Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

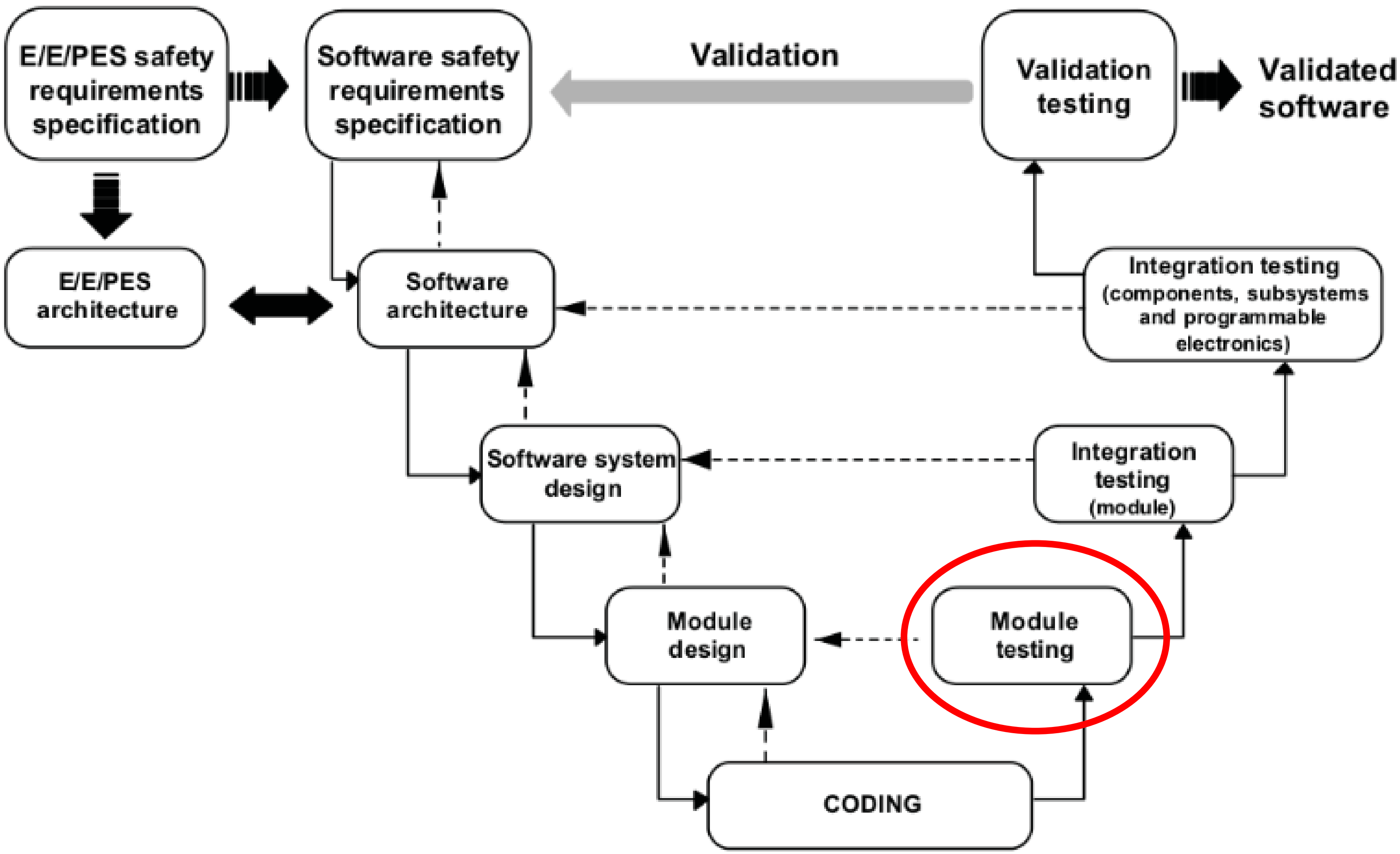
Lecture 07:

Testing

Christoph Lüth, Dieter Hutter, Jan Peleska

Where are we?

Testing in the Development Cycle



What is Testing?

Testing is the proce
intent of finding err

- In our sense, testi
- The **aim** of testing
  - derivation of
    compared to
  - inconsistency b
  - structural features of ... that cause a faulty behavior of a program

BUT: testing can prove the **absence** of errors under certain hypotheses – so-called **complete** test methods

see http://www.informatik.uni-bremen.de/agbs/jp/papers/test-automation-huang-peleska.pdf

This concept is closely related to model checking

It is well known that Dijkstra hated model checking and frowned upon testing …

> Program testing can be used to show the presence of bugs, but never to show their absence.
>
> *E.W. Dijkstra, 1972*

Why is testing so important?

- Even if one day code can be completely verified using formal methods, tests will still be required because--

- for embedded systems, the correctness of the HW/SW integration must be verified by testing, because--

- as of today, it is infeasible to provide a correct and complete formal model for complete HW/SW systems, comprising:
  - source code,
  - machine code,
  - CPU micro code,
  - firmware on interface hardware,
  - CPUs, busses, caches, memory, and interface boards.

- This will stay infeasible in the foreseeable future

The Testing Process

- Test cases, test plan, etc.
- System-under-test (s.u.t.)
  - Aka. TOE (target-of-evaluation) in CC
  - Aka. Implementation-under-test
- Warning -- test literature is quite expansive:

> Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.
>
> *Hetzel, 1983*

Test Levels

- **Component** and **unit tests**
  - test at the interface level of single components (modules, classes)
- **Integration test**
  - testing interfaces of components fit together
- **System test**
  - functional and non-functional test of the complete system from the user's perspective
- **Acceptance test**
  - testing if system implements contract details

Test Methods

- Static vs. dynamic
  - With **static** tests, the code is **analyzed** without being run. We cover these methods as static program analysis later
  - With **dynamic** tests, we **run** the code under controlled conditions, and check the results against a given specification

- Central question: where do the **test cases** come from?
  - **Black-box**: the inner structure of the s.u.t. is opaque, test cases are derived from specification **only.**
  - **Grey-box:** some inner structure of the s.u.t. is known, e.g. module architecture.
  - **White-box**: the inner structure of the s.u.t. is known, and tests cases are derived from the source code **and** coverage objectives for the source code

Black-Box Tests

- Limit analysis:
  - If the specification limits input parameters, then values **close** to these limits should be chosen
  - Idea is that programs behave **continuously**, and errors occur at these limits
- Equivalence classes:
  - If the input parameter values can be decomposed into **classes** which are treated equivalently, test cases have to cover all classes
- Smoke test:
  - "Run it, and check it does not go up in smoke."

Example: Black-Box Testing

- Equivalence classes or limits?

> **Example: A Company Bonus System**
> The loyalty bonus shall be computed depending on the time of employment. For employees of more than three years, it shall be 50% of the monthly salary, for employees of more than five years, 75%, and for employees of more than eight years, it shall be 100%.

- Equivalence classes or limits?

> **Example: Air Bag**
> The air bag shall be released if the vertical acceleration a_v equals or exceeds $15\ {}^{m}/_{s^2}$. The vertical acceleration will never be less than zero, or more than $40\ {}^{m}/_{s^2}$.

Black-Box Tests

- Quite typical for **GUI tests**, or **functional testing**
- Testing **invalid input**: depends on programming language – the stronger the typing, the less testing for invalid input is required
  - Example: consider lists in C, Java, Haskell
  - Example: consider object-relational mappings[1] (ORM) in Python, Java

1) Translating e.g. SQL-entries to objects

Complete Model-based Black-box Testing

- Create a model *M* of the expected system behaviour
- Specify a **fault model** *(M, ≤, Dom)* with reference model *M*, **conformance relation** ≤ and **fault domain** *Dom* (a collection of models that may or may not conform to M)
- Derive test cases from fault model
- The resulting test suite is **complete** if
  - Every conforming SUT will pass all tests (**soundness**)
  - Every non-conforming SUT whose true behavior is reflected by a member of the fault domain fails at least on test case (**exhaustiveness**)
  - (nothing is guaranteed for SUT behaviors outside the fault domain)

Example: the W-Method

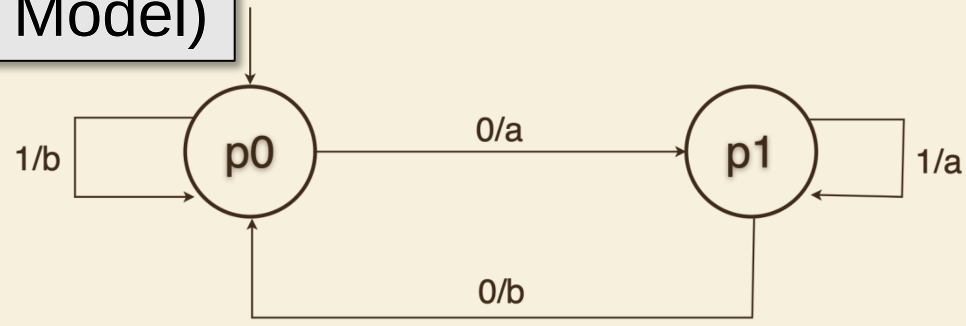
- The W-Method specifies a recipe for constructing complete test suites for finite state machines (FSMs) with conformance relation "$\sim$"
  language equivalence (I/O-equivalence):

  - Create a state cover *V*
  - Create a characterization set *W*
  - Assume that implementation has at most m ≥ n states (n is the number of states in the observable, minimized reference model)
  - Create test suite according to formula

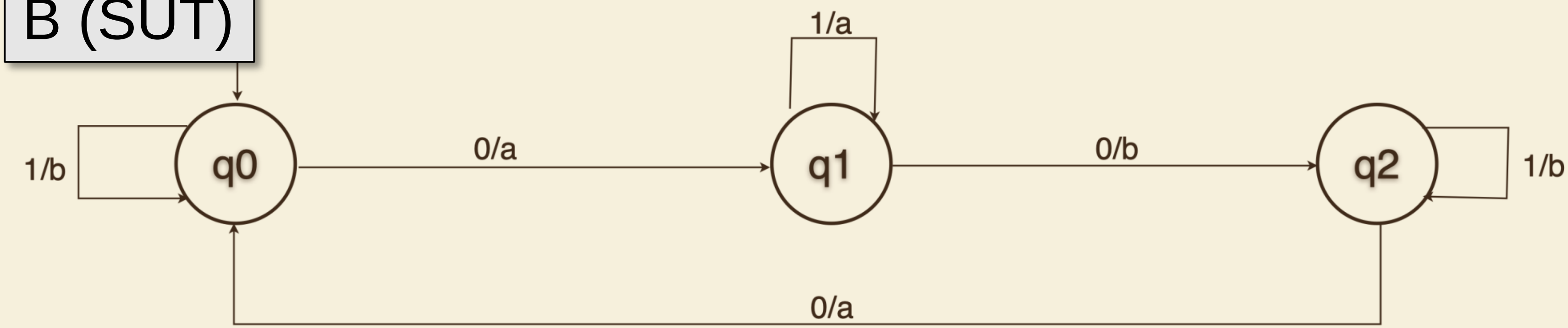
$$\mathcal{W} = V.\left(\bigcup_{i=0}^{m-n+1} I^i\right).W$$

I : input alphabet
I^i : input traces of length i
$A.B$: all traces of A concatenated with all traces from B

A (Reference Model)



B (SUT)



$$
\begin{aligned}
n &= 2 \\
m &= 3 \\
V &= \{\varepsilon, 0\} \\
W &= \{1\}
\end{aligned}
$$

$$\bigcup_{i=0}^{m-n+1} I^i = \{\varepsilon, 0, 1, 0.0, 0.1, 1.0, 1.1\}$$

$$
\begin{aligned}
\mathcal{W}(A) &= V.(\bigcup_{i=0}^{m-n+1} I^i).W \\
&= \{1, 0.1, 1.1, 0.0.1, 0.1.1, 1.0.1, 1.1.1 \\
&\quad 0.0.0.1, 0.0.1.1, 0.1.0.1, 0.1.1.1\}
\end{aligned}
$$

SUT error is uncovered by 0.0.0.1

Property-based Testing

- In property-based testing (or random testing), we generate **random** input values, and check the results against a given **executable** specification.

- Attention needs to be paid to the **distribution** values.

- Works better with **high-level languages,** where the datatypes represent more information on an abstract level and where the language is powerful enough to write comprehensive executable specifications (i.e. Boolean expressions).
  - Implementations for e.g. Haskell (QuickCheck), Scala (ScalaCheck), Java

- Example: consider list reversal in C, Java, Haskell
  - Executable spec: reversal is idempotent and distributes over concatenation.
  - Question: how to generate random lists?

White-Box Tests

- In white-box tests, we derive test cases based on the structure of the program (**structural testing**)
  - To abstract from the source code (which is a purely syntactic artefact), we consider the **control flow graph** of the program.

Def: Control Flow Graph (CFG)

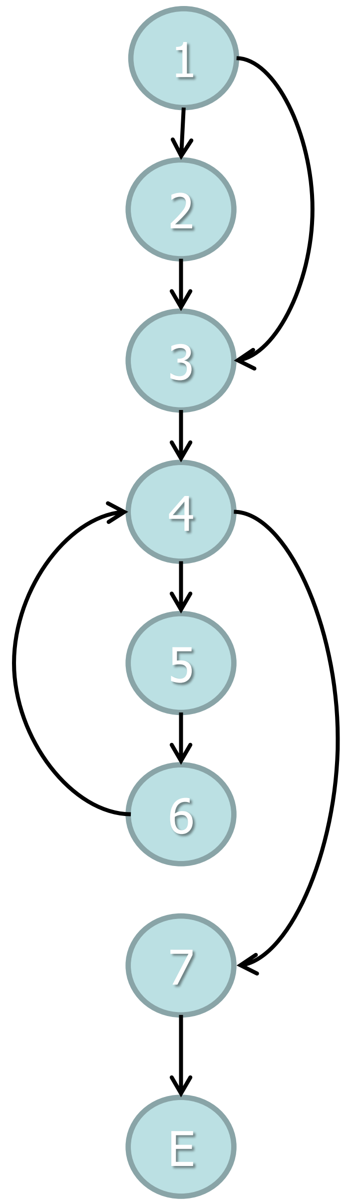
- nodes as elementary statements (e.g. assignments, **return**, **break**, . . .), as well as control expressions (e.g. in conditionals and loops), and
- vertices from *n* to *m* if the control flow can reach a node *m* coming from a node *n*.

- Hence, **paths** in the CFG correspond to **runs** of the program.

Example: Control-Flow Graph

```
if (x < 0)  /*1*/  {
   x= – x; /*2*/
}
z = 1;  /*3*/

while (x > 0)  /*4*/ {
  z = z * y;  /*5*/
  x = x – 1;   /*6*/
}
return z;  /*7*/
```



An execution path is a path though the cfg ending with an exit node.

Examples:

- [1,3,4,7, E]
- [1,2,3,4,7, E]
- [1,2,3,4,5,6,4,7, E]
- [1,3,4,5,6,4,5,6,4,7, E]
- …

Coverage

- **Statement coverage**:
  Measures the percentage of statements that were covered by the tests.
  100% statement coverage is reached if each **node** in the CFG has been visited at least once.

- **Branch coverage**:
  Measures the percentage of **edges** (emanating from branching or non-branching nodes) covered by the tests.
  100% branch coverage is reached if every edge of the CFG has been traversed at least once.

- **Path coverage**:
  Measures the percentage of CFG **paths** that have been covered by the tests.
  100% path coverage is achieved if every path of the CFG has been covered at least once.

Decision Coverage

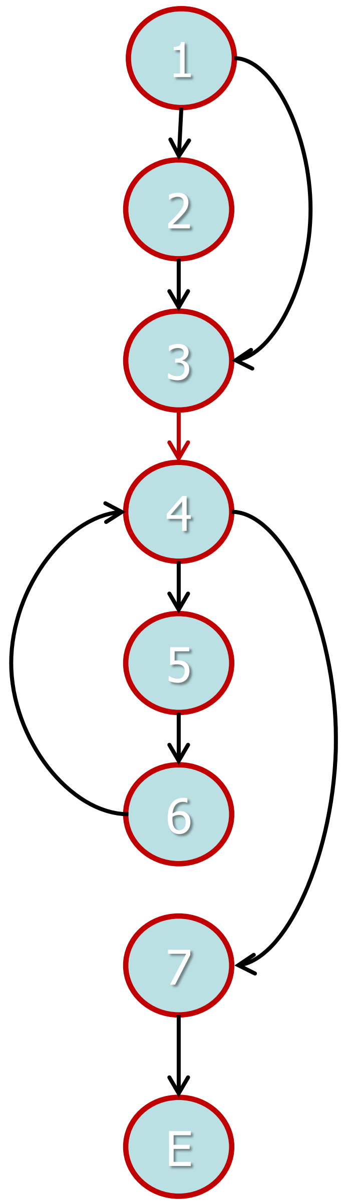
- **Decision coverage**:
  Measures the coverage of conditional branches (i.e., edges emanating from conditional nodes). 100% decision coverage is reaches if the tests cover all conditional branches.

- **Decision coverage vs. branch coverage:**
  - If branch coverage is 100%, then decision coverage is 100% and vice versa.
  - A lower percentage *p < 100%* of branch coverage, however, has a different meaning than a decision coverage of *p,* because
  - branch coverage considers all edges, whereas
  - decision coverage considers edges emanating from decision nodes only

Example: Statement Coverage

```
if (x < 0)    /*1*/  {
   x= – x;   /*2*/
   }
z = 1;  /*3*/

while (x > 0)  /*4*/ {
  z = z * y;   /*5*/
  x = x – 1;   /*6*/
  }
return z;   /*7*/
```



- Which (minimal) path covers all statements?

  p = [1,2,3,4,5,6,4,7,E]

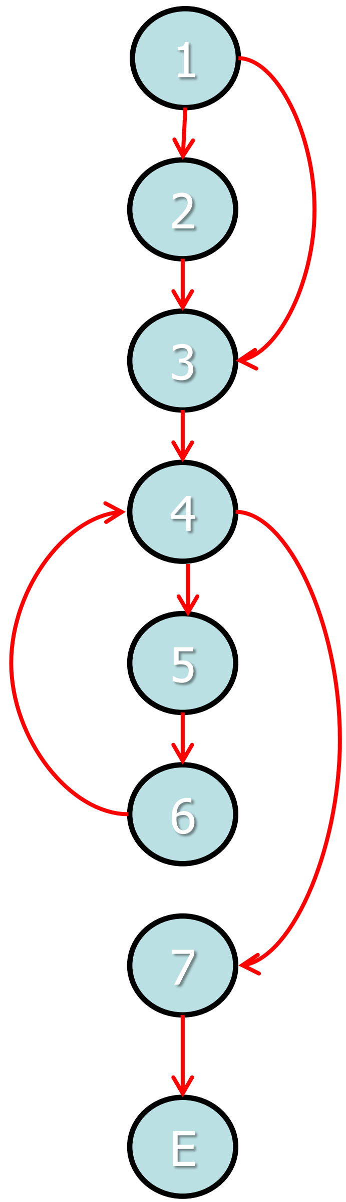
- Which state generates p?

  x = -1
  y any
  z any

Example: Branch Coverage

```
if (x < 0)    /*1*/  {
   x= – x;   /*2*/
   }
z = 1;  /*3*/

while (x > 0)  /*4*/ {
  z = z * y;   /*5*/
  x = x – 1;   /*6*/
  }
return z;   /*7*/
```



- Which (minimal) paths cover all vertices?

$$p_1 = [1,2,3,4,5,6,4,7,E]$$
$$p_2 = [1,3,4,7,E]$$

- Which states generate p_1, p_2?

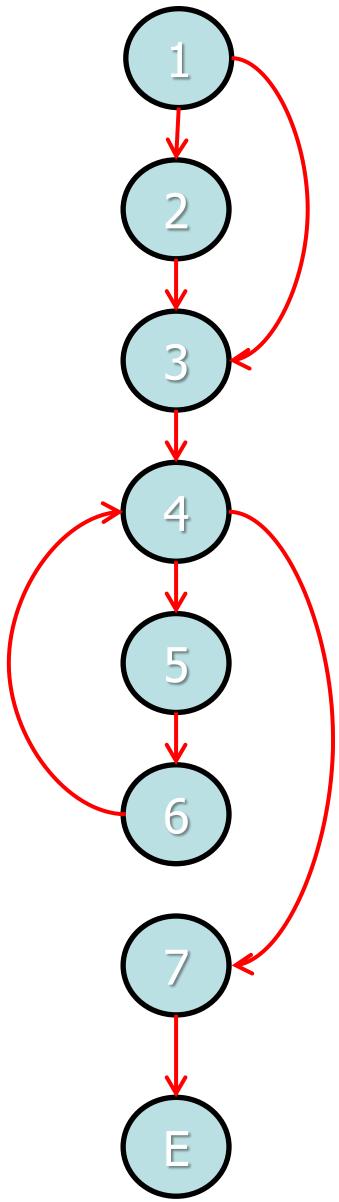
	p_1	p_2
x	-1	0
y	any	any
z	any	Any

- Note p_3 with x= 1 does not add coverage.

Example: Path Coverage

```
if (x < 0)    /*1*/  {
   x= – x;   /*2*/
   }
z = 1;  /*3*/

while (x > 0)  /*4*/ {
  z = z * y;   /*5*/
  x = x – 1;   /*6*/
  }
return z;   /*7*/
```



- How many paths are there?
- Let $q_1 = [1,2,3]$
  $q_2 = [1,3]$
  $p = [4,5,6]$
  $r = [4,7,E]$

  then all paths are

$$P = (q_1|q_2)\ p^*\ r$$

- Number of possible paths:

$$|P| = 2 \cdot MaxInt - 1$$

Statement, Branch and Path Coverage

- **Statement Coverage**:
  - Necessary but not sufficient, not suitable as only test approach.
  - Detects dead code (code which is never executed).
  - About 18% of all defects are identified.
- **Branch coverage**:
  - Least possible single approach.
  - Needs to be achieved by (specification-based) tests for avionic software of DAL-C – does not suffice for DAL-B or DAL-A.
  - Detects dead code, but also frequently executed program parts.
  - About 34% of all defects are identified.
- **Path Coverage:**
  - Most powerful structural approach;
  - Highest defect identification rate (close to 100%);
  - But no **practical** relevance.

Decision Coverage Revisited

- Decision coverage requires that for all decisions in the program, each possible outcome is considered once.
- **Problem**: cannot sufficiently distinguish Boolean expressions.
  - Example: for `A || B`, the following are sufficient:

A	B	Result
False	False	False
True	False	True

  - But this does not distinguish `A || B` from `A`; `B` is effectively not tested.

Decomposing Boolean Expressions

- The binary Boolean operators include conjunction $x \wedge y$, disjunction $x \vee y$, or anything expressible by these (e.g. exclusive disjunction, implication)

> **Elementary Boolean Terms**
> An elementary Boolean term does not contain binary Boolean operators, and cannot be further decomposed.

- An elementary term is a variable, a Boolean-valued function, a relation (equality $=$, orders $<, \leq, >, \geq$, etc.), or a negation of these.
- This is a fairly syntactic view, e.g. $x \leq y$ is elementary, but $x < y \vee x = y$ is not, even though they are equivalent.
- In formal logic, these are called **literals.**

Simple Condition Coverage

- For each decision in the program, each elementary Boolean term (condition) evaluates to *True* and *False* at least once
- Note that this does not say much about the possible value of the condition
- Example:

```
if (temperature > 90 && pressure > 120) {... }
```

C1	*C2*	*Result*	
False	False	False	
False	True	False	-- These two would be enough
True	False	False	-- for condition coverage
True	True	True	

Modified Condition Coverage

- It is not always possible to generate all possible combinations of elementary terms, e.g. 3 <= x && x < 5.
- In modified (or minimal) condition coverage, all possible combinations of those elementary terms the value of which determines the value of the whole condition need to be considered.
- Example: 3 <= x && x < 5

3 <= x	x < 5	Result
False	False	False
False	True	False
True	False	True
True	True	True

- Another example: (x > 1 && ! p) || p

Modified Condition/Decision Coverage

- Modified Condition/Decision Coverage (MC/DC) is required by the "aerospace norm" **DO-178B** for Level A software.

- It is a **combination** of the previous coverage criteria defined as follows:

  - Every point of entry and exit in the program has been invoked at least once;
  - Every decision in the program has taken all possible outcomes at least once;
  - Every condition (i.e. elementary Boolean terms earlier) 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.

How to achieve MC/DC

- **Not**: Here is the source code, what is the minimal set of test cases?
- **Rather**: From requirements we get test cases, do they achieve MC/DC?
- Example:
  - Test cases:

Source Code:
Z = (A || B) && (C || D)

Test case	**1**	**2**	**3**	**4**	**5**
Input A	F	F	T	F	T
Input B	F	T	F	T	F
Input C	T	F	F	T	T
Input D	F	T	F	F	F
Result Z	*F*	*T*	*F*	*T*	*T*

Question: do test cases achieve MC/DC?

Source: Hayhurst *et al*, A Practical Tutorial on MC/DC. NASA/TM2001-210876

Example: MC/DC

Determining MC/DC:

1. Are all decisions covered?
2. Eliminate masked inputs (recursively)
   - False for && masks other input
   - True for || masks other input
3. Remaining unmasked test cases must cover all conditions.

Source Code

Z = (A || B) && (C || D)

Test case	**1**	**2**	**3**	**4**	**5**
Input A	F	F	T	F	T
Input B	F	T	F	T	F
Input C	T	F	F	T	T
Input D	F	T	F	F	F
Result Z	*F*	*T*	*F*	*T*	*T*

Here:

- Result is both F and T, so decisions covered.
- Masking:
  - In test case 1, C and D are masked
  - In test case 3, A and B are masked
  - Recursive masking as shown
- Remaining cases cover T, F for A, B, C, D
  - MC/DC achieved
  - In fact, test case 4 not even needed (?)

Summary

- (Dynamic) Testing is the controlled execution of code, and comparing the result against an expected outcome.
- Testing is (traditionally) the main way for **verification.**
- Depending on how the test cases are derived, we distinguish **white-box** and **black-box** tests.
- In black-box tests, we can consider **limits** and **equivalence classes** for input values to obtain test cases.
- In white-box tests, we have different notions of **coverage**: statement coverage, path coverage, condition coverage, etc.
- Next week: **Static testing** aka. static **program analysis**