Structural Testing (White-box Testing)

SWE 261P

“Structural” testing

- Judging test suite thoroughness based on the *structure* of the program itself
  - Also known as “white-box”, “glass-box”, or “code-based” testing
  - To distinguish from functional (requirements-based, “black-box” testing)
    - “Structural” testing is still testing product functionality against its specification. Only the measure of thoroughness has changed.

Why structural (code-based) testing?

- One way of answering the question "What is *missing* in our test suite?"
  - If part of a program is not executed by any test case in the suite, faults in that part cannot be exposed
  - But what's a "part"?
    - Typically, a control flow element or combination:
    - Statements (or CFG nodes), Branches (or CFG edges)
    - Fragments and combinations: Conditions, paths
    - "Flows" of data
- Complements functional testing: Another way to recognize cases that are treated differently

No guarantees

- Executing all control flow elements does not guarantee finding all faults
  - Execution of a faulty statement may not always result in a failure
    - Why?

No guarantees

- Executing all control flow elements does not guarantee finding all faults
  - Execution of a faulty statement may not always result in a failure
    - The state may not be corrupted when the statement is executed with some data values
    - Corrupt state may not propagate through execution to eventually lead to failure
    - The bug may involve multiple parts of the program interacting
- So... what is the value of structural coverage?
  - Increases confidence in thoroughness of testing
    - Removes some obvious *inadequacies*

Structural testing *complements* functional testing

- Control flow testing includes cases that may not be identified from specifications alone
  - Typical case: implementation of a single item of the specification by multiple parts of the program
  - Example: hash table collision (invisible in interface spec)
- Test suites that satisfy control flow adequacy criteria could fail in revealing faults that can be caught with functional criteria
  - Typical case: missing path faults

Structural testing in practice

- Create functional test suite first, then measure structural coverage to identify what is missing
- Interpret unexecuted elements
  - may be due to natural differences between specification and implementation
  - or may reveal flaws of the software or its development process
    - inadequacy of specifications that do not include cases present in the implementation
    - coding practice that radically diverges from the specification
    - inadequate functional test suites

- Attractive because automated
  - coverage measurements are convenient progress indicators
  - sometimes used as a criterion of completion
    - use with caution: does not ensure *effective* test suites

Test-Adequacy Criteria

Statement testing

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once
- Coverage:

$$\frac{\#\ \text{executed statements}}{\#\ \text{executable statements}}$$

goal: equal

- Rationale: a fault in a statement can only be revealed by executing the faulty statement

Statements or blocks?

- Nodes in a control flow graph often represent basic blocks of multiple statements
  - Some standards refer to *basic block* coverage or *node coverage*
  - Difference in granularity, not in concept
- No essential difference
  - 100% node coverage == 100% statement coverage (and 0%==0%)
    - but levels will likely differ in between
  - A test case that improves one will improve the other
    - although perhaps not by the same amount

"Satisfying" test adequacy criteria

- A test-adequacy criterion is satisfied when
  - All elements of the criterion have been executed by the specified number of test cases
    - (typically, this means at least one test case... for example, for statement coverage, all statements have been executed by at least one test case)
  - All test cases pass (at least, this is by the formal definitions of test-adequacy criteria)

Coverage is not size

- Coverage does not strictly depend on the number of test cases (although, there is often a correlation in practice)
- Coverage adequacy is often attained by adding test cases until the criterion is satisfied
- Minimizing test suite size is seldom the goal
  - small test cases make failure diagnosis easier
  - a failing test case in $T_{size=2}$ gives more information for fault localization than a failing test case in $T_{size=1}$

Branch testing

- Adequacy criterion: each branch (labeled edge in the CFG) must be executed at least once
- Coverage:

$$\frac{\#\ \text{executed branches}}{\#\ \text{branches}}$$

100% branch → 100% statement,reverse is not sure

Statements vs branches

- Traversing all edges of a graph causes all nodes to be visited
  - So test suites that satisfy the branch adequacy criterion for a program P also satisfy the statement adequacy criterion for the same program — the branch coverage criterion "subsumes" the statement coverage criterion
- The converse is not true
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)

```
if (i < 0) {
    do something;
}
i++;
```

“All branches” can still miss conditions

- Example:

  a > 0 && b >= 0

- Branch adequacy criterion could be satisfied by varying only b
  - A faulty sub-expression might never determine the result
  - We might never really test the faulty condition, even though we tested both outcomes of the branch

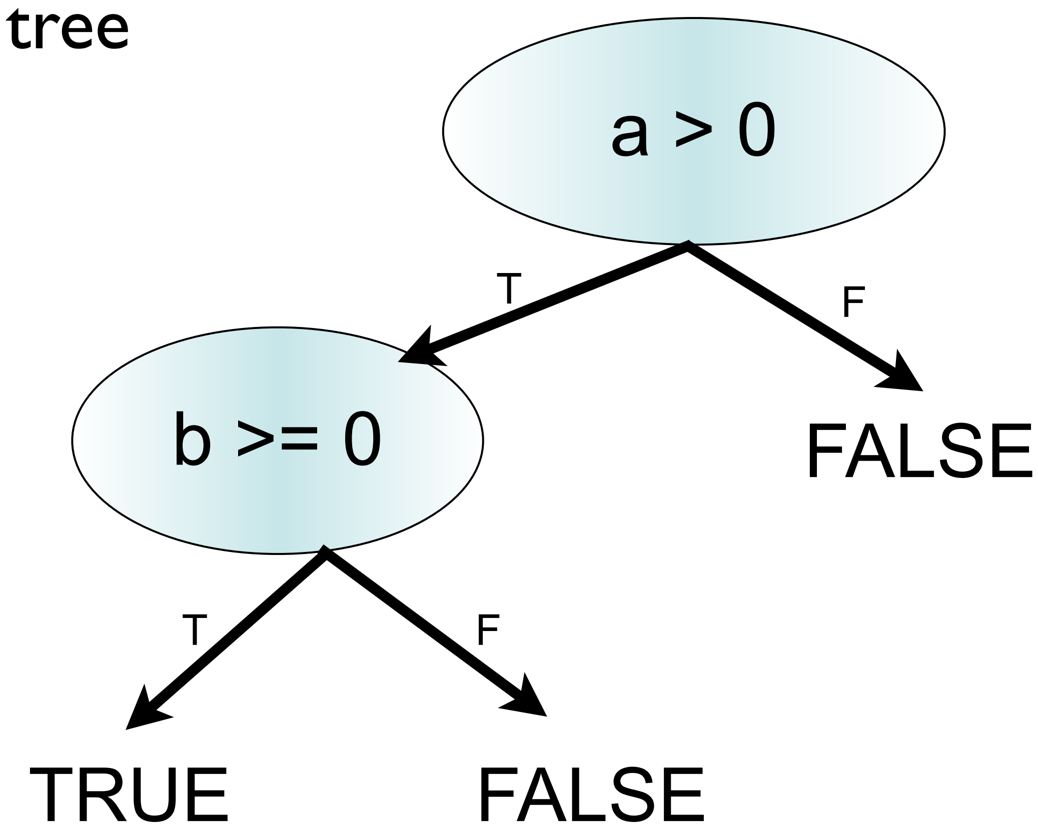
branch is stronger than statement

Condition testing

- Branch coverage exposes faults in how a computation has been decomposed into cases
  - intuitively attractive: check the programmer's case analysis
  - but only roughly: groups cases with the same outcome
- Condition coverage considers case analysis in more detail
  - also *individual conditions* in a compound Boolean expression
    - e.g., both parts of `a > 0 && b >= 0`

Covering branches and conditions

- Branch and condition adequacy:
  - cover all conditions and all decisions
- Compound condition adequacy:
  - Cover all possible evaluations of compound conditions
  - Cover all branches of a decision tree



Compound conditions: Exponential complexity

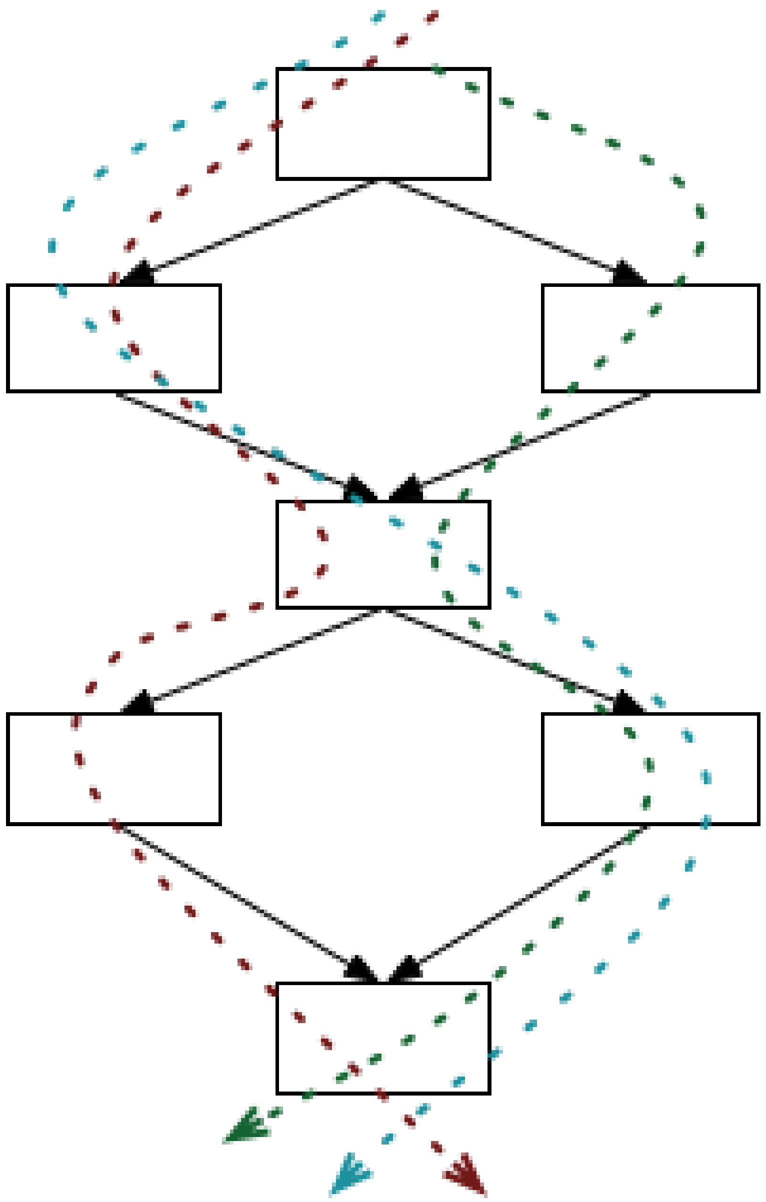
`(((a || b) && c) || d) && e`

Test Case	a	b	c	d	e
(1)	T	—	T	—	T
(2)	F	T	T	—	T
(3)	T	—	F	T	T
(4)	F	T	F	T	T
(5)	F	F	—	T	T
(6)	T	—	T	—	F
(7)	F	T	T	—	F
(8)	T	—	F	T	F
(9)	F	T	F	T	F
(10)	F	F	—	T	F
(11)	T	—	F	F	—
(12)	F	T	F	F	—
(13)	F	F	—	F	—

- short-circuit evaluation often reduces this to a more manageable number, but not always

Paths? (Beyond individual branches)

it's harder to achieve, in some program can't



- Should we explore sequences of branches (paths) in the control flow?
- Many more paths than branches
  - A pragmatic compromise will be needed

Method-level testing

- Method coverage
  - each method is executed at least once
  - comes "for free" with unit testing
- Method-call coverage
  - each method may be called from multiple call sites, and each call site can call many method
- Method exit testing
  - method may have multiple exit points (multiple returns, exception exit points)

Data-Flow Testing

- All-Defs
- All-Uses
- All-C-Uses/Some-P-Uses
- All-P-Uses/Some-C-Uses
- All-DU-Paths
- . . . (many, many data-flow criteria and their relationships described in the Clarke, Podgurski, Richardson, Zeil paper "A Comparison of Data Flow Path Selection Criteria")

Satisfying structural criteria

in some company, 70-80%

- Sometimes criteria may not be satisfiable
  - The criterion requires execution of
    - statements that cannot be executed as a result of
      - defensive programming
      - code reuse (reusing code that is more general than strictly required for the application)
    - conditions that cannot be satisfied as a result of
      - interdependent conditions
    - paths that cannot be executed as a result of
      - interdependent decisions

Satisfying structural criteria

- Large amounts of dead code may indicate serious maintainability problems
  - But some unreachable code is common even in well-designed, well-maintained systems
- Solutions:
  - make allowances by setting a coverage goal less than 100%
  - require justification of elements left uncovered
    - This is a requirement of the FAA for commercial airplane systems (i.e., RTCA-DO-178B and EUROCAE ED-12B for modified MC/DC)
  - refactor and eliminate the dead code