Software module testing (component testing)

Istvan Majzik
majzik@mit.bme.hu

Budapest University of Technology and Economics
Dept. of Measurement and Information Systems

Typical development steps and V&V tasks

Requirement analysis

System specification
- Checking completeness, consistency, feasibility, verifiability
- Assuring traceability

Architecture design
- Trade-off analysis, interface analysis, fault effects analysis
- Model based quantitative evaluation

Module design
- Formal verification by (temporal logic based) model checking
- Equivalence checking

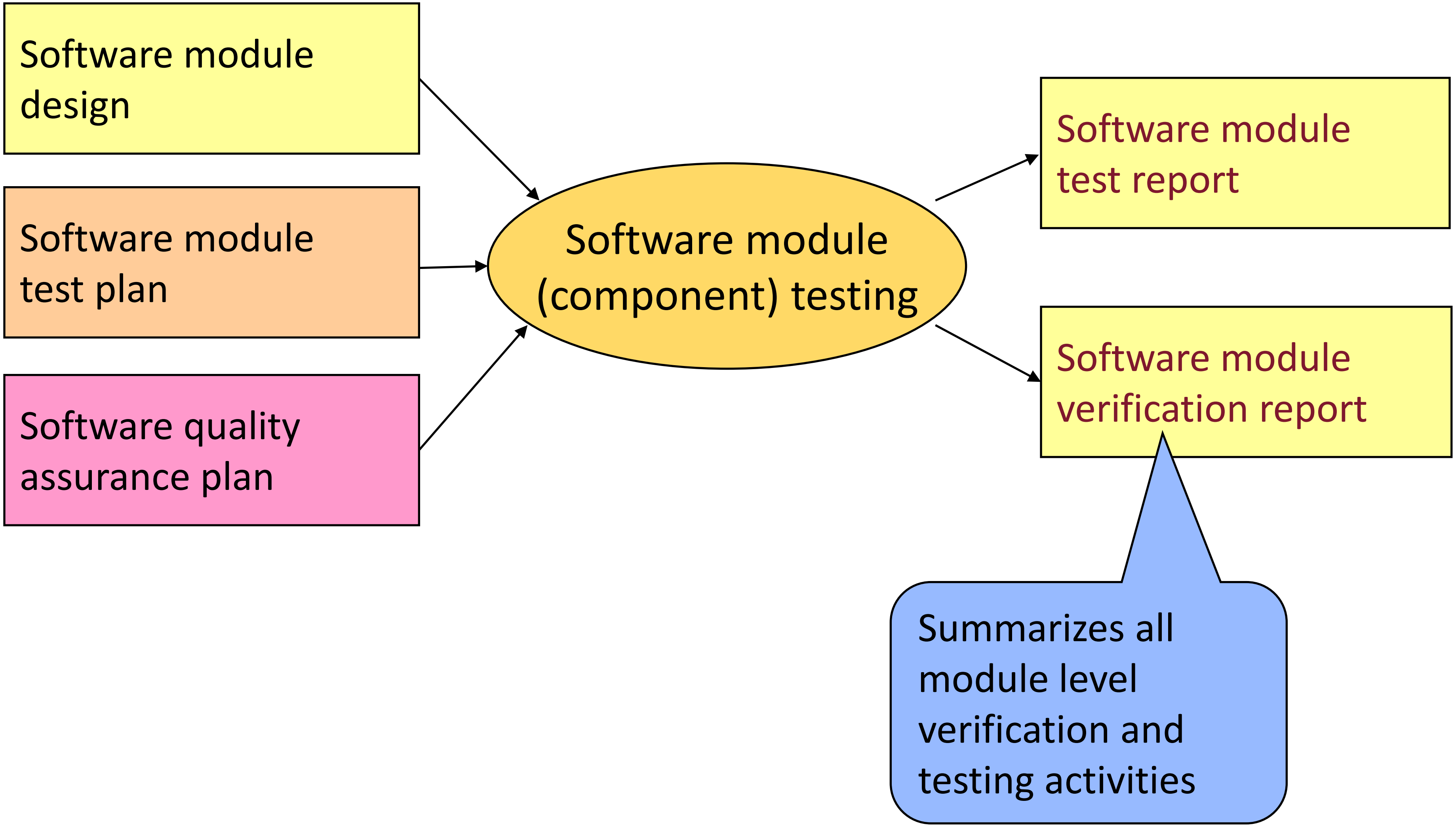
Module implementation
- Source code analysis
- Proof of program correctness by theorem proving
- Software model checking with abstraction
- Module (component) testing

System integration

System delivery

Operation, maintenance

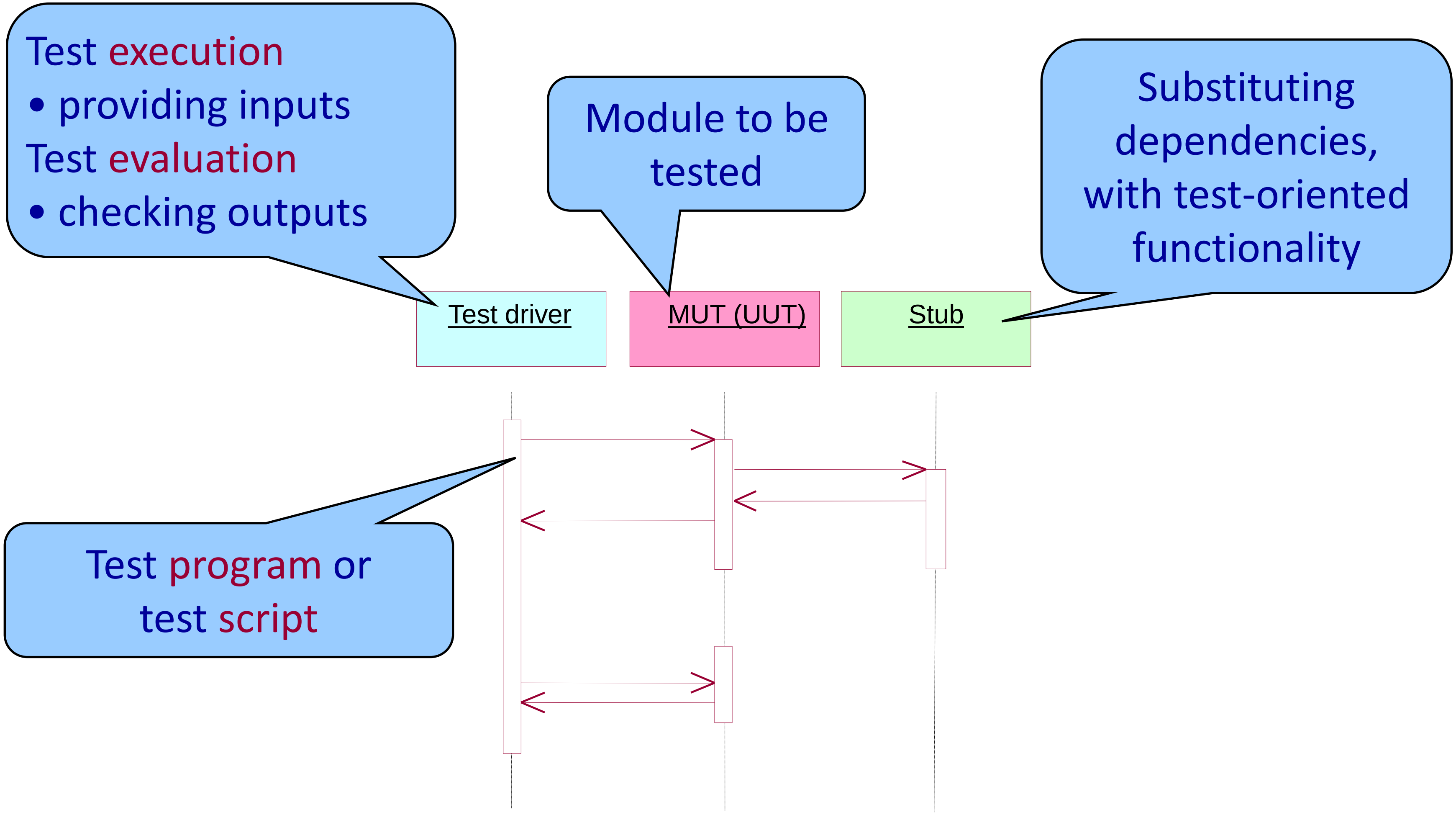
Inputs and outputs of the phase



Goals of testing

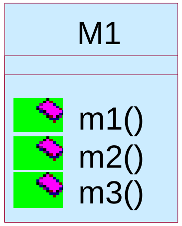
- Testing:
  - Running the program in order to detect faults
- Exhaustive testing:
  - Running programs in all possible ways (with all possible inputs)
  - Hard to implement in practice
- Observations:
  - Dijkstra: Testing is able to show the presence of faults, but not able to show the absence of faults.
  - Hoare: Testing can be considered as part of an inductive proof: If the program runs correctly for a given test input then it will run correctly in case of similar inputs.

Test environment: Module testing

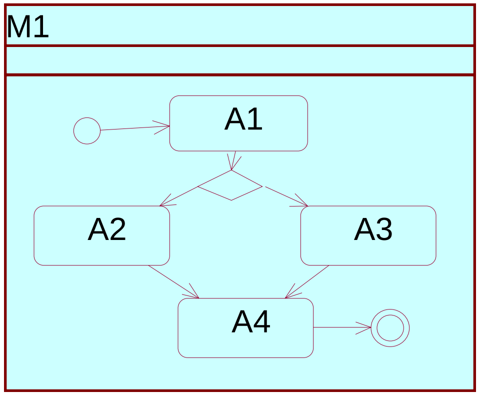


Test approaches

- Specification based (functional) testing
  - The system is considered as a "black box"
  - Only the external behaviour (functionality) is known (the internal behaviour is not)
  - Test goals: checking the existence of the specified functions and absence of extra functions



- Structure based testing
  - The system is considered as a white box
  - The internal structure (source) is known
  - Test goals: coverage of the internal behaviour (e.g., program graph)



Specification based testing (functional testing)

Goals and overview

Goals:

- Based on the functional specification,
- find representative inputs (test data)

for checking the correctness of the implementation

Overview of techniques:

1. Equivalence partitioning
2. Boundary value analysis
3. Cause-effect analysis
4. Combinatorial techniques
5. Finite state automaton based techniques
6. Use case based testing

Example: Requirements in standards (EN 50128)

- Software design and implementation:

TECHNIQUE/MEASURE		Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
14.	Functional/ Black-box Testing	D.3	HR	HR	HR	M	M
15.	Performance Testing	D.6	-	HR	HR	HR	HR
16.	Interface Testing	B.37	HR	HR	HR	HR	HR

- Functional/black box testing (D3):

1.	Test Case Execution from Cause Consequence Diagrams	B.6	-	-	-	R	R
2.	Prototyping/Animation	B.49	-	-	-	R	R
3.	Boundary Value Analysis	B.4	R	HR	HR	HR	HR
4.	Equivalence Classes and Input Partition Testing	B.19	R	HR	HR	HR	HR
5.	Process Simulation	B.48	R	R	R	R	R

1. Equivalence partitioning

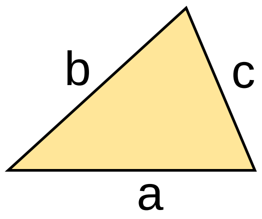
- Input and output equivalence classes
  - Data that are expected to cover the same faults (execute the same part of the program)
  - Each equivalence class: represented by a test input
  - The correctness in case of the remaining inputs follows from the principle of induction
- Test data selection is a heuristic procedure
  - Input data that trigger the same service
  - Valid and invalid input data

    → valid and invalid equivalence classes

    - Invalid data: Robustness testing

Example: Equivalence classes (partitions)

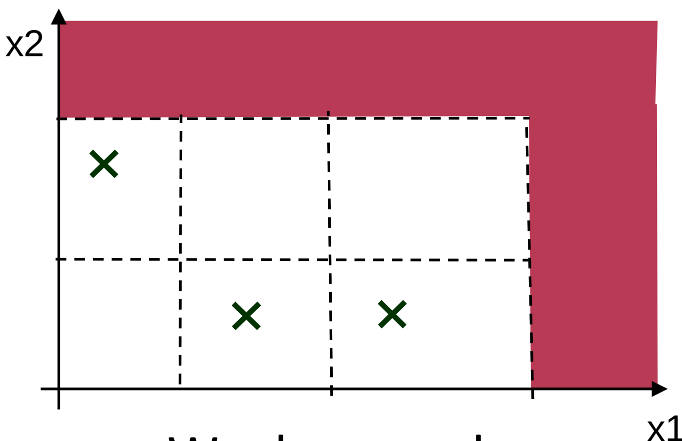
- Classic example: Triangle characterization program
  - Inputs: Lengths of the sides (here: 3 integers)
  - Outputs: Equilateral, isosceles, scalene



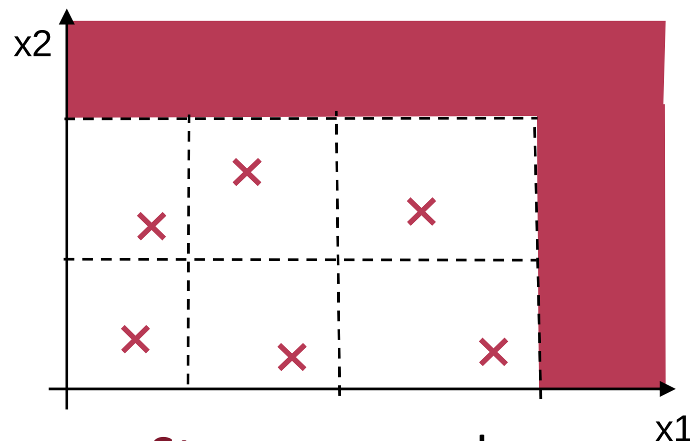
- Test data for equivalence classes
  - Equilateral: 3, 3, 3
  - Isosceles: 5, 5, 2 (similarly for the other sides)
  - Scalene: 5, 6, 7
  - Not a triangle: 1, 2, 5 (similarly for the other sides)
  - Just not a triangle: 1, 2, 3
  - Invalid inputs
    - Zero value: 0, 1, 1
    - Negative value: -3, -5, -3
    - Not an integer: 2, 2, ’a’
    - Less inputs than needed: 3, 4
  - …

Using equivalence classes

- Tests in case of having several inputs:
  - Valid (normal) equivalence classes:
    Test data should cover as much equivalence classes as possible
  - Invalid equivalence classes:
    First covering each invalid equivalence class separately, then combining them systematically
- Weak and strong equivalence classes:



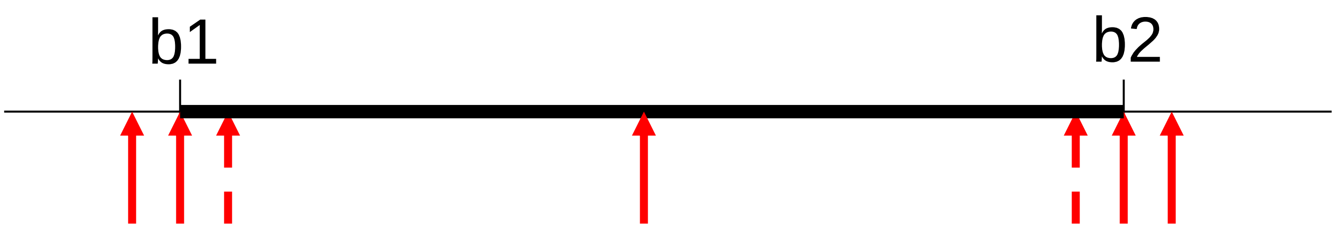
Weak normal equivalence classes



Strong normal equivalence classes

2. Boundary value analysis

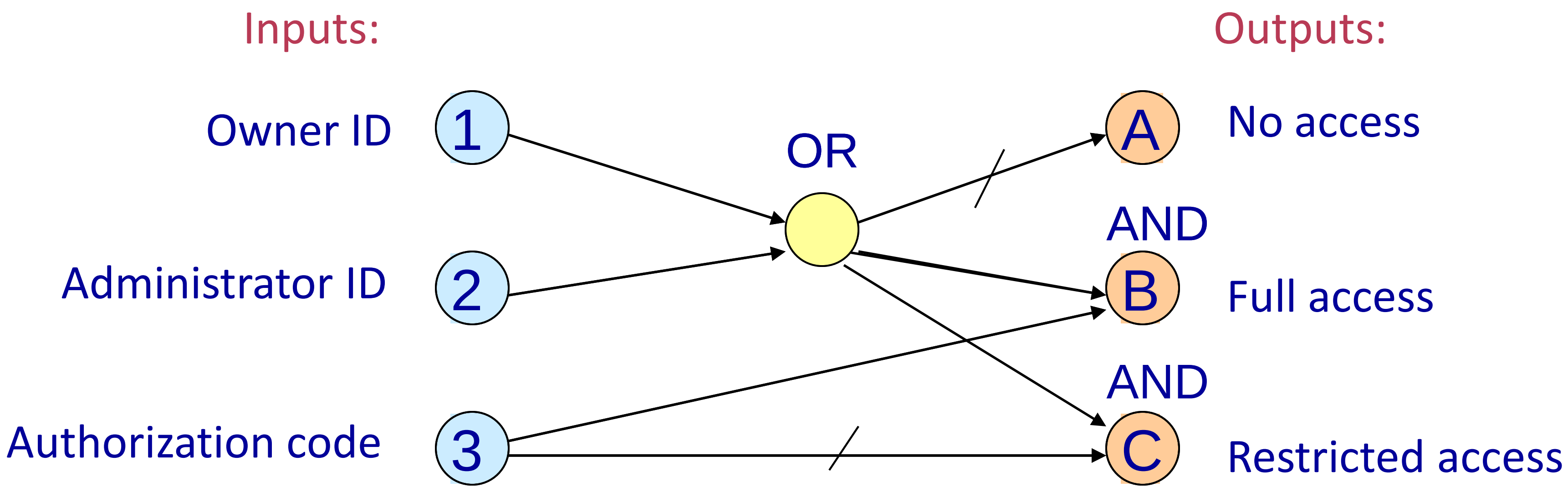
- Examining the boundaries of data partitions (equivalence classes)
  - Input and output partitions are also examined
  - To be applied for upper/lower bounds
- Typical problems found by testing
  - Incorrect relational operations in the code
  - Incorrect input/output conditions in loops
  - Incorrect size of data structures (access), …
- Typical test data:
  - A boundary requires 3 tests, a partition requires 5-7 tests:



3. Cause-effect analysis

- Examining the relation of inputs and outputs, if it is combinational
  - Causes: input equivalence classes
  - Effects: output equivalence classes
  - Using Boolean variables to represent these
- Boole-graph: relations of causes and effects
  - AND, OR relations
  - Implicitly: invalid combinations
- Decision table: Covering the Boole-graph
  - Rows: Inputs and corresponding outputs
  - Columns represent test data

Example: Cause-effects analysis



		T1	T2	T3
Inputs	**1**	0	1	0
	2	1	0	0
	3	1	1	1
Outputs	**A**	0	0	1
	B	1	1	0
	C	0	0	0

4. Combinatorial techniques

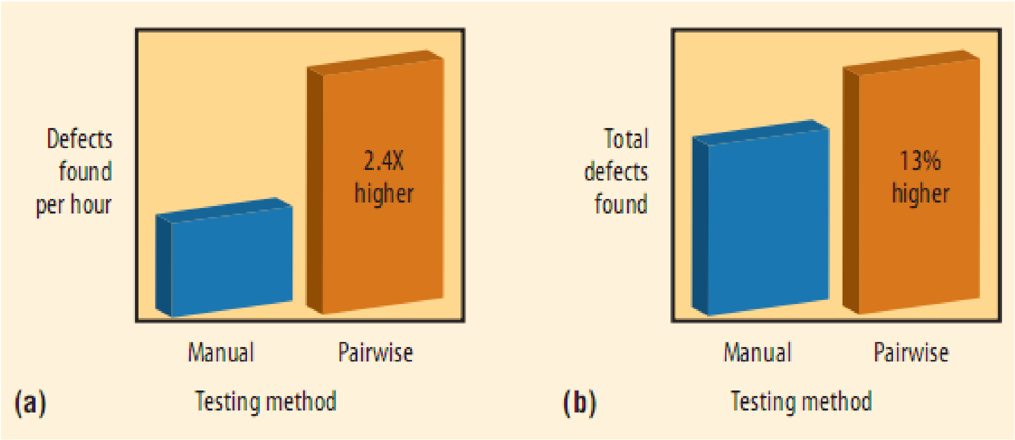
Goal: Testing the combinations of parameters
  - Problems are often caused by rare combinations
  - But the number of all combinations can be high

- "Best guess" ad-hoc testing
  - Based on intuition, covering typical faults
- "Each choice" testing
  - All parameter values shall be tested (at least once)
- "n-wise" testing
  - For each n parameters (selected out of all parameters) testing all possible combinations of their potential values
  - "Pairwise" testing: Special case with n = 2
  - Tool support: e.g., http://www.pairwise.org

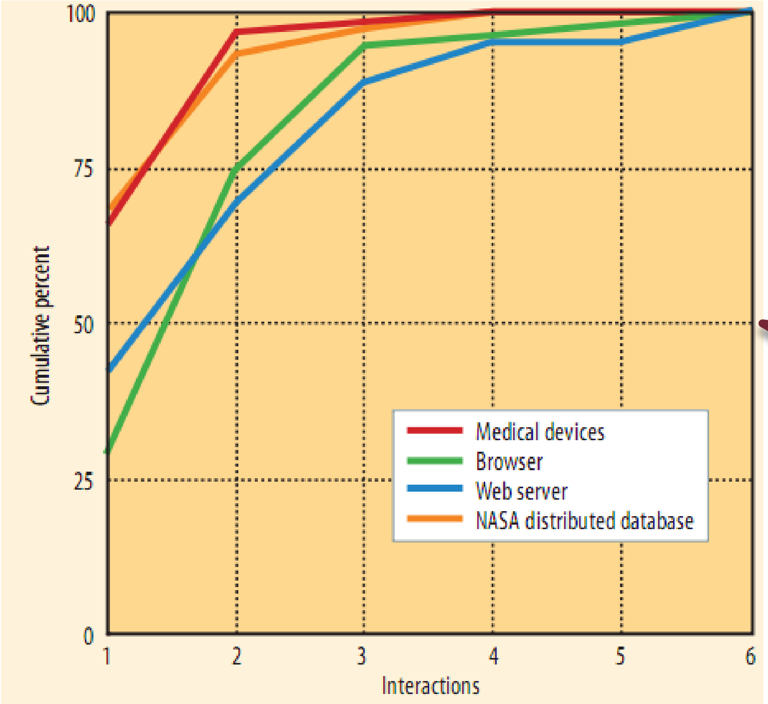
Example: Pairwise testing

- Given input parameters and potential values:
  - OS: Windows, Linux
  - CPU: Intel, AMD
  - Protocol: IPv4, IPv6
- All combinations:
  - 8 combinations are possible
- "Pairwise" testing: A potential test suite:
  - T1: Windows, Intel, IPv4
  - T2: Windows, AMD, IPv6
  - T3: Linux, Intel, IPv6
  - T4: Linux, AMD, IPv4

Efficiency of n-wise testing



Comparing ad hoc and pairwise testing (10 projects)

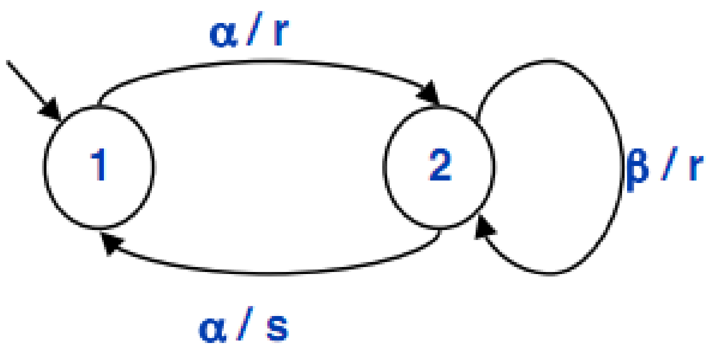


Many faults are triggered by combinations of 2 or 3 parameters

Source: R. Kuhn et al. „Combinatorial Software Testing", IEEE Computer, 42:8, 2009

5. Finite state automaton based testing

- Specification is given as a finite state automaton
- Typical testing goals:
  - Covering (testing) all states, all transitions
  - Trying also transitions that are not allowed (implicit)



- Problems:
  - Determining the state of the tested system
  - Setting initial state
- Methods
  - Automated test input generation (see later)

6. Use case based testing

- Deriving test cases from the specified use cases
  - Use cases: often given with preconditions and post-conditions
  - Test oracles: checking the post-conditions
- Typical test cases:
  - Main path (“happy path”, “mainstream”): 1 test case
  - Alternative paths: separate test cases
  - Robustness testing: Tests for violating preconditions
- Mainly higher level testing
  - System tests, acceptance tests

Using the methods together

Typical application of the basic methods:

1. Equivalence partition based
2. Boundary value analysis
3. Cause-effect analysis, or combinatorial, or finite state automaton based (depending on the specification)

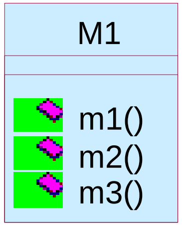
Extension: Random testing

- Generating random test data
  - Fast test generation, with low computational effort
- Fault coverage cannot be estimated
- Difficult to evaluate the test results:
  - Computing the expected results (simulation)
  - Only “smoke checking” (identifying rough failures like crash)

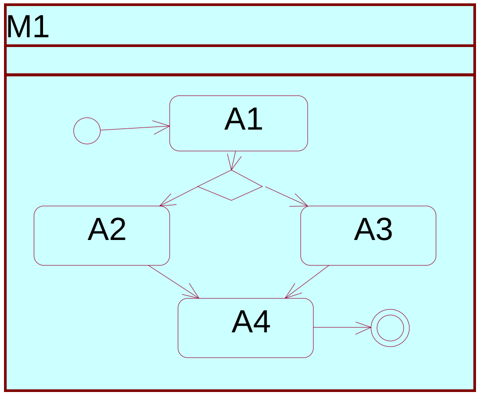
Structure based testing

Test approaches

- Specification based (functional) testing
  - The system is considered as a “black box”
  - Only the external behaviour (functionality) is known (the internal behaviour is not)
  - Test goals: checking the existence of the specified functions and absence of extra functions

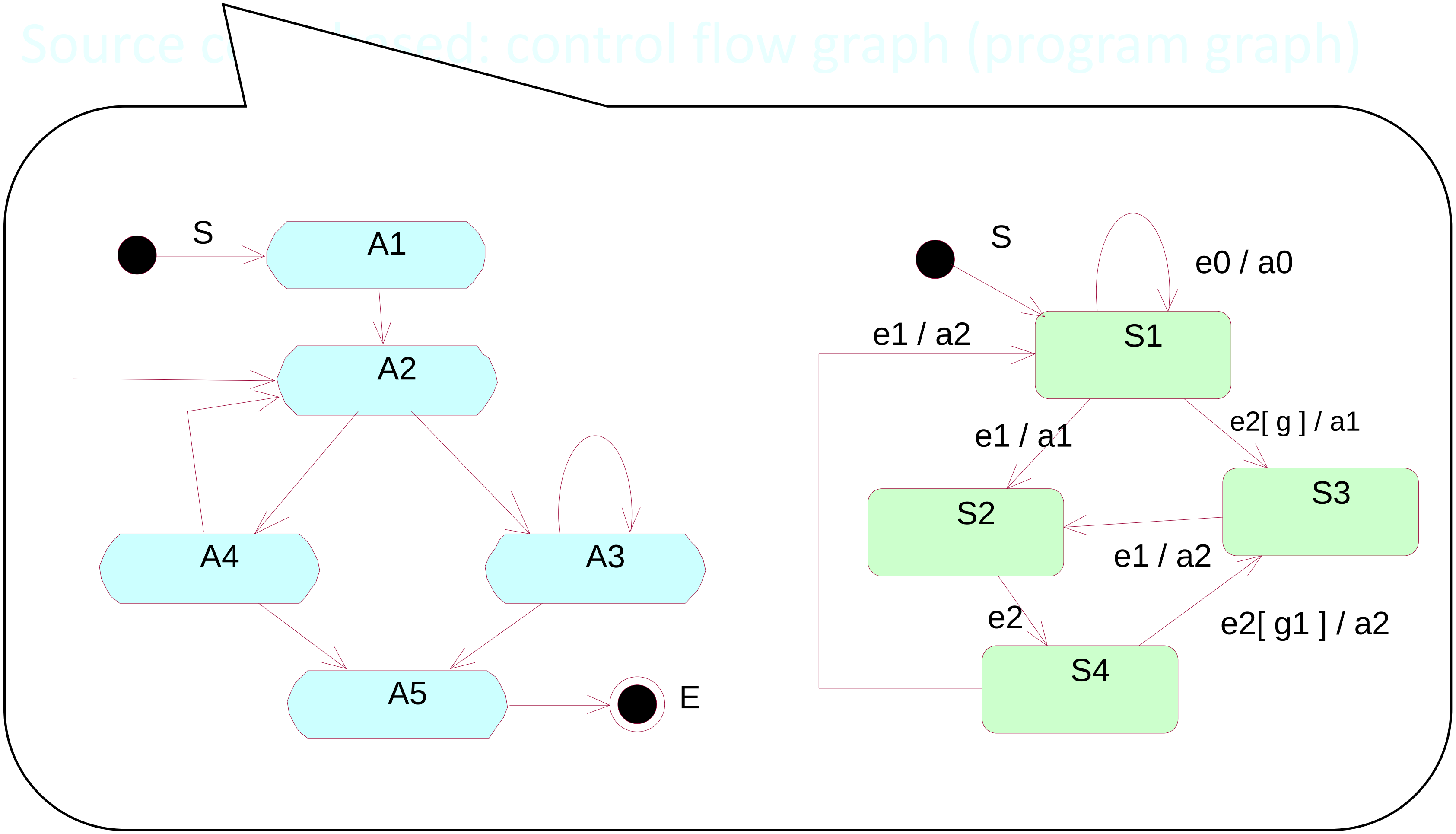


- Structure based testing
  - The system is considered as a white box
  - The internal structure (source) is known
  - Test goals: coverage of the internal behaviour (e.g., program graph)



The internal structure

- Well-specified representation:
  - Model-based: state machine, activity diagram
  - Source c ased: control flow graph (program graph)



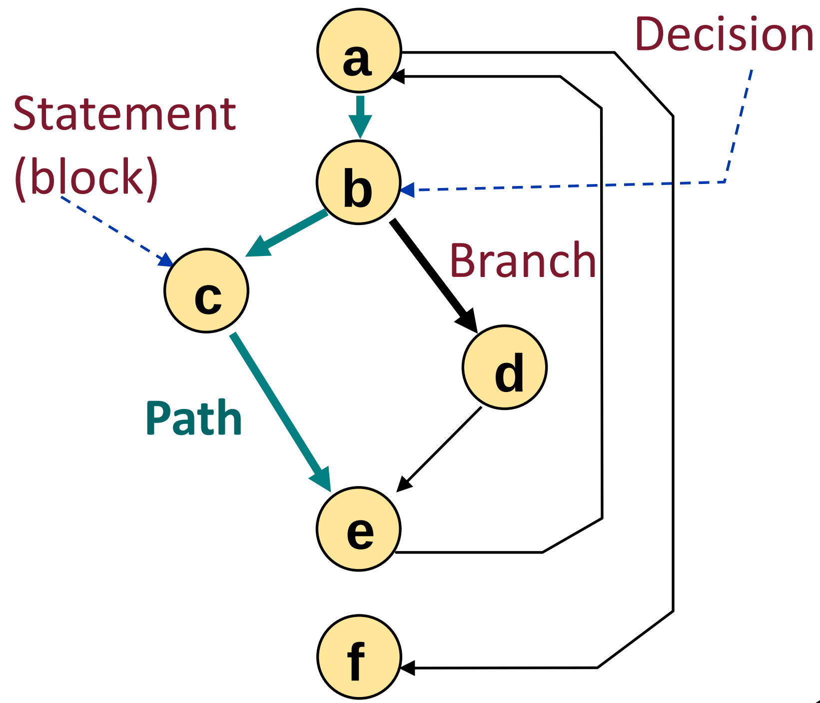
The internal structure

- Well-specified representation:
  - Model-based: state machine, activity diagram
  - Source code based: control flow graph (program graph)

Source code:

```
a:  for (i=0; i<MAX; i++) {
b:        if (i==a) {
c:              n=n-i;
          } else {
d:              m=n-i;
          }
e:        printf(“%d\n”,n);
    }
f:  printf(“Ready.”)
```

Control flow graph:



Test coverage metrics

Characterizing the quality of the test suite:
Which testable elements were tested

1. Statements → Statement coverage
2. Decisions → Decision coverage
3. Conditions → Condition coverage
4. Execution paths → Path coverage

This is not fault coverage!

Standards require test coverage (DO-178B, EN 50128,...)

- 100% statements coverage is a typical basic requirement

Overview of test coverage criteria

- Control flow based test coverage criteria
  - Statement coverage
  - Decision coverage
  - Condition coverage (several metrics)
  - Path coverage
- Data flow based test coverage criteria
  - Definition – usage coverage
  - Definition-clear path coverage
- Combination of techniques

Basic concepts

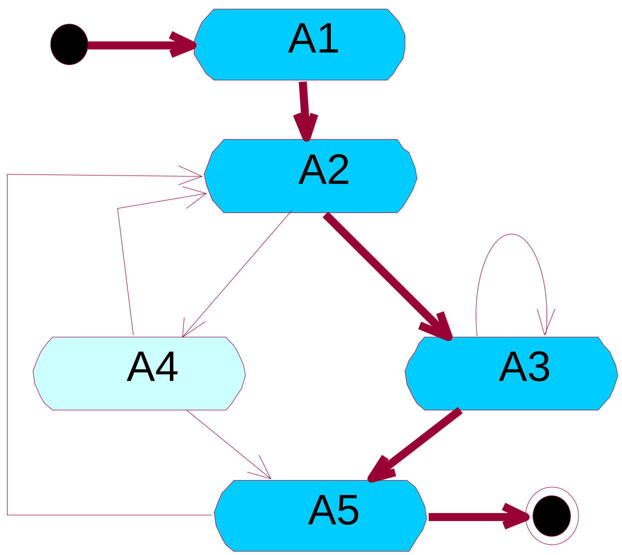
- Statement
- Block
  - A sequence of one or more consecutive executable statements without branches
- Condition
  - Logical expression without logical operators (AND, OR, …)
- Decision
  - Logical expression consisting of one or more conditions combined by logical operators (AND, OR, …)
  - Determines branches in if(…), while(…), etc.
- Path
  - A sequence of executable statements of a component, typically from an entry point to an exit point

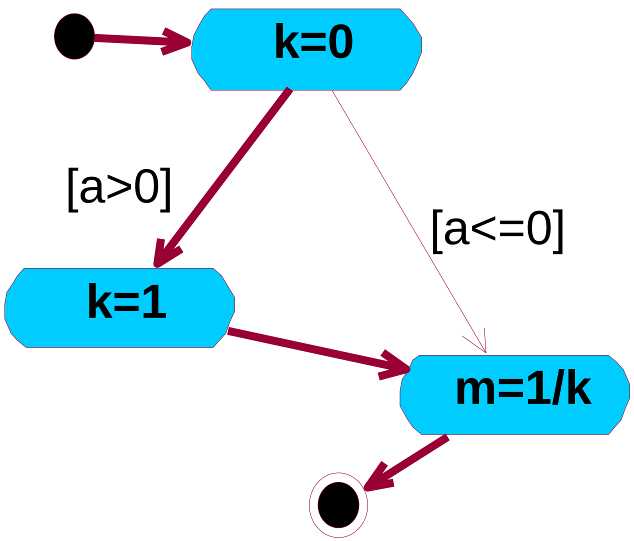
1. Statement coverage

Definition:

$$\frac{\text{Number of executed statements during testing}}{\text{Number of all statements}}$$

Does not take into account branches without statements





Statement coverage: 80%

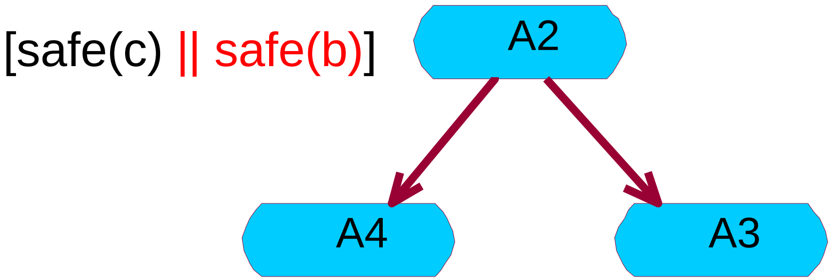
Statement coverage: 100%

2. Decision coverage

Definition:

$$\frac{\text{Number of decision branches reached during testing}}{\text{Number of all potential decision branches}}$$

Does not take into account all combinations of conditions



100% decision coverage
is possible without setting
safe(b) = true

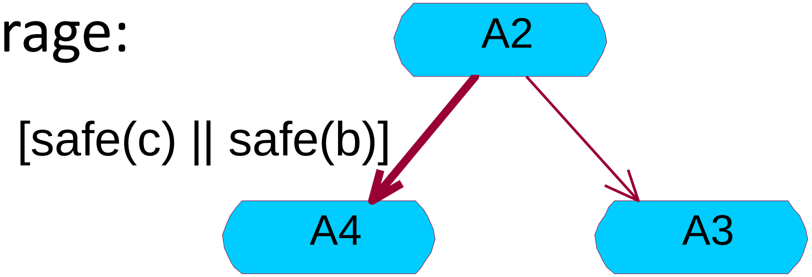
3. Condition coverage

Generic definition:

$$\frac{\text{Number of tested combinations of conditions}}{\text{Number of targeted combinations of conditions}}$$

Definitions (regarding the “targeted combinations”):

- Every condition is set to both true and false during testing
  - Does not yield 100% decision coverage!
  - Example for 100% condition coverage:
    1. safe(c) = true, safe(b) = false
    2. safe(c) = false, safe(b) = true



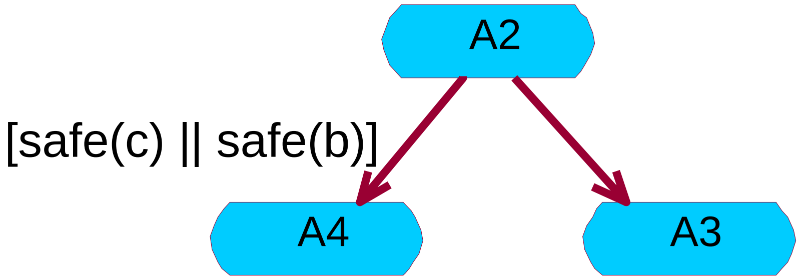
- Every condition is evaluated to both true and false
  - Not the same as above due to lazy evaluation

4. Condition/decision coverage (C/DC)

- Definition:
  - Each decision takes every possible outcome (branch)
  - Each condition in a decision takes every possible outcome

Example for 100% C/DC coverage:

1. safe(c) = true, safe(b) = true
2. safe(c) = false, safe(b) = false



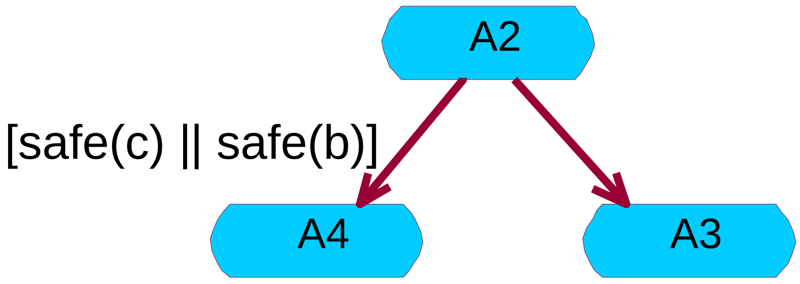
Does not take into account whether the condition has any effect (e.g., when safe(c) = false, changing safe(b) to true)

5. Modified condition/decision coverage (MC/DC)

- Definition:
  - Each decision takes every possible outcome (branch)
  - Each condition in a decision takes every possible outcome
  - Each condition in a decision is shown to independently affect the outcome of the decision

Example for 100% MC/DC coverage:

1. safe(c) = true, safe(b) = false
2. safe(c) = false, safe(b) = true
3. safe(c) = false, safe(b) = false

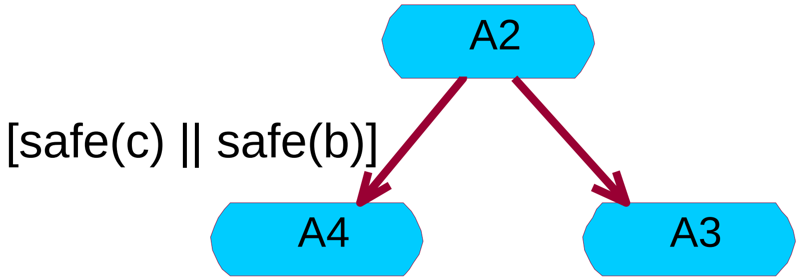


6. Multiple condition coverage

- Definition:
  - All combinations of conditions is tested
    - For n conditions: 2^n test cases may be necessary (less with lazy evaluation)
    - Sometimes not practical (e.g. in avionics systems there are programs with more than 30 conditions in a decision)

100% multiple condition coverage:

1. safe(c) = true, safe(b) = false
2. safe(c) = false, safe(b) = true
3. safe(c) = false, safe(b) = false
4. safe(c) = true, safe(b) = true



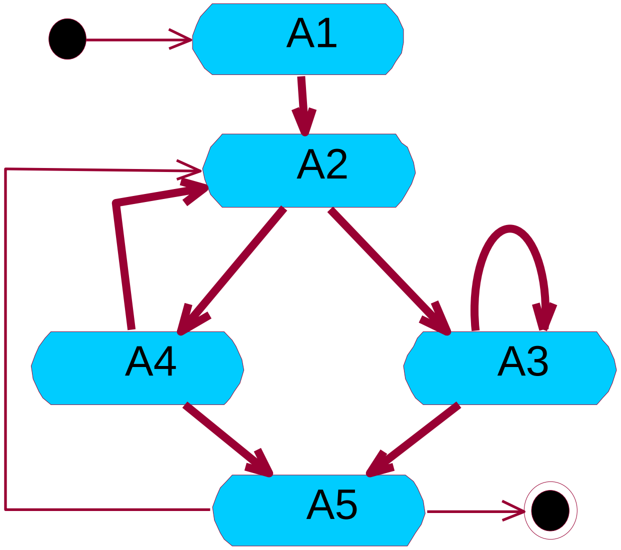
7. Basic path coverage

Definition:

$$\frac{\text{Number of independent paths traversed during testing}}{\text{Number of all independent paths}}$$

100% path coverage implies:

- 100% statement coverage, 100% decision coverage
- 100% multiple condition coverage is not implied



Path coverage: 80%

Statement coverage: 100%

A structure based testing technique

- Goal: Covering independent paths
  - Independent paths from the point of view of testing: There is a statement or decision branch in the path, that is not included in the other path
- The maximal number of independent paths:
  - CK: cyclomatic complexity
  - In regular control flow graphs (connected, single entry/exit): $CK(G)=E-N+2$, where

    E: number of edges

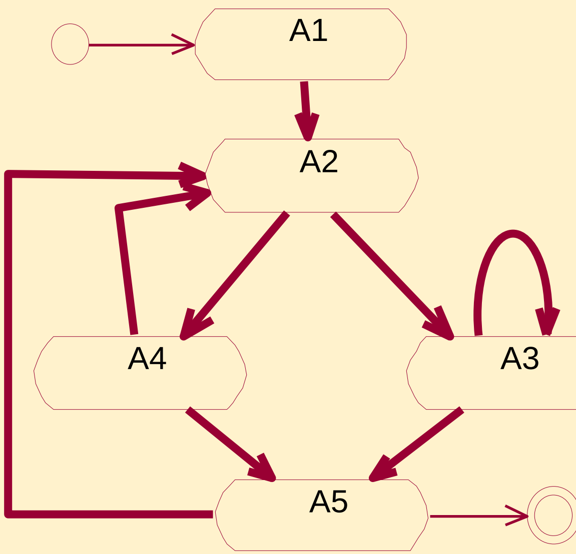
    N: number of nodes

    in the control flow graph G
- The set of independent graphs is not unique

A structure based testing technique

- Goal: Covering inde
  - Independent path
    There is a stateme
    that is not include
- The maximal num
  - CK: cyclomatic c
  - In regular cont :
    CK(G)=E-N+2, wher
    - E: number of edges
    - N: number of nodes

      in the control flow graph G
- The set of independent graphs is not unique



N=5,
E=8,
CK=5,
Max. 5 independent paths!

Generating structure based test sequences

- Conceptual algorithm:
  - Selecting maximum CK independent paths
  - Generating inputs to traverse these paths, each after the other
- Problems:
  - Not all paths can be traversed: Conditions along the selected path may be contradictory
  - Loops: Loop executions shall be limited (minimized)
- There are no fully automated tools to generate test sequences for 100% path coverage
  - Symbolic execution: With SMT solver
  - Limitations: Loops, data types, external libraries, …

Other coverage metrics (examples)

- Loop
  - Loops executed 0 (if applicable), 1, or multiple times
- Race
  - Multiple threads executed on the same block of statements
- Relational operator
  - Boundary values tried in case of relational operators
- Weak mutation
  - Tests for detecting the mutation of operators or operands
- Table
  - Jump tables (state machine implementation) testing
- Linear code sequence and jump
  - Covering linear sequences in the source code
    (with potential branches but executed in linear order)
- Object code branch
  - Machine instruction level coverage of conditional branches

Example: Testing for control flow based coverage

```
Product getProduct(String name, Category cat){
  if (name == null || !cat.isValid)
     throw new IllegalArgumentException();

  Product p = ProductCache.getItem(name);

  if (p == null){
     p = DAL.getProduct(name, cat);
  }

  return p;
}
```

Exercise: Generate test cases for 100% statement coverage, decision coverage, and C/DC coverage

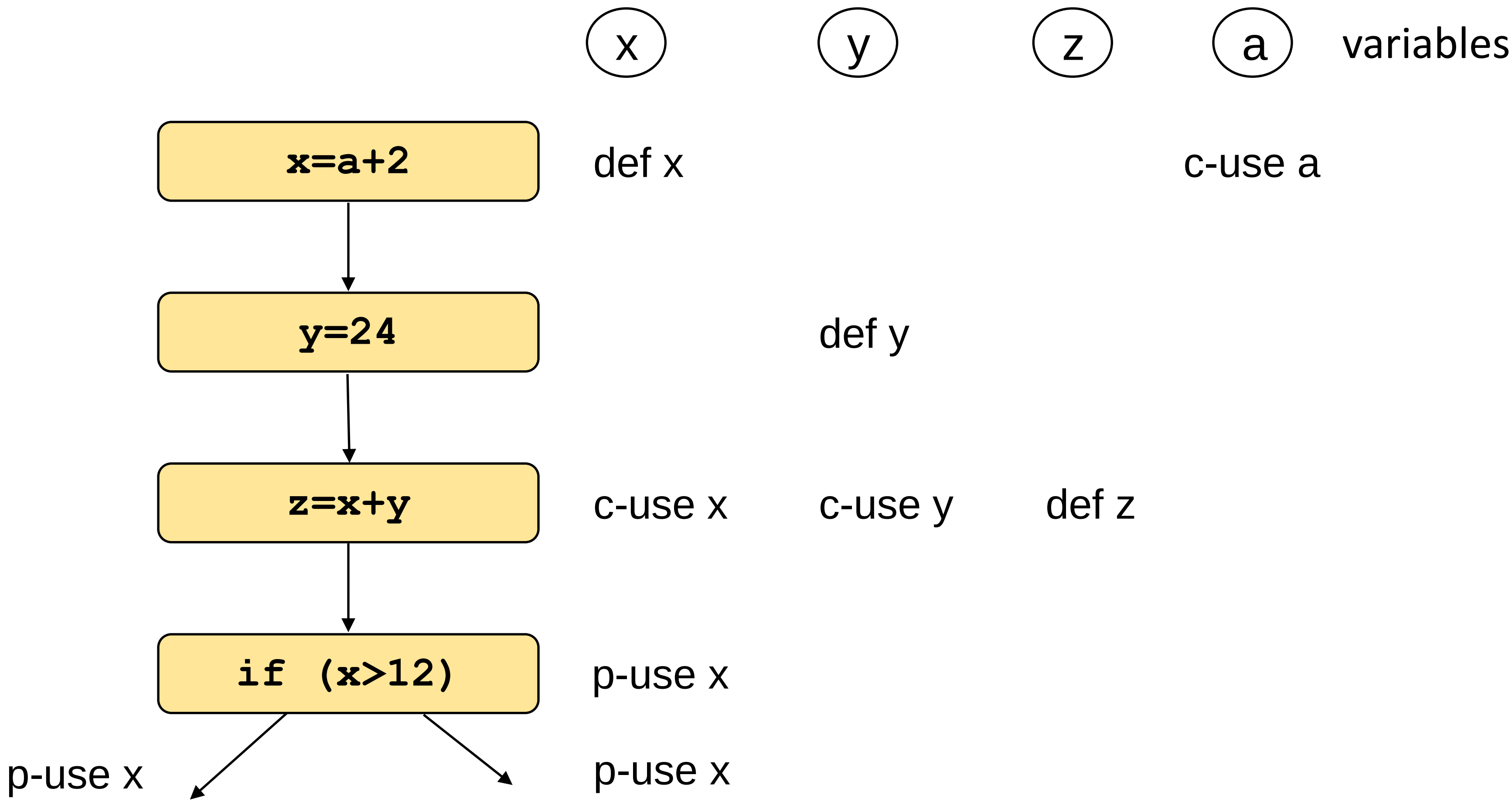
Overview of test coverage criteria

- Control flow based test coverage criteria
  - Statement coverage
  - Decision coverage
  - Condition coverages
  - Path coverage
- Data flow based test coverage criteria
  - Definition – usage coverage
  - Definition-clear path coverage
- Combination of techniques

Data flow based test criteria

- Goals of testing
  - Definition (value assignment) and use of the variables
  - Checks: Is there an incorrect assignment? Is it used in incorrect way?
- Labeling the program graph:
  - def(v): definition of variable v (by assigning a value)
  - use(v): using variable v
    - p-use(v): using v in a predicate (for a decision)
    - c-use(v): using v in computation
- Notation for paths:
  - def-clear v path: there is no def v label
  - def-use v (shortly d-u v) path:
    - Starts with def v label, ends with p-use v or c-use v label
    - Between these there is a def-clear v path
    - There is no internal loop (or the full d-u v path is a loop)

Example: Labeling the program graph



All-defs coverage criterion

- **All-defs**:

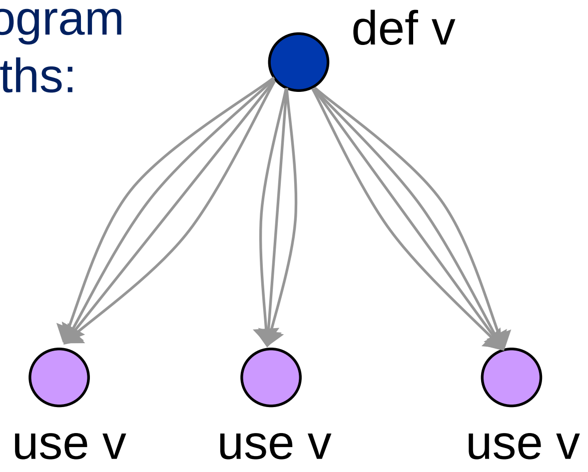
  For all **v** variables, from all **def v** statements:

  At least one **use v** statement is reached

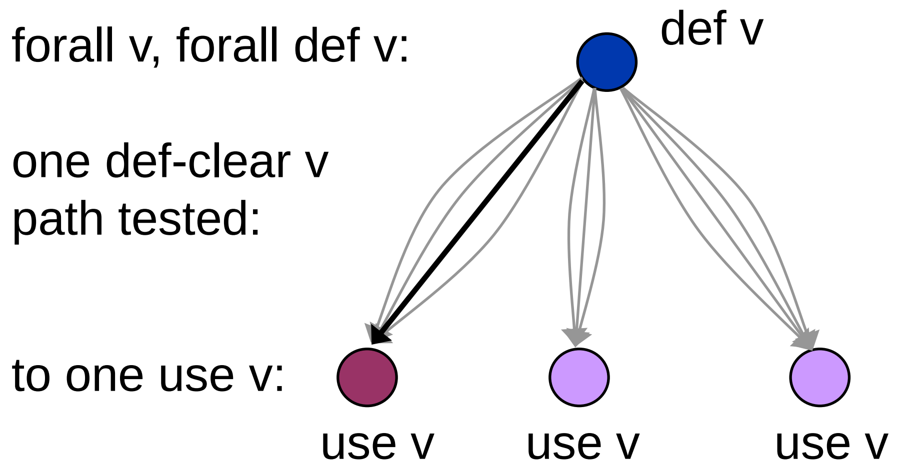
  by at least one **def-clear v** path

  (here **use v** may be either **p-use v** or **c-use v**)

Representing program paths:



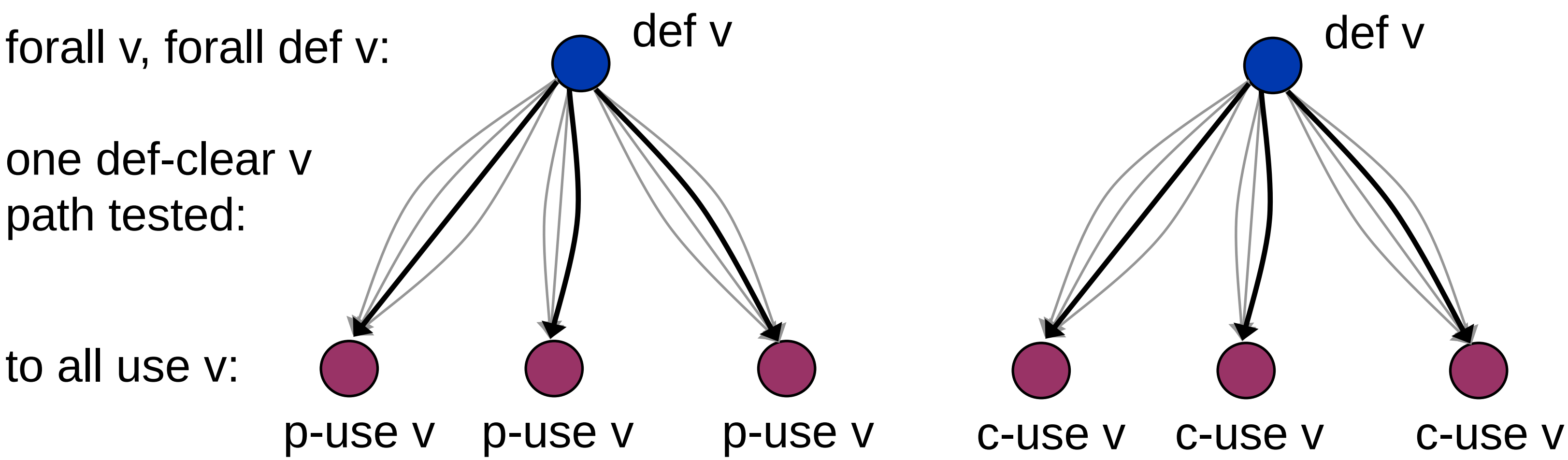
All-defs coverage:



All-p-uses, all-c-uses, all-uses criteria

- All-p-uses / all-c-uses:

  For all **v** variables, from all **def v** statements:
  All **p-use v / c-use v** statements are reached
  by at least one **def-clear v** path



- All-uses:

  For all **v** variables, from all **def v** statements:
  All **use v** statements are reached by
  at least one **def-clear v** path

All-paths and all-du-paths criteria

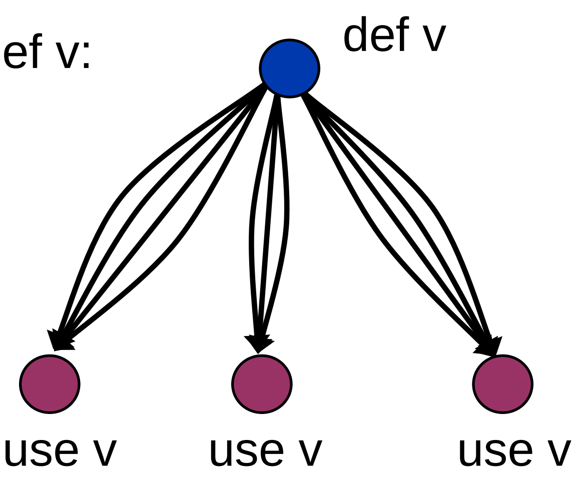
- **All-paths**:
  - For all **v** variables, from all **def v** statements :
    To all **use v** statements all executable **def-clear v** paths are tested
  - In case of loops multiple executions are distinguished
- **All-du-paths**:
  - For all **v** variable, from all **def v** statements:
    To all **use v** statements all **d-u v** paths are tested

All-paths coverage:

forall v, forall def v:

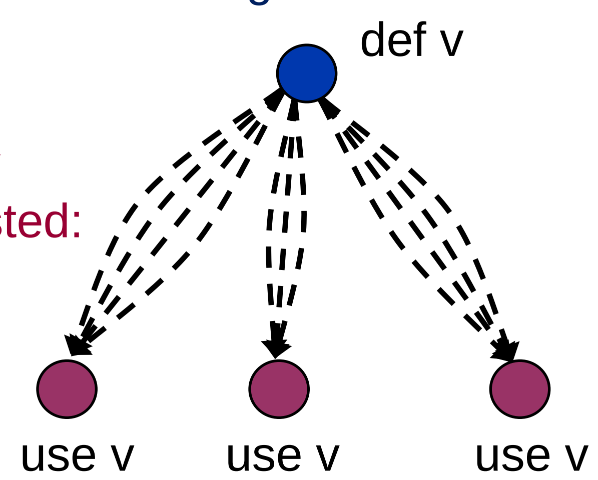
all def-clear v
path tested:

to all use v:

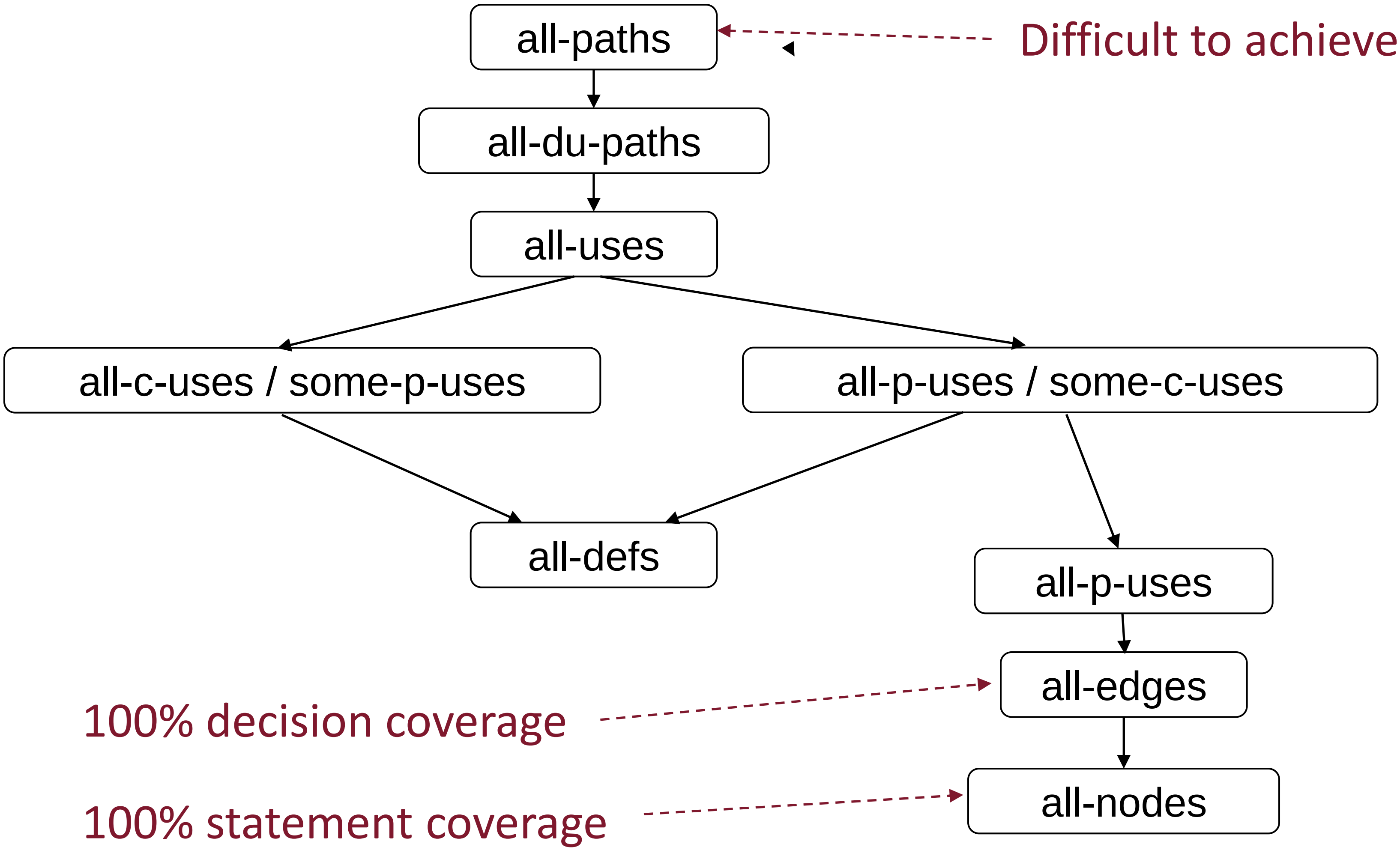


All-du-paths coverage:

all d-u v
path tested:



Hierarchy of data flow based test coverage criteria



Using test coverage metrics

- What are these good for?
  - Finding parts of the program (source code) where testing is weak
    - Test suite shall be extended
  - Finding redundant test cases (that cover the same part of the program)
    - Data dependency shall be considered: different types of faults can be tested by different data on the same path
  - Indirect measure of code quality is the coverage of successful tests
  - Rather, measure of the completeness of the test suite
    - Testing phase may be terminated on the basis of the coverage
- What are these not good for?
  - To identify requirements that were not implemented
  - To simply “cover” program parts without considering the expectations

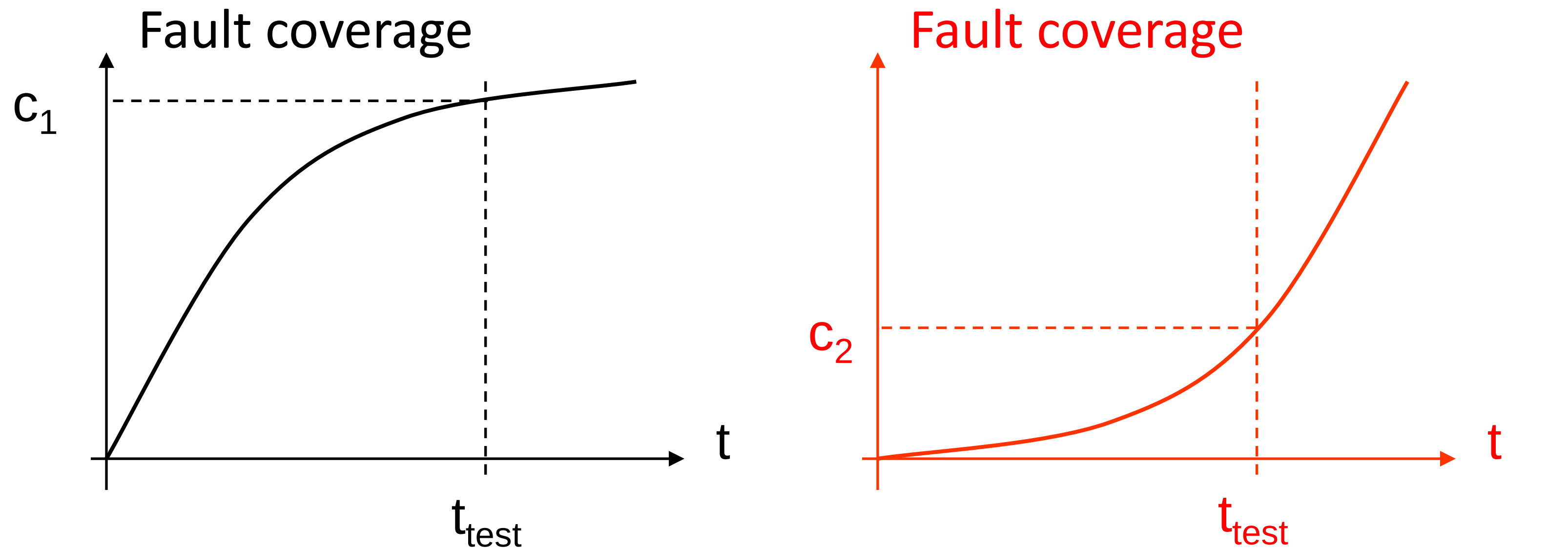
Execution of test cases

Execution order (prioritization) of the test cases:

If the number of faults is expected to be low:

First the more efficient tests (with higher fault coverage)

- Covering longer paths
- Covering more difficult decisions



Summary: Module test design techniques

- Specification and structure based techniques
  - Many (more or less orthogonal) techniques
  - Specification based testing is the primary approach
- Only basic techniques are used commonly
  - Exception: Safety-critical systems
  - E.g. DO178-B requires MC/DC coverage analysis
- Combination of techniques is useful:
  - Example (Microsoft report):

    Specification based testing: 83% code coverage achieved

    + exploratory testing: 86% code coverage

    + structure based testing: 91% code coverage