Structural Testing

Dr. Paul West

Department of Computer Science
College of Charleston

February 27, 2014

“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?

- 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
- Complements functional testing: Another way to recognize cases that are treated differently
  - Recall fundamental rationale: Prefer test cases that are treated differently over cases treated the same

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
- 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 see 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

Statement Testing

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once
- Coverage: $\frac{number\ of\ executed\ statements}{number\ of\ statements}$
- Rationale: a fault in a statement can only be revealed by executing the faulty statement

Statements or blocks?

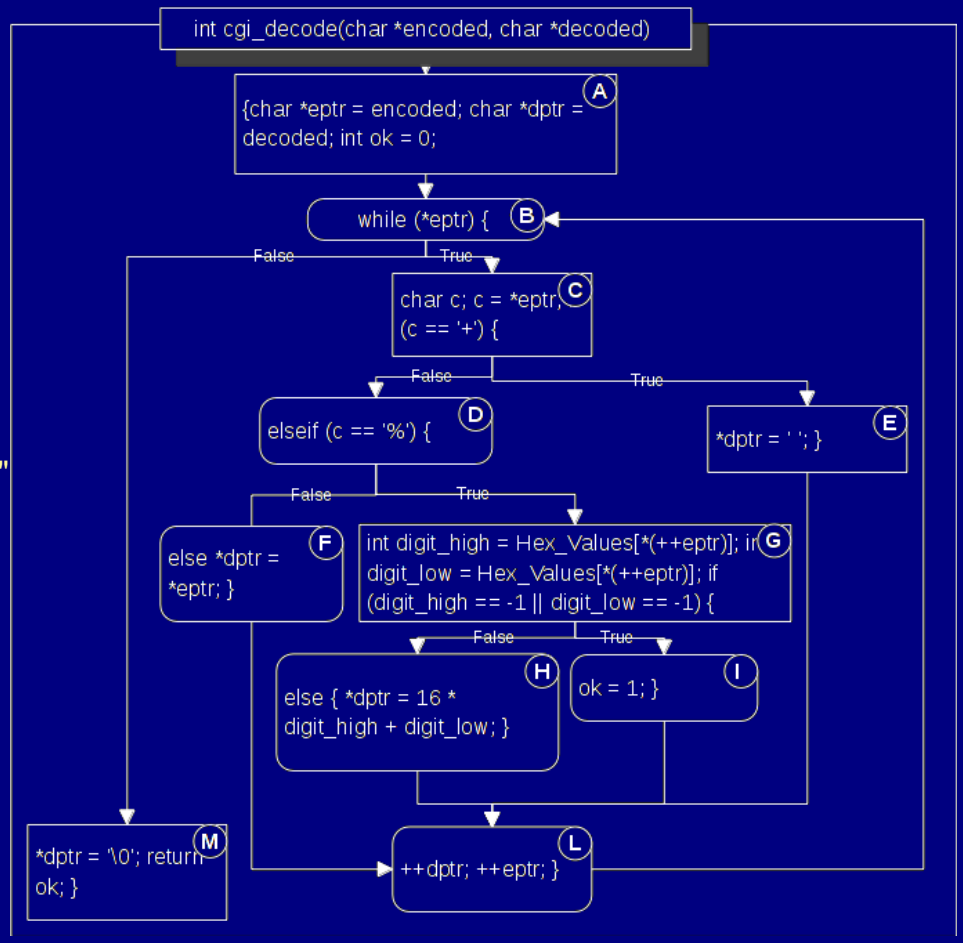
- Nodes in a control flow graph often represent basic blocks of multiple statements
  - Some standards refer to basic block coverage or node coverage
  - Difference in granularity, not in concept
- No essential difference
  - 100% node coverage <-> 100% statement coverage
    - but levels will differ below 100%
  - A test case that improves one will improve the other
    - though not by the same amount, in general

Example

```
T0 =
{"", "test",
"test+case%1Dadequacy"}
17/18 = 94% Stmt Cov.

T1 =
{"adequate+test%0Dexecution%7U"
18/18 = 100% Stmt Cov.

T2 =
{"%3D", "%A", "a+b",
"test"}
18/18 = 100% Stmt Cov.
```

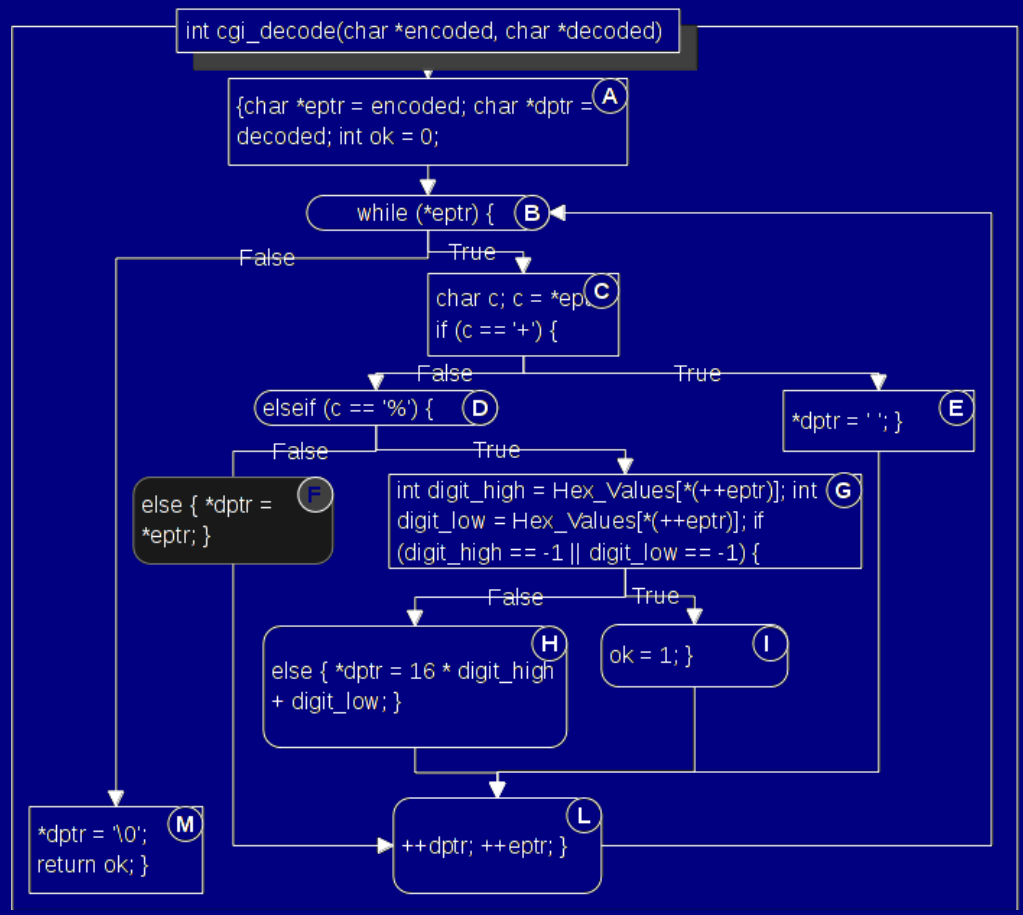


Coverage is Not Size

- Coverage does not depend on the number of test cases
  - T0 , T1 : T1 >coverage T0 T1 <cardinality T0
  - T1 , T2 : T2 =coverage T1 T2 >cardinality T1
- Minimizing test suite size is seldom the goal
  - small test cases make failure diagnosis easier
  - a failing test case in T2 gives more information for fault localization than a failing test case in T1

“All Statements” can miss some cases

- Complete statement coverage may not imply executing all branches in a program
- Example:
  - Suppose block F was missing
  - Statement adequacy would not require false branch from D to L




```
T3 =
{"", "+%0D+%4J"}
100% Stmt Cov.
No false branch from D
```

Branch Testing

- Adequacy criterion: each branch (edge in the CFG) must be executed at least once
- Coverage: $\frac{\#executed\ branches}{\#branches}$

```
T3 = {"", "+%0D+%4J"}
100% Stmt Cov. 88% Branch Cov. (7/8 branches)

T2 = {"%3D", "%A", "a+b", "test"}
100% Stmt Cov.  100% Branch Cov. (8/8 branches)
```

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 converse is not true (see T3)
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)

“All Branches” Can Still Miss Conditions

- Sample fault: missing operator (negation)
  digit_high == 1 || digit_low == -1
- Branch adequacy criterion can be satisfied by varying only digit_low
  - The 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