WHITE BOX TESTING

Week 10

White Box Testing

sum = 0 + 1 + 2 + … + *i*; *i* ∈ [0, 100]

- read(i);
- **if** ((i < 0) || (i > 100))
  - error();
- **else**
  - { sum=0; x=0;
  - **while** (x < i)
    - { x=x+1;
    - **if** (i==10) sum=1; **else** sum=sum+x; }
  - print(sum); }

Black box *test cases*

- $i = -1$ OK
- $i = 0$ OK
- $i = 1$ OK
- $i = 50$ OK
- $i = 99$ OK
- $i = 100$ OK
- $i = 101$ OK
- *However*:
- $i = 10 \Rightarrow$ ***FAILURE!*** **(sum=1)**

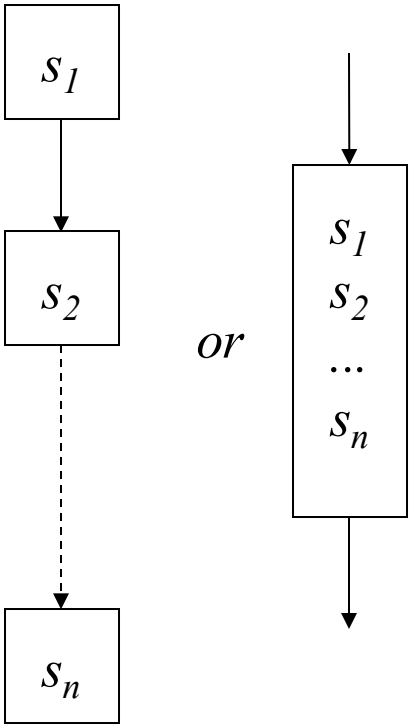
White-box testing principles

- details of source code analyzed
- design of test cases on the basis of code structure
- ***execution path*:** a certain sequence of program statements, executed when starting the program with a certain input (test case)
- different test cases => different execution paths
- ***control-flow testing*:** based on the execution order of the statements
- ***data-flow testing*:** based on the processing of the data during execution

White-box testing principles

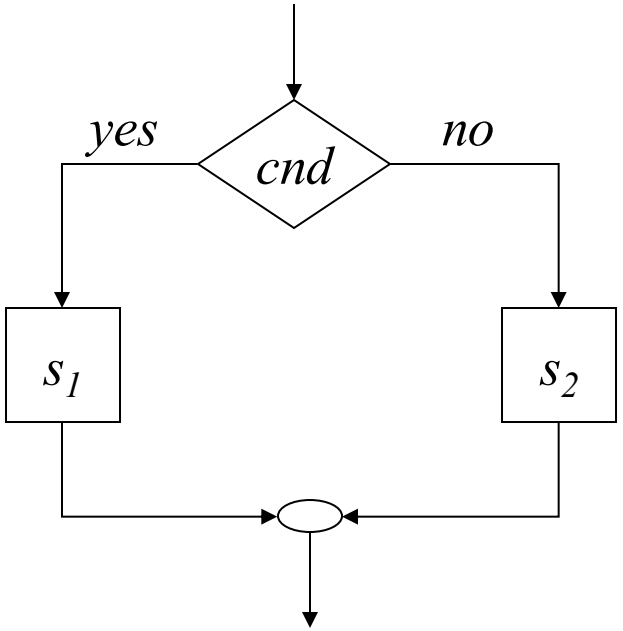
- ***(control) flow graph*:** abstraction of the program's control flow, in graphical form
- ***data-flow graph*:** abstraction of the program's data flow (for a certain input variable), in graphical form; usually extension of control-flow graph
- control-flow graph, data-flow graph automatically produced
- test cases designed from the graphs
- ***coverage*:** the relative amount of statements (and others) executed during testing, computed from control-flow/data-flow graph

Flow graph structures



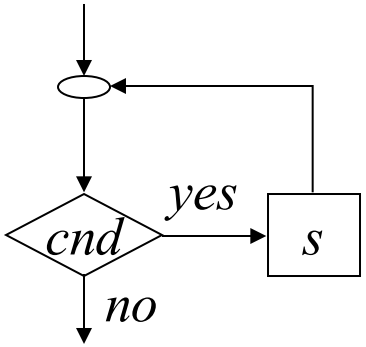
Statements sequence

s_1; s_2; ...; s_n ;

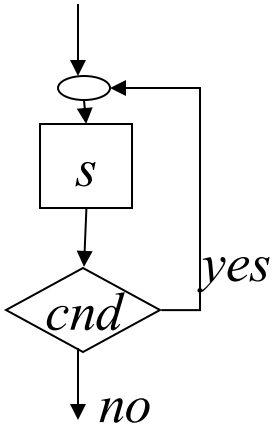


Conditional (if) statement:
if cnd s_1 *else* s_2.

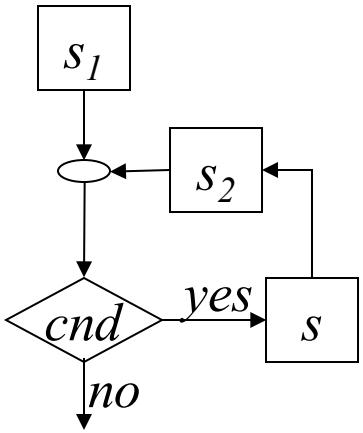
Flow graph structures



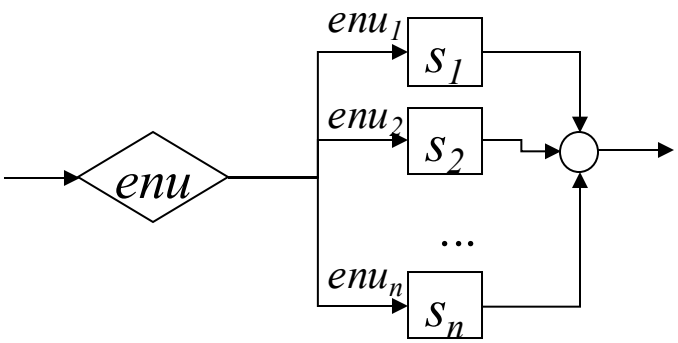
While-do Loop Statement
***While** cnd **do** s;*



Do-while Loop Statement
***Do** s **while** cnd;*



For Loop statement: (Iteration)
***For** (s1; cnd;s2) **do** s;*



Switch-case statement: (multiple decision)
***Switch** (enu) { **case** enu1: s1; **case** enu2: s2; ...;}*

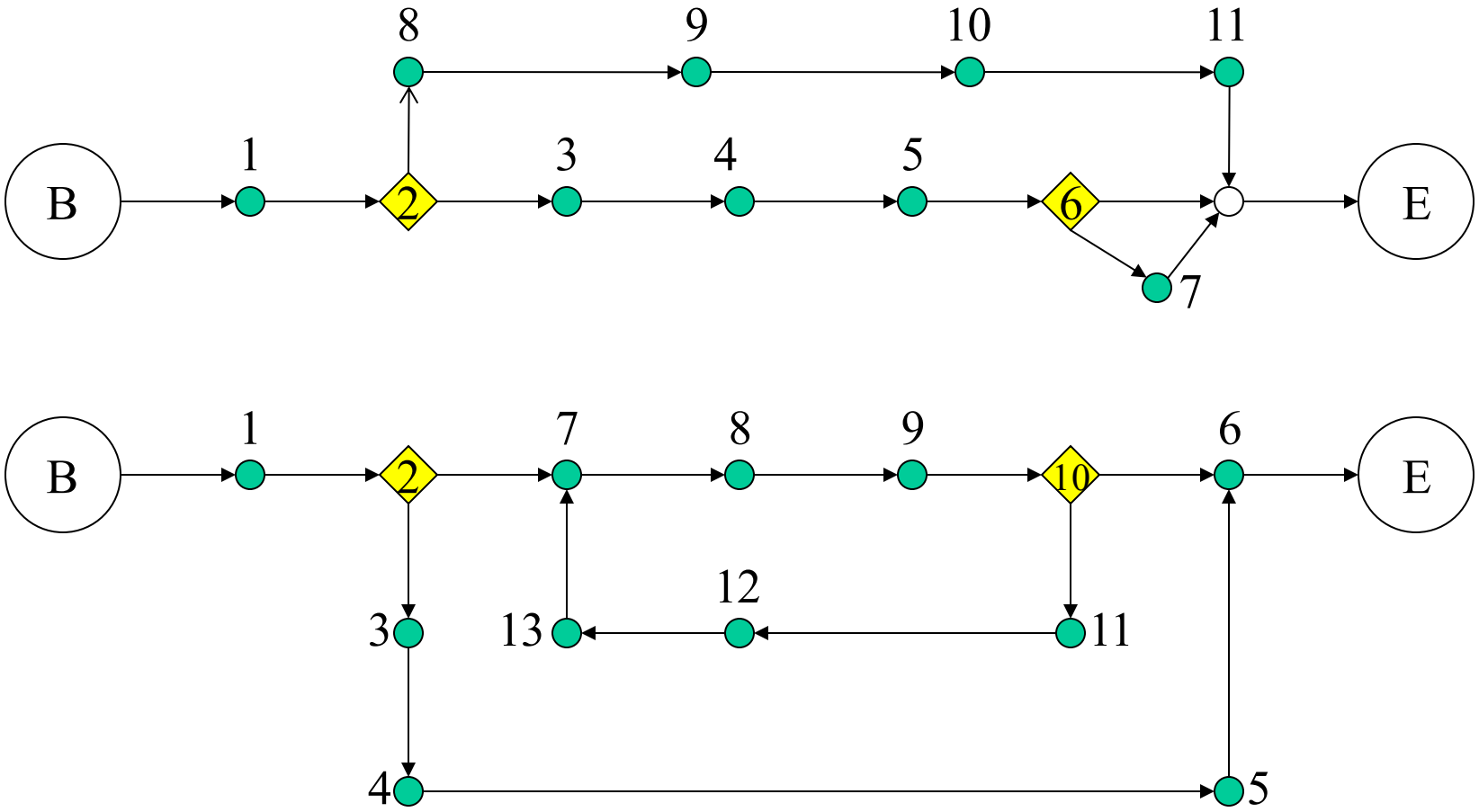
Control-flow testing

- ***Coverage***: how extensively the program has been (or will be) tested with a given set of test cases
- the (relative) number of *nodes* (statements) in the flow graph executed during testing
- the (relative) number of *edges* (control transitions) in the flow graph traversed during testing
- ***Statement coverage***: each node (statement) has to be executed at least once
- ***Branch coverage***: each edge (transition) has to be traversed at least once
- a large number of variations of different coverage power
- special target: ***loop testing***

Execution Path

- a *sequence of nodes and connecting edges* from the unique begin-node of the flow graph to the unique end-node of the graph.
- A ***certain instance of the relevant program execution***
- May contain the same node several times: *loops*.

White-box (structural) testing

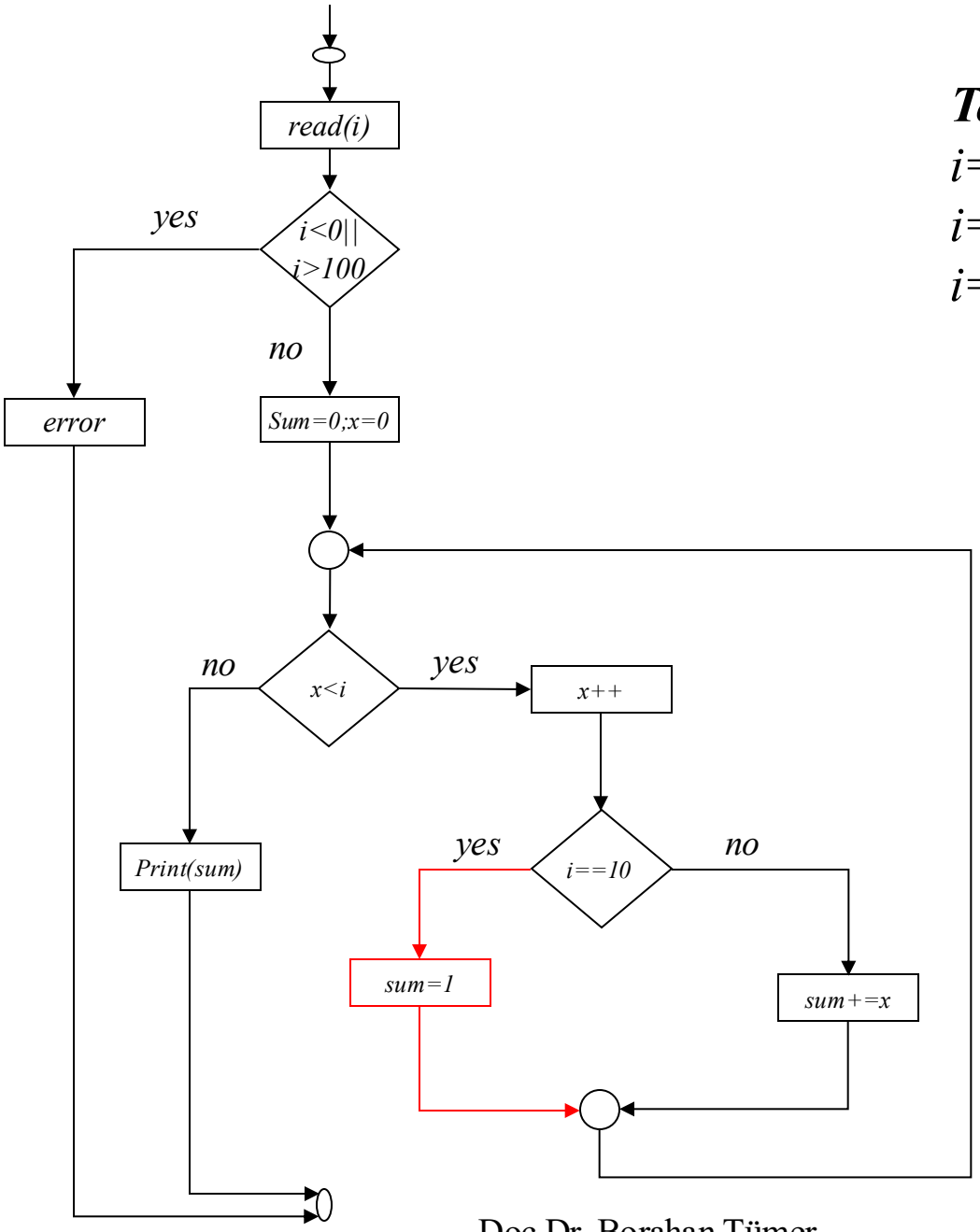


Statement coverage criterion

- A set *P* of execution paths satisfies the ***statement coverage criterion*** if and only if for all *nodes n* in the flow graph, there is at least one path *p* in *P* such that *p* contains the node *n* ≡ Each statement of the program is executed at least once during testing, by some test case.
  - criterion met => *complete* (100%) statement coverage
  - criterion not met => partial statement coverage (< 100%)
  - **begin-node, end-node** and **junctions excluded**
  - complete coverage surprisingly hard to achieve in practice
  - "dead code" / conditional compilation

Branch coverage criterion

- A set *P* of execution paths satisfies the ***branch coverage criterion*** if and only if for all *edges e* in the flow graph, there is at least one path *p* in *P* such that *p* contains the edge *e* ≡ Each control-flow branch / decision (*true / yes*, *false / no*) is taken at least once during testing, by some test case.
  - criterion met => *complete* (100%) branch coverage
  - complete branch coverage => complete statement coverage (branch coverage *subsumes* statement coverage)
  - usually more tests are needed for complete branch coverage than needed for complete statement coverage
  - branch coverage is more extensive: the criterion is stronger than the statement coverage criterion
  - criterion not met => partial branch coverage (< 100%)

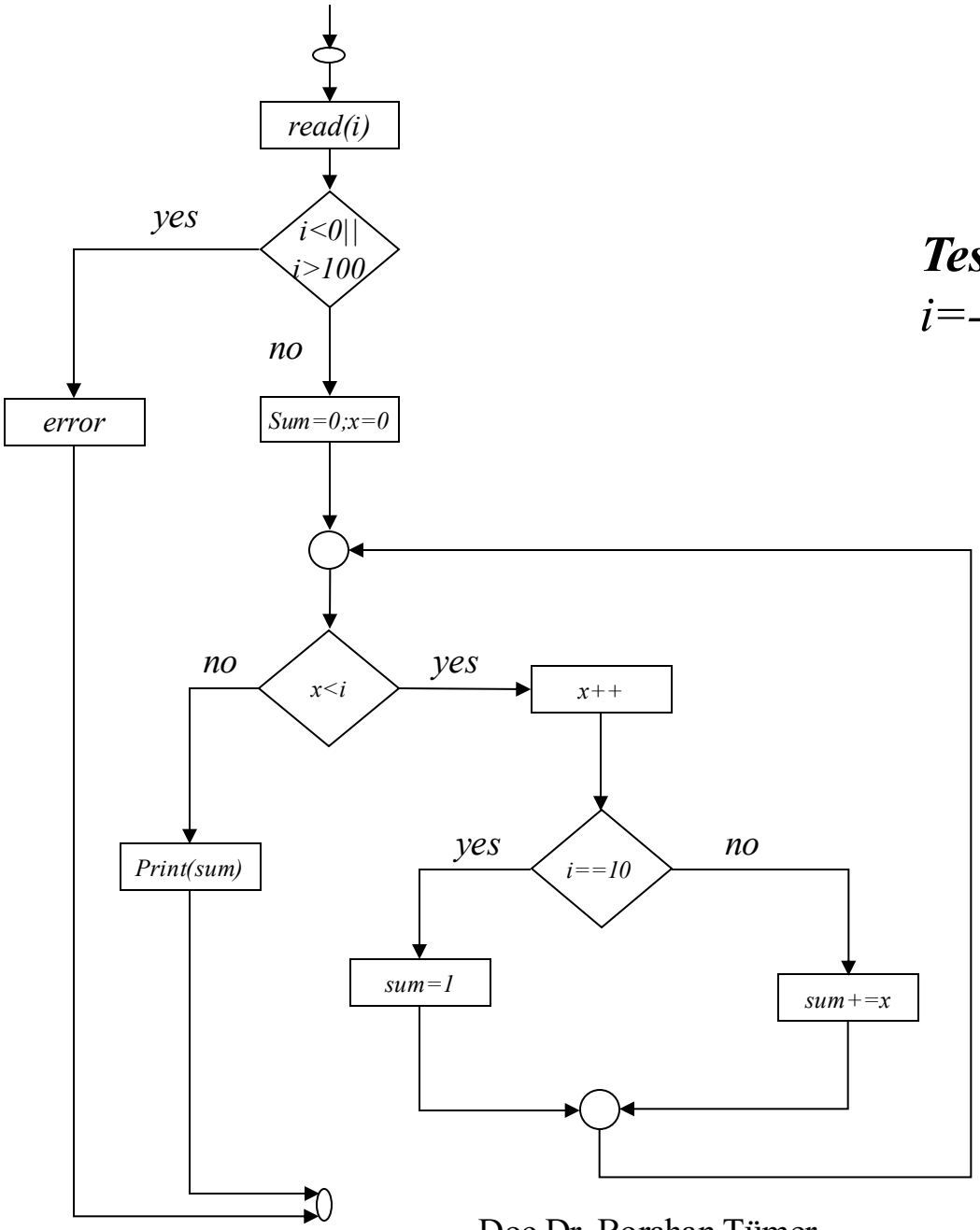


Test cases:
i=-1; i=0; i=1;
i=50; i=99;
i=100; i=101

Statement Coverage:
9/10=90%

Branch coverage:
11/13=84.6%

***red-lined statement is not executed!**



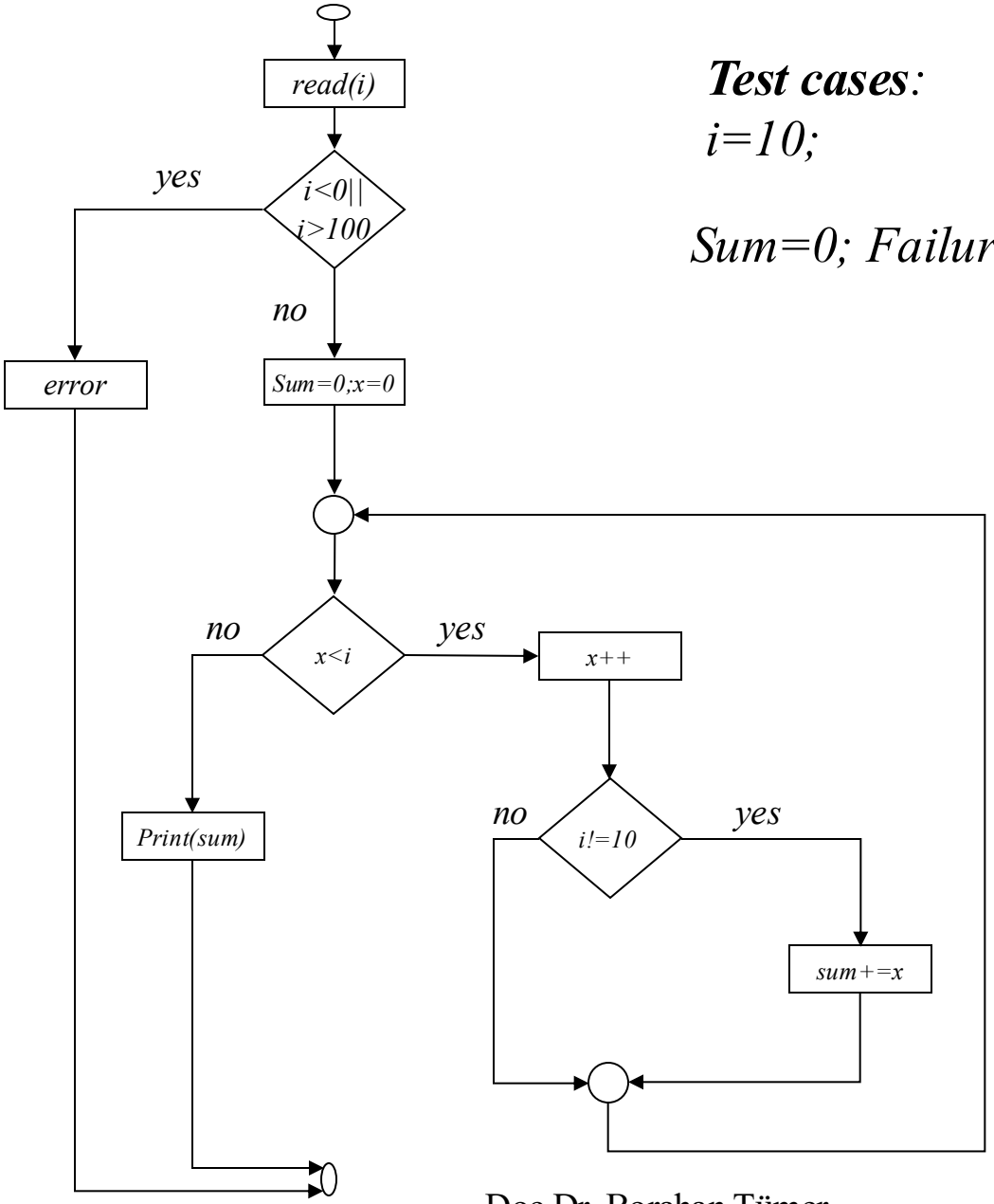
***Test cases**:*
i=-1; i=1; i=10;

Statement Coverage:
10/10=100%

Branch coverage:
15/15=100%

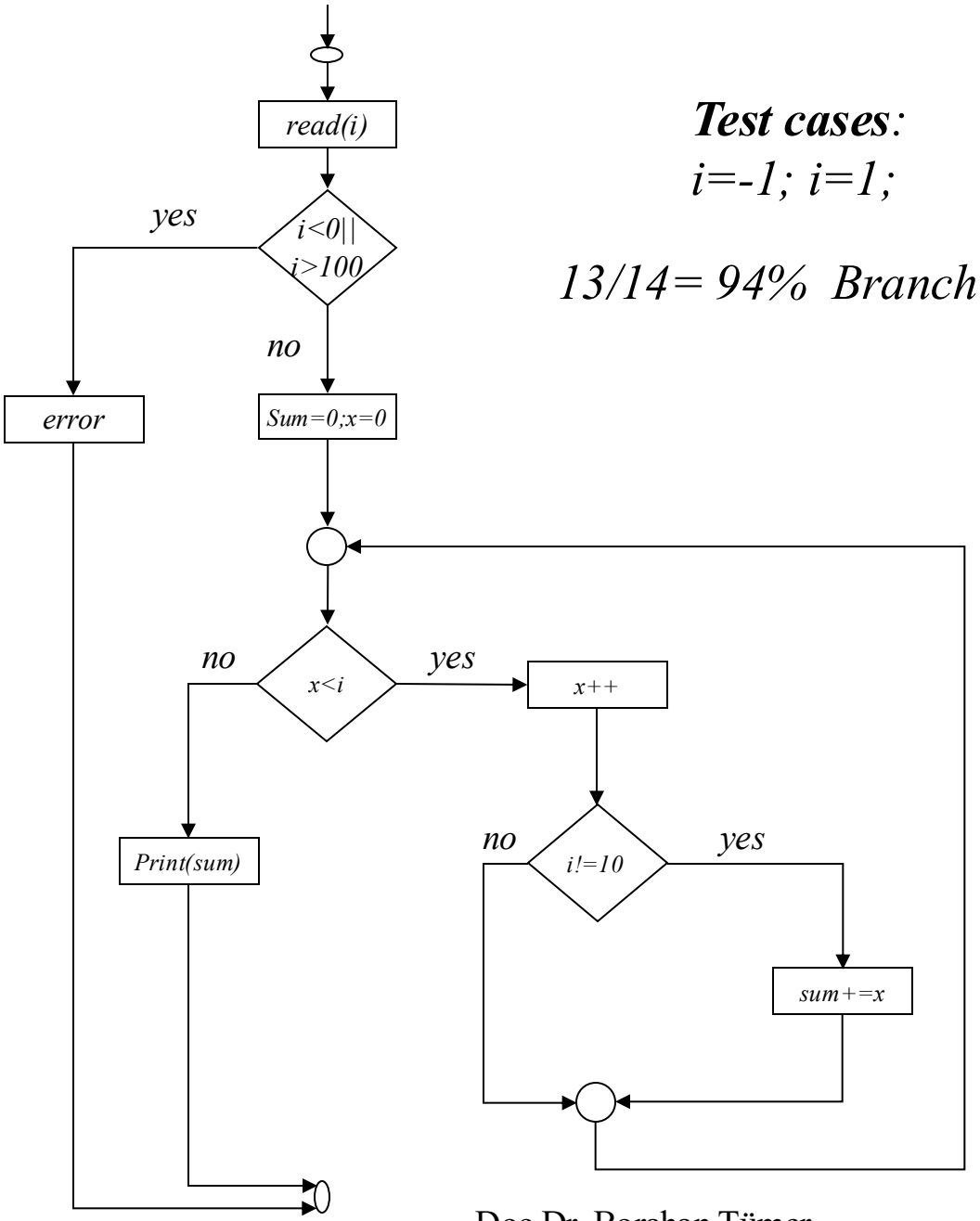
Statement coverage ≠ Branch coverage

- read(i);
- **if** ((i < 0) || (i > 100)) error() **else**
- { sum=0; x=0;
  - **while** (x < i)
  - { x=x+1; **if** (i <> 10) sum=sum+x; }
  - print(sum);
- }



***Test cases**:*
i=10;

Sum=0; Failure!!!



***Test cases**:*

i=-1; i=1;

13/14= 94% Branch coverage

Condition coverage criterion

- A set *P* of execution paths satisfies the ***condition coverage criterion*** if and only if for every control node in the flow graph consisting of atomic predicates (c_1, c_2, …, c_n), c_i yields *true* (*yes*) when evaluated within a path p_1 in *P* and c_i yields *false* (*no*) when evaluated within a path p_2 in *P*, $i = 1, 2, \ldots, n$.
  - internal structure of composite control predicates taken into account: (i < 0) || (i > 100)
  - each predicate ("i<0," "i>100") tested separately for both "true" and "false."

Multicondition coverage criterion

- A set *P* of execution paths satisfies the ***multicondition coverage criterion*** if and only if for every control node in the flow graph consisting of atomic predicates (c_1, c_2, …, c_n), each possible *combination* of their truth values (*true/yes*, *false/no*) is evaluated within at least one path *p* in *P*.
  - stronger requirement than for condition coverage all the *combinations*
  - for 2 atomic predicates: (*true*,*true*), (*true*,*false*), (*false*,*true*), (*false*,*false*)
  - for 3 atomic predicates: 8 combinations, etc….

Path coverage criterion

- Path coverage the strongest criterion
  - usually impossible to reach

Independent path coverage criterion.

- Independent path coverage stronger than branch coverage
  - used also in complexity analysis of programs

Cyclomatic Complexity

- ***Cyclomatic complexity (CC)*** of a piece of code is the *number of linearly independent (LI) paths* through the piece of code.

- ***E.g.*** For a piece of code *with no decision structures*, ***CC=1***, meaning there is a *single* LI path.

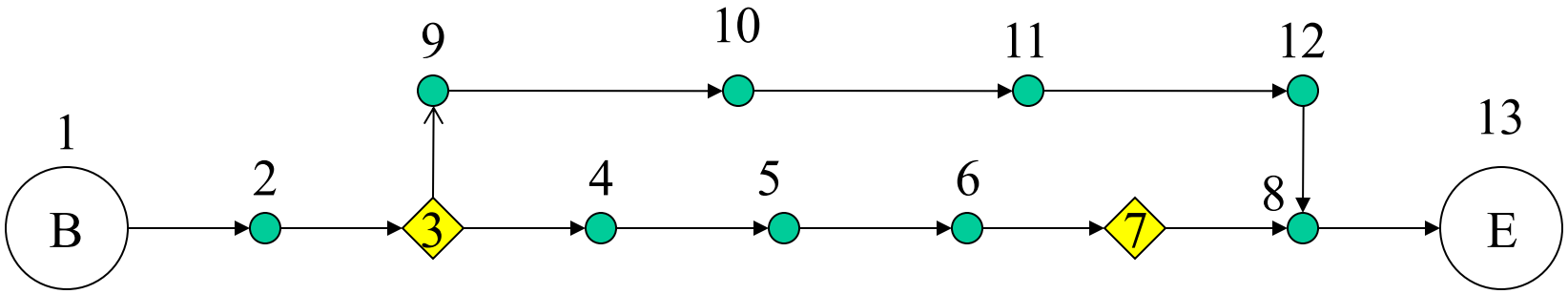
Cyclomatic Complexity

- Given that the source code is represented as a directed graph (i.e., a control-flow graph), CC can be formulated as

$$CC = E - N + 2$$

  - where
    - E is the number of edges and
    - N is the number of nodes
  - of the graph.

Cyclomatic Complexity



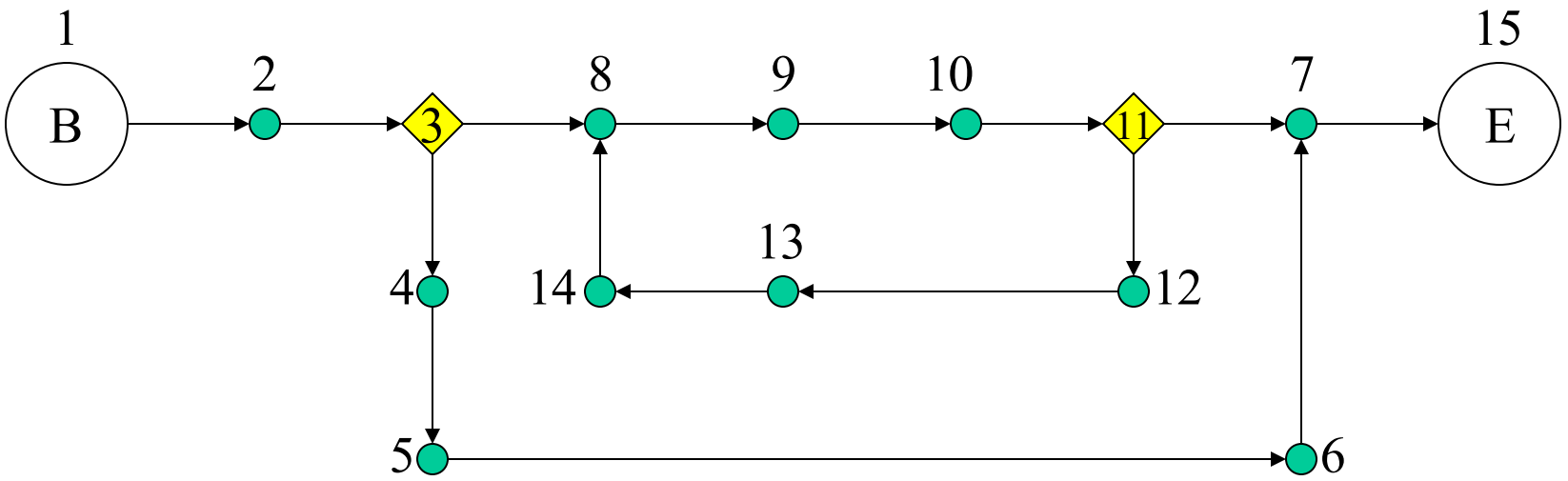
E=13;

N=13;

CC=E-N+2=2

CC=2

Cyclomatic Complexity



E=16;

N=15;

CC=E-N+2=3

CC=3

Independent paths

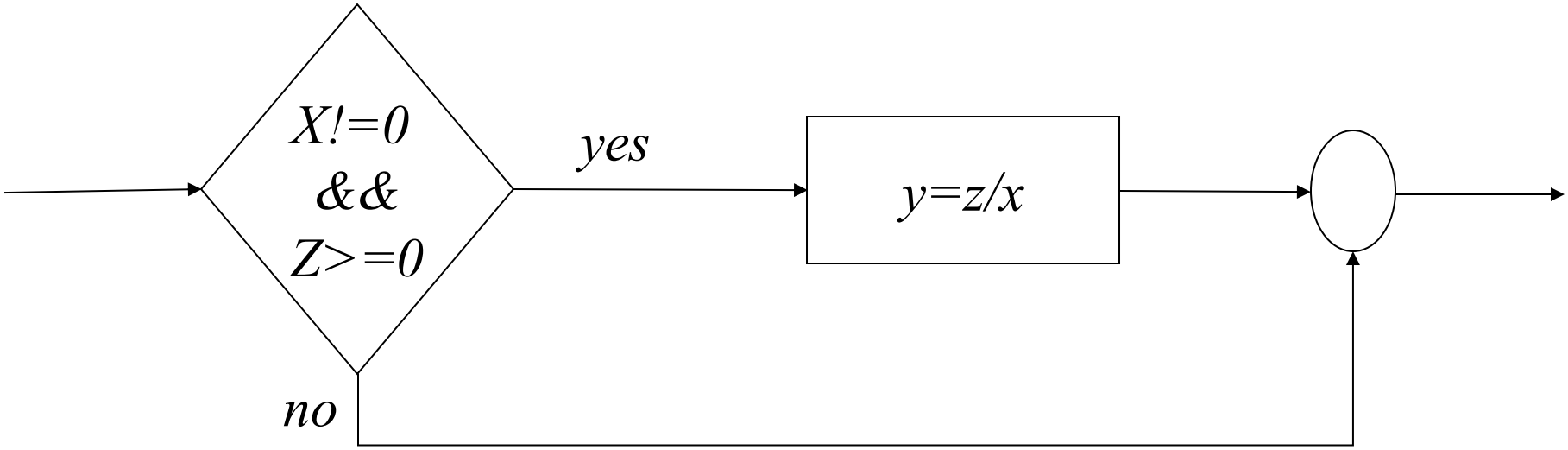
1 2 3 8 9 10 11 7 15

1 2 3 4 5 6 7 15

1 2 3 8 9 10 11 12 13 14 8 9 10 11 7 15

An example for coverage comparison

if ((x != 0) && (z >= 0)) y=z/x ;



Comparison of coverages

- Complete statement coverage: ($x = 1, z = 0$) **[1 test]**
- Complete branch coverage: ($x = 1, z = 0$) for the *yes* branch, ($x = 1, z = -1$) for the *no*-branch **[2 tests]**
- Complete condition coverage: ($x = 0, z = 0$) for combination (*false*, *true*), ($x = 1, z = -1$) for combination (*true*, *false*) (*yes*-branch unexplored !) **[2 tests]**
- Complete multicondition coverage: ($x = 1, z = 0$) for combination (*true*, *true*), ($x = 1, z = -1$) for combination (*true*, *false*), ($x = 0, z = 0$) for the combination (*false*, *true*), ($x = 0, z = -1$) for the combination (*false*, *false*) **[4 tests]**

Selection of test cases

- A certain code coverage criterion and percentage (e.g. 100% branch coverage) has been chosen ⇒ one has to design test cases to reach the criterion.

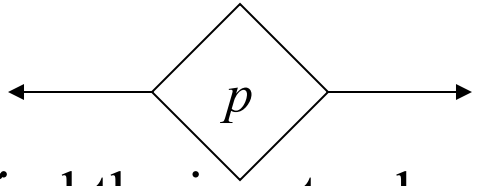
1) Construct a flow graph for the program (with a white-box tool).
2) Choose the execution paths that satisfy the criterion.
3) For each execution path, design a test case (program input-output) that activates a traversal of the path.
4) Execute the tests (with the white-box tool that calculates the coverage).
5) If the required coverage has not been reached, return to step 3 to design additional test cases for those execution paths that have not been traversed yet during testing.

Path sensitization

- The process of designing a test case for a particular execution path
  - In general, *path sensitization* is undecidable: there is no algorithm that can find a suitable test case for each possible path.
  - Symbolic execution and equation solving tools succeed in some cases.
  - A *heuristic*: Begin with the control conditions of a branch at the *end* of the path. Select such variable values that will satisfy these conditions. Repeat this analysis for each prior branch in the path until you reach the entry node of the flow graph. Use the selected values of the input variables as the test case for the path.
  - There may be infeasible paths that cannot be executed with any input, caused by short-circuit evaluation, contradictory or mutually exclusive control conditions, redundant control predicates, or dead code

Path Sensitization.. Cont'd

- the crucial points in a flow-graph are those where the execution diverges, that is, the ***control predicates*** of branches



- one has to find the input values such that when executing the program with the input, control branches into the desired direction and the predicate *p* obtains the corresponding value (*true* / *false*) or value combination
  - *note* 1: *p* may depend on the input just indirectly
  - *note* 2: it may not be possible to obtain all required truth values for *p*: ((x == 1) && (x==2))

Loop Testing

- ***Testing of simple loops***: 0 iterations (no looping), *minimum* number of iterations (possibly 0), *minimum*+1 iterations, *typical* number of iterations, *maximum*-1 iterations, *maximum* number of iterations, *maximum*+1 iterations (should not be feasible)
  - note: loops with fixed iteration control may not be executable (testable) with all suggested iteration patterns

    for (j=0; j < 999; ++j) { … }
- ***Testing of serial loops***:
  - if there is no data-flow relationship between the loops, test them both as simple loops
  - if there is a data-flow relationship between the loops, test them as if the loops were nested
- ***Testing of unstructured ("spaghetti") loops***: test the loop with an equivalent simple / serial / nested loop as model
  - spaghetti code should be rewritten into structured form, for testing as well as for maintenance purposes

Loop Testing ...2

- ***Testing of nested loops***:
  - There would be too many tests when repeating all the inner loop tests every time an outer loop is iterated, so:

  1) The *innermost loop* is tested *first* using the *simple-loop strategy*. The *other loops* are iterated their *minimum number of times*.
  2) Set up the looping conditions of the previously tested loop such that it will be iterated a ***suitable*** number of times (***minimum***, ***typical***, or ***maximum***).
  3) Proceed to testing the outer loop which is nesting the previously tested one, using the simple-loop strategy. (The *outer loops* are iterated their *minimum number of times*, the inner loops are iterated their *suitable number of times*.)
  4) *Repeat the steps 2 and 3, until the outermost loop has been tested*.
  5) Set up a test that will *iterate all loops their maximum number of times*.

Loop Testing ...3

Example:

```
Loop #1
    …
    Loop #2
        …
        Loop #3
            …
            Loop #4
                …
            }
        }
    }
}
```

1. Loop 4: simple loop strategy (Loop 1-3 minimum)
2. Loop 4: suitable + Loop 3 simple (Loop 1, 2 minimum)
3. Loop 4, 3: suitable + Loop 2 simple (Loop 1 minimum)
4. Loop 4-2: suitable + Loop 1 simple
5. Loop 4-1: maximum

References

[1] Myers, *The Art of Software Testing*, 1978

Ex: