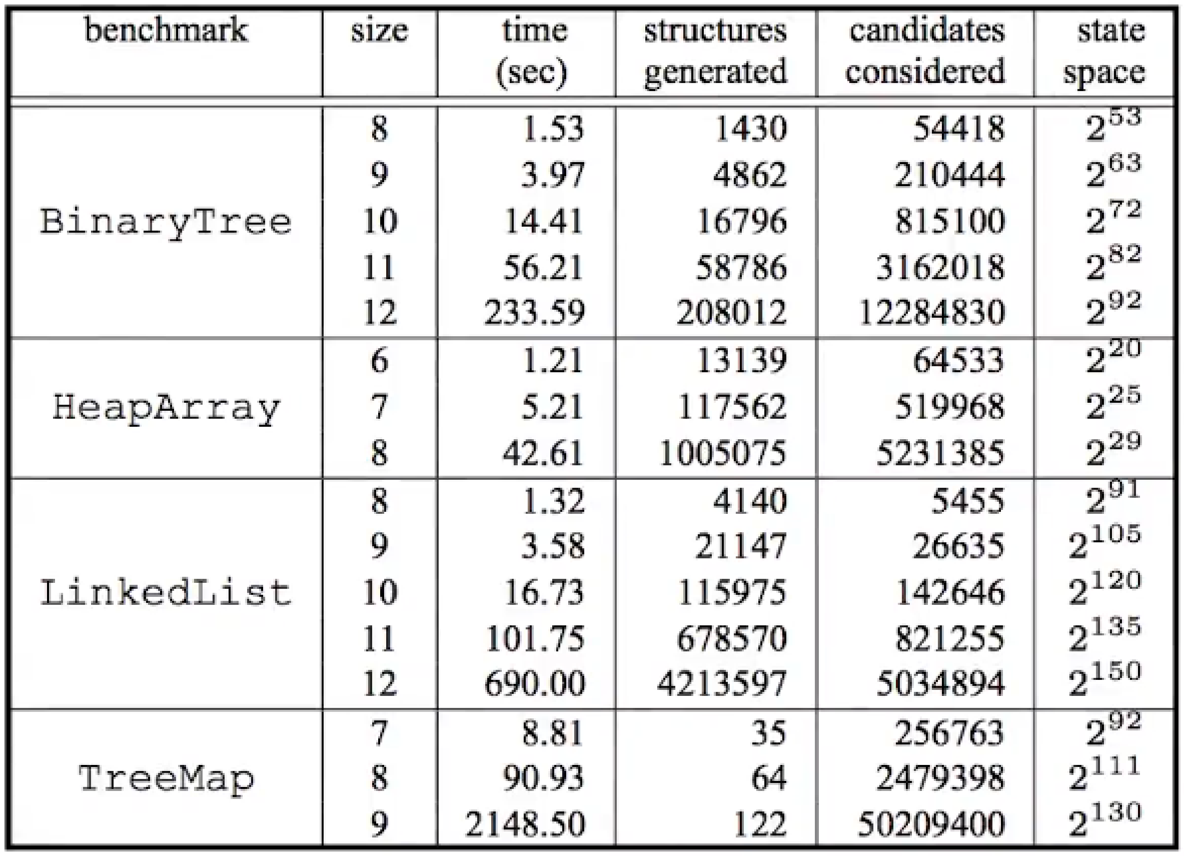
2. What the next two legal, non-isomorphic shapes Korat generates?
    - Shape 1
        - Root: N0
        - N0 left: null
        - N0 right: N1
        - N1 left: null
        - N1 right: N2
        - N2 left: null
        - N2 right: null
    - Shape 2
        - Root: N0
        - N0 left: null
        - N0 right: N1
        - N1 left: N2
        - N1 right: null
        - N2 left: null
        - N2 right: null

Enumerating Binary Trees 2

1. Assume the following shape of the tree:
    - Root: N0
    - N0 left: null
    - N0 right: N1
    - N1 left: null
    - N1 right: N2
    - N2 left: null
    - N2 right: null
2. What the next two legal, non-isomorphic shapes Korat generates?
    - Shape 1
        - Root: N0
        - N0 left: N1
        - N0 right: null
        - N1 left: null
        - N1 right: null
        - N2 left: null
        - N2 right: null
    - Shape 2
        - Root: N0
        - N0 left: N1
        - N0 right: N2
        - N1 left: null
        - N1 right: null
        - N2 left: null
        - N2 right: null

Experimental Results

1. Korat's scheme for eliminating illegal and redundant test cases has been shown to be highly effective



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}

Korat Experimental Results

Strengths and Weaknesses

1. Strengths:
   - Strong when we can enumerate all possibilities
     - E.g., Four nodes, two edges per node
   - Good for:
     - Linked data structures
     - Small, easily specified procedures
     - Unit testing
2. Weaknesses:
   - Weaker when enumeration is weak:
     - Integers, floating-point numbers, Strings
   - No links in the data to exploit isomorphism

Weaknesses

1. Korat is only as good as the pre- and post-conditions

```
Pre: is_member(x, list)
List remove(Element x, List list) {
    if (x == head(list))
        return tail(list);
    else
        return cons(head(list), remove(x, tail(list)));
}
Post: is_list(list')
```

Feedback-Directed Random Testing

1. Randoop: Random Tester for Object-Oriented Programs
    - Complementary to a deterministic test generator like Korat
        - Need to generate lists of different sizes as well as different sequences of list operations
    - Randoop generates object-oriented unit tests consisting of a sequence of method calls that set up state such as creating and mutating objects
        - Followed by an assert statement

Overview

1. Test is a sequence of public library methods that an application using the library can call
    - Uniform random testing was suitable for security testing or mobile apps, but creates too many illegal or redundant tests
    - Idea: Randomly create new test guided by feedback from previously created tests
    - Recipe:
        - Build new sequences incrementally, extending past sequences
        - As soon as a sequence is created, execute it
        - Use execution results to guide test generation towards sequences that create new object states

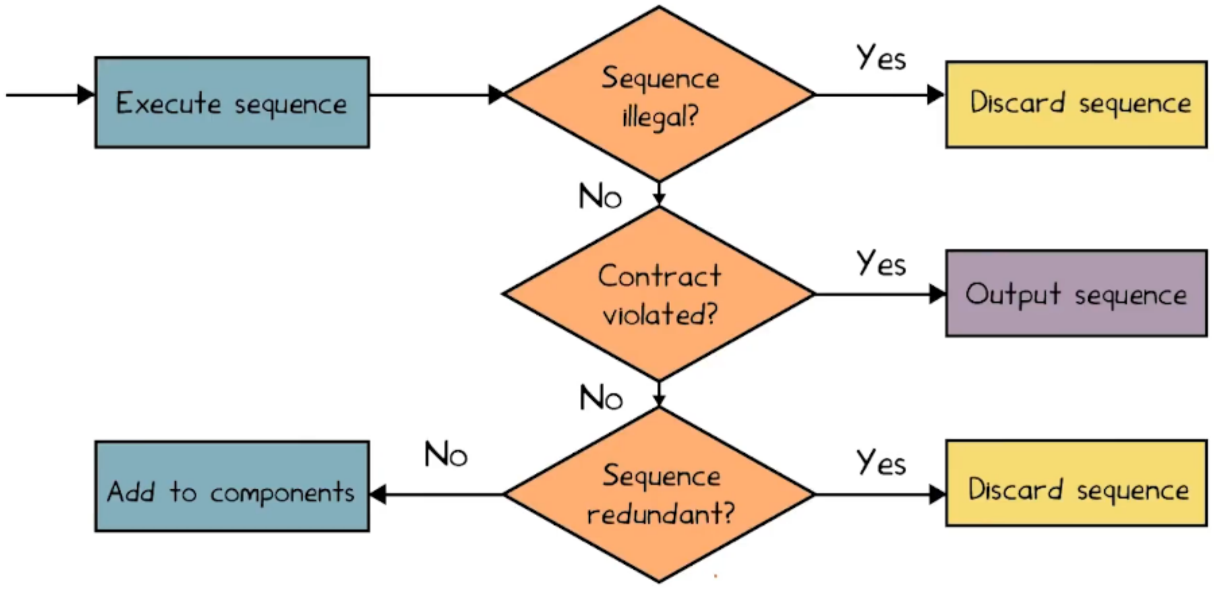
Randoop

1. Input
    - Classes under test
    - Time limit
    - Set of contracts: Properties the object generated must satisfy
        - o.hashCode() throws no exception
        - o.equals(o) == true
2. Output
    - Contract-violating test cases
        - Sequence of calls to public methods in input classes
        - Contract: Assertion about the result in the final call of the sequence

Randoop Algorithm

1. Components
    - Components = {int i = 0; boolean b = false; ...} // seed components
        - Grows with longer sequences as the algorithm progresses
2. Repeat until time limit expires
    - Create a new sequence
        - Random pick a method call Tret m(T1,...,Tn)
        - For each argument of type Ti, randomly pick sequence Si from components that constructs an object vi of that type
        - Create Snew = S1; ...; Sn; Tret vnew = m(v1...vn);
    - Classify new sequence snew:
        - Discard
        - Output as test
        - Add to components

Classifying a Sequence



Randoop Classifying a Sequence

Illegal Sequences

1. What makes a sequence illegal?
    - Sequences that "crash" before contract is checked
        - e.g., throw an exception

Redundant Sequences

1. How does Randoop determine if a sequence is redundant?
    - Maintain set of all objects created in execution of each sequence
    - New sequence is redundant if object created during its execution belongs to above set (using equals to compare)
    - This is an imperfect approach; might miss some bugs that can only be found through extending a redundant sequence
        - User can configure
    - Could also use more sophisticated state equivalence methods

Some Errors Found by Randoop

1. JDK containers have four methods that violate o.equals(o) contract
2. Javax.xml creates objects that cause hashCode and toString to crash, even though objects are well-formed XML constructs
3. Apache libraries have constructors that leave fields unset, leading to NPE on calls of equals, hashCode, and toString
4. Net framework has at least 175 methods that throw an exception forbidden by the library specification (NPE, out-of-bounds, or illegal state exception)
5. Net framework has 8 methods that violate o.equals(o) contract

Randoop Test Generation 1

1. Write the smallest sequence that Randoop can possibly generate to create a valid BinaryTree

```
Node v = null
BinaryTree bt = new BinaryTree(v);
```

2. Once generated, how does Randoop classify it?
    - Adds it to components for future extension

Randoop Test Generation 2

1. Write the smallest sequence that Randoop can possibly generate that violates the assertion in removeRoot().

```
BinaryTree bt = new BinaryTree(null);
bt.removeRoot();
```

2. Once generated, how does Randoop classify it?
    - Discards it as illegal

Randoop Test Generation 3

1. Write the smallest sequence that Randoop can possibly generate that violates the assertion in BianryTree's constructor.

```
Node v1 = new Node(null, null);
Node v2 = new Node(v1, v1);
BinaryTree bt = new BinaryTree(v2);
```

2. Can Randoop create a BinaryTree object with cycles using the given API?
    - No

Korat and Randoop

1. Identify which statements are true for each test generation technique:

	Korat	Randoop
Uses type information to guide test generation	X	X
Each test is generated fully independently of past tests		
Generates tests deterministically	X	
Suited to test method sequences		X
Avoids generating redundant tests	X	X

Test Generation: The Bigger Picture

1. Two powerful test generation techniques: Korat and Randoop
    - Why didn't automatic test generation become popular decades ago?
        - Weak-type systems
            - Test generation relies heavily on type information
            - C, Lisp just didn't provide the needed types
        - Contemporary languages lend themselves better to test generation
            - Java, UML

Conclusion

1. Automatic test generation is a good idea
    - Key: Avoid generating illegal and redundant tests
2. Even better, it is possible to do
    - At least for unit tests in strongly-typed languages

3. Being adopted in industry
    - Likely to become widespread