Rigorous Software Engineering- Week 3 (Lectures)

- Author: Ruben Schenk
- Date: 15.03.2021
- Contact: ruben.schenk@inf.ethz.ch

5.3.3 Combinatorial Testing

Combining equivalence classes and boundary testing leads to many values for each input to test. This can lead to a "combinatorial explosion" when testing all possible combinations.

We want to reduce test cases to make the effort feasible through:

- Semantic constraints
- Combinatorial selection
- Random selection

Semantic Constraints

Through eliminating combinations we might be able to reduce the number of possible test cases. Elimination is done by inspecting test cases for unnecessary combinations:

- Especially for invalid values
- Use problem domain knowledge

However, especially when there are many input values, for instance, for the fields of objects, too many combinations still remain.

Combinatorial Selection

One might focuses on all `possible combinations` of each pair of inputs isntead on all possible combinations of all inputs.

Example: Consider a method with four boolean parameters:

- Combinatorial testing requires $2^4 = 16$ test cases
- Pairwise-combinations testing requires only 5 test cases: TTTT, TFFF, FTFF, FFTF, FFFTT

I.e. in `pariwise-comninaions` we look that each pair of variables appear in all possible combinations, e.g. in our previous case TT, TF, FT, and FF.

Pairwise-combinations testing reduces the number of test cases significantly while detecting most errors. It should however be combined with other approaches to detect errors that are triggered by more complex interactions among parameters.

5.4 Structural Testing

We first look at a motivating example: Given a non-null array of integers, sort the array in-place in ascending order.

```
public void sort(int[] a) {
    if(a == null || a.length < 2) {
        // array is trivially sorted
        return;
    }
    // check if array is already sorted
    for(int i = 0; i < a.length - 1; i++) {
        if(a[i] > a[i+1]) {
            break;
        }
    }
    if(i >= a.length - 1) {
        // array is already sorted
        return;
    }
    // use quicksort to sort array
    ...
}
```

Basic Blocks

A `basic block` is a sequence of statements such that the code in a basic block:

- has `one entry point`: no code within it is the destination of a jump instruction anywhere in the program
- has `one exit point`: only the last instruction causes the program to execute code in a different basic block

Furthermore, whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once, in order.

Intraprocedural Control Flow Graphs

An `intraprocedural control flow graph` (CFG) of a procedure p is a graph $G = (N, E)$ where:

- N is the set of basic blocks in p plus designated entry and exit blocks
- E contains:
  - an edge from a to b with condition c iff. the execution of basic block a is succeeded by the execution of basic block b if condition c holds
  - and edge $(entry,\ a,\ true)$ if a is the first basic block of p
  - edges $(b,\ exit,\ true)$ for each basic block b that ends with a return statement

The CFG can serve as an adequacy criterion for test cases. The more parts are executed, the higher the chance to uncover a bug.

Statement Coverage

We can asses the quality of a test suite by measuring how much of the CFG it executes, i.e. number of executed statements over the total number of statements gives you the `statement coverage`.

Branch Coverage

The idea is to test all possible branches in the control flow. An edge (m, n, c) in a CFG is a `branch` iff. there is another edge (m, n', c') in the CFG with $n \neq n'$. We define the `branch coverage` as the number of executed branches over the total number of branches.

Complete branch coverage implies complete statement coverage. It is the most widely used adequacy criterion in the industry.

Path Coverage

The idea is to test all possible paths thorugh the CFG. A `path` is a sequence of nodes $n_1, \ldots, n_k$ such that n_1 is an entry and n_k is an exit. We define the `path coverage` as the number of executed paths over the total number of paths.

Complete path coverage implies complete statement coverage and complete branch coverage. However, complete path coverage is not feasible for loops.

Loop Coverage

The idea is to test for each loop zero, one, and more that one consecutive iterations. The the `loop coverage` is given by the number of executed loops with 0, 1, and more that 2 iteration divided by the total number of loops * 3.