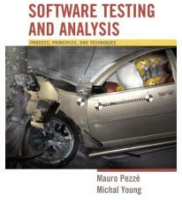
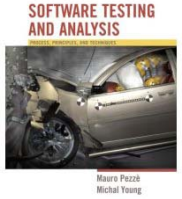
Structural Testing



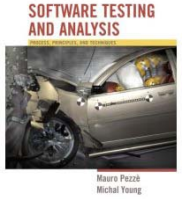
Learning objectives

- 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



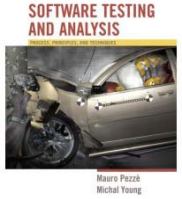
“Structural” testing

- 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 is still testing product functionality against its specification. Only the measure of thoroughness has changed.



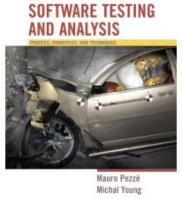
Why structural (code-based) testing?

- 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
- 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



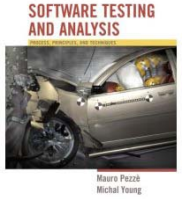
No guarantees

- 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*



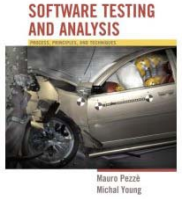
Structural testing *complements* functional testing

- 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

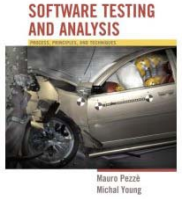


Statement testing

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

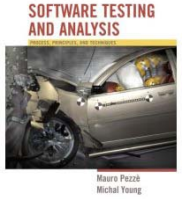
$$\frac{\text{\# executed statements}}{\text{\# statements}}$$

- 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
- 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

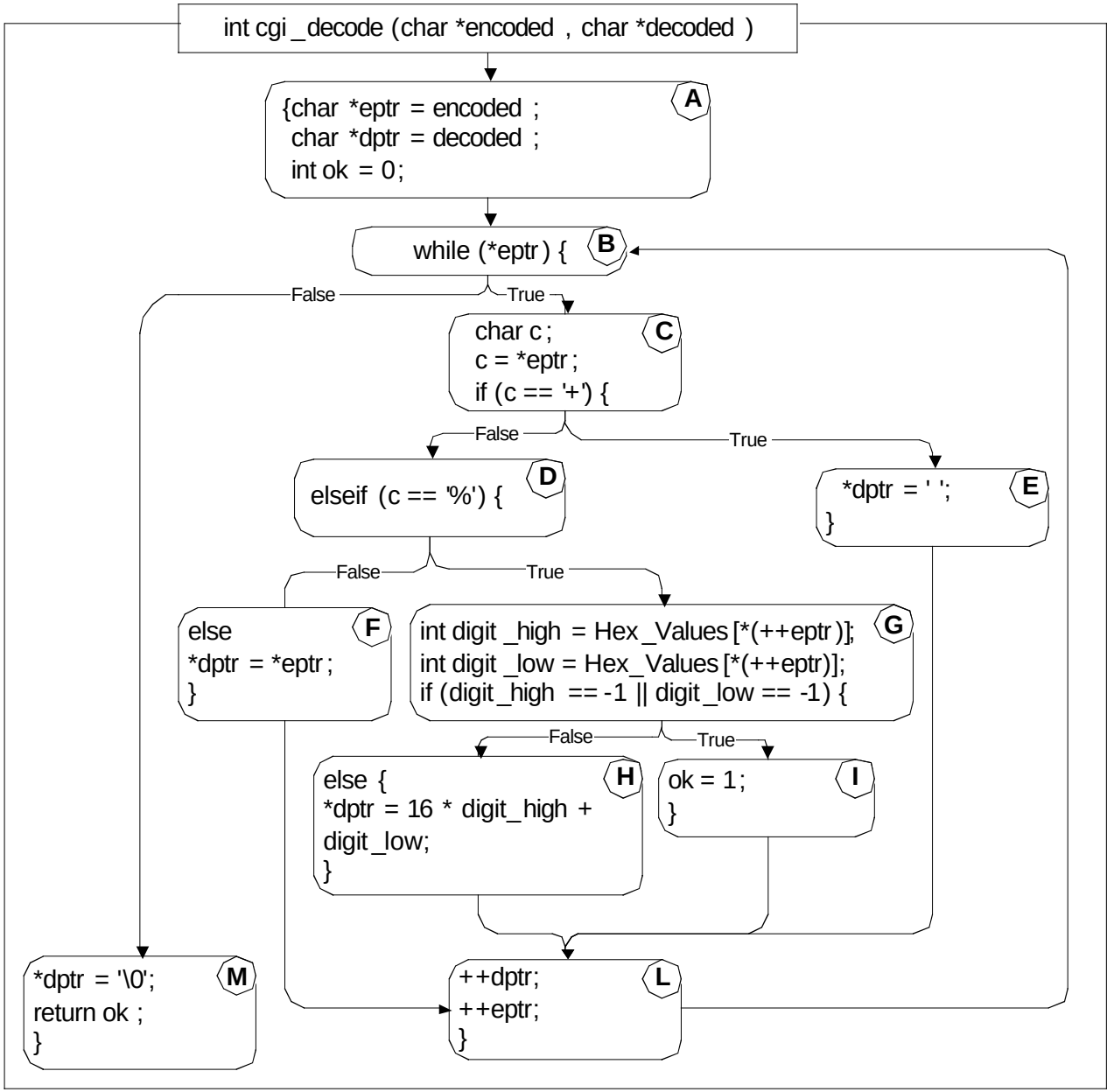


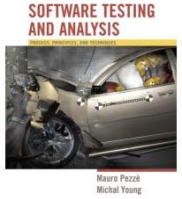
Example

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

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

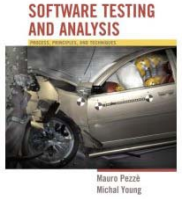
T_2 =
{“%3D”, “%A”, “a+b”,
“test”}
18/18 = 100% Stmt Cov.





Coverage is not size

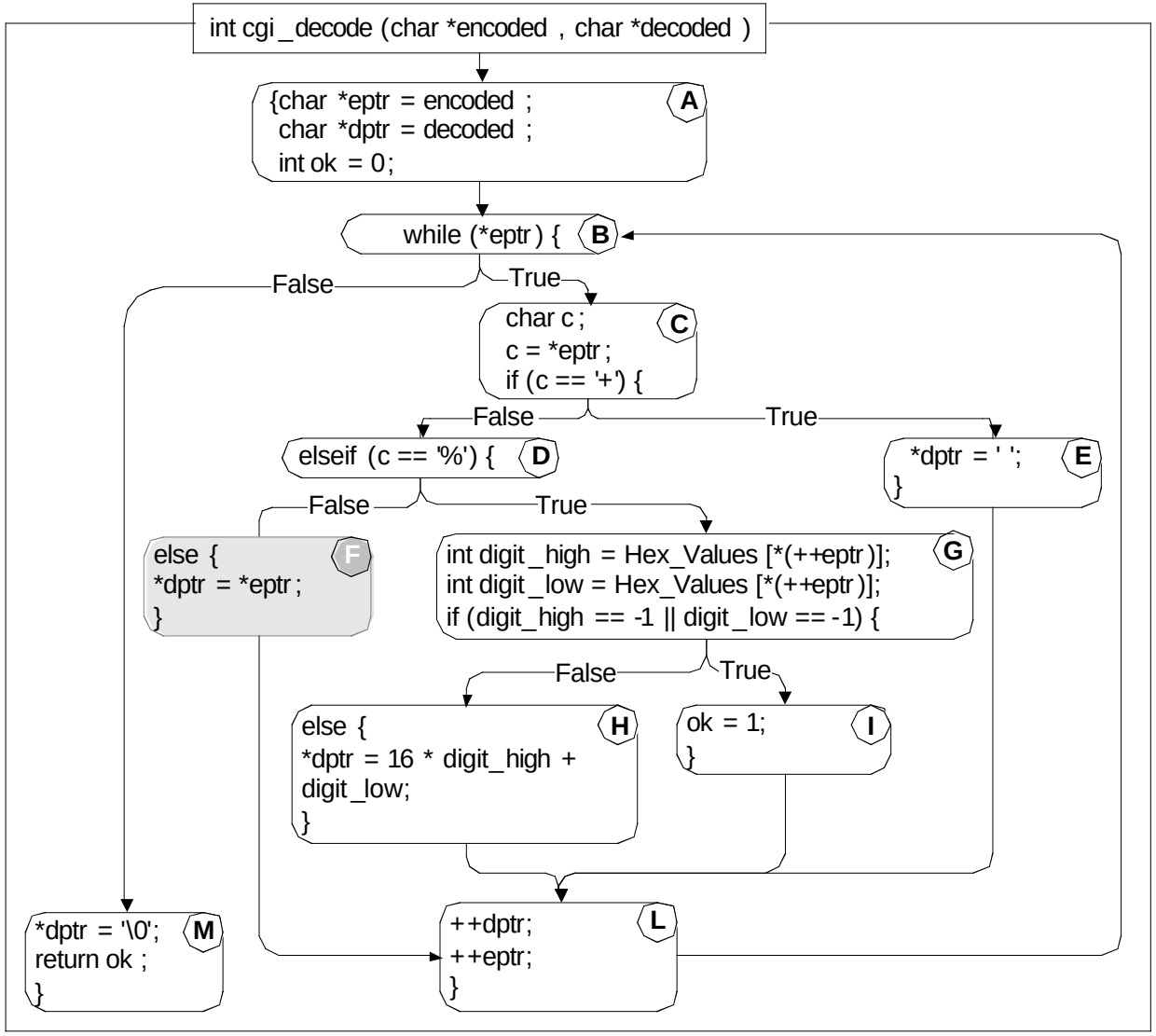
- Coverage does not depend on the number of test cases
  - $T_0, T_1 : T_1 >_{coverage} T_0$ $\quad T_1 <_{cardinality} T_0$
  - $T_1, T_2 : T_2 =_{coverage} T_1$ $\quad T_2 >_{cardinality} T_1$
- Minimizing test suite size is seldom the goal
  - small test cases make failure diagnosis easier
  - a failing test case in T_2 gives more information for fault localization than a failing test case in T_1

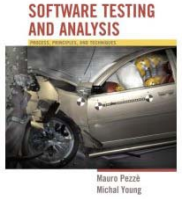


"All statements" can miss some cases

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

T_3 =
{"", "+%0D+%4J"}
100% Stmt Cov.
No *false* branch from D





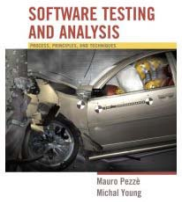
Branch testing

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

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

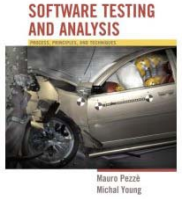
T_3 = {"", "+%0D+%4J"}
100% Stmt Cov. 88% Branch Cov. (7/8 branches)

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)

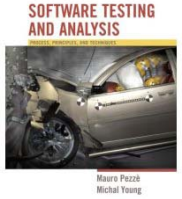


“All branches” can still miss conditions

- Sample fault: missing operator (negation)

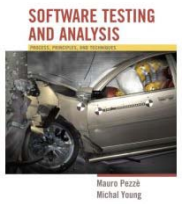
  digit_high == 1 || digit_low == -1

- 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



Condition testing

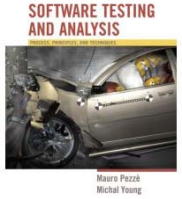
- Branch coverage exposes faults in how a computation has been decomposed into cases
  - intuitively attractive: check the programmer's case analysis
  - but only roughly: groups cases with the same outcome
- Condition coverage considers case analysis in more detail
  - also *individual conditions* in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

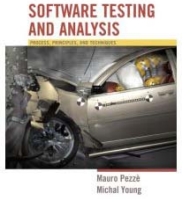
T4 = {“first+test%9Ktest%K9”}

satisfies basic condition adequacy

does not satisfy branch condition adequacy

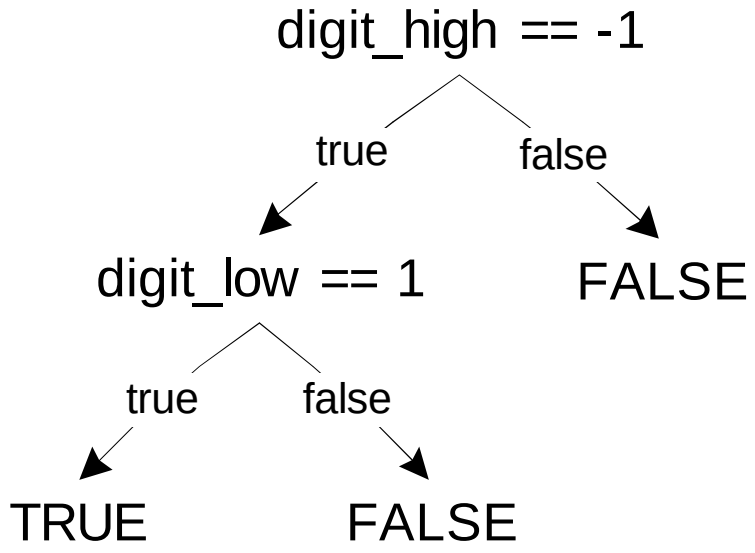
Branch and basic condition are not comparable

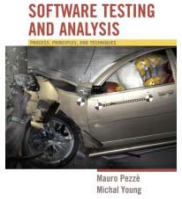
(neither implies the other)



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



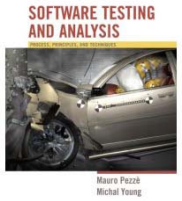


Compound conditions: Exponential complexity

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

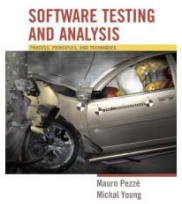
Test Case	a	b	c	d	e
(1)	T	—	T	—	T
(2)	F	T	T	—	T
(3)	T	—	F	T	T
(4)	F	T	F	T	T
(5)	F	F	—	T	T
(6)	T	—	T	—	F
(7)	F	T	T	—	F
(8)	T	—	F	T	F
(9)	F	T	F	T	F
(10)	F	F	—	T	F
(11)	T	—	F	F	—
(12)	F	T	F	F	—
(13)	F	F	—	F	—

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



Modified condition/decision (MC/DC)

- Motivation: Effectively test important combinations of conditions, without exponential blowup in test suite size
  - "Important" combinations means: Each basic condition shown to independently affect the outcome of each decision
- Requires:
  - For each basic condition C, two test cases,
  - values of all *evaluated* conditions except C are the same
  - compound condition as a whole evaluates to *true* for one and *false* for the other



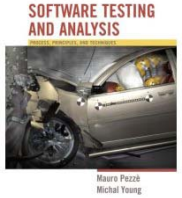
MC/DC: linear complexity

- N+1 test cases for N basic conditions

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

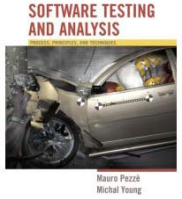
Test Case	a	b	c	d	e	outcome
(1)	true	--	true	--	true	true
(2)	false	true	true	--	true	true
(3)	true	--	false	true	true	true
(6)	true	--	true	--	false	false
(11)	true	--	false	false	--	false
(13)	false	false	--	false	--	false

- Underlined values independently affect the output of the decision
- Required by the RTCA/DO-178B standard



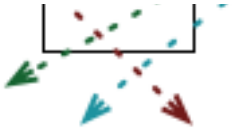
Comments on MC/DC

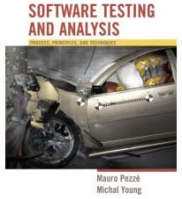
- MC/DC is
  - basic condition coverage (C)
  - branch coverage (DC)
  - plus one additional condition (M): every condition must *independently affect* the decision's output
- It is subsumed by compound conditions and subsumes all other criteria discussed so far
  - stronger than statement and branch coverage
- A good balance of thoroughness and test size (and therefore widely used)



Paths? (Beyond individual branches)

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

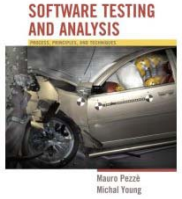




Path adequacy

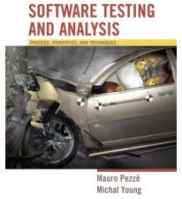
- Decision and condition adequacy criteria consider individual program decisions
- Path testing focuses consider combinations of decisions along paths
- Adequacy criterion: each path must be executed at least once
- Coverage:

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



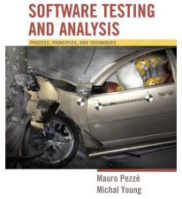
Practical path coverage criteria

- 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

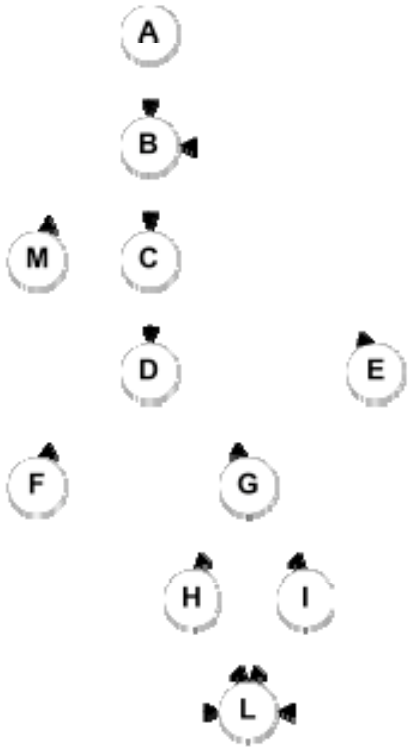


Boundary interior path testing

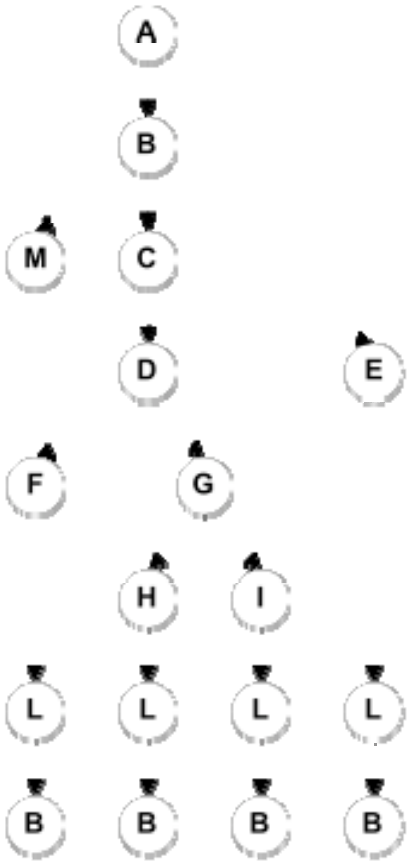
- Group together paths that differ only in the subpath they follow when repeating the body of a loop
  - Follow each path in the control flow graph up to the first repeated node
  - The set of paths from the root of the tree to each leaf is the required set of subpaths for boundary/interior coverage



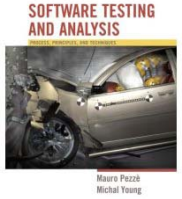
Boundary interior adequacy for cgi-decode



(I)



(■)

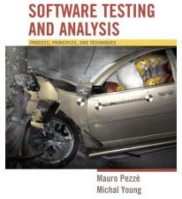


Limitations of boundary interior adequacy

- The number of paths can still grow exponentially

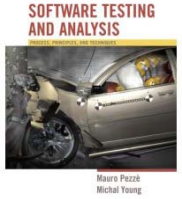
```
if (a) {
   S1;
}
if (b) {
   S2;
}
if (c) {
   S3;
}
...
if (x) {
   Sn;
}
```

- 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



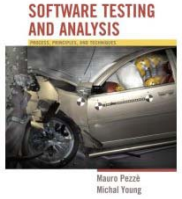
Loop boundary adequacy

- 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



LCSAJ adequacy

- Linear Code Sequence And Jumps: sequential subpath in the CFG starting and ending in a branch
  - TER_1 = statement coverage
  - TER_2 = branch coverage
  - TER_{n+2} = coverage of n consecutive LCSAJs

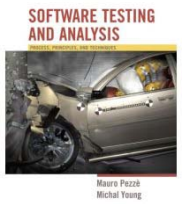


Cyclomatic adequacy

- Cyclomatic number:
  number of independent paths in the CFG
  - A path is representable as a bit vector, where each component of the vector represents an edge
  - "Dependence" is ordinary linear dependence between (bit) vectors
- If e = #edges, n = #nodes, c = #connected components of a graph, it is:
  - e - n + c for an arbitrary graph
  - e - n + 2 for a CFG
- Cyclomatic coverage counts the number of independent paths that have been exercised, relative to cyclomatic complexity

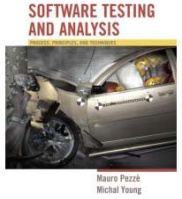
Towards procedure call testing

- The criteria considered to this point measure coverage of control flow within individual procedures.
  - not well suited to integration or system testing
- Choose a coverage granularity commensurate with the granularity of testing
  - if unit testing has been effective, then faults that remain to be found in integration testing will be primarily interface faults, and testing effort should focus on interfaces between units rather than their internal details

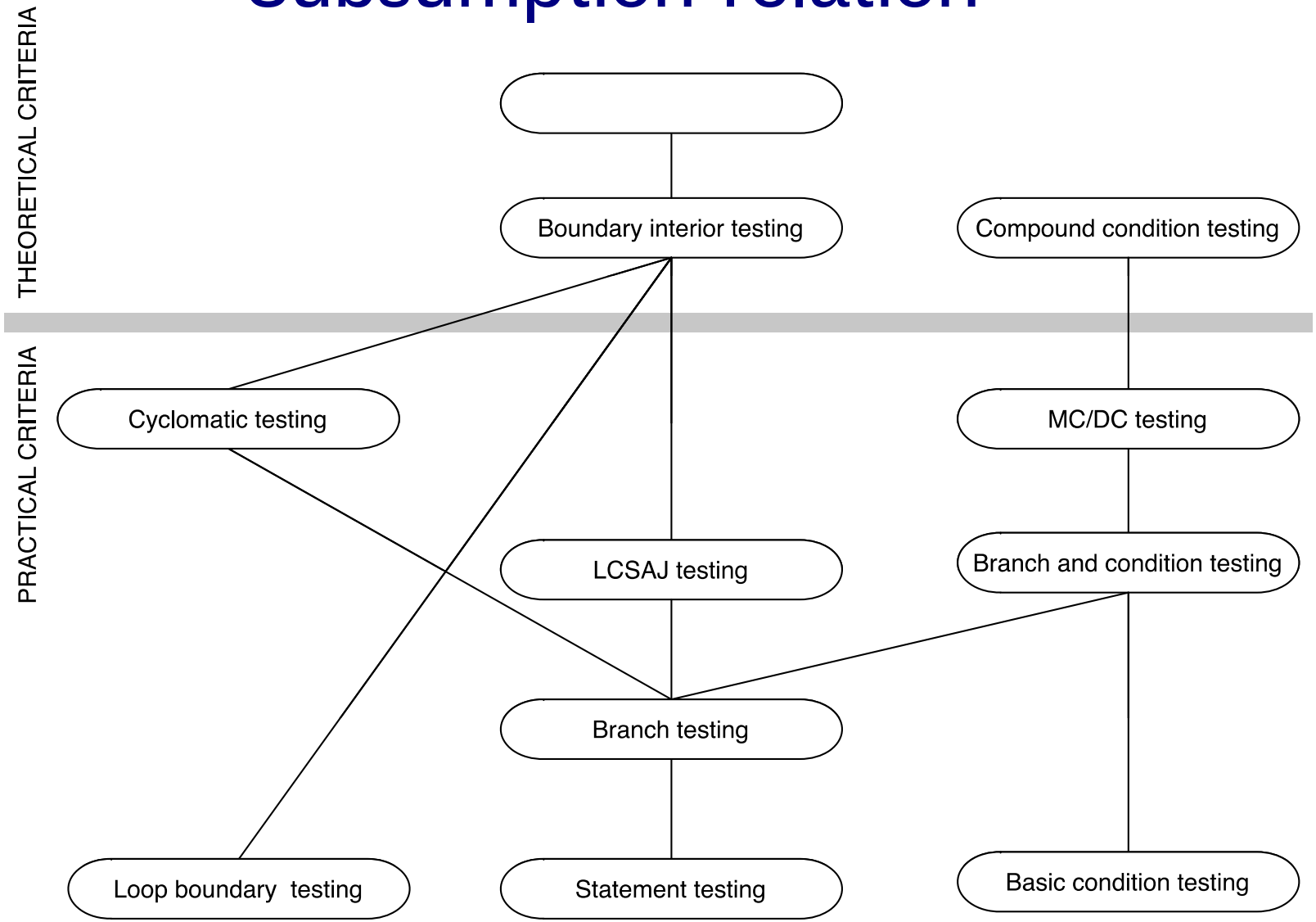


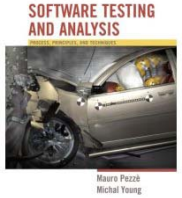
Procedure call testing

- Procedure entry and exit testing
  - procedure may have multiple entry points (e.g., Fortran) and multiple exit points
- Call coverage
  - The same entry point may be called from many points



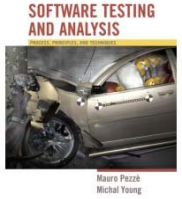
Subsumption relation





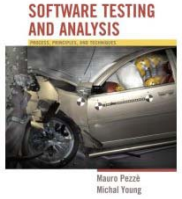
Satisfying structural criteria

- 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



Satisfying structural criteria

- Large amounts of *fossil* 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
    - RTCA-DO-178B and EUROCAE ED-12B for modified MC/DC



Summary

- We defined a number of adequacy criteria
  - NOT test design techniques!
- Different criteria address different classes of errors
- Full coverage is usually unattainable
  - Remember that attainability is an undecidable problem!
- …and when attainable, "inversion" is usually hard
  - How do I find program inputs allowing to cover something buried deeply in the CFG?
  - Automated support (e.g., symbolic execution) may be necessary
- Therefore, rather than requiring full adequacy, the "degree of adequacy" of a test suite is estimated by coverage measures
  - May drive test improvement