

Software Verification and Validation

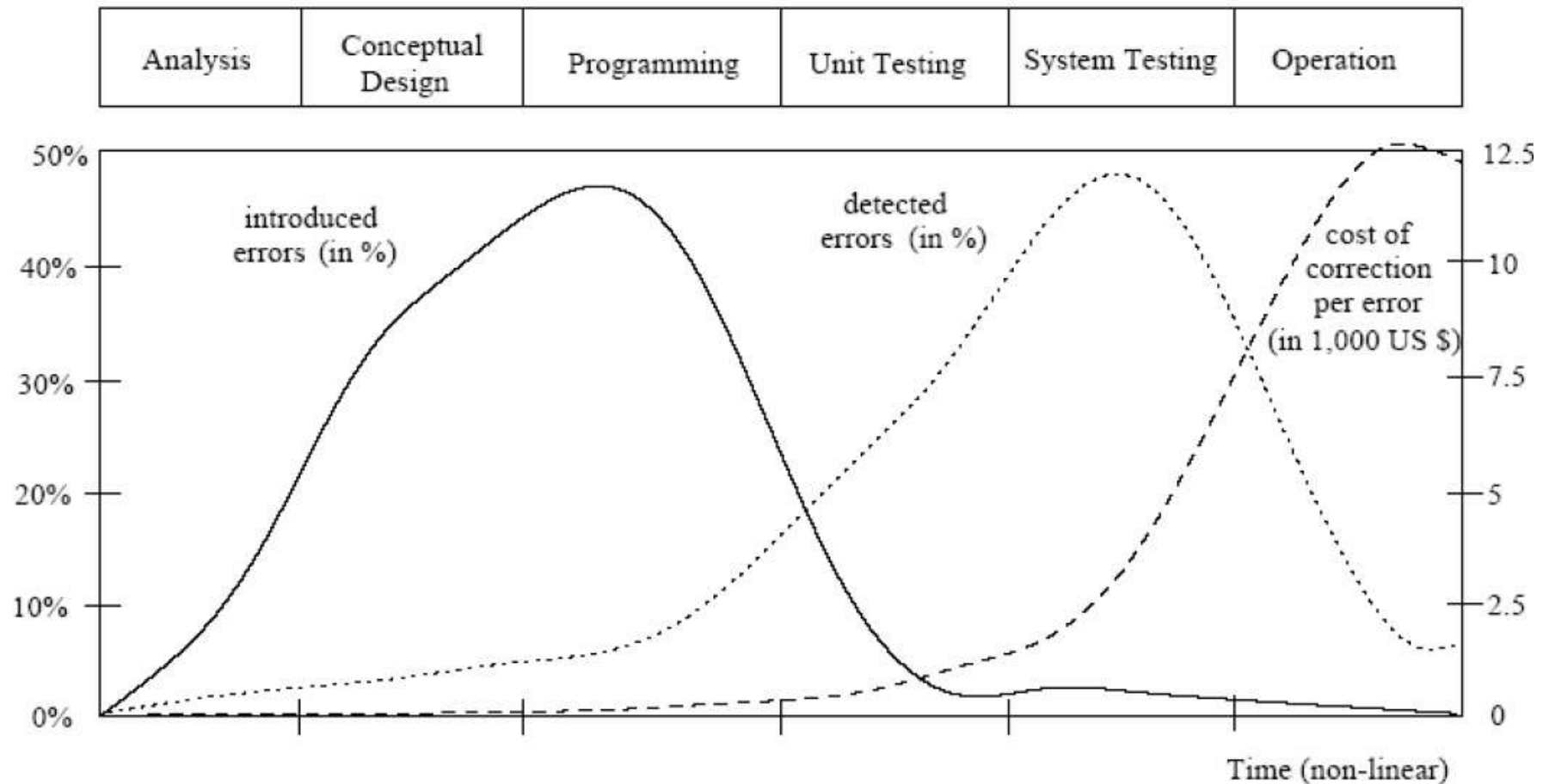
VIMIMA11 Design and integration of embedded systems

Balázs Scherer

Software failures

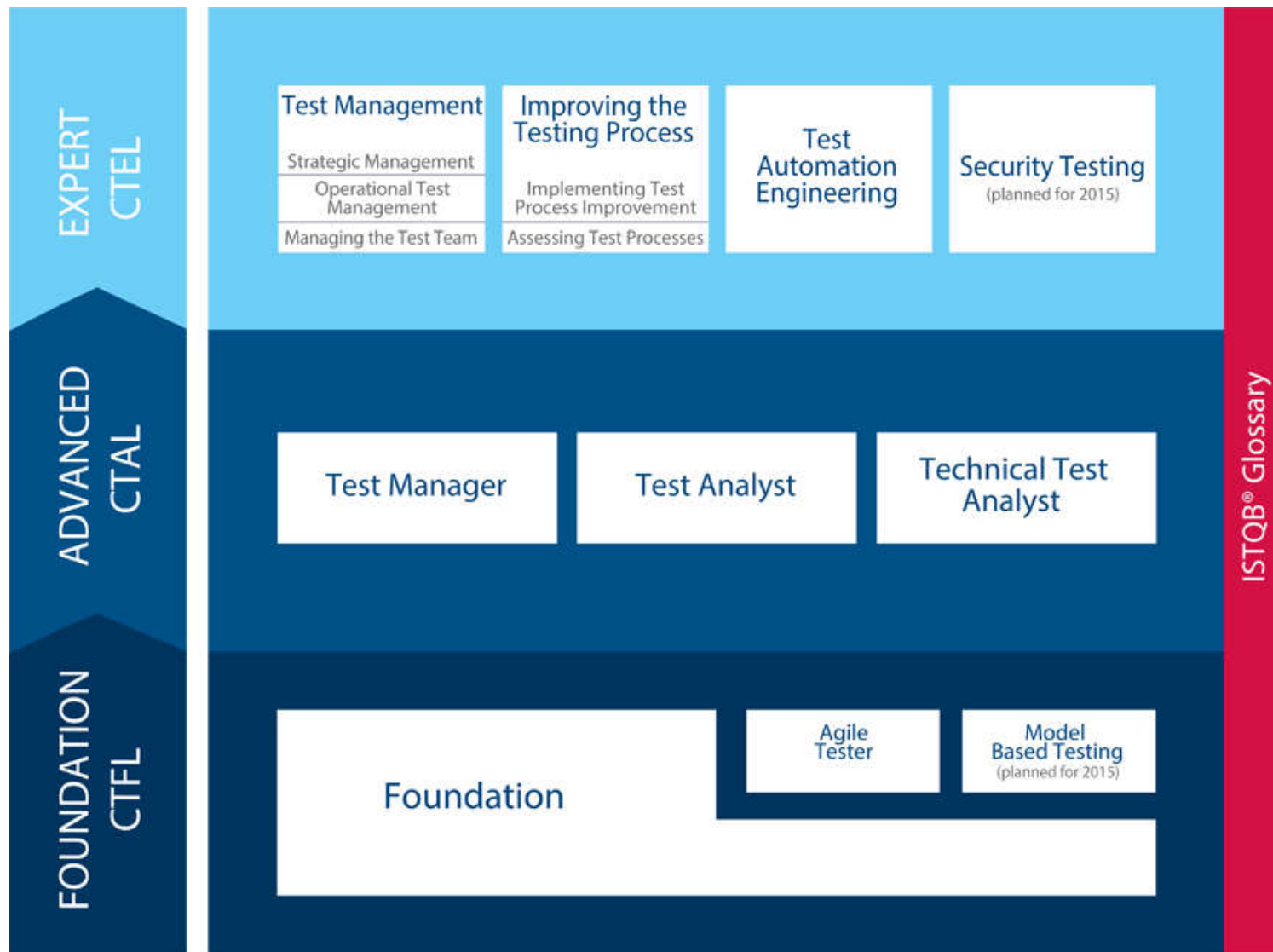
- **Error:** Human operation that leads to an undersigned behavior.
- **Fault:** The *Error's* occurrence in the code, many times called *bug*. Executing this code part leads to **Failure**
- **Failure:** Aanomalous behavior of the software

Importance of the verification



From: P. Liggesmeyer et al., Qualitätssicherung Software-basierter technischer Systeme, Informatik Spektrum, 21:249-258, 1998. Quoted after J.P. Katoen, Principles of Model Checking, 2004/5. Copyright © by the authors.

ISTQB: *International Software Testing Qualifications Board*



Fundamentals of testing

- **Testing shows presence of defects:** Testing can show that defects are present, but cannot prove that there are no defects. Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found it is not a proof of correctness.
- **Exhaustive testing is impossible:** Testing everything (all combinations of is impossible inputs and preconditions) is not feasible except for trivial cases. Instead of exhaustive testing, we use risks and priorities to focus testing efforts.
- **Early testing:** Testing activities should start as early as possible in the software or system development life cycle and should be focused on defined objectives.

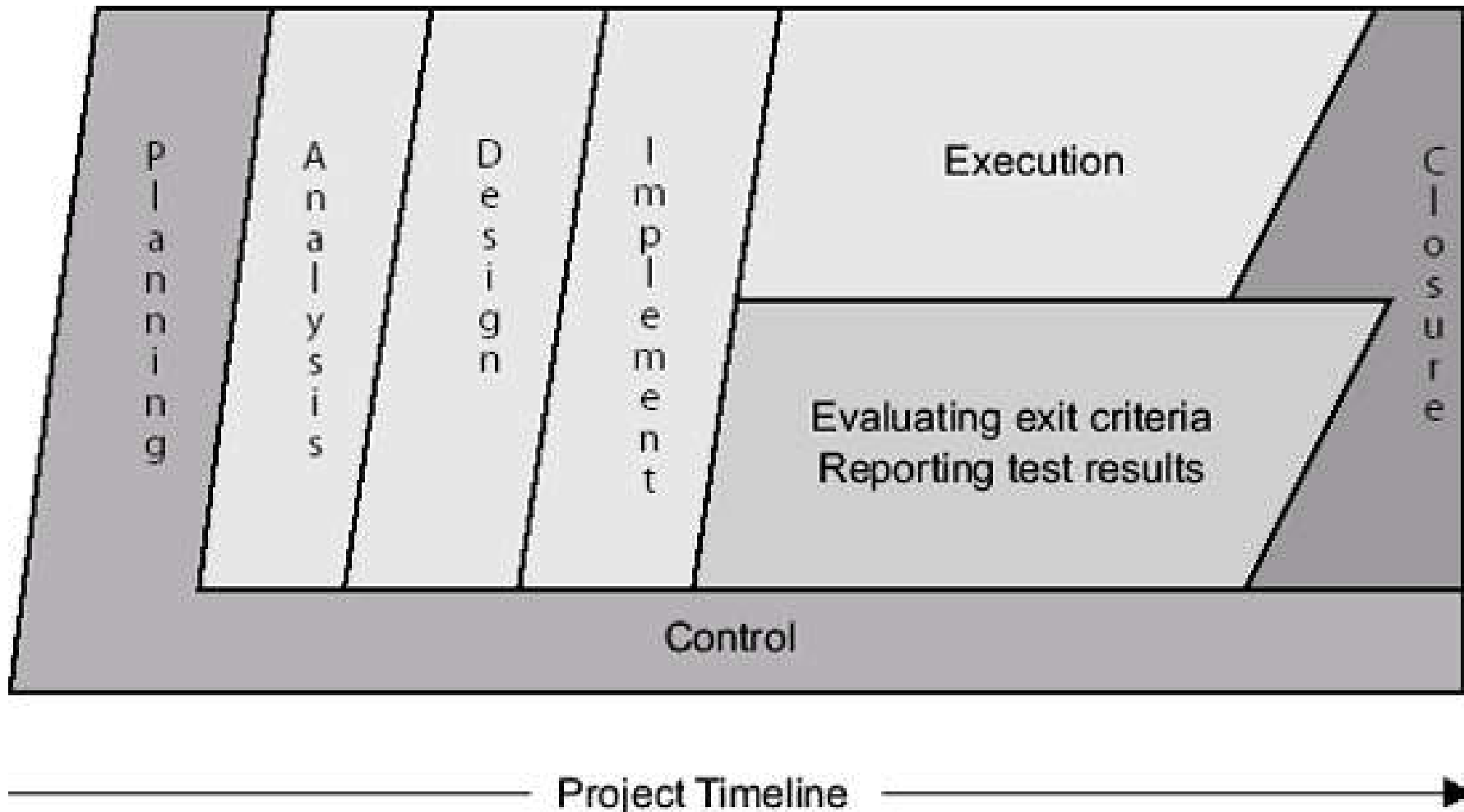
Fundamentals of testing

- **Defect clustering:** A small number of modules contain most of the defects discovered during prerelease testing or show the most operational failures.
- **Pesticide paradox:** If the same tests are repeated over and over again, eventually the same set of test cases will no longer find any new bugs. To overcome this 'pesticide paradox', the test cases need to be regularly reviewed and revised, and new and different tests need to be written to exercise different parts of the software or system to potentially find more defects.
- **Testing is context dependent:** Testing is done differently in different contexts. For example, safety-critical software is tested differently from an e-commerce site.

Fundamentals of testing

- **Absence-of-errors fallacy:** Finding and fixing defects does not help if the system built is unusable and does not fulfill the users' needs and expectations.

Testing life cycle



Test planning and control

- Determine the scope and risks and identify the objectives of testing
 - What software, components, systems or other products are in scope for testing
 - Are we testing primarily to uncover defects, to show that the software meets requirements, to demonstrate that the system is fit for purpose or to measure the qualities and attributes of the software?
- Determine the test approach
 - Test techniques to use
 - Coverage to achieve
- Implement the test policy and/or the test strategy
- Determine the required test resources

Test planning and control

- Schedule test analysis and design tasks, test implementation, execution and evaluation
- Determine the exit criteria: The testing do not last for exhaustion. There is a need for a well determined exit criteria, a coverage metric to achieve.
- Measure and analyze the results of reviews and testing
- Monitor and document progress, test coverage and exit criteria
- Provide information on testing
- Initiate corrective actions

Test analysis and design

- Review the test basis
 - product risk analysis, requirements, architecture, design specifications, and interfaces
- Identify test conditions based on analysis of test items, their specifications and behavior
 - high-level list of what we are interested in testing
- Design the tests
 - Selecting test methods based on test conditions
- Evaluate testability of the requirements and system
 - The requirements may be written in a way that allows a tester to design tests; for example, if the performance of the software is important, that should be specified in a testable way
- Design the test environment set-up and identify any required infrastructure and tools.

Test implementation and execution

- take the test conditions and make them into test cases
- Put together a high-level design for our tests
- Setup test environment
- Develop and prioritize **test cases**
- Create **test suites** from the **test cases** for efficient test execution: A test suite is a logical collection of test cases which naturally work together. Test suites often share data and a common high-level set of objectives.
- Implement and verify the environment
- Execute the test suites and individual test cases
- Log the outcome of test execution
- Compare actual results to expected results
- Analyze differences and repeat testing

Test implementation and execution

- take the test conditions and make them into test cases
- Put together a high-level design for our tests
- Setup test environment
- Develop and prioritize **test cases**
- Create **test suites** from the **test cases** for efficient test execution: A test suite is a logical collection of test cases which naturally work together. Test suites often share data and a common high-level set of objectives.
- Implement and verify the environment
- Execute the test suites and individual test cases
- Log the outcome of test execution
- Compare actual results to expected results
- Analyze differences and repeat testing

Evaluating exit criteria, reporting, and test closure

- Comparing test results to **exit criteria**
- Assess if more tests are needed or if the exit criteria specified should be changed
- Write a **Test summary report**

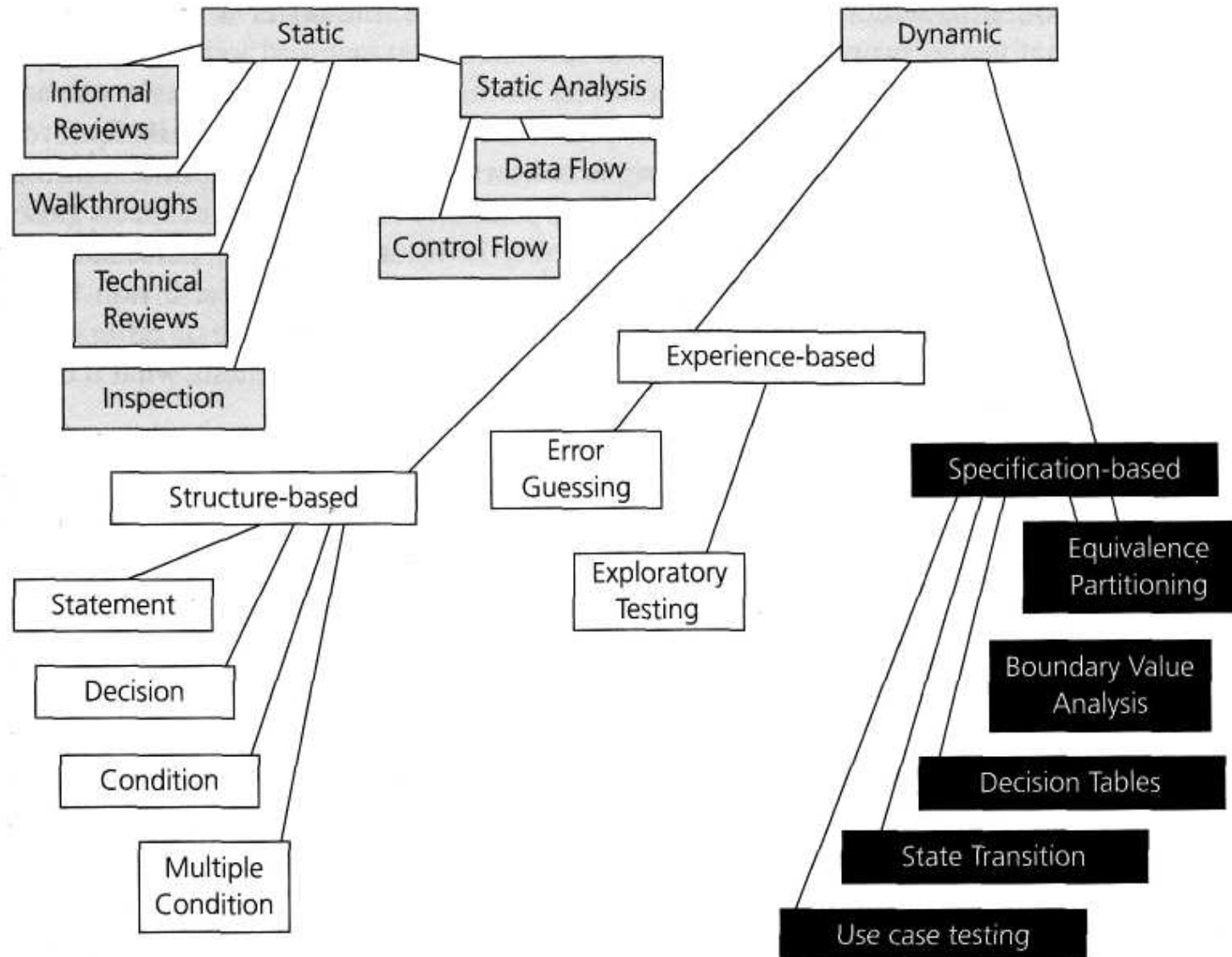
- Finalize and archive testware, such as scripts, the test environment, and any other test infrastructure, for later reuse.
- Evaluate how the testing went and analyze lessons learned for future releases and projects.

Who should perform the tests?

- Can the developer test its own software or system?
 - hard to find the problem in our work (do not understand well the specification)
 - Do not test everything (not expert at test methods, forget for example negative tests)
 - + No one knows the software or system better

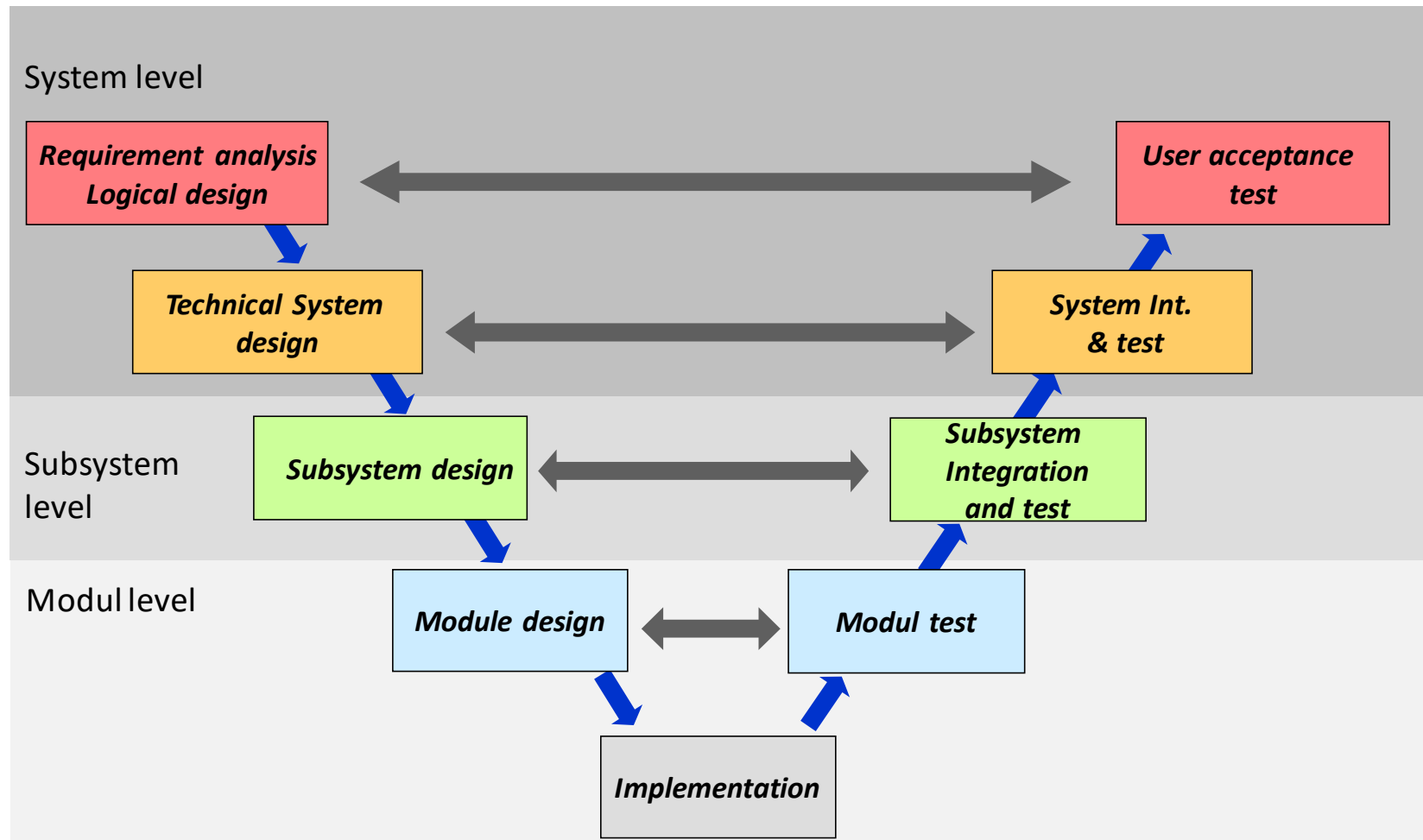
- Why testing independence is required
 - + independent: different point of view, can see problems invisible for the developer
 - + knows testing methods
 - + required by standards
 - Less knowhow about the system

Verification and test techniques



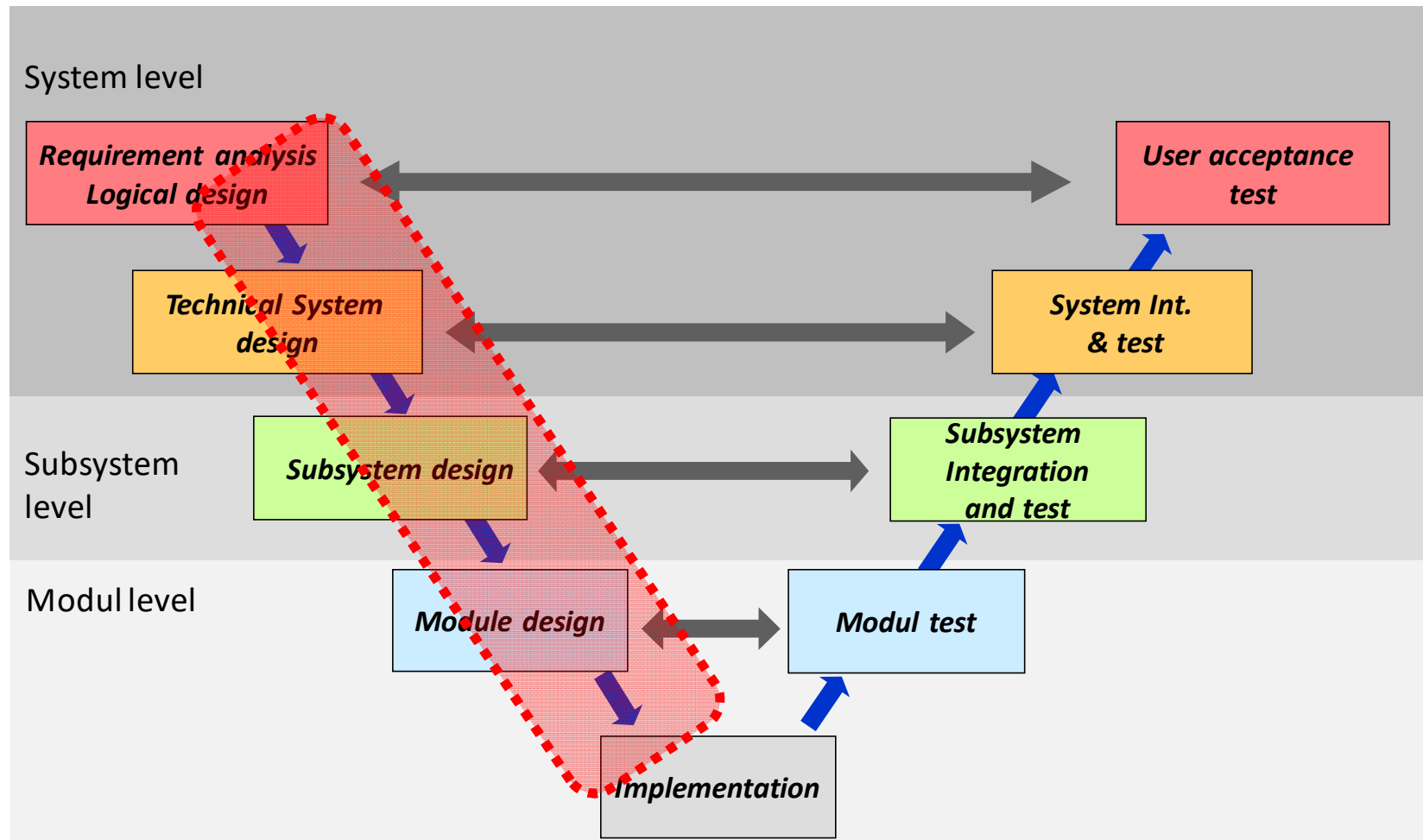
Static verification methods

- Do not requires an executable code
 - Can be used in early phases

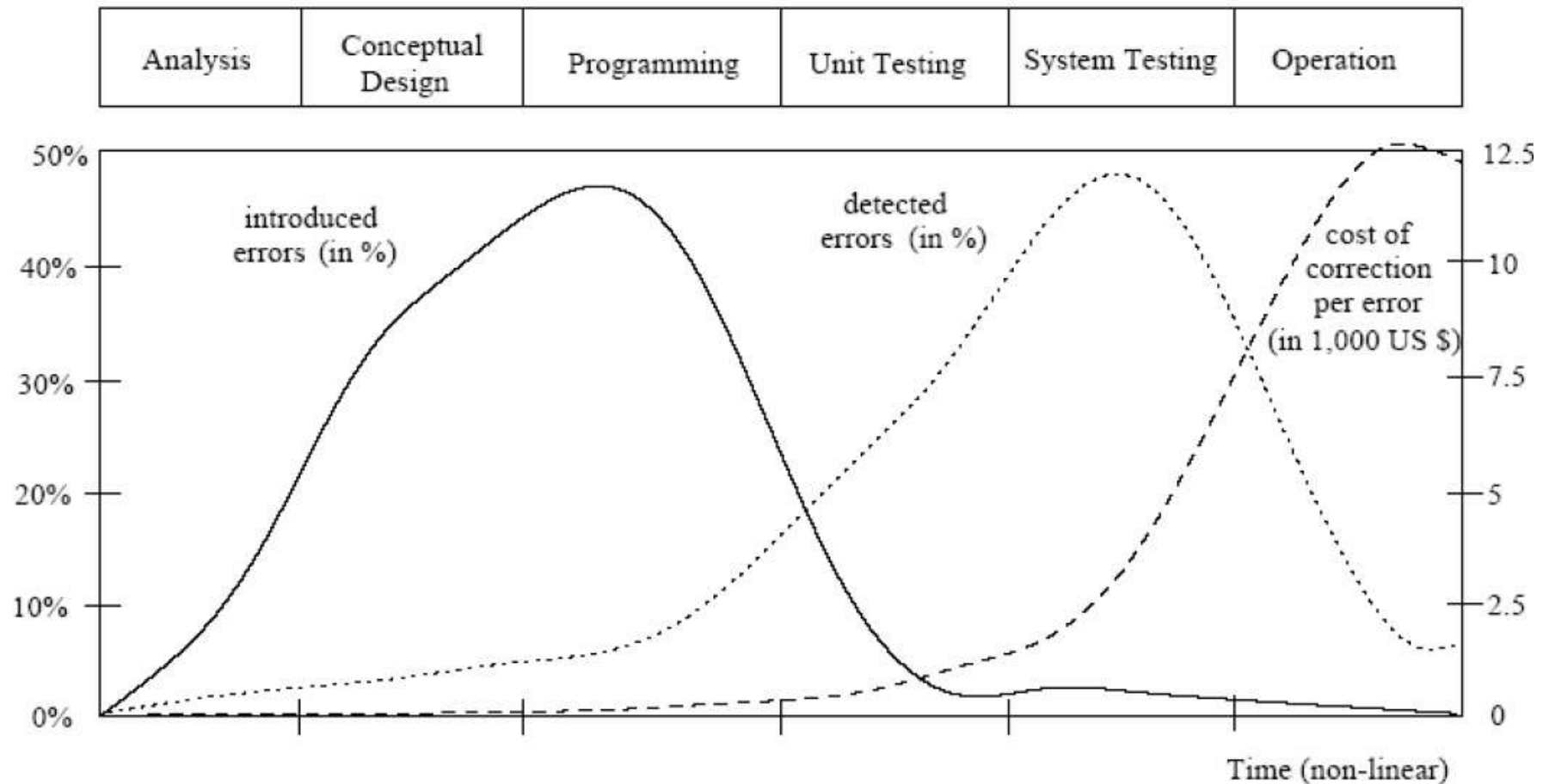


Static verification methods

- Do not requires an executable code
 - Can be used in early phases



Importance of the verification



From: P. Liggesmeyer et al., Qualitätssicherung Software-basierter technischer Systeme, Informatik Spektrum, 21:249-258, 1998. Quoted after J.P. Katoen, Principles of Model Checking, 2004/5. Copyright © by the authors.

Static verification: Reviews

- Informal review
 - No formal process
 - Pair, or technical lead review, code or design
 - Results may be documented
 - Usefulness depends on the reviewer
 - Inexpensive way to get some benefits
- Walkthrough
 - Meeting led by the author to the peer group
 - May take the forms of scenarios or dry runs
 - Open ended meeting, with optional preparations
 - May vary in practice from quite informal to very formal
 - Main purpose is learning, and gaining understanding, and finding defects

Static verification: Reviews

■ Technical review

- Documented, defined defect detection process
- Peers and technical experts are present
- Typically lead by a trained moderator
- There is pre-meeting preparations
- Optionally use checklists
- Review report is created
- Main purpose is discussing, making decisions, solving technical problems, checking conformance to specification, plans, and standards

■ Inspections

- Lead by trained moderator
- Detailed roles, formal process with checklists
- Specified entry and exit criteria
- Formal follow up process
- Purpose is finding defects

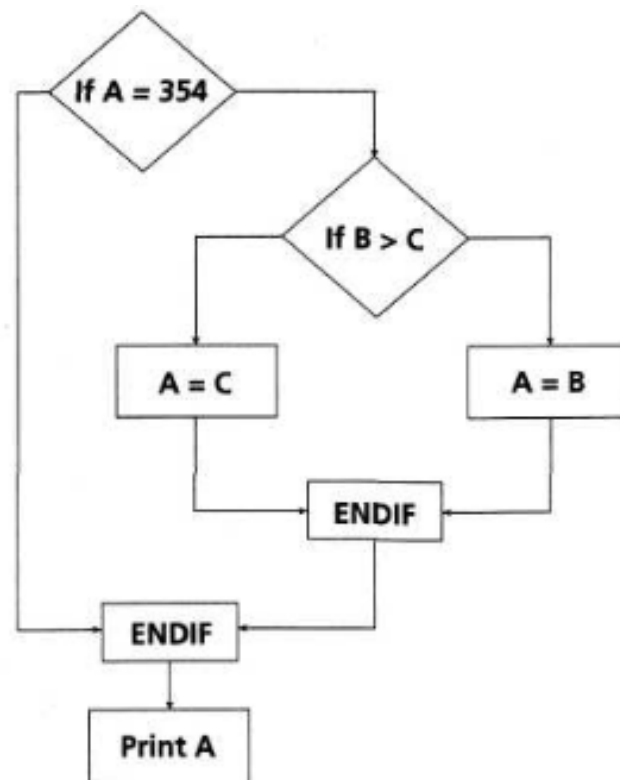
Static verification: static analyzers

- Checking against the coding rules
 - MISRA-C compatibility analysis
- Code metrics
 - The 20% of code case the 80% of the problems
 - Cyclomatic complexity analysis and calculations
 - Comment ratio
- Data flow verification
 - Are there usage of non initialized variables?
 - Are there variable under, or overflows, are the a dividing with 0?
- Control flow analysis
 - Are there unreachable code lines

Cyclomatic complexity


- Gives information about the complexity of the code. Useful to identify problematic software parts, and to estimate the required testing effort.

$M = E - N + 2$ *E*: Edges of the graph *N*: Nodes in the graphs



$$8 - 7 + 2 = 3$$

Static analysis tool example: PolySpace

- Software of MathWorks 
- Can be used for C/C++ and Ada languages
- Checking compliance with MISRA-C rules
- Uses abstract interpretation to discover Run-Time problems
 - Array over indexing
 - Errors in pointer usage
 - Using non initialized variables
 - Usage of bad arithmetical operations (divide with 0, square root from negative number)
 - Over or under flow of variables
 - Not allowed type conversions
 - Unreachable code, endless loops

PolySpace in operation

P
r
o
v
e
n

Green:
reliable

Red:
faulty

Gray:
dead

Orange:
unproven

```
static void Pointer_Arithmetic (void)
{
  int array[100];
  int i, *p = array;

  for(i = 0; i < 100; i++, p++)
    *p = 0;

  if(get_bus_status() > 0) {
    if (get_oil_pressure() > 0)
      *p = 5;   Static verification:
    else
      i++;
  }

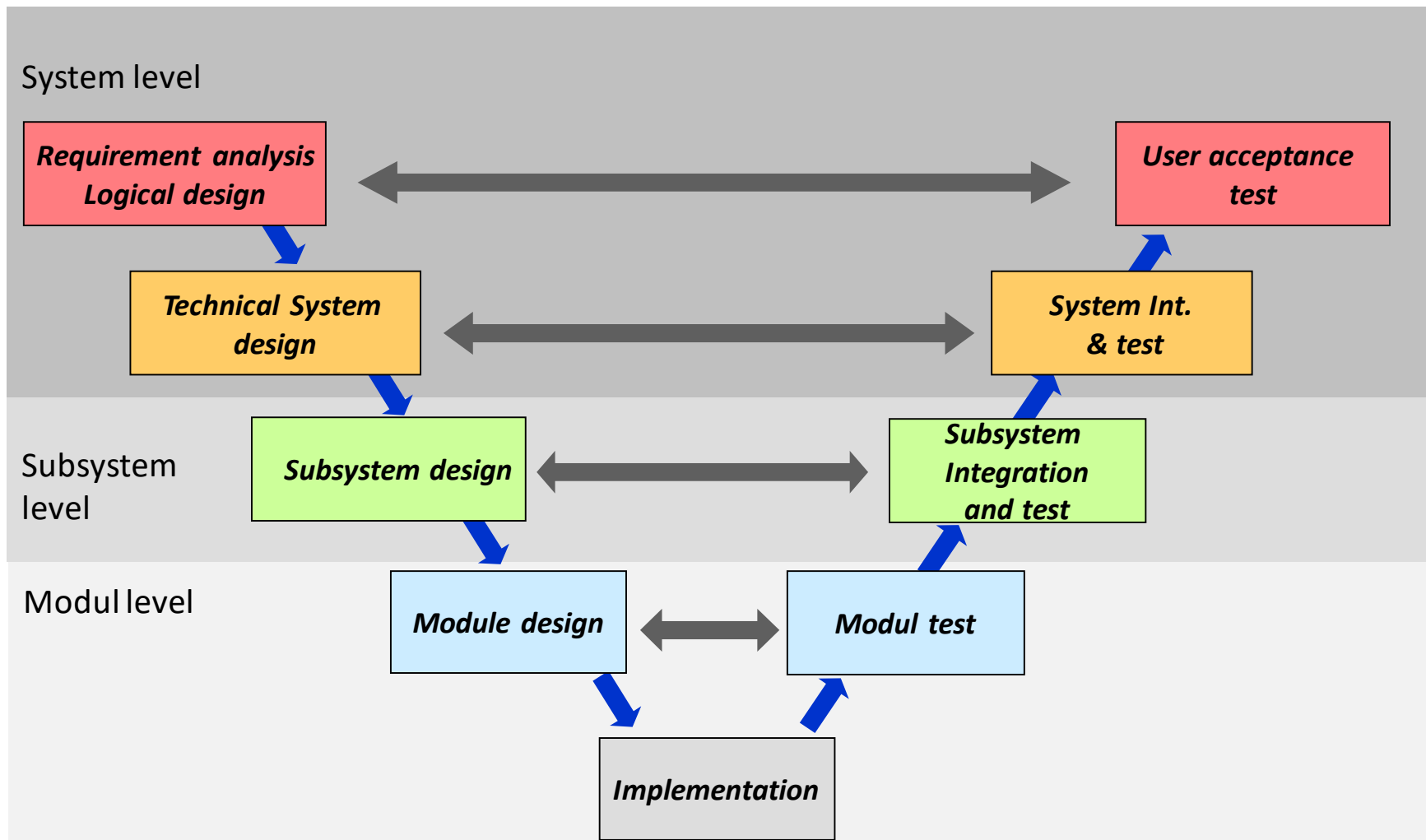
  i = get_bus_status();
  if (i >= 0) { *(p-i) = 10; }

  if ((0 < i) && (i <= 100)) {
    p = p - i;
    *p = 5;
  }
}
```

- **Green:**
reliable, good code
- **Red:**
faulty in every
circumstances
- **Grey:**
unreachable code
- **Orange:**
unproven to be
good or bad, need to
be checked by the
programmer

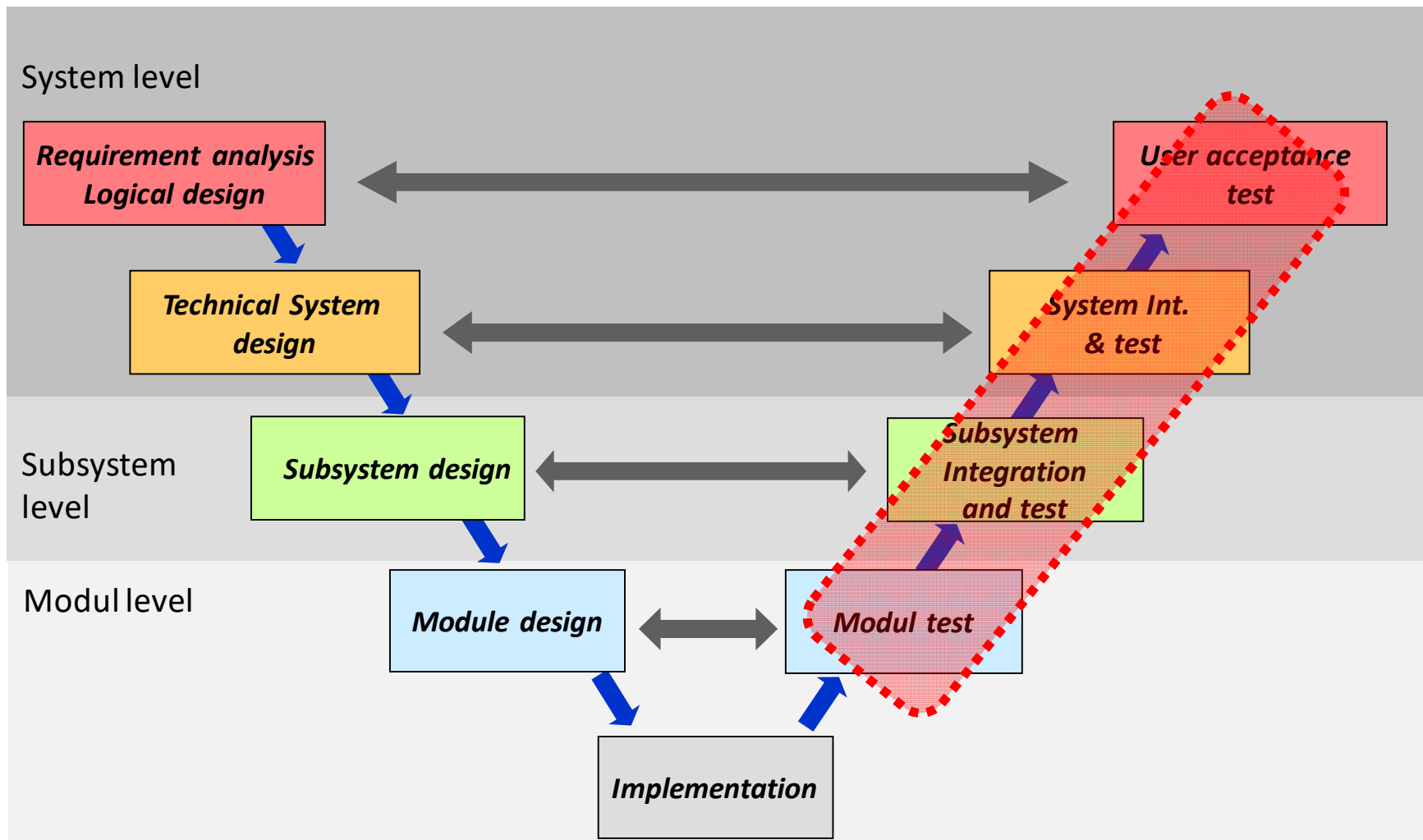
Dynamic verification techniques

- An executable code is needed for these methods: Testing

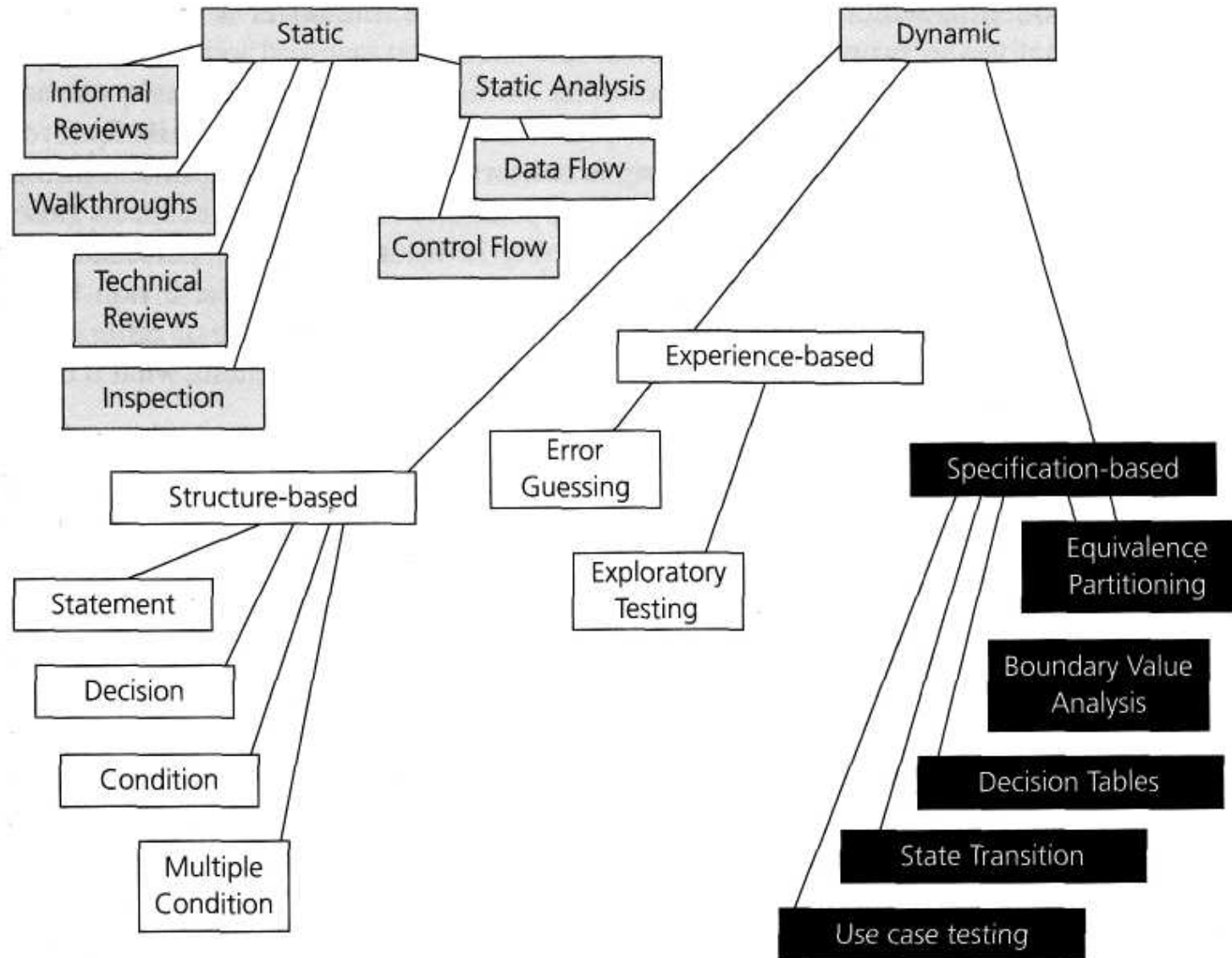


Dynamic verification techniques

- An executable code is needed for these methods: Testing



Dynamic methods



Test types

- **Functional tests**
 - Usually black box tests
 - Based on specifications
- **Non functional tests**
 - Maintainability, usability, reliability, efficiency
 - Performance, Load, Stress test
- **Structure based tests**
 - White box tests
 - Coverage based tests
- **Software change tests**
 - Confirmation test: test after bug fixing. Ensuring the bug fix is done well
 - Regression test: test after minor software change to ensure the change not affected the main functionality

White box, Grey box, and Black box tests

- **White box test**
 - The source code is known, and used during the test
 - Typically structure based test
 - Used mainly at the bottom part of V-model
- **Black box tests**
 - The internal behavior of the system is not known, the test is focus on the inputs and the outputs
 - Typically specification based test
 - Used during the whole verification & validation part of the V-model
- **Grey box tests**
 - Focus on the inputs and outputs
 - The internal structure, source code is not known, but there are information about the critical internal states of the system
 - Typically used in case embedded systems during integration

Specification based techniques

- The specification and requirements of the software or system to test is given. The goal is to create test data that is able to assure that the system or software meets its specification and requirements, in an efficient way.
- Techniques
 - equivalence partitioning
 - boundary value analysis;
 - decision tables;
 - state transition testing

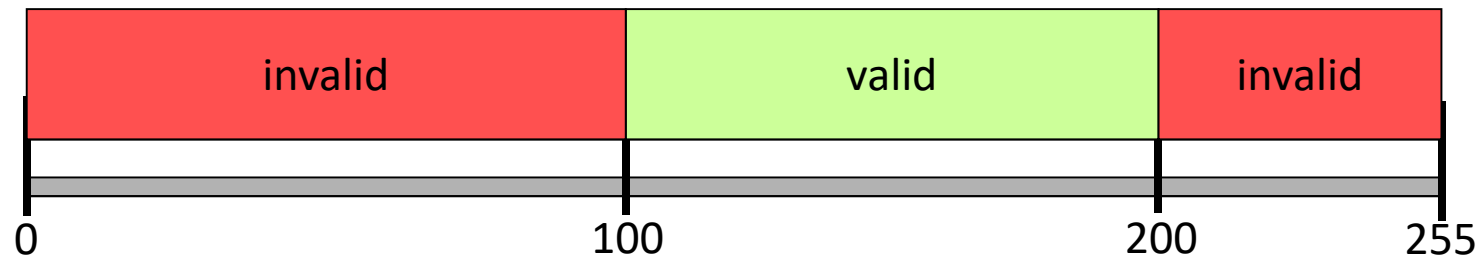
Equivalence partitioning

- Goal: Identify Equivalence Classes
 - divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same (i.e. the system should handle them equivalently), hence 'equivalence partitioning
 - we are assuming that all the conditions in one partition will be treated in the same way by the software
 - equivalence-partitioning technique then requires that we need test **only one condition from each partition**
 - Wrong identification of partitions can cause problems
- Typical partitions or classes
 - Value ranges
 - Value sets
 - Special clusters

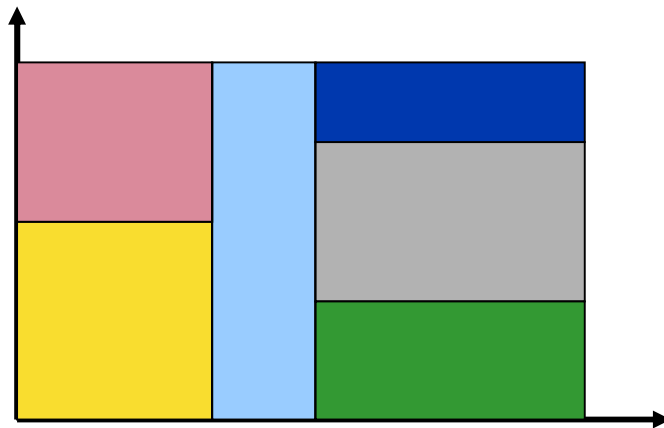
Equivalence partitioning examples

■ Examples

- Simple value ranges



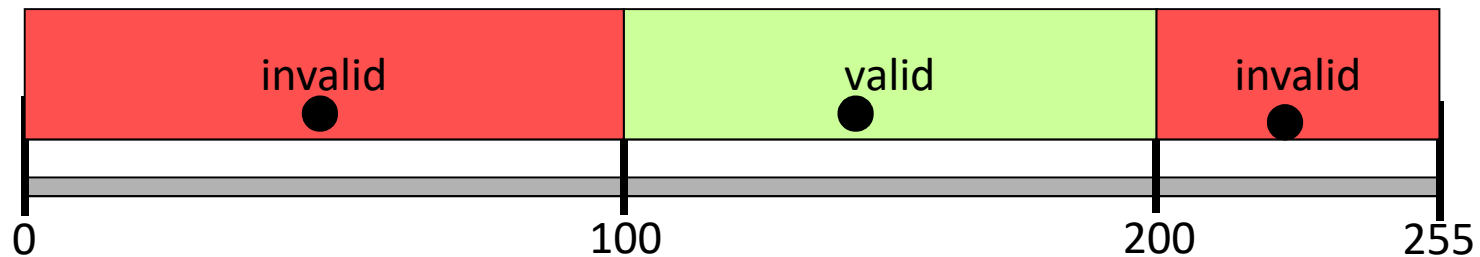
- Complex variable sets



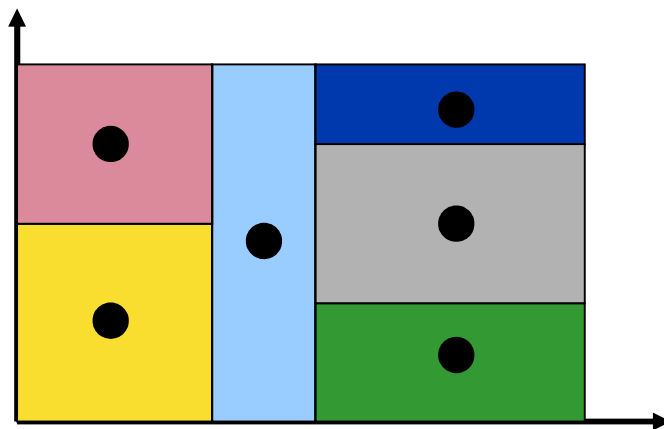
Equivalence partitioning examples

■ Examples

- Simple value ranges

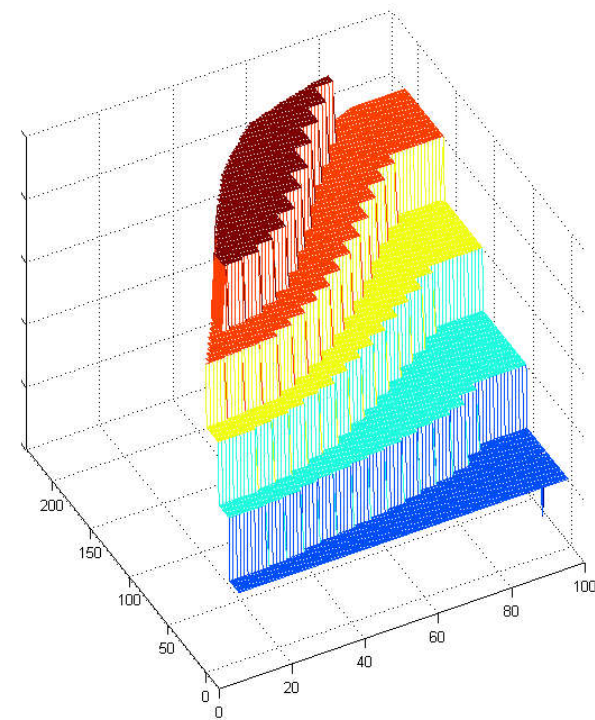
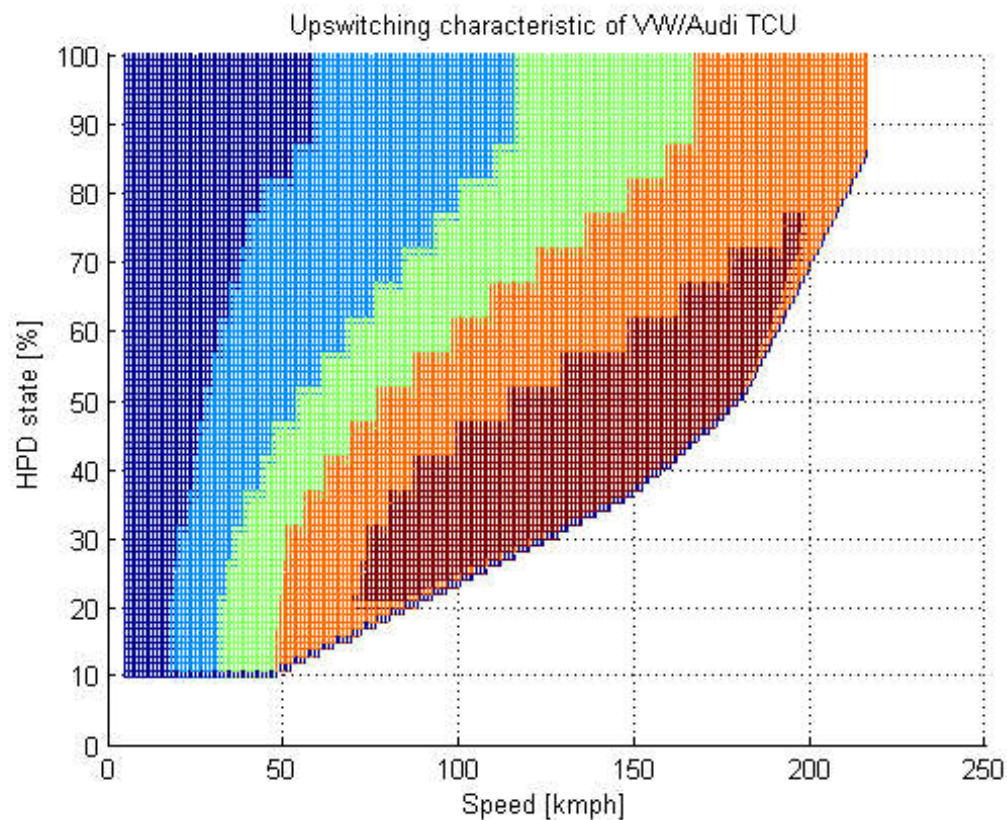


- Complex variable sets



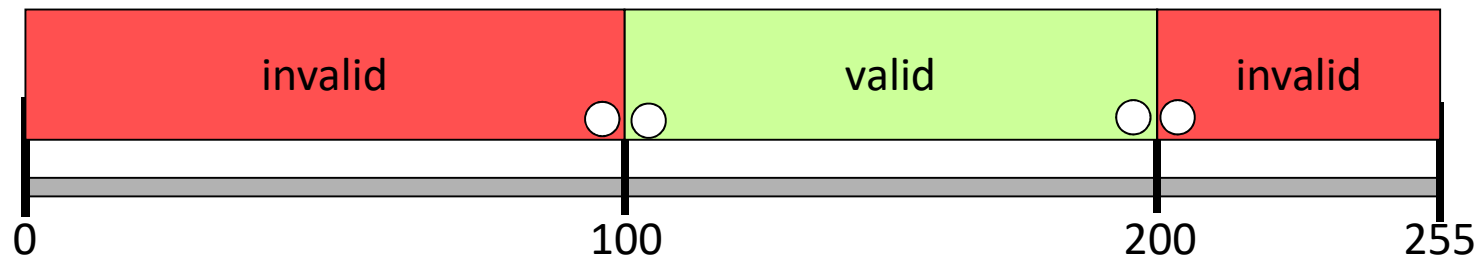
Equivalence partitioning examples

- Up switching characteristic of an automatic transmission controller



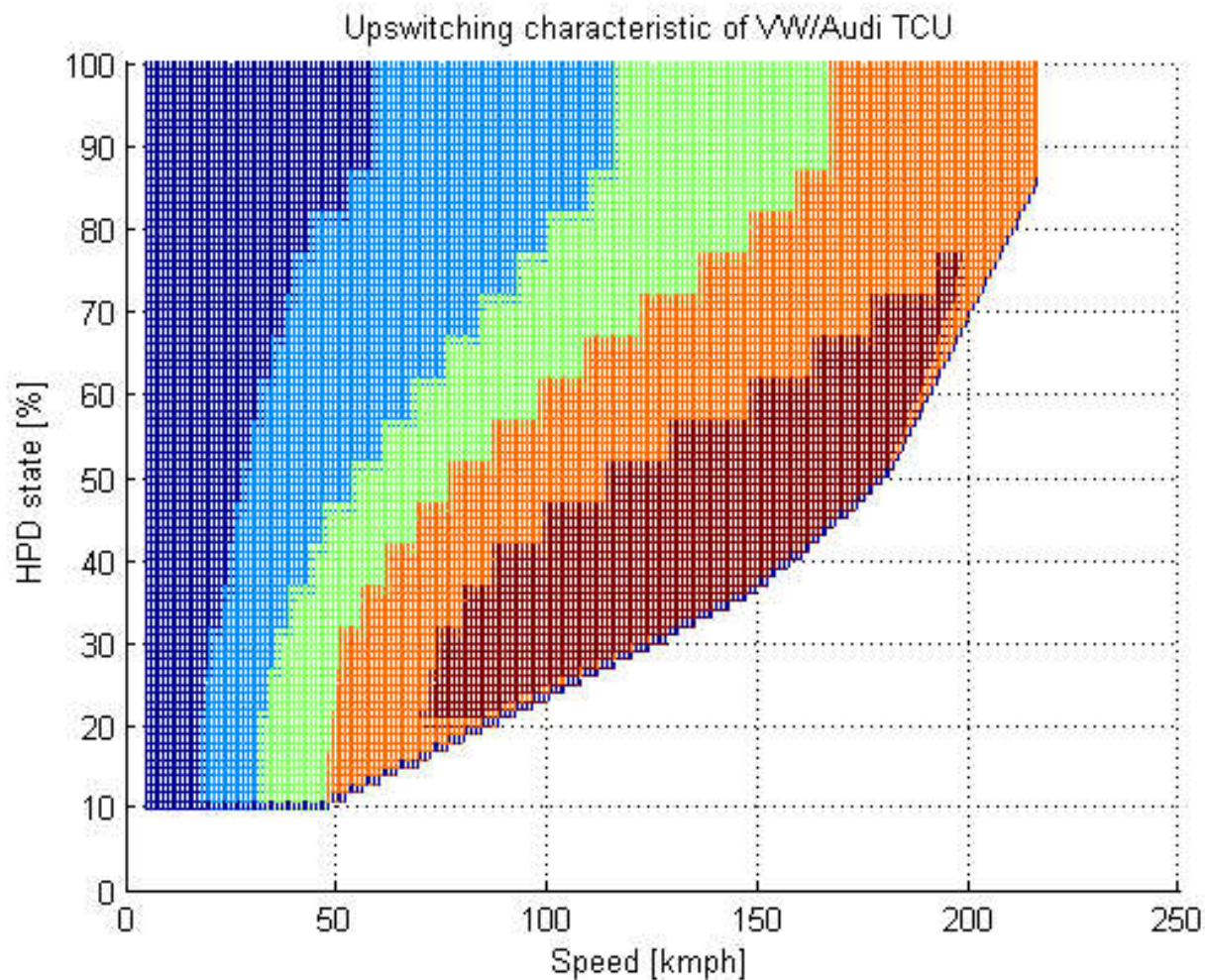
Boundary value analysis

- Defects like to hide in the partition boundaries
 - Wrong decision conditions
 - Wrong iteration exit or enter criteria
 - Wrong data structure handling
- Test both boundary of a cluster or partition



In a real case it can be far from trivial

- Up switching characteristic of an automatic transmission controller



Why should we both perform equivalence and boundary tests?

- Do we cover the whole partition by an boundary value test?

Why should we both perform equivalence and boundary tests?

- Do we cover the whole partition by an boundary value test?
 - Theoretically yes. It can work in an ideal world.
 - If the boundary value test signals a defect: is the whole partition works wrong, or just the boundary is not good?
 - must test some values inside the equivalence partition
 - Testing only extreme situations give less confidence,
 - must test some values inside the equivalence partition
 - Sometimes it is extremely hard to find the boundaries
 - Sometimes there can be a wrong partitioning

Cause-effect analysis, decision table

- Analyzing input relationship to output
 - **Cause:** an input partition
 - **Effect:** an output partition
 - Using bool algebra to this
 - $\text{speed} > 50\text{km/h}$
 - $\text{Gas pedal} < 30\%$
- Bool-graph
 - AND, OR relationships
 - Denied combinations
- Decision table
 - One column is one test case

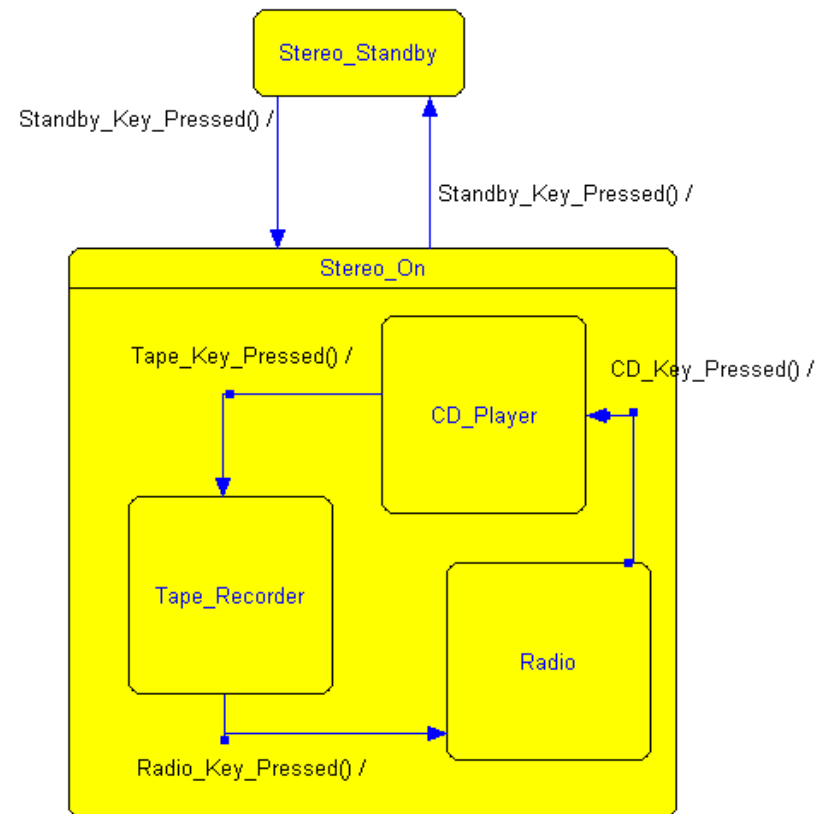
Cause-effect analysis, Example

- Webshop discount analysis for a customer

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
<i>New customer (15%)</i>	T	T	T	T	F	F	F	F
<i>Loyalty card (10%)</i>	T	T	F	F	T	T	F	F
<i>Coupon (20%)</i>	T	F	T	F	T	F	T	F
Actions								
<i>Discount (%)</i>	X	X	20	15	30	10	20	0

State transition testing

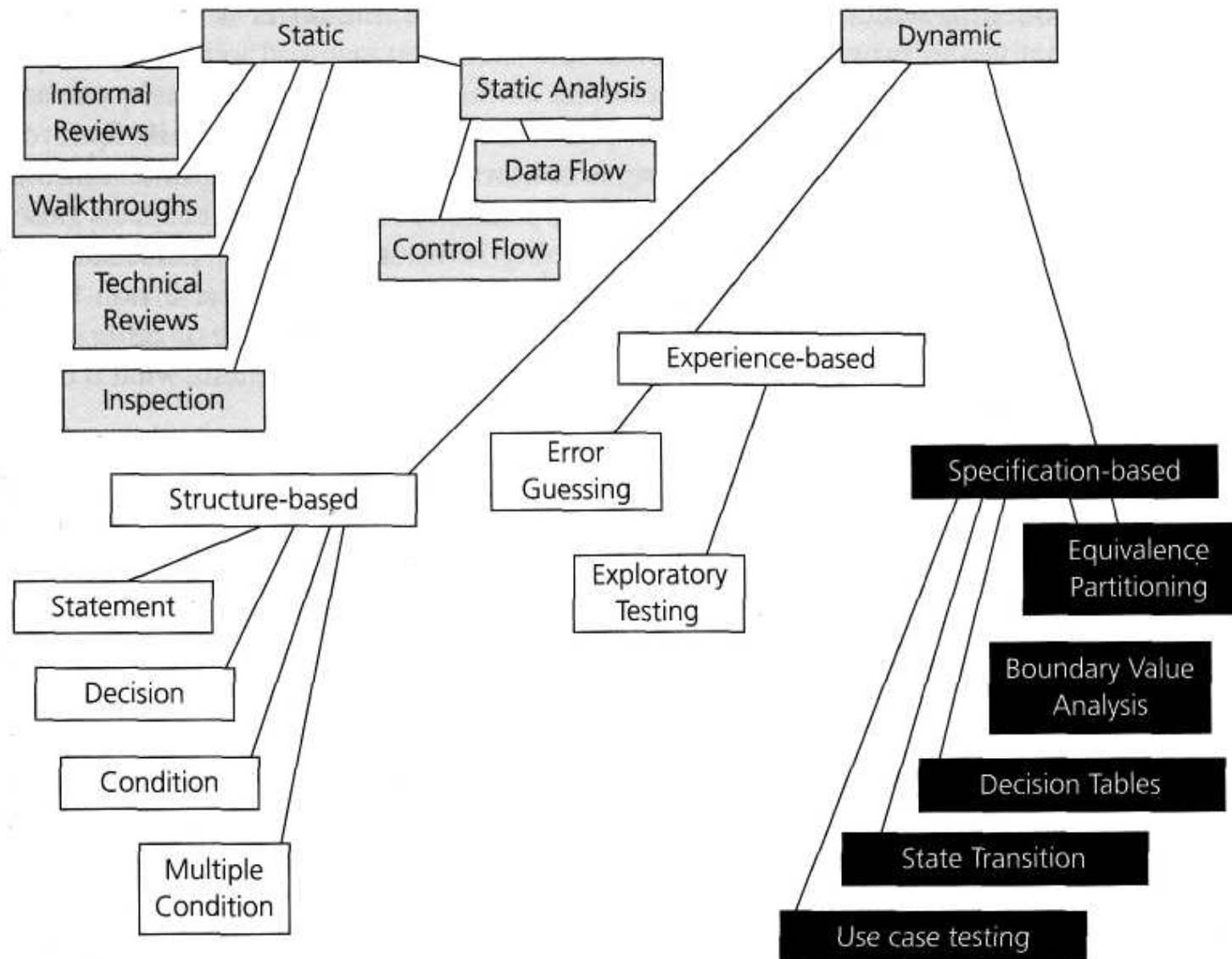
- Using the state diagram
- Test goals:
 - the states that the software may occupy
 - the transitions from one state to another
 - the events that cause a transition
 - the actions that result from a transition
 - Testing for invalid transitions



Use case testing

- Typically used at the final stage of integration as system tests
- Test scenarios generated from user stories

Structure based tests



Structure based methods

- White box methods
 - Requires the execution of code
- Gives easy to understand measurement values
- Alone it is not a test
- Do not protect against malfunction

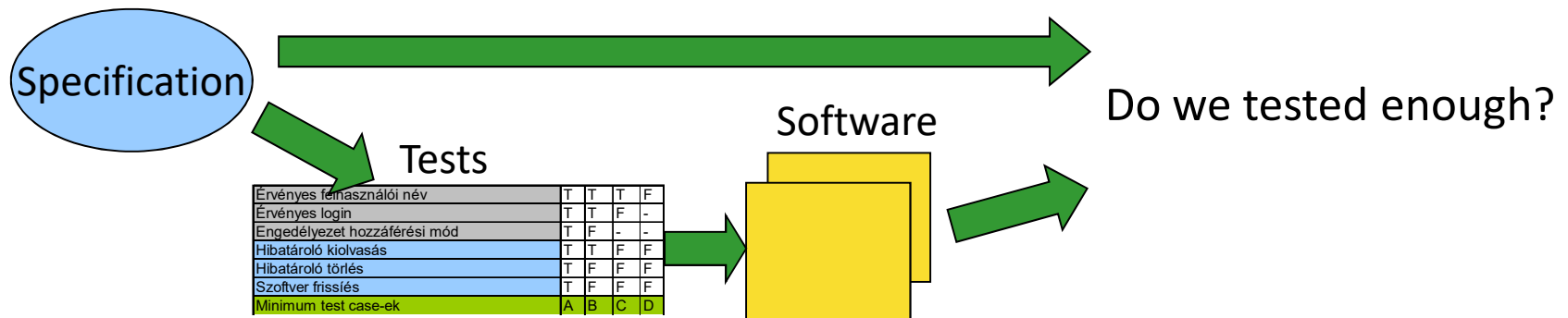
- Methods
 - Statement coverage
 - Decision coverage
 - Condition coverage
 - Path coverage

Terms

- Statement: one C language statement
- Statement Block
 - Continuous series of statements. Without branch, or decision.
- Condition
 - Simple logical condition (AND, OR ...)
- Decision
 - Decision based on one or more logical conditions
- Branch
 - A possible outcome of a Decision
- Path
 - Series of statements between the modules input and the output

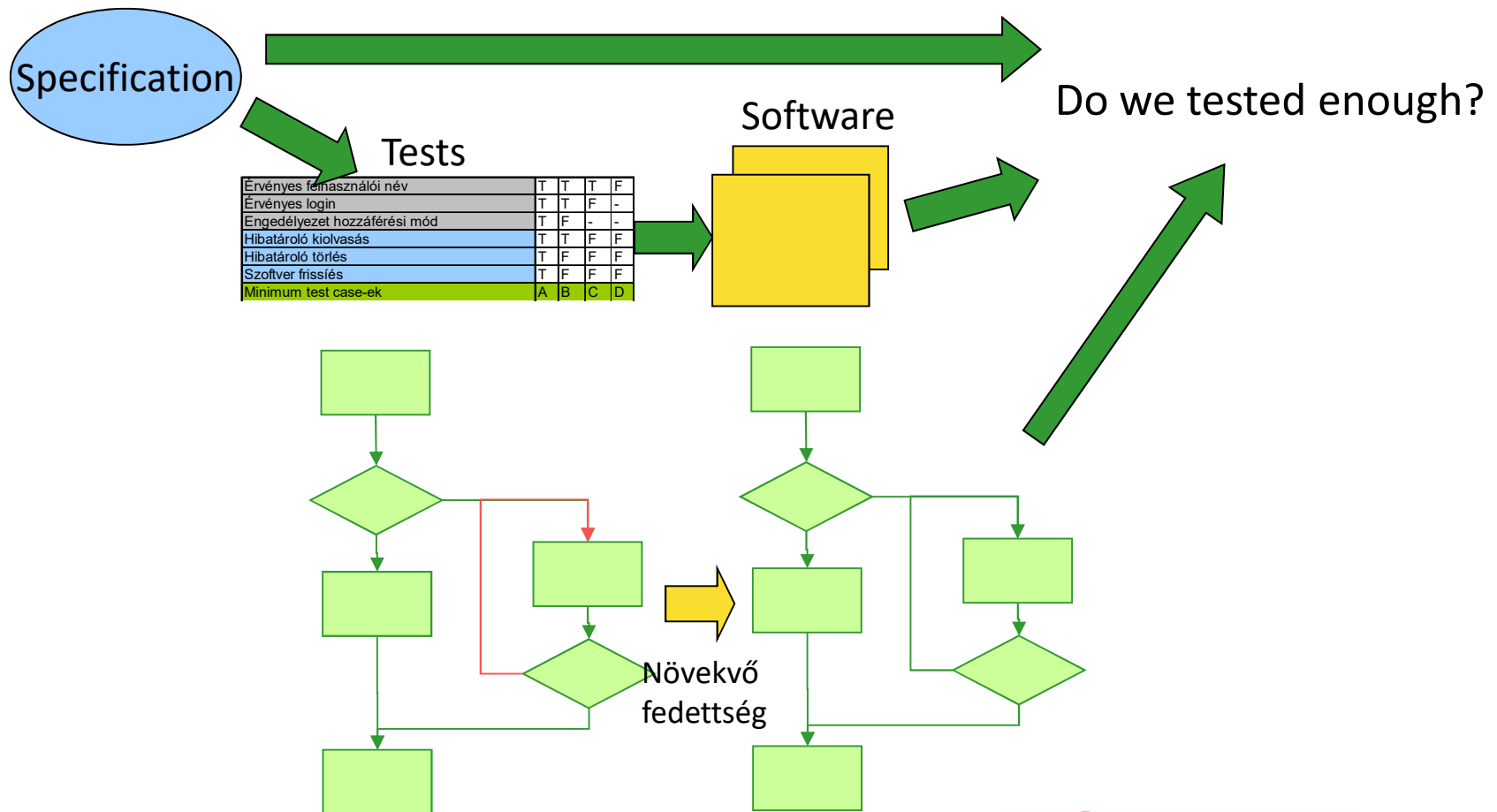
What do structure based methods good for?

- Typical application it to determine end of test condition.



What do structure based methods good for?

- Typical application it to determine end of test condition.

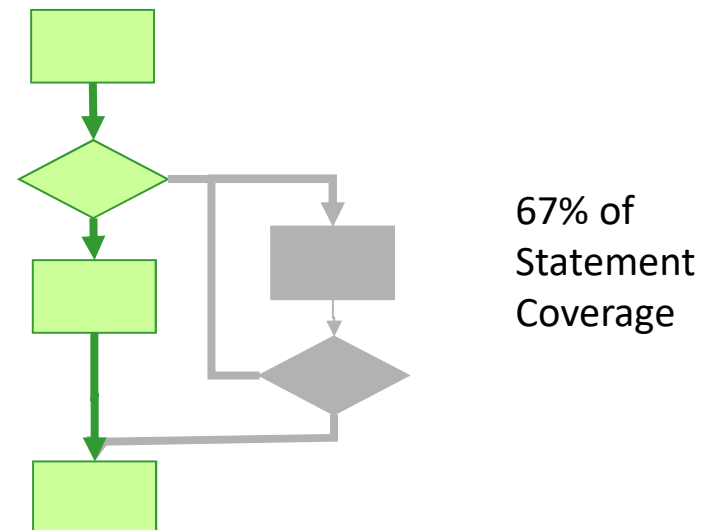
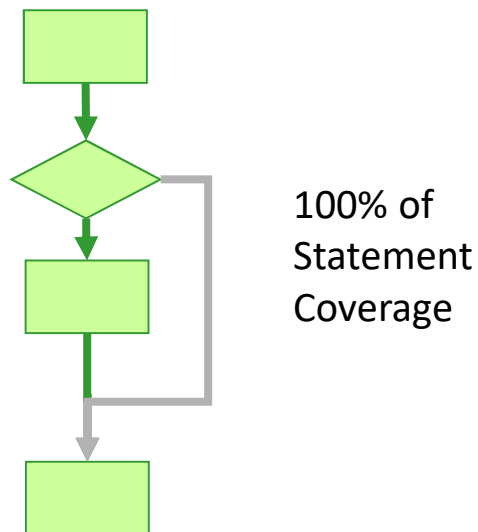


Statement coverage

- Definition:

$$\text{Statement coverage} = \frac{\text{Number of statements exercised}}{\text{Total number of statements}} * 100\%$$

Not a strong metric

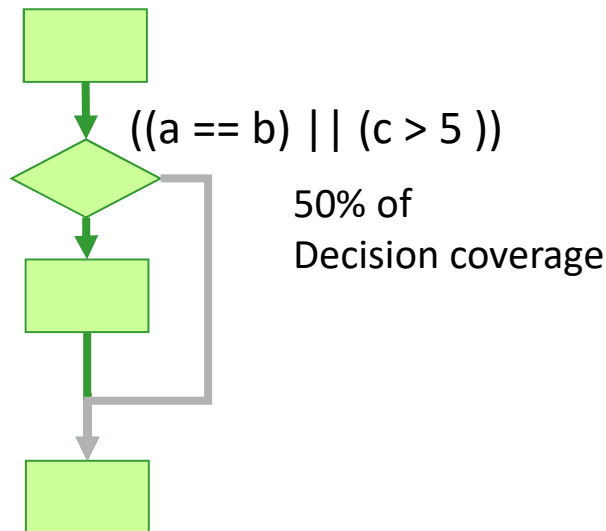


Decision coverage

- Definition:

$$\text{Decision coverage} = \frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} * 100\%$$

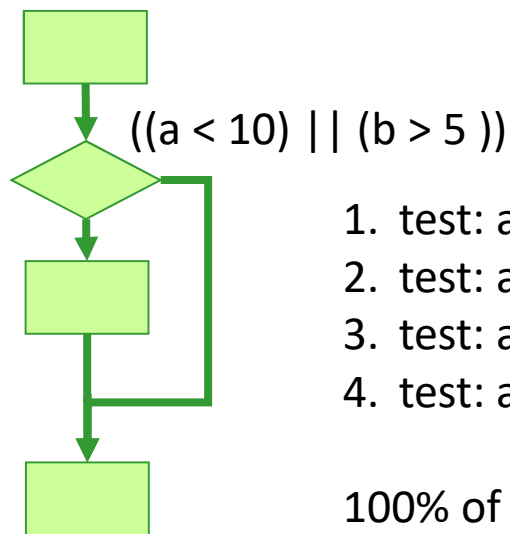
Rather strong metric



Multiple Condition Coverage

■ Definition:

- Every possible condition combination will be tested
- There is a huge grow in the test cases



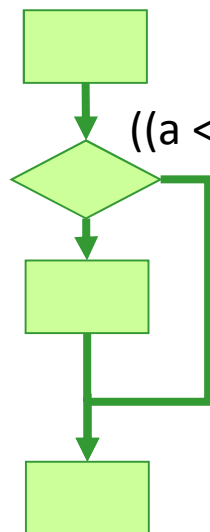
1. test: a = 5 (T), b = 0 (F)
2. test: a = 12 (F), b = 10 (T)
3. test: a = 15 (F), b = 2 (F)
4. test: a = 6 (T), b = 8 (T)

100% of
Multiple Condition Coverage

MC/DC coverage

■ Definition:

- Each entry and exit point is invoked
- Each decision takes every possible outcome
- Each condition in a decision takes every possible outcome
- Each condition in a decision is shown to independently affect the outcome of the decision



1. test: a = 5 (T), b = 0 (F)
2. test: a = 12 (F), b = 10 (T)
3. test: a = 15 (F), b = 2 (F)

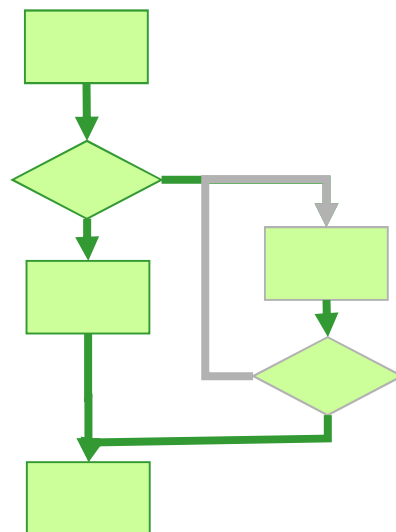
100% of
MC/DC coverage

Path coverage

- Definition:

$$\text{Path Coverage} = \frac{\text{Number of executed path}}{\text{Total number of possible path}} * 100\%$$

Too strong metric



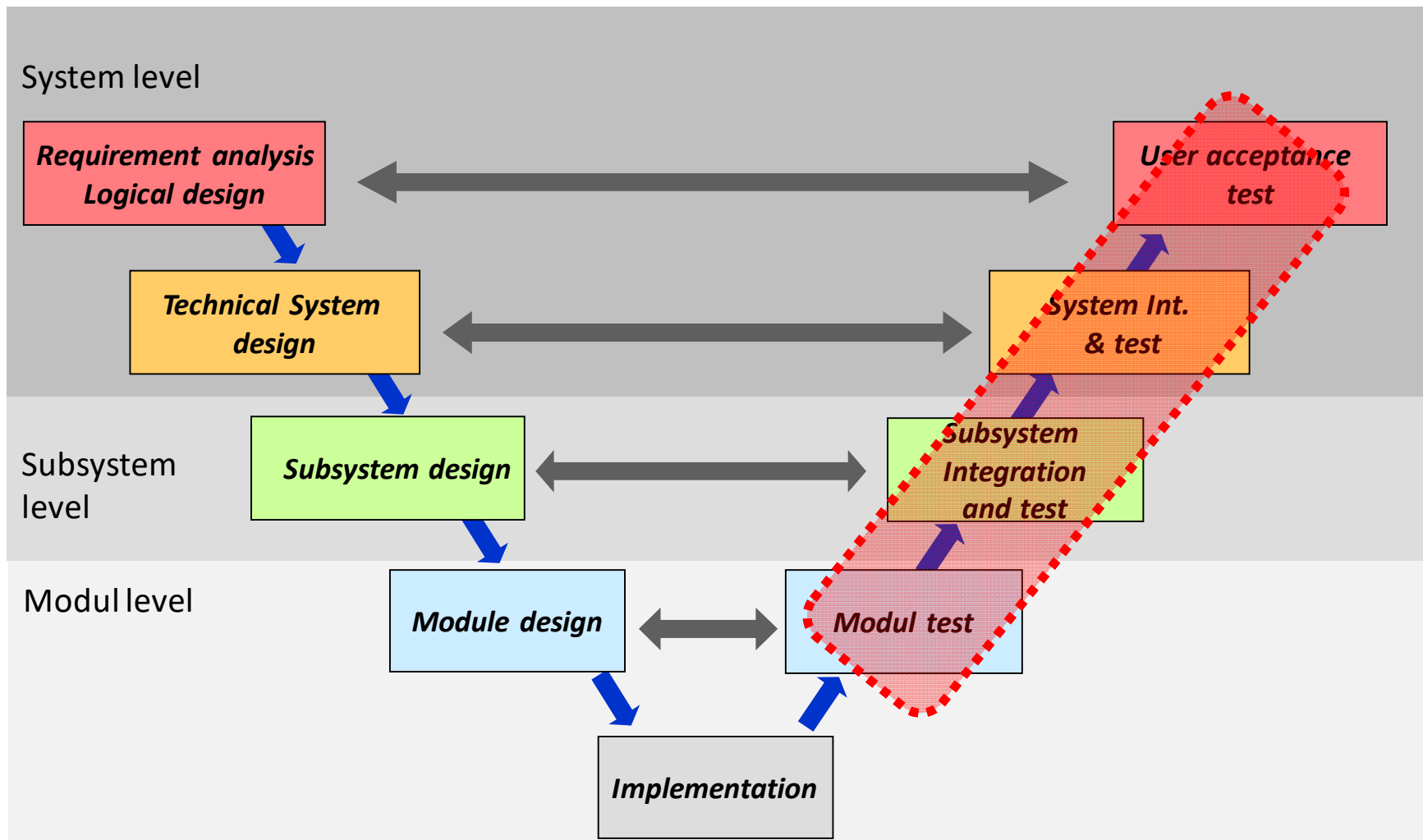
67% of
Path coverage

Non systematic tests

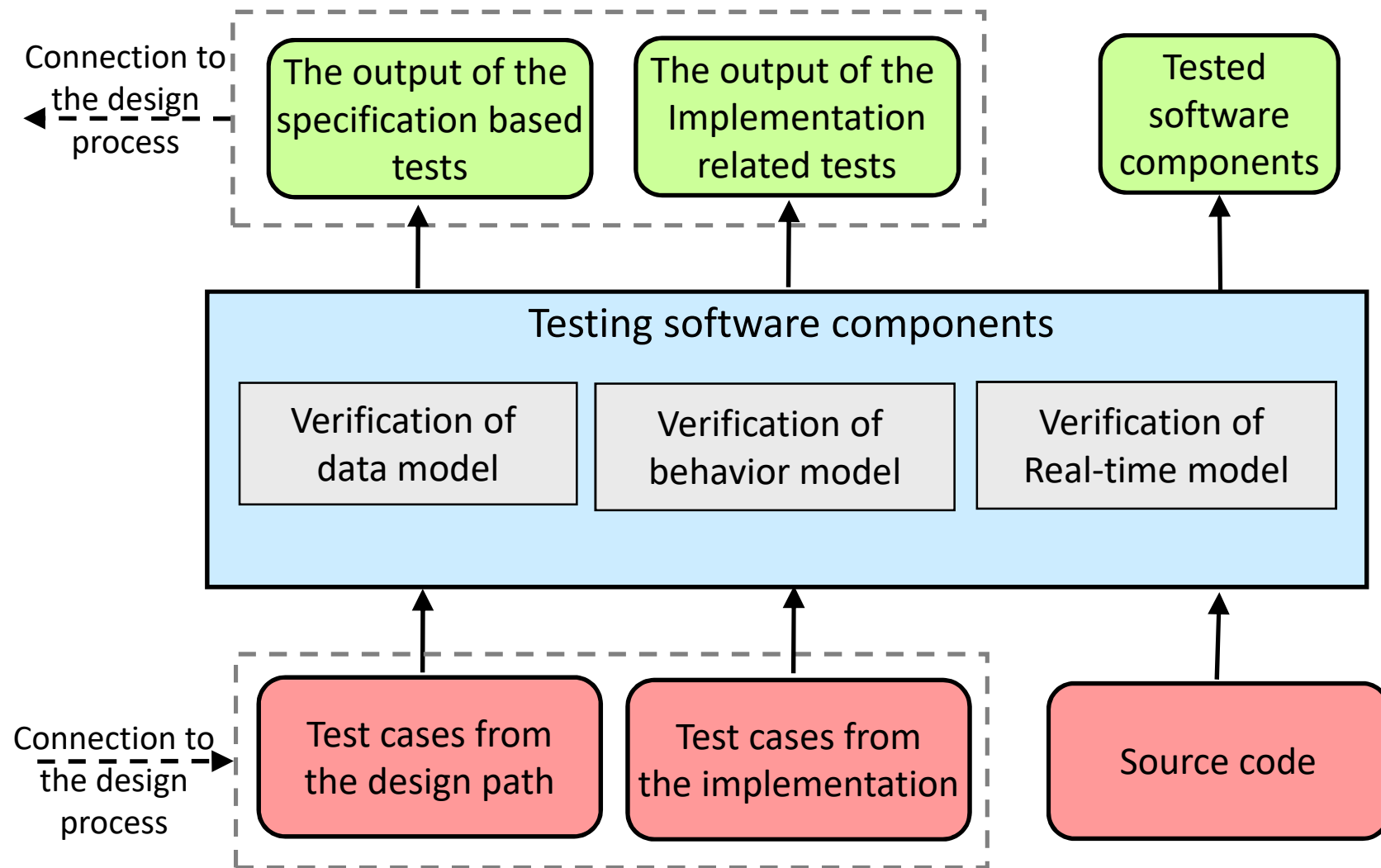
- Ad-hoc trial and error
- Error guessing
 - Executed after normal tests
 - There are no rules for error guessing
 - Typical conditions to try include division by zero, blank (or no) input, empty files and the wrong kind of data
- User testing
 - Acceptance tests made by the end users
 - Based on the usage scenarios

Dynamic test techniques in V-model

- Application of the methods above



Software Component Testing

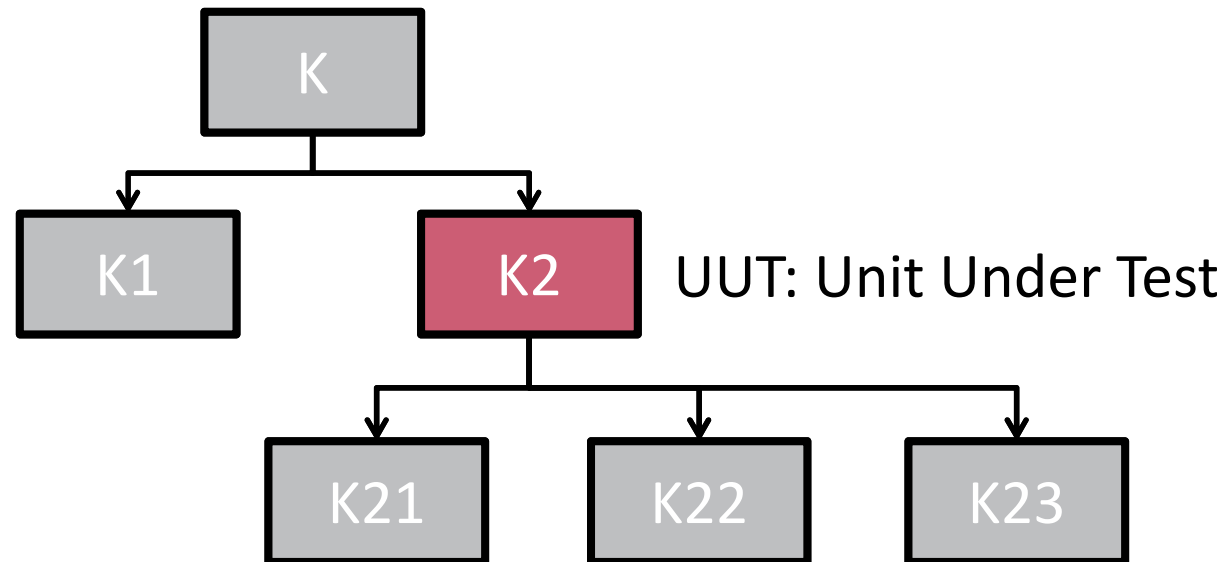


Module testing

- Goal is to find the problems as early as possible
 - Typically done by the developer
- Every module is handled separately
 - Low complexity tests
 - Easy to localize the problem
- Easier to integrate individually tested modules
 - Way to handle the complexity
- Module tests can be automated
 - Should be executed many times, therefore it should be fast and easy to execute

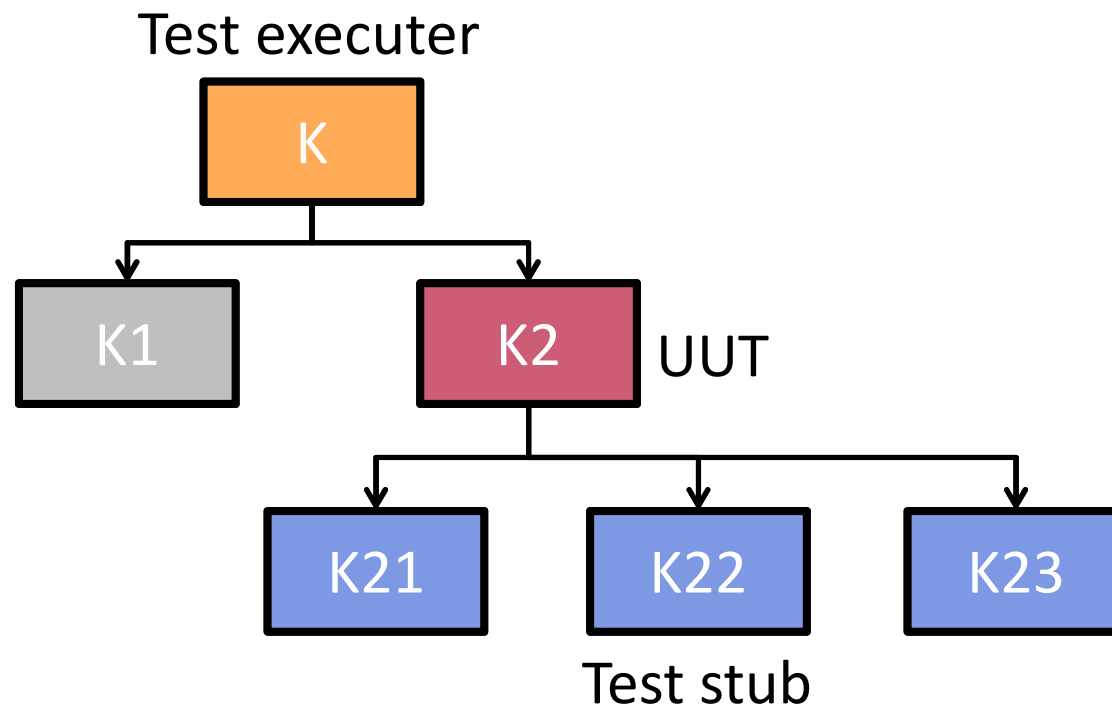
Typical problems of module testing

- The modules should be tested as separate component, but it depends on other modules
 - Isolation



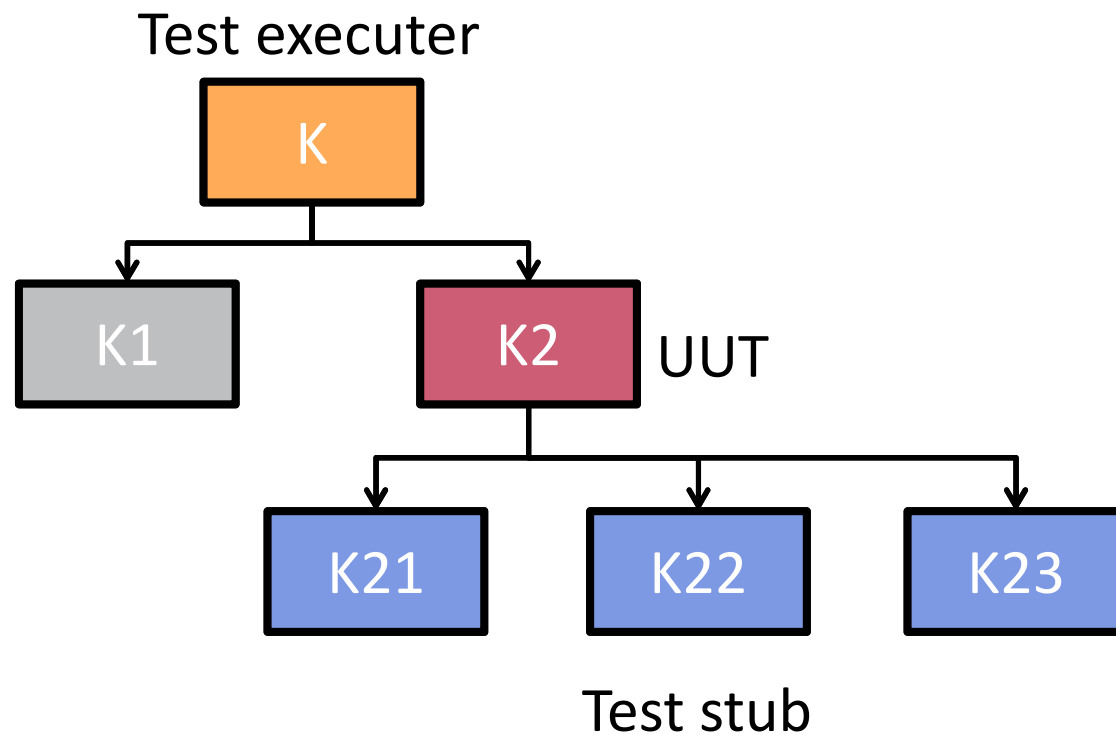
Typical problems of module testing

- The modules should be tested as separate component, but it depends on other modules
 - Isolation



Typical problems of module testing

- The modules should be tested as separate component, but it depends on other modules
 - Isolation



- Many times not executed in the real hardware: emulation

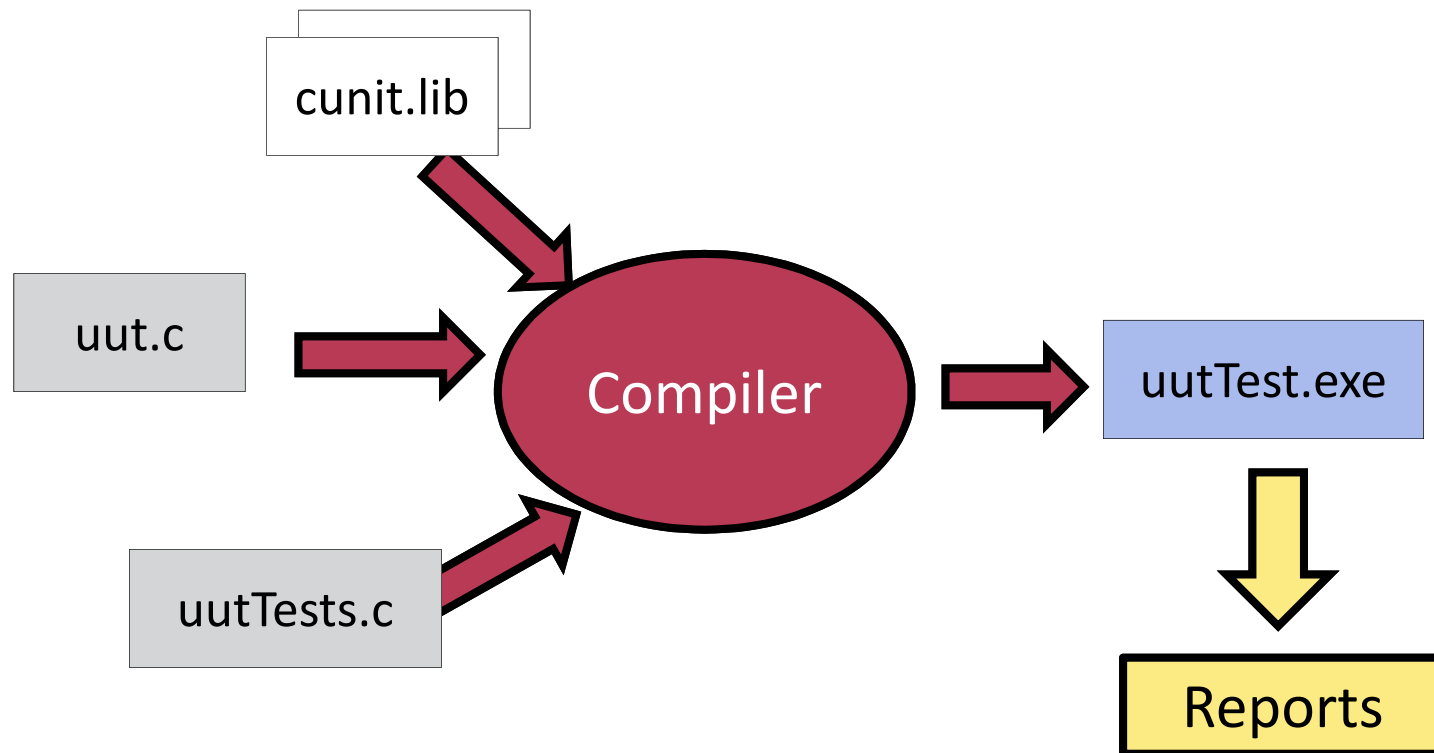
Requirements for unit tests in automotive

Methods		ASIL			
		A	B	C	D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes	+	++	++	++
1c	Analysis of boundary values ^a	+	++	++	++
1d	Error guessing ^b	+	+	+	+
^a This method applies to interfaces, values approaching and crossing the boundaries and out of range values.					
^b "Error guessing tests" can be based on data collected through a "lessons learned" process and expert judgment.					

Methods		ASIL			
		A	B	C	D
1a	Statement coverage	++	++	+	+
1b	Branch coverage	+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage)	+	+	+	++

Tools of specification based tests

- xUnit, cUnit, Unity: Easy libraries for automated unit tests



- Using simple assertions:
CU_ASSERT_TRUE(a), CU_ASSERT_EQUAL(a, b);

Tools of structure based tests

- **Software based tools**

- Code instrumentation
 - Overhead: time, data and programmemory
- We test code with instrumentation. If we remove the instrumentation, then that is not the code we tested.

Tools of structure based tests

■ Software based tools

- Code instrumentation
 - Overhead: time, data and program memory
- We test code with instrumentation. If we remove the instrumentation, then that is not the code we tested.

■ Running the code in an Emulator

- There is no need for code instrumentation
- Not exactly the same environment as the real one
 - Peripheral handling, timing
- Using the debugger

Tools of structure based tests

■ Hardware based measurement

- Monitoring memory data and address lines
 - On chip memories of modern microcontrollers do not make it possible
- Internal microcontroller dependent support needed
 - Trace modules
 - ARM Embedded trace module
- Costly hardware interface

Instrumentation tool: GCOV

- Part of the GCC toolchain, can be ported to embedded systems
- Instruments the code and creates logfiles

```
#include <stdio.h>
int main (void)
{
    1   int i;
    10  for (i = 1; i < 10; i++)
        {
            9   if (i % 3 == 0)
                3   printf ("%d is divisible by 3\n", i);
            9   if (i % 11 == 0)
                #####   printf ("%d is divisible by 11\n", i);
            9   }
    1   return 0;
    1 }
```

Hardware measurement based tool

■ Hardveres coverage tool

[B::Data.ListHll /Cflag NoOK]

coverage	addr/line	source
ok	610	ast.field2 = 2;
ok	612	ast = func4(ast);
ok	614	j = (*funcptr)();
ok	616	start:
ok	617	j = func5((int) j
partial	619	if (j == 0)
never	620	goto start
ok	622	vfloat = 2.0;
ok	624	func6(vfloat, (fl
ok	626	vdouble = 2.0;
ok	628	func7(vdouble, (d
ok	630	func8();
ok	632	func9();

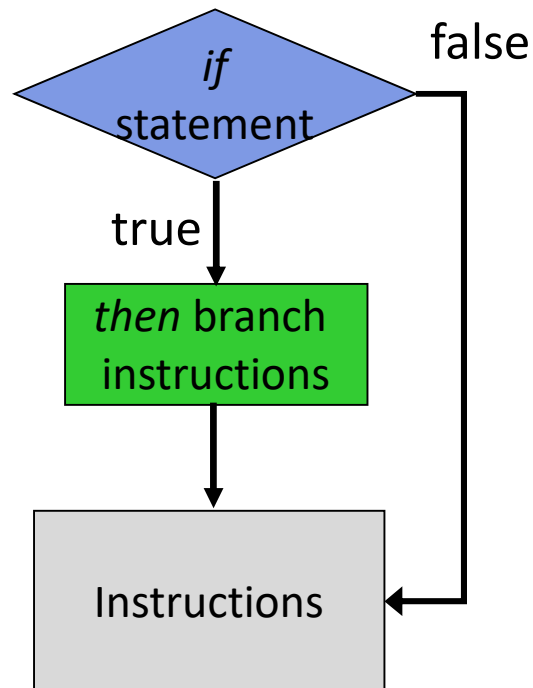


B::Trace.COVeRage.ListFunc

address	coverage	executed 0%	50%	100	taken	nottaken
P:000010CC-00002377	partial	88.033%			48.	46.
R:000010CC-000010D3	never	0.000%			0.	0.
R:000010D4-000010E3	ok	100.000%			0.	0.
R:000010E4-0000118F	partial	93.023%			1.	1.
R:00001190-000011DF	ok	100.000%			1.	1.
R:000011E0-00001223	ok	100.000%			1.	1.
R:00001224-0000131B	partial	88.709%			1.	1.
R:0000131C-00001373	ok	100.000%			1.	1.
R:00001374-00001383	partial	50.000%			0.	0.
R:00001384-0000138B	partial	92.857%			0.	0.
R:000013BC-000013D3	partial	66.666%			0.	0.
R:000013D4-00001447	partial	89.655%			0.	0.
R:00001448-000014D3	partial	74.285%			0.	0.
R:000014D4-0000176F	partial	99.401%			0.	0.
R:00001770-00001803	partial	89.189%			2.	2.
R:00001804-00001CFF	partial	99.373%			33.	33.
R:00001D00-00001D7B	partial	19.354%			1.	0.
R:00001D7C-00001DCB	partial	90.000%			1.	1.
R:00001DCC-00001DE3	partial	66.666%			0.	0.
R:00001DE4-00001DFF	partial	71.428%			0.	0.
R:00001E00-00001E13	partial	60.000%			0.	0.
R:00001E14-00001E37	partial	77.777%			0.	0.
R:00001E38-00001E53	partial	71.428%			0.	0.

Questions about condition coverage

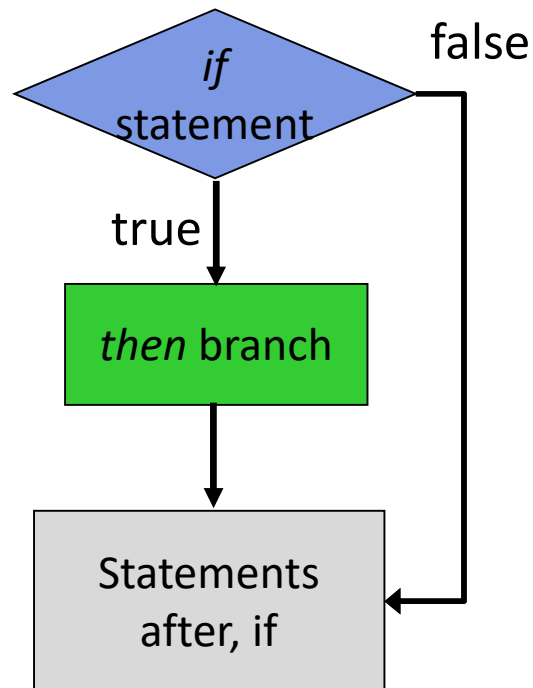
- Example code: `if ((a == 10) && (b == 20))`



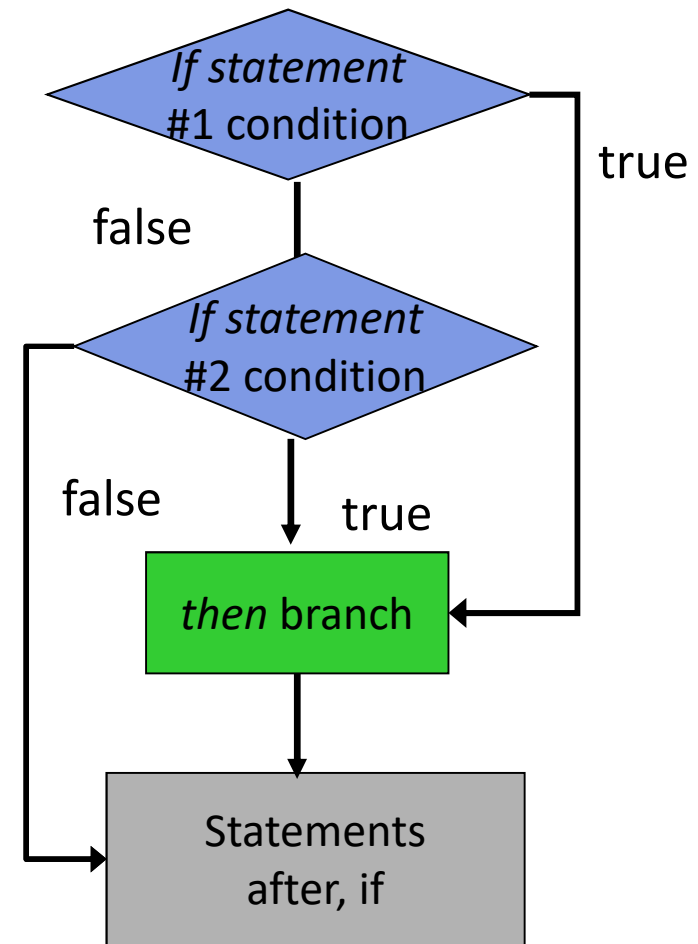
- How can we measure condition coverage???

Questions about condition coverage

- Example code: `if ((a == 10) && (b == 20))`

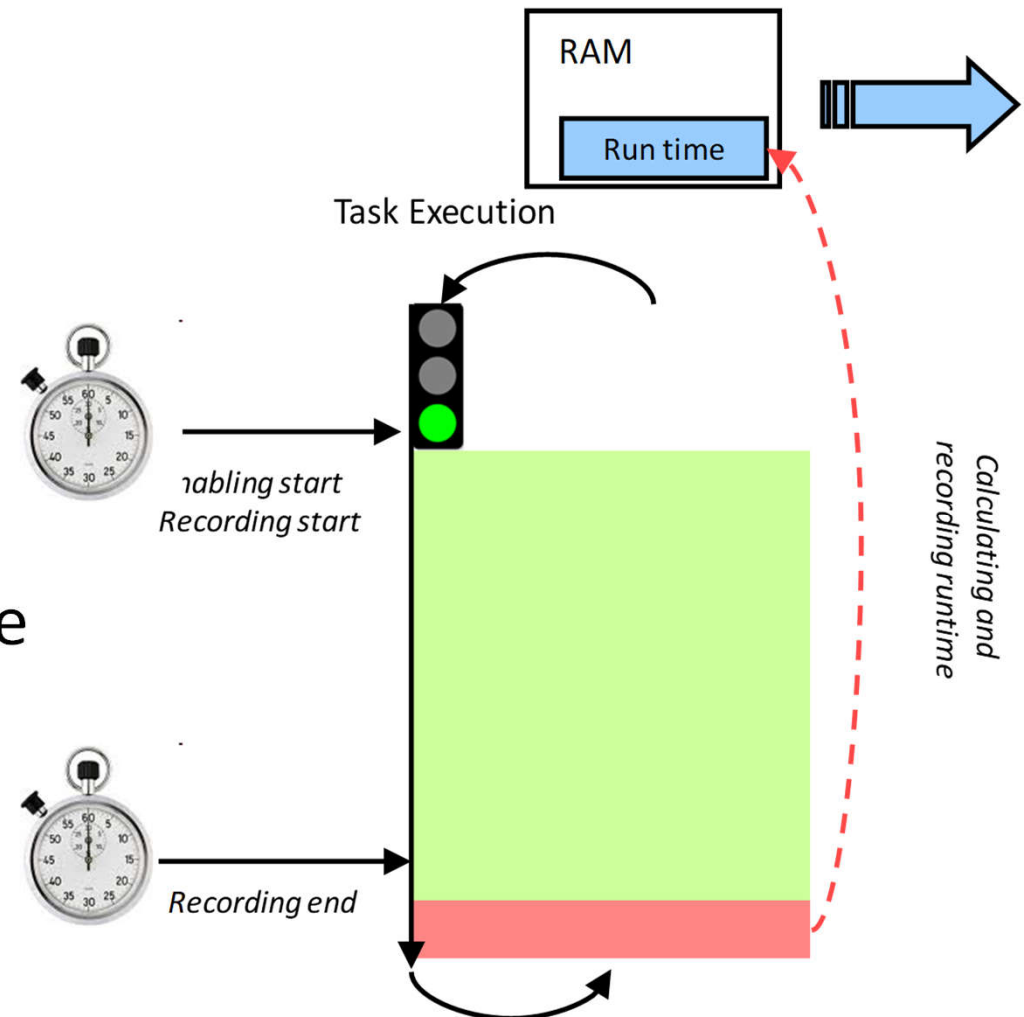


Basic Blokkokras



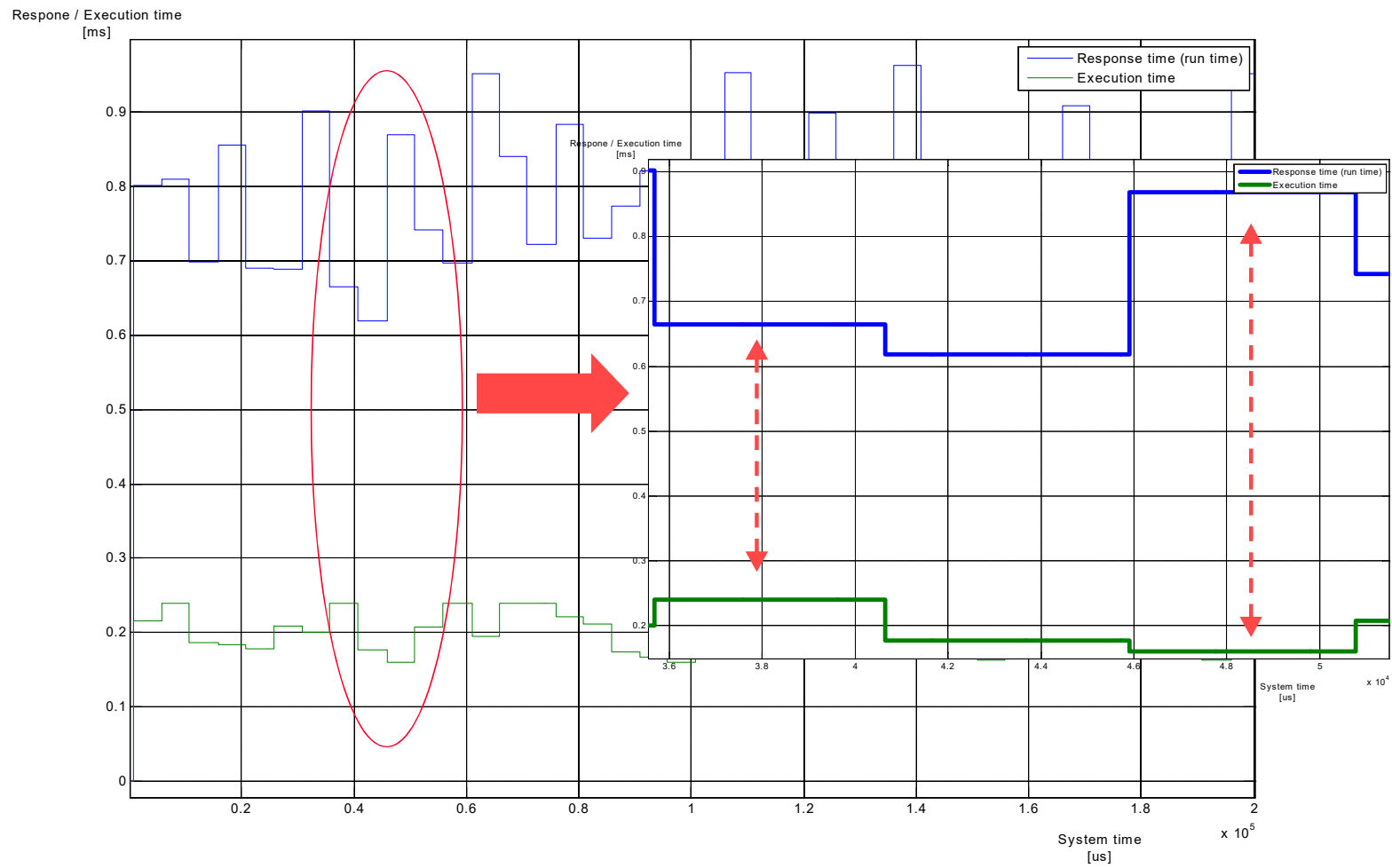
Real-time model checking

- Schedulability calculation
- What information do we need?
- It is enough to measure the response time?



Response time van be tricky

- Execution time which is important



Execution time is needed for schedulability calculations

- **DMA: Deadline Monotonic analysis**

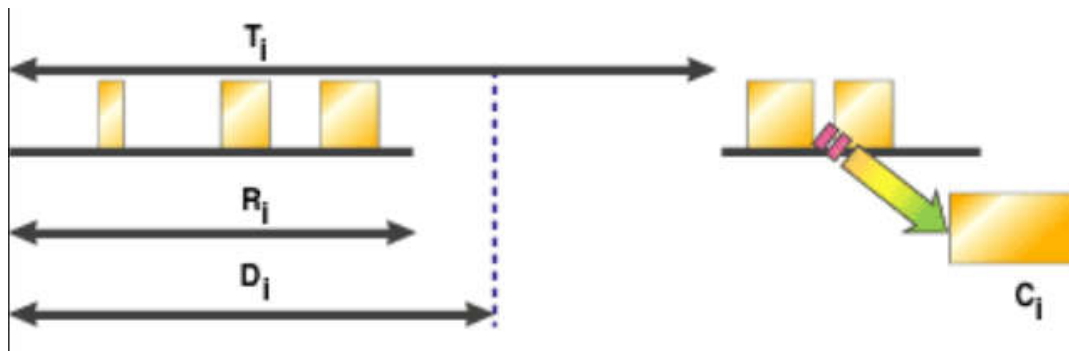


Table 1 DMA example

Task	T	C	D
1	250ms	5ms	10ms
2	10ms	2ms	10ms
3	330ms	25ms	50ms
4	1000ms	29ms	1000ms

Table 2 Calculation of worst-case Task 3 response time

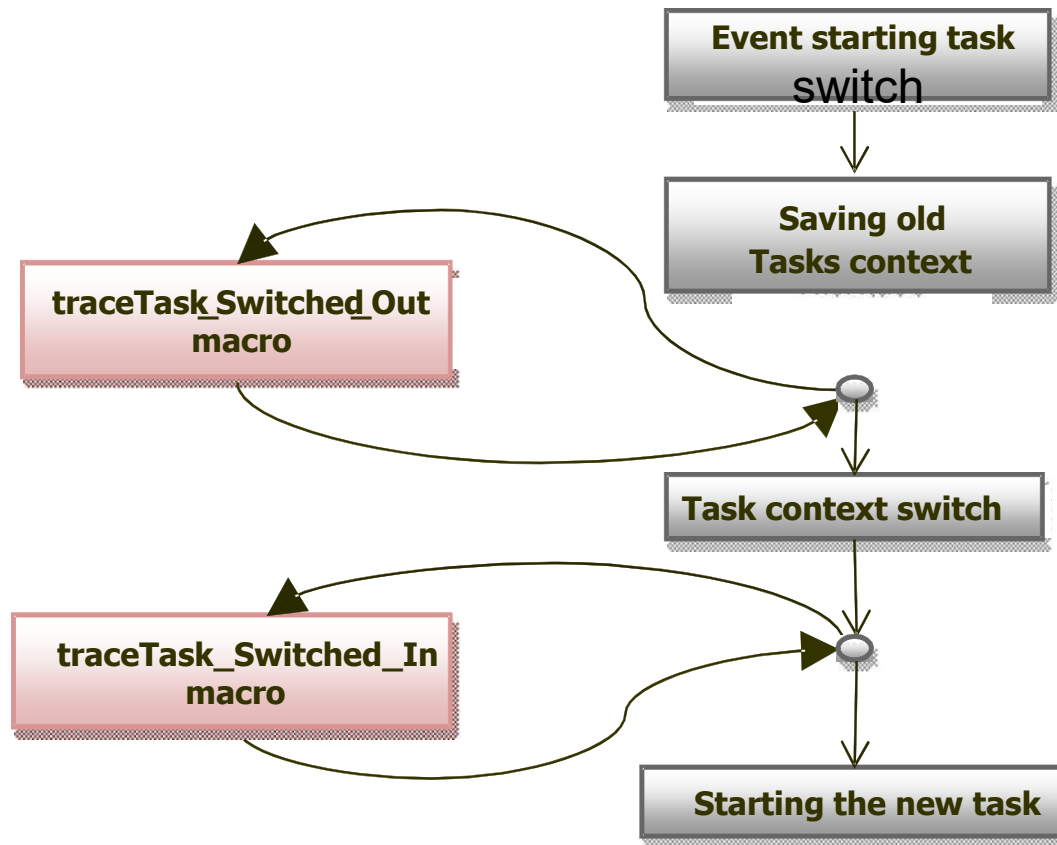
Step	R ⁿ	I	R ⁿ⁺¹
1	0	0	25
2	25	5+3x2=11	36
3	36	5+4x2=13	38
4	38	5+4x2=13	38

$$R_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

- How to measure execution time?

RTOS Trace hooks

- For example FreeRTOS has such hooks

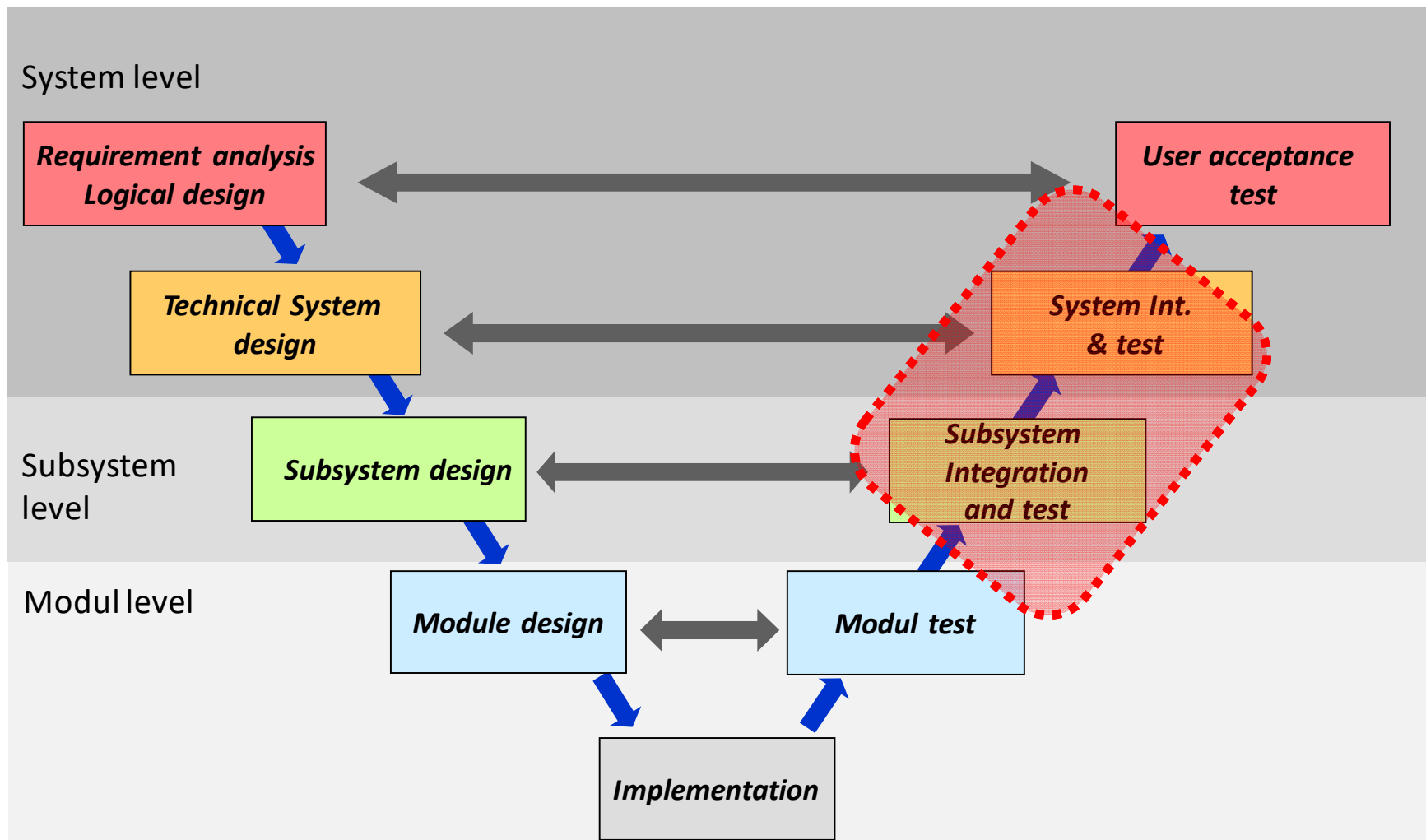


DWT: Data Watchpoint Trace

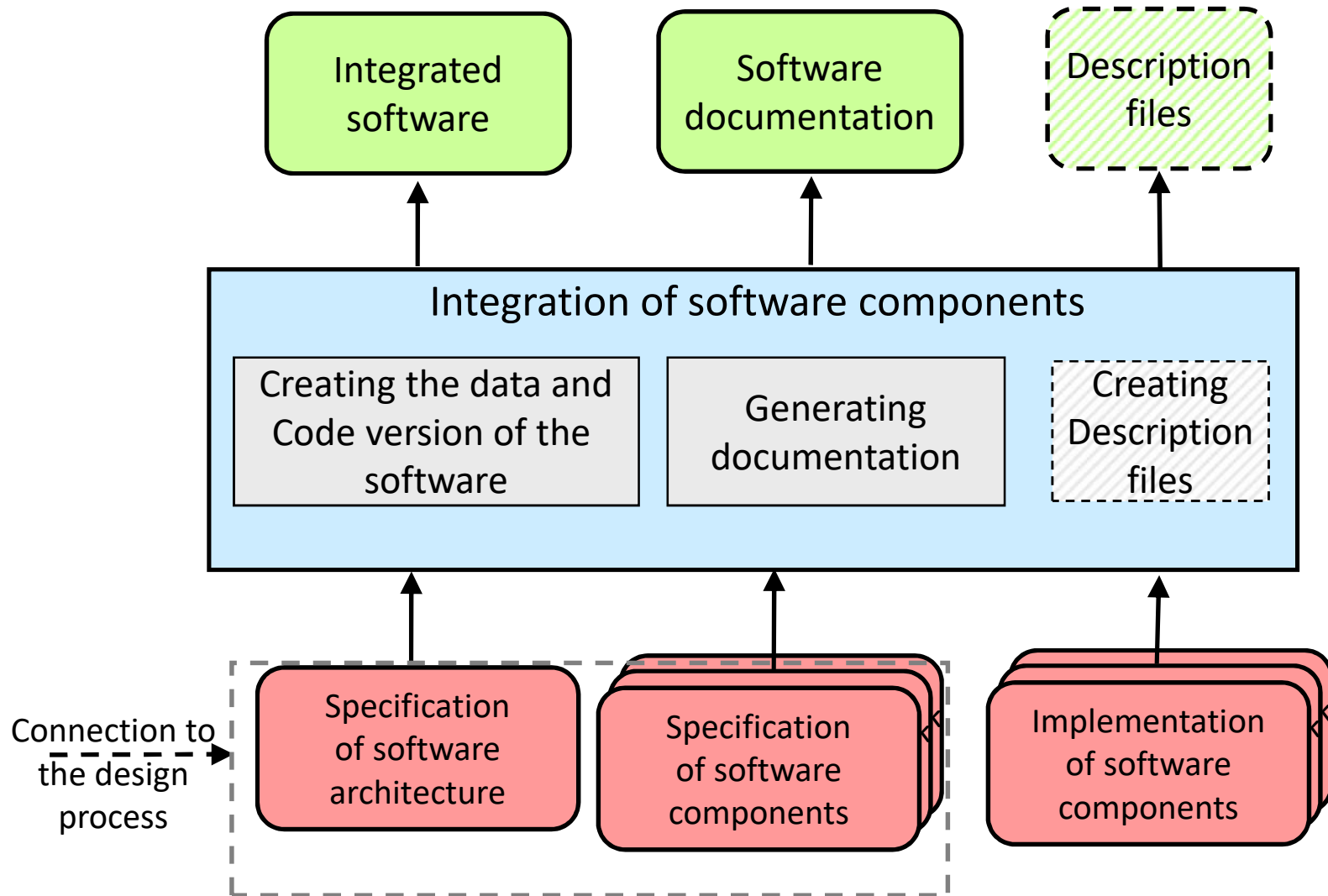
- 4 comparator: data address / program counter
 - Hardware watchpoint: put the processor into debug state
 - ETM trigger: start trace packest
 - PC sample trigger
 - Data address sample trigger
- Counters
 - Clock cycle counter
 - Sleep cycle counter
 - Interrupt overhead counter
- PC sampling
- Interrupt trace

Integration tests

- Application of the methods above



Software integration



Integration methods: *Big-bang*

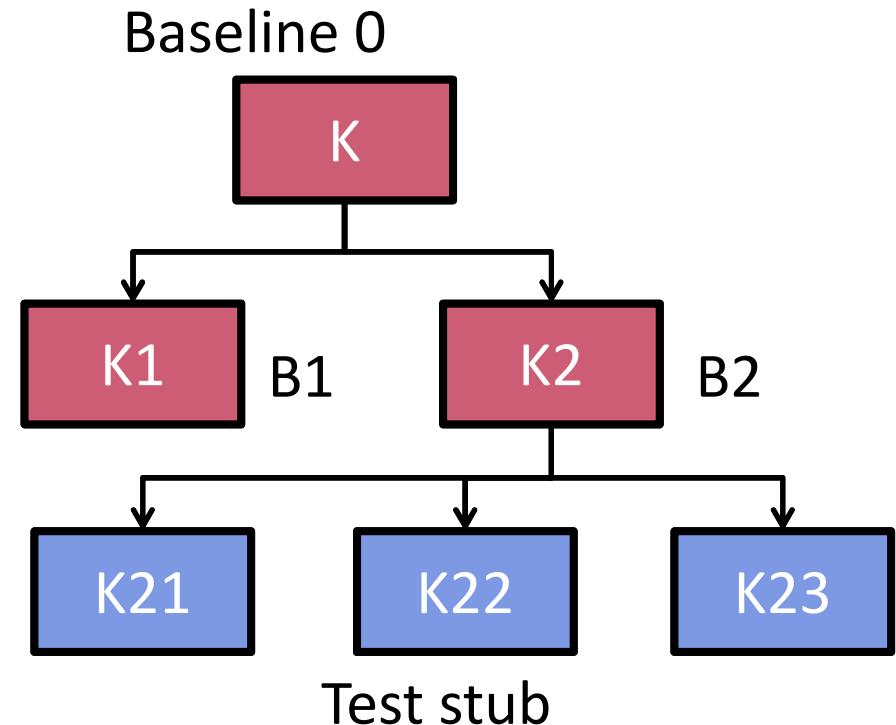
- Big-bang
 - Integration of all the tested components
 - Harder to find the problems
 - Confirmation testing after bug fixing requires more effort
 - Can be effective in case of small systems

Incremental integration

- Baseline 0: 1 tested components...
- Baseline 1: 2 tested components ...
- Baseline 2: 3 tested components ...
- Benefits
 - Easier to localize the problems
 - Easier to recover after problems

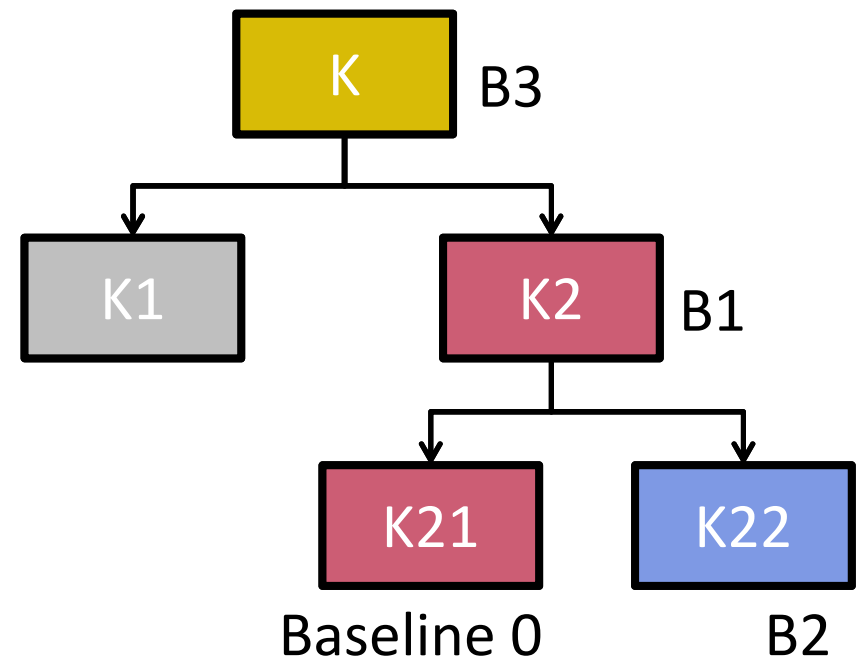
Top-down integration

- From top to bottom
- There is a need for test stubs
 - Simple printf
 - Time delays
 - Simple calculation based response
 - Response from a table
- Benefits
 - The higher level functionality is tested first
- Drawbacks
 - There is a need for test stubs
 - The details are tested late
 - The result is just a simulation for very long time



Bottom-up integration

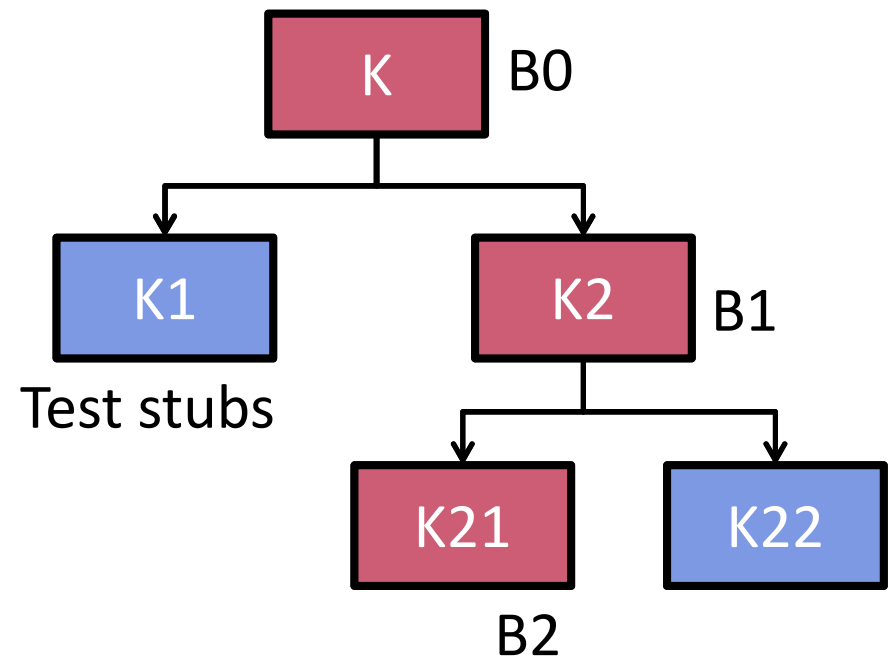
- Start from bottom, to up
- Benefits
 - Good for hardware intensive systems
 - We have real data and real timing from the start
- Drawbacks
 - High level functions are tested at the end
 - There is a need for test drivers, and stubs



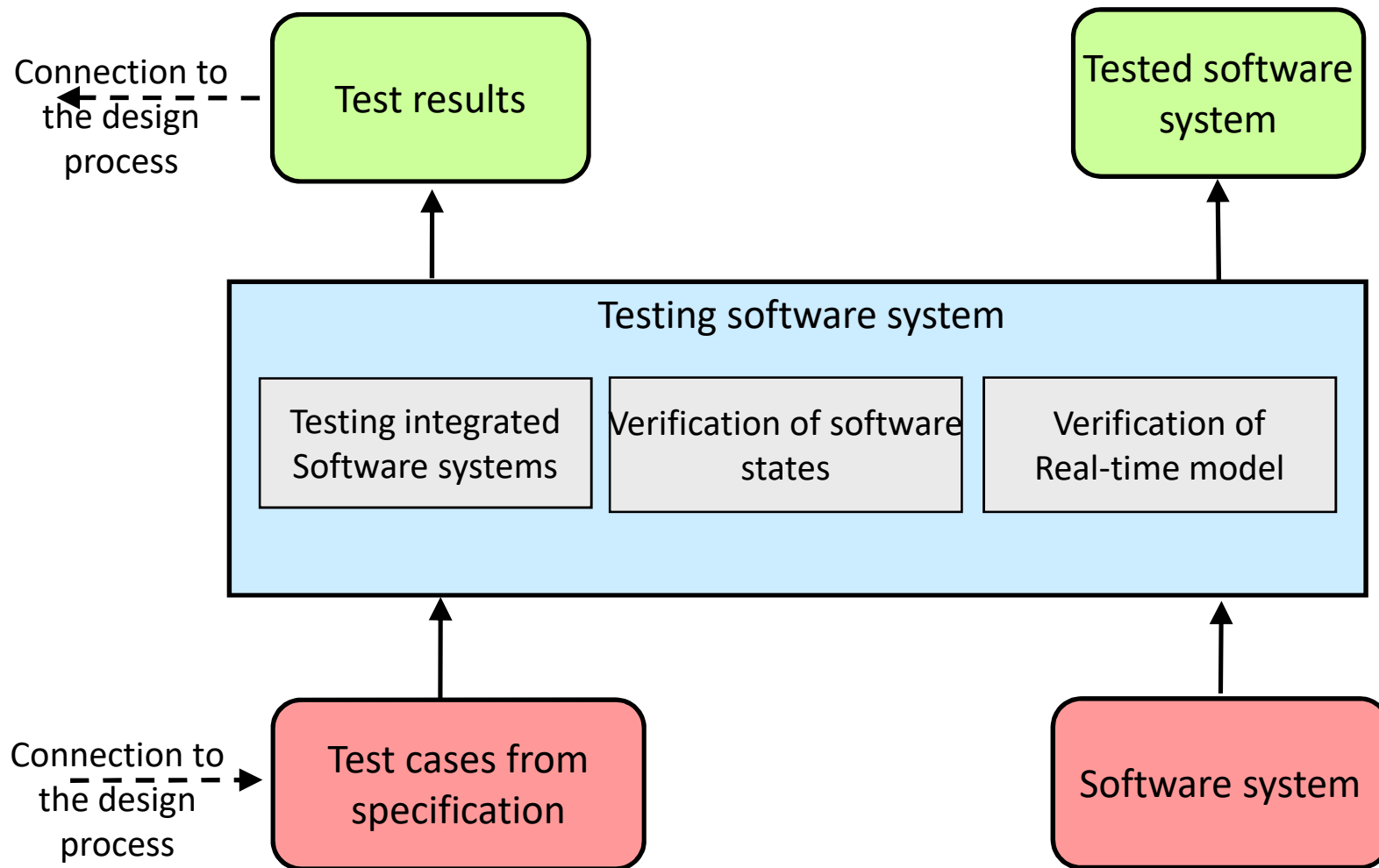
Test stubs

Functional integration

- Function by function integration
- Benefits
 - The highest abstraction level is tested many times
 - Having real responses early
 - Real working functionality very early
- Drawbacks
 - Many test stubs needed



Testing software systems



Typically used integration tests

- Specification based techniques
 - equivalence partitioning
 - boundary value analysis
 - decision tables
 - state transition testing

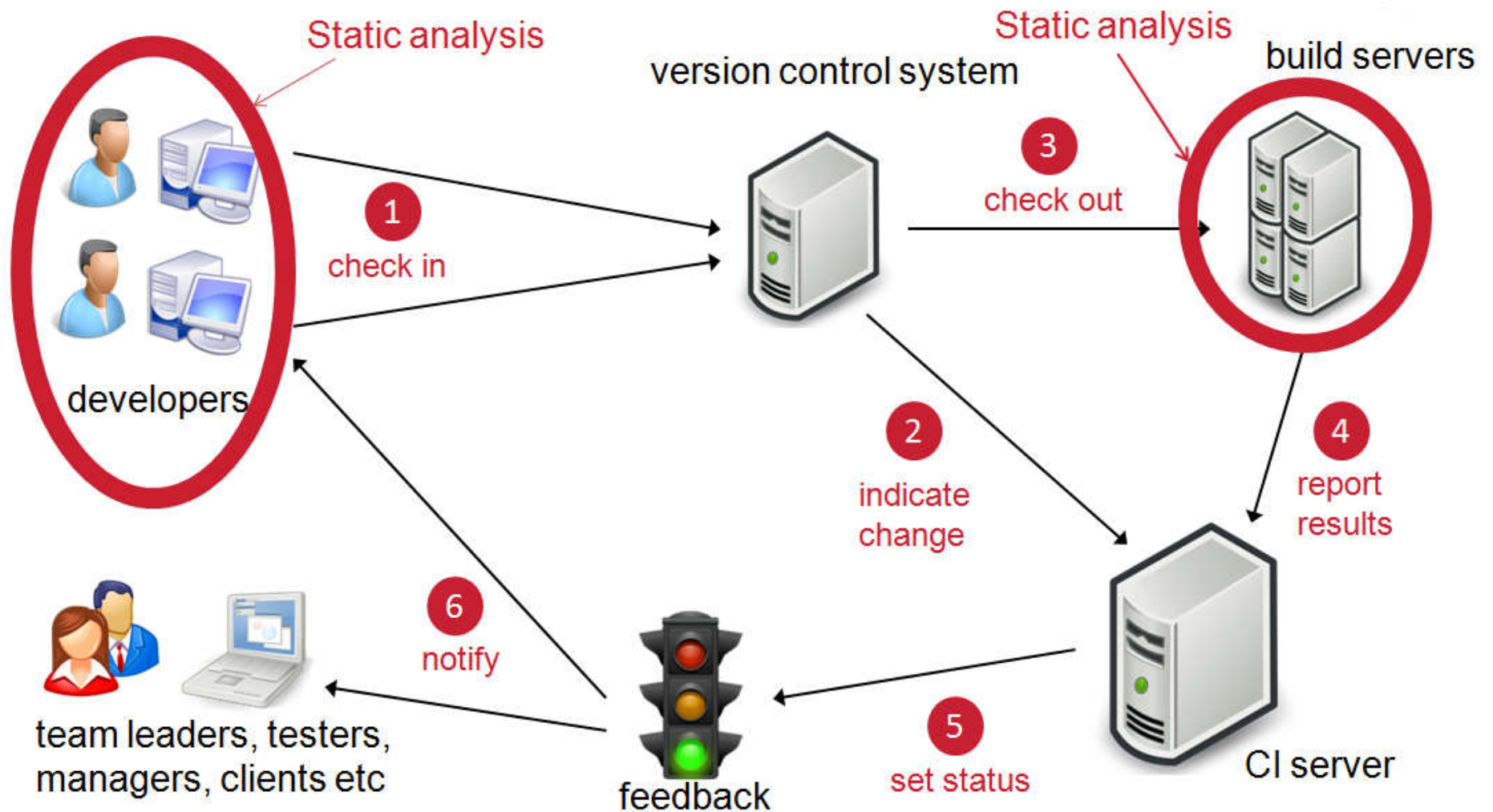
- No functional tests
 - Performance and load tests
 - Load tests
 - Usability tests

Continuous integration

- After Commit
 - Automatic sending for review
 - Automatic build
 - Automatic unit test
- Jenkins
- Gerrit



Continuous integration

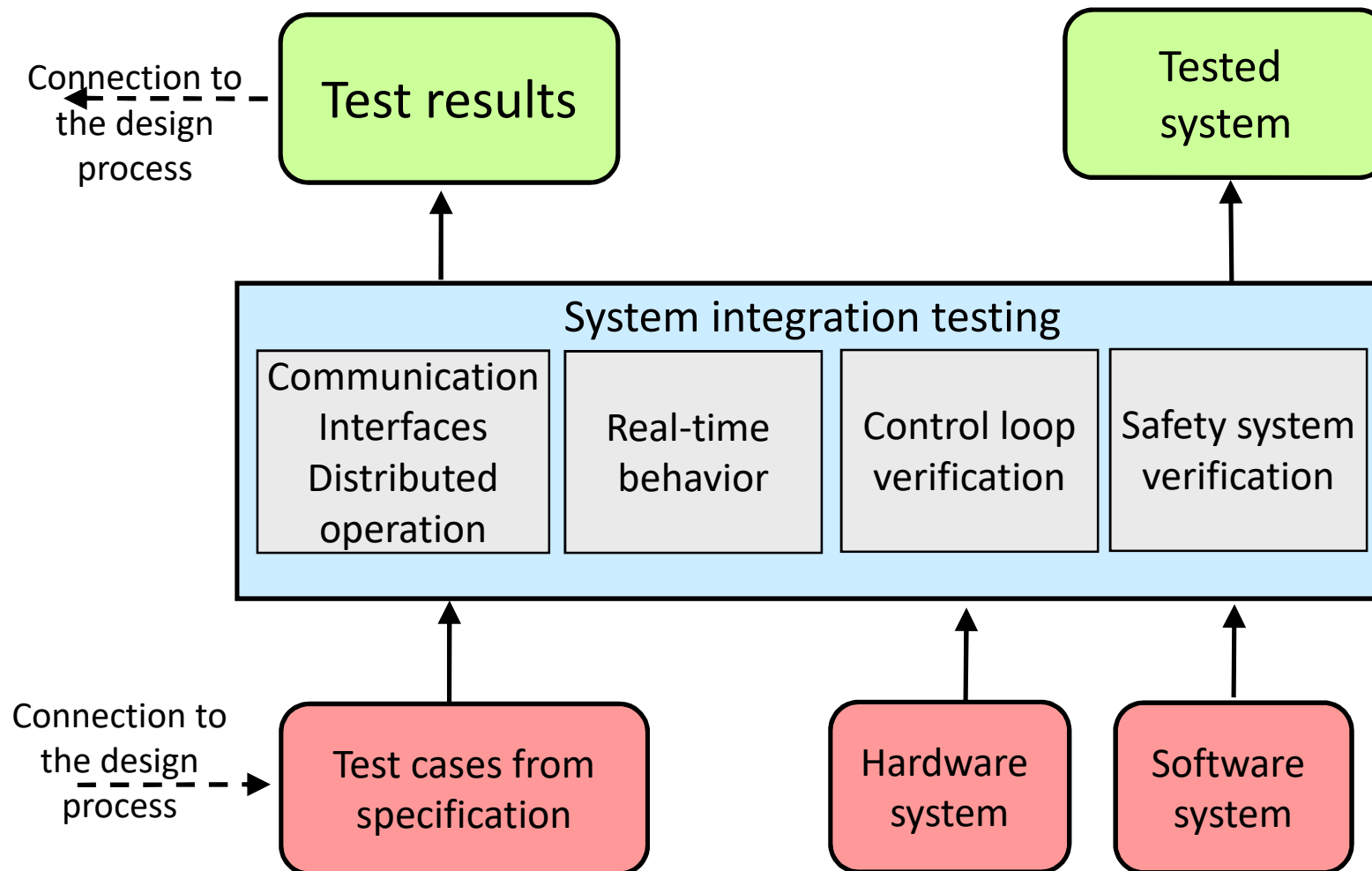


System integration testing

VIMIMA11 Design and integration of embedded systems

Balázs Scherer

System integration testing

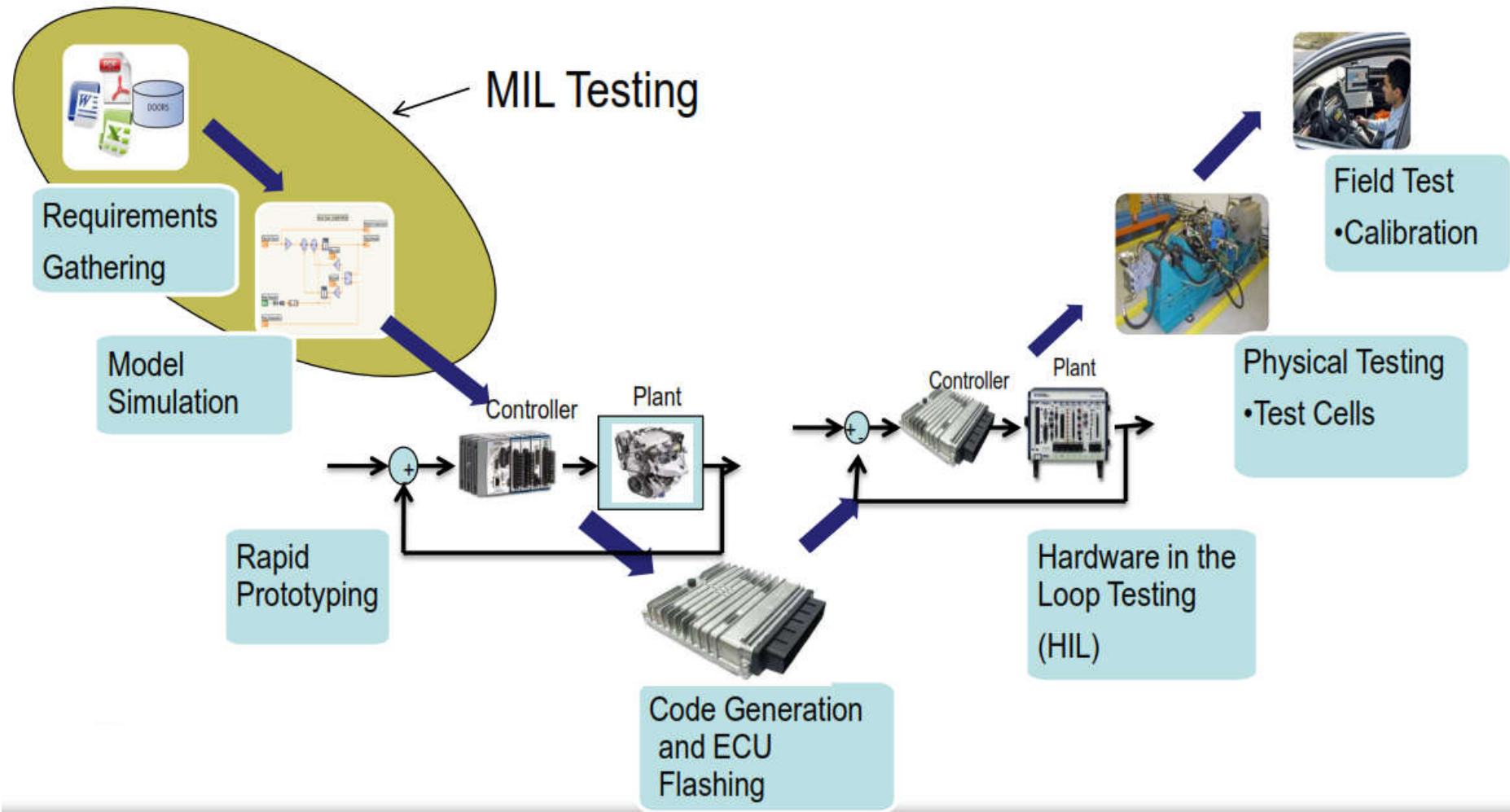


System integration test requirements ISO26262

Methods		ASIL			
		A	B	C	D
1a	Hardware-in-the-loop	+	+	++	++
1b	Electronic control unit network environments ^a	++	++	++	++
1c	Vehicles	++	++	++	++

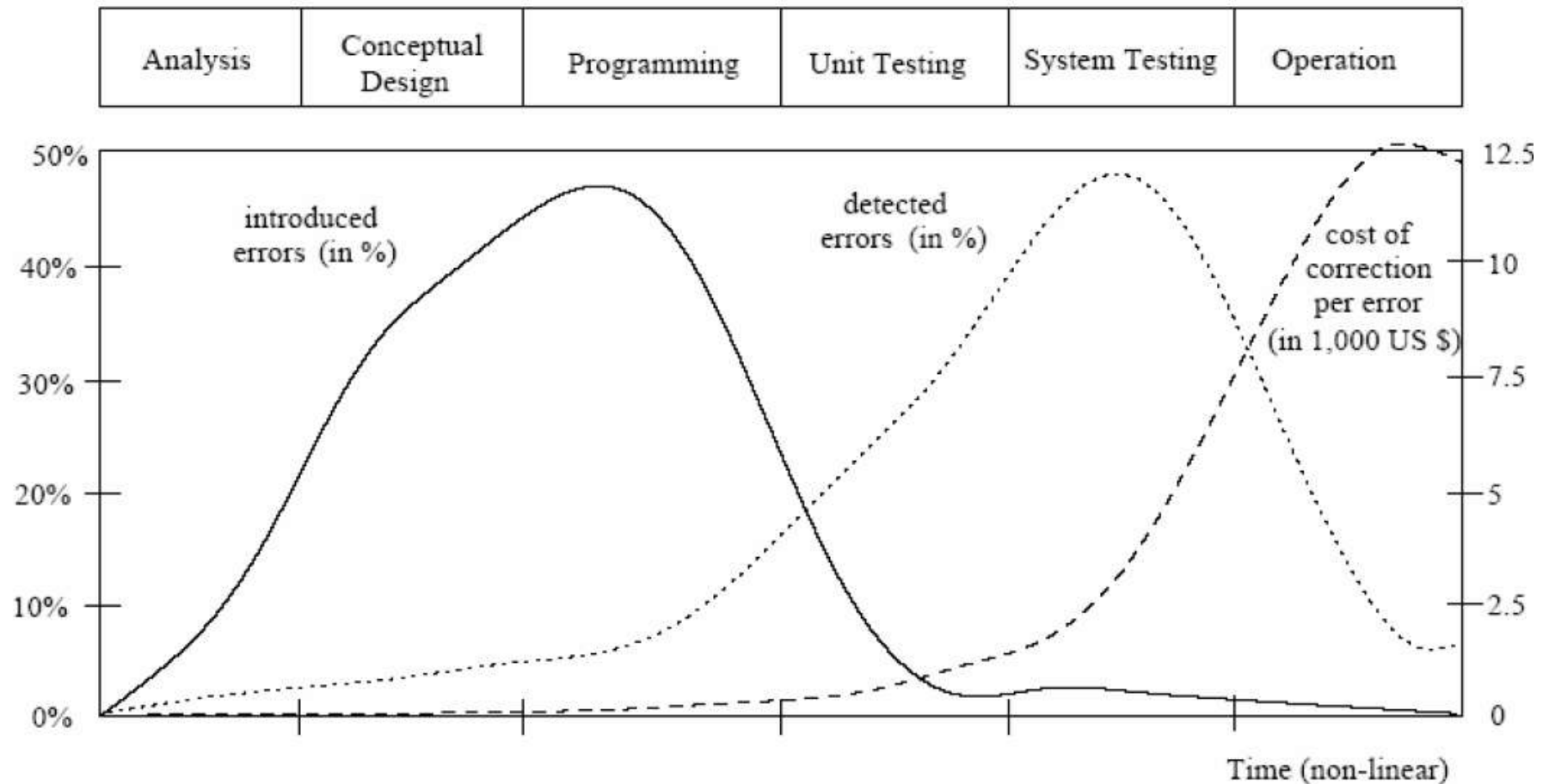
^a Examples are "lab-cars", "rest of the bus" simulations or test benches partially or fully integrating the electrical systems of a vehicle.

MIL, SIL, HIL, Test cells in V-modell



Source ni.com

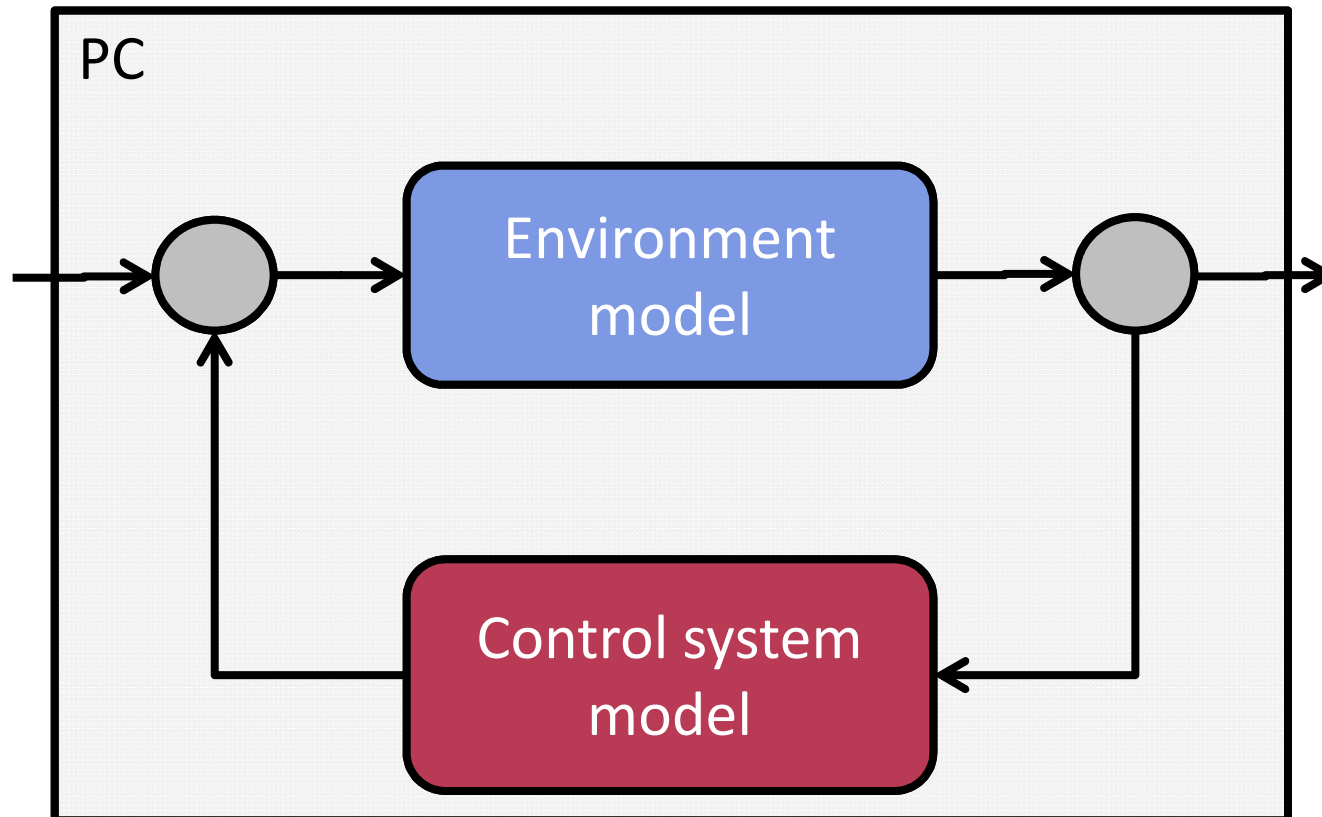
Importance of early testing



From: P. Liggesmeyer et al., Qualitätssicherung Software-basierter technischer Systeme, Informatik Spektrum, 21:249-258, 1998. Quoted after J.P. Katoen, Principles of Model Checking, 2004/5. Copyright © by the authors.

Model-In-the-Loop teszt

- Main functions can be tested very early
 - Simulink, LabVIEW

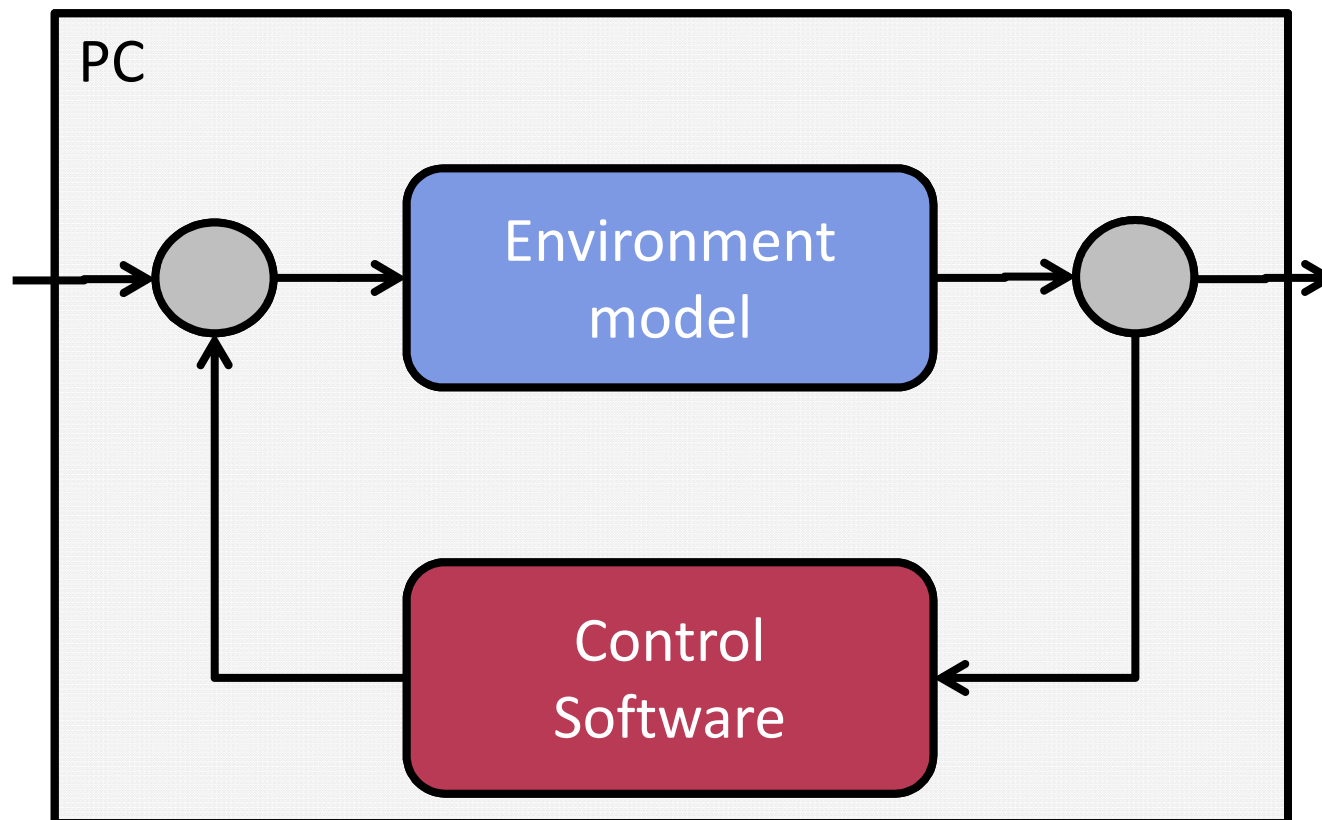


MIL (Model In the Loop) test

- Benefits
 - The main functions are tested very early
 - Domain specific languages are used: Simulink, LabVIEW
 - Very fast testing possibility
- Drawbacks
 - Details can be tested: real-time behavior, architecture dependency

Software-In-the-Loop tests

- Software generated from model
 - Code generation problems, and data precision errors can be found

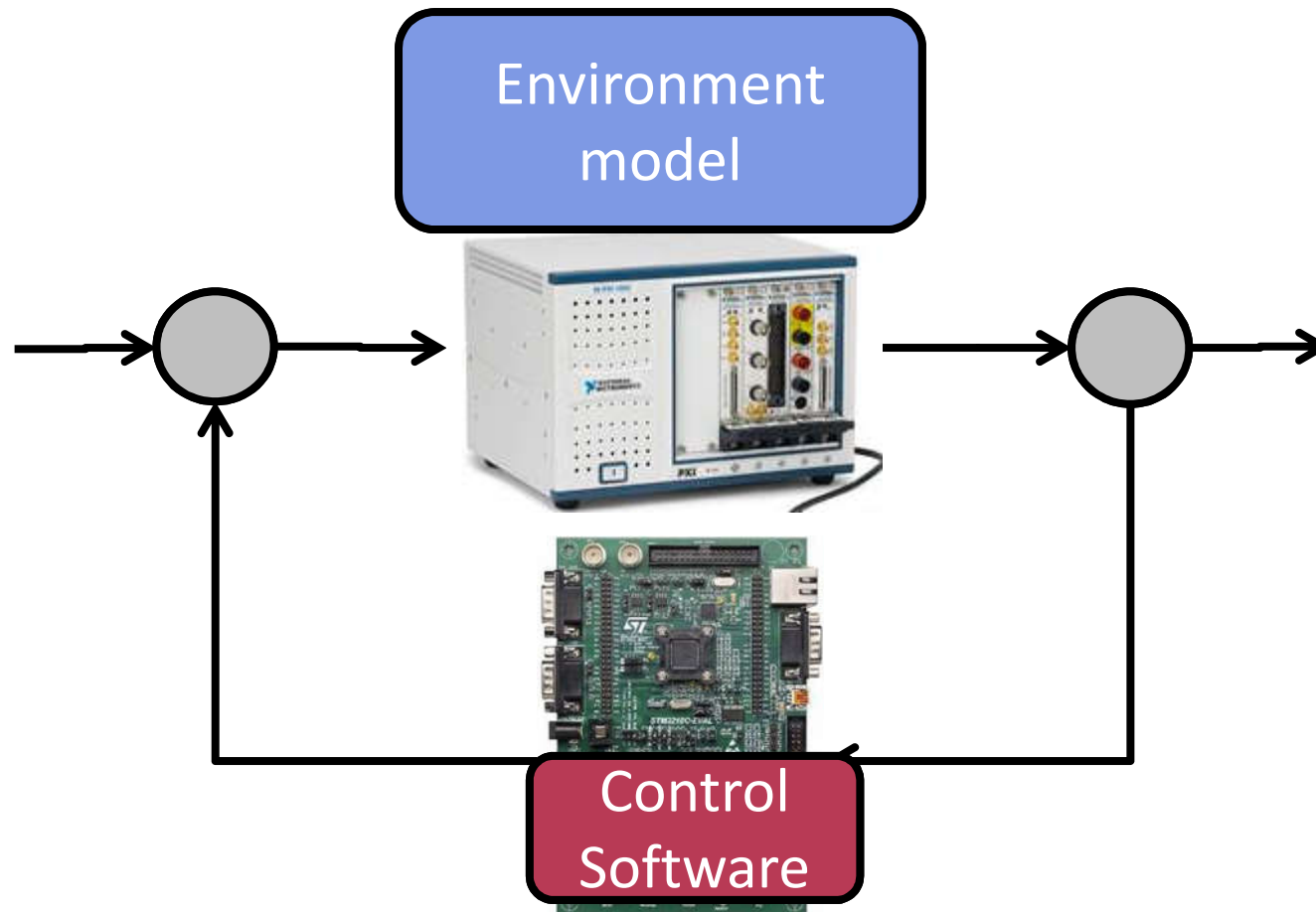


Software-In-the-Loop tests

- Benefits
 - Code generation can be verified
 - Data precision can be verified
- Drawbacks
 - Code do not use the same processor as in the real system:
 - Real-time problems can not be verified
 - Real peripheral problems can not be verified

Processor-In-the-Loop test

- Real processor is used, with real peripherals

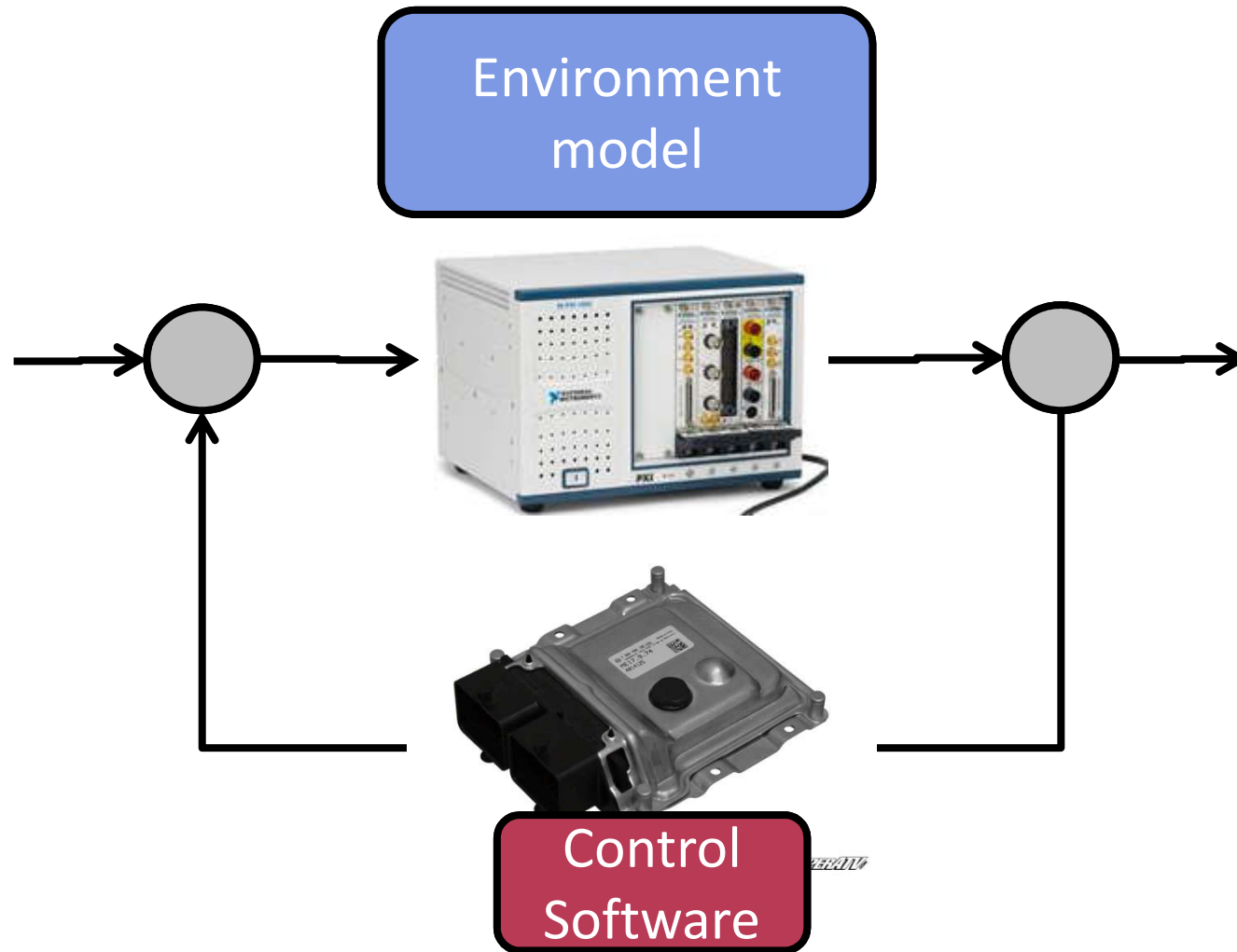


Processor-In-the-Loop test

- Benefits
 - Real-time behavior
 - Real arithmetic, and peripherals
- Drawbacks
 - Many hardware functions are still simulated

Hardware-In-the-Loop test

- Real hardware (ECU) is used



Hardware-In-the-Loop test

- Benefits
 - Real-time behavior
 - Real arithmetic, and peripherals
 - Repeatable tests
 - Safety critical situation can be examined without causing a problem
- Drawbacks
 - The environmental model should be very accurate

Hardware-In-The-Loop test environments



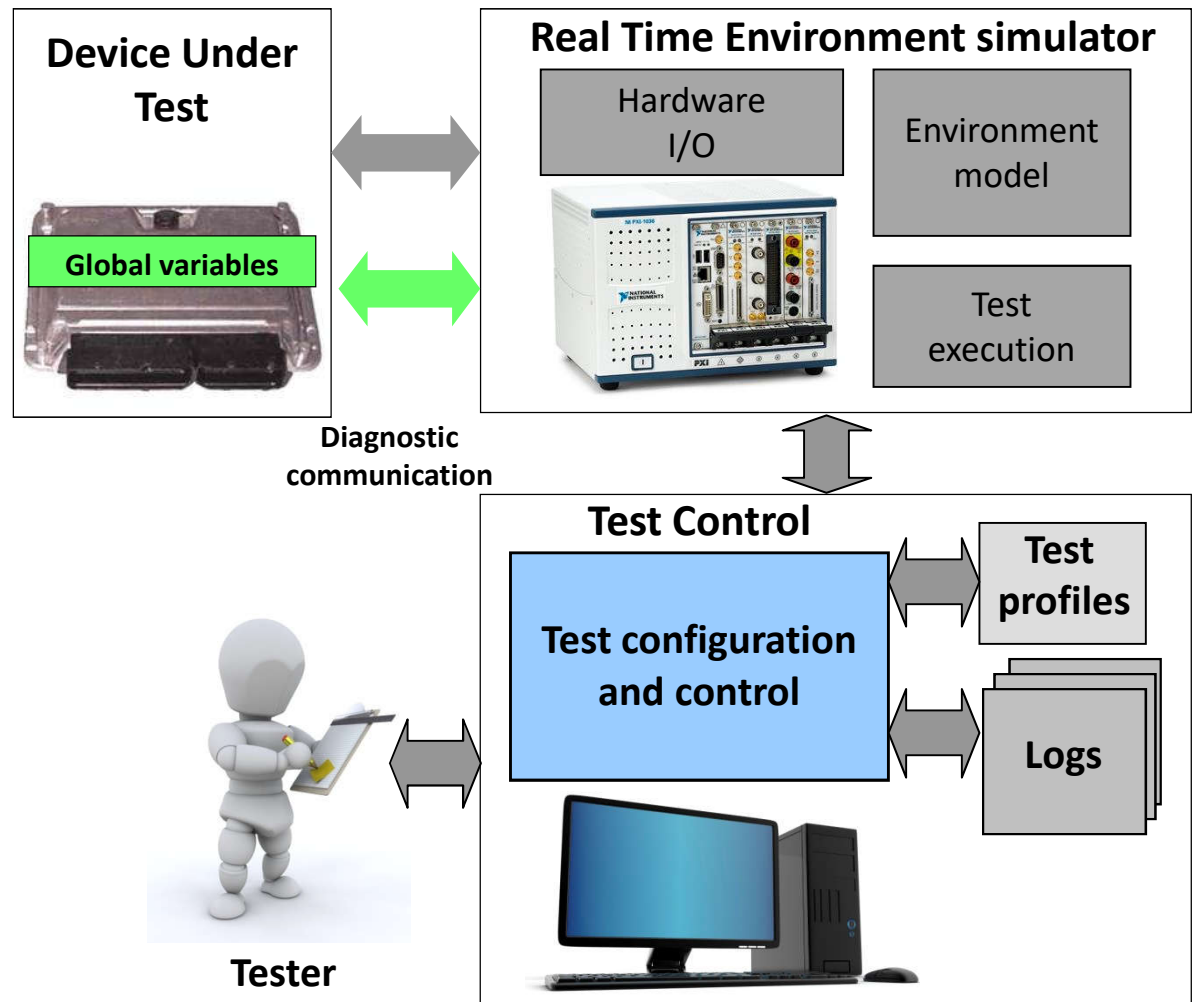
A typical HIL test architecture

■ Interfacing

- Digital I/O
- Analog I/O
- Communication lines

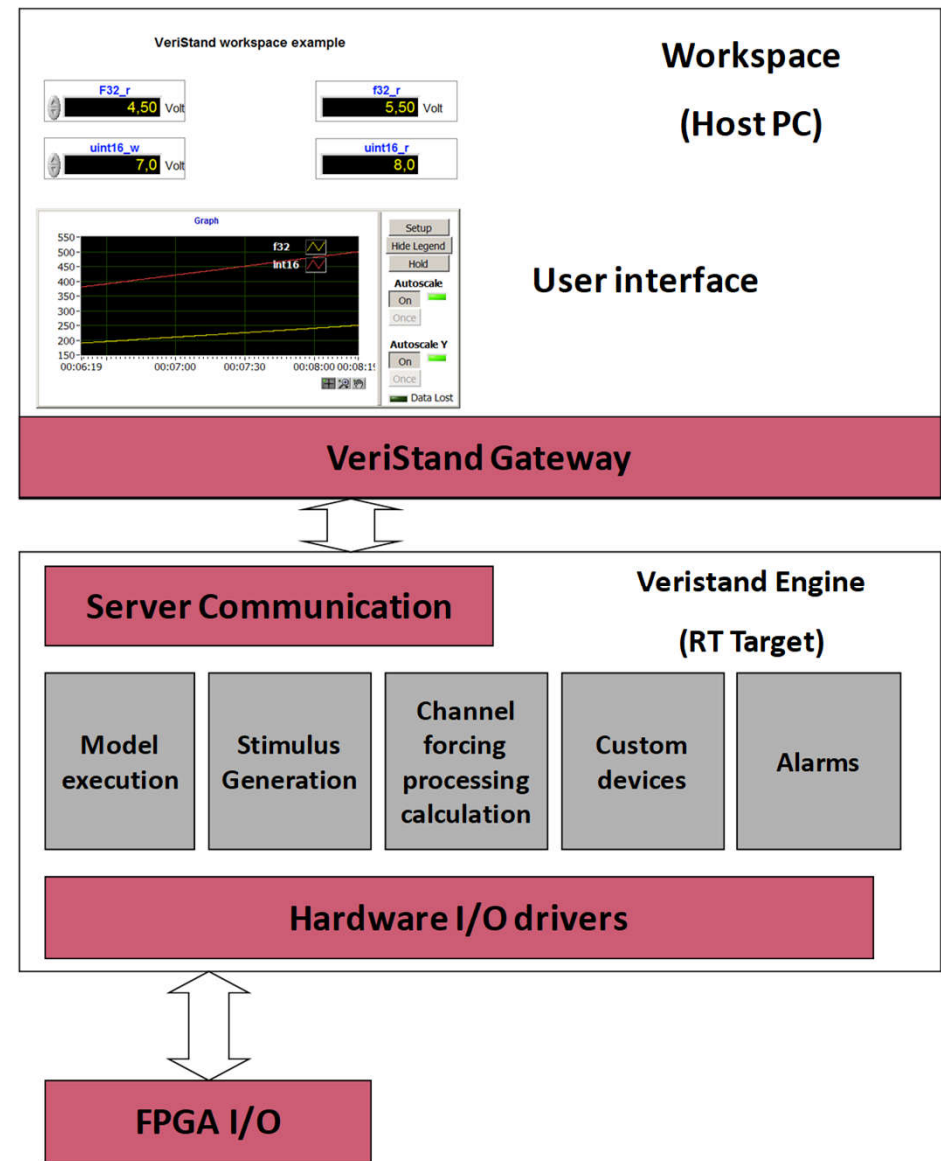
■ Extended information

- Diagnostic communication



NI VeriStand

- Widespread HIL test development environment
- Many supported HW for traditional measurements
- Many possible model representation mode
- Real-Time operation is possible
- *Extendable, with custom functions*



Maturity phase, long term tests

- Long term tests
 - Parallel to many devices: 5-100 pcs. accelerated lifetime simulation tests
 - $N \times 1000$ hours of testing: continuous control and logging is needed
 - Typically Climate chamber, shock pads, shake pads included
 - There is no need for complex environmental simulation

- Requires automated tests

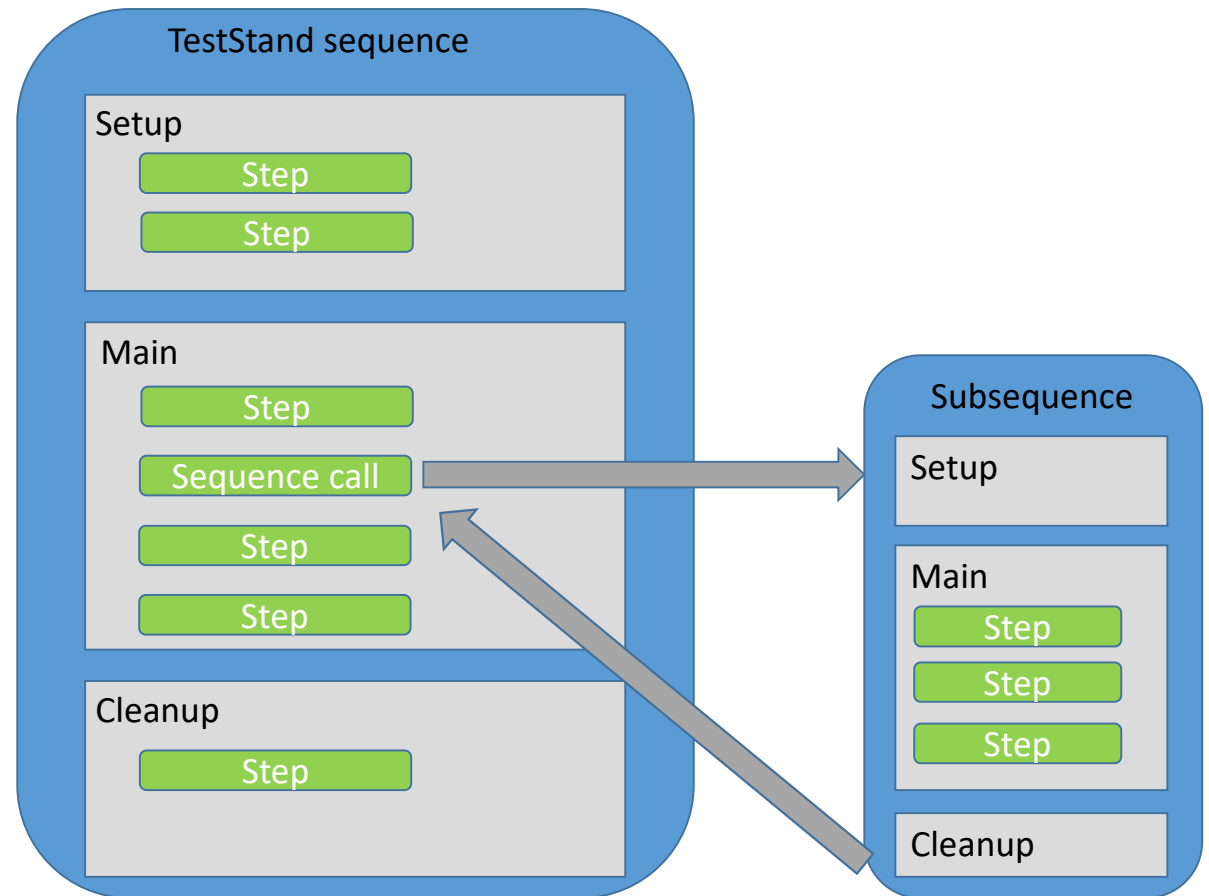
Development environments for test automation

- Able to build a test sequence from test steps
 - Cycles, if, case conditions can be used
- Parameters and Limit values are configurable for test steps
 - Typically read from a file
- Supports parallel testing
 - Typically many devices tested together

NI-TestStand

■ TestStand properties

- *Creating LabVIEW, C#, or Python based test steps*
- *Assigning limits and parameters to test steps*
- *Controlling test sequences*
- *Creating reports after test*



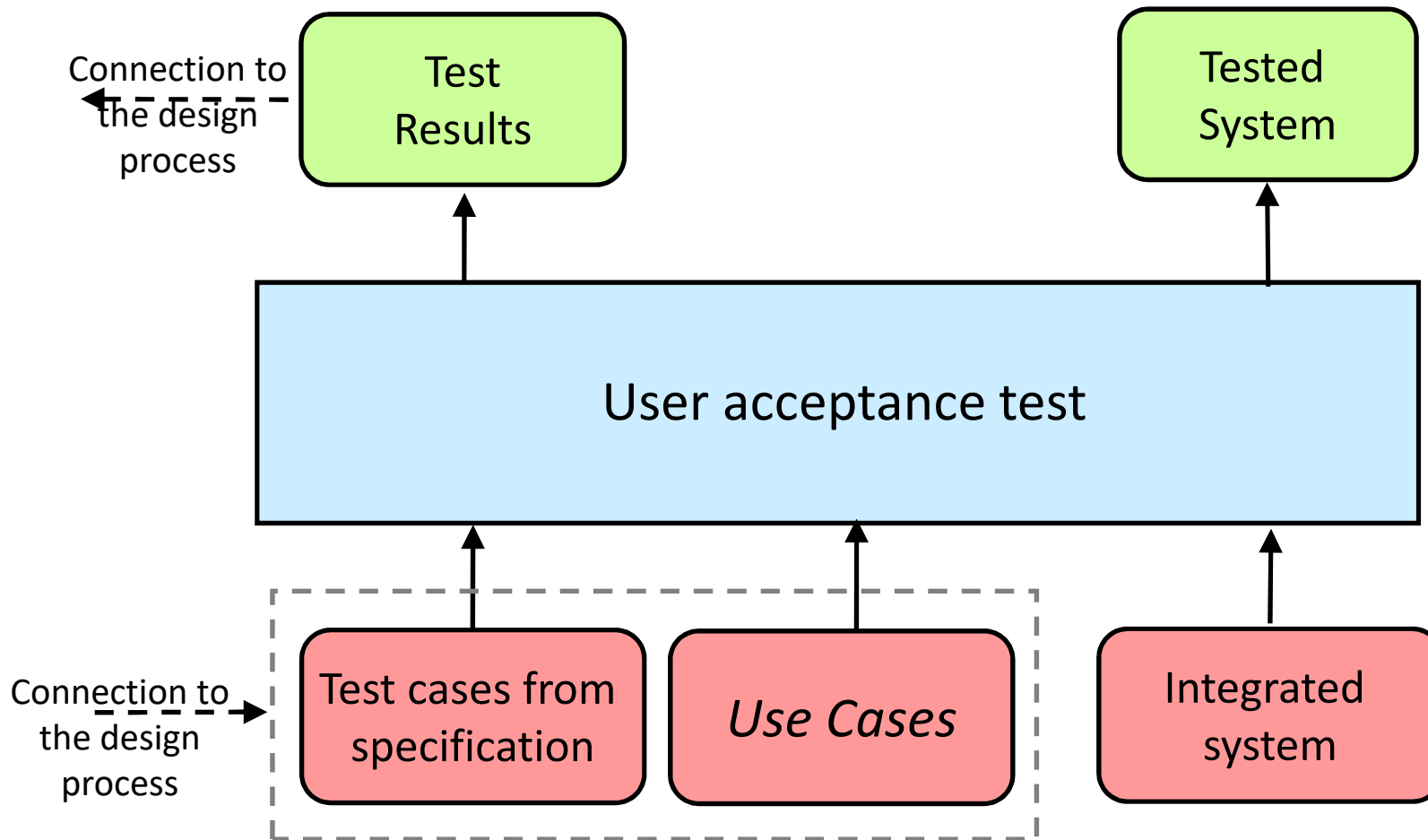
NI-TestStand user interface

The screenshot shows the NI TestStand Sequence Editor interface. The main window is titled "NI TestStand - Sequence Editor [Edit]". The menu bar includes File, Edit, View, Execute, Debug, Configure, Source Control, Tools, Window, and Help. The toolbar contains various icons for file operations and execution. The interface is divided into several panes:

- Insertion Palette:** Located on the left, it contains "Step Types" (Pass/Fail Test, Numeric Limit Test, Multiple Numeric Limit Test, String Value Test, Action, Sequence Call, Statement, Label, Message Popup, Call Executable, Property Loader, FTP Files, Additional Results) and "Flow Control" (If, Else). Below it are "Templates" for Steps, Variables, and Sequences.
- Sequence File 1:** The central pane shows a table with columns "Step", "Description", and "Settings". The table contains "Setup (0)", "Main (0)", and "Cleanup (0)", with a placeholder "<Insert Steps Here>". A red box labeled "1. Sequence editing" highlights this area.
- Sequences:** Located on the right, it shows a list of sequences, with "MainSequence" selected. A red box labeled "2. Sequence selection" highlights this pane.
- Variables:** Also on the right, it shows a list of variables: "Locals ('MainSequence')", "Parameters ('MainSequence')", "FileGlobals ('Sequence File 1')", "StationGlobals", "ThisContext", and "RunState". A red box labeled "4. Variables" highlights this pane.
- Step Settings:** Located at the bottom, it shows "There are no steps selected." A red box labeled "5. Step Settings" highlights this pane.

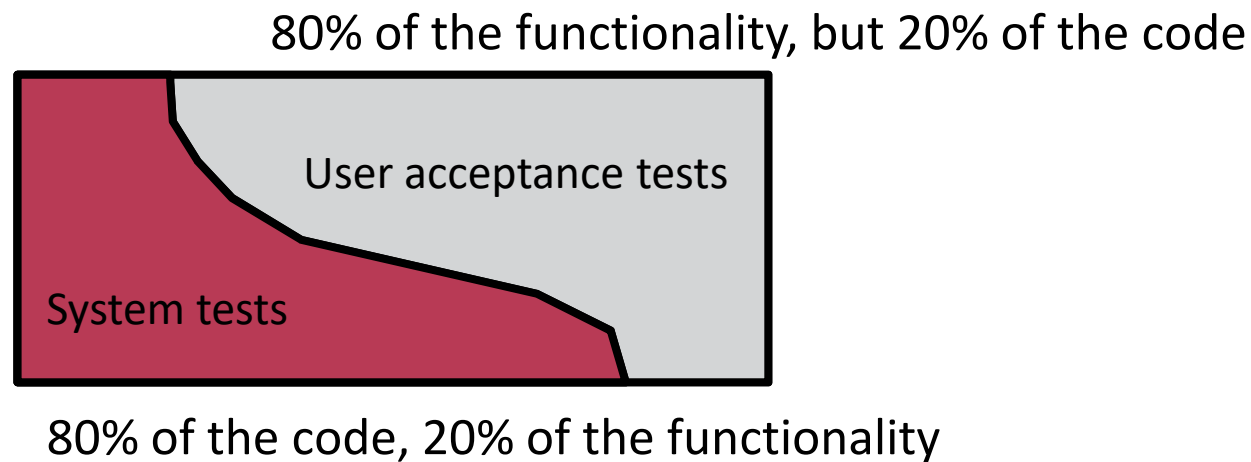
At the bottom of the window, the status bar displays: "User: administrator", "Model: BatchModel.seq", "No Steps Selected", and "Number of Steps: 0".

User acceptance tests



User acceptance testing

- Involving the end-user to the testing
 - The end-user knows what is really needed
 - Different point of view



- Alfa test
 - At the developers site involving the end-user
- Beta test
 - At the real environment

Manufacturing tests

- Very similar to a simplified environment test
 - Executing a test sequence
 - Parallel execution is needed
- Many device need to be tested in a small period of time
 - Verification of the manufacturing not the design
 - Only the main functionality is tested

Manufacturing tests

- Bed of nails tester
 - In-Circuit-Test point in the PCB
- Very simplified environmental tests if any
 - Main functions is tested only
 - Many times special test software is downloaded to the UUT
- Vision based tests
 - Verifying the manufacturing, component placement and soldering

