Software Analysis Problem 7 Submit

tags: `class`

1. What is the major difference between fuzzing and mutation testing?

Fuzzing is a technique that automatically generate invalid or random inputs to a program, whereas mutation testing involves making small changes to the program to be tested to evaluate the quality of the quality of existing test cases. A high quality test case should cause mutants to fail while only allowing the original version to pass.

2. Given the following program and the resources for only 6 mutations of the program, what statements would you mutate, how and why?

```
int gcd(int a, int b)
{
  // Everything divides 0
  if (a == 0)
    return b;
  if (b == 0)
    return a;

  // base case
  if (a == b)
    return a;

  // a is greater
  if (a > b)
    return gcd(a-b, b);

  return gcd(a, b-a);
}
```

`return gcd(a-b, b)` or `return gcd(a, b-a)` would be a good place to change. For example, we can change `a-b` to `b-a`. With mutation testing, the goal is to identify weakly tested parts of code. The recursive part is the most error-prone part and is more likely to not have full testing coverage. Therefore, mutating these 2 statement will be more conducive to evaluating tests.

3. How would you use delta debugging in the following two strings

a. Split the string in half and see which half causes a failure. Keep narrowing down the range with this process.
b. Split the string in half and see which half causes a failure. Keep narrowing down the range with this process.

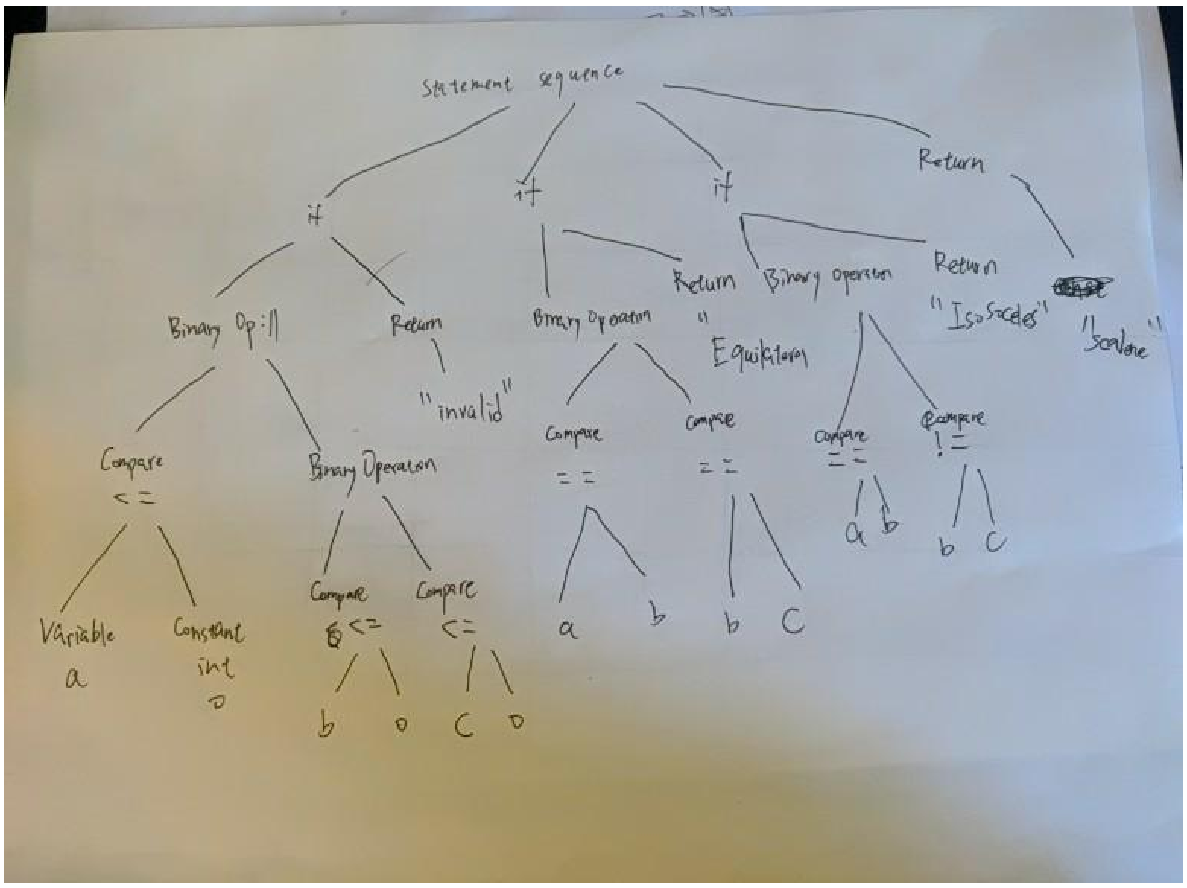
4. Give a concrete example of a problem that model checking would be good to use for testing.

Vending machines are a good example that can be tested with model checking since it is essentially a finite state machine.

5. Relate program repair to patch verification. That is, how would you use these two techniques.

I would use first automatic program repair techniques and then use patch verification to check if the generated repair is correct.

6. The following program has a bug. How would you repair it using the GenProg technique presented in lecture. Show the AST and the repair



```
int triangle (int a, int b, int c) {
    if(a <= 0 || b <= 0 || c <= 0)
        return INVALID;
    if(a == b && b == c)
        return EQUILATERAL;
    if (a == b || b != c)
        return ISOSOCELES
    return SCALENE;
}
```

To automatically repair this program, we must first codify desired behavior. The test cases are as follows:

1. (a, b, c) = (-3, 1, 1)->return: INVALID
2. (a, b, c) = (5, 5, 5)-> return: EQUILATERAL
3. (a, b, c) = (4, 4, 5)-> return: ISOSOCELES
4. (a, b, c) = (3, 5, 5), return: ISOSOCELES
5. (a, b, c) = (3, 4, 5), return: SCALENE

First, we will run the test cases to see if the program returns correct values. GenProg will find that test case 4 is incorrect. Next, GenProg performs *fault localization* strategy to reduce the search space. Test case 4 demonstrating the error visits lines 1-2, 4, and 6-7. Mutation and crossover operations are therefore focused on lines 6-7 which is the if statement for checking ISOSOCELES. To further constrain the search, GenProg has built-in assumption that most defects can be repaired by adapting existing code from another location in the program, which means a program that makes a mistake in one location often handles a similar situation correctly in another. Although the ISOSOCELES check does not do a correct check on triangle's sides of length, the EQUILATERAL if statement does: `if(a == b && b == c)`. Fault localization biases the modifications toward ISOSOCELES. The restriction to use only existing code for insertions further limits the search, and eventually GenProg tries inserting the check from EQUILATERAL into ISOSOCELES. With this version of ISOSOCELES if statement check passes all 5 test cases.

7. Describe the process as to how AFL and KLEE generate inputs respectively

AFL

AFL will remove the unecessary parts of the test case by removing code in such a way that the behavior the of program will not be affected by the removal. Then, it will mutate the file repeatedly using a variety of fuzzing techniques. If any of the generated mutations resulted in a new state transition recorded by the instrumentation, AFL will add mutated output as a new entry in the queue.

KLEE

KLEE solves the current path’s constraints (called its path condition) to produce a test case when it detects an error or when a path reaches an exit call. The test case will follow the same path when rerun on an unmodified version of the checked program.

8. What can an MVICFG do that an ICFG cannot?

- Enables efficient, precise program verification.
- Allows users to see the changes in program paths and program behaviors
- Correlating multiple program versions. Using MVICFGs

Comparison

An MVICFG is a union of a set of ICFGs for program versions. MVICFG integrates and compares control flow of multiple versions of programs. It is a demand-driven and path sensitive symbolic analysis that traverses the MVICFG for detecting bugs related to software changes and versions.

ICFG is representing control flow for a program. It is a graph that combines CFGs of all program procedures by connecting procedure entries and exits with their call sites. Each procedure can have multiple procedure entry nodes and multiple procedure exit nodes. The successors of a call site node are the procedure entry node, and the associated call site exit nodes.