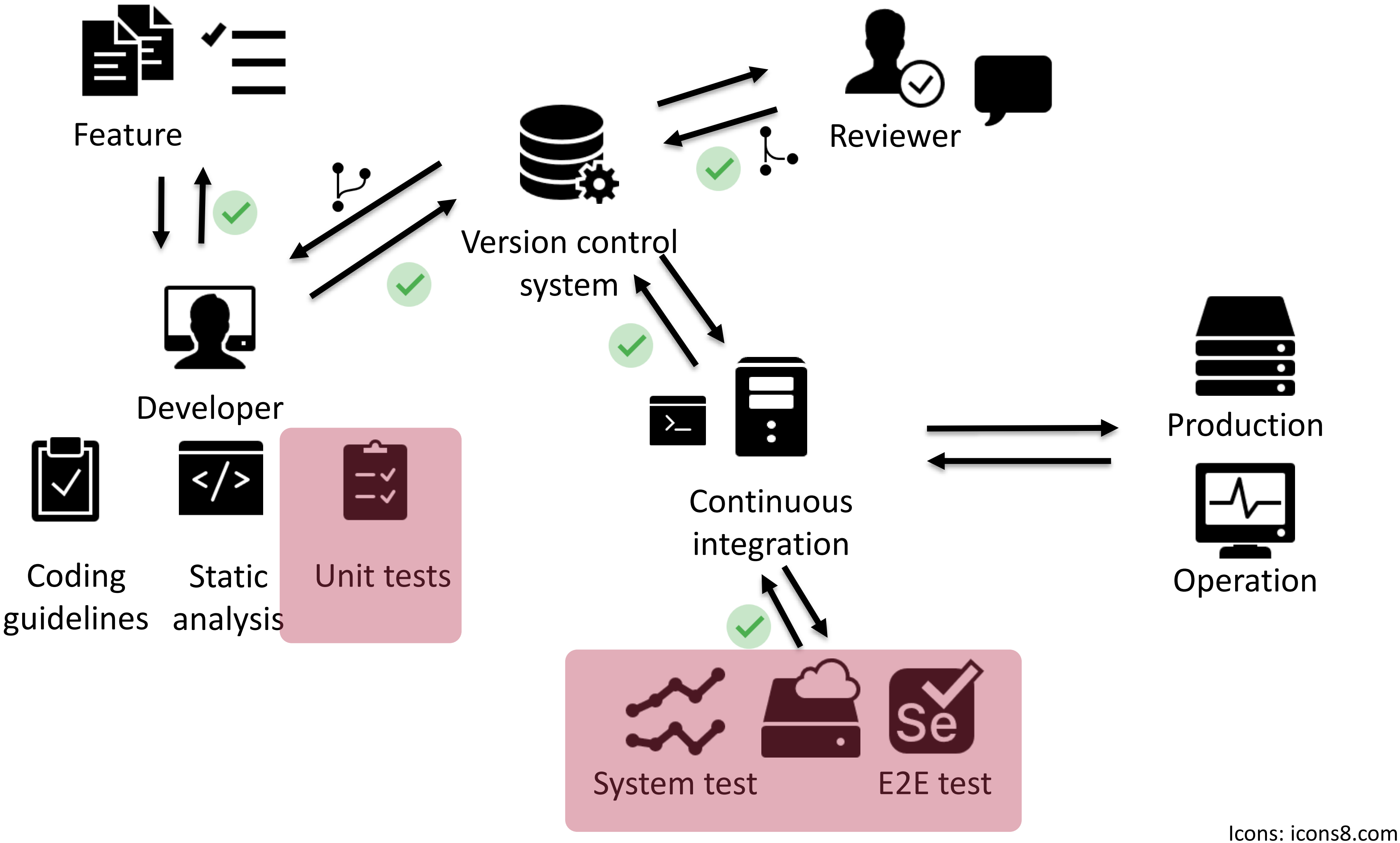
Test Design Techniques

Zoltán Micskei, István Majzik

Department of Measurement and Information Systems

MŰEGYETEM 1782

Overview

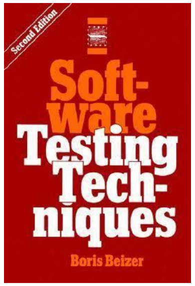


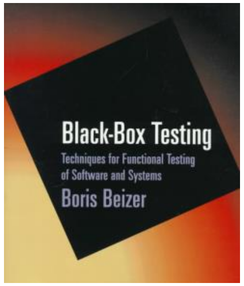
Icons: icons8.com

Why is test design important?

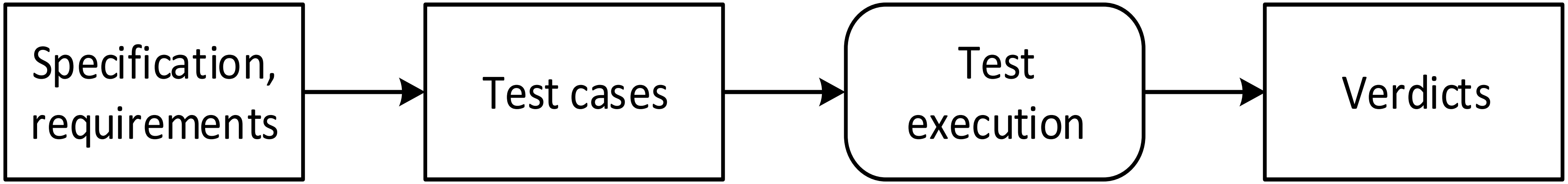
„More than the act of testing, the act of designing tests is one of the best bug preventers known."

Boris Beizer





Basic concepts



- SUT: system under test
- Test case
  - a set of test inputs, execution conditions, and expected results developed for a particular objective
- Test suite
- Test oracle
  - A principle or mechanism that helps you decide whether the program passed the test
- Verdict: result (pass / fail / error / inconclusive...)

Problems and tasks

- Test selection
  - What test inputs and test data to use?
- Oracle problem
  - How to get/create reliable oracle?
- Exit criteria
  - How long to test?
- Testability
  - Observability + controllability

Test design techniques

Goal: Select test cases based on test objectives

Specification-based

- SUT: black box
- Only spec. is known
- Testing specified functionality

Structure-based

- SUT: white box
- Inner structure known
- Testing based on internal behavior

Coverage metrics

- What % of testable elements have been tested
- Testable element
  - Specification-based: requirement, functionality…
  - Structure-based: statement, decision…
- Coverage criterion: X % for Y coverage metric
- This is not fault coverage!

How to use coverage metrics?

Evaluation (measure)	Selection (goal)
• Evaluate quality of existing tests • Find missing tests	• Design tests to satisfy criteria

SPECIFICATION-BASED TESTING

Test design techniques

Goal: Select test cases based on test objectives

Specification-based

- SUT: black box
- Only spec. is known
- Testing specified functionality

Structure-based

- SUT: white box
- Inner structure known
- Testing based on internal behavior

Specification-based techniques

Equivalence classes

Boundary values

Use case / user story

Combinatorial testing

Decision tables

…

Equivalence class partitioning

- Input and output equivalence classes:
  - Data that are expected to cover the same faults (cover the same part of the program)
  - Goal: Each equivalence class is represented by one test input (selected test data)

- Highly context-dependent
  - Needs to know the domain and the SUT!
  - Depends on the skills and experience of the tester

Selecting equivalence classes

- Selection uses heuristics
  - Initial: valid and invalid partitions
  - Next: refine partitions
- Typical heuristics:
  - Interval (e.g. 1-1000)
    - < min, min-max, >max
  - Set (e.g. RED, GREEN, BLUE)
    - Valid elements, invalid element
  - Specific format (e.g. first character is @)
    - Condition true, condition false
  - Custom (e.g. February from the months)

Deriving test cases from equiv. classes

- Combining equiv. classes of several inputs
- For valid (normal) equivalence classes:
  - test data should cover as much equivalence classes as possible
- For invalid equivalence classes:
  - first covering the each invalid equivalence class separately
  - then combining them systematically

EXERCISE Equivalence partitions

Requirement: The loan application shall be denied if the requested amount is larger than 1M Ft and the customer is a student, unless the amount is less than 3M Ft and the customer has repaid a previous loan (of any kind).

- Input parameters? Equivalence classes?
- Any questions regarding the requirement?

Specification-based techniques

Equivalence classes

Boundary values

Use case / user story

Combinatorial testing

Decision tables

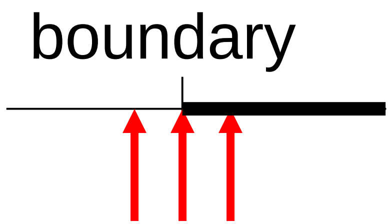
…

2. Boundary value analysis

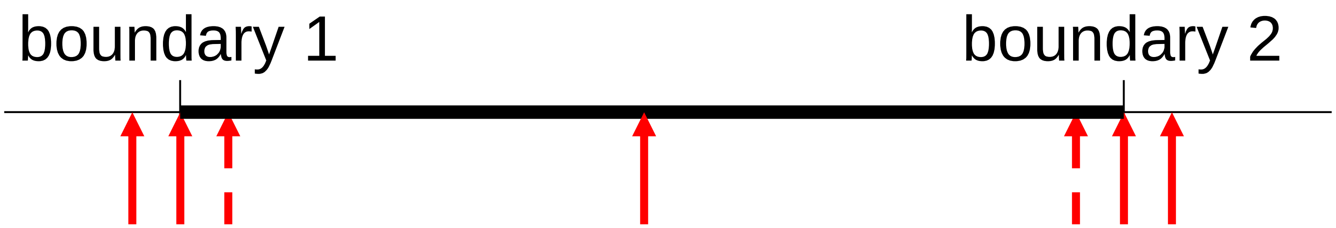
- Examining the boundaries of data partitions
  - Focusing on the boundaries of equivalence classes
  - Both input and output partitions
- Typical faults to be detected:
  - Faulty relational operators,
  - conditions in cycles,
  - size of data structures,
  - …

Typical test data for boundaries

- A boundary requires 3 tests:



- An interval requires 5-7 tests:



EXERCISE Boundary values

Requirement: If the robot detects that a human is closer than 4 meter, then it has to slow down, and if it is closer than 2 meter, then it has to stop.

- What values to use for testing?
- Any other questions regarding the requirement?

Specification-based techniques

- Equivalence classes
- Boundary values
- Use case / user story
- Combinatorial testing
- Decision tables
- …

Deriving tests from use cases

- Typical test cases:
  - 1 test for main path („happy path”, „mainstream”)
    - Oracle: checking post-conditions
  - Separate tests for each alternate path
  - Tests for violating pre-conditions

- Mainly higher levels (system, acceptance…)

STRUCTURE-BASED TESTING

Test design techniques

Goal: Select test cases based on test objectives

Specification-based

- SUT: black box
- Only spec. is known
- Testing specified functionality

Structure-based

- SUT: white box
- Inner structure known
- Testing based on internal behavior

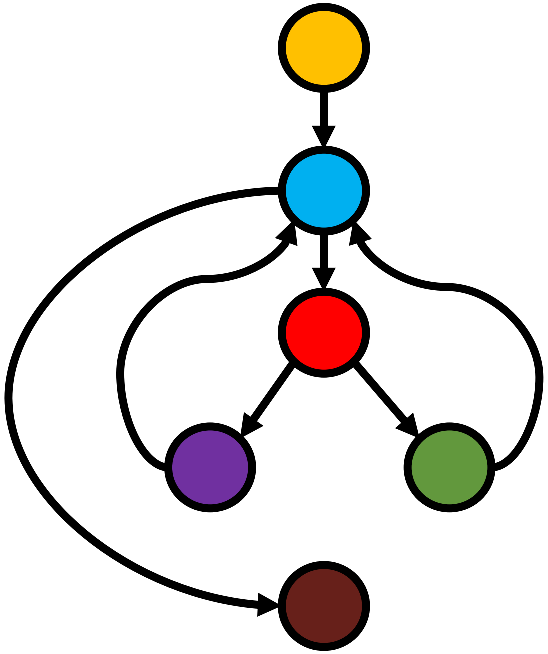
What is “internal structure”?

In case of code: structure of the code (CFG)

Source code:

```
int a = read();
while(a < 16) {
  if(a < 10) {
   a += 2;
  } else {
   a++;
  }
}
a = a * 2;
```

Control-flow graph:



Note: We will not go in details for constructing CFGs

Basic concepts

```
int t = 1;
Speed s = SLOW;

if (! started){
   start();
}

if (t > 10 && s == FAST){
   brake();
} else {
   accelerate();
}
```

Statement

Block

Condition

Decision

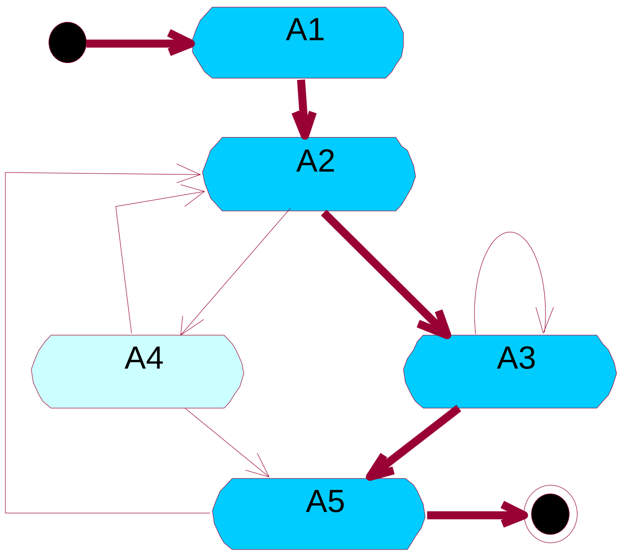
Branch

Basic concepts

- Statement
- Block
  - A sequence of one or more consecutive executable statements containing no branches
- Condition
  - Logical expression without logical operators (and, or…)
- Decision
  - A logical expression consisting of one or more conditions combined by logical operators
- Path
  - A sequence of events, e.g., executable statements, of a component typically from an entry point to an exit point.

1. Statement coverage

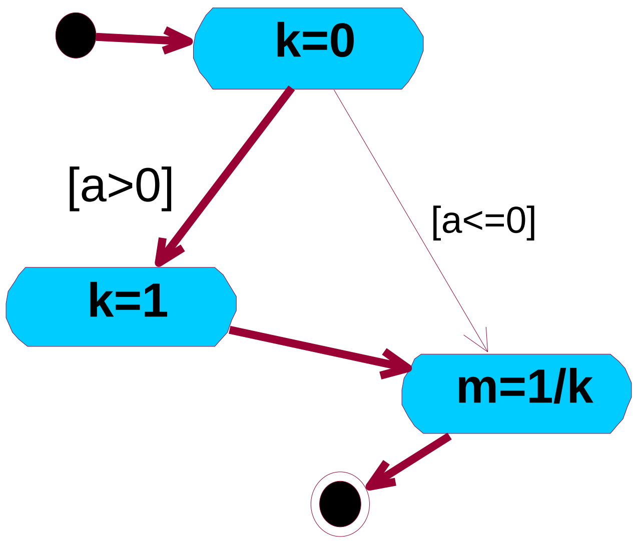
$$\frac{\text{Number of statements executed during testing}}{\text{Number of all statements}}$$



Statement coverage: 4/5 = 80%

Assessing statement coverage

All statement is executed at least once



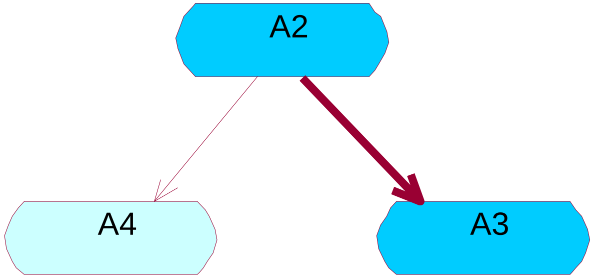
Statement coverage: 100%

BUT: [a<=0] branch missing!

Does not guarantee coverage of empty branches

2. Decision coverage

$$\frac{\text{Outcomes of decisions taken during testing}}{\text{Number of all possible outcomes}}$$



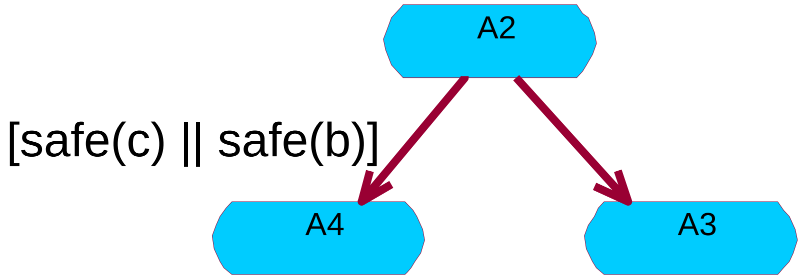
Decision coverage: 1/2 = 50%

How many outcomes can a decision have?

Assessing decision coverage

All statement is executed at least once

All outcomes of decisions are covered



100% decision coverage:

#	safe(c)	safe(b)
1	T	F
2	F	F

`safe(b) == True` missing!

Does not take into account all combinations of conditions!

Additional coverage criteria (see MSc)

- Condition Coverage
- Condition/Decision Coverage (C/DC)
- Modified Condition/Decision Coverage (MC/DC)
- Multiple Condition Coverage (MCC)
- Loop Coverage
- …
- All-Defs Coverage
- All-Uses Coverage
- …

EXERCISE Structure-based testing

```
int pow(int n, int k) {
  if (n < 0 || k < 0) {
    return -1;
  }
  int p = 1;
  for (int i = 0; i < k; i++) {
    p *= n;
  }
  return p;
}
```

Construct the CFG for the code!
Design test cases for:

- 100% statement coverage
- 100% decision coverage

Calculating coverage in practice

- Every tool uses different definitions
- Implementation
  - Instrument source/byte code
  - Adding instructions to count coverage

```
if (a > 10){
  CoveredBranch(1, true);
  b = 3;
} else {
  CoveredBranch(1, false);
  b = 5;
}
send(b);
```

See also: [Is bytecode instrumentation as good as source code instrumentation](), 2013.

Using test coverage criteria

- Can be used for:
  - Find not tested parts of the program
  - Measure “completeness” of test suite
  - Can be basis for exit criteria

- Cannot be used for:
  - Finding/testing missing or not implemented requirements
  - Only indirectly connected to code quality

Using test coverage criteria

- Experience from Microsoft
  - „Test suite with high code coverage and high assertion density is a good indicator for code quality.”
  - „Code coverage alone is generally not enough to ensure a good quality of unit tests and should be used with care.”
  - „The lack of code coverage to the contrary clearly indicates a risk, as many behaviors are untested.”

    (Source: „Parameterized Unit Testing with Microsoft Pex”)
- Related case studies:
  - „*Coverage Is Not Strongly Correlated with Test Suite Effectiveness*”, 2014. DOI: 10.1145/2568225.2568271
  - „*The Risks of Coverage-Directed Test Case Generation*”, 2015. DOI: 10.1109/TSE.2015.2421011

Test design techniques

- Specification and structure based techniques
  - Many orthogonal techniques
  - Every techniques need practice!

- Combination of techniques is useful:
  - Example (Microsoft report):

    specification based: 83% code coverage

    + exploratory: 86% code coverage

    + structural: 91% code coverage

Summary

Test design techniques

Goal: Select test cases based on test objectives

Specification-based	**Structure-based**
• SUT: black box • Only spec. is known • Testing specified functionality	• SUT: white box • Inner structure known • Testing based on internal behavior

Specification-based techniques

- Equivalence classes
- Boundary values
- Use case / user story
- Combinatorial testing
- Decision tables
- …

What is "internal structure"?

In case of code: structure of the code (CFG)

Source code:

```
int a = 1;
while(a < 16) {
  if(a < 10) {
    a += 2;
  } else {
    a++;
  }
}
a = a * 2;
```

Control-flow graph: