Structural Testing

Learning objectives

- Understand rationale for structural testing
  - How structural (code-based or glass-box) testing complements functional (black-box) testing
- Recognize and distinguish basic terms
  - Adequacy, coverage
- Recognize and distinguish characteristics of common structural criteria
- Understand practical uses and limitations of structural testing

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 can still test product functionality against its specification.
    - But include test cases that may not be identified from specifications alone.
    - The measure of thoroughness (i.e., adequacy criteria) 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
- 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 (e.g., protection mechanism)
- 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

- 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
  - 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:

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

- 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
  - A block has a single entry and a single exit

- Correspondence
  - 100% node coverage <-> 100% statement coverage

Example

```
#include "hex_values.h"
/**
 * @title cgi_decode
 * @desc
 *     Translate a string from the CGI encoding to plain ascii text
 *      '+' becomes space, %xx becomes byte with hex value xx,
 *      other alphanumeric characters map to themselves
 *
 *     returns 0 for success, positive for erroneous input
 *          1 = bad hexadecimal digit
 */
int  cgi_decode(char *encoded, char *decoded) {
   char *eptr = encoded;
   char *dptr = decoded;
   int ok=0;
   while (*eptr) {
      char c;
      c = *eptr;
      /* Case 1:  '+' maps to blank */
      if (c == '+') {
         *dptr = ' ';
      } else if (c == '%') {
         /* Case 2:  '%xx' is hex for character xx */
         int   digit_high = Hex_Values[*(++eptr)];
         int   digit_low   = Hex_Values[*(++eptr)];
         /* Hex_Values maps illegal digits to -1 */
         if (    digit_high == -1 || digit_low == -1 ) {
            /* *dptr='?'; */
            ok=1; /* Bad return code */
         } else {
            *dptr = 16* digit_high + digit_low;
         }
         /* Case 3:  All other characters map to themselves */
      } else {
         *dptr = *eptr;
      }
      ++dptr;
      ++eptr;
   }
   *dptr = '\0';                  /* Null terminator for string */
   return ok;
}
```

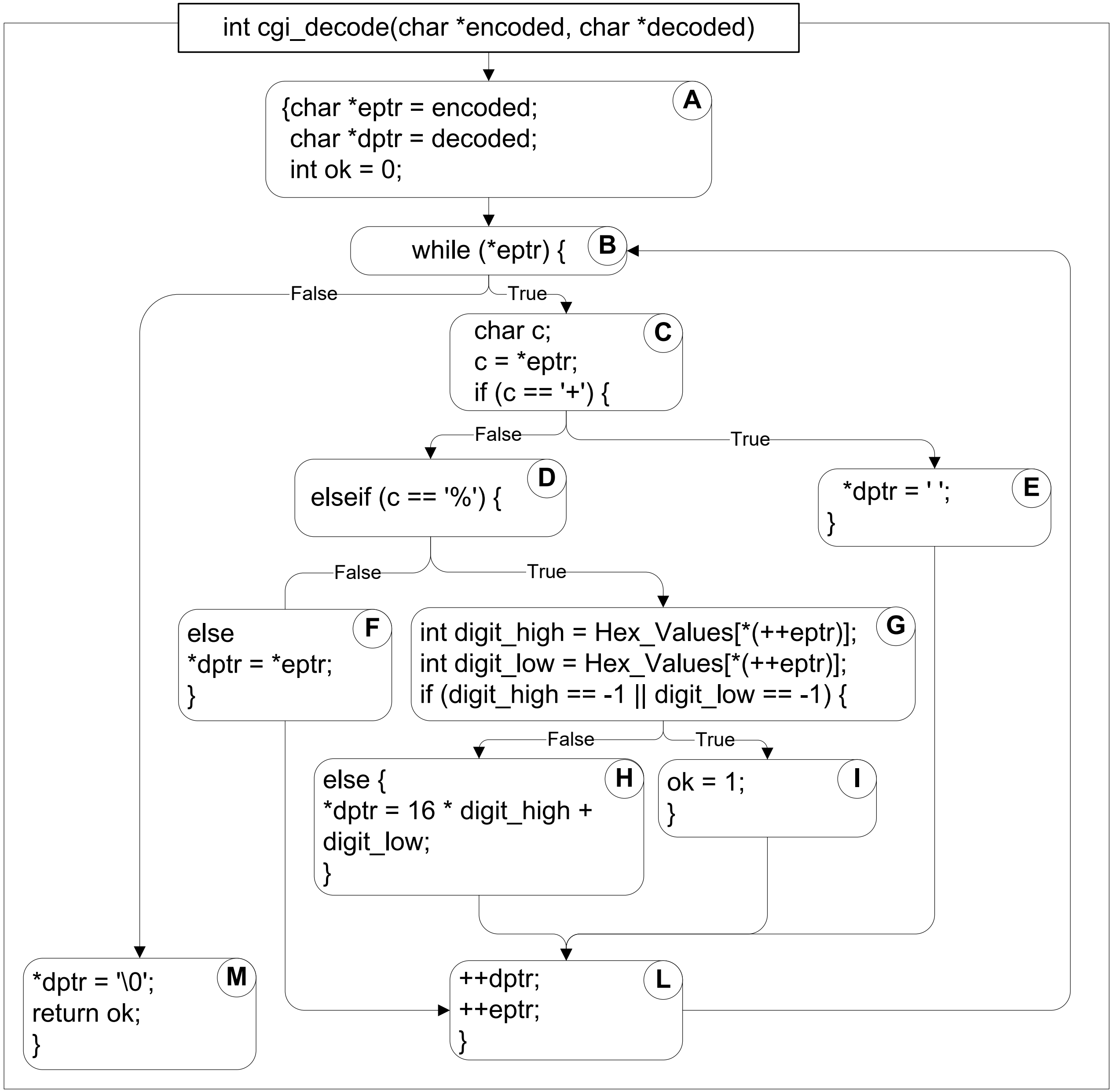
(c) 2007 Mauro Pezzè & Michal Young

Example

T_0 =
{“”, “test”, “test+case%1Dadequacy”}
17/18 = 94% Stmt Cov.

T_1 =
{“adequate+test%0Dexecution%7U”}
18/18 = 100% Stmt Cov.

T_2 =
{“%3D”, “%A”, “a+b”, “test”}
18/18 = 100% Stmt Cov.

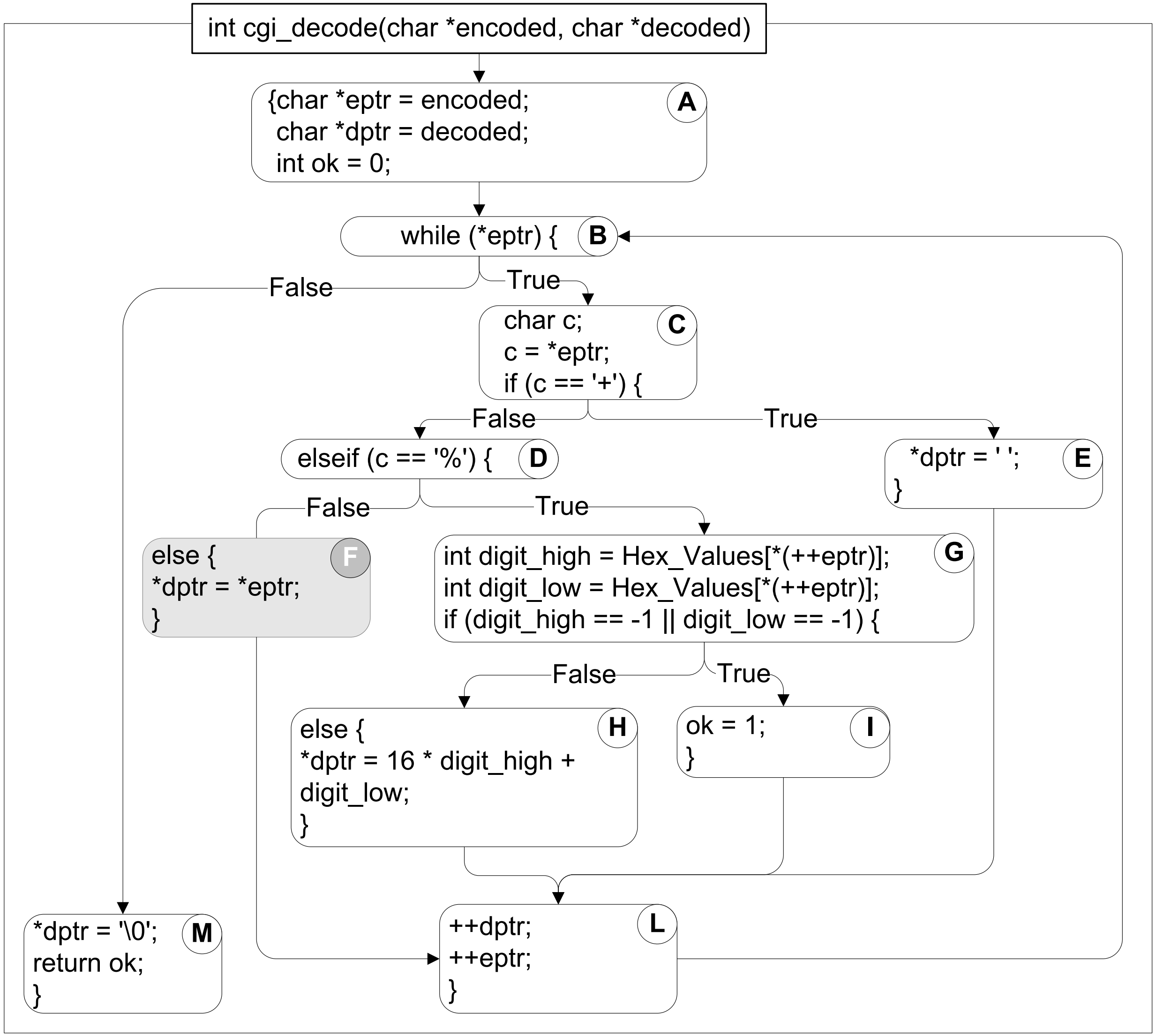


Coverage is not size

- Coverage does not depend on the number of test cases
  - $T_1 >_{coverage} T_0$ although T_1 contains more test cases than T_0
  - $T_2 =_{coverage} T_1$ although T_2 contains more test cases than T_1

"All statements" can miss some cases

- Complete statement coverage may not imply executing all branches in a program
- Example:
  - <u>**Suppose block F were taken out from the source code**</u>
  - Statement adequacy would not require *false* branch from D to L



T_3 =
{"", "+%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:

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

T_3 = {“”, “+%0D+%4J”}
100% Stmt Cov. 88% Branch Cov. (7/8 branches)

T_2 = {“%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 T_3)
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)

"All branches" can still miss conditions

- Sample fault: if line 27 was replaced by the following faulty statement (missing negation):

  `digit_high == 1 || digit_low == -1`

- Branch adequacy criterion could still 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

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., in “digit_high == 1 || digit_low == -1”
    - consider ”digit_high == 1 “, “digit_high != 1”, ”digit_low == -1 “, “digit_low != 1”

Basic condition testing

- Adequacy criterion: each basic condition must be executed at least once
- Coverage:

$$\frac{\text{\# truth values taken by all basic conditions}}{2 * \text{\# basic conditions}}$$

Basic conditions vs branches

- Basic condition adequacy criterion can be satisfied without satisfying branch coverage

T4 = {“first+test%9Ktest%K9”}

satisfies basic condition adequacy

does not satisfy branch condition adequacy

- “digit_high == -1 || digit_low == -1” is always true

Branch and basic condition are not comparable

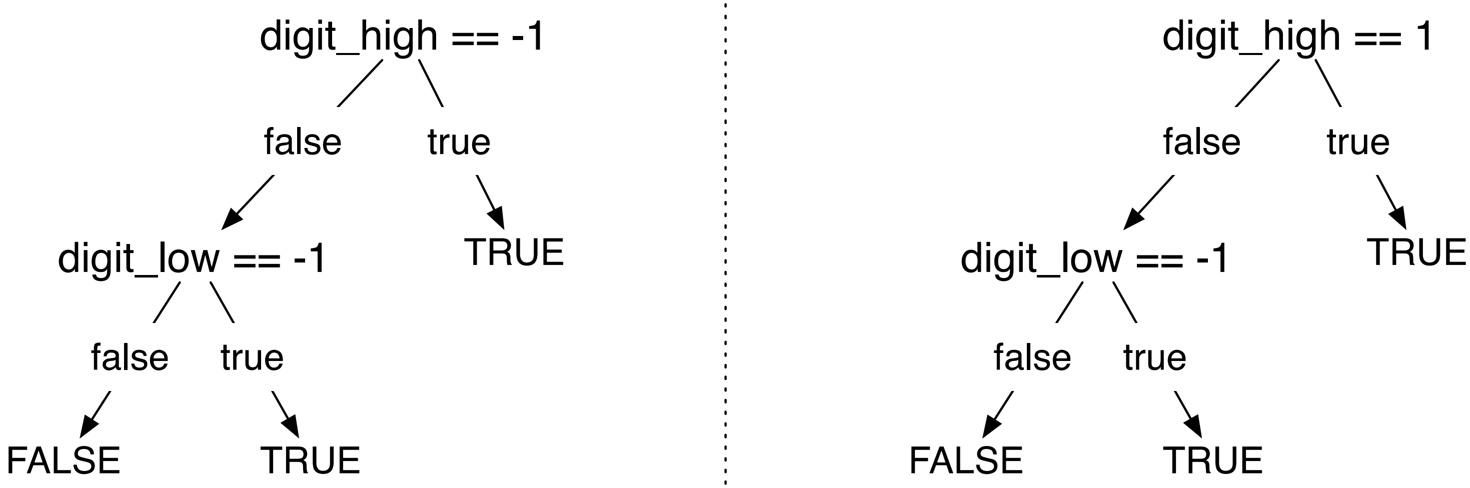
(neither implies the other)

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

Original version from page line 27, figure 12.1, line 27

With hypothesized fault from page 219, section 12.4



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

Path adequacy

- Decision and condition adequacy criteria consider individual program decisions
- Path testing focuses consider combinations of decisions along paths
- Adequacy criterion: each path must be executed at least once
- Coverage:

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

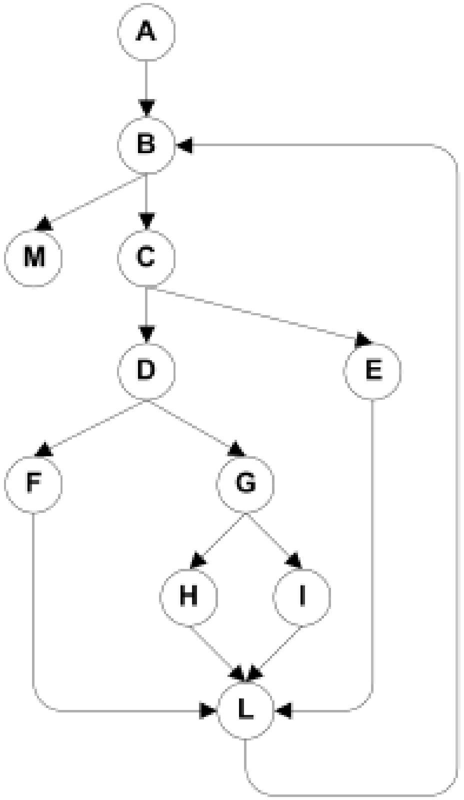
Practical path coverage criteria

- The number of paths in a program with loops is unbounded
  - the simple criterion is usually impossible to satisfy
- For a feasible criterion: Partition infinite set of paths into a finite number of classes
- Useful criteria can be obtained by limiting
  - the number of traversals of loops
  - the length of the paths to be traversed
  - the dependencies among selected paths

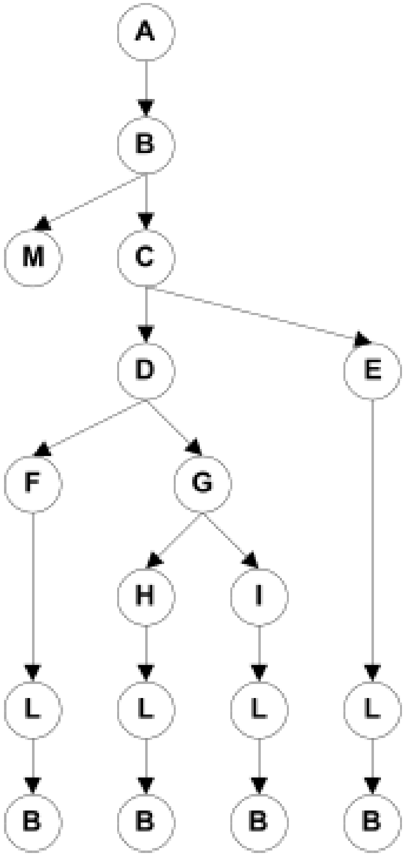
Boundary interior path testing

- Group together paths that differ only in the subpath they follow when repeating the body of a loop
  - Follow each path in the control flow graph up to the first repeated node
  - The set of paths from the root of the tree to each leaf is the required set of subpaths for boundary/interior coverage

Boundary interior adequacy for cgi-decode



(i)



(ii)

Limitations of boundary interior adequacy

- The number of paths can still grow exponentially

```
if (a) {
   S1;
}
if (b) {
   S2;
}
if (c) {
   S3;
}
...
if (x) {
   Sn;
}
```

- The subpaths through this control flow can include or exclude each of the statements Si, so that in total N branches result in 2^N paths that must be traversed
- Choosing input data to force execution of one particular path may be very difficult, or even impossible if the conditions are not independent

Loop boundary adequacy

- Variant of the boundary/interior criterion that treats loop boundaries similarly but is less stringent with respect to other differences among paths
- Criterion: A test suite satisfies the loop boundary adequacy criterion iff for every loop:
  - In at least one test case, the loop body is iterated zero times
  - In at least one test case, the loop body is iterated once
  - In at least one test case, the loop body is iterated more than once
- Corresponds to the cases that would be considered in a formal correctness proof for the loop

Satisfying structural criteria

- 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