Automatic Test Generation
First, about Purify

• Paper about Purify (and PurifyPlus) posted
• How do you monitor reads and writes:
  – insert statements before and after reads, writes in code
    • can still be done with binaries
• But this affects performance
  – Without watching reads/writes, the overhead is small
  – With reads/writes, can be 10X slowdown
  – This is still OK for such a debugging tool
Homework 1

• Due Monday (Sep 23, 9 AM)
• Make sure you’ve started already
• If you’re using Java 7 (e.g., on EDLab):
  javac -g -source 6 -target 6 *.java
Questions?
Automated Test Generation 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 pre-condition also holds
Example

Pre-condition: \( l.\text{contains}(x) \)

```java
List remove(LinkedList l, Element x) {
    if (x == l.head())
        return l.tail();
    else
        return
            new LinkedList(l.head(), remove(l.tail(), x));
}
```

Post-condition: \(!l.\text{contains}(x)\)

Does this post-condition hold?

How can the pre-condition change for the post condition to hold?
Are pre- and post-conditions a good idea?

- Most useful if they are executable
  - written in the programming language itself
  - a special case of assertions

- Recommended by software engineers
  - and everyone who studies software engineering

- Can reduce ambiguity in specification

- May be somewhat imprecise and incomplete
  - full pre- and post-conditions may be more complex than the code!
  - still useful even if they do not cover every situation
Using Pre- and Post-Conditions

- Pre-/Post-Conditions are specifications

- To perform a test:
  - Generate an input (any input)
  - Check that the test input satisfies the pre-condition
  - Run test
  - Check that the test result satisfies the post-condition

Helps run tests, might even help write them!
How can we generate tests?

• Randomized testing
• Mutation Testing
• Korat
Random Testing

• Feed random inputs to a program

• Observe whether it behaves “correctly”
  – execution satisfies pre- and post-conditions
  – or just doesn’t crash
    (A simple pre/post condition)
Random Testing: Good and Bad News

• Randomization is highly effective
  – easy to implement
  – provably good coverage for enough tests

• But
  – to say anything rigorous, we must be able to characterize the distribution of inputs
  – easy for string utilities
  – harder for systems with more arcane input
    for example, parsers for context-free grammars
What about staged components?

If we only control the input to the whole system (input 1), can we test the circle well?
Mutation Analysis

• How do we know our test suite is any good?

• Idea: Test variations on the program
  – for example, replace $x > 0$ with $x < 0$
  – or replace $i$ by $i+1$ or $i-1$

• If the test suite is good, it should report failed tests in the variants
Mutation testing is one way to check automated testing.
Mutation Analysis Summary

• Mutate each statement in the program in finitely many different ways
• Each modification is one mutant
• Check if a set of mutants is adequate
• Find a set of test cases that distinguishes the program from the mutants
What Justifies Mutation Testing?

- Competent programmer assumption
  - the program is close to correct

- Mutations are representative of common errors
  - off by one errors, wrong comparison errors

- It formalizes test writing
  - we write tests for corner cases and off-by-one errors. There are an infinite number of them. This way, we formalize the process.

- This is a start
  - testing does not stop here
Back to automated testing

• Generate mutants of program $P$

• Generate tests
  (somehow)

• 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

This is the hard part!

- Use weakest-preconditions
  - work backwards from statement to inputs

- Take short paths through loops
  - try it 0 times, 1 time, 2 times

- Generate symbolic constraints on inputs that must be satisfied

- Solve for inputs
What if a mutant is equivalent to the original?

• No test will kill it

• In practice, this is a real problem
  – hard to solve

• We could try to prove program equivalence
  – but automating this is very hard
  – undecidable problem
Korat: A way to generate tests

Use pre- and post-conditions to generate tests automatically
Problem Korat tackles:

• There are infinitely many tests
  – which finite subset should we pick?

• And even finite subsets can be too big
  – we need a subset which yields good coverage
  – without a lot of redundancy
    • many tests will just test the same thing
    • we need a way to select a diverse test suit
Small test case hypothesis:

If there exists a test that causes the program to fail, there exists a small test case that causes the program to fail.

If a list function works on lists of length 0, 1, 2, and 3, it probably works on all lists.
Korat’s insight

• Use the small test hypothesis

• We can often do a good job by testing all inputs up to a certain, small size
How do we generate test inputs?

- 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: put it all together

- User selects maximum input size $k$
- Generate all possible inputs up to size $k$
- Discard inputs where pre-condition is false
- Run the program on remaining inputs
- Check the results using the post-condition
Example: Binary Trees

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

• 3 nodes
  – 2 slots per node (left and right)
  – 4 possible values (one of the nodes or null) for
    • each slot
    • the root

$$4 \times (4 \times 4)^3 = 2^{14} = 16,384$$ possible trees
That’s 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?
Actually, I lied

- 16,384 trees is a gross overestimate!
- Many of the shapes are not trees:
- Many trees are isomorphic
How many trees really?

- There are only 9 distinct binary trees on 3 or fewer nodes.
Use our constraints to help us

• 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

• And use the constraints on distinctness of inputs
Observe the pre-condition

• Instrument the pre-condition
  – add code to observe it at runtime
  – in particular, record fields of the input the precondition accesses

• Observation:
  – if the pre-condition does not access a field, then the result of the pre-condition did not depend on that field.
Binary tree example

• Pre-condition checks
  – if the root is null
    return false
  – all nodes must be unique
    • no cycles
    • every node has one parent
      (except the root, which has 0)
Example:

- Consider the following “tree”

- The pre-condition accesses only the root
  - since the root is null, every possible shape for the other nodes would yield the same result

- This single input eliminates 25% of the tests
Karat enumerates the tests

• Start with the smallest
• Next test generated by
  – expanding a null pointer field
  – backtracking if all possibilities for a field are exhausted

• Never enumerate parts of input not examined by the precondition
Isomorphic tests

• We also want to avoid isomorphic tests
  – distinct trees with the same shape
• Number all objects within a type
• Number all fields
  – in the pre-condition access order
• When backtracking on field $f$
• Check if next object in ordering results in lexicographically least of structures of this shape
Error specifications

We can have two specifications:

- Normal behavior specification

- Error behavior specification
  under what circumstances exceptions are thrown
Korat Results

- Eliminating redundant tests is very effective
  - there are only 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 and methods
  – unit testing
Weaknesses (conditions)

Only as good as the pre- and post-conditions

Pre-condition: l.contains(x)

```java
List remove(LinkedList l, Element x) {
  if (x == l.head())
    return l.tail();
  else
    return
      new LinkedList(l.head(), remove(l.tail(), x));
}
```

Post-condition: !(l.contains(x))
Weaknesses (conditions)

Only as good as the pre- and post-conditions

Pre-condition: !(l.isEmpty())

List remove(LinkedList l, Element x) {
    if (x == l.head())
        return l.tail();
    else
        return new LinkedList(l.head(), remove(l.tail(), x));
}

Post-condition: l.isList()
Weaknesses (large data structures)

• Strong when we can enumerate all possibilities
  – four nodes, two edges per node
• Weaker when enumeration is weak
  – integers
  – floating point numbers
  – strings
Weakness (nondeterminism)

Not as good for nondeterministic methods

For example, what about a condition that says “Every packet sent is eventually acknowledged by the receiver”? 
Test generation

• Automatic test generation is a good idea
• Types languages are a plus for generation
  – C++, Java, UML (C, Lisp do not provide needed types)
• Works well for unit tests
• Being adopted in industry
• Promising future