CSC 509 Lecture Notes Week 2

Assignment 1 Ideas

Concepts Underlying Testing Research

I. **As described in assignment 1 writeup,**
everyone briefly presents their selected paper.

II. **After the paper presentations,** we'll finish up testing terminology from Lecture Notes Week 1

III. **Some more specific testing terminology.**

A. Consider last five years of ISSTA pubs.

B. There have been 140 papers over these years.

C. Here are the top five keywords used:

Top ISSTA Keywords, cont'd

1. [automated] test [case] generation (27 times)
2. static analysis (18 times)
3. symbolic execution (11 times)
4. dynamic analysis (10 times)
5. coverage (8 times)

D. **Majority of papers describe testing *tools*.**

1. Many tools generate executable tests.

2. Others perform analysis before, during, after, or instead of tests.

IV. So how does a testing tool do these things?

IV. So how does a testing tool do these things?

A. At the core is a *program compiler*.

IV. So how does a testing tool do these things?

A. At the core is a ***program compiler***.

B. It generates an ***annotated parse tree***,
or comparable structure.

IV. So how does a testing tool do these things?

A. At the core is a ***program compiler***.

B. It generates an ***annotated parse tree***,
or comparable structure.

C. It also generates an ***symbol table***.

IV. So how does a testing tool do these things?

A. At the core is a ***program compiler***.

B. It generates an ***annotated parse tree***,
or comparable structure.

C. It also generates an ***symbol table***.

D. The generation, analysis, execution, or coverage procedure ***traverses the tree***.

IV. So how does a testing tool do these things?

A. At the core is a ***program compiler***.

B. It generates an ***annotated parse tree***,
or comparable structure.

C. It also generates an ***symbol table***.

D. The generation, analysis, execution, or coverage procedure ***traverses the tree***.

E. During the traversal, ***tool-specific*** rules are applied for the problem at hand.

V. Consider the following example:

```
public class Example {
  /**
   * Return true if the given integer is
   * positive and even.
   */
  /*@
     ensures \result == i > 0 && i % 2 == 0
   @*/
  public static boolean isPositiveEven(int i) {
    if (i > 0 && i % 2 == 0)
      return true;
    else
      return false;
  }
}
```

VI. (Symbolic) Execution

A. Parse program code.

B. At each node, apply execution rule.

C. E.g., to evaluate ’>’ operator, do this:

(Symbolic) Execution, cont'd

```
public Value evalGreaterThan(
    TreeNode expr, SymbolTable symtab) {

  Value v1 = eval(expr.child1, symtab);
  Value v2 = eval(expr.child2, symtab);

  return new BooleanValue(v1.val > v2.val)
}
```

(Symbolic) Execution, cont'd

D. Difference between regular and symbolic --

1. For regular, lookup var value in symbol table.

2. For symbolic, use string var name, produce result with string concatenation.

3. E.g., to ***symbolically*** evaluate '>' operator:

(Symbolic) Execution, cont'd

```
public Value evalGreaterThan(
    TreeNode expr, SymbolTable symtab) {

  Value v1 = eval(expr.child1, symtab);
  Value v2 = eval(expr.child2, symtab);

  if (isLiteral(v1) && isLiteral(v2)
    return new BooleanValue(v1.val > v2.val)
  else
    return new StringValue(
          v1.val + ">" + v2.val);
}
```

VII. Blackbox Test Case Generation

A. Apply well-known rules for test cases.

B. E.g., five cases for a numeric range:

1. well below bound
2. 1 below bound
3. at bound
4. 1 above bound
5. well above bound

Blackbox Test Case Generation, cont'd

C. To implement, e.g. range tests

1. parse ***pre and post-conditions***
2. traverse tree
3. for inequality of the form "x < C", eject test code like this for each rule-based value:

```
x = nextRuleBasedValue();
assertTrue(validatePostcond(x,C));
```

VIII. **Whitebox Test Case Generation**

A. Apply similar well-know rules to blackbox.

B. Parse ***code*** & traverse tree.

C. Similar code ejection to blackbox.

IX. **Coverage**

A. Annotate program tree with line numbers.

B. Execute tree.

C. At each tree node with line number, increment it's ***execution count*** annotation.

D. Do post-execution analysis for coverage report.

X. Static Analysis, e.g., for Smart Regression

A. Traverse changed portion of parse tree.

B. Determine for each method in symbol table if it is reachable.

C. If so, mark its tests as requiring re-execution.

XI. Dynamic Analysis, E.g., Smart Regression

A. Parse test code.

B. For each called test method, memoize its results.

C. If test method called again, use memoized value.