Modèle et génération automatique de code

Alexandre Chapoutot

ENSTA Paris

2022-2023

Part I

Lecture 5

Software architecture

Software Architecture

From IEEE 1471, architecture of a "software-intensive system":

> *The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.*

Architecture an important first step

- Architecture is the set of **earliest design decisions**
  - hardest to change
  - most critical to get right
- Architecture is **the first artifact design** on which a system's quality attributes are considered.

Quality attributes:

- **From the end user:** performance, availability, usability, security
- **From the developer:** Maintainability, portability, re-usability, testability
- **From the business:** time-to-market, cost-and-benefits, integration with legacy systems, etc.

Design principles and properties

- Divide-and-conquer: Abstraction, Hierarchical structure
- Modularity: coupling and cohesion
- Information hiding
- Limited complexity

Focus on Coupling and cohesion i.e., structural criteria on individual modules and their interactions.

Cohesion the glue that keeps a module together

Coupling the strength of the connection between modules

Cohesion type[1]

- **Coincidental** (worst): is when parts of a module are grouped arbitrarily; the only relationship between the parts is that they have been grouped together (e.g. Utility module).
- **Logical**: is when parts of a module are grouped because they are logically categorized to do the same thing (e.g., mouse an keyboard handlers).
- **Temporal**; is when parts of a module are grouped by when they are processed - the parts are processed at a particular time in program execution (e.g., exception handler).
- **Procedural**: is when parts of a module are grouped because they always follow a certain sequence of execution (file permission checks).
- **Communicational**: is when parts of a module are grouped because they operate on the same data.
- **Sequential**: is when parts of a module are grouped because the output from one part is the input to another part like an assembly line.
- **Functional** (best): is when parts of a module are grouped because they all contribute to a single well-defined task of the module.

[1] `http://en.wikipedia.org/wiki/Cohesion_(computer_science)`

Coupling type[2]

- **Content** (high): one module relies on the internal workings of another module.
- **Common**: two modules share the same global data
- **External**: modules share an externally imposed data format, or communication protocol
- **Control**: one module controls the flow of another, by passing it information on what to do
- **Stamp**: modules share a composite data structure and use only part of it
- **Data:** modules share data through, e.g., through parameters
- **Message** (low): Component communicate via message passing
- **No coupling**: Modules do not communicate at all with one another

[2] http://en.wikipedia.org/wiki/Coupling_(computer_programming)

A good design

Goal

Strong cohesion and weak coupling

Consequences

- Simpler interfaces
- Simpler communication
- Simpler correctness proof
- Modification on a module less impact on other modules
- Re-usability increased
- Comprehensibility increased

Complexity[3]

- Measure a certain feature of the software
- Use these numbers as a criterion to assess or to guide the design
- Higher value then higher complexity then more effort

For example:

- number of lines of code (problem of verbosity of languages)
- Halstead complexity measures: n_1 number of unique operator, n_2 number of unique operands, N_1 total number of operators, N_2 total number of operands.
  - Size of vocabulary: $n = n_1 + n_2$
  - Program length: $N = N_1 + N_2$
  - Volume: $V = N \log_2 n$
  - Programming effort: $E = V/L$, etc.
- Cyclomatic complexity (from the CFG) $M = E - N + 2P$
  - E: number of edges
  - N: number of nodes
  - P: number of connected components

[3] `http://en.wikipedia.org/wiki/Halstead_complexity_measures`
`http://en.wikipedia.org/wiki/Cyclomatic_complexity`

Conclusion

Recipe

- **Essence of the design process**: decompose system into parts
- **Wanted properties of a decomposition**:
  - coupling/cohesion,
  - information hiding,
  - (layers of) abstraction
- **Complexity of a decomposition:** these properties are expressed with numbers

Several design methods

- Theses ideas are still valid for model-based design.
- But some notions, *e.g.*, complexity, have to be refine/clarify in the context of models.

Software testing

Computer-aided design

Introduction to software testing

Software testing

Definition (tentative)

Detection of unwanted behaviors with program execution process in order to validate the software.

Mainly two kinds of test activities:

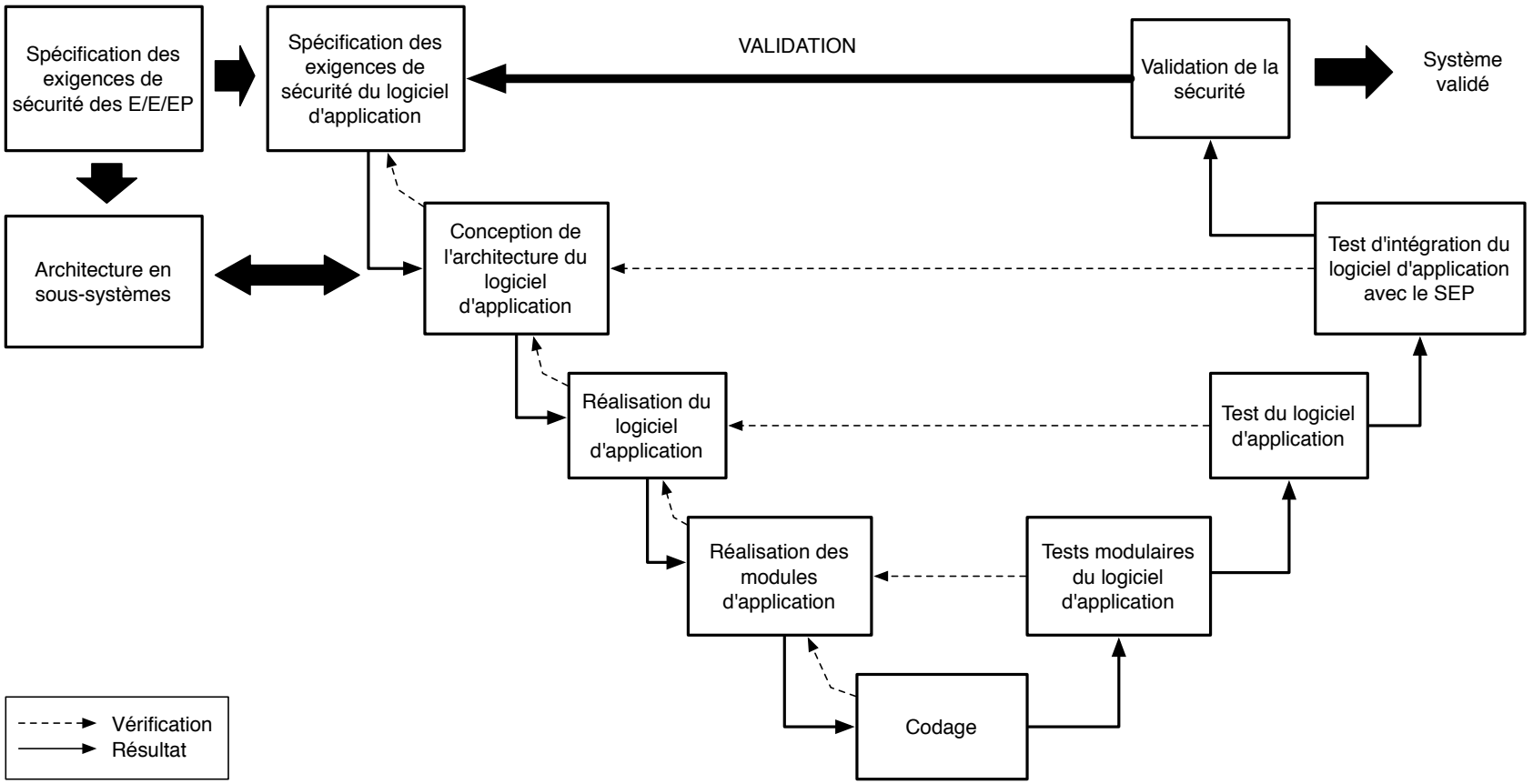
- **Structural testing:** To detect implementation errors.
- **Functional testing:** Is the software fulfill its specification?

Best way: combine both

Other verification methods:

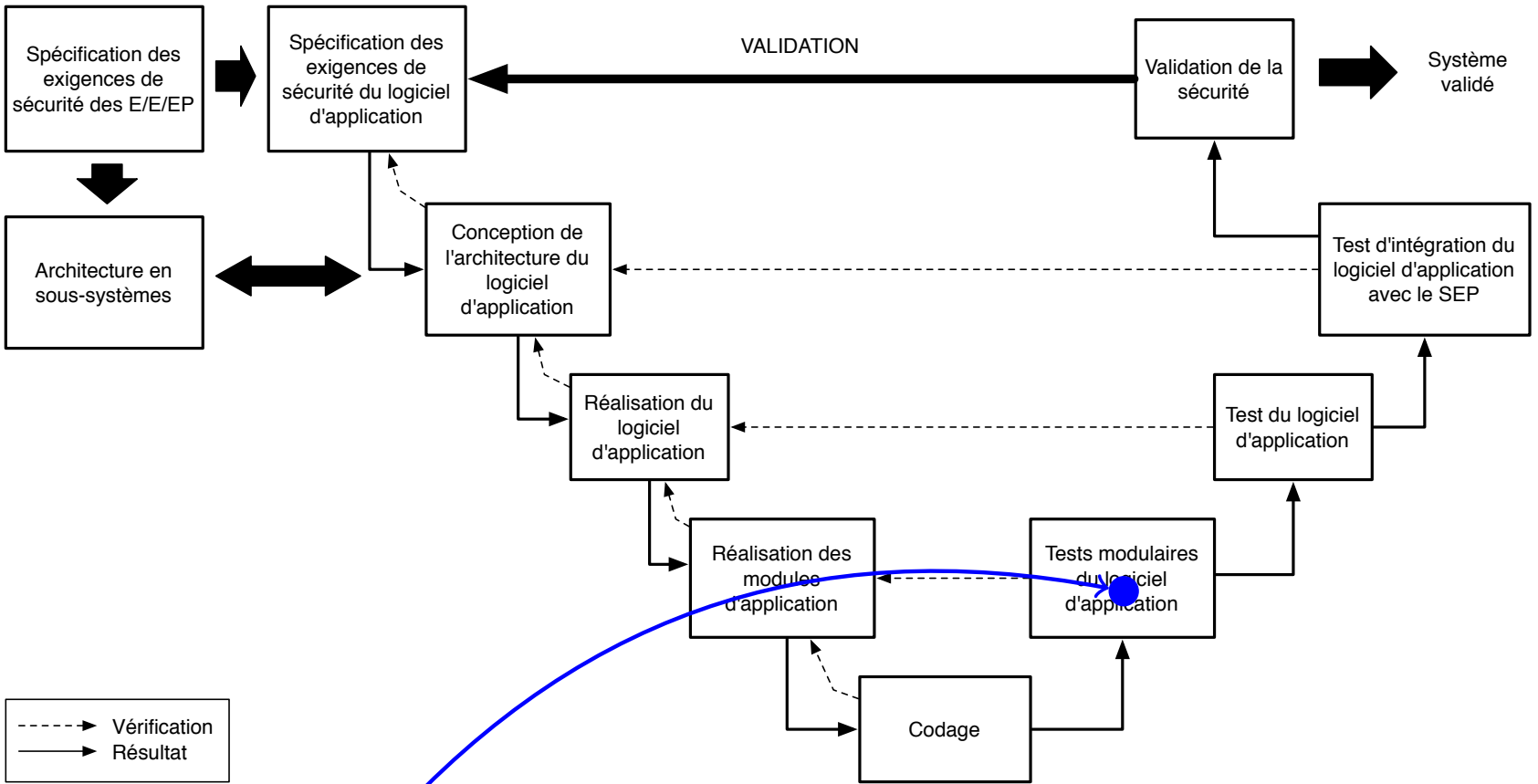
- **Code inspection**: may detect static errors but not dynamic ones and does not scale up well.
- **Formal methods**: gives mathematical arguments to validate the software. (Note: model and executable are not the same)

Testing activities in the cycle of development



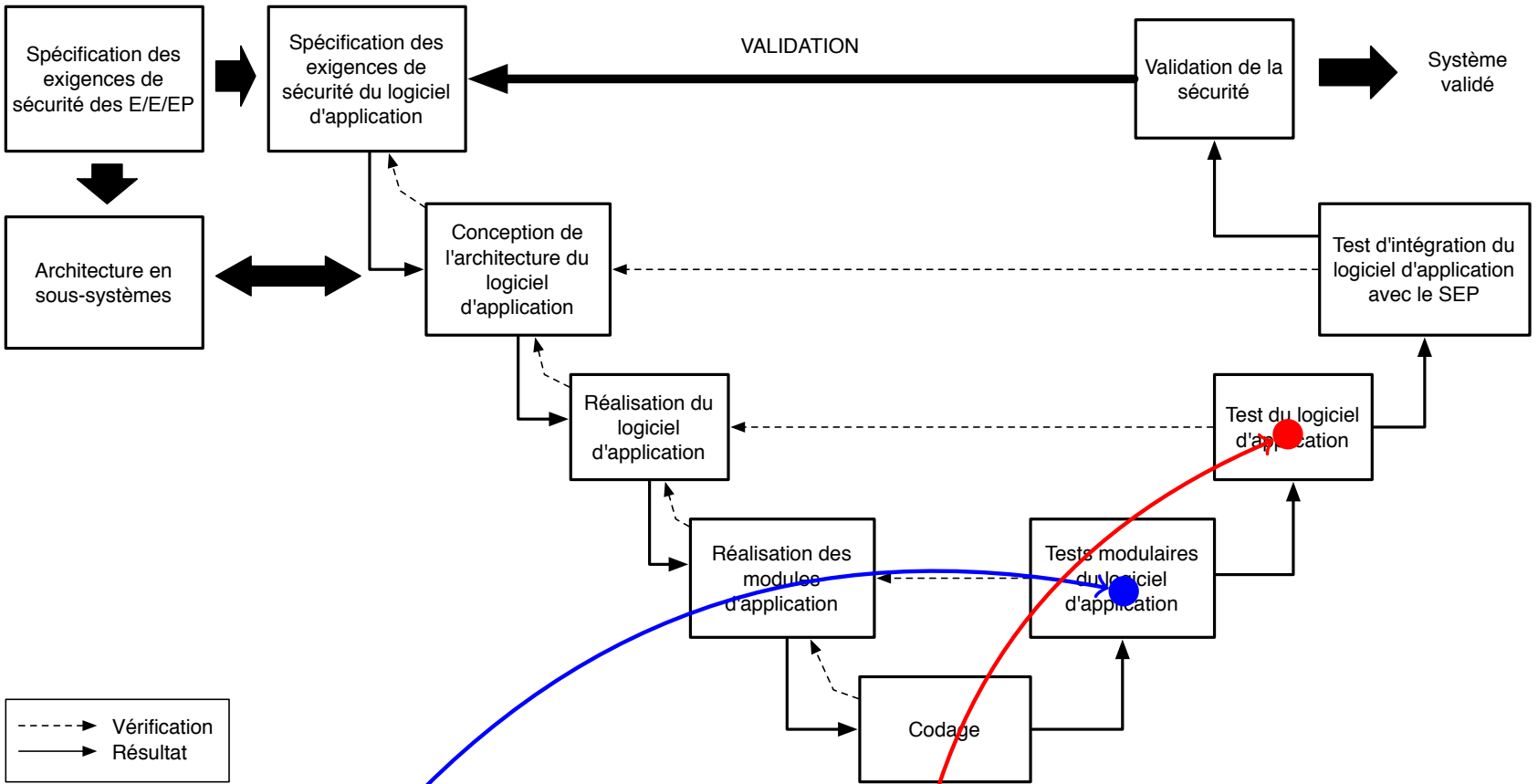
- Unit testing .
- Integration testing software/software: .
- Integration testing software/hardware: .

Testing activities in the cycle of development



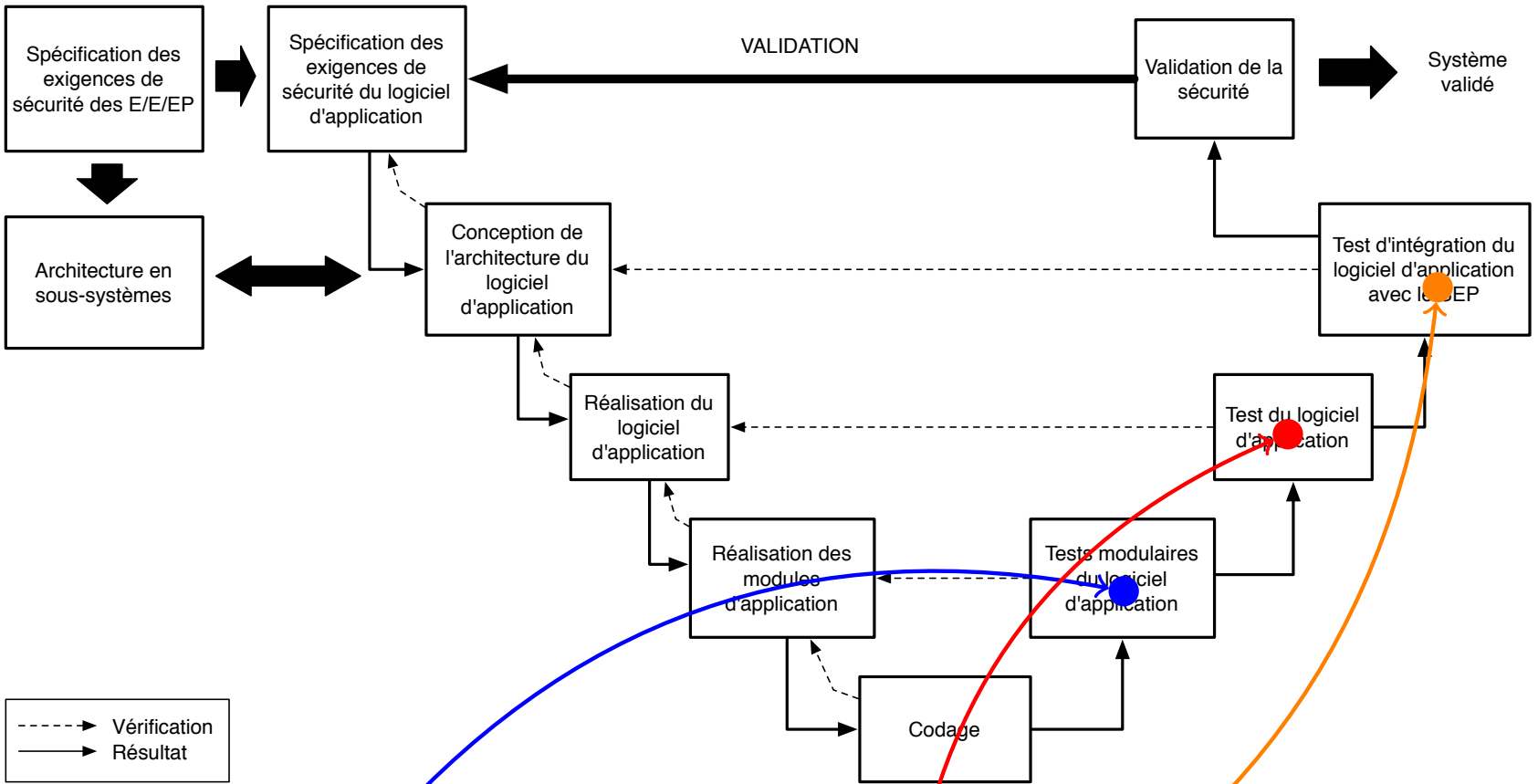
- Unit testing :
- Integration testing software/software: .
- Integration testing software/hardware: .

Testing activities in the cycle of development



- Unit testing :
- Integration testing software/software: .
- Integration testing software/hardware: .

Testing activities in the cycle of development



- Unit testing :
- Integration testing software/software: .
- Integration testing software/hardware: .

Test activity and the development levels

Unit testing: separately testing each software component.

Integration testing: testing the good behavior when components are composed.

Conformance testing: validation of the adequacy between code and specification.

But we can also find: **regression testing:** verify that software corrections/updates do not introduce errors; **target testing:** verify that the code on the target is correct ($\neq$ **host testing**).

Mainly two approaches:

- **Black box:** (i.e., functional testing) testing activity from the specifications, a model, function prototypes or component interfaces.
- **White box:** (i.e., structural testing) testing activity from the source code.

Software testing challenges

The main difficulty in software testing come from the definition of:

- a set of inputs in function of a **test coverage criterion**.
- a set of outputs in function of a determined set of inputs (**oracle** problem).

and the definition of **stubs**.

Importance of test cases:

- the **test cases** is a representation of particular values of the input defined in the component specification.
- A complete software testing $\Rightarrow$ full specification coverage.
  but combinatorial explosion:
  e.g. addition of two 32 bits integers requires 2^{64} test cases!

Testing coverage

Goal of the coverage

- For the client: defined the level of trust in the software.
- For the verifier: defined the contract between the client and the tester
- For the tester: defined the measure criterion.

Problem: stop criterion: When should we stop testing?

- Negative criterion: blocking bugs, we cannot test the remainder of the software.
- Positive criterion: rate of coverage
  ideally 100% but may require "justifications"

Test cases and coverage

Each test case must increase the testing coverage.

Computer-aided design

Introduction to software testing

Structural testing

Test structural

Software testing activity based on the software structure.

Goals

- Detect implementation errors;
- Verify that the software does not do more than its specification;
- Verify that there is no bugs like:
  e.g. overflow, non initialization variable, etc.

Stopping criterion

- in function of the code structure.

Structural testing coverage

Definition

Measure the rate of tested source code.

One sees these coverages

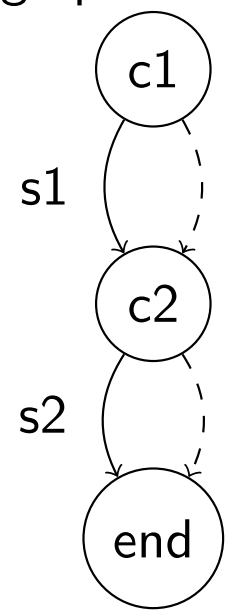
- **Control flow coverage:** instructions, branches, paths, LCSAJ
- **Logical condition coverage**

Example: instruction coverage

Source code

```
procedure P
   (in Boolean C1;
    in Boolean C2)
is
begin
    if C1 then
       s1;
    end if;
    if C2 then
       s2;
    end if;
end P;
```

Control flow graph



Test1

c1 = true
s1

c2 = true
s2

Remark

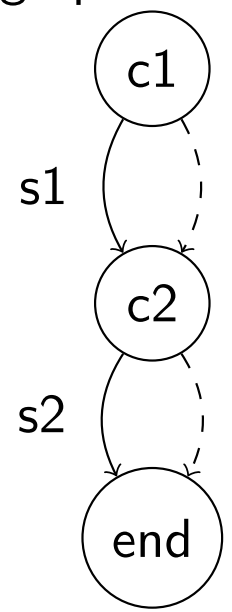
Only one test is enough.

Example: branch coverage

Source code

```
procedure P
   (in Boolean C1;
    in Boolean C2)
is
begin
    if C1 then
       s1;
    end if;
    if C2 then
       s2;
    end if;
end P;
```

Control flow graph



Test 1	Test 2
c1 = true	false
s1	
c2 = false	true
	s2

Remark

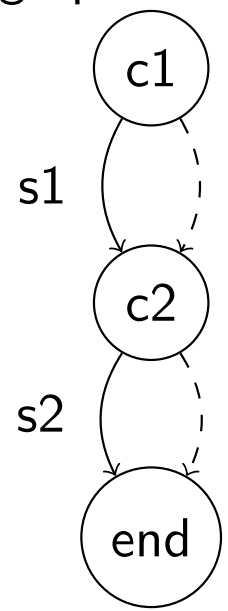
Two tests are required.

Example: path coverage

Source code

```
procedure P
   (in Boolean C1;
    in Boolean C2)
is
begin
    if C1 then
       s1;
    end if;
    if C2 then
       s2;
    end if;
end P;
```

Control flow graph



Test 1	Test 2	Test 3	Test 4
c1 = true	true	false	false
s1	s1		
c2 = true	false	true	false
s2		s2	

Remark

Four tests are required.

Question what about loops, is this coverage feasible?

Linear Code Subpath And Jump (LCSAJ).

The idea is to split the source code into linear parts in order to have paths coverage.

A piece of vocabulary

In a control flow graph, we can distinguish:

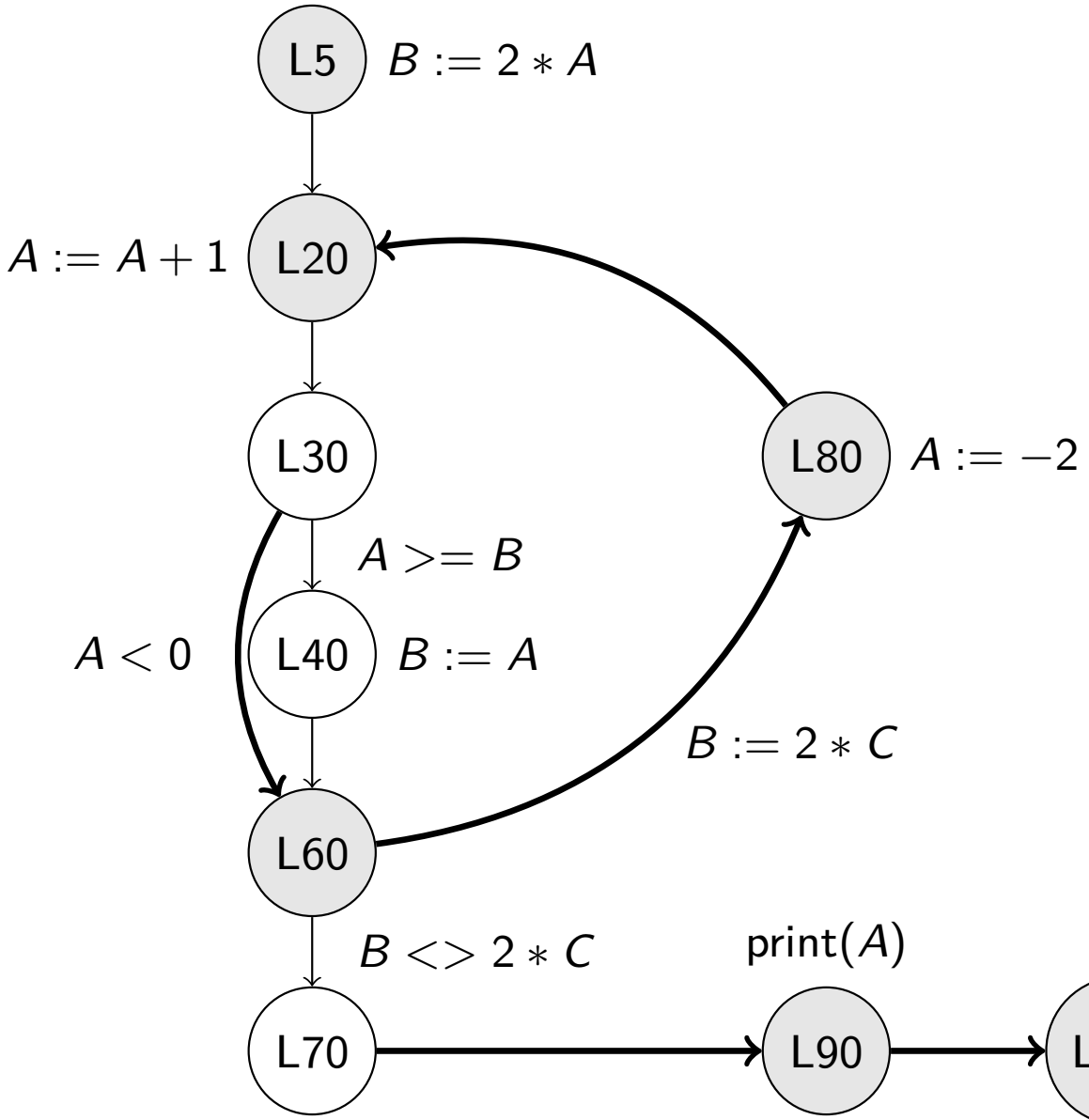
- Special nodes: input, output or the target of a jump.
- Special edges: jumps and by default the edge to the output.

Definition

We call LCSAJ a path going from a special node to an other special node containing only one special edge between the last node and the node before.

- **Pros:** a better coverage than branches coverage without being to expensive.

Example: LCASJ paths



LCASJ:

- L5, L20, L30, L60
- L5, L20, L30, L40, L60, L80
- L5, L20, L30, L40, L60, L70, L90
- etc.

Limitation of control flow coverage

Two programs doing the same computation:

```
procedure P1
   (in Boolean C1;
    in Boolean C2)
is
begin
    S := False;
    if C1 then
       if C2 then
          S := True;
       end if;
    end if;
end P1;
```

```
procedure P2
   (in Boolean C1;
    in Boolean C2)
is
begin
    S := C1 and C2;
end P2;
```

Paths coverage: 3 tests for P1 and 1 test for P2.
The previous coverage are too dependent on the control flow.

Partial conclusion one never uses the source code itself to test the software but the specifications!!!

Logical conditions coverage

In control-flow coverage, we do not take into account the values of nodes.

Principle

In the logical condition coverage, instead of going once in the "true" branch and in the "false" branch, we seek the different way to make condition "true" or "false".

Consequence

We increase the confidence in the software.

Logical condition coverage: truth table

```
procedure P
  (in Boolean C1; in Boolean C2; in Boolean C3)
is
begin
    if C1 or C2 or C3
    then
        s1;
    else
        s2;
    end if;
end P;
```

Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8
(v,v,v)	(v,v,f)	(v,f,v)	(v,f,f)	(f,v,v)	(f,v,f)	(f,f,v)	(f,f,f)
s1	s1	s1	s1	s1	s1	s1	
							s2

Also called: **multiple condition** coverage 2^n

Strategies to limit the combinatorial explosion

- **Strategy for OR operator**, do
  - one test with all the sub-conditions set to "false".
  - one test for each sub-conditions set to "true" one by one.
  - one test with all sub-conditions set to "true".

  Consequence: $n + 2$ tests
- **Strategy for AND operator**, do
  - one test with all sub-conditions set to "true".
  - one test for each sub-conditions set to "false" one by one.

  Consequence: $n + 1$ tests

Called also MC/DC (Modified Condition/ Decision Coverage)

- Decision: same as branches coverage.
- Condition: each condition must be true or false.
- Condition/Decision: combination of the two previous criteria.
- MC/DC: independence of each conditions.

Computer-aided design

Introduction to software testing

Functional testing

Functional testing

also called black box testing.

Goal

- Check the software behaviors in regards to the specification.
- Check that some constraints are fulfilled (*e.g.* performance, memory consumption, etc.) or quality factors are met (*e.g.* portability, maintainability, etc.)

Stopping criteria the functional coverage is qualitative
We cannot know a priori the number of tests to do.
We have to use the only one thing we know: the specification.

Remark usually we use structural coverage to help in this task!

Recall on specifications

A specification must describe **at least**:

- the software functions;
- the software interface;
- the development constraints.

Kinds of coverage

1. Nominal testing;
2. Functional limit testing;
3. Robustness testing;
4. Conformance testing.

1 and 2 test the function of the software.
3 and 4 are quality factors.

Software function testing

To test the functional behaviors

Two cases:

- **Nominal testing** check the conformance to the specification for a normal behavior of the software.
- **Limit testing:** check the behaviors at the functional limits of the software.

Building test cases with partition analysis

Idea of the method

group together each input into equivalent classes, *i.e.* inputs which produce the same functional behaviors.

Partition analysis

Recipe: for each function in the specification

- Determine inputs and their domains of values.
- From the control part of the specification split the input domain into equivalent classes.
- For each equivalence classes:
  - choose one element in it
  - from the process part of the specification determine the output value associated to the chosen inputs.

Oracle problem

- when algorithms are too complex, *e.g.*, controller
- all the inputs are not available for the tester (*e.g.*, clocks)

Limit and outside limit tests

Example

- **Function:** Produit_valeurs_absolues
- **Input:** E1 and E2
- **Output:** S
- **Processing:** this function compute the absolute value of the product of E1 and E2.

Equivalence classes:

$E1$	$E2$
$[MinInt, -1]$	$[MinInt, -1]$
$[0, MaxInt]$	$[0, MaxInt]$

Definition: Limit tests

Functional limit test: taking values at the limits of each functional equivalence classes.

Definition: outside limit tests

Functional outside limit test: taking values outside the limits of functional equivalence classes.

Example: limit and outside limit tests

If E1 and E2 has the same functional domain: $[-100, 100]$

Limit tests

E1		E2	
$[-100, -1]$	-100	$[-100, -1]$	-57
$[-100, -1]$	-1	$[0, 100]$	64
$[0, 100]$	0	$[-100, -1]$	-5
$[0, 100]$	100	$[0, 100]$	98
$[-100, -1]$	-59	$[-100, -1]$	-1
$[0, 100]$	48	$[-100, -1]$	-100
$[-100, -1]$	-63	$[0, 100]$	0
$[0, 100]$	75	$[0, 100]$	100

Outside limit tests

E1		E2	
$[-100, -1]$	-234	$[-100, -1]$	-42
$[-100, -1]$	-84	$[0, 100]$	115
$[0, 100]$	32	$[-100, -1]$	-567
$[0, 100]$	174	$[0, 100]$	39

Data coverage

Remark

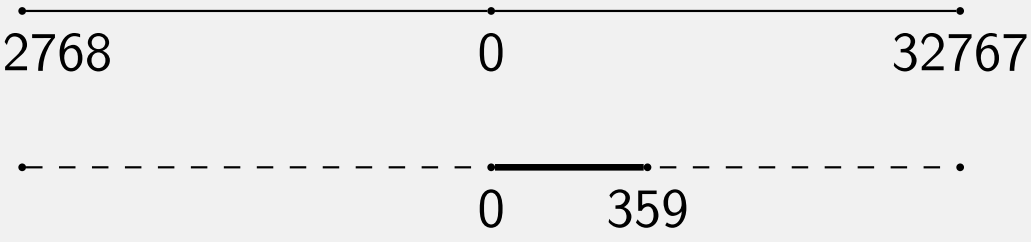
In the previous coverages, we did not interest in manipulated data format nor numerical values to make condition to true or false.

The data coverage tries to choose:

- the right numerical values to make a condition true or false. (**remarkable value**)
- the limit values of the input.
- the numerical values to get particular values or behaviors (overflow, division by zero, etc.)

Example: data coverage

Domain of a data




```
procedure P
   (in Integer E)
is
begin
    if E > 30
    then
        S1;
    end if;
end P;
```

Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
359	31	30	0	−32768	32767
s1	s1				s1

Data coverage

It is possible to split the domain of each data in **equivalence classes**.
Each class defines a particular behaviors of the data.
In the previous example, two classes are possible $[-32768, 29]$ and $[30, 32767]$. Note: 30 is a remarkable value.

From the tester point of view, we need two kinds of tests:

- **Limit testing:** taking values at the limits of each functional equivalence classes.

- **Outside limit testing:** taking values outside the limits of functional equivalence classes.

A word on stubs

When a component uses an external function `f` and we have or not this function, in a first step we wish to test the behavior of the component independently of `f`.

The **stubbing** technique aims at:

- add one artificial input in the test cases. This input is associated to the output of `f`.
- write a **stub** to replace the code of `f`: this code only returns the value set by the test case.

Remark

With this technique the tester can fully control the tests to do.

Conclusion: software testing

> *Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?*
>
> *Brian Kernighan*

Main industrial challenge:

- The verification step is highly important: economic issues.
- But testing activity a comlex task starting in the specification step of the cycle of development.

Efficient way to do testing

Combine structural and functional testing:

- define the test cases from the speficiation
- compute the covergare from these test cases

Model testing

1. Software architecture
2. Software testing
3. Model testing

Computer-aided design

Introduction to model testing

Context

Model vs Software

- Software: statements, loop, conditional
- Model: State/Transition or equations

Coverage criteria on software do not applied on models!

Model-based design

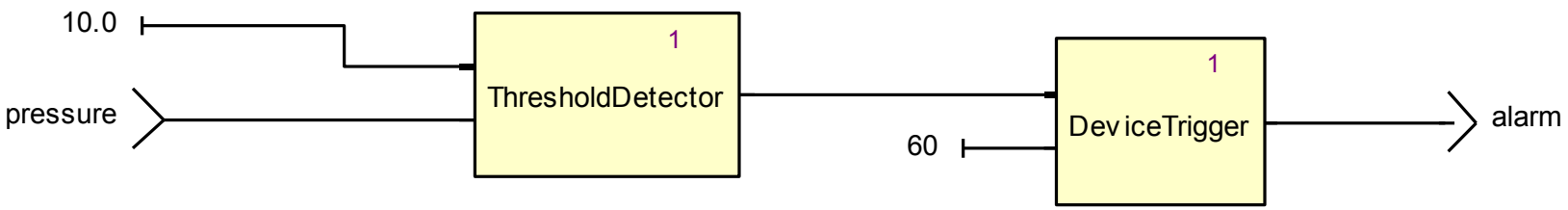
Fact: source code automatically generated from models.

Need to apply testing coverage on models

SOMCA: safety implication in performing **SO**ftware **M**odel **C**overage **A**nalysis

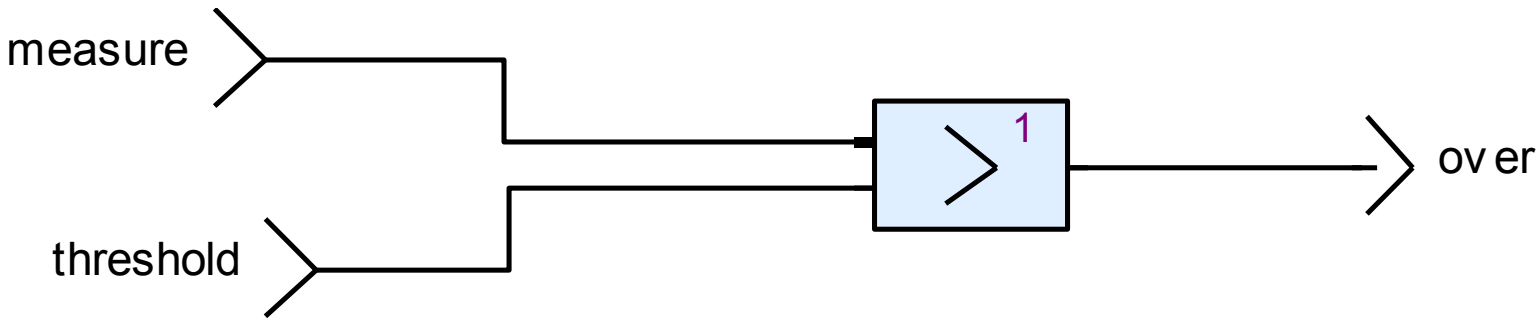
Example: Model Coverage

A small program: a pressure detector
Main operator



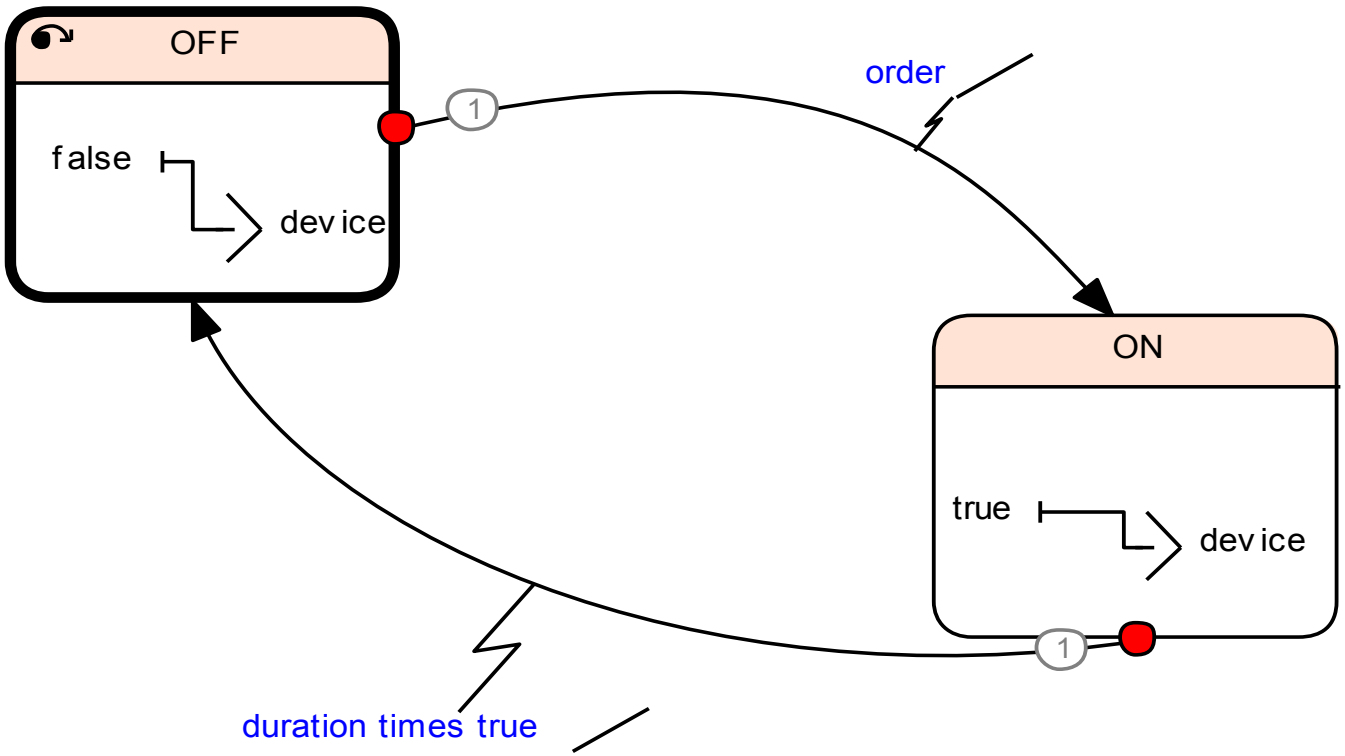
Example: Model Coverage

A small program: a pressure detector
Comparator



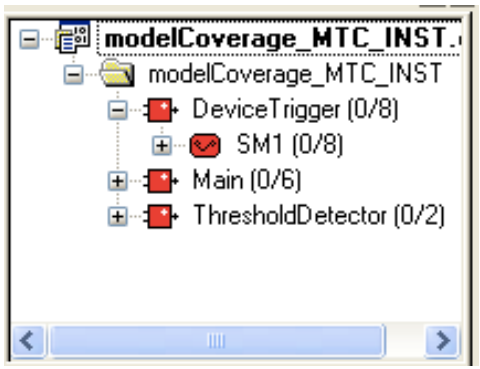
Example: Model Coverage

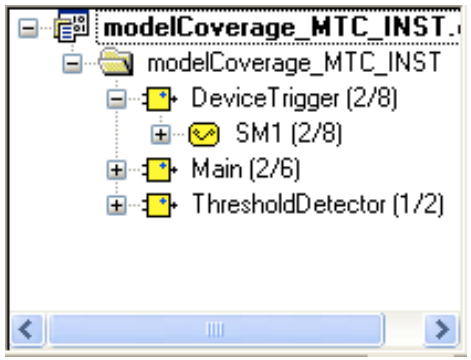
A small program: a pressure detector
Automaton

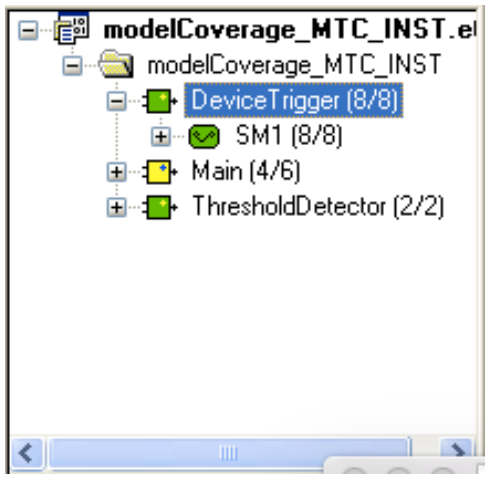


Example: Model Coverage

A small program: a pressure detector







- Predefined coverage criteria: Decision coverage, MC/DC, Masking MC/DC
- Predefined integration criteria: control activation, control and data activation

SOMCA: criteria

Criterion Name	Block Diagrams DAL					State Diagrams DAL				
	A	B	C	D	E	A	B	C	D	E
SOMCA Criterion 1: Range coverage	√	√	√			√	√	√		
SOMCA Criterion 2: Functionality coverage	√	√	√			√	√	√		
SOMCA Criterion 3: Modified input coverage	√	√	√			√	√	√		
SOMCA Criterion 4: Activation coverage	√	√	√			√	√	√		
SOMCA Criterion 5: Local Decision Coverage	√	√				NA	NA	NA	NA	NA
SOMCA Criterion 6: Logic Path coverage	√					NA	NA	NA	NA	NA
SOMCA Criterion 7: Parent State coverage	NA	NA	NA	NA	NA	√	√			
SOMCA Criterion 8: State History coverage	NA	NA	NA	NA	NA	√				
SOMCA Criterion 9: Transition coverage	NA	NA	NA	NA	NA	√	√	√		
SOMCA Criterion 10: Transition Decision Coverage	NA	NA	NA	NA	NA	√	√			
SOMCA Criterion 11: Transition MC/DC	NA	NA	NA	NA	NA	√				
SOMCA Criterion 12: Event coverage	NA	NA	NA	NA	NA	√	√	√		
SOMCA Criterion 13: Activating event coverage	NA	NA	NA	NA	NA	√	√			
SOMCA Criterion 14: Level-N Loop coverage	NA	NA	NA	NA	NA	√ (3)*	√ (2)*	√ (1)*		

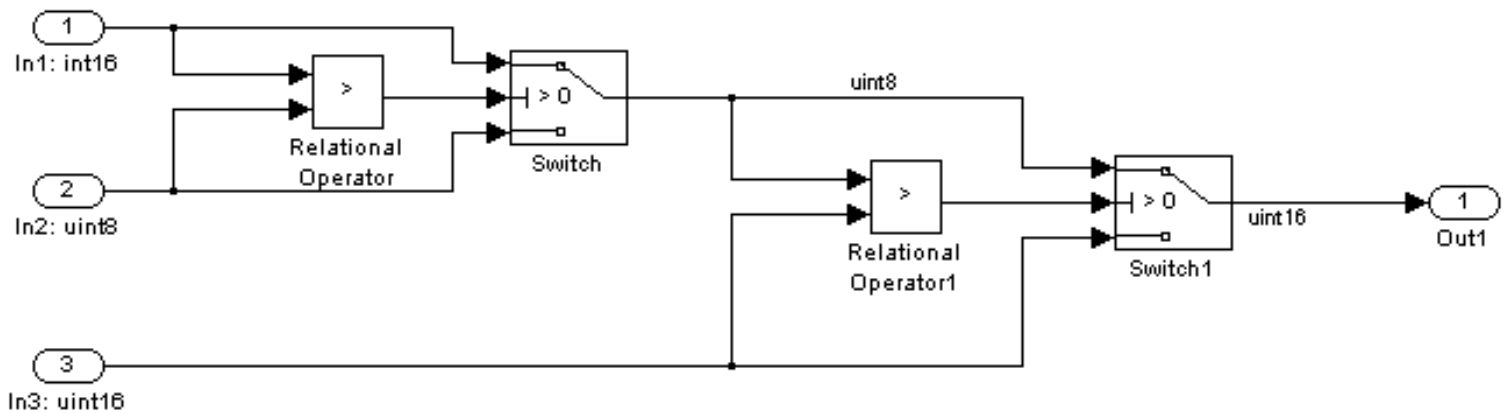
(* The number in brackets means the N level used in the coverage criterion)

Range coverage

Definition

All the significant values of the inputs and outputs of each model component must be exercised. That is

- All singular points of functional components
- All equivalence classes
- Continuous and discontinuous input signals.

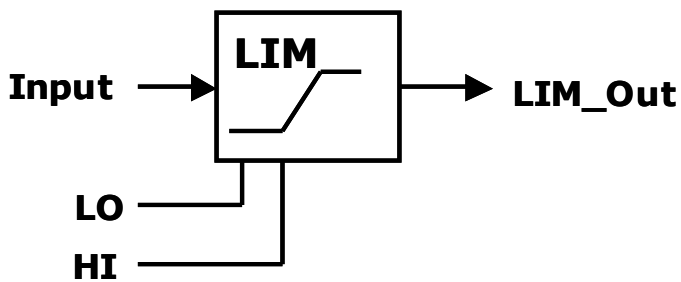


Functionality coverage

Definition

All characteristics of the functionality in context must be exercised for each component. That is

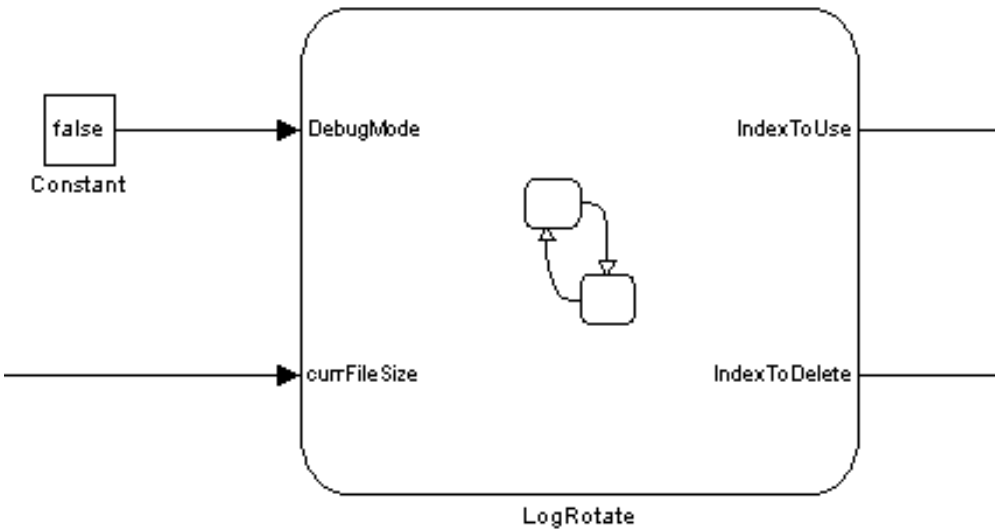
- Activation of characteristic (*e.g.*, watchdog function is triggered)
- Reaching internal conditions (*e.g.*, saturation block fed with an input over the upper limit)
- Stability of transfer functions (*e.g.*, fed with step wave, sine wave, etc.)



Modified input coverage

Definition

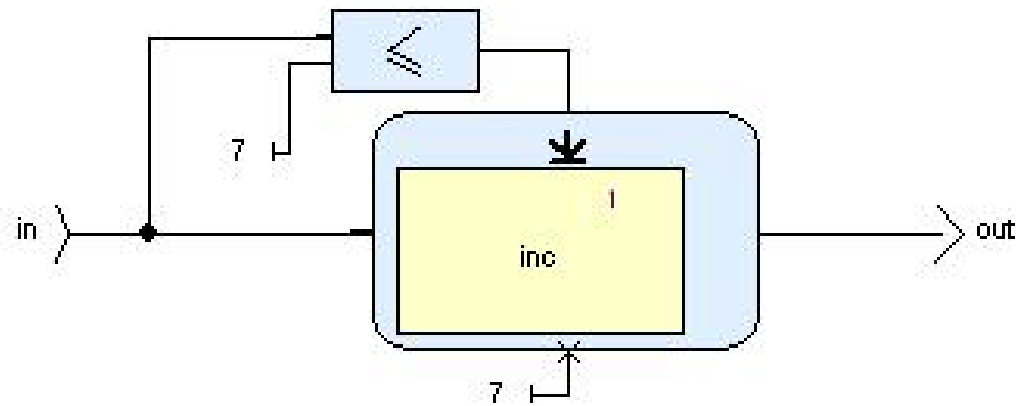
All inputs of every block and state diagram have changed at least once



Activation coverage

Definition

All model elements whose execution depends on an external rule or signal must be activated



Local decision coverage

Definition

Every block decision and Boolean output of a basic block has taken on all possible values at least once

This criterion applies the classical Decision Coverage to all blocks in a block diagram that depends on a Boolean expression, including:

- **Logical operators:** the output of each logical operators has taken both the True and False values (*e.g.*, and, or, xor, not, ...)
- **Relational operators:** the output of each relation operator has taken both the True and False values (*e.g.*, equals, greater than, less than, ...)
- **Switch blocks:** the decision of the switch/if block has taken both the True and False values, activating both inputs.
- **Selectors:** the decision of the selector has taken both the True and False values, enabling the processing of the contents of each branch.

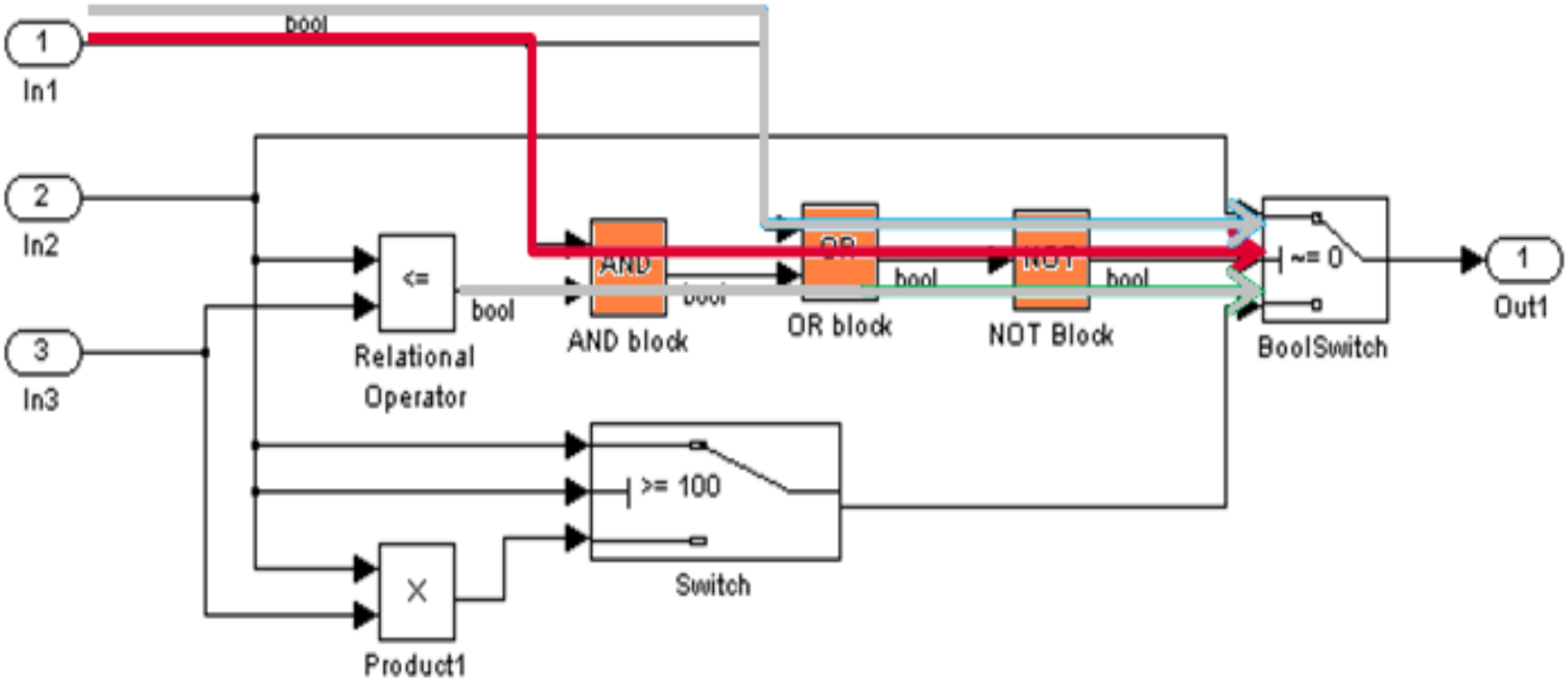
Logic path coverage

Definition

The input of every logic path has been shown to independently affect the output of the logic path.

For satisfying this criterion it is necessary to verify that a change in the input of a logic path modifies the output of that path when all the links of the path are active (*i.e.*, no block masks any of the links of the path). This requires at least two different verification cases where all links of each logic path are active, and also that the output link has both the True and False values.

Logic path coverage



- Test vector 1 = In1 = False, In2 = 0, In3 = 0
- Test vector 2 = In1 = True, In2 = 0, In3 = 1

The OR block masks the middle path in test vector 2

Parent state coverage

Definition

All states and substates have been entered and exited (except for those without exit transitions), and all substates have been active at least once when parent state exits.

Comment: The Parent State Coverage criterion not only requires that all substates have been activated, but also that the parent state (which contains different substates, and thus are active whenever one of its substates is active) has been exited when each specific substate was active.

State history coverage

Definition

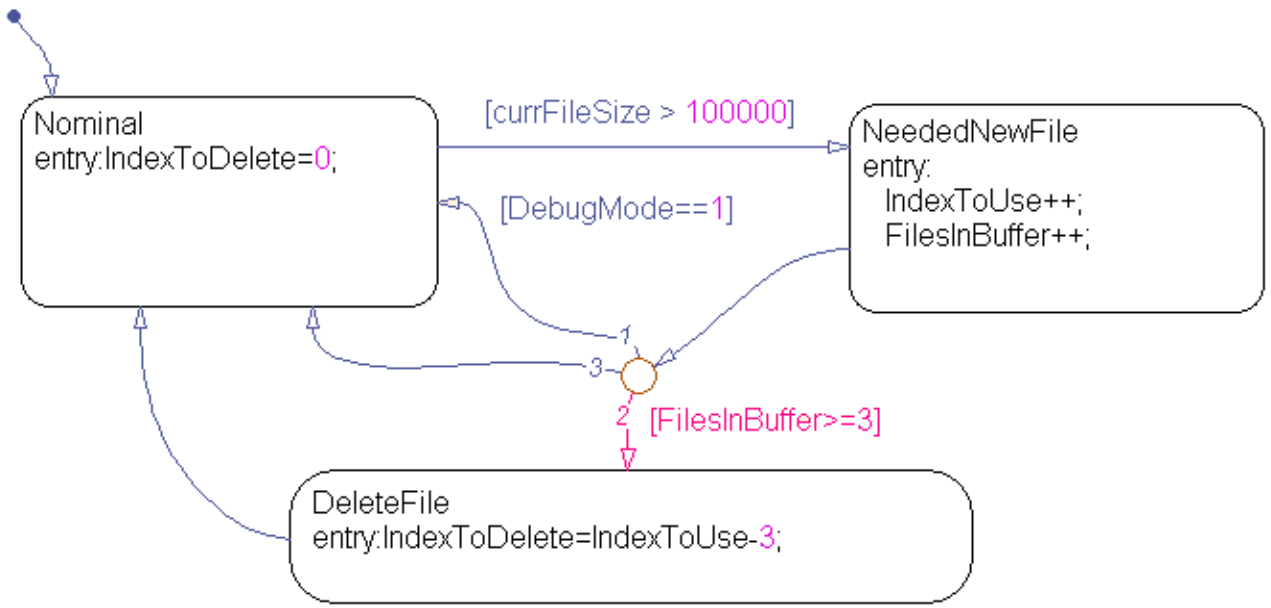
All states and substates have been entered and exited, and all substates that were active when parent state exits have been later re-entered.

Comment: The State History Coverage criterion is similar to the Parent State Coverage one, but also requires that the parent state is later re-entered and the last active substate reactivated.

Transition coverage

Definition

All transitions of the diagram have been exercised.



Note: The DebugMode prevents the execution of others transition if equals to 1

Transition decision coverage

Definition

Every decision in the transition decisions has taken all possible outcomes at least once.

Comment: This criterion improves the coverage provided by Transition Criterion by checking the decisions associated to the transitions.

Transition MC/DC

Definition

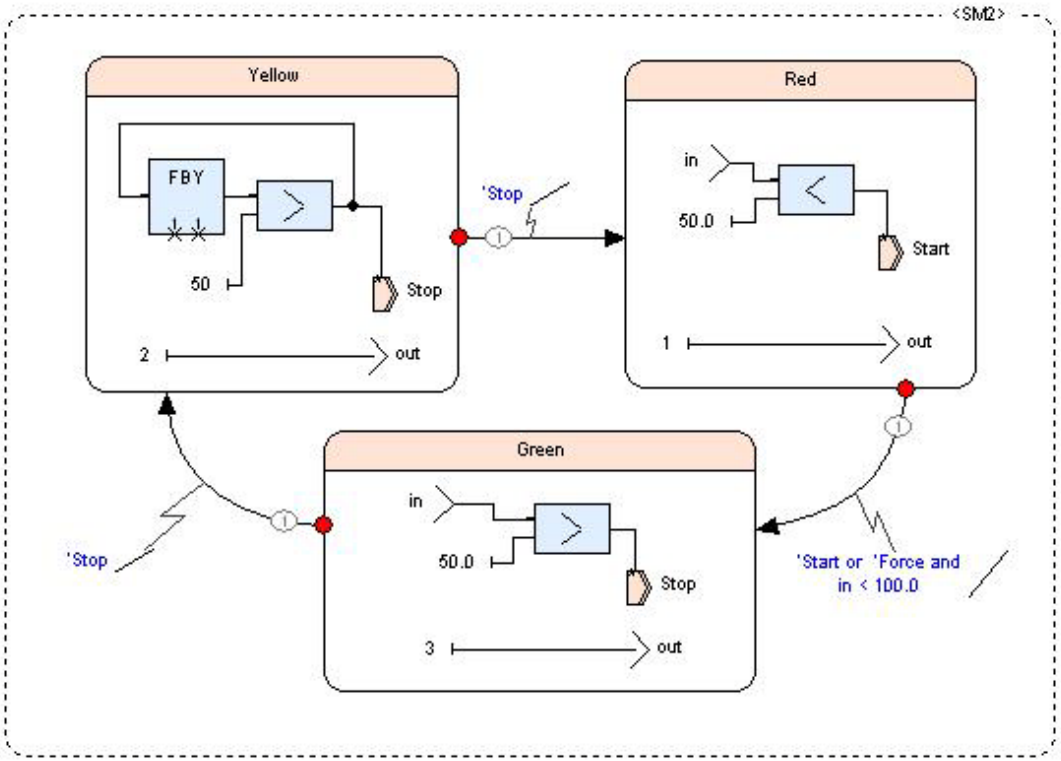
Every condition in a transition decision have taken on all possible outcomes at least once, and every decision in a transition decision have taken all possible outcomes at least once, and each condition in a transition decision has been shown to independently affect the transition decision's outcome.

Comment: This is the adaptation of the classic MC/DC for the State Machines formalism. This criterion affects all the conditions evaluated in all the diagram transitions and it is only applicable to State Machines and not to Block Diagram formalism.

Event coverage

Definition

All external events are received at least once in each state that has transitions associated to them.



Comment: This criterion covers the reception of events when there are transitions associated to them, providing coverage over the events that trigger them.

Activation event coverage

Definition

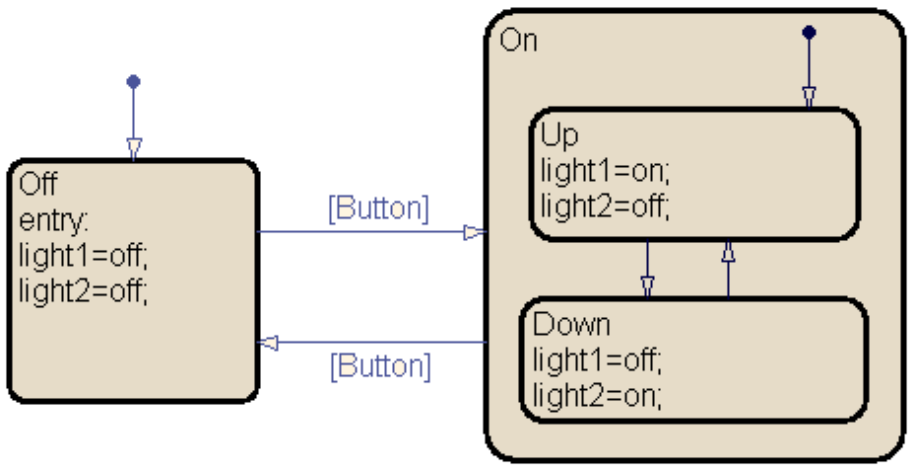
All external events activate a transition at least once in each state that has transitions associated to them.

Comment: This criterion is similar to the Event Coverage, but adds the requirement that the event received activates the associated transition at least once. This criterion only requires that the event activates each transition associated to it.

Level-N loop coverage

Definition

All loops of depth N, with same inputs values maintained for a given number of iterations (m), present in the model have been executed.



Comment: A loop of depth N in the model is a transition path of length N that starts and ends in the same state. The "static input" qualifier refers that the loop is present for a fixed combination of input data. The cycle must be completed without any change of the input signals to the model.

Conclusion

Testing Model Coverage is important to assert the absence of unintended functions.

Model Testing activity is still a complex task in regards to software testing.

Model Testing Coverage Criteria, as SOMCA defined them, are new in the industry

A question on verification

Model coverage vs Source Code coverage vs Object Code coverage?