# Dynamic Test

---

## Contents

☐ Properties and goals

☐ Structural testing
- Control flow testing
- Data flow testing

☐ Functional test

☐ Diversified test

## Properties and Goals

☐ Properties of the dynamic test
  - Executable program is provided with concrete input values and is executed
  - Program may be tested in the real environment
  - Never complete
  - Correctness of the tested program cannot be proven

☐ Characteristics of the application of dynamic test methods in practice
  - Widely-used
  - Often very unsystematically applied
  - Tests often not reproduceable
  - Diffuse activity (Management problems)

---

## Properties and Goals

☐ The goal of dynamic testing is the generation of test cases that are
  - Representative
  - Fault sensitive
  - Distinct from each other (minimal redundancy)
  - Economic

## Structural Testing

☐ Evaluation of the adequacy and completeness of the test cases on the basis of the software structure. Determination of the correctness of the outputs based on the specification
  ▪ Benefit: Code structure is considered (instructions, branches, data accesses, etc.)
  ▪ Disadvantage: Forgotten (not implemented), but specified functions cannot be detected
☐ Two approaches
  ▪ Control flow testing
  ▪ Data flow testing

---

## Structural Testing
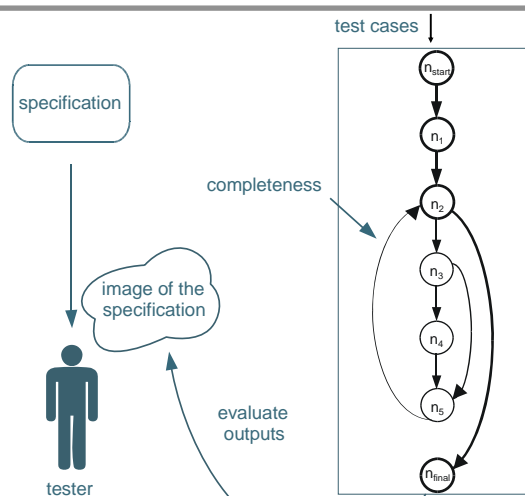
## Control Flow Testing

- ☐ Statement coverage test
- ☐ Branch coverage test
- ☐ Condition coverage test
  - ▪ simple
  - ▪ minimal multiple
  - ▪ multiple
- ☐ LCSAJ-based test
- ☐ Boundary interior-path test
- ☐ Structured path test
- ☐ Path test

**Control flow testing** is based on the control structure respectively on the control flow. The basis is the control flow graph

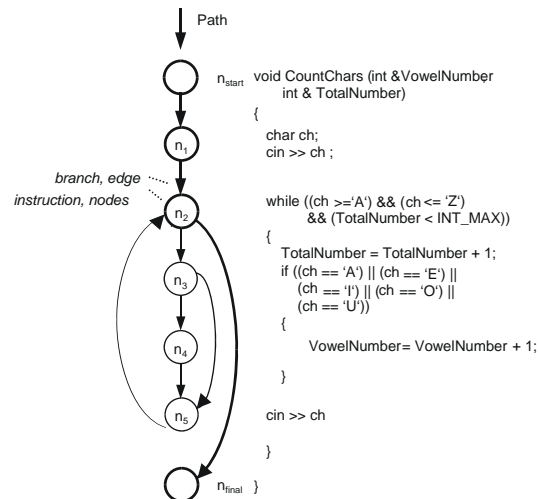---

## Control Flow Testing

### Example

```
void CountChars(int &VowelNumber, int &TotalNumber)
// Precondition: VowelNumber <= TotalNumber
{ char ch;
  cin>>ch;
  while ((ch >= 'A')&&(ch <= 'Z')&&
              (TotalNumber < INT_MAX))
  { TotalNumber = TotalNumber+1;
     if((ch == 'A')||(ch == 'E')||(ch == 'I')||
            (ch == 'O')||(ch == 'U'))
     {           VowelNumber = VowelNumber + 1;
     }
     cin>>ch;
  } //end while
}
```

Path

$n_{start}$    void CountChars (int &VowelNumber
         int & TotalNumber)
{
   char ch;
   cin >> ch ;

*branch, edge*
*instruction, nodes*

while ((ch >='A') && (ch <= 'Z')
     && (TotalNumber < INT_MAX))
{
   TotalNumber = TotalNumber + 1;
   if ((ch == 'A') || (ch == 'E') ||
     (ch == 'I') || (ch == 'O') ||
     (ch == 'U'))
   {
      VowelNumber= VowelNumber + 1;

   }

cin >> ch

}

$n_{final}$   }

---

☐ The statement coverage is the simplest control flow test technique. It is also referred to as $C_0$-test

☐ The goal of the statement coverage is to execute each statement at least once, i.e., the execution of all nodes of the control flow graph

☐ The statement coverage rate is the relation of the executed instructions to the total number of the instructions

$$C_{instruction} = \frac{number\ of\ executed\ instructions}{number\ of\ instructions}$$

☐ Then all instructions of the module to be tested are executed at least once a complete statement coverage test is achieved

➔ No untested code !?

☐ Statement coverage test demands the execution of all nodes of the control flow graph, i.e., the corresponding program paths must contain all nodes of the control flow graph

☐ Test case
call of *CountChars* with:  totalnumber = 0
input chars:  'A', '1'
path:  $(n_{start}, n1, n2, n3, n4, n5, n2, n_{final})$

☐ Observation
  ▪ The test path contains all nodes
  ▪ **but it does not contain all edges** of the control flow graph. The edge (n3,n5) is not contained

---

☐ Statement coverage is considered to be a weak criterion. It has a limited practical importance

☐ The standard RTCA DO-178B for software applications in aviation demands to apply statement coverage to level-C-software. In case of a software failure such a software can cause a major failure condition

☐ Branch coverage aims at executing all branches of the program to be tested. This requires the execution of all edges of the control flow graph. It is also referred to as $C_1$-test

☐ Branch coverage is a stricter test technique than statement coverage. Statement coverage is fully contained in branch coverage. Branch coverage **subsumes** statement coverage

☐ Branch coverage is generally considered as a minimal criterion in software unit testing

☐ The standard RTCA DO-178B requires branch coverage testing for level-B-software

---

☐ Example

Branch coverage demands the execution of all edges of the control flow graph. This is achieved if every decision of the unit under test had at least once the logical value *false* and *true*

☐ Test case
call of CountChars with:      totalnumber = 0
input chars:      „A", „B", „1"
flow path:      $(n_{start}, n_1, n_2, n_3, n_4, n_5, n_2, n_3, n_5, n_2, n_{final})$

☐ The test path contains all edges. In particular it contains the edge $(n_3, n_5)$ which is not necessarily executed by statement coverage. Branch coverage subsumes statement coverage

□ **Question:** Is branch coverage adequate for testing of complicated, composite decisions?

□ Examples

- Simple decision: if (x > 5)...;
  - The decision (x > 5) can be regarded as sufficiently tested if both logical values occurred within the test. The decision subdivides the possible test data into two classes and demands that at least one test date is selected from every class

- Complex decision: if (((u == 0) || (x > 5)) && ((y < 6) || (z == 0))) ...
  - A test of the decision (((u == 0) || (x > 5)) && ((y < 6) || (z == 0))) against both logical values cannot be regarded as sufficient, as the structure of the decision is not considered appropriately
  - A complete branch coverage test can be achieved e.g. with the following test cases

    Test case 1: u = 1, x = 4, y = 5, z = 0
    Test case 2: u = 0, x = 6, y = 5, z = 0

□ Assumption: Composite decisions are tested from left to right. The evaluation of decisions stops when its logical value is known. This is referred to as incomplete evaluation of decisions

□ Test case 1 leads to the following situation

- Value 1 of the variable u for the first condition of the OR-connection gives the logical value false. Therefore the second condition of the OR-connection defines the logical value of the OR-connection. The choice of the value 4 for the variable x inserted into the second condition (x > 5) also gives the logical value false. Thus the connection of the first two decisions also has the logical value false. Due to the subsequent AND-connection it is already known at this time that the overall decision has the logical value false. This result is independent from the logical values of the 3rd and 4th condition. This test case thus does not test these parts of the decision

- In many cases the logical values of some conditions are not tested. Independently of the fact if they are tested the logical value false for the first condition in an AND-connection masks the logical values of all further conditions. Such a test case thus is "blind" with regard to faults in the remaining conditions

- □ Test case 2 causes the following situation
    - ▪ The choice of the value 0 for the variable u has the effect that the first condition (u == 0) has the logical value true. Due to the OR-connection of the first two conditions it is ensured that the result of the first OR-connection is true. The second condition has not to be tested. The testing can be directly continued with the first condition of the second OR-connection. The value 5 of the variable y causes the logical value true for the condition (Y < 6). Due to the OR-operator it is ensured at that time that the overall result will be true, independently of the logical value of the fourth condition. Thus this test case is "blind" with regard to the faults in the 2nd and 4th condition
- □ The test cases 1 and 2 cause a complete branch coverage. None of the two test cases tests the fourth partial decision. Then decisions are evaluated from left to right the conditions at the right hand site may remain untested

➔ branch coverage is usually inadequate for testing compound decisions

---

□ The decision (((u ==0) || (x > 5)) && ((y < 6) || (z ==0))) is abbreviated to ((A || B) && ((C || D). We assume that between the values of the variables u, x, y, and z no dependences exist. Then the partial decisions A, B, C, and D can be *true (T)* or *false (F)* independently of each other. Concerning a complete evaluation of decisions 16 combinations of logical values are possible

| | A | B | C | D | A\|\|B | C\|\|D | (A\|\|B)&&(C\|\|D) |
|---|---|---|---|---|---|---|---|
| 1 | F | F | F | F | F | F | F |
| 2 | F | F | F | T | F | T | F |
| 3 | F | F | T | F | F | T | F |
| 4 | F | F | T | T | F | T | F |
| 5 | F | T | F | F | T | F | F |
| 6 | F | T | F | T | T | T | T |
| 7 | F | T | T | F | T | T | T |
| 8 | F | T | T | T | T | T | T |
| 9 | T | F | F | F | T | F | F |
| 10 | T | F | F | T | T | T | T |
| 11 | T | F | T | F | T | T | T |
| 12 | T | F | T | T | T | T | T |
| 13 | T | T | F | F | T | F | F |
| 14 | T | T | F | T | T | T | T |
| 15 | T | T | T | F | T | T | T |
| 16 | T | T | T | T | T | T | T |

□ The simple condition coverage demands the test of all simple conditions concerning *true* and *false*

□ Benefits: simple, low test costs

□ Disadvantages

- Limited performance
- In general (concerning the complete evaluation of decisions) it cannot be guaranteed that the simple condition coverage subsumes the branch coverage

---

□ A simple condition coverage can be achieved, e.g., with the two test cases 6 and 11. The four simple conditions A, B, C, D are tested each against *true* and *false*

□ The conditions (A || B) and (C || D) and the decision ((A || B) && (C || D)) are *true* in both cases

□ These test cases do **not** achieve a complete branch coverage

|    | A | B | C | D | A\|\|B | C\|\|D | (A\|\|B)&&(C\|\|D) |
|----|---|---|---|---|------|------|-----------------|
| 1  | F | F | F | F |      |      |                 |
| 2  | F | F | F | T |      |      |                 |
| 3  | F | F | T | F |      |      |                 |
| 4  | F | F | T | T |      |      |                 |
| 5  | F | T | F | F |      |      |                 |
| 6  | F | T | F | T | T    | T    | T               |
| 7  | F | T | T | F |      |      |                 |
| 8  | F | T | T | T |      |      |                 |
| 9  | T | F | F | F |      |      |                 |
| 10 | T | F | F | T |      |      |                 |
| 11 | T | F | T | F | T    | T    | T               |
| 12 | T | F | T | T |      |      |                 |
| 13 | T | T | F | F |      |      |                 |
| 14 | T | T | F | T |      |      |                 |
| 15 | T | T | T | F |      |      |                 |
| 16 | T | T | T | T |      |      |                 |

- If the test cases 1 and 16 were chosen a complete branch coverage would have been achieved
- As the example shows there are test cases which fulfill the simple condition coverage without ensuring a branch coverage
- The simple condition coverage does **not** ensure the branch coverage

|    | A | B | C | D | A\|\|B | C\|\|D | (A\|\|B)&&(C\|\|D) |
|----|---|---|---|---|------|------|-----------------|
| 1  | F | F | F | F | F    | F    | F               |
| 2  | F | F | F | T |      |      |                 |
| 3  | F | F | T | F |      |      |                 |
| 4  | F | F | T | T |      |      |                 |
| 5  | F | T | F | F |      |      |                 |
| 6  | F | T | F | T |      |      |                 |
| 7  | F | T | T | F |      |      |                 |
| 8  | F | T | T | T |      |      |                 |
| 9  | T | F | F | F |      |      |                 |
| 10 | T | F | F | T |      |      |                 |
| 11 | T | F | T | F |      |      |                 |
| 12 | T | F | T | T |      |      |                 |
| 13 | T | T | F | F |      |      |                 |
| 14 | T | T | F | T |      |      |                 |
| 15 | T | T | T | F |      |      |                 |
| 16 | T | T | T | T | T    | T    | T               |

0101**seda**010100
software engineering dependability
© Prof. Dr. Liggesmeyer, 21

---

- If decisions are evaluated incomplete from left to right only 7 combination of truth-values exist (instead of 16)
- The test cases 6 and 11 which produce a simple condition coverage if decisions are evaluated completely are mapped to test cases III and VII
- In contrast to the incomplete evaluation of decisions the two test cases cause no complete simple condition coverage

|      | Test cases      | A | B | C | D | A\|\|B | C\|\|D | (A\|\|B)&&(C\|\|D) |
|------|-----------------|---|---|---|---|------|------|-----------------|
| I    | 1, 2, 3, 4      | F | F | - | - | F    | -    | F               |
| II   | 5               | F | T | F | F | T    | F    | F               |
| III  | 6               | F | T | F | T | T    | T    | T               |
| IV   | 7, 8            | F | T | T | - | T    | T    | T               |
| V    | 9, 13           | T | - | F | F | T    | F    | F               |
| VI   | 10, 14          | T | - | F | T | T    | T    | T               |
| VII  | 11, 12, 15, 16  | T | - | T | - | T    | T    | T               |

0101**seda**010100
software engineering dependability
© Prof. Dr. Liggesmeyer, 22

## Control Flow Testing
## Simple Condition Coverage

- ☐ Partial decision B can be tested against *false* only by selecting test case I
- ☐ To test D against *false* either test case II or V has to be executed
- ☐ A simple condition coverage is possible, e.g., with the test cases I, II, III and VII
- ☐ In addition these test cases ensure a complete branch coverage
- ☐ This rules is valid in every situation: **If decisions are evaluated incompletely the simple condition coverage subsumes branch coverage**

|      | A | B | C | D | A\|\|B | C\|\|D | (A\|\|B)&&(C\|\|D) |
|------|---|---|---|---|------|------|------------------|
| I    | F | F | - | - | F    | -    | F                |
| II   | F | T | F | F | T    | F    | F                |
| III  | F | T | F | T | T    | T    | T                |
| IV   | F | T | T | - | T    | T    | T                |
| V    | T | - | F | F | T    | F    | F                |
| VI   | T | - | F | T | T    | T    | T                |
| VII  | T | - | T | - | T    | T    | T                |

---

## Control Flow Testing
## Condition/Decision Coverage

- ☐ The condition/decision coverage guarantees a complete branch coverage in addition to a simple condition coverage
- ☐ It demands that the branch coverage is taken into account explicitly in addition to the condition coverage
- ☐ As this is already ensured by the simple condition coverage test concerning an incomplete evaluation of decisions this method is important only for the case of the complete evaluation of decisions
- ☐ Benefits: simple, low test costs, branch coverage is ensured
- ☐ Disadvantages
  - ▪ Limited performance
  - ▪ Structure of decisions is not really considered

- The execution of the test cases 5 and 12 results in a complete condition/decision coverage, as the partial decisions A, B, C, and D **and** the overall decision are each evaluated to *true* and *false*
- This is possible without testing the composite conditions (A || B) and (C || D) against both logical values
- The condition/decision coverage tests simple conditions and decisions
- **It widely ignores the decomposition of compound decisions into conditions on several levels**

| | A | B | C | D | A\|\|B | C\|\|D | (A\|\|B)&&(C\|\|D) |
|---|---|---|---|---|---|---|---|
| 1 | F | F | F | F | | | |
| 2 | F | F | F | T | | | |
| 3 | F | F | T | F | | | |
| 4 | F | F | T | T | | | |
| 5 | F | T | F | F | T | F | F |
| 6 | F | T | F | T | | | |
| 7 | F | T | T | F | | | |
| 8 | F | T | T | T | | | |
| 9 | T | F | F | F | | | |
| 10 | T | F | F | T | | | |
| 11 | T | F | T | F | | | |
| 12 | T | F | T | T | T | T | T |
| 13 | T | T | F | F | | | |
| 14 | T | T | F | T | | | |
| 15 | T | T | T | F | | | |
| 16 | T | T | T | T | | | |

---

- The minimal multiple condition coverage test demands that besides the simple conditions and the decision also all composite conditions are tested against *true* and *false*
- As decisions can be hierarchically structured it is useful to consider this structure during testing
- This condition coverage technique takes into account the structure of decisions in a better way than the methods presented above, as all nesting levels of a compound decision are equally considered

## Control Flow Testing
## Minimal Multiple Condition Coverage

☐ Concerning a complete evaluation of decisions the execution of the test cases 1 and 16 results in a complete minimal multiple condition coverage (all conditions A, B, C, D, (A || B) and (C || D) and the decision ((A || B) && (C || D)) are tested against both logical values)

☐ Upon closer examination it can be recognized that these two test cases do not test the logic structure of the decision in a really useful way: If the decision incorrectly was ((A && B) || (C && D)), none of the two test cases would have detected this, although all operators would be faulty. For all conditions and the overall decision identical logical values would have appeared. The test cases are "blind" towards this fault

|    | A | B | C | D | A\|\|B | C\|\|D | (A\|\|B)&&(C\|\|D) |
|----|---|---|---|---|------|------|----------------|
| 1  | F | F | F | F | F | F | F |
| 2  | F | F | F | T |   |   |   |
| 3  | F | F | T | F |   |   |   |
| 4  | F | F | T | T |   |   |   |
| 5  | F | T | F | F |   |   |   |
| 6  | F | T | F | T |   |   |   |
| 7  | F | T | T | F |   |   |   |
| 8  | F | T | T | T |   |   |   |
| 9  | T | F | F | F |   |   |   |
| 10 | T | F | F | T |   |   |   |
| 11 | T | F | T | F |   |   |   |
| 12 | T | F | T | T |   |   |   |
| 13 | T | T | F | F |   |   |   |
| 14 | T | T | F | T |   |   |   |
| 15 | T | T | T | F |   |   |   |
| 16 | T | T | T | T | T | T | T |

---

## Control Flow Testing
## Minimal Multiple Condition Coverage

☐ Concerning an incomplete evaluation of decisions, e.g., the four test cases I, II, VI, and VII are required
  ▪ higher test costs
  ▪ better results

☐ If the decision incorrectly was ((A && B) || (C && D)), e.g., test case I would have proceeded differently. The conditions A, C, (A && B) and (C && D) would have been evaluated to *false*. The conditions B and D would not have been evaluated. The overall decision is *false*. The same result is obtained, but in a different way. The evaluation of the decision is broken off at other points which is a chance for the detection of faults

|     | A | B | C | D | A\|\|B | C\|\|D | (A\|\|B)&&(C\|\|D) |
|-----|---|---|---|---|------|------|----------------|
| I   | F | F | - | - | F | - | F |
| II  | F | T | F | F | T | F | F |
| III | F | T | F | T | T | T | T |
| IV  | F | T | T | - | T | T | T |
| V   | T | - | F | F | T | F | F |
| VI  | T | - | F | T | T | T | T |
| VII | T | - | T | - | T | T | T |

**Control Flow Testing**
**Modified condition/decision coverage**

☐ The modified condition/decision coverage requires test cases which demonstrate that every condition can influence the logical value of the overall decision independently of the other conditions

☐ The application of this method is required by the standard RTCA DO-178 B for flight critical software (level A)

☐ Basically the method aims at a test as extensive as possible with justifiable test costs

- The relation between the number of conditions and the required test cases is linear

- For the test of a decision with *n* conditions at least *n+1* test cases are required. The maximum number of test cases is *2n*

---

**Control Flow Testing**
**Modified Condition/Decision Coverage**

☐ Test of the condition B with the test cases 2 and 6

- Show **identical** logical values for the conditions A, C, and D

- **Differ** in the logical values of the condition B. In test case 2 condition B has the logical value *false*. In test case 6 condition B is *true*

- **Differ** in the overall result (test case 2 gives the overall result *false*, while in test case 6 the decision has the value *true*)

☐ Thus it is proven that the simple condition B can **independently** influence the logical value of the overall decision

☐ A corresponding situation is given for the test cases 2 and 10 concerning A, 9 and 10 concerning D, and 9 and 11 concerning C

| | A | B | C | D | A\|\|B | C\|\|D | (A\|\|B)&&(C\|\|D) |
|---|---|---|---|---|---|---|---|
| 1 | F | F | F | F | | | |
| 2 | F | F | F | T | F | T | F |
| 3 | F | F | T | F | | | |
| 4 | F | F | T | T | | | |
| 5 | F | T | F | F | | | |
| 6 | F | T | F | T | T | T | T |
| 7 | F | T | T | F | | | |
| 8 | F | T | T | T | | | |
| 9 | T | F | F | F | T | F | F |
| 10 | T | F | F | T | T | T | T |
| 11 | T | F | T | F | T | T | T |
| 12 | T | F | T | T | | | |
| 13 | T | T | F | F | | | |
| 14 | T | T | F | T | | | |
| 15 | T | T | T | F | | | |
| 16 | T | T | T | T | | | |

□ Concerning an incomplete evaluation of decisions it is necessary to modify the requirement, to retain the logical values of the respectively not tested conditions, while the logical values of the condition under test and the overall decision change

□ Now, for every simple condition the existence of a test case pair is required which

- Covers both logical values concerning this condition
- Covers both logical values concerning the overall decision
- Has identical logical values for all other simple conditions or was not evaluated at this point

□ Example: Test cases I and VII are testing the condition A. They give different logical values for the condition A and the overall decision and concerning the remaining conditions only have logical values if these were not evaluated in the respectively other test case

|  | A | B | C | D | A \|\| B | C \|\|D | (A \|\|B) && (C \|\| D) |
|---|---|---|---|---|---|---|---|
| I | F | F | - | - | F | - | F |
| II | F | T | F | F | T | F | F |
| III | F | T | F | T | T | T | T |
| IV | F | T | T | - | T | T | T |
| V | T | - | F | F | T | F | F |
| VI | T | - | F | T | T | T | T |
| VII | T | - | T | - | T | T | T |

---

□ A complete modified condition/decision coverage causes a branch coverage on the object code level
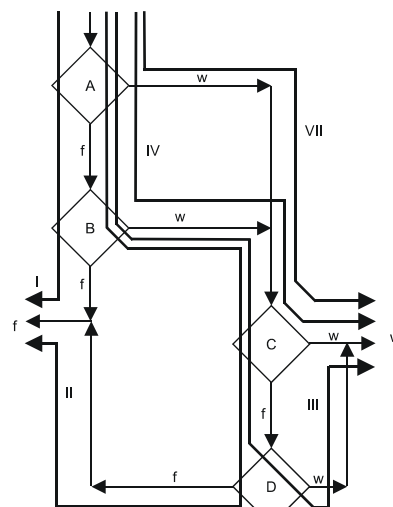
**Control Flow Testing**
**Modified Condition/Decision Coverage**

☐ But: Not every branch
coverage test on the object
code level causes a complete
modified condition/decision
coverage

---
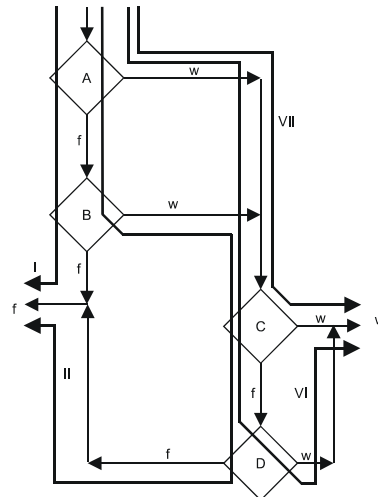
**Control Flow Testing**
**Multiple Condition Coverage**

☐ The multiple condition coverage
requires the test of all value
combinations of the conditions

☐ Benefits

- Very extensive test
- Subsumes the branch coverage
  test and all other condition
  coverage test techniques

☐ Disadvantages

- High test costs ($2^n$ test cases for a
  decision which is contains $n$
  simple conditions)
- Sometimes there exists no test
  data for certain combinations
  (e.g., because of incomplete
  evaluation of decisions or
  dependences between conditions)

|    | A | B | C | D | A\|\|B | C\|\|D | (A\|\|B)&&(C\|\|D) |
|----|---|---|---|---|------|------|-----------------|
| 1  | F | F | F | F | F    | F    | F               |
| 2  | F | F | F | T | F    | T    | F               |
| 3  | F | F | T | F | F    | T    | F               |
| 4  | F | F | T | T | F    | T    | F               |
| 5  | F | T | F | F | T    | F    | F               |
| 6  | F | T | F | T | T    | T    | T               |
| 7  | F | T | T | F | T    | T    | T               |
| 8  | F | T | T | T | T    | T    | T               |
| 9  | T | F | F | F | T    | F    | F               |
| 10 | T | F | F | T | T    | T    | T               |
| 11 | T | F | T | F | T    | T    | T               |
| 12 | T | F | T | T | T    | T    | T               |
| 13 | T | T | F | F | T    | F    | F               |
| 14 | T | T | F | T | T    | T    | T               |
| 15 | T | T | T | F | T    | T    | T               |
| 16 | T | T | T | T | T    | T    | T               |

☐ A program execution causes the execution of a program path which usually contains several branches and instructions

☐ Question: How can this be taken into account by a test technique?

© Prof. Dr. Liggesmeyer, 35

---

☐ A complete path coverage requires the execution of all different paths of the program to be tested
  ▪ A path p is a sequence of nodes $(i, n_1,.., n_m, j)$ in the control flow graph with the start node i and the end node j

☐ Disadvantages
  ▪ The path coverage test normally is not executable for real programs, as they can have an infinite number of paths. Assuming that the maximum value of an Integer-variable is 32767, we get $2^{32768}-1$ test paths for the operation *CountChars*. This is roughly $1{,}41 \cdot 10^{9864}$ paths. The required test time for a test that runs non-stop and executes 1000 paths per second would be $4{,}5 \cdot 10^{9853}$ years. For comparison: The age of the earth is estimated to roughly $4{,}5 \cdot 10^9$ years. Therefore, a complete path coverage test of the operation *CountChars* is absolutely impossible
  ▪ Often a fraction of the paths is not executable

☐ Question: How can the path coverage test be modified so that it is feasible?

© Prof. Dr. Liggesmeyer, 36

☐ The Structured path test distinguishes only paths that execute a loop not more than k times. This avoids the explosion of the number of paths caused by loops

☐ The structured path test with k=2 is called boundary interior coverage

☐ The boundary interior coverage differentiates the three cases *no loop execution, one loop execution* and *at least two loop executions*. This is especially useful due to the possible interactions between variables before, in and after the loop

---

☐ **Example**

The following test cases are necessary for a boundary interior test of the operation *CountChars*

▪ 1. Test case for the path outside of the loop
The execution with *totalnumber = INT_MAX* results in the non-execution of the loop body

test path: $n_{start}$, $n_1$, $n_2$, $n_{final}$

- 2. Boundary test cases
    - a. The execution with *totalnumber = 0* and the input of the character string *A1* causes the entering of the loop body, the execution of the *true*-branch of the selection, and subsequently the termination of the loop

        test path: $n_{start}, n_1, n_2, n_3, n_4, n_5, n_2, n_{final}$
    - b. The execution with *totalnumber = 0* and the input of the character string *B1* causes the entering of the loop body, the execution of the *false*-branch of the selection and subsequently the termination of the loop

        test path: $n_{start}, n_1, n_2, n_3, n_5, n_2, n_{final}$

---

- 3. Interior test cases
    - a. The execution with *totalnumber = 0* and the input of the character string *EIN1* causes three executions of the loop body. At the first two executions the *true*-branch of the selection is passed through. The third loop execution is irrelevant for the test

        test path: $n_{start}, n_1, n_2, n_3, n_4, n_5, n_2, n_3, n_4, n_5, n_2, n_3, n_5, n_2, n_{final}$
    - b. The execution with *totalnumber = 0* and the input of the character string *AH!* causes two executions of the loop body. At the first execution the *true*-branch of the selection is passed through. At the second execution the *false*-branch is passed. The exclamation mark terminates the execution of the loop which is allowed for the interior test after the second execution of the loop body

        test path: $n_{start}, n_1, n_2, n_3, n_4, n_5, n_2, n_3, n_5, n_2, n_{final}$

c. The execution with *totalnumber = 0* and the input of the character string *HH!* causes two executions of the loop body. At both executions the *false*-branch of the selection is passed through. The exclamation mark terminates the loop execution

test path: $n_{start}, n_1, n_2, n_3, n_5, n_2, n_3, n_5, n_2, n_{final}$

d. The execution with *totalnumber = 0* and the input of the character string *HA!* causes two executions of the loop body. At the first execution the *false*-branch of the selection is passed through. At the second execution the *true*-branch of the selection is passed through. The exclamation mark terminates the loop execution

test path: $n_{start}, n_1, n_2, n_3, n_5, n_2, n_3, n_4, n_5, n_2, n_{final}$

☐ The seven test cases are sufficient for the complete test of the loop according to the boundary interior criterion

---
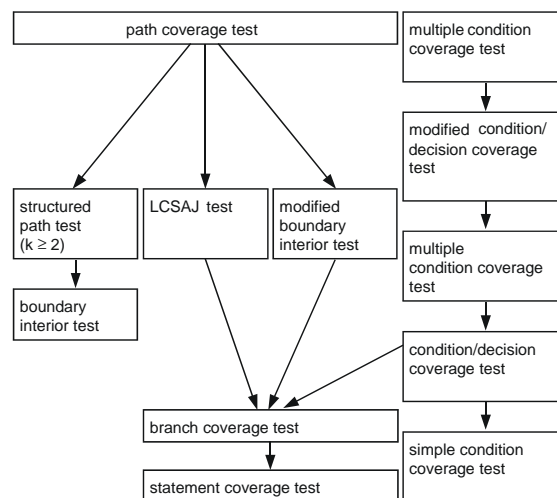
□ Data flow testing is based on the data flow. The basis is the control flow graph enhanced by data flow attributes

□ Accesses to variables are assigned to one of the classes

  ▪ write: definition *(def)*

  ▪ read: computational use *(c-use)*

  ▪ read: predicate use *(p-use)*

0101**seda**010100
software engineering dependability   © Prof. Dr. Liggesmeyer, 43

---

**Data Flow Testing**
**Control Flow Graph with Data Flow Attributes for CountChars**



```
void CountChars ( int &vowel number,
     int &total number)
{
  cin >> ch ;

    while ((ch >= 'A') && (ch <= 'Z')
        && (total number < INT_MAX))
    {

        total number = total number + 1;

        if ((ch == 'A') || (ch == 'E') ||
           (ch == 'I') || (ch == 'O') ||
           (ch == 'U'))
        {
            vowel number = vowel number + 1;
        }

    cin >> ch
  }

}
```

def (total number) — $n_{in}$
def (vowel number)

def (ch) — $n_1$

$n_2$
p-use (ch)
p-use (total number)
c-use (total number) — $n_3$
def (total number)

$n_4$
p-use (ch)

c-use (vowel number) — $n_5$
def (vowel number)

def (ch) — $n_6$

c-use (vowel number) — $n_{out}$
c-use (total number)

$n_{Start}$

$n_{Final}$

software engineering dependability   © Prof. Dr. Liggesmeyer, 44

☐ Example

- The instruction y = x+1; contains a c-use of the variable x, followed by a definition (def) of the variable y
- The instruction IF (x=0) THEN... contains a p-use of the variable x

☐ The all defs-criterion demands

- That every definition (all defs) of a variable is used at least once in a computation or a predicate. The objective of an assignment to a variable is that this value is used somewhere once again. The tests have to be chosen in such a way that this is tested at least once for every assignment to every variable

---

☐ All p-uses-test

- The all p-uses-test requires that every p-use that exists w.r.t. each definition is taken into account during testing

☐ All c-uses-test

- The all c-uses-test requires that every c-use that exists w.r.t. each definition is taken into account during testing

☐ All c-uses / some p-uses-test resp. all p-uses / some c-uses-test

- If no c-uses resp. p-uses exist for some variable definitions, it is required that at least one p-use resp. c-use is tested

☐ All uses-test

- All c-uses-test + All p-uses-test

---

□ Test path for the all p-uses-test
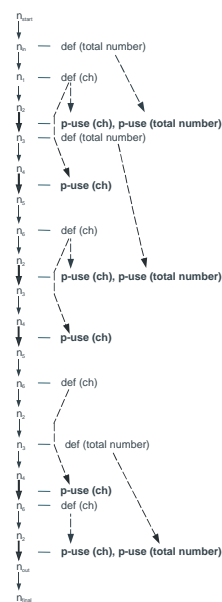
**Data Flow Testing**
**Relations of the Control Flow Tests (Subsumes Hierarchy)**



path coverage
all paths

data context coverage classified

all du-paths

required k-tuples-test

data context coverage simple

all uses

all c-uses/ some p-uses

all p-uses/ some c-uses

all c-uses

all defs

all p-uses

branch coverage all edges

statement coverage all nodes

seda
software engineering dependability

© Prof. Dr. Liggesmeyer, 49

---
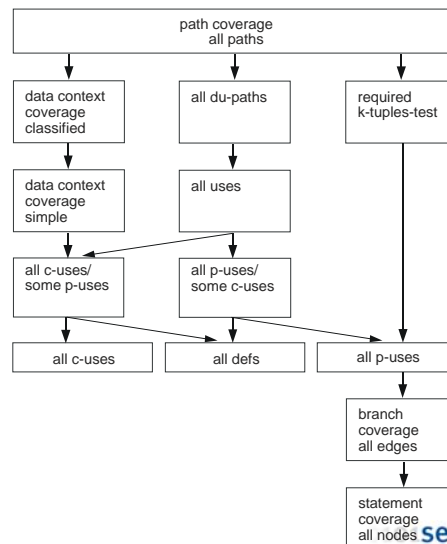
**Functional Test (Specification-based Test)**

☐ Determination of the adequacy and the completeness of the test cases as well as derivation of the test data and evaluation of the outputs based on the specification

- Benefits: Completeness w.r.t. the specification is checked. The test cases are systematically drawn from the specification
- Disadvantage: Information represented by the code is discarded

0101 seda 010100
software engineering dependability

© Prof. Dr. Liggesmeyer, 50

## Functional Test (Specification-based Test)



specification

test cases

completeness

image of the
specification

evaluate
outputs

tester

---
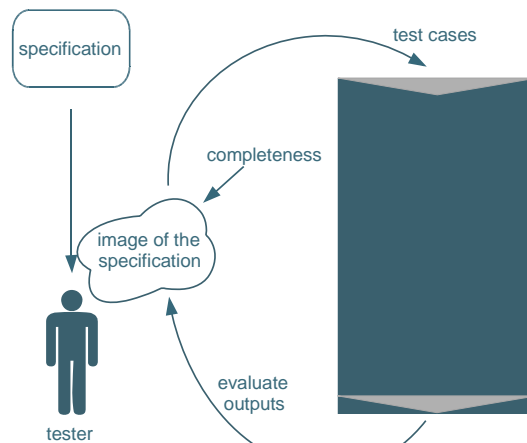
## Functional Test (Specification-based Test)
## Equivalence Partitioning

☐ Identification of equivalence classes based on the specification (Divide & Conquer)

☐ All Values from an equivalence class
  ▪ Shall cause an identical behavior and
  ▪ Shall belong to the same specified program function

☐ All specified program functions are tested with values from the equivalence class assigned to them

☐ Equivalence classes are also generated from the outputs

☐ If a value range is specified as the valid input domain for a particular input variable, this range represents a valid equivalence class which is enframed by invalid equivalence classes at its lower and upper boundary

☐ Example

- Input range: $1 \leq x \leq 99$
- One valid equivalence class: $1 \leq x \leq 99$
- Two invalid equivalence classes
  - $x < 1$
  - $x > 99$

---

☐ The equivalence classes are to be numbered. For the generation of test cases from the equivalence classes two rules have to be applied

- The test cases for valid equivalence classes are generated by the selection of test data from as many valid equivalence classes as possible
- The test cases for invalid equivalence classes are generated by the choice of a test date from an invalid equivalence class. It is combined with values which are extracted exclusively from valid equivalence classes

☐ Selection of the concrete test data from an equivalence class according to different criteria

☐ Often used: test of the equivalence class boundaries **(boundary value analysis)**

☐ A program for the inventory management of a shop is capable to register deliveries of wooden boards

- If wooden boards are delivered, the sort of the wood is entered
- The program knows the wood sorts Oak, Beech, and Pine
- Furthermore, the length is given in centimeters which is always between 100 and 500
- As delivered number a value between 1 and 9999 can be given
- In addition, the delivery gets an order number
- Every order number for wood deliveries begins with the letter H

---

Equivalence classes

| Input | Valid Equivalence Class | Invalid Equivalence Class |
|---|---|---|
| Sort | 1) Oak<br>2) Beech<br>3) Pine | 4) All others, e.g. steel |
| Length | 5) 100 <= Length <= 500 | 6) Length < 100<br>7) 500 < Length |
| Number | 8) 1 <= Number <= 9999 | 9) Number < 1<br>10) 9999 < Number |
| Order number | 11) First character is H | 12) First character is not H |

Test cases according to equivalence partitioning combined with boundary value analysis

| Test case | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Testes Equivalence Classes | 1, 5L, 8L, 11 | 2, 5U, 8U | 3 | 4 | 6U | 7L | 9U | 10L | 12 |
| Sort | **Oak** | **Beech** | **Pine** | **Steel** | Oak | Oak | Oak | Oak | Oak |
| Length | **100** | **500** | 200 | 200 | **99** | **501** | 200 | 200 | 200 |
| Number | **1** | **9999** | 100 | 100 | 100 | 100 | **0** | **10000** | 100 |
| Order number | **H1** | H2r | H54 | H54 | H54 | H54 | H54 | H54 | **J1** |
| | | | | | | | | | |

---

**Exercise**

☐ The class "triangle" contains the lengths of the triangle sides side1, side2 and side3 as integer-attributes. The operation "type ()" determines the type of the triangle on the basis of these side lengths. The following cases are differentiated

- No triangle: data error of the side lengths
- Equilateral
- Right-angled
- Isosceles
- Scalene

☐ The type right-angled is output with priority, i.e., if for example a scalene triangle is right-angled, not scalene but right-angled is output

□ Example: section of a specification

- Parameters
  - PORT_A: calling phone
  - PORT_B: called phone

- PORT_A identifies the connection from which a call is to be set up. The actual state of the call setup is globally available. Depending on this a new state arises after the evaluation of the transferred action. The delivered state is CUT, if the call setup was terminated, it is PENDING, if the call setup is in progress but not completed yet. It is READY, if the call setup was successfully completed. In this case PORT_B delivers the connection of the selected subscriber, otherwise the data content of PORT_B is undefined. A call setup requires the sequence OFFHOOK (DIGIT_N)* and the digit sequence must represent a valid number. ONHOOK always leads to the complete termination of the call. If TIMEOUT occurs, ONHOOK brings the software back into the initial state (CUT)

---
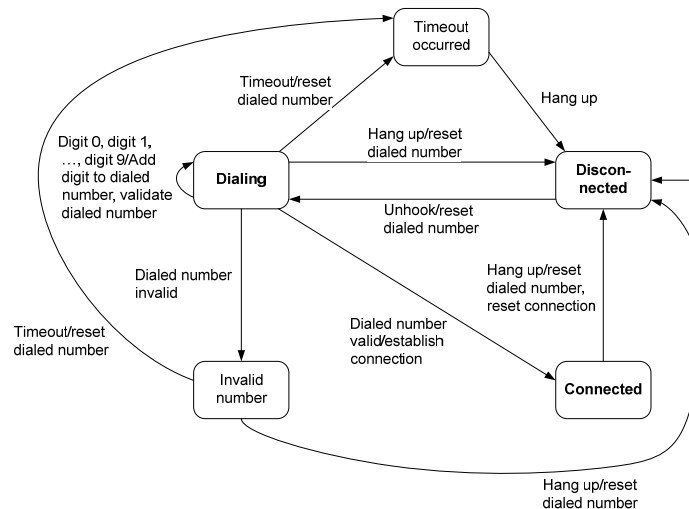
□ The minimal test strategy is to cover each state at least once.

□ A better solution is to cover each transition at least once, which leads, e.g., to the following test cases

- Cut, Offhook ➔ Pending, Onhook ➔ Cut
- Cut, Offhook ➔ Pending, Timeout ➔ Timeout occured, Onhook ➔ Cut
- Cut, Offhook ➔ Pending, Digit 0..9 ➔ Checking, Number incomplete ➔ Pending, Digit 0..9 ➔ Checking, ... , Number valid ➔ Ready, Onhook ➔ Cut
- Cut, Offhook ➔ Pending, Digit 0..9 ➔ Checking, Number incomplete ➔ Pending, Digit 0..9 ➔ Checking, ... , Number invalid ➔ Invalid number, Onhook ➔ Cut

□ Furthermore, it is useful to test all events if transitions can be initiated by more than one events. The result is a hierarchy of test techniques
**all states ⊆ all transitions ⊆ all events**

□ Important: Do not forget to test the failure treatment!

---
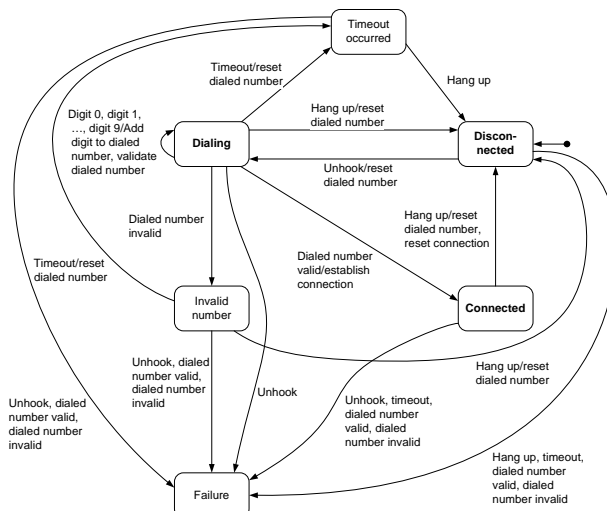
+ State-based tests can be used in unit and system testing.

+ It has widespread use particularly in technical applications such as industry automation, avionics, or the automotive industry.

- In state charts of large systems, there tends to be an explosion in the number of states, which leads to a considerable increase in transitions.

---

☐ According to /Beizer 90/ a transaction is a processing module from the view of a system user. Transactions consist of a sequence of processing steps.

☐ Representation forms for the notation of transaction flow:
  ▪ Flow diagram /Beizer 90/
  ▪ Sequence diagrams (Message Sequence Chart (MSC) in the object oriented method UML)

+ A good basis for generating test cases. It directly specifies possible test cases.

- Sequence diagrams display only one out of many different options.

---

☐ Decision tables or decision trees can be used as a basis for function-oriented tests.

+ They guarantee a certain test-completeness by way of their methodical approach.

- The size of this representation increases exponentially with the number of conditions.

☐ Example:

The following application specifies whether an e-commerce enterprise settles orders per bill. Whether the payment of a bill is possible is determined by if the customer is a new customer, if the order amount is greater than 1000€ and if he is a private customer. The three conditions result in eight combinations.

---

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Conditions | Customer = New Customer | N | N | N | N | Y | Y | Y | Y |
| | Order Value > 1000 € | N | N | Y | Y | N | N | Y | Y |
| | Customer Type = Private Customer | N | Y | N | Y | N | Y | N | Y |
| Action | Bill payment is possible | Y | Y | Y | Y | Y | Y | Y | N |

| Conditions | Customer = New Customer | N | - | - | Y |
|---|---|---|---|---|---|
| | Order Value > 1000 € | - | - | N | Y |
| | Customer Type = Private Customer | - | N | - | Y |
| Action | Bill payment is possible | Y | Y | Y | N |

---

**Every path from the root to a leaf of the tree corresponds to a test case.**
**Therefore, there would be four test cases.**

## Diversified Test

☐ Test of several software versions against each other
☐ Back to Back Test
- Implementation of 2, 3, or even more versions by independent programmers based on the same specifications
- Evaluation of the outputs by automated comparison
- Benefit: test execution (incl. checking of outputs) can be done automatically (saves time and money)
- Disadvantages: Multiple implementation is required. Faults occurring in all versions are not detected

☐ Mutations Test
- In fact no test method but a possibility to evaluate the efficiency (error detection rate) of test methods. Not explained here

☐ Regression Test
- Test of the present version against the previous version in order to identify undesired changes of the behavior (e.g. by faults introduced during modification and fault correction)
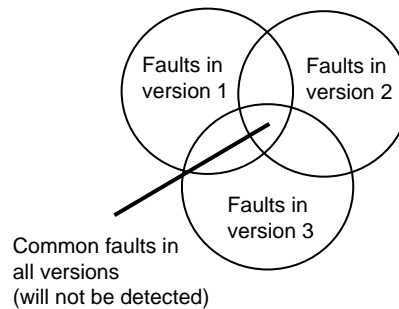
---

## Diversified Test
## Back to Back Test
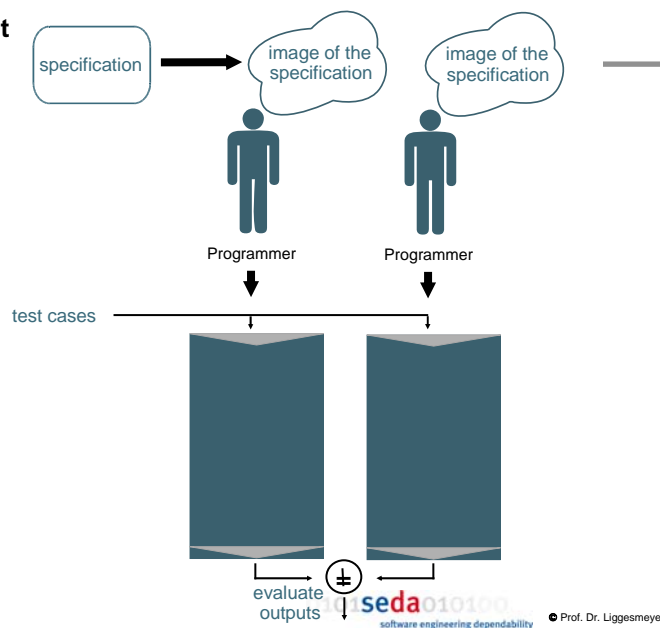
## Diversified Test
## Back to Back Test

□ The Back to Back Test requires the multiple realization of software modules based on identical specifications

□ The Back to Back Test is economically applicable, if outstanding safety and/or reliability requirements exist or an automatic evaluation of the outputs is desired or required
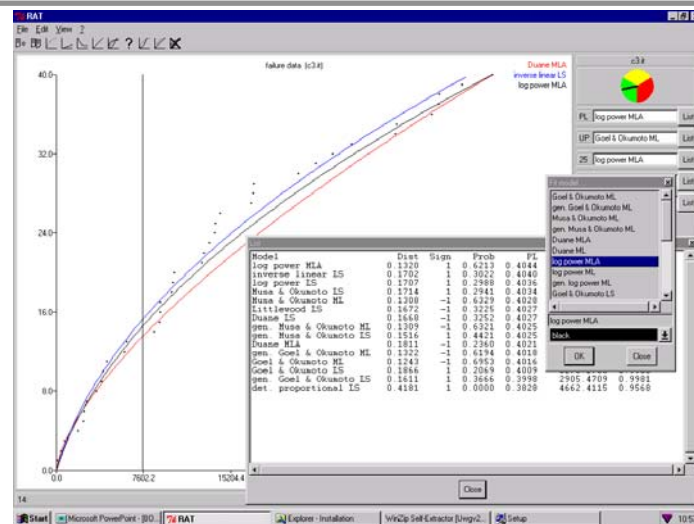
□ Common faults remain undetected

Faults in version 1

Faults in version 2

Faults in version 3

Common faults in all versions
(will not be detected)

---

## Some additional Dynamic Test „Techniques"

□ Boundary value analysis
- The boundary value analysis selects test data from boundaries

□ Special values testing / Error guessing
- Special values testing selects test cases based on the expertise of experienced testers => not acceptable as a single technique, but maybe ok in combination this other techniques, e.g. equivalence partitioning

□ Stochastic test, also random test
- Random test selects test data that fulfils certain statistical requirements. It is not identical with the ad hoc-procedure of unsystematic testing
- Random testing is usually used in combination with statistical techniques, that allow to determine and predict reliability on a quantitative basis. It may also be used as the test data generation technique for Back to Back testing

# Evaluation of data from a Stochastic Test