SoberIT
Software Business and Engineering Institute

T-76.5613 Software Testing and Quality Assurance

Lecture 3, 18.9.2006

White-Box Testing Techniques

Juha Itkonen
SoberIT

Content

- What are white-box testing techniques
- Control flow testing
  - Statement coverage
  - Branch coverage
  - Condition coverage
  - Multicondition coverage
- How to use white box testing tools
  - Example of one testing tool

White-box Testing (structural testing)

- Testing is based on the knowledge of the inner structure of the software
- Test cases are designed and selected based on code
  - to exercise specific internal code structures
- White-box techniques often focus on satisfying a certain test coverage criteria
  - covering all statements, branches, conditions, ...
- Different test cases => different execution paths

Faults in boolean expressions

```
x && (y || z)
```

- Perhaps wrongly parenthesized
- Perhaps || should have been && or vice versa
- Perhaps **!** was omitted
- Perhaps the expression was made too complicated
- Perhaps there are still checks missing
- Perhaps a wrong variable was used
- Perhaps short-circuiting was not considered

An example

0 + 1 + 2 + … + *i*, *i* ∈ [0, 100]

```
read(i);
if ((i < 0) || (i > 100)) {
    error("Input value out of range");
} else {
    sum=0; x=0;
    while (x < i) {
        x=x+1;
        if (i==10) sum=1; else
        sum=sum+x;
    }
    print(sum);
}
```

Test derived using black-box approach

$i = -1$	OK
$i = 0$	OK
$i = 1$	OK
$i = 50$	OK
$i = 99$	OK
$i = 100$	OK
$i = 101$	OK

- $i = 10$ $\Rightarrow$ ***Failure!*** **(sum = 1)**

White-box test design techniques

- Control-flow based coverage criteria
  - Based on the execution order of code statements
    - Statement coverage
    - Decision coverage
    - Condition coverage
    - Decision/condition (multi-condition) coverage
    - ...
- Data flow based coverage criteria
  - Based on the values of variables and their flow through the program
    - All definitions testing
    - All uses testing
    - ...

Flow graphs

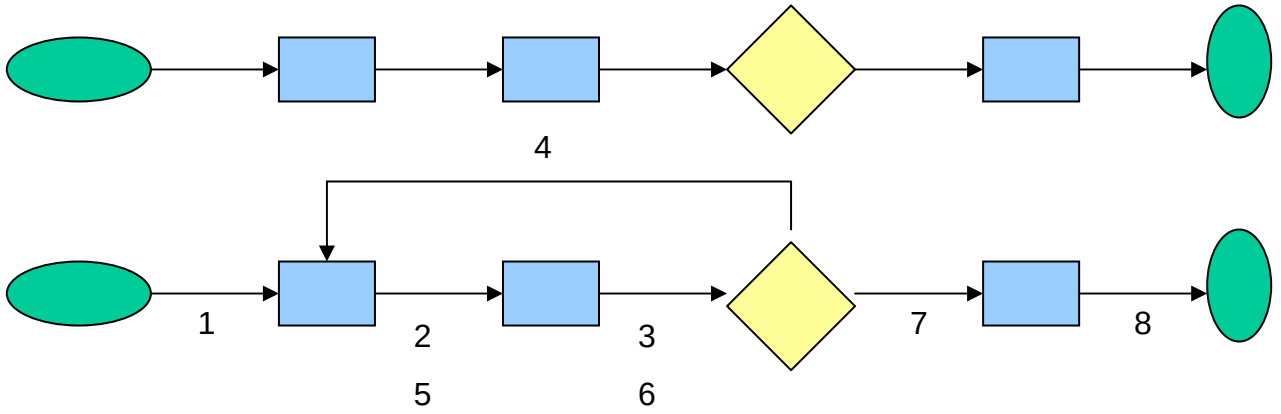
- **Control-flow graph**: a graphical representation of program's internal flow of control
- **Data-flow graph**: a graphical representation of program's internal data flow (with respect to some variable); usually an extension of a control-flow graph

- Flow graphs can be produced automatically
- Test cases are planned based on the flow graphs

- Code coverage
  - The relative amount of executed statements during testing
  - Calculated from control-flow or data-flow graph

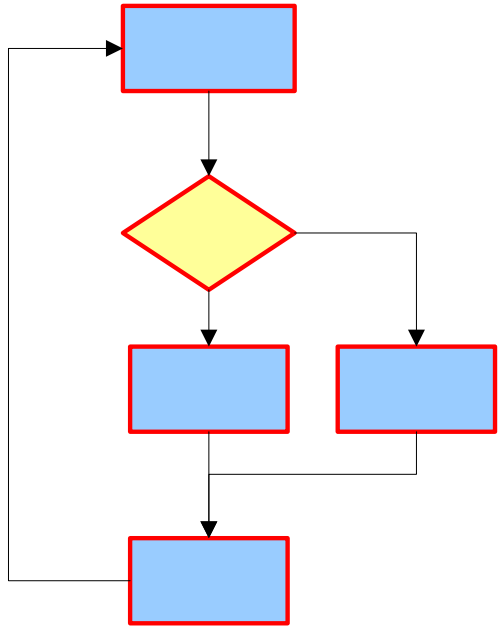
Execution path

- Execution path is a sequence of nodes (and connecting edges) from the unique begin-node of the flow graph to the unique end-node of the graph.
- May include same edges and nodes several times (loops)



Statement coverage

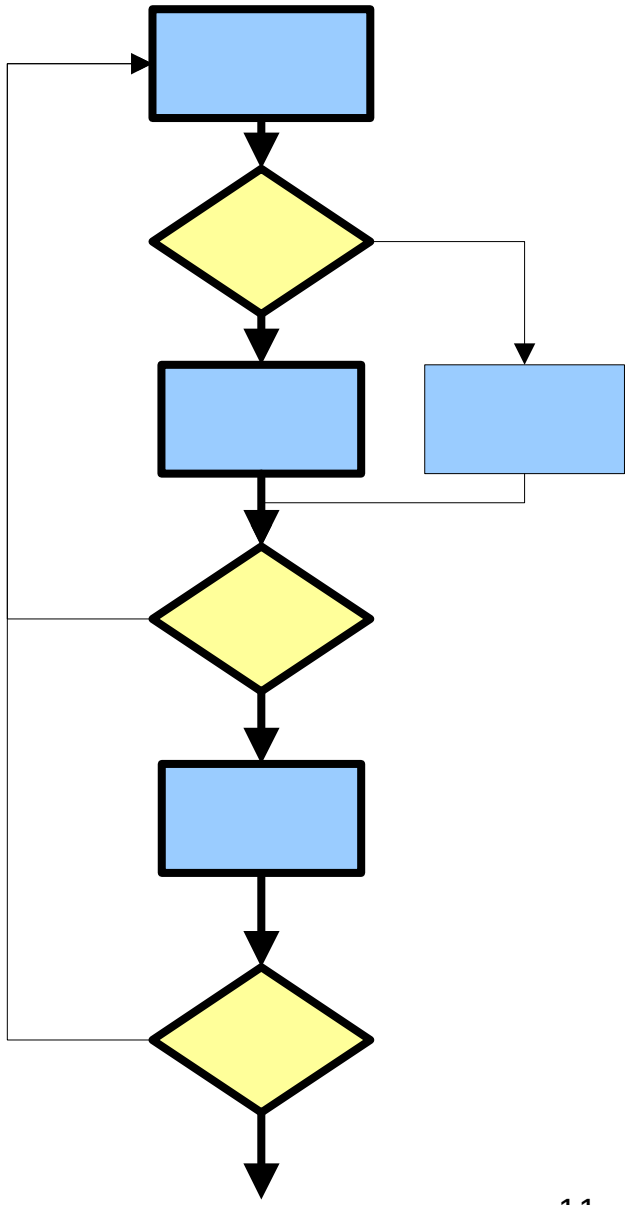
- *A set P of execution paths satisfies the statement coverage criterion if and only if for all nodes n in the flow graph, there is at least one path p in P such that p contains the node n.*
- Each program sentence is executed at least once by some test case
  - Satisfies the criterion → complete (100%) statement coverage
  - Does not satisfy the criterion → partial (<100 %) statement coverage
  - Surprisingly difficult to achieve complete coverage in real life
    - Dead code
    - Error handling code and other rare conditions
    - Inadequate test cases
    - Conditional compiling



Statement coverage

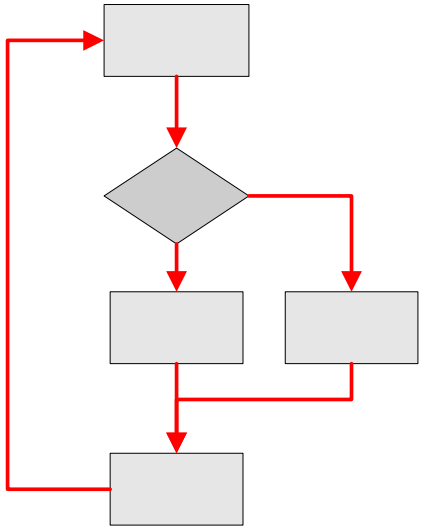
- Percentage of executable statements exercised by a test suite
  - Number of statements exercised/total number of statements
- Example:
  - Program has 7 statements
  - Tests exercise 6 statements
  - Statement coverage = 85,7%

Typical ad hoc testing achieves 60 - 75%



Decision (branch) coverage

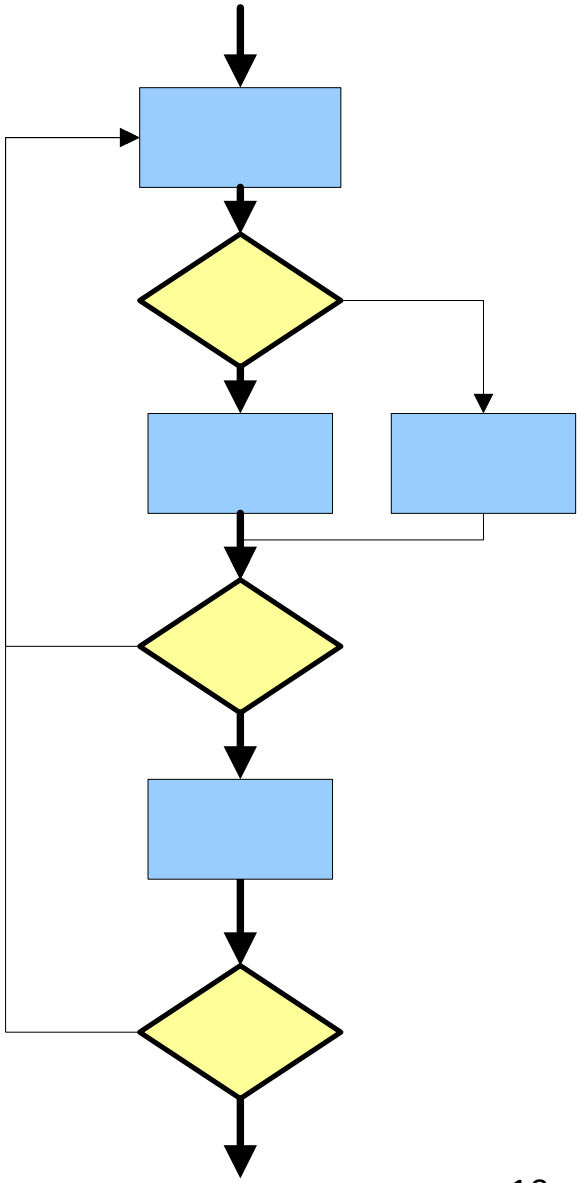
- *A set P of execution paths satisfies the branch coverage criterion if and only if for all edges e in the flow graph, there is at least one path p in P such that p contains the edge e.*
- Each control flow branch is executed at least once by some test case
  - Satisfies the criterion → complete (100 %) branch coverage
  - Perfect branch coverage → perfect statement coverage
  - Usually achieving branch coverage takes more test cases than achieving statement coverage
  - Branch coverage is ”stronger” than statement coverage



Decision (branch) coverage

- Percentage of decision outcomes exercised by a test suite
  - Number of decisions outcomes exercised/total number of decision outcomes
- Example:
  - Program has 6 branches
  - Tests exercises 3 branches
  - Branch coverage = 50%
    - Statement coverage = 85,7%

Typical ad hoc testing achieves 40 – 60%



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);
}
```

Example

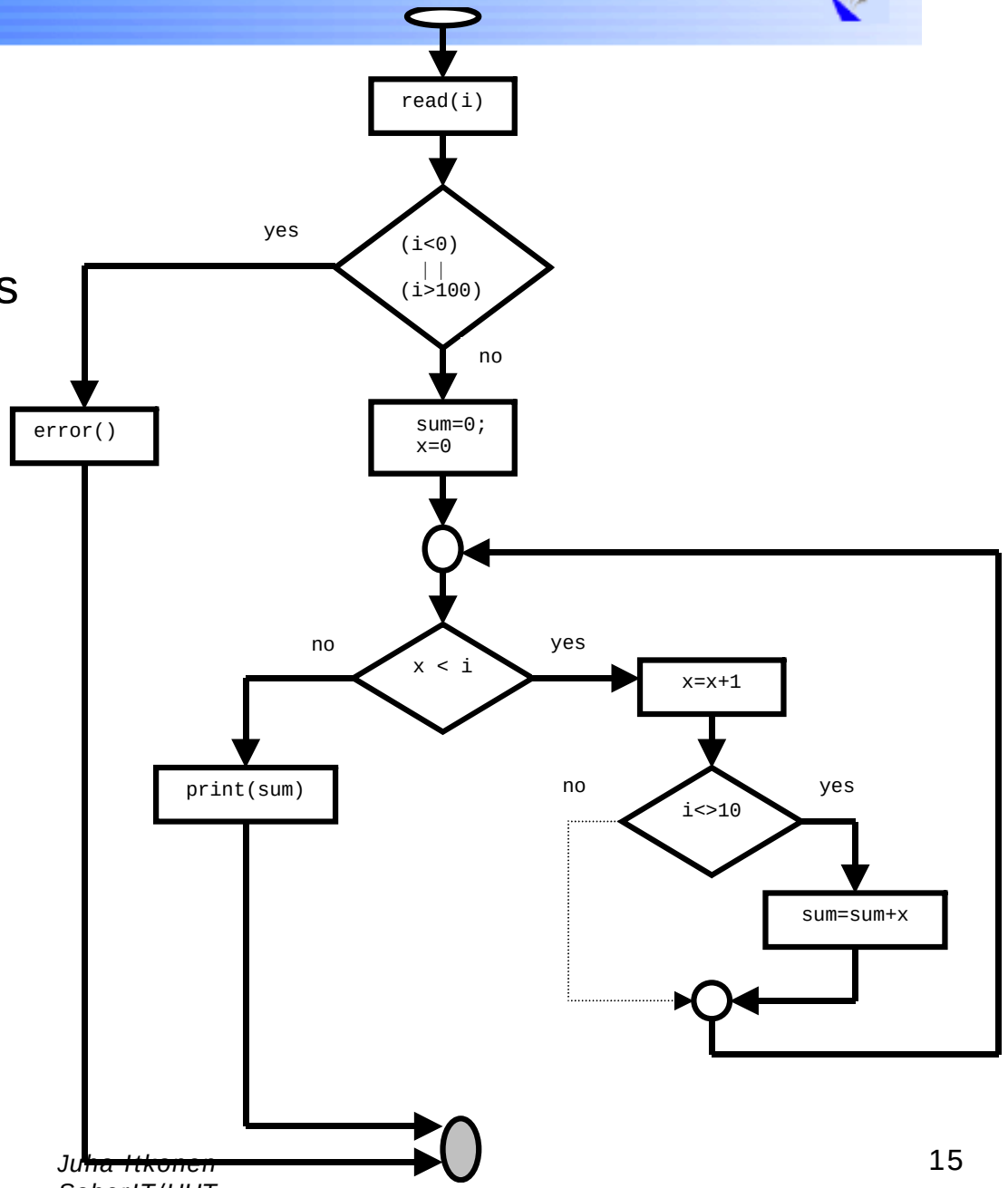
- Perfect statement coverage is not always perfect branch coverage
- Test cases:
  - i=-1
  - i=1
- 100% statement coverage

Branch Coverage: 5 / 6 = 83,3 %

Test case i=10:

-> sum=0

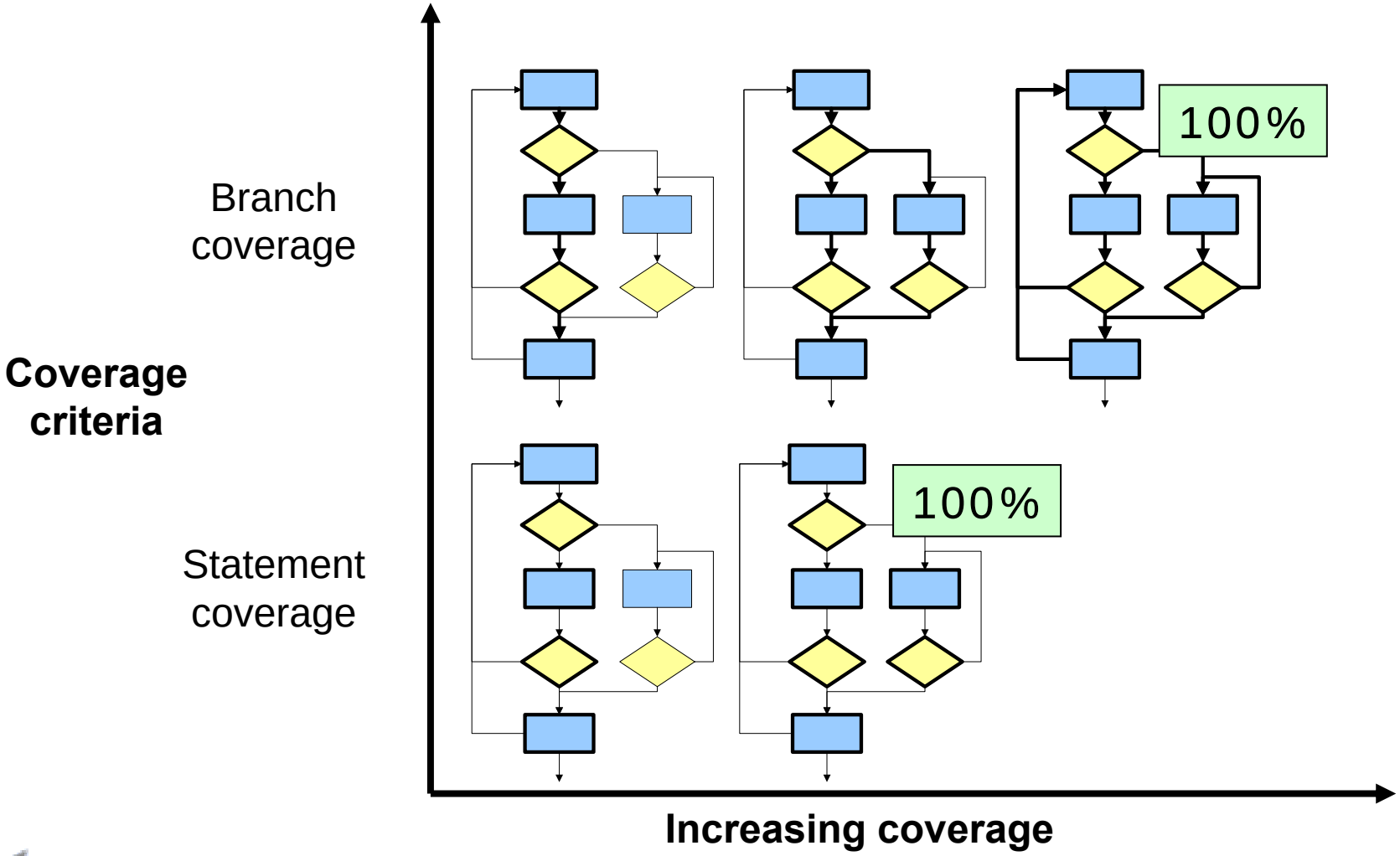
-> failure



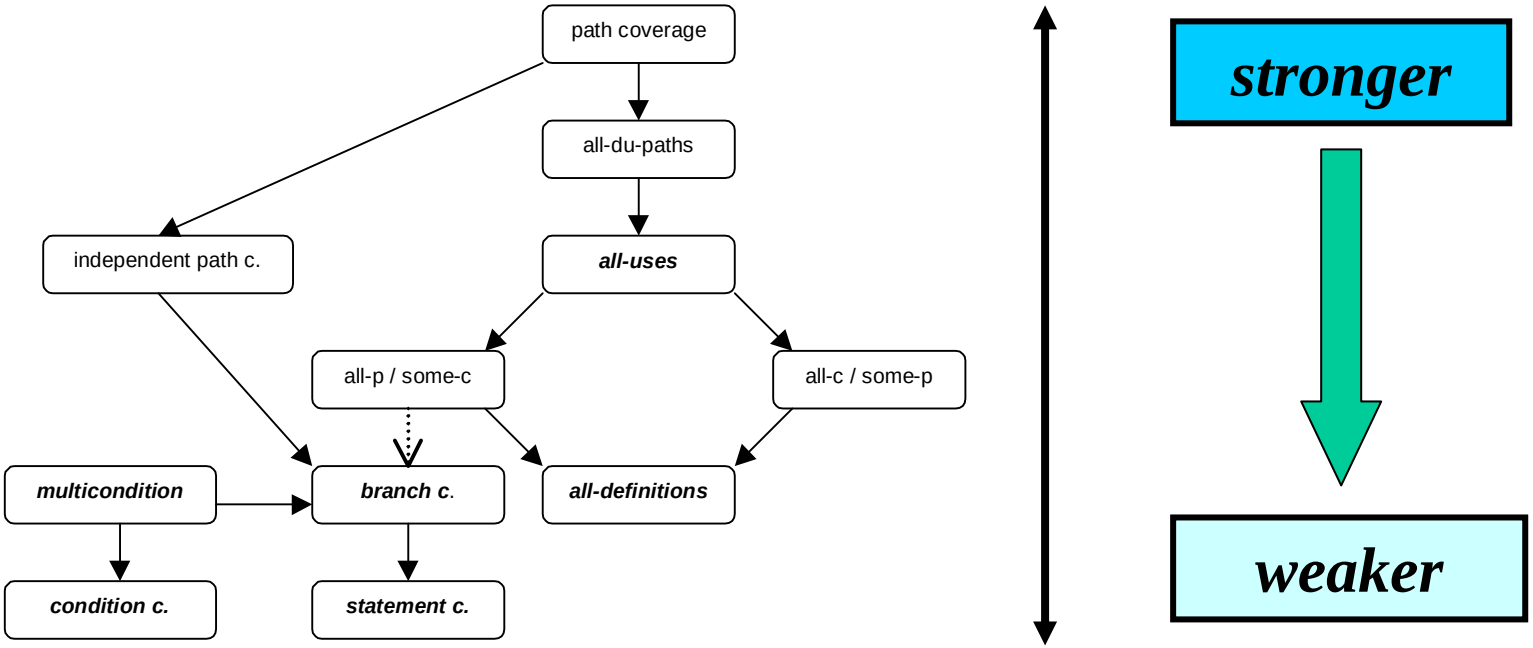
Condition Coverage

- Condition Coverage
  - The internal structure of a structured control predicates is taken into account
  - Both true and false values of each predicate are exercised
  - E.g. (a<0 || b>0) -> (true, false) and (false,true)
- Decision/condition (multi-condition) Coverage
  - The most advanced control-flow testing strategy of practical relevance
  - Execute all sub-predicate boolean value combinations
    - E.g. (a<0 || b>0)
    - (true, true), (true, false), (false, true), (false, false)
  - Test each sub-predicate combination separately

Strength of structural coverage



Comparison of white-box methods



Note!: Hierarchy is just directing, not absolute

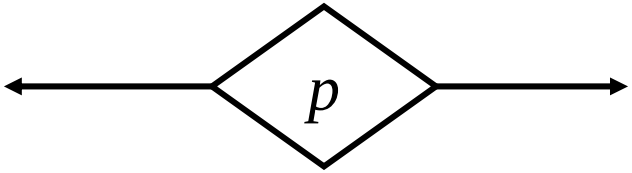
- o problem: relationship between data- and control flow methods
  (*all-p / some-c coverage* → *branch coverage*)
- o What about a program with no variables?

Choosing the test cases

- Having chosen a coverage criteria => plan test cases to fulfil the criteria
- Construct the (control) flow graph for the program
- Choose the minimum number of execution paths that satisfy the criterion
- For each selected path, prepare a test case that activates the execution of the path
- Execute tests
- Unless coverage achieved, plan more test to add up to the coverage

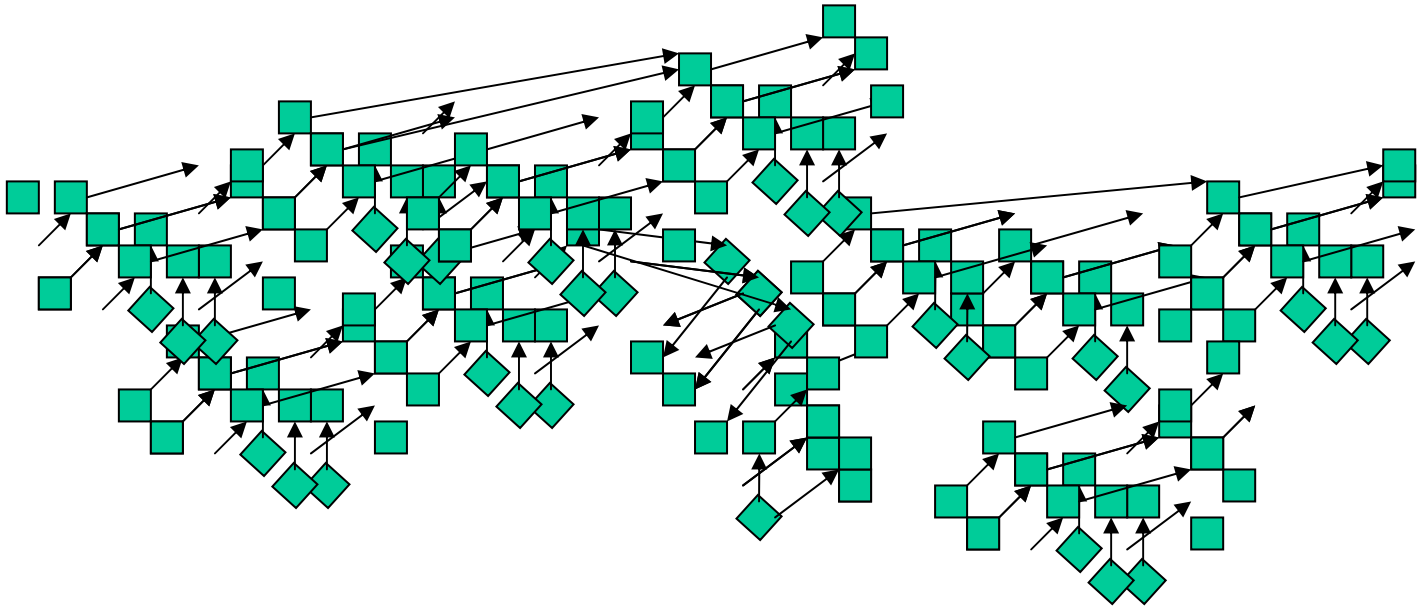
Choosing the inputs

- Critical sections of code are the control predicates that make execution diverge into different directions



- Inputs must be chosen so that when executing the program, correct branch is achieved or the control predicate p gets the aimed boolean values
- Note! All boolean values are not necessarily possible
  - Often not possible to achieve 100% multi-condition coverage e.g. (i<0 || i>100) ->
    (true, true) combination is not possible

Practical Relevance



Without a good white-box testing tool analyzing and controlling coverage testing is hopeless.

Example: Using coverage tools

- We used open-source coverage tool EMMA as part of the exercise 1 grading of the last year's course
  - http://emma.sourceforge.net/
  - *Just one tool, plenty of different tools available*

EMMA coverage tool

- Coverage metrics that Emma provides
  - Class, method, line, and basic block coverage
  - Basic block is a sequence of bytecode instructions without any jumps or jump targets
    - One line can include more than one basic blocks
  - Cannot measure branches or paths
  - but can detect on bytecode level partially covered lines

Example: Coverage of last year unit testing exercise

- Line (statement) coverage of Assignment 1
  - Age.java -> 63,7%
    - ~50 LOC
  - AgeComparator.java -> 95,5%
    - ~ 100 LOC
- Line (statement) coverage of Assignment 2
  - AgreementChoice.java -> 70,3%
    - ~ 50 - 100 LOC
  - AgreementChoiceComparator.java -> 80,5%
    - ~ 100 -200 LOC

Lets see examples of a coverage reports...

EMMA Coverage Report (generated Mon Oct 17 15:43:38 EEST 2005)

[all classes][electionmachine.candidate.fields]

COVERAGE SUMMARY FOR SOURCE FILE [Age.java]

name	method, %	block, %	line, %
Age.java	67% (2/3)	34% (10/29)	62% (5/8)

COVERAGE BREAKDOWN BY CLASS AND METHOD

name	method, %	block, %	line, %
class Age	67% (2/3)	34% (10/29)	62% (5/8)
getComparatorClass (): Class	0% (0/1)	0% (0/2)	0% (0/1)
validateValue (String): void	100% (1/1)	23% (5/22)	60% (3/5)
Age (String, Properties): void	100% (1/1)	100% (5/5)	100% (2/2)

```
214                             - criterionGroup))
215                     return CandidateComparator.SECOND_CANDIDATE_IS_BETTER_MATCH;
216             else
217                     return CandidateComparator.CANDIDATES_ARE_EQUAL_MATCH;
218
219         }
220
221         // Rule #3: Not in same group, on the edge
222         else {
223             if (Math.abs(arg0Index - criterionIndex) < Math.abs(arg1Index
224                             - criterionIndex))
225                     return CandidateComparator.FIRST_CANDIDATE_IS_BETTER_MATCH;
226             else if (Math.abs(arg0Index - criterionIndex) > Math.abs(arg1Index
227                             - criterionIndex))
228                     return CandidateComparator.SECOND_CANDIDATE_IS_BETTER_MATCH;
229             else {
230                     if (arg0Group == criterionGroup)
231                             return CandidateComparator.FIRST_CANDIDATE_IS_BETTER_MATCH;
232                     else if (arg1Group == criterionGroup)
233                             return CandidateComparator.SECOND_CANDIDATE_IS_BETTER_MATCH;
234                     else
235                             return CandidateComparator.CANDIDATES_ARE_EQUAL_MATCH;
236             }
237         }
238
239     }
240
241 }
```

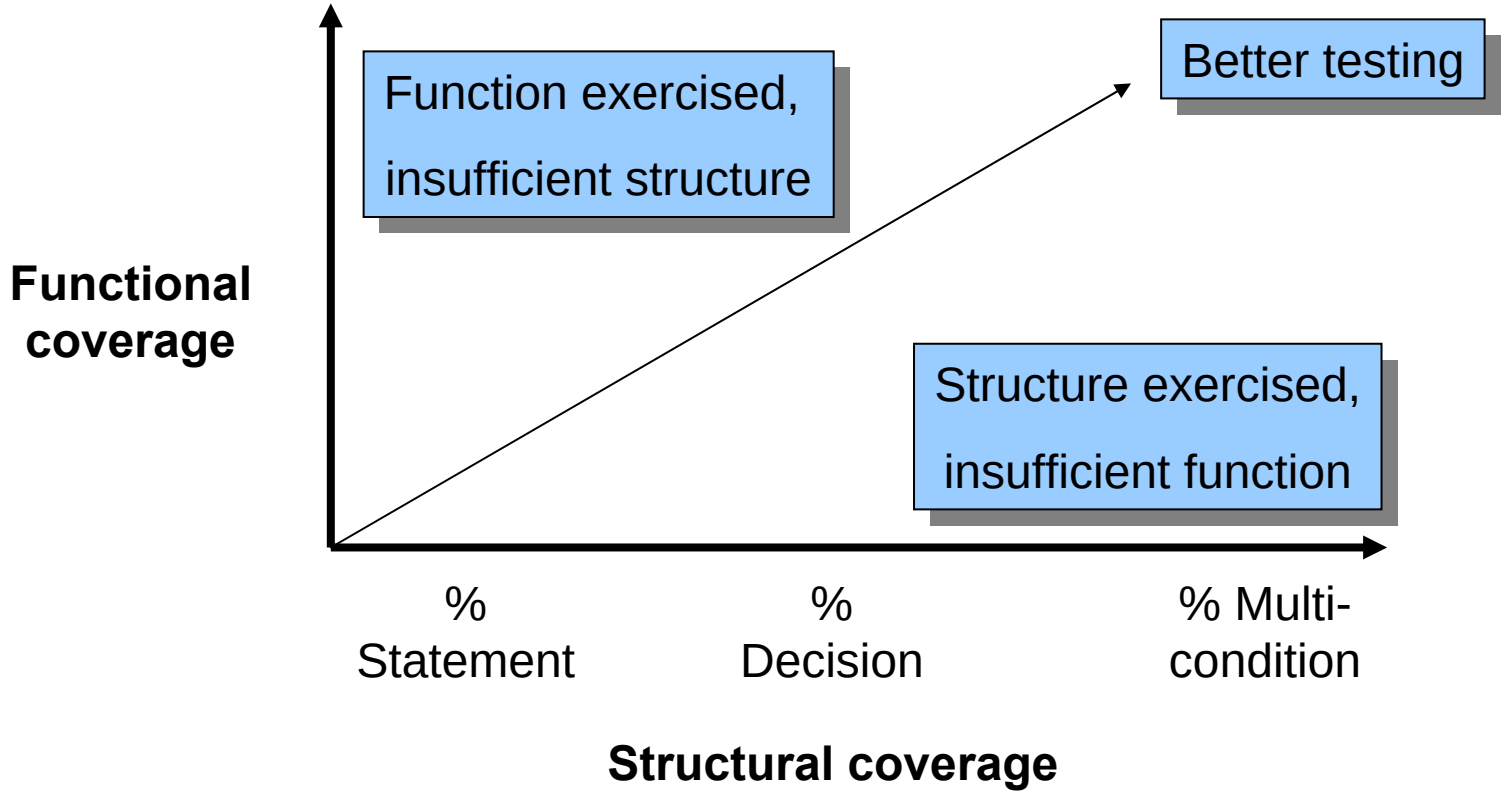
What structural coverage tool can tell us

- Missing tests
  - Do we need more tests?
- Weak spots in tests
  - Do we need better tests?
- Risky areas that are hard to cover
  - Should we try to cover those areas by other type of tests or careful reviews?
- Unreachable code, old code, debug code…
  - Should we remove the dead code?
  - Should we replace the debug code with automated tests?
- What can we assume if the coverage is high?
  - Only that our tests cover the code concerning the used coverage criteria
  - What can we say about the quality of our tests?

Coverage for measuring quality of test cases

- Coverage tools can be used to control and guide testing
- Coverage tools highlight non-covered areas of code
  - More tests needed
- Mainly applicable on unit testing level
  - Missing tests can be identified by reading coverage reports and code
  - Easy to create tests that exercise a specific code block
- Coverage only shows that the tests exist
  - It does not measure how well the test cases reveal defects
  - Executing ,e.g., all code branches does not guarantee that the tests actually show if the branches are incorrect

The test coverage trap



100% coverage does not mean 100% tested!

Structural coverage alone does not make good quality tests

Figure out examples of defects that would not necessarily be detected by coverage based testing

Why is structural coverage not enough

- Missing features
  - Missing code
- Timing and concurrency issues
- Different states of the software
  - Astronomical amount of different paths through the software
  - Sequences and order of executing different paths
  - Same path reveals different defects in different states
- Variations of input conditions
- Variations of data
- Qualities
  - Performance, load
  - Security
  - Usability
  - Compatibility, portability
  - ...

Each coverage metric provides a different view to coverage and helps to uncover different things.