Lecture 17: Automated Test Generation 2

CS 5150, Spring 2026

Announcement

- No office hours next week on Monday (Saikat is travelling)
- Please email me if you want to meet next week.

How Good Is Your Test Suite?

- How do we know that our test suite is good?
  - Too few tests: may miss bugs
  - Too many tests: costly to run, bloat and redundancy, harder to maintain
- Two approaches:
  - Code coverage metrics
  - Mutation analysis (or mutation testing)

Code Coverage

- Metric to quantify extent to which a program's code is tested by a given test suite
- Given as percentage of some aspect of the program executed in the tests
- 100% coverage rare in practice: e.g., (provably) unreachable code
  - Often required for safety-critical applications

Types of Code Coverage

- Function coverage: which functions were called?
- Statement coverage: which statements were executed?
- Branch coverage: which branches were taken?
- Many others: line coverage, condition coverage, basic block coverage, path coverage, …

Mutation Analysis

- Founded on “competent programmer assumption”:

  The program is close to correct to begin with

- Key idea: Test variations (mutants) of the program
  - Replace x > 0 by x < 0
  - Replace w by w + 1, w - 1
- If test suite is good, should report failed tests in the mutants
- Find set of test cases to distinguish original program from its mutants

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

QUIZ: Code Coverage Metrics

Test Suite: `{ foo(1, 0) }`

Statement Coverage: ☐ %

Branch Coverage: ☐ %

Give arguments for another call to `foo(x, y)` to add to the test suite to increase both coverages to 100%.

x = ☐ y = ☐

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

QUIZ: Code Coverage Metrics

Test Suite: `{ foo(1, 0) }`

Statement Coverage: 80 %

Branch Coverage: 50 %

Give arguments for another call to `foo(x, y)` to add to the test suite to increase both coverages to 100%.

x = 1 y = 1

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

QUIZ: Mutation Analysis - Part 1

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 x <= y → x > y	☐	☐
Mutant 2 x <= y → x != y	☐	☐

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

Is the test suite **adequate** with respect to both mutants? ○ Yes ○ No

PollEv: PollEv.com/cs5150sp26

QUIZ: Mutation Analysis - Part 1

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 x <= y → x > y	☐	☑
Mutant 2 x <= y → x != y	☑	☑

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

Is the test suite **adequate** with respect to both mutants?

○ Yes ☑ No

QUIZ: Mutation Analysis - Part 2

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 x <= y → x > y	☐	☑
Mutant 2 x <= y → x != y	☑	☑

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

Give a test case which Mutant 2 fails but the original code passes.

assert:

foo(☐,☐) == ☐

PollEv: PollEv.com/cs5150sp26

QUIZ: Mutation Analysis - Part 2

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 x <= y → x > y	☐	☑
Mutant 2 x <= y → x != y	☑	☑

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

Give a test case which Mutant 2 fails but the original code passes.

assert:

foo(1 , 0) == 0

Beyond random input generation

- **Grammar-Based Fuzzing**: Generate inputs conforming to a specified grammar (aware of input structure)

```
<start> ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= +<factor> | -<factor> | (<expr>) | <int> |
<int>.<int>
<int> ::= <digit><int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Arithmetic expression grammar

Generate

```
(-+++((1 + (+7 - (-1 * (++-+7.7 - -+-4.0))))) * +--4 - -(6) + 64) 8.2 - 27
- -9 / +((+9 * --2 + --+-+-((-1 * +(8 - 5 - 6)) * (-((-+(((+(4))))) - ++4) /
+(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6) * ---+6)) / +--(+-+-7 * (-0 *
(+(((((2)) + 8 - 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1) / 482) / +++-
+0)))) * -+5 + 7.513)))) - ...
```

Arithmetic expression

- **Symbolic execution**: Collect “symbolic” path constraints and solve them to systematically explore all “program paths”. Need rigorous program analysis/very expensive.

Collecting Path Constraints

```
int foo(int x, int y) {
   int z = 0;
   if (x <= y) {
       if (y>999){
          z = x+999;
       }
       else{
           z=x;
       }
   } else {
       z = y;
   }
   return z;
}
```

(x<=y)

(x<=y) AND (y>999)

(x<=y) AND !(y>999)

!(x<=y)

Path 1: (x<=y) AND (y>999)

Path 2: (x<=y) AND !(y>999)

Path 3: !(x<=y)

Solve constraints to generate x and y using an SMT/constraint solver to cover all paths!

Beyond random input generation

- **Grammar-Based Fuzzing**: Generate inputs conforming to a specified grammar (aware of input structure)

```
<start> ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= +<factor> | -<factor> | (<expr>) | <int> |
<int>.<int>
<int> ::= <digit><int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Arithmetic expression grammar

Generate

```
(-+++((1 + (+7 - (-1 * (++-+7.7 - -+-4.0))))) * +--4 - -(6) + 64) 8.2 - 27
- -9 / +((+9 * --2 + --+-+-((-1 * +(8 - 5 - 6)) * (-((-+(((+(4))))) - ++4) /
+(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6) * ---+6)) / +--(+-+-7 * (-0 *
(+(((((2)) + 8 - 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1) / 482) / +++-
+0)))) * -+5 + 7.513)))) - ...
```

Arithmetic expression

- **Symbolic execution**: Collect “symbolic” path constraints and solve them to systematically explore all “program paths”. Need rigorous program analysis/very expensive.
- **Gray-box fuzzing**: Instrument code to capture basic blocks covered; Optimization problem: generate inputs to maximize coverage

Test Oracles and Alternatives

- Test Oracle: A mechanism for determining what the expected output is for a given input: ***What is the specification for the code?***
- **Typical oracles**: crashes, hang/infinite loop, buffer overflows/use-after-free (AddressSanitizer), race conditions (ThreadSanitizers), …
- Alternatives (when oracles/specs are not available):
  - **Differential Testing**: Run same input across multiple implementations of same algorithm (E.g., render same web page with Chrome and Firefox)
  - **Metamorphic Testing**: For a given change in inputs, does the output change *as expected* for the same program? (e.g, $\sin(x+2\pi) = \sin(x)$)
- **Property-Based Testing**: Use properties/specs that hold over any input. Then generate random inputs to test the property.
  - **Examples**: (Push(Stack, X) => Pop(Stack)) == X, "*Sorting should preserve order of elements with equal*" keys (stability), No duplicates in Binary Search Tree

What Have We Learned?

- Random testing is effective for automatically finding bugs
- Should complement not replace systematic, formal testing
- Must generate test inputs from a reasonable distribution in order to be effective (e.g., using grammars)
- Specifying test oracles remains a hard challenge