Automated Testing & Verification

Automatic Test Generation

Galeotti/Gorla/Rau
Saarland University
Testing & Debugging

- **Testing**: Find inputs that make the program fail
- **Debugging**: The process of finding the cause of a failure.
Test Case

- **Test Case Values/Test Input/Test Data**: Values that satisfy a test requirement

- **Expected output**: The result of the test case if the software under test behaves as expected.
  - Test Oracle
Adequacy criteria

- Goals of a Test Suite
  - All instructions
  - All conditions (decisions)
  - All paths
  - All inputs

- Goals might not be feasible
  - Example: dead code
  - Detection of unfeasible goals is undecidable in general

- 100% goal coverage might not be possible
Our ideal

- To generate automatically tests
  - Satisfying adequacy criteria
  - Avoid writing the tests ourselves
  - Find bugs faster

- Update tests automatically if the software evolves
It requires:

- Choosing adequacy criteria
- Generate inputs
- Check output
- Extend the spectra of possible inputs in a test case
  - We have no idea where the failure could be
Fuzz testing

- **Idea:** See how an application endures “noise” in the input

- **Inspired in real life:**
  - Unix `ssh` and `vi` use in a dial-up connection.

- **Fuzzers**
  - Tools that create mal-formed random inputs to a program
    - To crash the program
    - To find security flaws
Fuzz Testing - Typical Targets

- File formats
  - e.g. documents, pictures

- Network protocols
  - servers and clients

- Specific interfaces
  - e.g. DOM
The Lexer can be exercised in detail

But few inputs reach the parser...

And even less inputs reach the runtime...
Random Testing

- Create program inputs randomly
- Observe if the program behaves “correctly”
  - Using explicit contracts (pre & posts)
  - Implicitly: runtime undeclared exceptions
- Advantages:
  - Easy to implement
  - Good coverage if the test suite is big enough
int double (int v) {
    return 2*v;
}

void testMe (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

- Possible inputs:
  - x = 2, y = 10
    - First IF is false
  - x = 102, y = -10
    - First IF is false
  - x = 4, y = 2
    - First IF is true
  - x = 34, y = 0
    - First IF is false
  - x = 40, y = 20
    - It hits ERROR
Simple Random Generation

- Given $m(x_1:T_1, \ldots, x_k:T_k)$ a method under test
- For each $x_i$:
  - If $T_i$ is a primitive data type
    - A random primitive value
  - If $T_i$ is a reference, choose randomly among:
    - The null value
    - The constructor with no arguments (if it exists)
Given this method:

```java
public static int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

We generate random values for a and b:

- a=124, b=3
- a=-15, b=0
- **Given this method:**

  ```java
def public static Integer largest(LinkedList<Integer> list) {
    int index = 0;
    int max = Integer.MIN_VALUE;
    while (index <= list.size()-1) {
      if (list.get(index) > max) {
        max = list.get(index);
      }
      index++;
    }
    return max;
  }
```

- **We can use LinkedList() or null**
  - list=null
  - list=[] (the empty list)
Each test sequence is independent from the other sequences (redundancy)

Complex objects tend to be rather simple
Given \( m(x_1:T_1, \ldots, x_k:T_k):T_{k+1} \) a method under test

For each \( x_i \):

- If \( T_i \) is a primitive data type
  - A random primitive value
  - A method \( m'(y_1:T'_1, \ldots, y_j:T'_j):T_i \) (the return value is \( T_i \))
    - Recursively apply the same procedure to \( y_1, \ldots, y_j \)

- If \( T_i \) is a reference, choose randomly among:
  - The null value
  - A method \( m'(y_1:T'_1, \ldots, y_j:T'_j):T_i \) (the return value is \( T_i \))
    - Recursively apply the same procedure to \( y_1, \ldots, y_j \)
Example 1

- Program under test:
  ```java
  public static int max(int a, int b) {
    if (a > b) {
      return a;
    } else {
      return b;
    }
  }
  ```

- Random test sequence:
  ```java
  int b = 254;
  LinkedList<Integer> l = new LinkedList<Integer>();
  l.add(0, 2);
  Integer x = l.get(0);
  int a = x.intValue();
  max(a, b)
  ```
Method under test:

```java
public static Integer largest(LinkedList<Integer> list) {
    int index = 0;
    int max = Integer.MIN_VALUE;
    while (index <= list.size()-1) {
        if (list.get(index) > max) {
            max = list.get(index);
        }
        index++;
    }
    return max;
}
```

Random test sequence:

```java
LinkedList<Integer> list0 = new LinkedList<Integer>();
list0.add(0,2);
int int0 = list0.getFirst();
list0.add(0,int0);
largest(l);
```
Some problems

- How do we know a test is “interesting”
  - We know some sort of Test Oracle

- How do we select non-redundant tests?
  - More generation time => More generated tests
  - Test Selection is very expensive
Randomized Test Generator for Object-Oriented Programs

Execution Feedback
Object level:
- `o.equals(o)` returns true
- `o.equals(o)` throws no exception
- `o.hashCode()` throws no exception
- `o.toString()` throws no exception

Method level:
- Method throws no `AssertionError`
- Method throws no `NullPointerException` if all args != null

User can add more contracts (Randooop interface)
Randoop: Test Selection

- Randoop classifies generated tests into:
  - Failing Test Cases:
    - Execution led to a contract failure
  - Normal Test Cases:
    - Execution did not violate any contract

- All failing tests are collected.

- Normal tests are filtered
Randoop: Filtering

- **Equality**
  - The `equals()` method is used to see if a given object was created before
  - A pool with all created objects is kept alive.

- **Null**
  - Avoid using a null return value (use “null” directly instead).

- **Exceptions**
  - If the test leads to a exception, do not extend that.
The execution of these two sequences is redundant

Smaller size improves program understanding.
The execution of these two sequences is redundant.

Both hashSets are equal w.r.t. the equals() method.
Test case exhibiting a failure
  - Fails on Sun 1.5, 1.6.

```java
public static void test1() {
    LinkedList l1 = new LinkedList();
    Object o1 = new Object();
    l1.addFirst(o1);
    TreeSet t1 = new TreeSet(l1);
    Set s1 = Collections.unmodifiableSet(t1);
    Assert.assertTrue(s1.equals(s1));
}
```
Regression Test

- Passes on Sun 1.5, fails on Sun 1.6 Beta 2.

```java
public static void test2() {
    BitSet b = new BitSet();
    Assert.assertEquals(64, b.size());
    b.clone();
    Assert.assertEquals(64, b.size());
}
```
Advantages:
- Few requirements
- Unbiased

Disadvantages:
- It does not benefit from source code information.
- It is difficult to find “deep” errors.
Exhaustive Testing - Idea

- Generate all non-isomorphic valid inputs up to a given size.
- Use programmatic contracts to decide if an input is valid.
- Prune search space efficiently.
Exhaustive Testing - Example

- Type information
  - The class declaration states the values a field can take

```java
class BinaryTree {
    Node root;

class Node {
    Node left;
    Node right;
}
}
```

- Enumerate all possible values up to a given length.
Naïve Algorithm

1. Select $k$ the maximum input size
2. Generate all inputs up to size $k$
3. Discard all inputs that do not satisfy the precondition
4. Execute the program
5. Check the postcondition
Example

- Binary trees
- How many instances of $k \leq 3$?
  - 3 nodes
  - 2 values per node (left, right)
  - 4 possible values (consider also null) for each node and the tree instance.
  - $4 \times (4 \times 4)^3 = 16,384$ instances!
  - For 4 nodes it grows to more than 1,000,000!
- Number of instances grow exponentially.
This enumeration does not take into account many important aspects:

- Many inputs are not trees
- Many inputs are isomorphic instances.
Automated Testing Based on Java Predicates

http://korat.sourceforge.net/index.html

Efficiently enumerates instances by:

- Monitoring which fields are accessed
  - Which values should I change?

- Avoiding isomorphic instances
  - Vector representation of the heap configuration.
boolean repOk() {
    if (this.root == null)
        return true;
    Set visited = new HashSet();
    visited.add(t.root);
    List workList = new LinkedList();
    workList.add(t.root);
    while (!workList.isEmpty()) {
        Node current = (Node)workList.removeFirst();
        if (current.left != null) {
            if (!visited.add(current.left))
                return false;
            workList.add(current.left);
        }
        if (current.right != null) {
            if (!visited.add(current.right))
                return false;
            workList.add(current.right);
        }
    }
    return true;
}
- Isomorphic instances
  - Identical except object names
public static Finitization finBinaryTree(int NUM_Node) {
    Finitization f = new Finitization(BinaryTree.class);
    ObjSet nodes = f.createObjects("Node", NUM_Node);
    nodes.add(null);
    f.set("root", nodes); // root in null + Node
    f.set("Node.left", nodes); // Node.left in null + Node
    f.set("Node.right", nodes); // Node.right in null+ Node
    return f;
}
**Binary Trees – Example (3 nodes)**

```java
class BinaryTree {
    Node root;
    
class Node {
        Node left;
        Node right;
    }
}
```

<table>
<thead>
<tr>
<th>this</th>
<th>No</th>
<th>N1</th>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>left</td>
<td>right</td>
<td>left</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>
Isomorphic enumeration
this.root

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null

null
this.root
No.left
No.right

this.repOk() == true
(move right)
Binary Trees Example

```
this.root
No
  left
No
  right
null

this.root
No
  left
No
  right
null

this.repOk()==false
(increase index)
```
Binary Trees Example

```
this.repOk() == true
(move right)
```

```
this.root
No.left
No.right
N1.left
N1.right
this.repOk() == true
(move right)
```
Binary Trees Example

```
this.repOk() == false
(increase index)
```
Binary Trees Example

```
this.root
No.left
No.right

this.repOk()==false
(increase index)
```
Binary Trees Example

```
this.root
N0.left
N0.right
N1.left
N1.right
N2.left
N2.right

this.repOk()==true
(backtrack)
```
```
this.repOk() == true
(move right)
```
Binary Trees Example

this.root = this.repOk() == true

(move right)

this.root
No.left
No.right
N1.left
N1.right

this.N0
null
null
null
null
null
null

this.N2

N1
null
null
null
null
null
null

N0
null
null
null
null
null
null
Binary Trees Example

```
this.repOk() == false
(increase index)
```
Binary Trees Example

```
this.repOk() == false
(this.root
  No
    left
    null
    right
    null
    N1
      left
      N0
        root
        left
        N1
        right
        N2
        left
        right

No  null  N1  N1  null  null  null

this.repOk() == false
(increase index)
```
this.repOk() == true
(backtrack)

this.root
No.left
No.right
N1.left
N1.right
N2.left
N2.right
Binary Trees Example
Isomorphic instances enumeration:
- Given all the already assigned values, Korat can only use the same Restriction on values (only previous values +1)

Prune un-interesting search space:
- Korat monitors the field accesses to judge which fields must be modified during backtracking.
For each non-isomorphic instance up to the chosen finitization satisfying the Java predicate

- Creates the instance using Java reflection
- Execute the selected method on the instance
- Check the Java predicates on exit
How do we deal with Strings, floats, etc.?

Korat does not build the instance using a method sequence.

The performance of the Java predicate directly affects the Test Case Generation process.

Exhaustive generation can be infeasible for bigger scopes (100? 1000? 100000?)
Recap

- **Randoop**
  - Random Testing with Feedback loop

- **Korat**
  - Efficient non-isomorphic test input generation

- How can we use the source code to generate inputs?
  - **Pex**: white-box test case generation using concolic execution.
Verifying program behavior

- Examples: ESC/Java₂, JMLForge
Verifying program behavior

//@ requires ...
//@ ensures ...
procedure m(x, y, z) {
    if (x > y) {
        ...
    } else {
        ...
    }
    if (z == y) {
        ...
    }
}
Incremental Constraint Solvers

- Incremental solvers (SMT/SAT) allow the user to add new clauses/axioms when a solution is produced
  - This could be used to enumerate more solutions
  - Or it could be used to enumerate “interesting” executions of the program
//@ requires ...
//@ ensures false;
procedure m(x,y,z) {
    if (x>y) {
        GOAL_0
        ...
    } else {
        GOAL_1
        ...
    }
    if (z==y) {
        GOAL_2
        ...
    } else {
        GOAL_3
    }
}
Counterexample:

- an execution trace (inputs) s.t. the postcondition fails (ensures “false”)
- some goals were covered

x=1 y=0 z=0
Test Inputs

1) x=1 y=0 z=0

we push a new axiom to the Constraint solver
Goal 1 OR Goal 2 should be covered now
Incremental Constraint Solver

Goal 1 or goal 3

SAT

x=9 y=9 z=9
(Goals 1 && 2 covered)

Test Inputs
1) x=1 y=0 z=0

2) x=9 y=9 z=9

we push a new axiom to the Constraint solver
Goal 1 OR Goal 3 should be covered now
Incremental Constraint Solver

---
goal 0 or goal2

---
goal3

Program + Goals

Translator

Incremental Constraint Solver

\[
\begin{align*}
\text{Goal 0} & \text{ && } 3 \text{ covered} \\
\text{we push a new axiom to the Constraint solver} \\
\text{Goal 3 should be covered now}
\end{align*}
\]

Test Inputs

1) \(x=1 \ y=0 \ z=0\)

2) \(x=9 \ y=9 \ z=9\)

3) \(x=15 \ y=-2 \ z=-4\)
Program + Goals → Translator → Incremental Constraint Solver → SAT

Test Inputs
1) x=1 y=0 z=0
2) x=9 y=9 z=9
3) x=15 y=-2 z=-4

No more goals to cover!

x=15 y=-2 z=-4
(Goals 0 && 3 covered)
FAJITA

- JML input programs
- Branch/Goal/Path coverage
- Incremental SAT-Solving
- [http://www.dc.uba.ar/fajita](http://www.dc.uba.ar/fajita)

Advantages
- Fewer calls to the constraint solver in worst case

Disadvantages
- Complete CFG has to be encoded as a formula (more complex than only a path)