White Box Testing

Based on the implementation

Why do white box testing?

- Black box testing based on requirements
  - requirements are rarely complete and accurate
- White box test cases may reveal problems with design and code decisions

Static Analysis versus Dynamic Analysis

- Dynamic analysis evaluates the execution of the program
- Static analysis evaluates the program without executing it
  - similar to compile time analysis
  - extensive type checking grew out of static analysis
  - cross reference table is derived during static analysis

Dynamic Analyses

- Assertions
- Coverage analysis
  - Statement coverage
    - Execute each source statement at least once
  - Branch coverage
    - Execute each branch of each conditional
  - Condition coverage
    - Execute the terms of a condition
  - Path coverage
  - Loop coverage
  - Data Flow Coverage
    - Execute the flow of data
  - Error seeding

Assertions

- Invariant statements about the code
- Automatically checked at run time
- If found to be false, default error message is raised
  - E.g., ASSERT (for all i, 1 ≤ i ≤ 10, a[i] < 100)
- Often have powerful operators
- Local assertions are true whenever they are encountered during execution
- Global assertions are true throughout the scope for which they are defined
  - Compiler will determine when the assertion must be checked

Coverage Analysis

- Based on a flow graph model of the program
  - Derived from source code
  - Nodes are statements or statement fragments
  - Edges represent potential flow of control
Example Flow Graph

Statement and Branch Coverage

- Design test cases so that each statement/branch is traversed at least once
- Can’t find a fault in a statement unless it is executed
  - may not find the fault because of coincidental correctness
  - coincidental correctness is very common
  - otherwise statement coverage would be a very effective technique

Branch coverage subsumes statement coverage

Coverage in Condition Testing

- Branch coverage does not guarantee that every term in the condition is exercised
  - \((a < b) \lor (d > 1)\)
  - \(T\) \(F\)
  - \(F\) \(F\)
  - impact of \((d > 1)\) is never evaluated
  - called hidden paths

Better Condition Testing

- Minimally, test every simple condition
  - \((a < b) \lor (d > 1)\)
  - \(T\) \(F\)
  - \(F\) \(T\)
  - \(F\) \(F\)
  - Better, test all combinations of true and false for each simple term in a compound conditional
    - \(T\) \(T\)
    - \(T\) \(F\)
    - \(F\) \(T\)
    - \(F\) \(F\)

Test relational operators

- Satisfy the relation by the smallest amount
  - Do not satisfy the relation by the smallest amount

  - example: \((d > 1)\)
    - \(d = 2\)
    - \(d = 1\)
Path Coverage

Potentially, an infinite # of paths

Approximating path coverage

- test all acyclic paths
- test all paths up to one iteration of each loop
- various heuristics for how to select paths in loops

- even if a path is tested and returns correct results, it may not be correct for all executions of that path
  - coincidental correctness

Data Flow Testing

- Exercises paths between the definition and use of a variable

Exercising data flow relationships

Data Structure testing

- Test boundary conditions
  - Example: linked list
    - Empty
    - 1 element
    - Typical elements
    - Maximum elements
      (if there is a max)
  - Be sure to handle amount of data specified in requirements

How do you know when to stop testing?

- when number of failures is low?
- when resources have expired?
- when alpha testers are not finding problems?

- test data adequacy measures the adequacy of your testing
  - not the adequacy of your code
Coverage monitoring

- instrument the code with probes
- measure the percentage of coverage that has been achieved with the test data
- in well tested systems:
  - 85% statement/branch coverage
  - dead code and lack of resources prevents 100% coverage
  - 50–70% data flow coverage
  - not all data flow can be executed
  - infeasible paths in the control flow graph

Automated testing tools

- White box coverage needs to be supported by automated tools
  - Parasoft, rational, SRA, etc.

Error seeding

- another method for evaluating the quality of your testing
- insert N "typical" bugs into the program
- see how many of these bugs are found by the test data
- adequacy = found bugs/seeded bugs

Summary

- Testing CANNOT prove that a program is correct
- Testing does not improve the quality of your code, but demonstrates the quality of your code
- Testing artifacts are important assets
- White box testing concentrates on implementation decisions
  - many white box techniques are dynamic analysis techniques that evaluate the quality of the testing
- Need to have a combination of black box, white box, and manual reviews