
# Introduction to Verification and Test of Embedded Systems

## SE767: Vérification & Test

Ulrich Kühne
ulrich.kuhne@telecom-paristech.fr
26/11/2018

# Objectives of this Course

- Understanding the role of test & verification in the development process
- Applying Test-Driven Design to embedded software
- Getting in touch with formal methods
- Understanding and writing a formal specification for a hardware module

# Course Structure

1. Introduction
   (this lecture)
2. Test-Driven Design of embedded software
   (lecture + exercise)
3. Embedded systems modeling
   (video lecture + exercise)
4. Introduction to formal methods
   (lecture + exercise)
5. Formal specification and verification of embedded hardware
   (lecture + exercise)

# Plan

# Why do we need to test and verify?

# Ariane 5

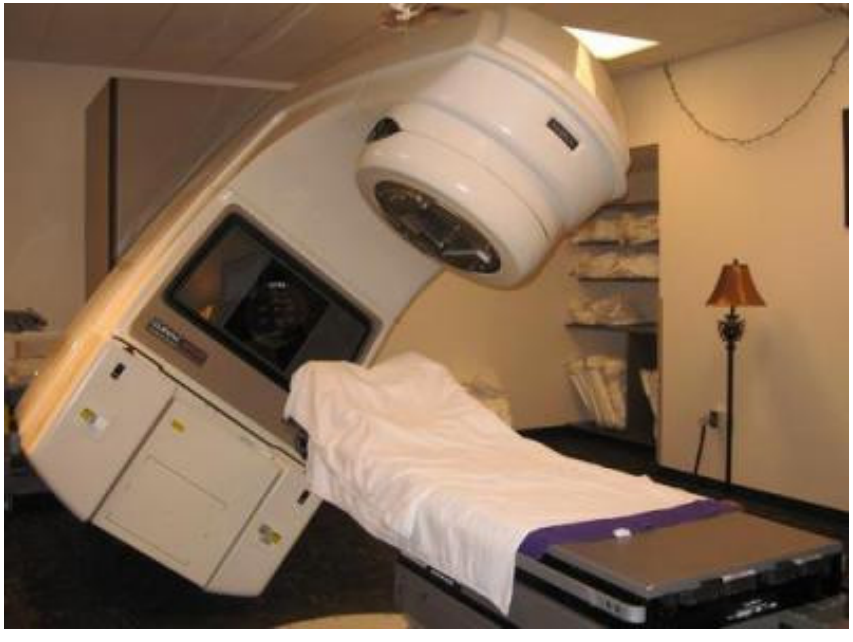
- Ariane 5 rocket explodes shortly after liftoff (1996)
- Integer overflow causing exception
- Legacy code (Ariane 4) used in flight control

# Mars Pathfinder

- Rover gets stuck on Mars (1997)
- Scheduling problem leads to permanent restarts
- Bug fixed on identical copy on earth, live update over the air

# Therac-25

- Therac-25 radiation therapy machine (1982)
- Software bug leads to race condition
- Patients exposed to lethal radiation dose



# Zune 30

- Zune-30 music players bricked during December 31, 2008
- Infinite loop in time service in loop (!) years
- Several New Year's Eve parties without music. . .

# Plan

# Goals of Testing & Verification

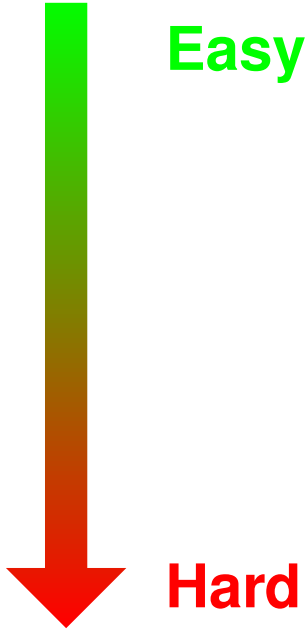
1. Find and eliminate bugs
2. Improve design quality
3. Reduce risk of user
4. Reduce risk of enterprise
5. Fulfill certification requirements
6. Improve performance
7. . . . ?

# Test & Verification Techniques

- Plug & pray
- Non-regression tests
- Unit testing
- Test-driven design
- Model-based design
- Formal methods
- Mathematical proofs

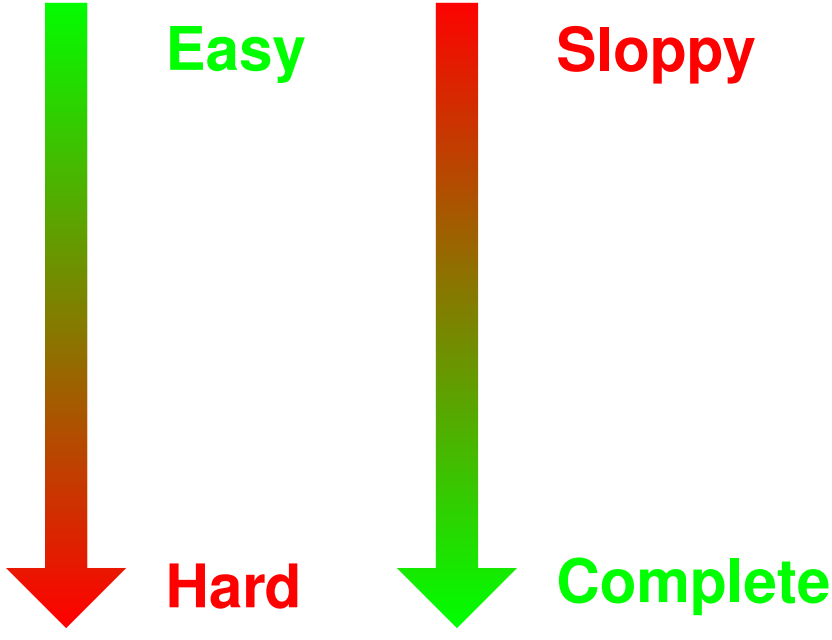
# Test & Verification Techniques

- Plug & pray
- Non-regression tests
- Unit testing
- Test-driven design
- Model-based design
- Formal methods
- Mathematical proofs



# Test & Verification Techniques

- Plug & pray
- Non-regression tests
- Unit testing
- Test-driven design
- Model-based design
- Formal methods
- Mathematical proofs



# Validation in the Development Cycle

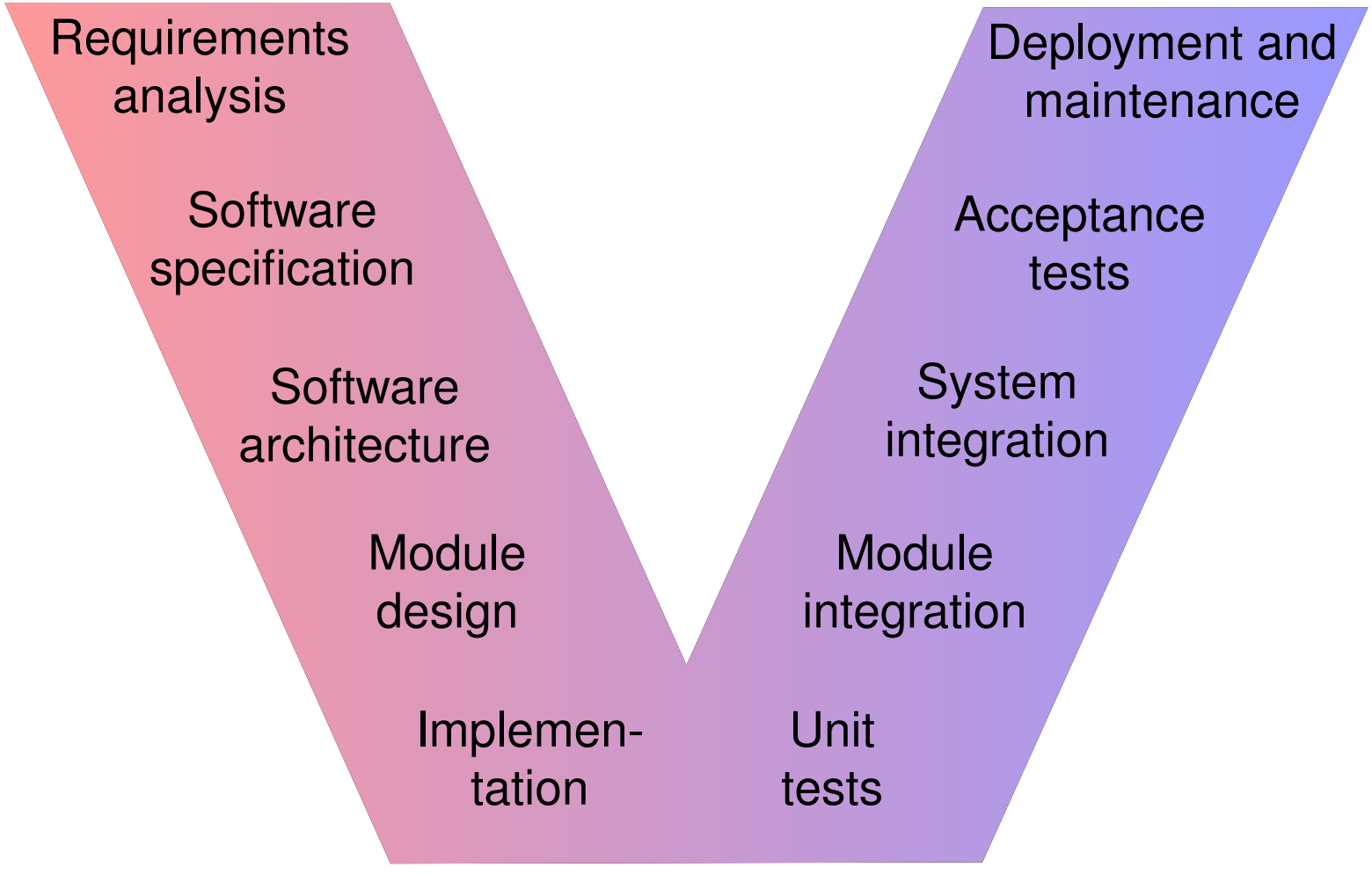
When should validation take place?

# Validation in the Development Cycle

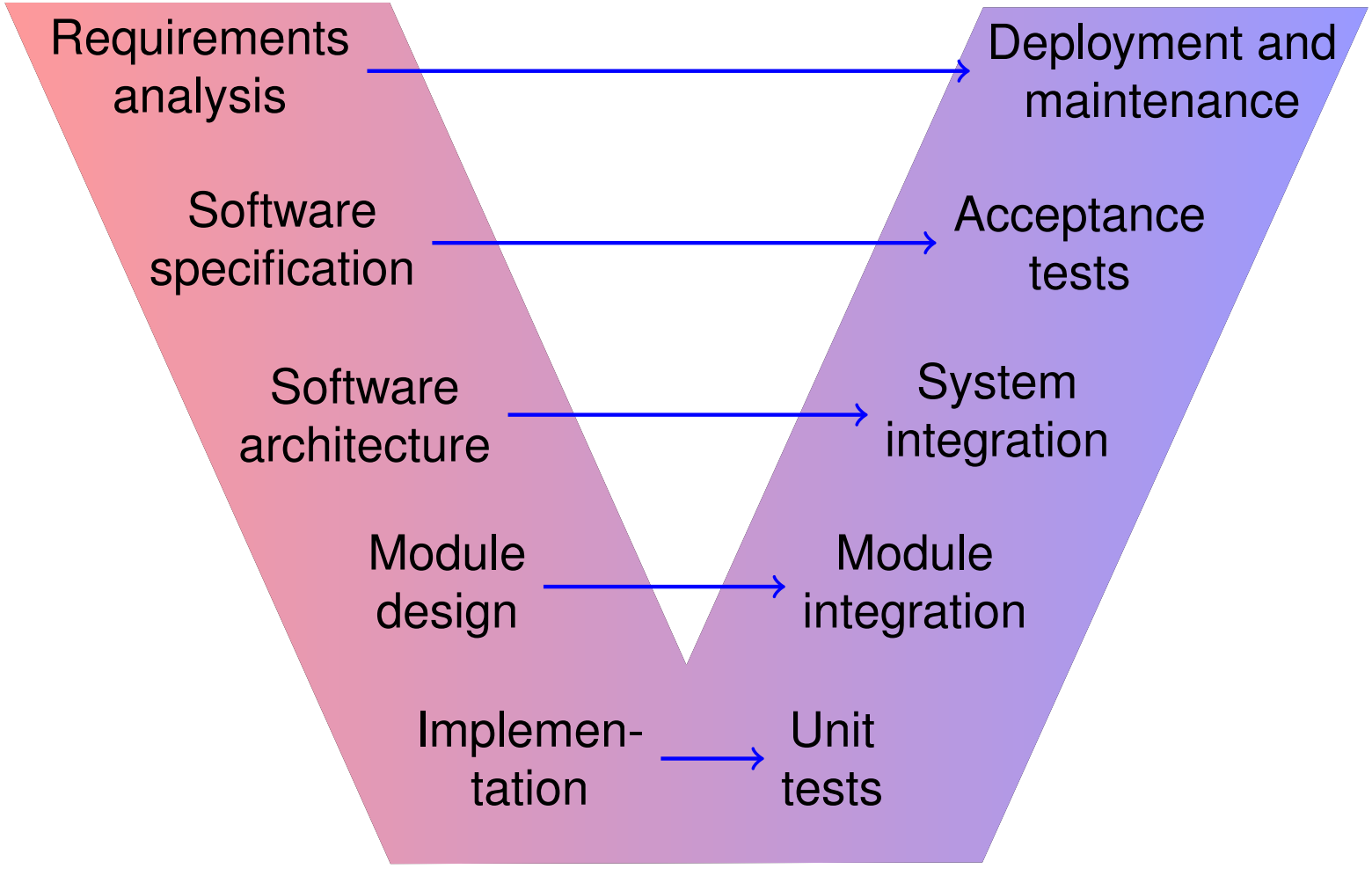

When should validation take place?

- As early as possible
- As often as necessary

# The V-Model

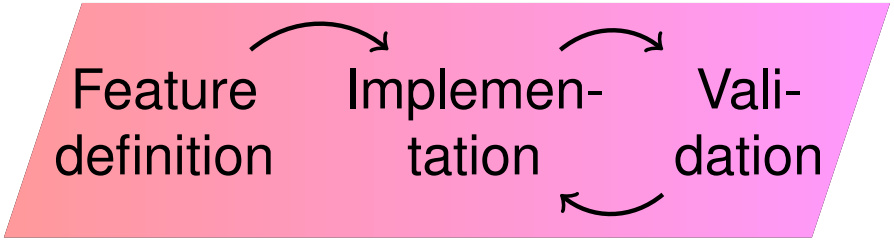

Requirements analysis
Software specification
Software architecture
Module design
Implementation
Unit tests
Module integration
System integration
Acceptance tests
Deployment and maintenance

# The V-Model


Requirements analysis
Deployment and maintenance
Software specification
Acceptance tests
Software architecture
System integration
Module design
Module integration
Implemen-tation
Unit tests

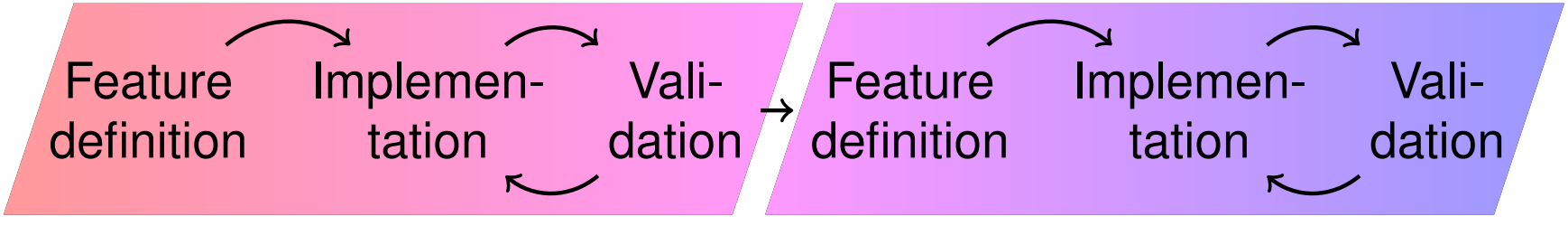
# Agile Design

- Continuous integration
- Break long validation cycles
- One feature (functionality) at a time



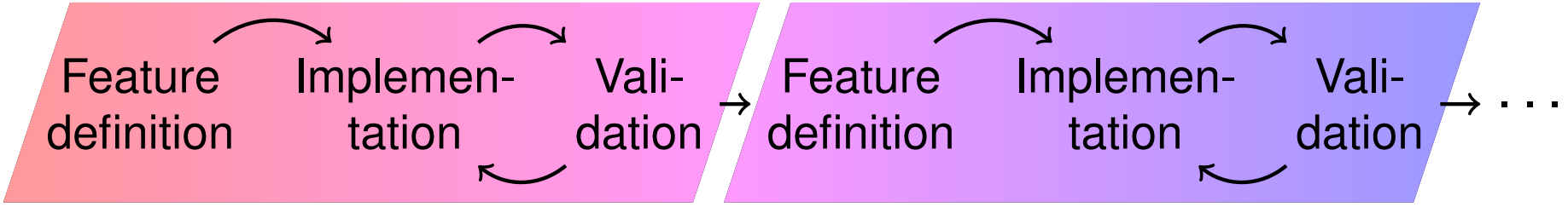
# Agile Design

- Continuous integration
- Break long validation cycles
- One feature (functionality) at a time



# Agile Design

- Continuous integration
- Break long validation cycles
- One feature (functionality) at a time



# What to Test?

How to define our test cases?

# What to Test?


How to define our test cases?

- Depends on abstraction level:
  Module tests, integration tests, acceptance tests, ...
- Depends on test strategy:
  Functional vs. structural testing
- Depends on test objectives:
  Bug hunting, coverage, performance, stress testing, ...
- Depends on context:
  Certification and safety norms (e.g. EN 50128)

# Functional vs. Structural Testing

## Functional Testing

- Black box testing
- Driven by specification
- Goal: Cover all specified functionality

## Structural Testing

- White box testing
- Driven by code structure
- Goal: Achieve structural coverage metrics

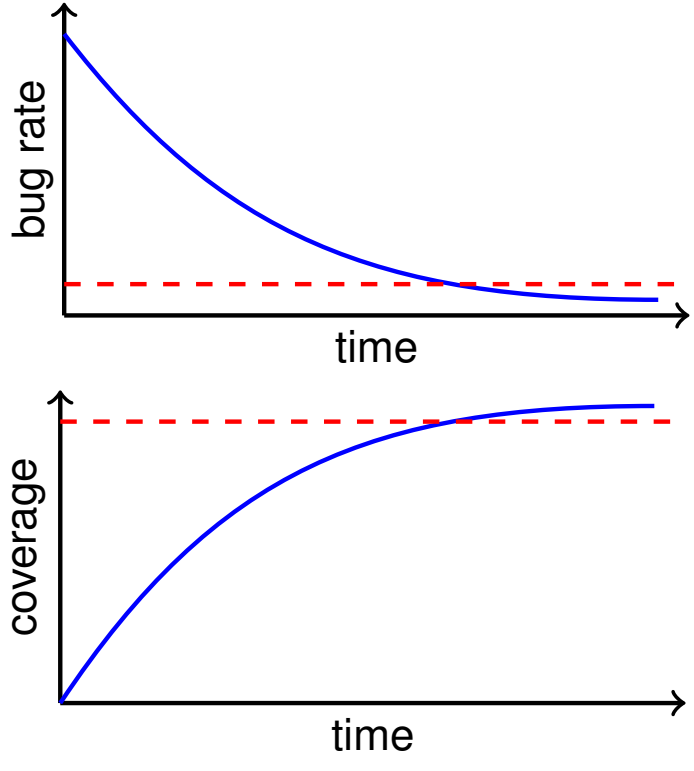
# Success Criteria


When do we stop testing?

# Success Criteria

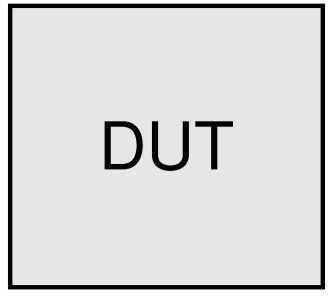

When do we stop testing?

- Boss says to stop (time budget)
- Bug rate below threshold
- Coverage above threshold

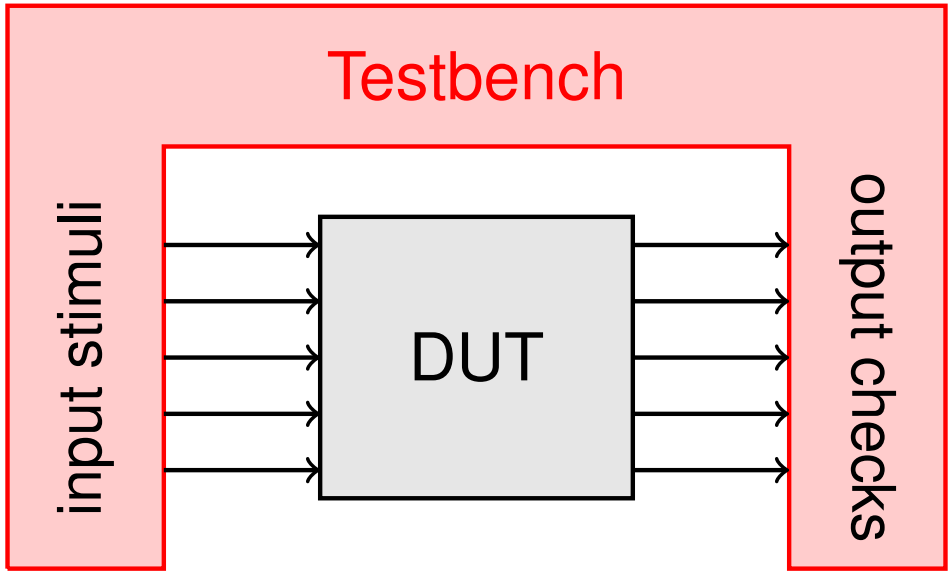


# Plan

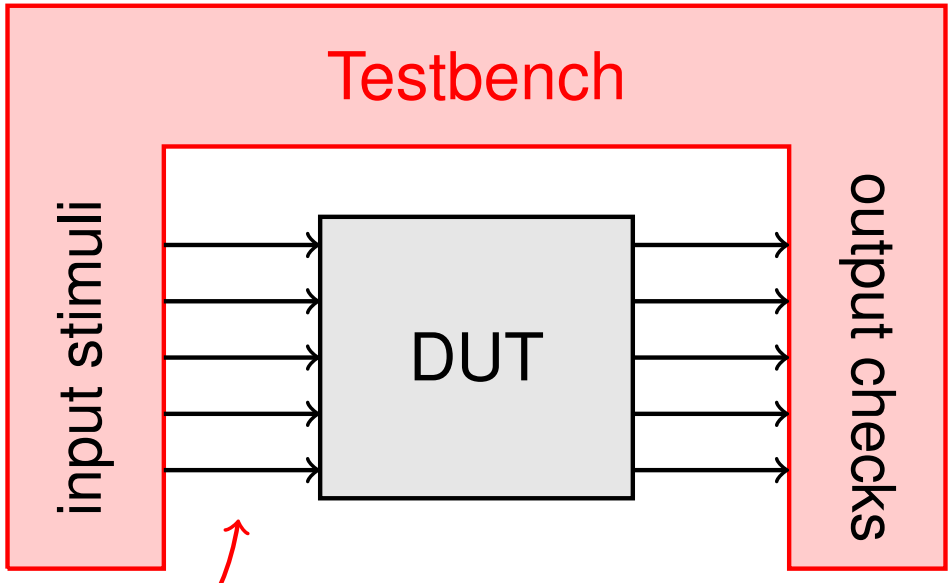
# Test Coverage



# Test Coverage



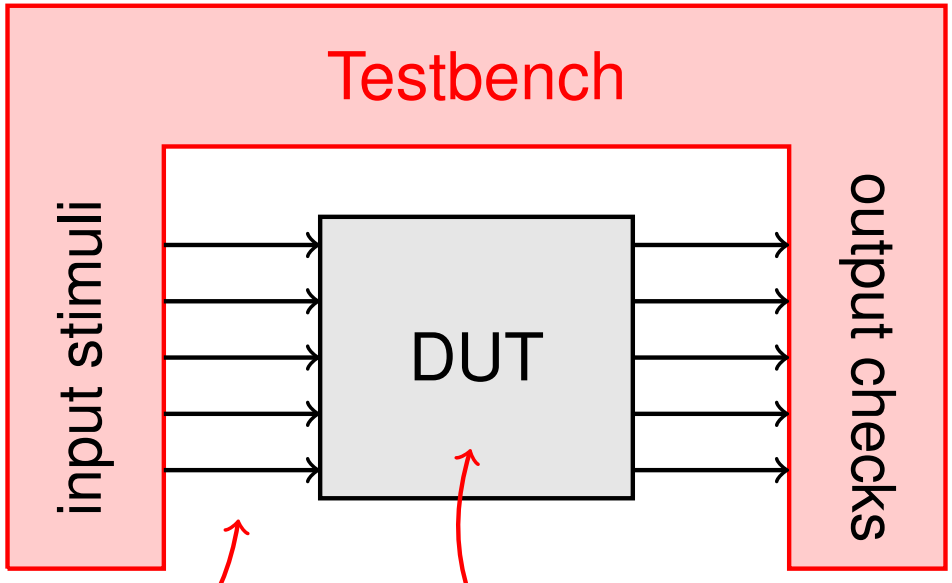
# Test Coverage




## Input Coverage

- Full coverage
- Boundary values
- Equivalence classes

# Test Coverage




## Input Coverage

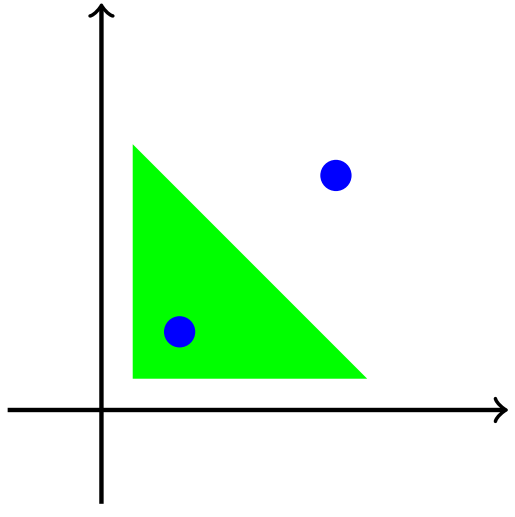
- Full coverage
- Boundary values
- Equivalence classes

## Code Coverage

- Statement coverage
- Branch coverage
- Path coverage

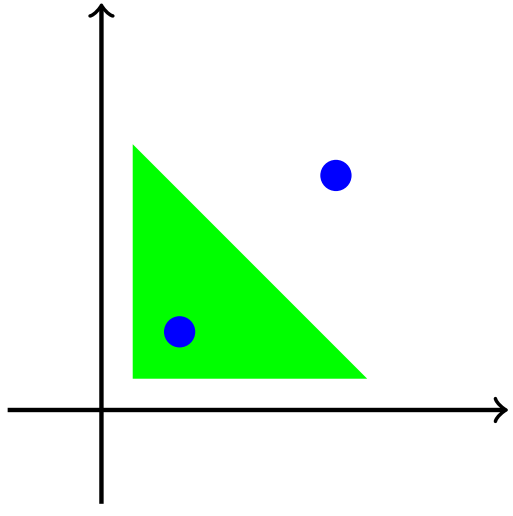
# Input Coverage: Equivalence Partitioning

- Black box approach
- Cover all interesting cases of input stimuli
- Determine equivalence classes that should entail the same behavior
- Select one test case for each class



# Input Coverage: Equivalence Partitioning

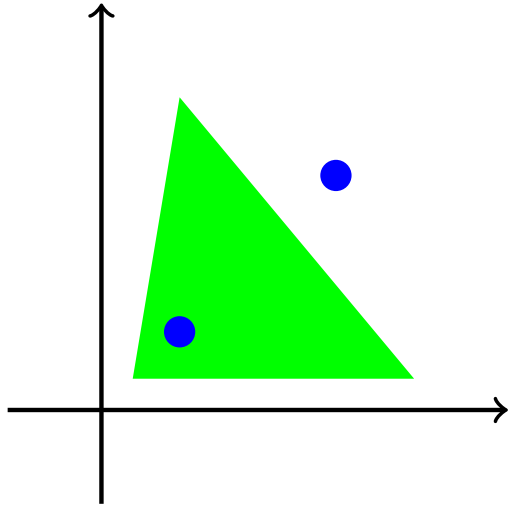
- Black box approach
- Cover all interesting cases of input stimuli
- Determine equivalence classes that should entail the same behavior
- Select one test case for each class



Are the equivalence classes consistent with the code? ?

# Input Coverage: Equivalence Partitioning

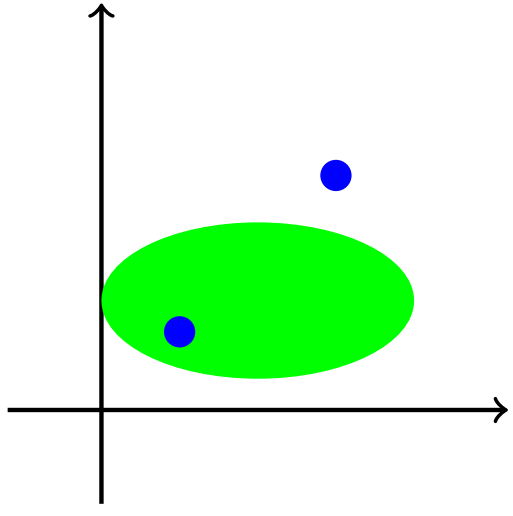
- Black box approach
- Cover all interesting cases of input stimuli
- Determine equivalence classes that should entail the same behavior
- Select one test case for each class



Are the equivalence classes consistent with the code? ?

# Input Coverage: Equivalence Partitioning

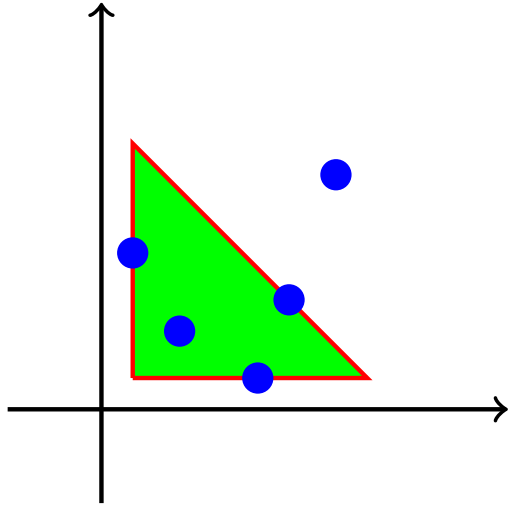
- Black box approach
- Cover all interesting cases of input stimuli
- Determine equivalence classes that should entail the same behavior
- Select one test case for each class



Are the equivalence classes consistent with the code? ?

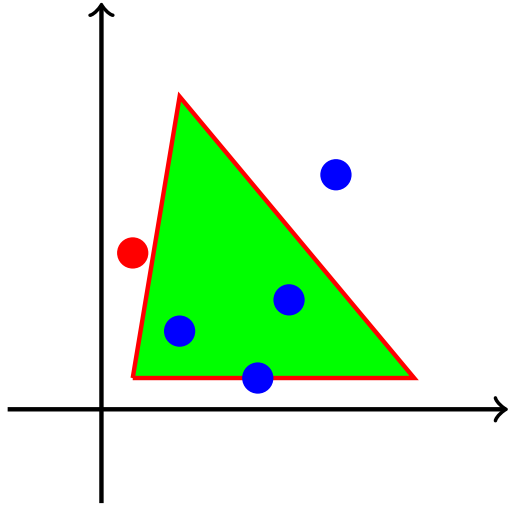
# Boundary Conditions

- Test values close to and on boundaries
- Reveals bugs due to *off-by-one* mistakes
- Test negative (i.e. failing) results



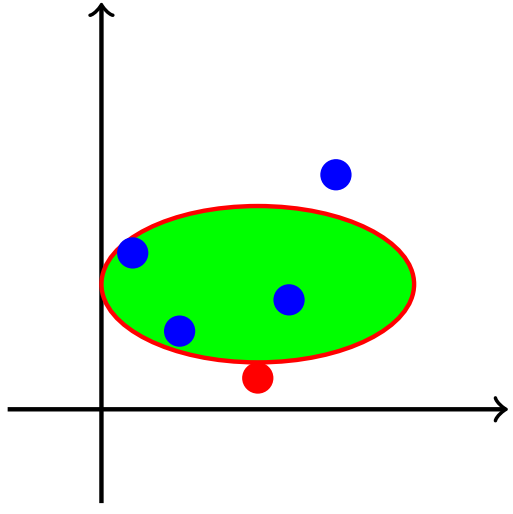
# Boundary Conditions

- Test values close to and on boundaries
- Reveals bugs due to *off-by-one* mistakes
- Test negative (i.e. failing) results



# Boundary Conditions

- Test values close to and on boundaries
- Reveals bugs due to *off-by-one* mistakes
- Test negative (i.e. failing) results



# Code Coverage

- Statement coverage
- Branch coverage
  - Decision coverage
  - Condition coverage
  - Condition/decision coverage
  - Modified condition/decision coverage (MC/DC)
  - Compound condition coverage
- Path coverage

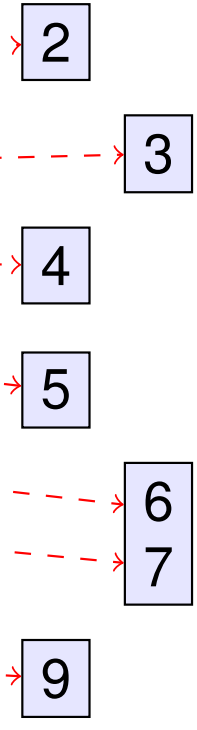
100% path cov. $\Rightarrow$ 100% decision cov. $\Rightarrow$ 100% statement cov.

# Control Flow Graph

```
01: int square(int x){
02:     if (x <= 0)
03:         x = -x;
04:     int s = 0, a = 1;
05:     for (int i=0; i<x; ++i){
06:         s += a;
07:         a += 2;
08:     }
09:     return s;
10: }
```

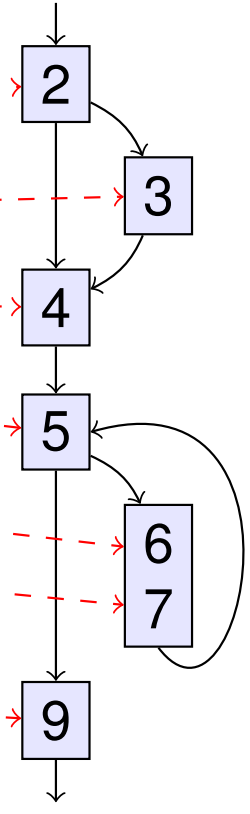
# Control Flow Graph

```
01: int square(int x){
02:     if (x <= 0)
03:         x = -x;
04:     int s = 0, a = 1;
05:     for (int i=0; i<x; ++i){
06:         s += a;
07:         a += 2;
08:     }
09:     return s;
10: }
```



# Control Flow Graph

```
01: int square(int x){
02:     if (x <= 0)
03:         x = -x;
04:     int s = 0, a = 1;
05:     for (int i=0; i<x; ++i){
06:         s += a;
07:         a += 2;
08:     }
09:     return s;
10: }
```



# Branch Coverage

```
1:  x = 0;
2:  if (a && (b || c))
3:      x = 1;
```

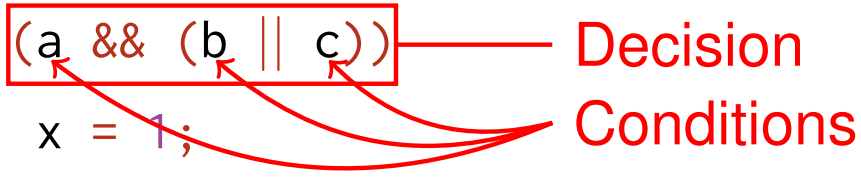
# Branch Coverage

```
1:  x = 0;
2:  if (a && (b || c))
3:      x = 1;
```

Decision

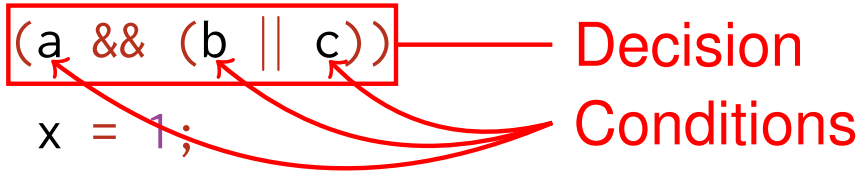
# Branch Coverage

```
1:  x = 0;
2:  if (a && (b || c))
3:      x = 1;
```



# Branch Coverage

```
1:  x = 0;
2:  if (a && (b || c))
3:      x = 1;
```




**Decision coverage** Every decision has taken all possible values at least once.

**Condition coverage** Every condition has taken all possible values at least once.

**Condition/decision coverage** Combination of decision and condition coverage.

**Modified condition/decision coverage** Condition/decision coverage plus every condition has been shown to independently affect the decision's outcome.

# Let's Cover...

```
1:  x = 0;
2:  if (a && (b || c))
3:      x = 1;
```

| Tests | Statement | Decision | Condition | C/D | MC/DC |
|---|---|---|---|---|---|
| $\{abc\}$ | | | | | |
| $\{\bar{a}bc, abc\}$ | | | | | |
| $\{a\bar{b}\bar{c}, \bar{a}bc\}$ | | | | | |
| $\{ab\bar{c}, \bar{a}\bar{b}c\}$ | | | | | |
| $\{\bar{a}bc, \bar{a}\bar{b}c, a\bar{b}\bar{c}, ab\bar{c}\}$ | | | | | |

# Let's Cover...

```
1:  x = 0;
2:  if (a && (b || c))
3:      x = 1;
```

| Tests | Statement | Decision | Condition | C/D | MC/DC |
|---|---|---|---|---|---|
| $\{abc\}$ | yes | no | no | no | no |
| $\{\bar{a}bc, abc\}$ | | | | | |
| $\{ab\bar{c}, \bar{a}bc\}$ | | | | | |
| $\{ab\bar{c}, \bar{a}\bar{b}c\}$ | | | | | |
| $\{\bar{a}bc, \bar{a}\bar{b}c, a\bar{b}\bar{c}, ab\bar{c}\}$ | | | | | |

# Let's Cover...

```
1:  x = 0;
2:  if (a && (b || c))
3:      x = 1;
```

| Tests | Statement | Decision | Condition | C/D | MC/DC |
|---|---|---|---|---|---|
| $\{abc\}$ | yes | no | no | no | no |
| $\{\bar{a}bc, abc\}$ | yes | yes | no | no | no |
| $\{ab\bar{c}, \bar{a}bc\}$ | | | | | |
| $\{ab\bar{c}, \bar{a}\bar{b}c\}$ | | | | | |
| $\{a\bar{b}c, \bar{a}\bar{b}c, a\bar{b}\bar{c}, ab\bar{c}\}$ | | | | | |

# Let's Cover...

```
1:  x = 0;
2:  if (a && (b || c))
3:      x = 1;
```

| Tests | Statement | Decision | Condition | C/D | MC/DC |
|---|---|---|---|---|---|
| $\{abc\}$ | yes | no | no | no | no |
| $\{\bar{a}bc, abc\}$ | yes | yes | no | no | no |
| $\{a\bar{b}\bar{c}, \bar{a}bc\}$ | no | no | yes | no | no |
| $\{a\bar{b}\bar{c}, \bar{a}\bar{b}c\}$ | | | | | |
| $\{\bar{a}\bar{b}c, \bar{a}\bar{b}c, a\bar{b}\bar{c}, ab\bar{c}\}$ | | | | | |

# Let's Cover...

```
1:  x = 0;
2:  if (a && (b || c))
3:      x = 1;
```

| Tests | Statement | Decision | Condition | C/D | MC/DC |
|---|---|---|---|---|---|
| $\{abc\}$ | yes | no | no | no | no |
| $\{\bar{a}bc, abc\}$ | yes | yes | no | no | no |
| $\{a\bar{b}\bar{c}, \bar{a}bc\}$ | no | no | yes | no | no |
| $\{a\bar{b}c, \bar{a}\bar{b}c\}$ | yes | yes | yes | yes | no |
| $\{\bar{a}bc, \bar{a}\bar{b}c, a\bar{b}\bar{c}, ab\bar{c}\}$ | | | | | |

# Let's Cover...

```
1:  x = 0;
2:  if (a && (b || c))
3:      x = 1;
```

| Tests | Statement | Decision | Condition | C/D | MC/DC |
|---|---|---|---|---|---|
| $\{abc\}$ | yes | no | no | no | no |
| $\{\bar{a}bc, abc\}$ | yes | yes | no | no | no |
| $\{a\bar{b}\bar{c}, \bar{a}bc\}$ | no | no | yes | no | no |
| $\{a\bar{b}\bar{c}, \bar{a}\bar{b}c\}$ | yes | yes | yes | yes | no |
| $\{\bar{a}bc, \bar{a}\bar{b}c, a\bar{b}\bar{c}, ab\bar{c}\}$ | yes | yes | yes | yes | yes |

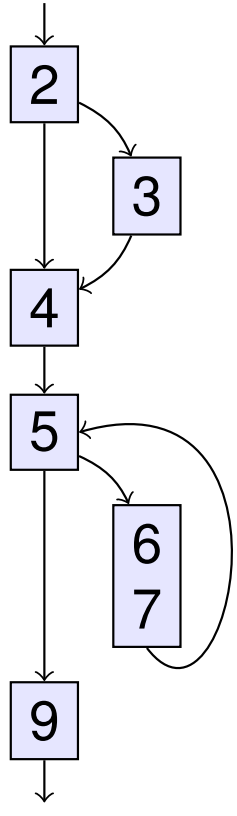
# Let's Cover...

```
1:  x = 0;
2:  if (a && (b || c))
3:      x = 1;
```

| Tests | Statement | Decision | Condition | C/D | MC/DC |
|---|---|---|---|---|---|
| $\{abc\}$ | yes | no | no | no | no |
| $\{\bar{a}bc, abc\}$ | yes | yes | no | no | no |
| $\{a\bar{b}\bar{c}, \bar{a}bc\}$ | no | no | yes | no | no |
| $\{a\bar{b}c, \bar{a}\bar{b}c\}$ | yes | yes | yes | yes | no |
| $\{\bar{a}bc, \bar{a}\bar{b}c, a\bar{b}\bar{c}, ab\bar{c}\}$ | yes | yes | yes | yes | yes |

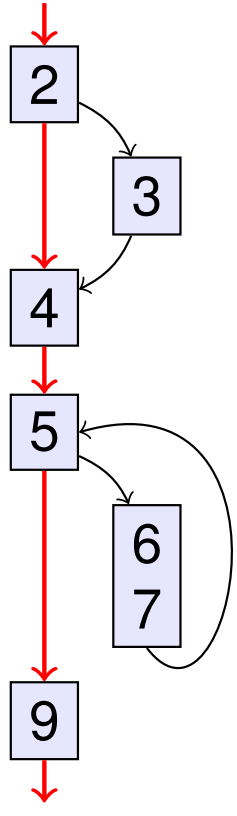
# Path Coverage

- Every execution path of the program has been explored



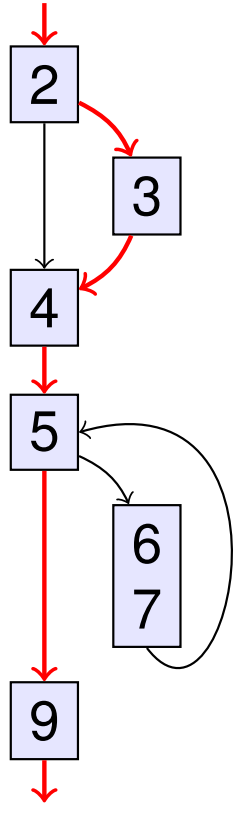
# Path Coverage

- Every execution path of the program has been explored



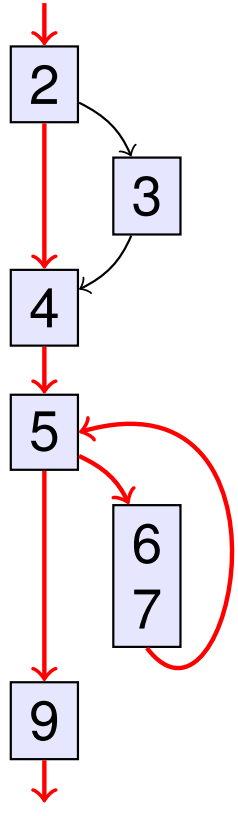
# Path Coverage

- Every execution path of the program has been explored



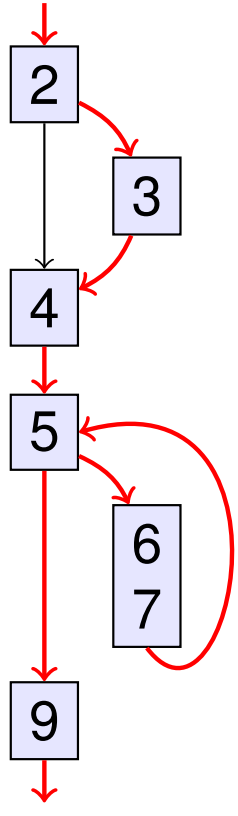
# Path Coverage

- Every execution path of the program has been explored



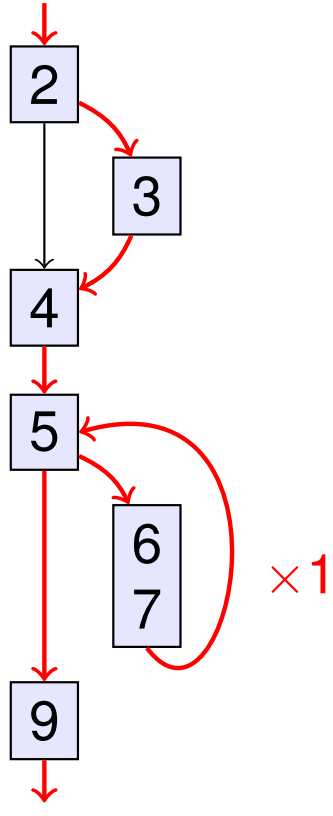
# Path Coverage

- Every execution path of the program has been explored



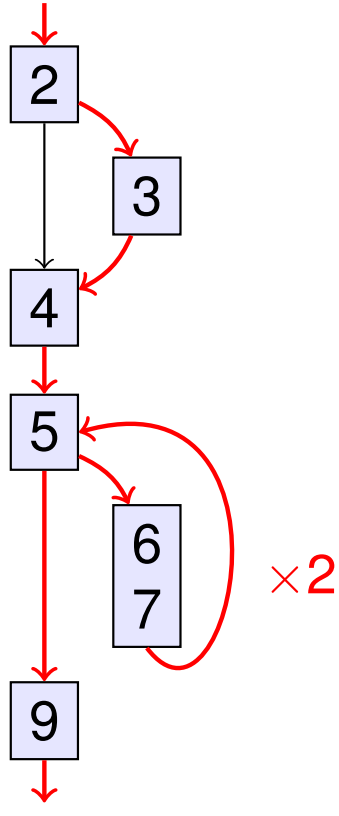
# Path Coverage

- Every execution path of the program has been explored



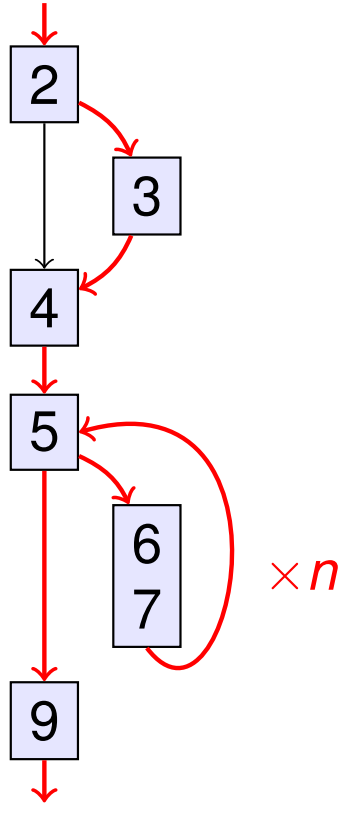
# Path Coverage

- Every execution path of the program has been explored



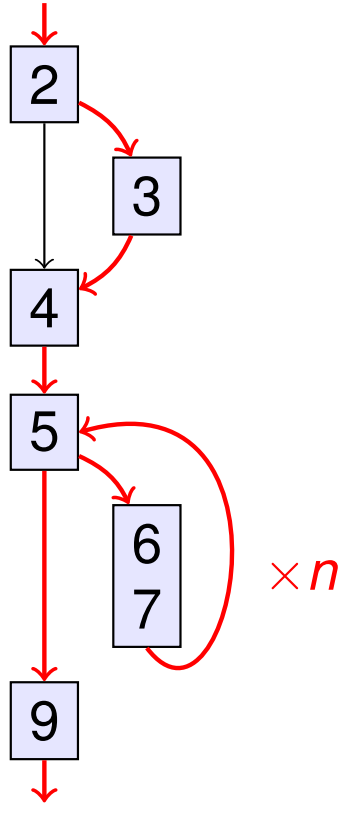
# Path Coverage

- Every execution path of the program has been explored
- Execute loops 0,1, more than 1 time (loop coverage)



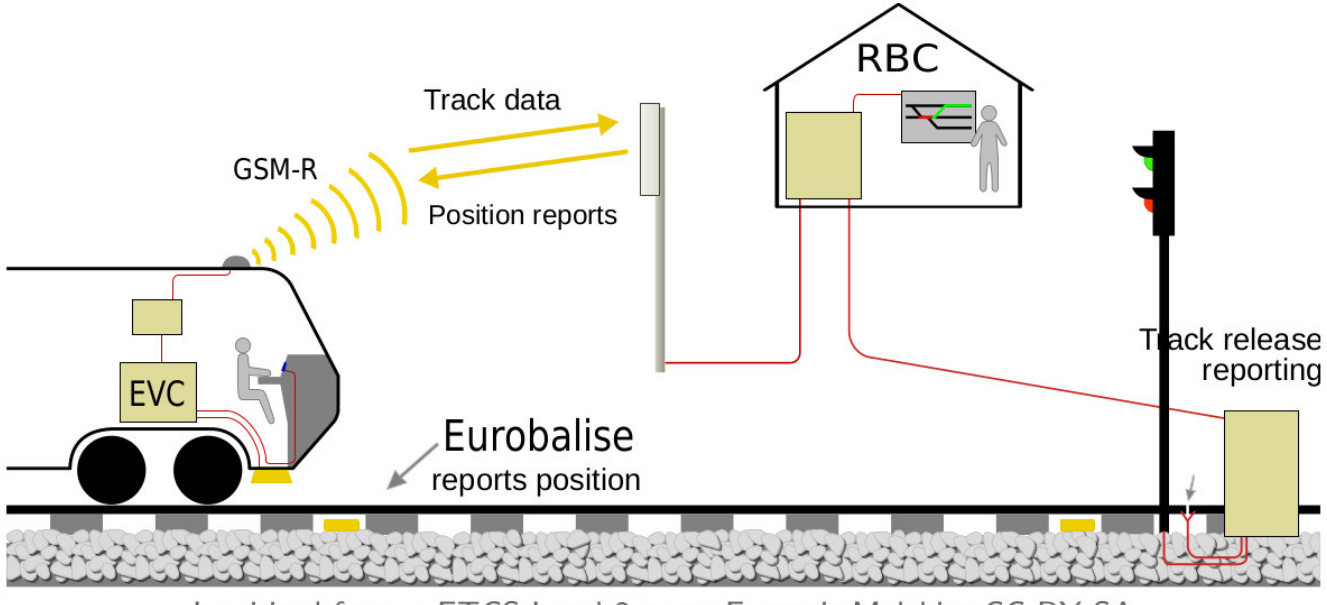
# Path Coverage

- Every execution path of the program has been explored
- Execute loops 0,1, more than 1 time (loop coverage)
- Exponential number of paths in general
- Infeasible paths



# Example: Railway Systems

- Safety-critical embedded systems
- Strict regulation by European and national agencies
- European Train Control System (ETCS)




Inspirited from « ETCS Level 2 » par François Melchior CC-BY-SA

# European Norm CENELEC EN 50128

- Development & validation process for railway systems
- Safety integrity levels SIL0 up to SIL4
- Organizational structure
- Development cycle (V model)
- Validation activities and reports for each project phase

DKE 351.3_2011-0207

EUROPEAN STANDARD
NORME EUROPÉENNE
EUROPÄISCHE NORM

**EN 50128**

June 2011

ICS 35.240.60; 45.020; 93.100

Supersedes EN 50128:2001

English version

**Railway applications -**
**Communication, signalling and processing systems -**
**Software for railway control and protection systems**

Applications ferroviaires -
Systèmes de signalisation, de télécommunication et de traitement -
Logiciels pour systèmes de commande et de protection ferroviaire

Bahnanwendungen -
Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme -
Software für Eisenbahnsteuerungs- und Überwachungssysteme

This European Standard was approved by CENELEC on 2011-04-25. CENELEC members are bound to comply with the CEN/CENELEC Internal Regulations which stipulate the conditions for giving this European Standard the status of a national standard without any alteration.

Up-to-date lists and bibliographical references concerning such national standards may be obtained on application to the Central Secretariat or to any CENELEC member.

This European Standard exists in three official versions (English, French, German). A version in any other language made by translation under the responsibility of a CENELEC member into its own language and notified to the Central Secretariat has the same status as the official versions.

CENELEC members are the national electrotechnical committees of Austria, Belgium, Bulgaria, Croatia, Cyprus, the Czech Republic, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, the Netherlands, Norway, Poland, Portugal, Romania, Slovakia, Slovenia, Spain, Sweden, Switzerland and the United Kingdom.

CENELEC
European Committee for Electrotechnical Standardization
Comité Européen de Normalisation Electrotechnique
Europäisches Komitee für Elektrotechnische Normung

**Management Centre: Avenue Marnix 17, B - 1000 Brussels**

© 2011 CENELEC - All rights of exploitation in any form and by any means reserved worldwide for CENELEC members.

Ref. No. EN 50128:2011 E

# Extracts from EN 50128

**Table A.18 – Performance Testing**

| TECHNIQUE/MEASURE | Ref | SIL 0 | SIL 1 | SIL 2 | SIL 3 | SIL 4 |
|---|---|---|---|---|---|---|
| 1. Avalanche/Stress Testing | D.3 | - | R | R | HR | HR |
| 2. Response Timing and Memory Constraints | D.45 | - | HR | HR | HR | HR |
| 3. Performance Requirements | D.40 | - | HR | HR | HR | HR |

**Table A.19 – Static Analysis**

| TECHNIQUE/MEASURE | Ref | SIL 0 | SIL 1 | SIL 2 | SIL 3 | SIL 4 |
|---|---|---|---|---|---|---|
| 1. Boundary Value Analysis | D.4 | - | R | R | HR | HR |
| 2. Checklists | D.7 | - | R | R | R | R |
| 3. Control Flow Analysis | D.8 | - | HR | HR | HR | HR |
| 4. Data Flow Analysis | D.10 | - | HR | HR | HR | HR |
| 5. Error Guessing | D.20 | - | R | R | R | R |
| 6. Walkthroughs/Design Reviews | D.56 | HR | HR | HR | HR | HR |

# Extracts from EN 50128

**Table A.21 – Test Coverage for Code**

| Test coverage criterion | Ref | SIL 0 | SIL 1 | SIL 2 | SIL 3 | SIL 4 |
|---|---|---|---|---|---|---|
| 1. Statement | D.50 | R | HR | HR | HR | HR |
| 2. Branch | D.50 | - | R | R | HR | HR |
| 3. Compound Condition | D.50 | - | R | R | HR | HR |
| 4. Data flow | D.50 | - | R | R | HR | HR |
| 5. Path | D.50 | - | R | R | HR | HR |

Requirements:

1) For every SIL, a quantified measure of coverage shall be developed for the test undertaken. This can support the judgment on the confidence gained in testing and the necessity for additional techniques.

2) For SIL 3 or 4 test coverage at component level should be measured according to the following:

   - 2 and 3; or
   - 2 and 4; or
   - 5

   or test coverage at integration level should be measured according to one or more of 2, 3, 4 or 5.

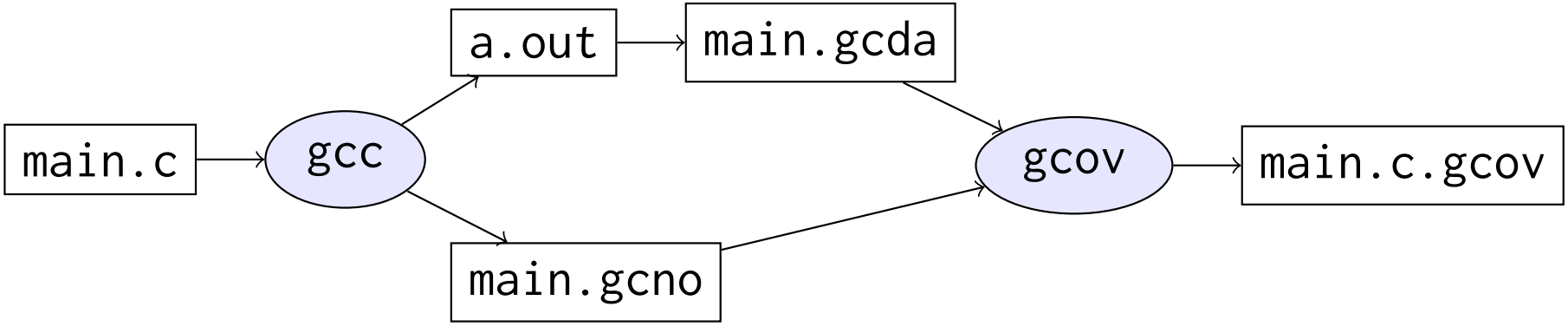
3) Other test coverage criteria can be used, given that this can be justified. These criteria depend on the software architecture (see Table A.3) and the programming language (see Table A.15 and Table A.16).

4) Any code which it is not practicable to test shall be demonstrated to be correct using a suitable technique, e.g. static analysis from Table A.19.

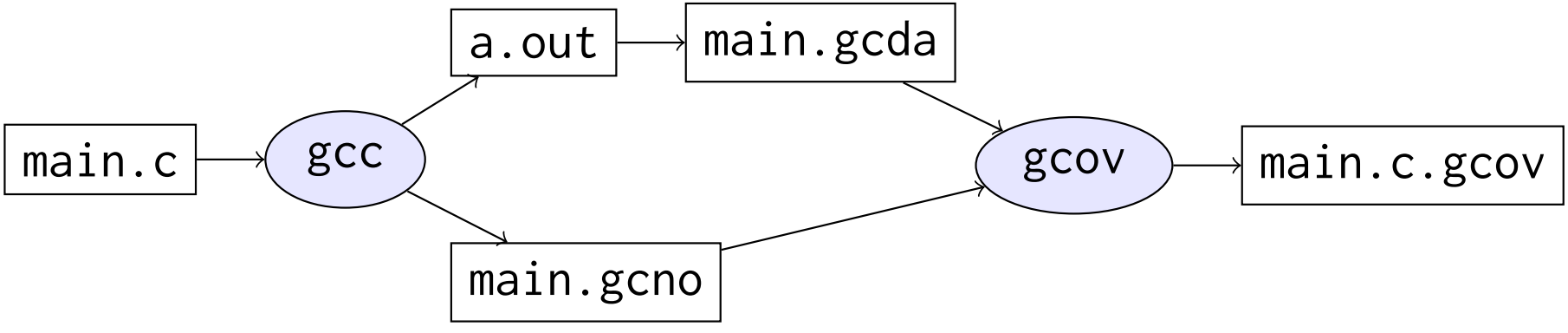
# Practical Code Coverage with `gcc` and `gcov`

- `gcc` has a magic option `--coverage`
- Instrumentation of binary code
- Count execution of each basic block
- Count branches taken/untaken
- Generate coverage report using `gcov` (or `lcov`)
- Integration into build system

# Coverage Tool Flow with gcov

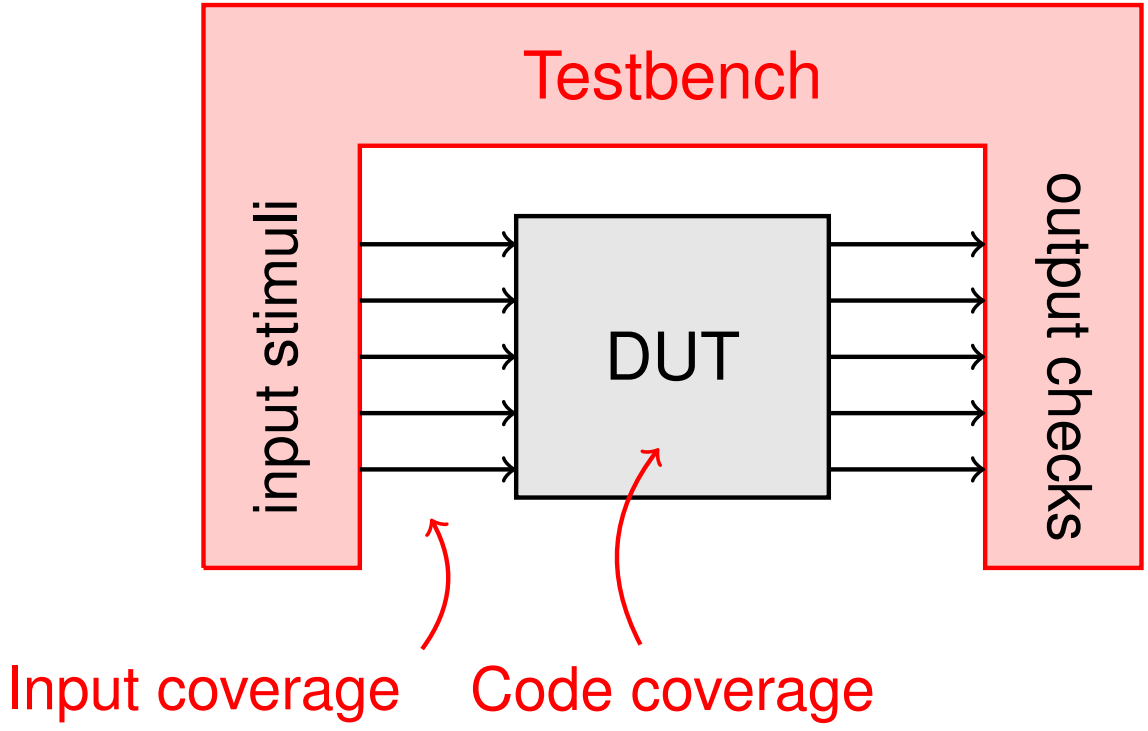


# Coverage Tool Flow with gcov

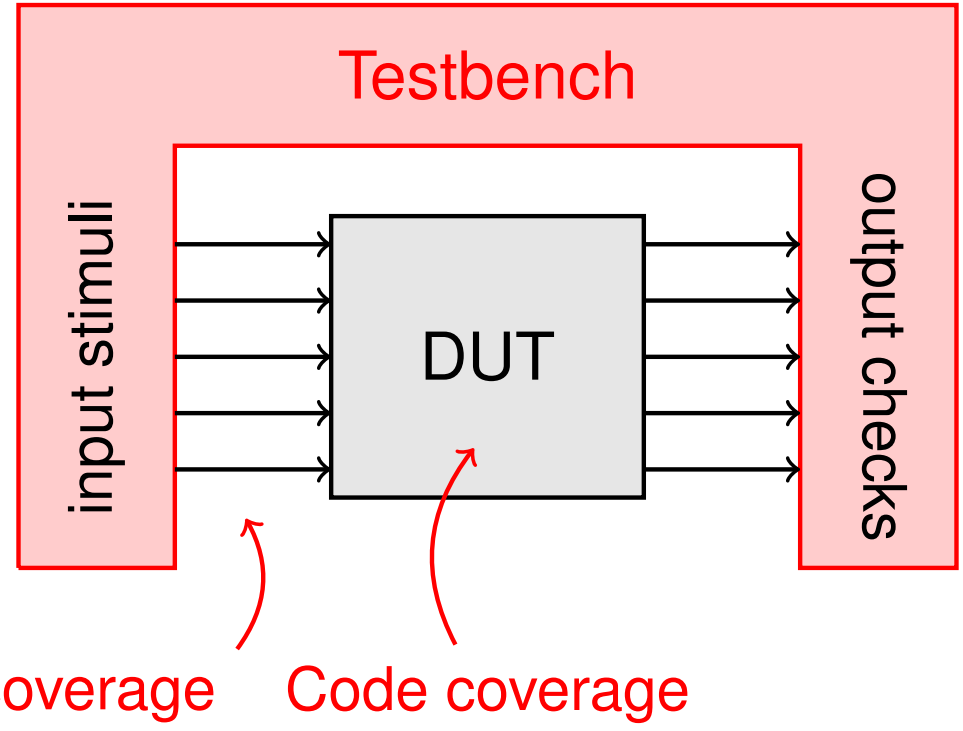



Let's do it...

# Test Coverage (Again)



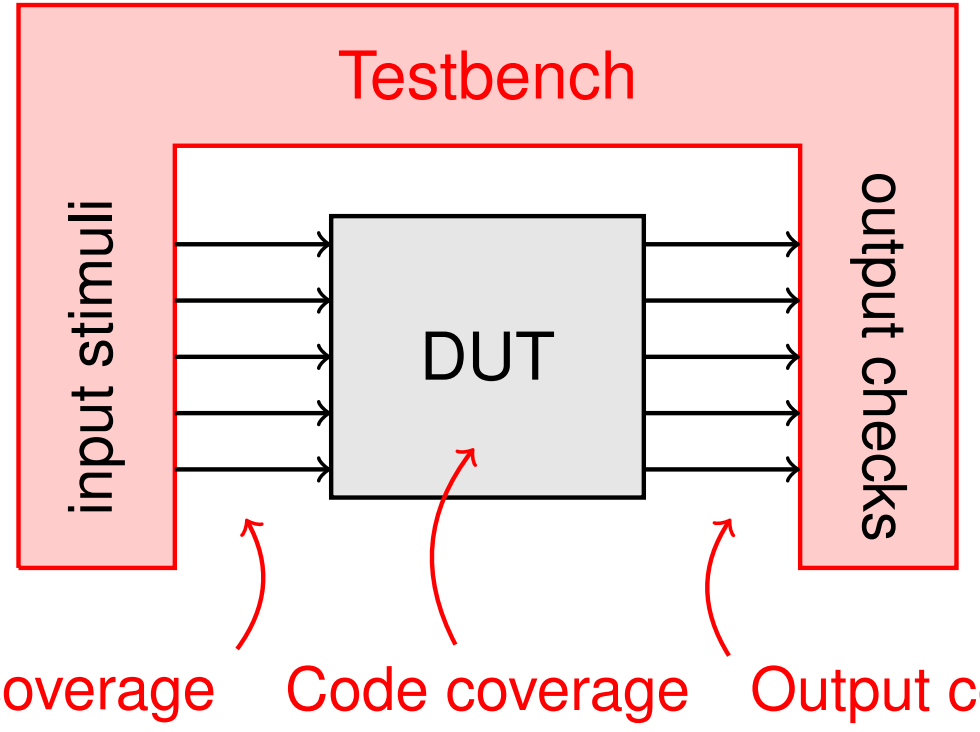
# Test Coverage (Again)




What am I missing here. . . ?

?

# Test Coverage (Again)




What am I missing here. . . ?

?

# Mutation Coverage

Did we check the right things at the output? ?

How to assess test bench quality? ?

# Mutation Coverage

Did we check the right things at the output? ?

How to assess test bench quality? ?

- Mutation coverage (aka error seeding)
- Randomly insert errors into the code (mutants)
- Check if the test bench captures (kills) them
- Compute ratio of killed mutants

# Mutation Coverage

Did we check the right things at the output? ?

How to assess test bench quality? ?

- Mutation coverage (aka error seeding)
- Randomly insert errors into the code (mutants)
- Check if the test bench captures (kills) them
- Compute ratio of killed mutants

# Mutation Coverage: Rationale

- Mutations should mimic typical mistakes
  - Loop condition off by one
  - Replace operators such as $<$ vs $\leq$
  - Modify constants
  - ...
- A test bench not detecting these mistakes should be improved
- Mutation coverage approximates ratio of real bugs found

# Mutation Coverage: Rationale

- Mutations should mimic typical mistakes
  - Loop condition off by one
  - Replace operators such as $<$ vs $\leq$
  - Modify constants
  - ...
- A test bench not detecting these mistakes should be improved
- Mutation coverage approximates ratio of real bugs found

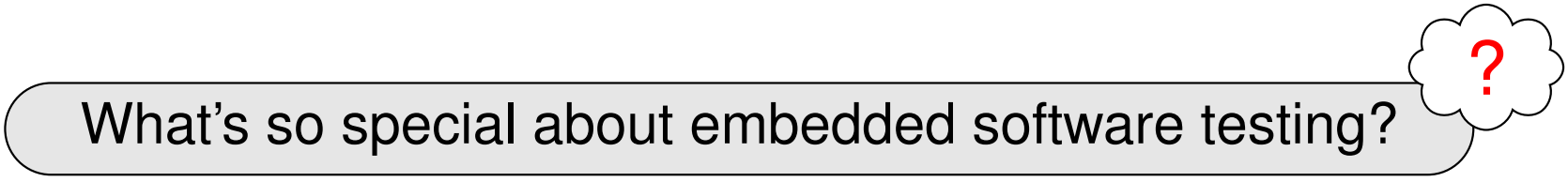
$$\frac{\text{Number of mutants killed}}{\text{Total number of mutants}} \approx \frac{\text{Number of real bugs found}}{\text{Total number of real bugs}}$$

# Summary Coverage

- Coverage metrics measure test quality
- Widely used in embedded industry
- Strong requirements for railway, aerospace, and automobile domains
- Simple coverage with `gcc` and `gcov`
- Comes at virtually no cost $\Rightarrow$ Use it!

# Plan

# Testing Embedded Software



# Testing Embedded Software

What's so special about embedded software testing?


- Runs on dedicated (expensive, scarce, buggy, . . . ) hardware
- Hardware not available (yet)
- Limited memory
- Limited debug capabilities
- Real-time
- Complex interactions with physical world
- Long build and upload times


# Embedded Testing Techniques

- Testing on target hardware
  -  High confidence in test results
  -  Long test cycles, hardware might not be available (yet)

# Embedded Testing Techniques

- Testing on target hardware
  -  High confidence in test results
  -  Long test cycles, hardware might not be available (yet)
- Emulating the target hardware (FPGA)
  -  Test results close to the real target
  -  Difficult to set up, HDL sources needed

# Embedded Testing Techniques

- Testing on target hardware
  -  High confidence in test results
  -  Long test cycles, hardware might not be available (yet)
- Emulating the target hardware (FPGA)
  -  Test results close to the real target
  -  Difficult to set up, HDL sources needed
- Testing on a virtual platform (SystemC, qemu, . . . )
  -  High performance, no dedicated hardware
  -  High development effort

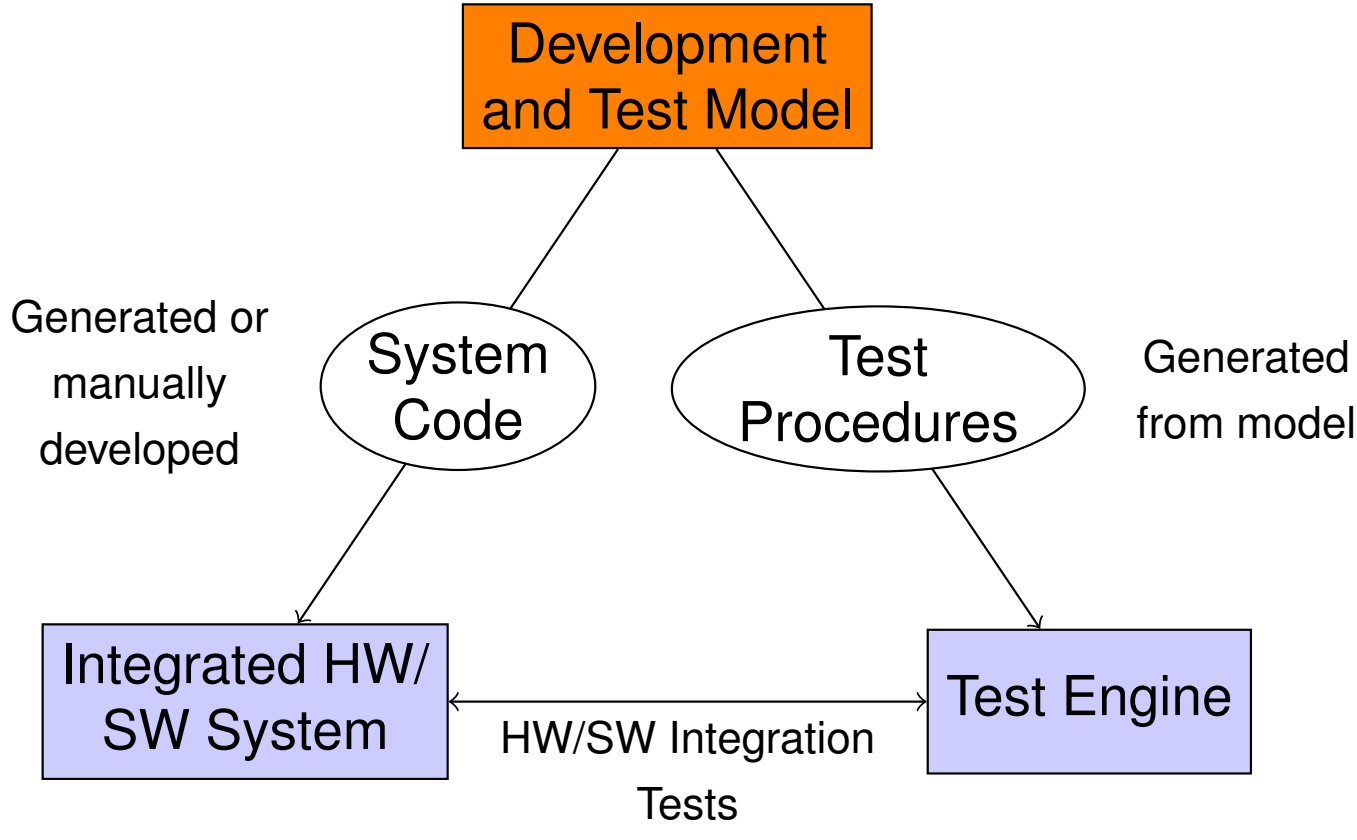
# Embedded Testing Techniques

- Testing on target hardware
  -  High confidence in test results
  -  Long test cycles, hardware might not be available (yet)
- Emulating the target hardware (FPGA)
  -  Test results close to the real target
  -  Difficult to set up, HDL sources needed
- Testing on a virtual platform (SystemC, qemu, . . . )
  -  High performance, no dedicated hardware
  -  High development effort
- Purely C-based shallow test harness
  -  Short test cycles, easy to set up
  -  Difficult for real-time systems

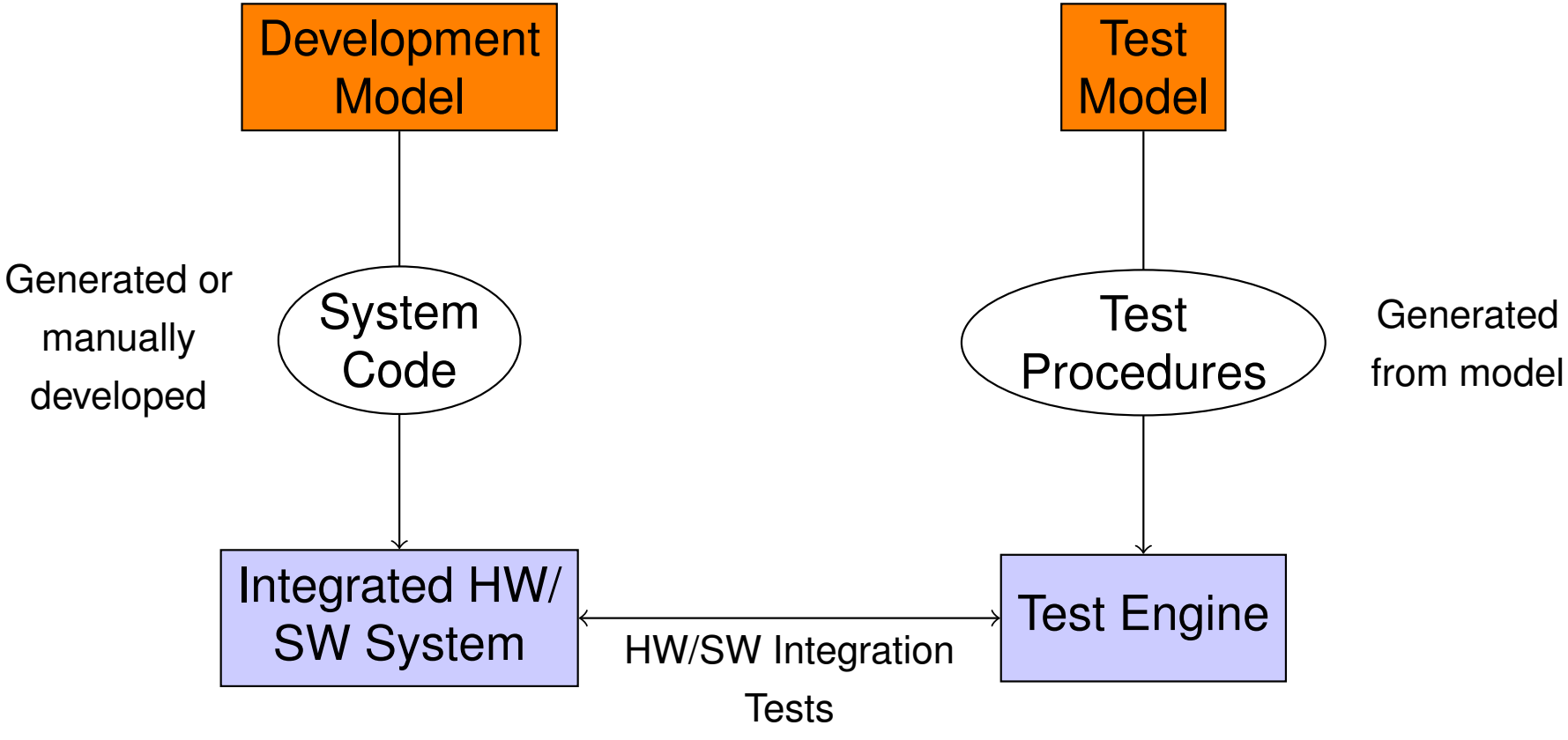
# Embedded Testing Techniques

- Testing on target hardware → SE743
  - ☺ High confidence in test results
  - ☹ Long test cycles, hardware might not be available (yet)
- Emulating the target hardware (FPGA) → SE744
  - ☺ Test results close to the real target
  - ☹ Difficult to set up, HDL sources needed
- Testing on a virtual platform (SystemC, qemu, . . . ) → SE747
  - ☺ High performance, no dedicated hardware
  - ☹ High development effort
- Purely C-based shallow test harness → Here!
  - ☺ Short test cycles, easy to set up
  - ☹ Difficult for real-time systems

# Alternative: Model-based Testing

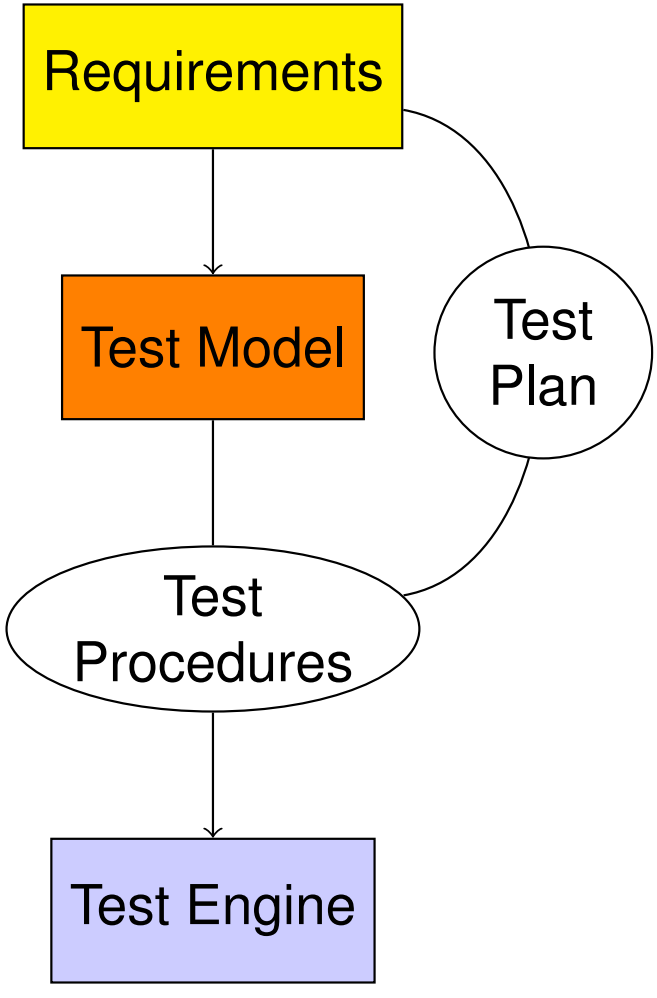


# Alternative: Model-based Testing

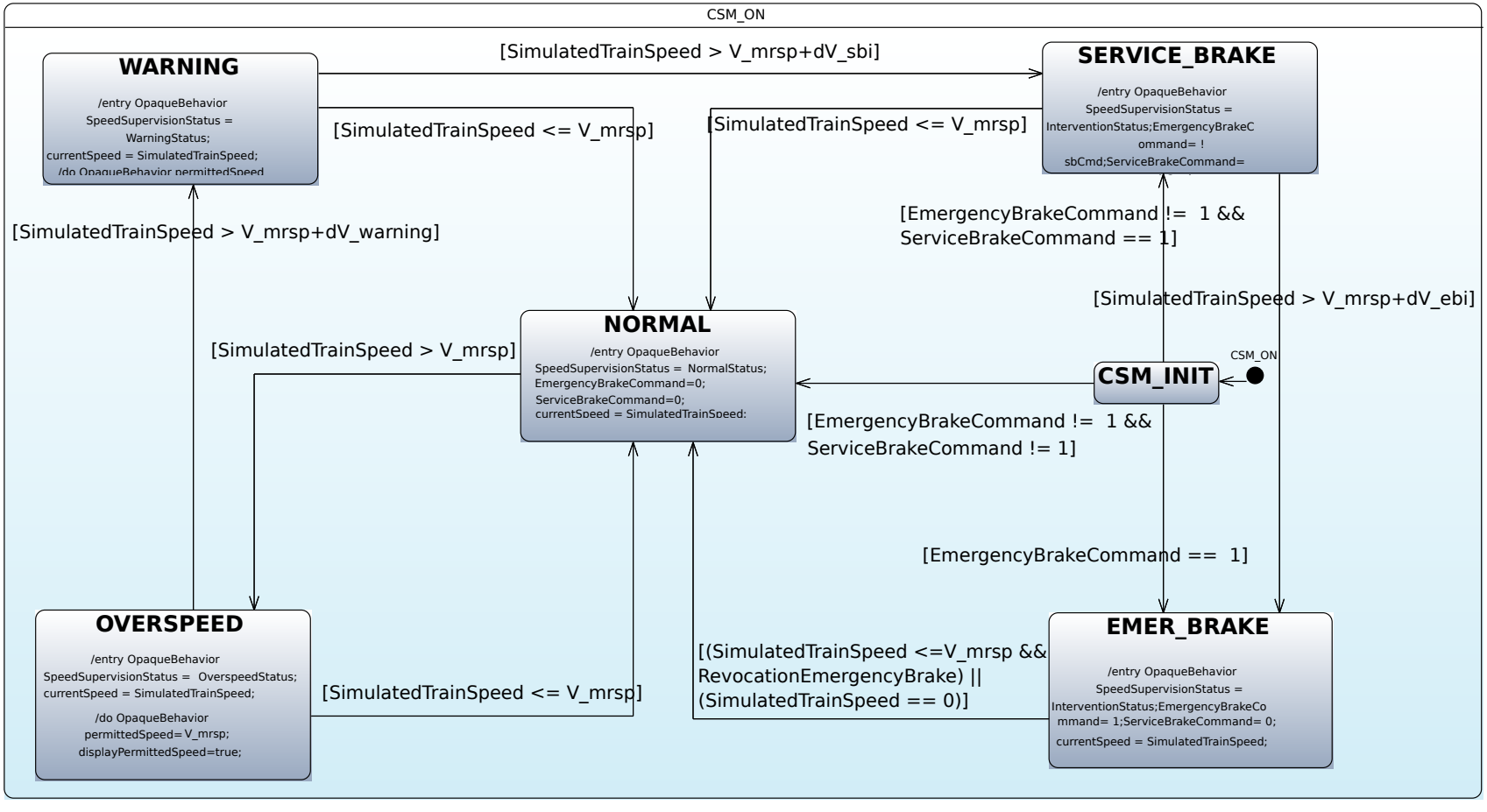


# Principles of Model-Based Testing

- Use of well-founded models (e.g. SysML state charts)
- Models serve for documentation and review
- Enables testing during development
- Automated generation of test cases
- Automated requirements traceability



# Example from Railway Domain




[Source: OpenETCS project, Cécile Braunstein]

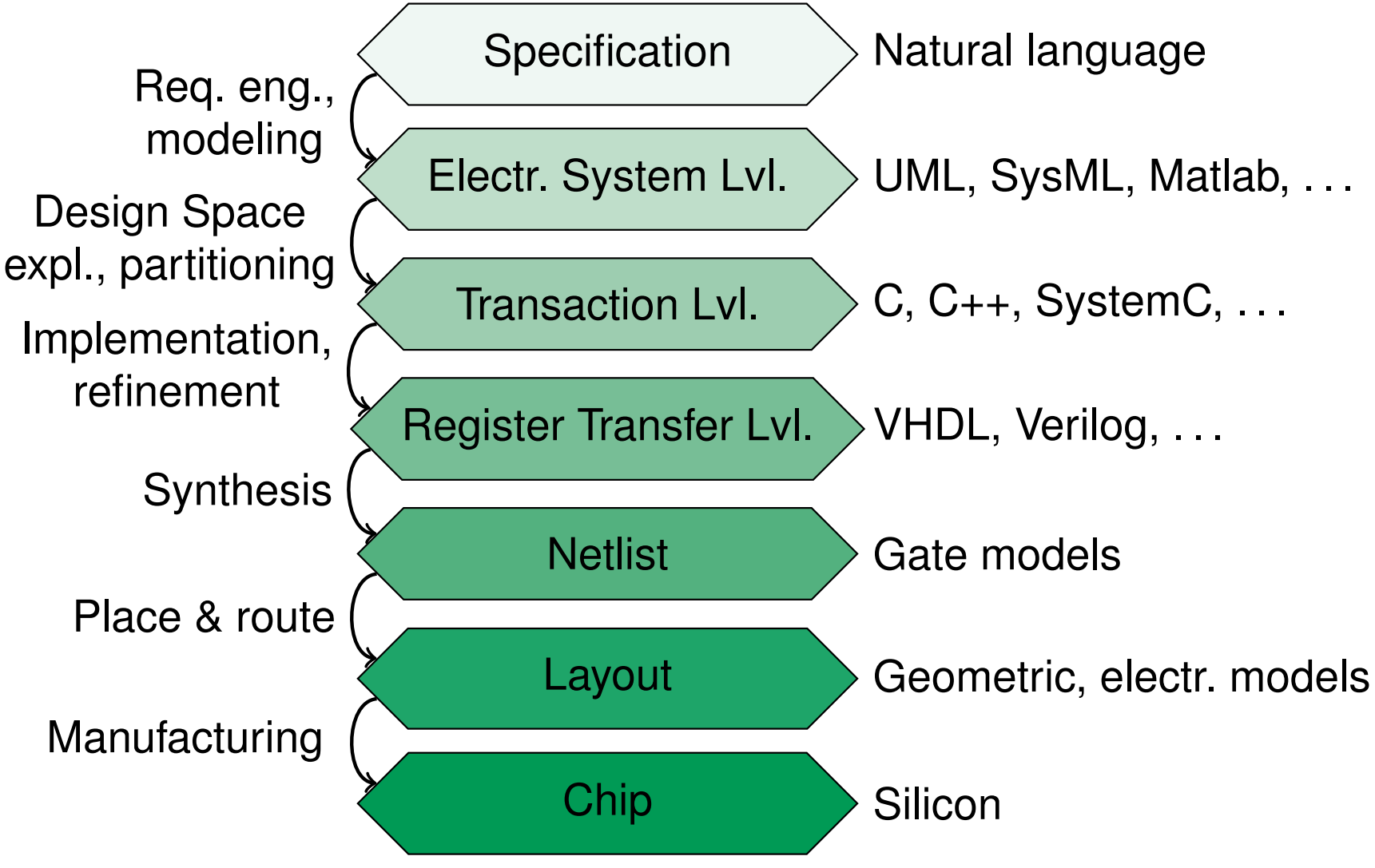
# Other Testing Aspects

- Mechanical testing
  - Vibrations
  - Shock
  - Standardized stress
- Environmental conditions
  - Temperature
  - Pressure
  - Humidity
  - Radiation
- Ageing

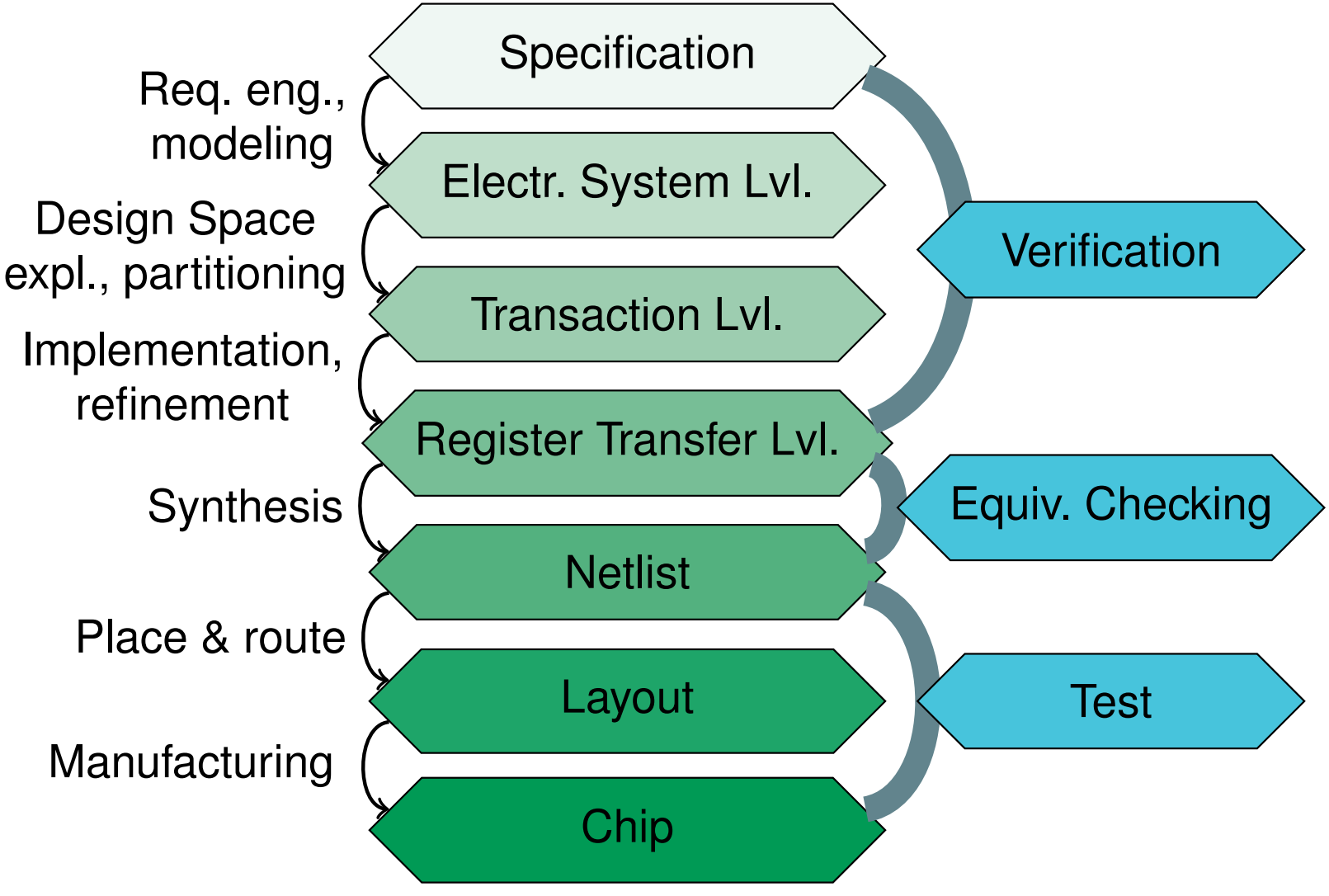

[Source: Institute of Space Systems, DLR]

# Plan

# Hardware Design Flow



# Hardware Design Flow



# Hardware Verification vs Test

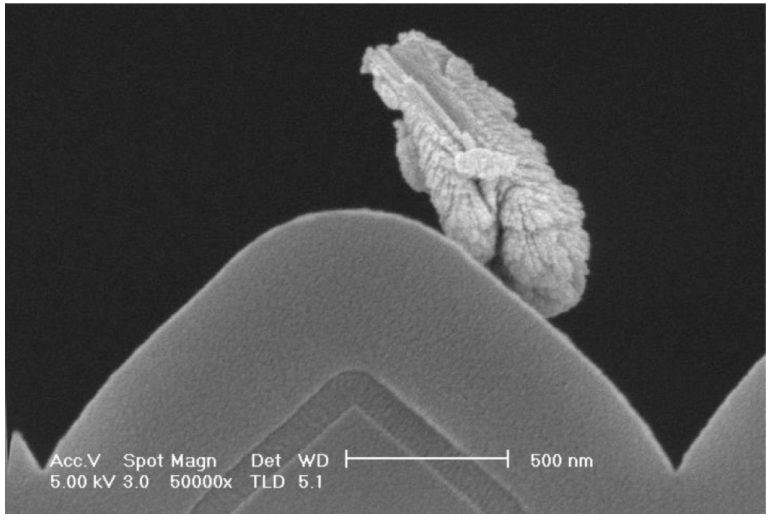
## Verification

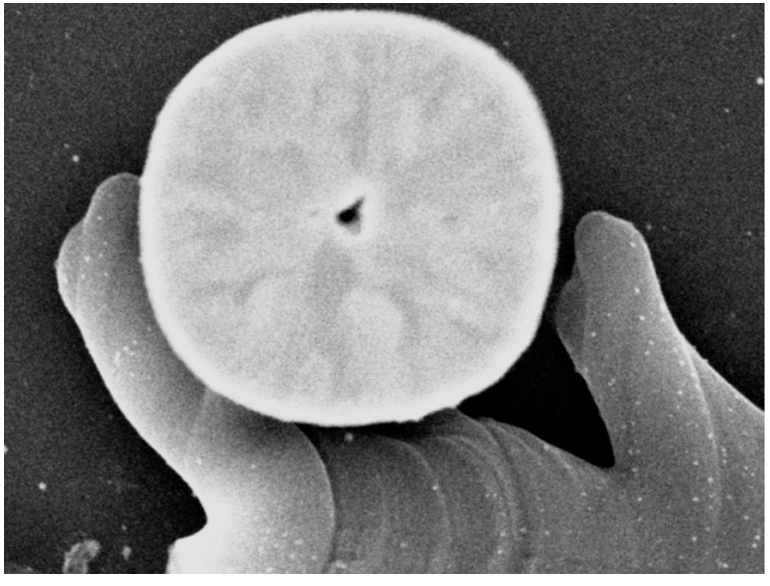
- Detect design bugs
- Extract properties from requirements
- Applied on RTL code
- High manual effort

## Test

- Detect physical defects
- Test generation from netlist according to fault model
- Applied on fabricated chips
- High automation

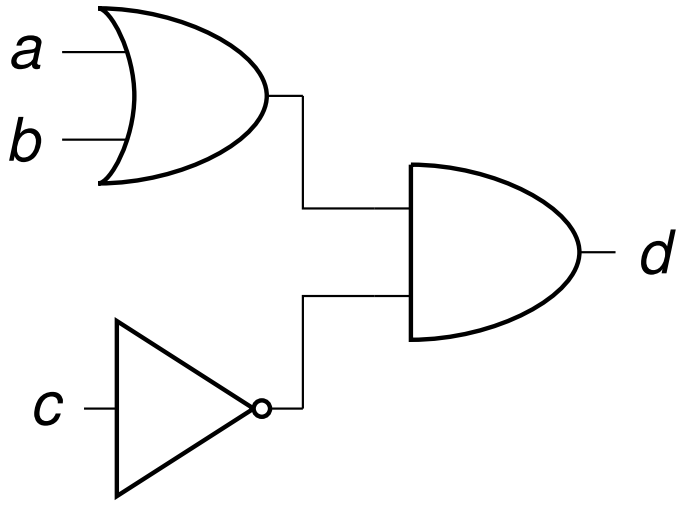
# Physical Defects



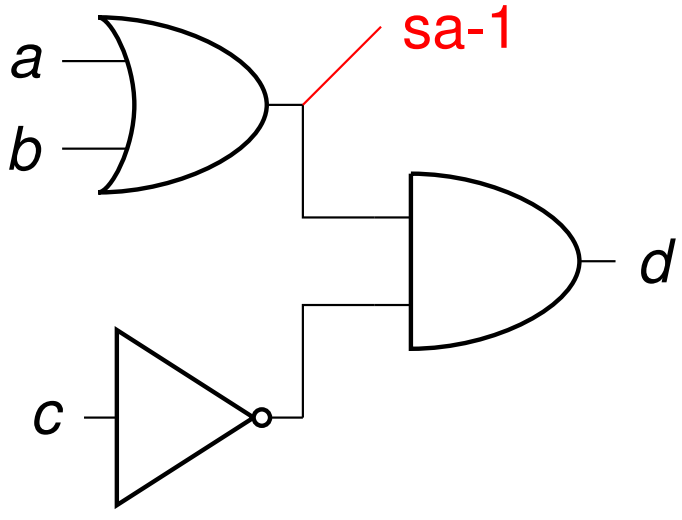
[Source: IEEE Spectrum “The Art of Failure”]

# Stuck-at Fault Model




| $a$ | $b$ | $c$ | $d$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

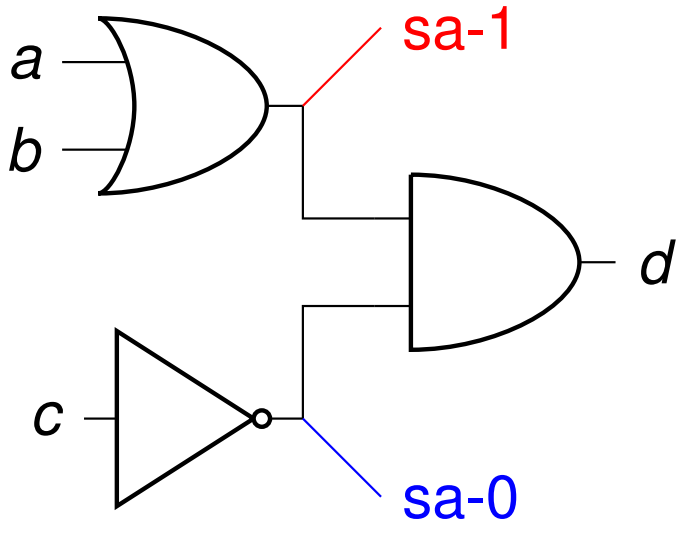
# Stuck-at Fault Model




| $a$ | $b$ | $c$ | $d$ |
|---|---|---|---|
| 0 | 0 | 0 | **0/1** |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

- $\langle 000 \rangle$ is a test vector for the shown stuck-at-1 fault

# Stuck-at Fault Model




| $a$ | $b$ | $c$ | $d$ |
|---|---|---|---|
| 0 | 0 | 0 | **0/1** |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | **1/0** |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | **1/0** |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | **1/0** |
| 1 | 1 | 1 | 0 |

- $\langle 000 \rangle$ is a test vector for the shown stuck-at-1 fault
- $\{\langle 010 \rangle, \langle 100 \rangle, \langle 110 \rangle\}$ are test vectors for the stuck-at-0 fault

# Automatic Test Pattern Generation

## ATPG

- Create a list of all possible (stuck-at) faults
- For each fault:
  - Find a test pattern
  - Drop all other faults detected by this pattern

- Untestable faults?
- Hard to test faults?
- Sequential tests?
- Test compression?

# Summary

- Validation on all levels of abstraction
- $>$ 50% of overall costs
- Crucial for project success and product quality
- Various techniques
  - Dynamic testing
  - Static verification
  - Model-based design
- Integration into development cycle

# Outlook

- Test-Driven Design of embedded software
- Introduction to formal methods
- Formal specification and verification of embedded hardware

# Preparation for Exercises

- Log in to GitLab:
  https://gitlab.telecom-paristech.fr
- Go to the GitLab group:
  https://gitlab.telecom-paristech.fr/MSSE/TestVerif/2018
- Request access to the group

# References I



DO-178B: Software Considerations in Airborne Systems and Equipment Certification, 1982.


EN 50128 - Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems.
Technical report, European Commitee for Electrotechnical Standardization, 2001.


James W. Grenning.
*Test Driven Development for Embedded C*.
Pragmatic Bookshelf, Raleigh, N.C, 1$^{st}$ edition, May 2011.


Kelly Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson.
A practical tutorial on modified condition/decision coverage, 2001.