Automatic Test Generation

Lecture 3
CS295

The Idea

• Automatically generate tests for software

• Why?
  – Find bugs more quickly
  – Conserve resources
    • No need to write tests
    • If software changes, no need to maintain tests
    • No need for testers?!

The Problem

• Automated testing is hard to do

• Probably impossible for whole systems

• Certainly impossible without specifications

Pre- & Post-Conditions

• A pre-condition is a predicate
  – assumed to hold before a function executes

• A post-condition is a predicate
  – Known to hold after a function executes
  – Whenever the precondition also holds

Example

Pre: member(x,l)

List remove(x : Elem ,l: List) {
  if (x == head(l))
    return tail(l);
  else
    return cons(head(l),remove(x,tail(l)));
}

Post: !member(x,l)

More on Pre- and Post-Conditions

• Most useful if they are executable
  – Written in the programming language itself
  – A special case of asserts

• Recommended by pundits
  – And academics

• Need not be precise
  – Full pre- and post-conditions may be more complex than the code!
  – But useful even if they do not cover every situation
Using Pre- and Post-Conditions

- Pre-/Post-Conditions are specifications
- To perform a test:
  - Check that the test input satisfies pre-condition
  - Run test
  - Check that the test result satisfies post-condition
- Doesn’t help write tests, but it helps run them

Three Automatic Techniques

- Randomized testing
- Mutation analysis
- Korat

Random Testing

- Feed random inputs to a program
- Observe whether it behaves "correctly"
  - Execution satisfies pre & post conditions
  - Or just doesn’t crash
    - A simple pre/post condition
- E.g., the Fuzz experiment on Unix utilities

Random Testing: Good News & Bad News

- Randomization is a highly effective technique
  - Easy to implement
  - Provably good coverage for enough tests
- But
  - To say anything rigorous, must be able to characterize the distribution of inputs
  - Easy for string utilities
  - Harder for systems with more arcane input
    - E.g., parsers for context-free grammars

Random Testing: Example

- The lexer is very heavily tested by random inputs
- But testing of subsequent stages is much less efficient

Mutation Analysis

- How do we know our test suite is any good?
- Idea: Test variations on the program
  - E.g., replace \( x > 0 \) by \( x < 0 \)
  - Replace \( I \) by \( I+1, I-1 \)
- If the test suite is good, it should report failed tests in the variants
  - This is mutation analysis
**Mutation Analysis**

- Modify (mutate) each statement in the program in finitely many different ways
- Each modification is one mutant
- Check for adequacy wrt the set of mutants
  - Find a set of test cases that distinguishes the program from the mutants

**What Justifies This?**

- The "competent programmer assumption"
  The program is close to right to begin with
- It makes the infinite finite
  We will inevitably do this anyway; at least here it is clear what we are doing
- This already generalizes existing metrics
  If it is not the end of the road, at least it is a step forward

**The Plan**

- Generate mutants of program \( P \)
- Generate tests
  - By some process
- For each test \( t \)
  - For each mutant \( M \)
    - If \( M(t) \neq P(t) \) mark \( M \) as killed
- If the tests kill all mutants, the tests are adequate

**Generating Tests**

- Mutation analysis seeks to generate adequate test sets automatically
- Must determine inputs such that
  - Mutated statement is reached
  - Mutated statement produces a result different from original

**Automatic Test Generation**

- This is not easy to do
- Approaches
  - Use weakest-preconditions
  - Work backwards from statement to inputs
    - Take short paths through loops
  - Generate symbolic constraints on inputs that must be satisfied
  - Solve for inputs

**A Problem**

- What if a mutant is equivalent to the original?
- Then no test will kill it
- In practice, this is a real problem
  - Not easily solved
    - Try to prove program equivalence automatically
    - Often requires manual intervention
Korat

• A test-generation research project

• Idea
  - Leverage pre- and post-conditions to generate tests automatically

• But how?

The Problem

• There are infinitely many tests
  - Which finite subset should we pick?

• And even finite subsets can be huge
  - Need a subset which gives good coverage
  - Without a lot of redundancy
    - Many tests will just test the same thing
    - Need a way to select a diverse test suite

An Insight

• We can often do a good job by testing all inputs up to a certain, small size

• The "small test case" hypothesis
  - If there is any test that causes the program to fail, there is a small test

• If a list function works on lists of length 0, 1, 2, and 3, it probably works on all lists.
  - E.g., because the function is oblivious to the length

How Do We Generate Test Inputs?

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

• Use the types
  - The class declaration shows what values (or null) can fill each field

• Simply enumerate all possible shapes with a fixed set of Nodes.

A Simple Algorithm

• User selects maximum input size \( k \)

• Generate all possible inputs up to size \( k \)

• Discard inputs where precondition is false

• Run program on remaining predicates

• Check the results using the postcondition

Example: Binary Trees

• How many binary trees are there of size \(<= 3\)?

• Calculation:
  - 3 nodes
  - 2 slots per node
    - Left and right children
  - 4 possible values (one of the nodes or nil) for
    - Each slot
    - The root

  \[ 4 \times (4 \times 4)^3 = 2^14 = 16,384 \text{ possible trees} \]
A Lot of Trees!

• The number of trees explodes rapidly
  - >1,000,000 trees of size \leq 4
  - >16,000,000 trees of size \leq 5
• Limits us to testing only very small input sizes
• Can we do better?

An Overestimate

• But 16,384 trees is a gross overestimate!
• Many of the shapes are not trees:
  ![Tree Shapes]
  
  Many trees are isomorphic

How Many Trees?

• In fact, there are only 9 distinct binary trees of size \leq 3

The Insight

• We want to avoid generating trees that don’t satisfy the pre-condition in the first place.
• That means we must use the pre-condition to guide the generation of tests

The Technique

• Instrument the pre-condition
  - Add code to observe its actions
  - In particular, record fields of the input the pre-condition accesses
• Observation
  - If the pre-condition does not access a field, then the result of the pre-condition did not depend on that field.

The Pre-Condition for Binary Trees

• Checks
  - The root may be null
    - Return false
  - All nodes are unique
    - No cycles
    - Every node has one parent
      - Except the root
Example: Using the Pre-Condition

- Consider the following “tree”
  - The pre-condition accesses only the root
    - Because 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

- Tests are enumerated
  - Beginning with the smallest
  - Next test generated by
    - Expanding a null pointer field
    - Backtracking if all possibilities for a field are exhausted

- Rule
  - Never enumerate parts of input not examined by the precondition

Isomorphic Tests

- We also want to avoid isomorphic tests
  - E.g., distinct trees with the same shape
- Number all objects within a type
- Number all fields
  - In order in which pre-condition accesses them
- When backtracking on field \( f \)
  - Check if next object in ordering results in lexicographically least of structures of this shape

Error Specifications

- Actually give two specifications
  - Normal behavior specification
  - Error behavior specification
    - Under what circumstances exceptions are thrown

Results

- The techniques for eliminating redundant tests are very effective
- E.g., there are 429 binary trees of size 7
  - Infeasible to test on trees this large without the techniques for eliminating redundant tests
- Time to generate and run all tests usually seconds, sometimes minutes

Strengths

- Good for
  - Linked data structures
  - Small, easily specified procedures/methods
  - Unit testing
  - Other opinions?
Weaknesses

Only as good as the pre- and post-conditions

Pre: member(x,l)

List remove(x : Elem ,l: List) {
    if (x == head(l))
        return tail(l);
    else
        return cons(head(l),remove(x,tail(l)));
}

Post: !member(x,l)

Weaknesses (Cont.)

- Strong when we can enumerate all possibilities
  - Four nodes, two edges per node

- Weaker when enumeration is weak
  - Integers
  - Floating point numbers
  - Strings

Weaknesses (Cont.)

Not as good for non-deterministic methods

"Every packet sent is eventually acknowledged by the receiver"

Weaknesses (Cont.)

- Probably not as strong for checking exception behavior as for normal behavior

- Paper is thin on examples of exception behavior

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
  - To enumerate all possible instances of a type
    - Up to a certain size
  - C, Lisp just didn't provide the needed types

- Contemporary languages lend themselves better to test generation
  - Java
  - UML
Conclusions

• Automatic test generation is a good idea

• Even better, it is possible to do
  - At least for unit tests
  - In strongly-typed languages

• Being adopted in industry
  - Likely to become widespread