Testing:
Test design and testing process

Zoltán Micskei

Based on István Majzik's slides
Dept. of Measurement and Information Systems

Overview

- Testing basics
  - Goals and definitions
- Test design
  - Specification based (functional, black-box) testing
  - Structure based (white-box) testing
- Testing process
  - Module testing
  - Integration testing
  - System testing
  - Validation testing

Basic definitions

What is the goal of testing?

What are the costs of testing?

What can be automated?

Definition of testing

"An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component."

IEEE Std 829-2008

Lots of other, conflicting definitions!

Basic concepts

Specification, requirements → Test cases → Test execution → Verdicts

- Test case
  - a set of test inputs, execution conditions, and expected results developed for a particular objective
- Test suite
  - several test cases for a component or system under test
- Test oracle
  - A source to determine expected results to compare with the actual result
- Verdict: result (pass / fail /error…)

Remarks on testing

Testing != Debugging

Exhaustive testing:

- Running the program in all possible ways (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 input then it will run similarly correctly in case of similar inputs.

Practical aspects of testing

- Testing costs may reach 50% of the development costs!
  - Test data generation
  - Test code implementation
  - Running the tests
  - Evaluation of the results

Typically manual work

May be automated

- Testing embedded systems:
  - Cross-development (different platforms)
  - Platform related faults shall be considered (integration)
  - Performance and timing related testing are relevant
- Testing safety-critical systems:
  - Prescribed techniques
  - Prescribed test coverage metrics

Testing in the standards (here: 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

Testing in the standards (here: EN 50128)

- Performance testing (D6):

TECHNIQUE/MEASURE	Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1. Avalanche/Stress Testing	B.3	-	R	R	HR	HR
2. Response Timing and Memory Constraints	B.52	-	HR	HR	HR	HR
3. Performance Requirements	B.46	-	HR	HR	HR	HR

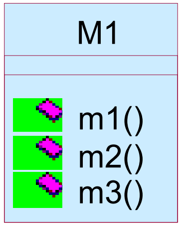
Test design

How can be test data selected?

Test approaches

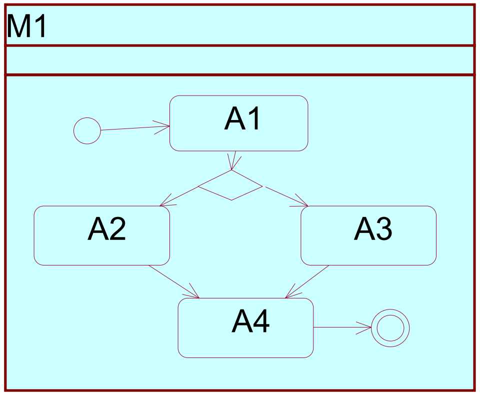
I. Specification based (functional) testing

- 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



II. 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)



I. Specification based (functional) testing

Goals:

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

for testing the functionality.

Overview of techniques:

1. Equivalence partitioning
2. Boundary value analysis
3. Cause-effect analysis
4. Combinatorial techniques

1. Equivalence partitioning

Input and output equivalence classes:

Data that are expected to *cover the same faults* (cover the same part of the program)

Goal: Each equivalence class is represented by a test input (selected test data); the correctness in case of the remaining inputs follows from the principle of induction

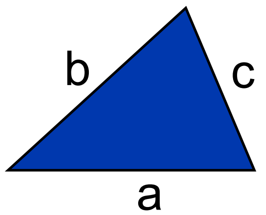
Test data selection is a heuristic procedure:

- Input data triggering the same service
- Valid and invalid input data

  –> valid and invalid equivalence classes
- Invalid data: Robustness testing

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
- How many tests are selected?
  - Beck: 6 tests, Binder: 65 tests, Jorgensen: 185 tests …

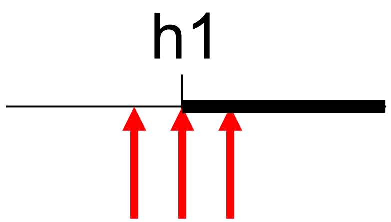


Valid/invalid equivalence classes

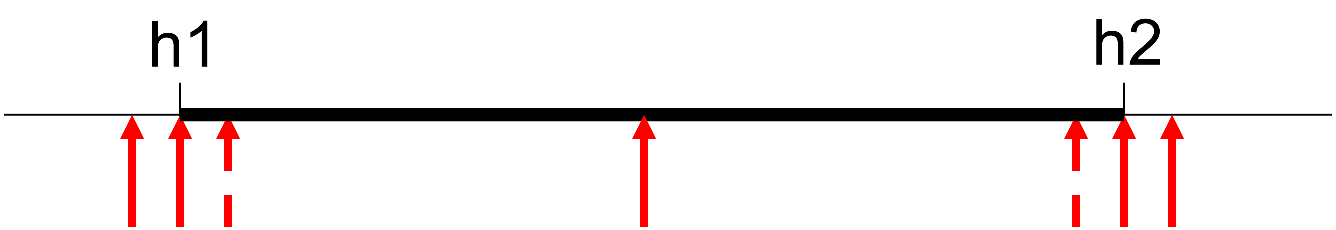
- Tests in case of several inputs:
  - Valid (normal) equivalence classes:
    test data should cover as much equivalence classes as possible
  - Invalid equivalence classes:
    first covering the each invalid equivalence class separately,
    then combining them systematically

2. Boundary value analysis

- Examining the boundaries of data partitions
  - Focusing on the boundaries of equivalence classes
  - Input and output partitions are also examined
  - Typical faults to be detected: Faulty relational operators, conditions in cycles, size of data structures, …
- Typical test data:
  - A boundary requires 3 tests:



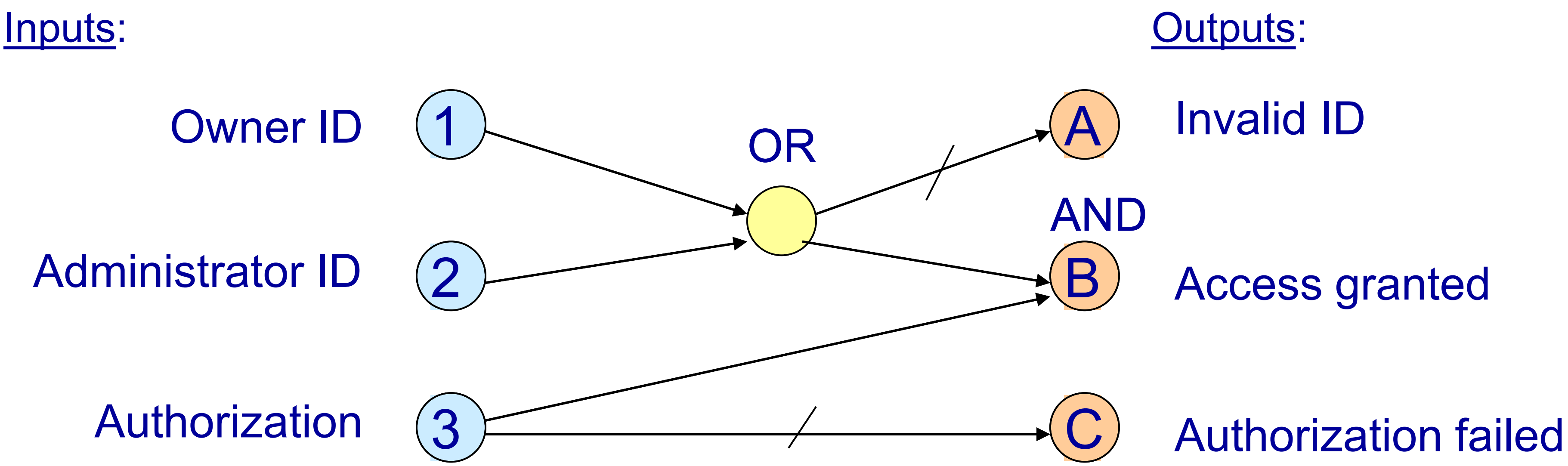
  - A partition requires 5-7 tests:



3. Cause-effect analysis

- Examining the relation of inputs and outputs (if it is simple, e.g., combinational)
  - Causes: input equivalence classes
  - Effects: output equivalence classes
- Boole-graph: relations of causes and effects
  - AND, OR relations
  - Invalid combinations
- Decision table: Covering the Boole-graph
  - Truth table based representation
  - Columns represent test data

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

- Several input parameters
  - Failures are caused by (specific) combinations
  - Testing all combinations requires too much test cases
  - Rare combinations may also cause failures
- Basic idea: N-wise testing
  - For each n parameters, testing all possible combinations of their potential values
  - Special case (n = 2): pairwise testing

Example: pair-wise testing

- Given input parameters and potential values:
  - OS: eCos, μc/OS
  - CPU: AVR Mega, ARM7
  - Protocol: IPv4, IPv6
- How many combinations are possible?
- How many test cases are needed for pairwise testing?

A potential test suite:

T1: eCos, AVR Mega, IPv4

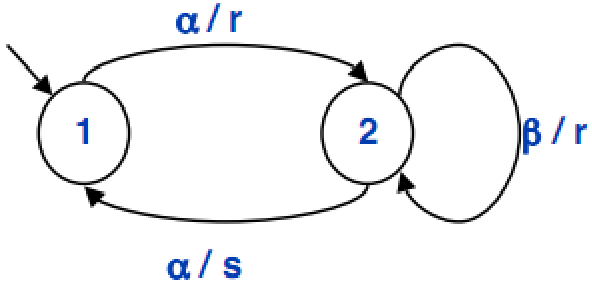
T2: eCos, ARM7, IPv6

T3: μc/OS, AVR Mega, IPv6

T4: μc/OS, ARM7, IPv4

Additional techniques

- Finite automaton based testing
  - The specification is given as a finite automaton
  - Typical test goals: to cover each state, each transition, invalid transitions, …

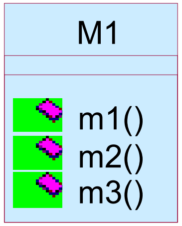


- Use case based testing
  - The specification is given as a set of use cases
  - Each use case shall be covered by the test suite
- Random testing
  - Easy to generate (but evaluation may be more difficult)
  - Low efficiency

Test approaches

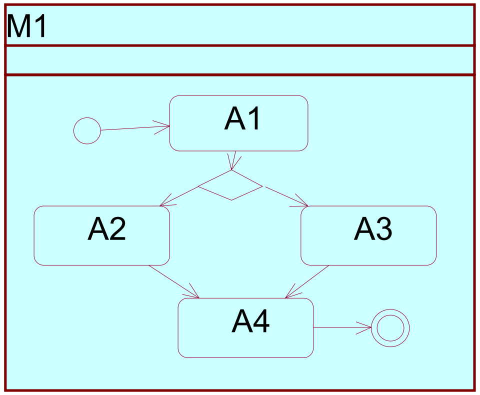
I. Specification based (functional) testing

- 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



II. 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)



II. Structure based testing

- Internal structure is known:
  - It has to be covered by the test suite
- Goals:

  There shall not remain such

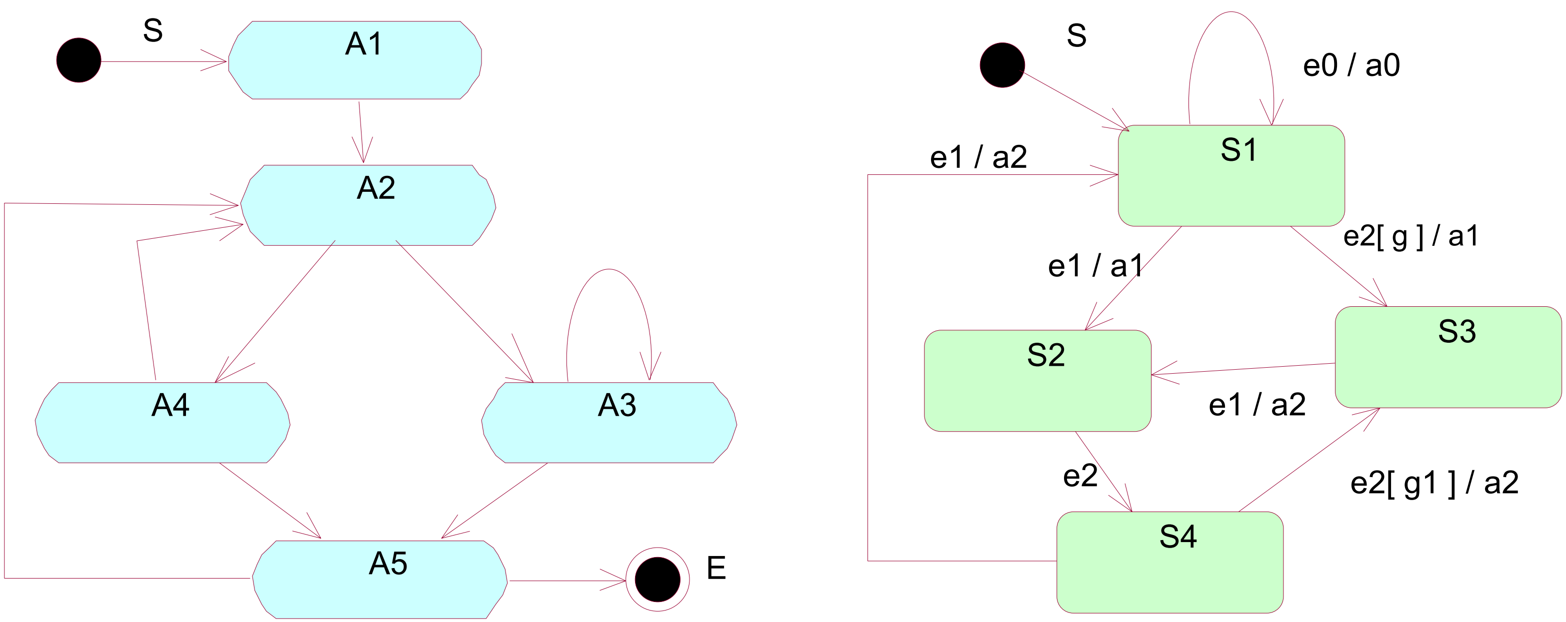
  - statement,
  - decision,
  - execution path

  in the program,

  which was not executed during testing

The internal structure

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



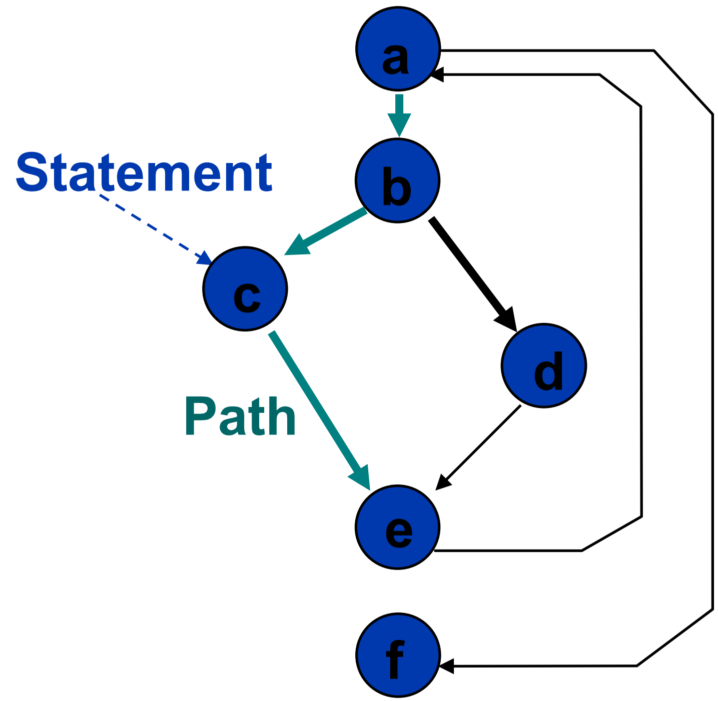
The internal structure

- 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:



Conditions and decisions

- Condition: a logical indivisible (atomic) expression
- Decision: a Boolean expression composed of conditions and zero or more Boolean operators

- Examples:
  - A decision with one condition:

    ```
    if (temp > 20) {…}
    ```

  - A decision with several conditions:

```
if (temp > 20 && (valveIsOpen || p == HIGH)) {…}
```

Test coverage metrics

Characterizing the quality of the test suite:

Which part of the 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 coverage (DO-178B, EN 50128,...)

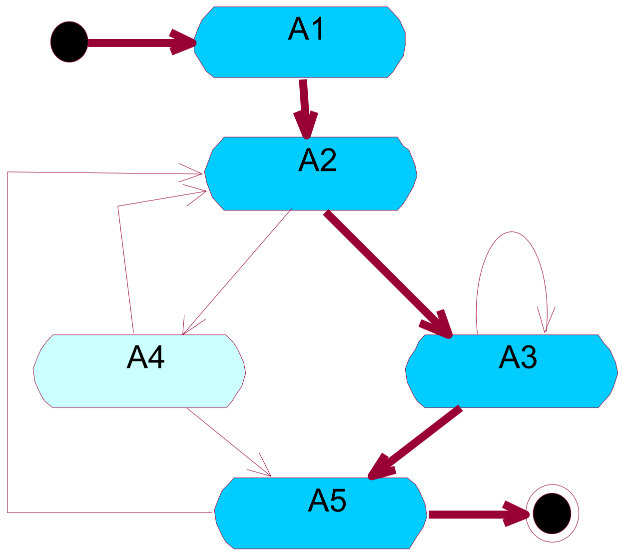
- 100% statements coverage is a basic requirement

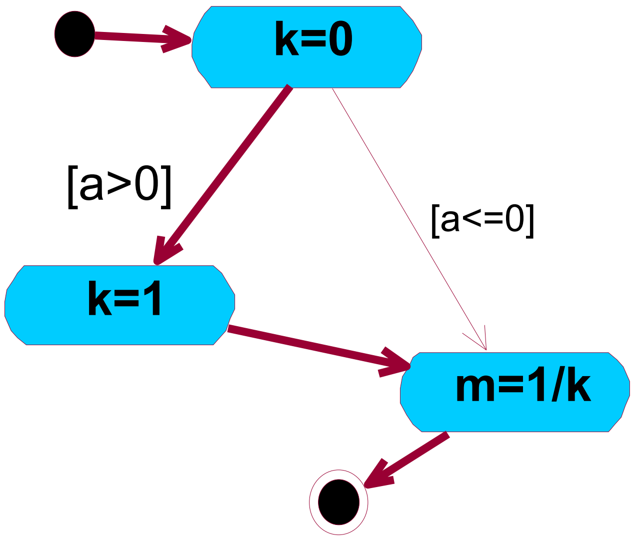
1. Statement coverage

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%

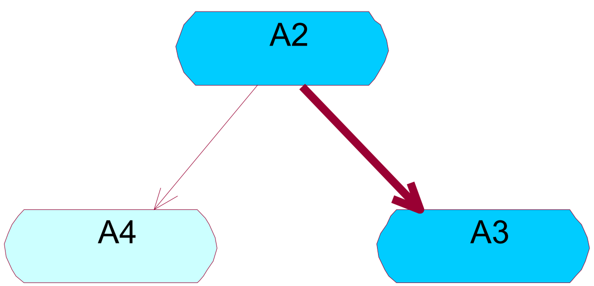
Statement coverage: 100%

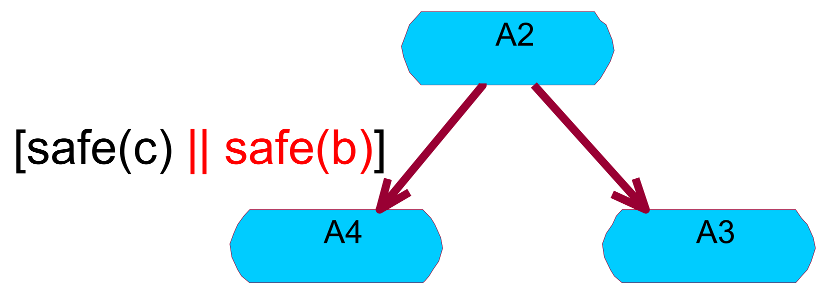
2. Decision coverage

Definition:

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

Does not take into account all combinations of conditions!





Decision coverage: 50%

Decision coverage: 100%

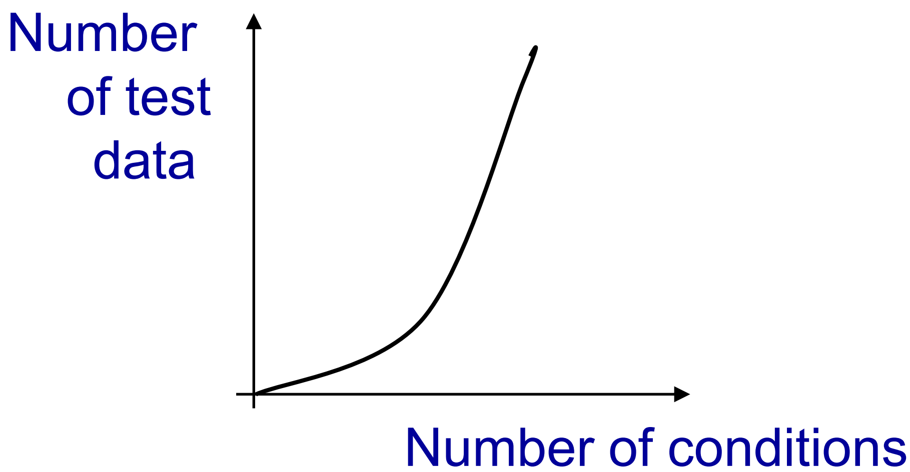
3. Multiple condition coverage

Definition:

$$\frac{\text{Number of condition combinations tried during testing}}{\text{Number of all condition combinations}}$$

Strong, but complex:

For n conditions 2^n test cases may be necessary!



In avionics systems there are programs with more than 30 conditions!

Other coverage criteria

MC/DC: Modified Condition/Decision Coverage

- It is used in the standard DO-178B to ensure that Level A (Catastrophic) software is tested adequately
- During testing followings must be true:
  - Each entry and exit point has been invoked at least once,
  - every condition in a decision in the program has taken all possible outcomes at least once,
  - every decision in the program has taken all possible outcomes at least once,
  - each condition in a decision is shown to independently affect the outcome of the decision.

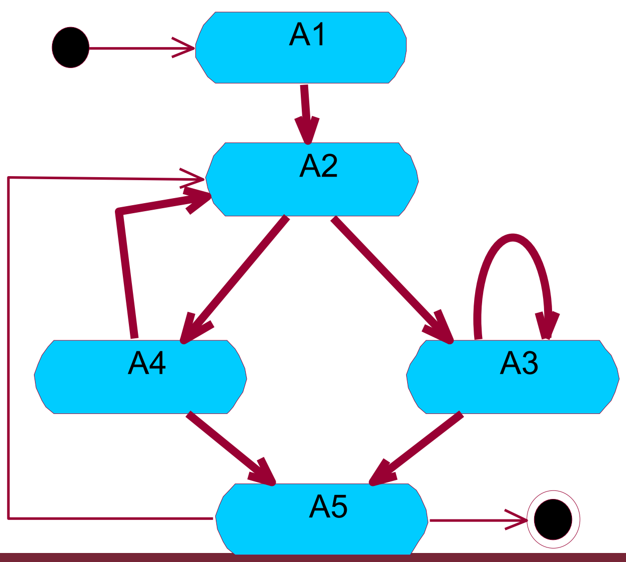
4. Path coverage

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%

Summary of coverage criteria

Table 1. Types of Structural Coverage

Coverage Criteria	Statement Coverage	Decision Coverage	Condition Coverage	Condition/ Decision Coverage	MC/DC	Multiple Condition Coverage
Every point of entry and exit in the program has been invoked at least once		•	•	•	•	•
Every statement in the program has been invoked at least once	•					
Every decision in the program has taken all possible outcomes at least once		•		•	•	•
Every condition in a decision in the program has taken all possible outcomes at least once			•	•	•	•
Every condition in a decision has been shown to independently affect that decision's outcome					•	•[8]
Every combination of condition outcomes within a decision has been invoked at least once						•

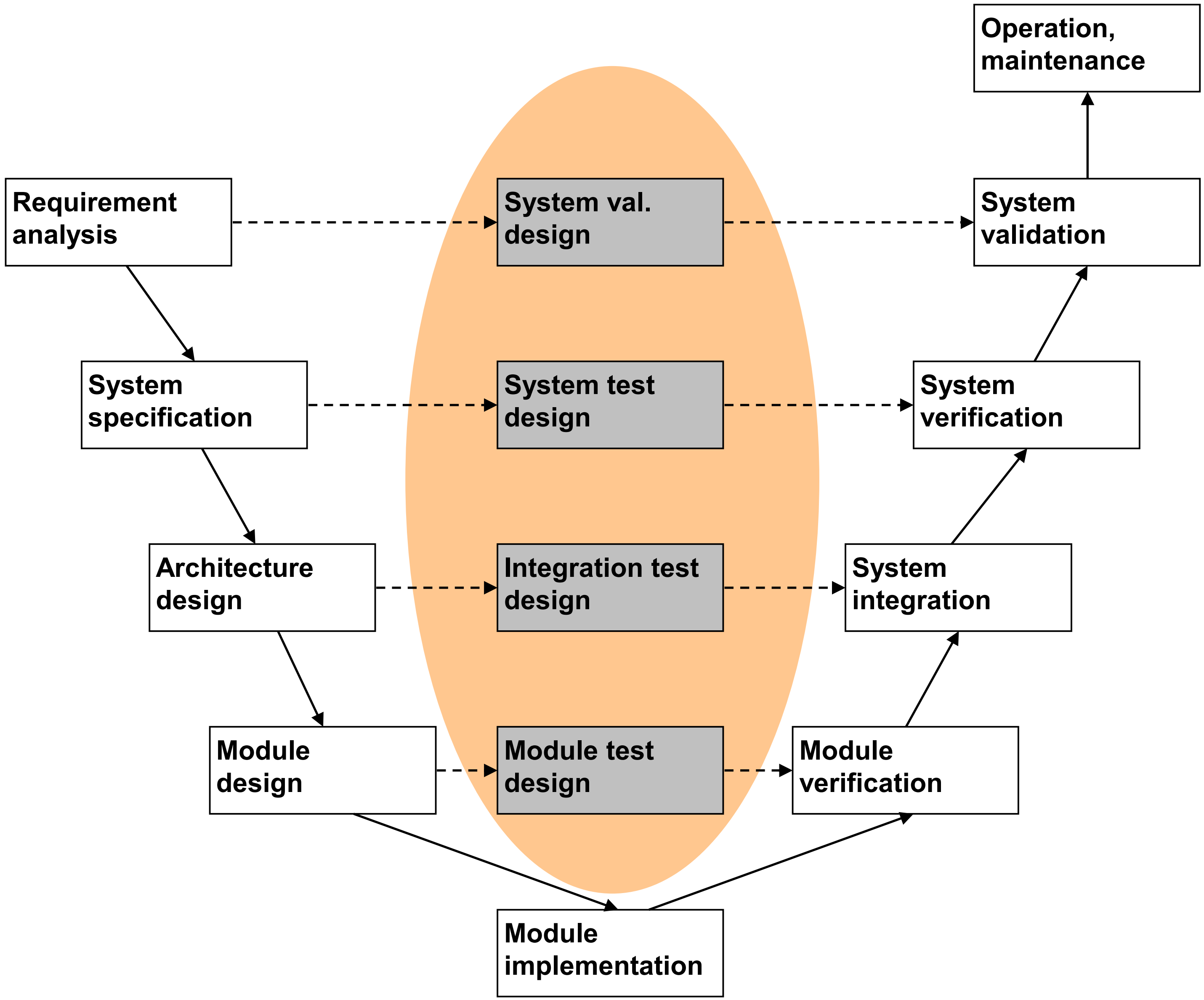
From: K. J. Hayhurst et al. A Practical Tutorial on Modified Condition/ Decision Coverage, NASA/TM-2001-210876

Testing process

What are the typical phases of testing?

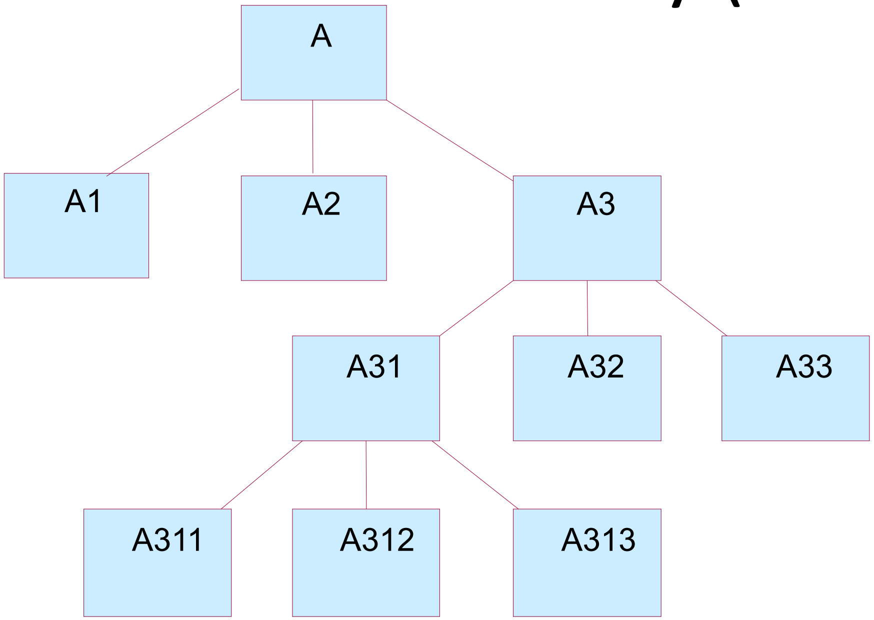
How to test complex systems?

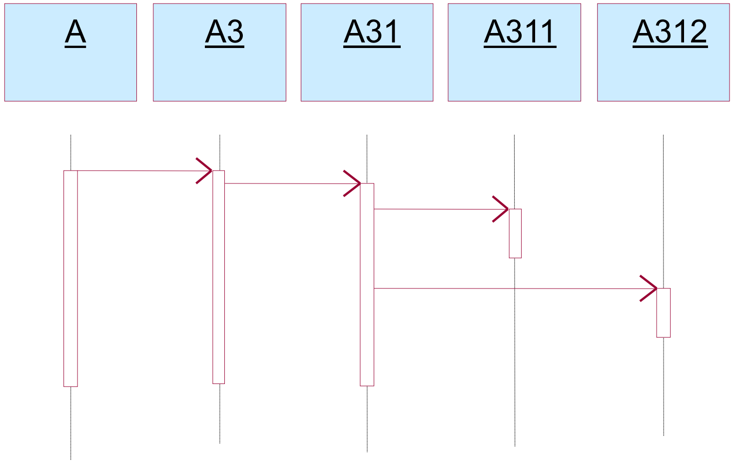
Testing and test design in the V-model



1. Module testing

- Modules:
  - Logically separated units
  - Well-defined interfaces
  - OO paradigm: Classes (packages, components)
- Module call hierarchy (in ideal case):



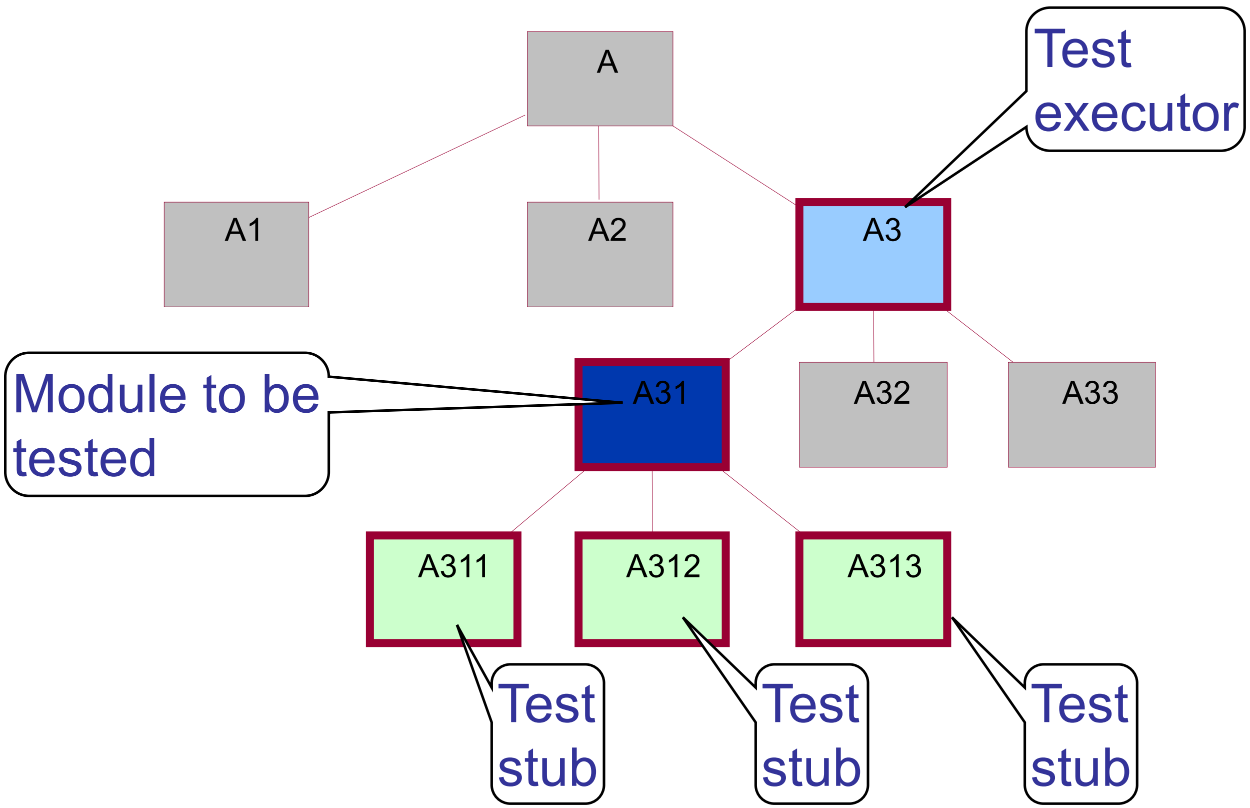


Module testing

- Lowest level testing
  - Integration phase is more efficient if the modules are already tested
- Modules can be tested separately
  - Handling complexity
  - Debugging is easier
  - Testing can be parallel for the modules
- Complementary techniques
  - Specification based and structure based testing

Isolated testing of modules

- Modules are tested separately, in isolation
- Test executor and test stubs are required
- Integration is not supported



Regression testing

Repeated execution of test cases:

- In case when the module is changed
  - Iterative software development,
  - Modified specification,
  - Corrections, ...
- In case when the environment changes
  - Changing of the caller/called modules,
  - Changing of platform services, ...

Goals:

- Repeatable, automated test execution
- Identification of functions to be re-tested

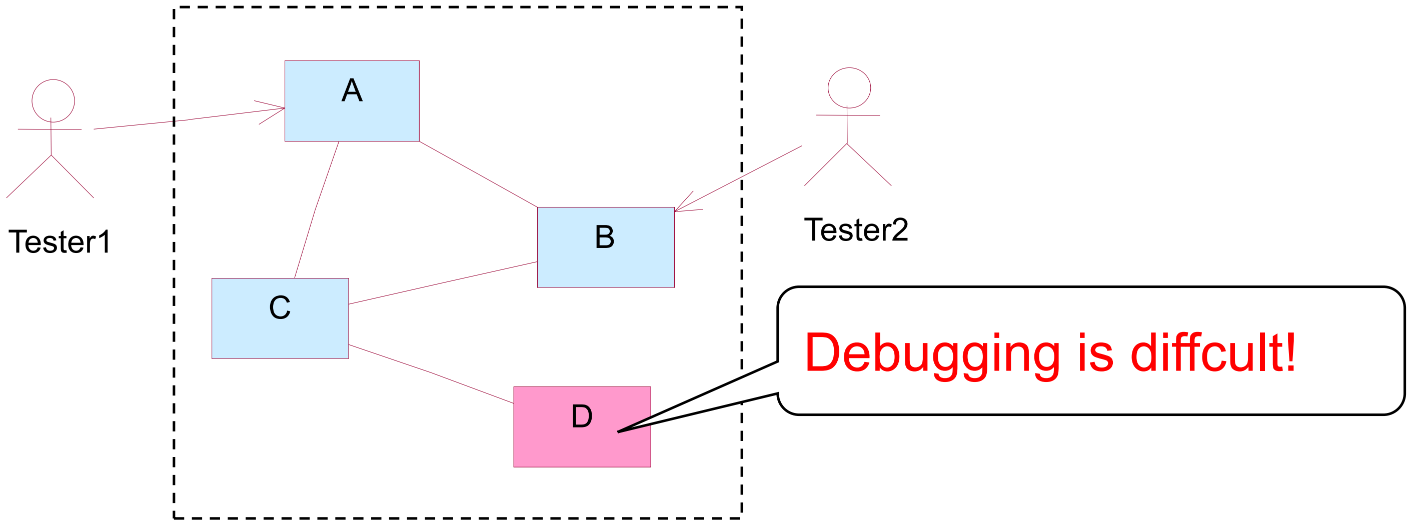
2. Integration testing

Testing the interactions of modules

- Motivation
  - The system-level interaction of modules may be incorrect despite the fact that all modules are correct
- Methods:
  - Functional testing: Testing scenarios
    - Sometimes the scenarios are part of the specification
  - (Structure based testing at module level)
- Approaches:
  - “Big bang” testing: integration of all modules
  - Incremental testing: stepwise integration of modules

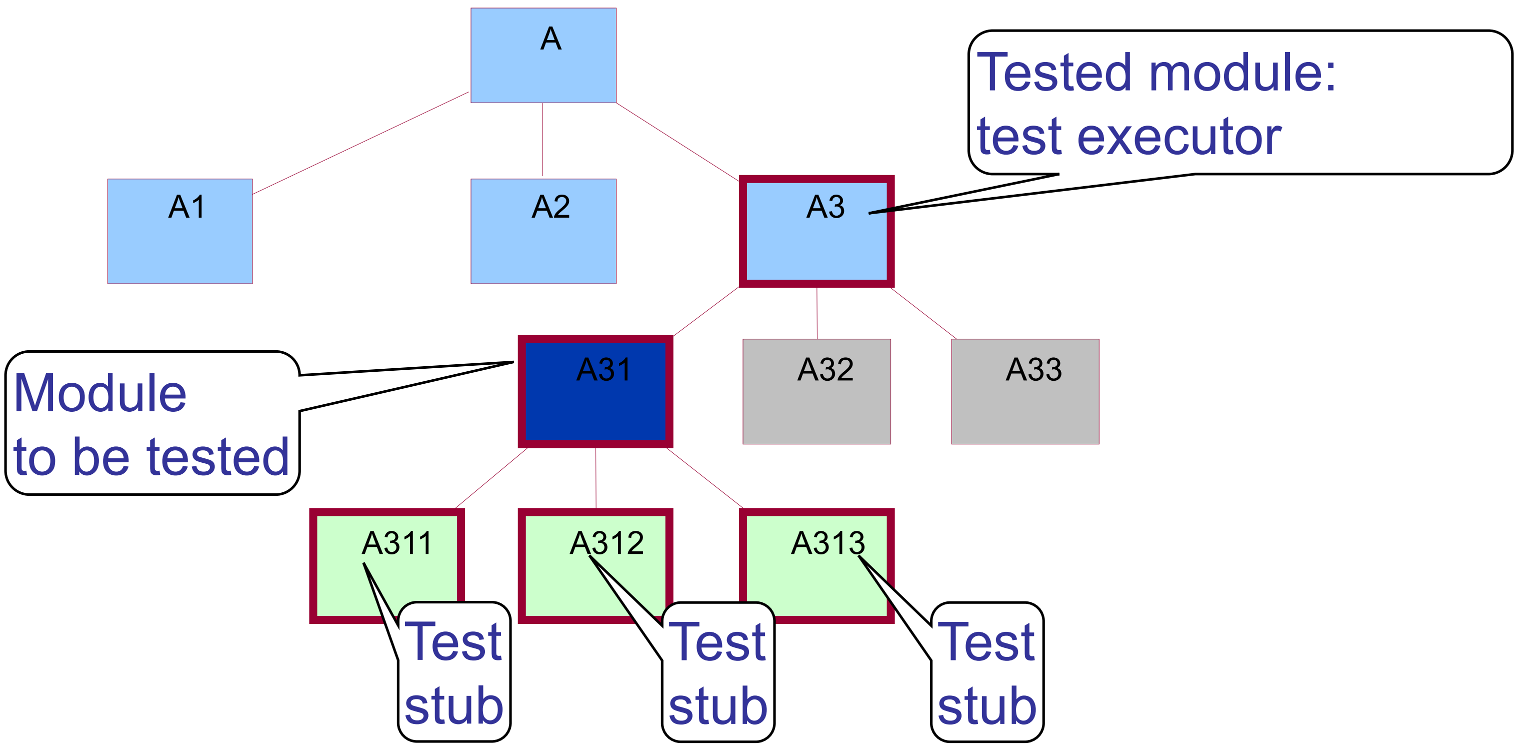
“Big bang” testing

- Integration of all modules and testing using the external interfaces of the integrated system
- External test executor
- Based of the functional specification of the system
- To be applied only in case of small systems



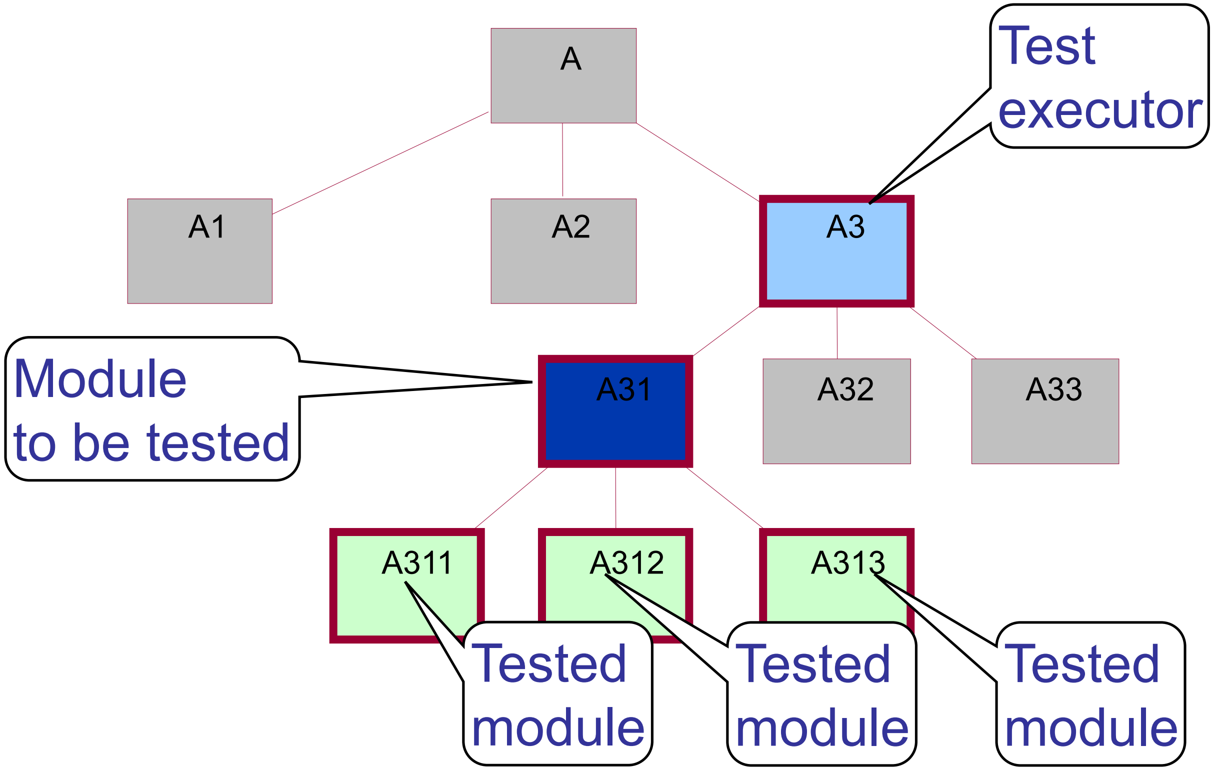
Top-down integration testing

- Modules are tested from the caller modules
- Stubs replace the lower-level modules that are called
- Requirement-oriented testing
- Module modification: modifies the testing of lower levels



Bottom-up integration testing

- Modules use already tested modules
- Test executor is needed
- Testing is performed in parallel with integration
- Module modification: modifies the testing of upper levels



Integration with the runtime environment

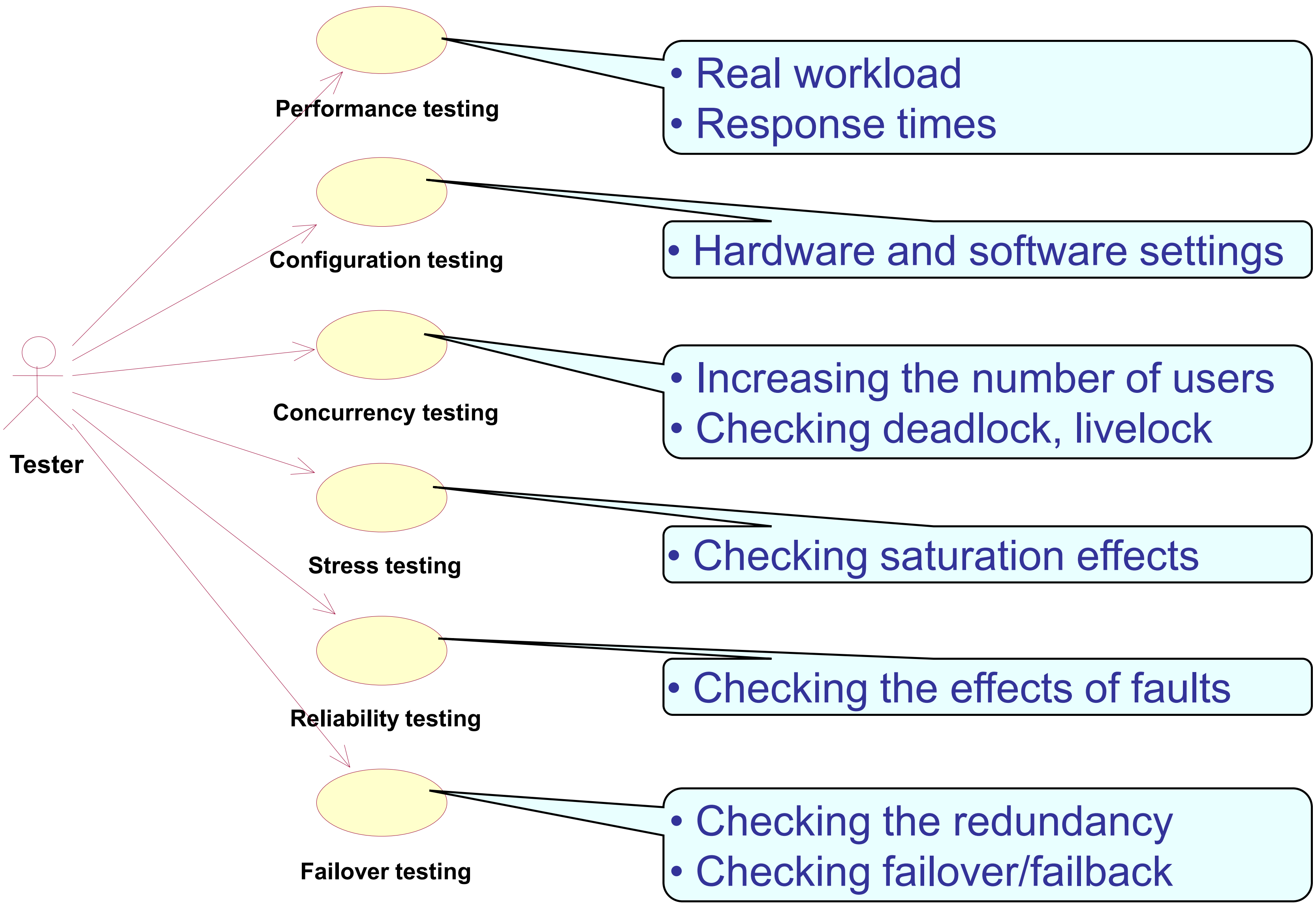
- Motivation: It is hard to construct stubs for the runtime environment
  - Platform services, RT-OS, task scheduler, …
- Strategy:
  1. Top-down integration of the application modules to the level of the runtime environment
  2. Bottom-up testing of the runtime environment
     - Isolation testing of functions (if necessary)
     - „Big bang” testing
       with the lowest level of the application module hierarchy
  3. Integration of the application with the runtime environment, finishing top-down integration

3. System testing

Testing on the basis of the system level specification

- Characteristics:
  - Performed after hardware-software integration
  - Testing functional specification + testing extra-functional properties as well
- Testing aspects:
  - Data integrity
  - User profile (workload)
  - Checking application conditions of the system (resource usage, saturation)
  - Testing fault handling

Types of system tests



4. Validation testing

- Goal: Testing in real environment
  - User requirements are taken into account
  - Non-specified expectations come to light
  - Reaction to unexpected inputs/conditions is checked
  - Events of low probability may appear
- Timing aspects
  - Constraints and conditions of the real environment
  - Real-time testing and monitoring is needed
- Environment simulation
  - If given situations cannot be tested in a real environment (e.g., protection systems)
  - Simulators shall be validated somehow

Relation to the development process

1. Module testing
   - Isolation testing
2. Integration testing
   - „Big bang” testing
   - Top-down testing
   - Bottom-up testing
   - Integration with runtime environment
3. System testing
   - Software-hardware integration testing
4. Validation testing
   - Testing user requirements
   - Environment simulation

Summary

- Testing techniques
  - Specification based (functional, black-box) testing
    - Equivalence partitioning
    - Boundary value analysis
    - Cause-effect analysis
  - Structure based (white-box) testing
    - Coverage metrics and criteria
- Testing process
  - Module testing
  - Integration testing
    - Top-down integration testing
    - Bottom-up integration testing
  - System testing
  - Validation testing