Automated test generation

https://www.youtube.com/playlist?list=PLF3-CvSRq2SaApl3Lnu6Tu_ecsBr94543

- The techniques we will learn in this lesson are more directed compared to random testing.

  They observe how the program under the test behaves on past tests in order to guide how to generate future tests.

  By being more directed, these techniques not only help find bugs more efficiently, but they also help to create a concise test suite that can be used for regression testing.

Outline

- Previously: random testing (fuzzing)
  - Security, mobile apps, concurrency
- Systematic testing using a tool named **Korat**
  - Linked data structures
- Feedback-directed random testing using a tool named **Randoop**
  - Classes, libraries

Korat

- Deterministic test generator
  - Research project MIT
- Idea behind:
  - Leverage pre-conditions and post-conditions to generate tests automatically

Pre- and post-conditions

- A **pre-condition** is a predicate
  - Assumed to hold before a function executes
- A **post-condition** is a predicate
  - Expectes to hold after a function executes, whenever the pre-condition also holds
- Pre- and post-conditions can be considered as **a special case of assertions**, which we saw in a previous lesson.

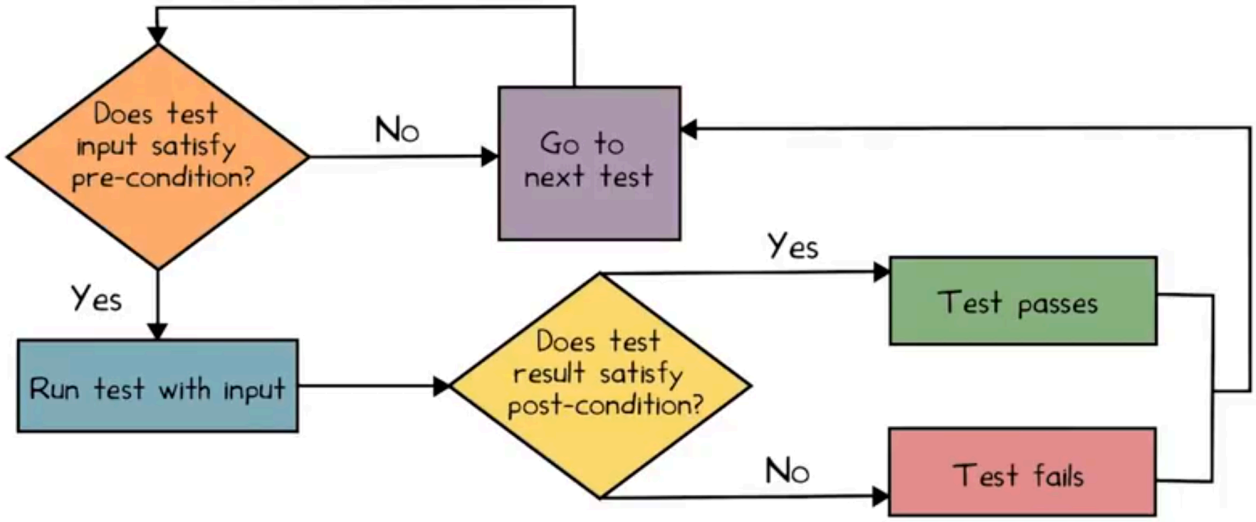
Example

```
Class Stack<T> {
      T[] array;
      int size;        frame conditions

      Pre: s.size() > 0
      T pop() { return array[--size]; }
      Post: s'.size() == s.size() - 1

      int size() { return size; }
```

More pre- and post-conditions

- Most useful if they are executable
  - Written in the programming language itself
  - A special case of **assertions**
- **Need not be precise**
  - May become more complex than the code
  - But useful even if they do not cover every situation

Using pre- and post-conditions



- Doesn't help write tests, but helps run them

The problem

- 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** (do not exercise interesting functionality of the software) and **redundant tests** (redundant ones that exercise the same facet of the software)
- Diverse: gives **good coverage**

An insight

- Often can do a good job by systematically **testing all inputs up to a small size**
- **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, probably works for all lists
    - E.g., because the function is oblivious to the length

How do we generate test inputs?

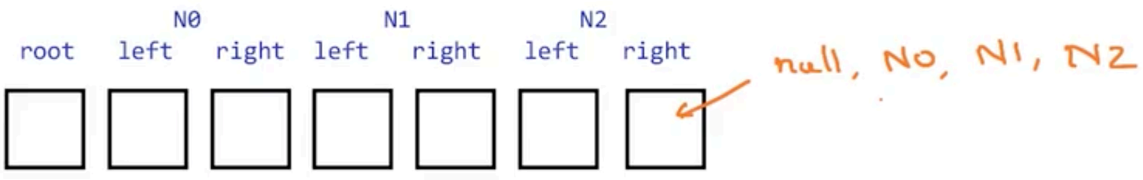
- Use the **types**
- Consider

```
Class BinaryTree {
      Node root;
      Class Node {
            Node left;
            Node right;
      }
}
```

- The class declaration shows what values (or null) can fill each field
- Simply enumerate all possible shapes with a fixed set of **Nodes**

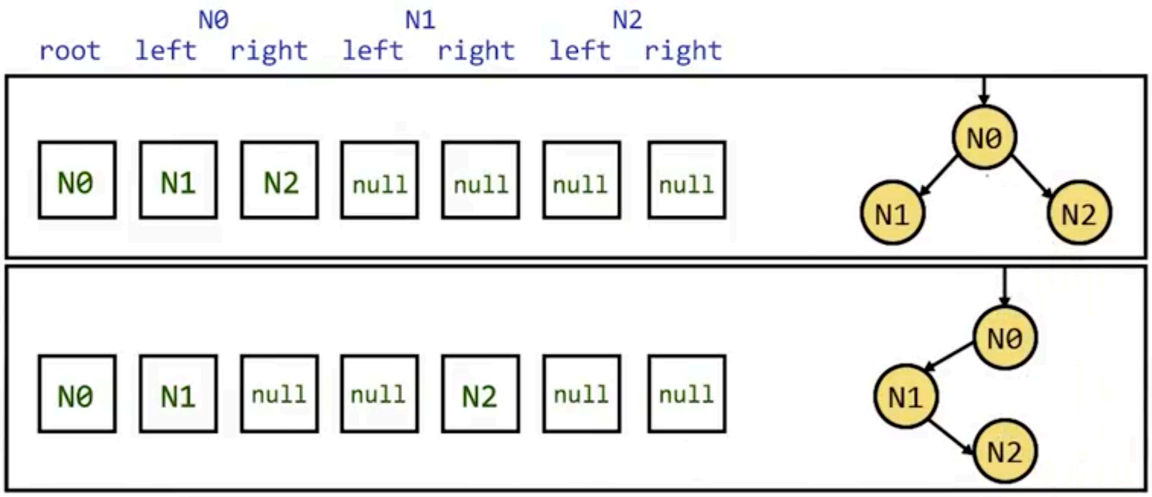
Scheme for representing shapes

- Order all possible values of each field
- Order all fields into a vector
- Each shape == vector of field values
- e.g., BinaryTree of up to 3 Nodes



Representing shapes

- In the following, each vector represents the adjacent shape

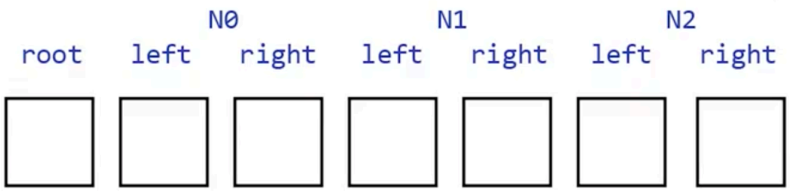


A simple algorithm

- User selects maximum input size **k**
- Generate all possible inputs up to size k
- Discard inputs where **pre-condition is false**
- Run program on remaining inputs
- Check results using **post-condition**

Enumerating shapes

- Korat represents each input shape as a vector of the following form:



- The total number of vectors of the above form is 4^7 = 16384.

  This is the size state of the problem of generating a binary tree with 3 known objects.

The general case for binary trees

- How many binary trees are there of size <= k?

  Calculation:

  - A BinaryTree object, bt
  - k Node objects, n0, n1, n2, n3 …
  - 2k+1 Node pointers
    - root (for bt)
    - left, right (for each Node object)
  - k+1 possible values (n0, n1, n2, … or null) per pointer

- (k+1)^(2k+1) possible “binary trees”

A lot of “trees”!

- The number of “trees” explodes rapidly
  - k = 3: over 16 000 “trees”
  - k = 4: over 1 900 000 “trees”
  - k = 5: over 360 000 000 “trees”
- Limits us to testing only very small input sizes
- Can we do better?

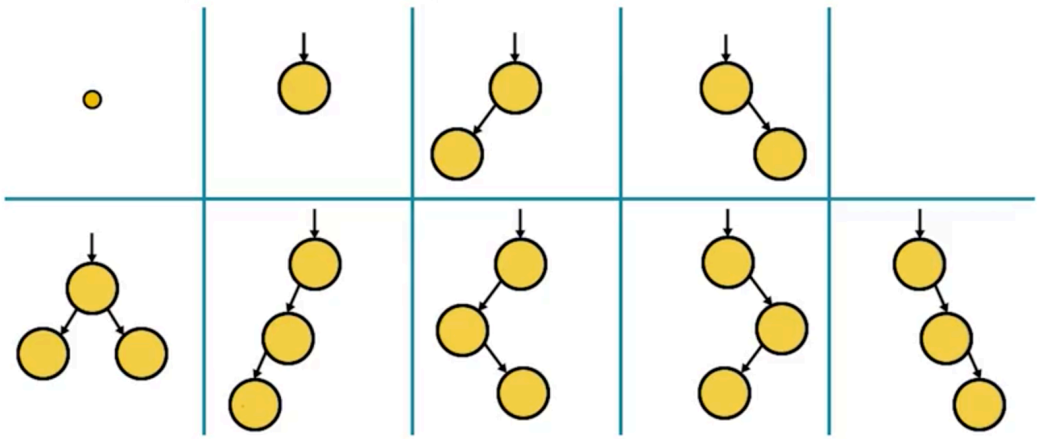
An overestimate

- (k+1)^(2k+1) trees is a gross overestimate
- Many of the shapes are not even trees:

- And many are isomorphic:

How many trees?

- Only 9 distinct binary trees with at most 3 nodes:



- To summarize, there are 2 central challenges:
  - **How to avoid generating** illegal test inputs, which in this case are **invalid trees?**
  - And how to avoid generating redundant test inputs, which in this case are **isomorphic trees**?

- These are challenges for any automated test generation technique, not only Korat.

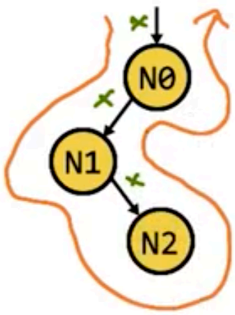
Another insight

- 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

- Instrument the **pre-condition**
  - Add code to observe its actions
  - Record fields accessed by the **pre-condition**
- **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



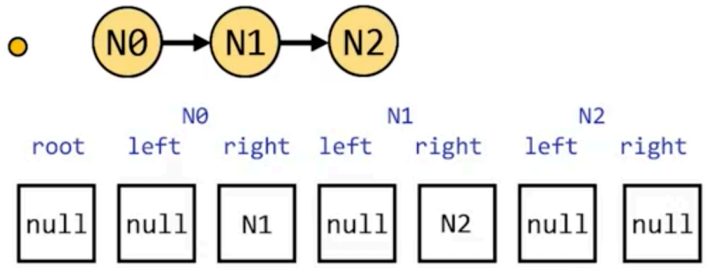
- Root may be null
- If root is not null
  - No cycles
  - Each node (**except root**) has one parent
  - Root has no parent

```
Class BinaryTree {
      Node root;
      Class Node {
            Node left;
            Node right;
      }
}

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;
}
```

- **Depth-first manner**

Example: using the pre-condition

- Consider the following **“tree”**:



- 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

- 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
- Paper describing the Korat algorithm: http://mir.cs.illinois.edu/marinov/publications/BoyapatiETAL02Korat.pdf

Example enumerating binary trees

- Legal, non-isomorphic shapes that Korat generates:

- So far, we have generated 7 non-isomorphic best cases.
  We could continue this process until the last 2 trees, with at most 3 nodes, are generated.

Experimental results

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}

- In practice, scheme for eliminating illegal and redundant test cases have proven to be highly effective.

Korat: strengths and weaknesses

- Strong when we can enumerate all possibilities
  - E.g., 4 nodes, 2 edges per node
- Good for:
  - **Linked data structures**
  - Small, easily specified procedures
  - Unit testing
- Weaker when enumeration is weak:
  - **Integers, floating-point numbers, strings**

Weaknesses

- 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_member(x, list')
```

- These are sufficient conditions to ensure that the remove method is functioning properly.
- If we are not as careful in our selection of pre- and post-conditions, we may end up with many useless tests, or even miss useful tests.

```
Pre: is_empty(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')
```

- Note that the pre-condition shown here, that the list object is not empty, is weaker than the earlier pre-condition that x be a member of the list.
  Whenever x is a member of the list, the list will be non-empty.
  So, we may end up with many useless tests that satisfy this weaker pre-condition, but not the earlier pre-condition.
- Conversely, Korat might not expand the element x because it is not accessed during this pre-condition resulting in missing useful tests.
  In other words, there may be bugs in this method that are dependent on the value of x that wouldn't be found by Korat using this pre-condition.
  Likewise, checking that the output of the remove method is still a list is weaker than the earlier post-condition, which stated that the output is a list that no longer contains x.
  Since this post-condition does not check that the list no longer contains x, it will silently succeed if the remove method has a bug that fails to remove x from the list, while still preserving the list structure.

Feedback-directed random testing

- a different test generation technique, which was first conceived in a tool called **Randoop**, which stands for Random Tester for Object Oriented Programs.
- While the deterministic test generation that Korat performs is better suited for generating different shapes of a data structure, Randoop is **better suited for creating different sequences of methods in an object-oriented library** that provides an application programming interface or API.
- This is not to say that one technique is better than another in testing. In fact, they are complementary to each other.

```
public static void test() {
      LinkedList l1 = new LinkedList();
      Object o1 = new Object();
      l1.addFirst(o1);
      TreeSet t1 = new TreeSet(l1);
      Set s1 = Collection.unmodifiableSet(t1);

      // This assertion fails
      assert(s1.equals(s1));
      }
```

- This test reveals 2 errors, one in the equals method (violation of the equals contract) and another in the TreeSet constructor method, which failed to throw a class cast exception as required by its specification.
- Formally speaking, 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 assertion** about the result of the final call.

  How does Randoop generate such tests?

Overview

- Problem with uniform random testing: creates too many **illegal** or **redundant** tests
- Idea: **randomly** create new test **guided by feedback** from previously created tests

  test == method sequence
- 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: input and output

- Input:
  - Classes under test
  - Time limit
  - Set of contracts
- Output:
  - Contract-violating test cases

```
LinkedList l1 = new LinkedList();
Object o1 = new Object();
l1.addFirst(o1);
TreeSet t1 = new TreeSet(l1);
Set s1 = Collection.unmodifiableSet(t1);

assert(s1.equals(s1));
```

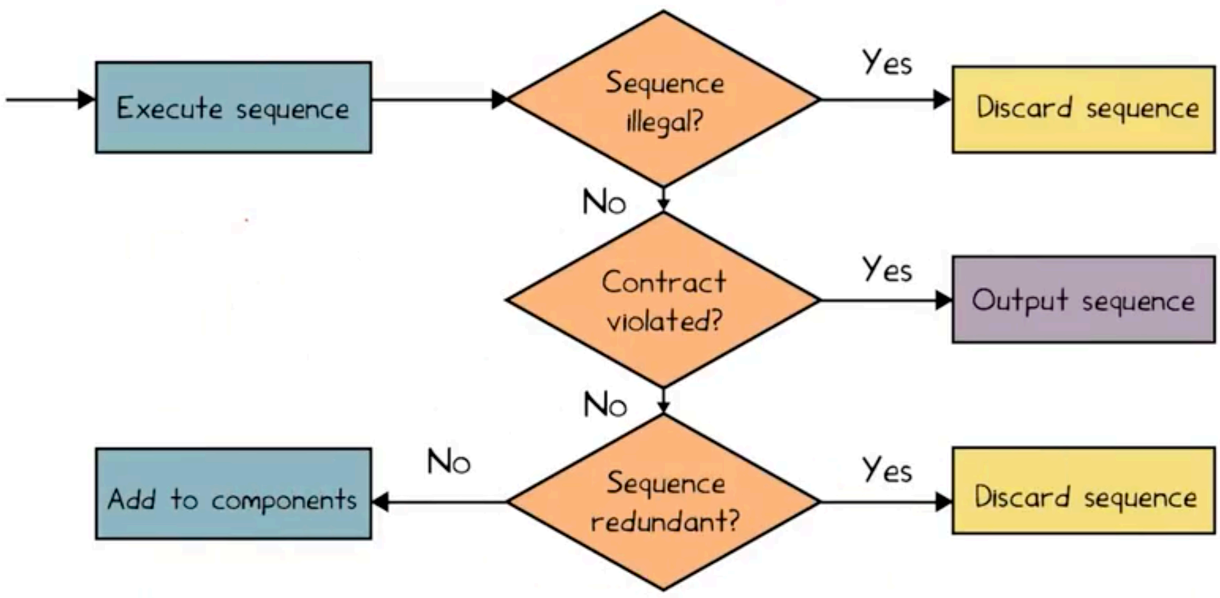
- No contract violated up to the end of the execution of the sequence consisting of the first 5 instructions:
  e.g., “o.hashCode() **throws no exception**”
- The final assertion should fail when executed:
  e.g., “o.equals(o) == true”

- These 2 conditions will be important in deciding what Randoop does with each test case it generates,
  - whether it discards the test case,
  - whether it outputs the test case, or
  - whether it retains the test case for extending in future test cases.

Randoop algorithm

- components = { int i = 0; boolean b = false; … } // seed components
- Repeat until time limit expires:
  - Create a new sequence
    - Randomly pick a method call $\mathbf{T_{ret}}$ **m(T1, …, Tn)**
    - For each argument of type **Ti**, randomly pick sequence **Si** from components that constructs an object **vi** of that type
    - Create $\mathbf{S_{new}}$ **= S1; …; Sn;** $\mathbf{T_{ret}}$ $\mathbf{v_{new}}$ **= m(v1… vn);**
  - Classify new sequence $\mathbf{S_{new}}$: discard/ output as test/ add to components

Classifying a sequence



Illegal sequences

- Sequences that "**crash**" before contract is checked
  - E.g., throw an exception

```
int i = -1;
Date d = new Date(2006, 2, 14);
d.setMonth(i); //pre: argument >= 0
assert(d.equals(d));
```

Redundant sequences

- 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)
- The following left-side sequence is redundant compared to right-side sequence:

```
Set s = new HashSet();
s.add("hi");


assertTrue(s.equals(s));
```

```
Set s = new HashSet();
s.add("hi");
s.isEmpty();

assertTrue(s.equals(s));
```

Some errors found by Randoop

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

Randoop test generation

- Consider

```
Class Node {
      Node left;
      Node right;
      public Node(Node l, Node r) {
            left = l;
            Right = r;
      }
}

Class BinaryTree {
      Node root;
      public BinaryTree(Node r) {
            Root = r;
            assert(repOk(this)); // repOk presented on page 8
      }
      public Node removeRoot() {
            assert(root != null);
            …
      }
}
```

- The smallest sequence that Randoop can possibly generate to create a valid BinaryTree:

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

- Once generated, Randoop classifies it
  - Adds components for future extension
  - Doesn’t discard it as illegal
  - Doesn’t output it as a bug
- The smallest sequence that Randoop can possibly generate that violates the assertion in RemoveRoot():

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

- Once generated, Randoop classifies it
  - Discards it as illegal
  - Doesn’t output it as a bug
  - Doesn’t add components for future extension
- The smallest sequence that Randoop can possibly generate that violates the assertion in BinaryTree’s constructor:

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

- Randoop cannot create a BinaryTree object with cycles using the given API

Korat and Randoop

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

Test generation: the bigger picture

- Why didn't automatic test generation become popular decades ago?
- Belief: weak-type systems
  - Test generation relies heavily on **type information**
  - **C, Lisp** just didn't provide the needed type
- Contemporary languages lend themselves better to test generation
  - **Java, UML**

What have we learned

- Automatic test generation is a good idea
  - Key: avoid generating **illegal** and **redundant** tests
- Even better, it is possible to do
  - At least for **unit tests** in **strongly-typed** languages
- Being adopted in industry
  - Likely to become widespread

Notă[1]