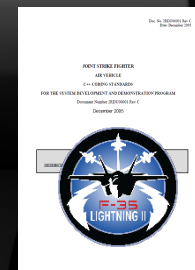


Funktionale Sicherheit mit automatisierten Softwaretests



CANTATA

Dynamisch & Statisch



Übersicht über Sicherheitsstandards und deren Inhalte

- ✓ Referenz zum V-Modell

ISO 26262 – Dynamische Tests

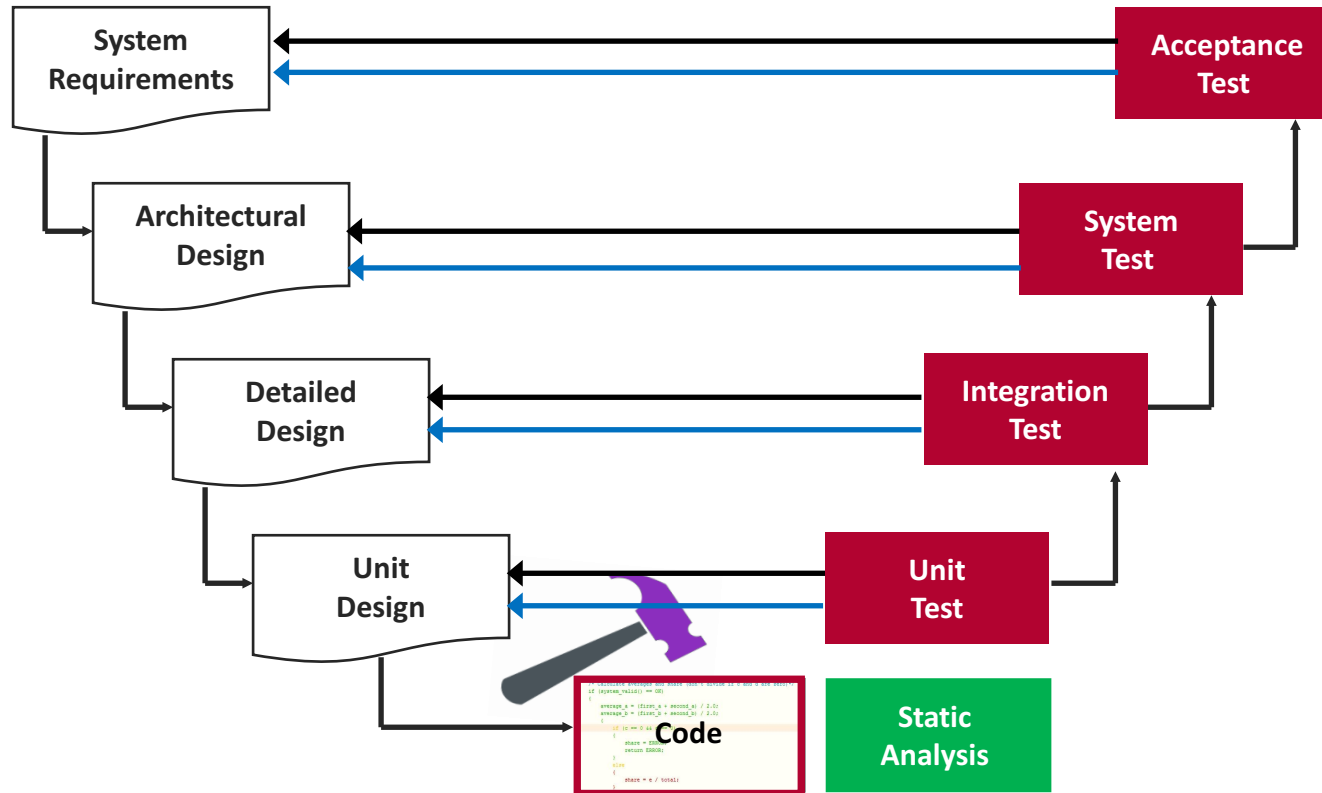
- ✓ Methoden
- ✓ Dynamische Analyse
- ✓ Testfallerstellung

ISO 26262 – Statische Analyse

- ✓ Implementierungsvorgaben
- ✓ Verifizierungsmethoden
- ✓ MISRA

Bosch Quality Improvement

Software Development Lifecycle



Verification Order

1 Does code meet the quality standard?

✓ Static Analysis

2 Does code do what it should?

✓ Functional Requirements Testing

✓ Non-functional Requirements Testing







3 Does code not do what it should not?

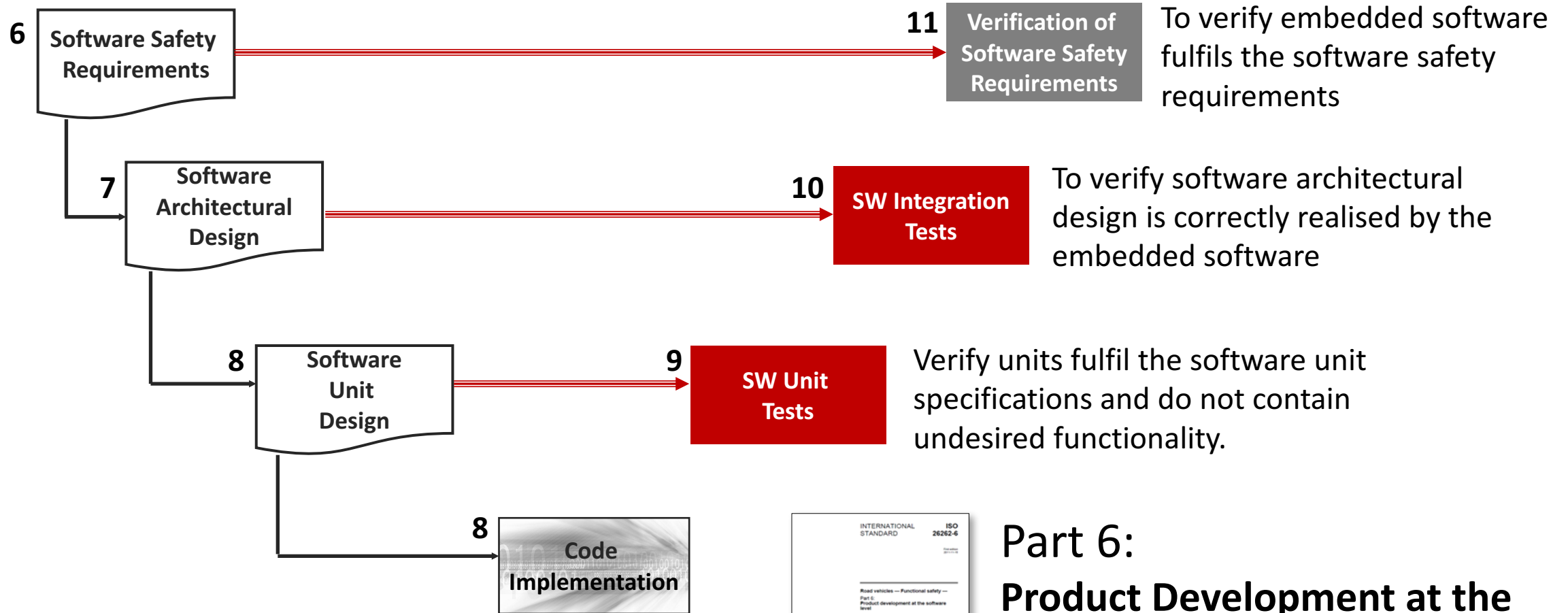
✓ Robustness Testing

4 Is code tested enough?

✓ Structural Coverage Testing

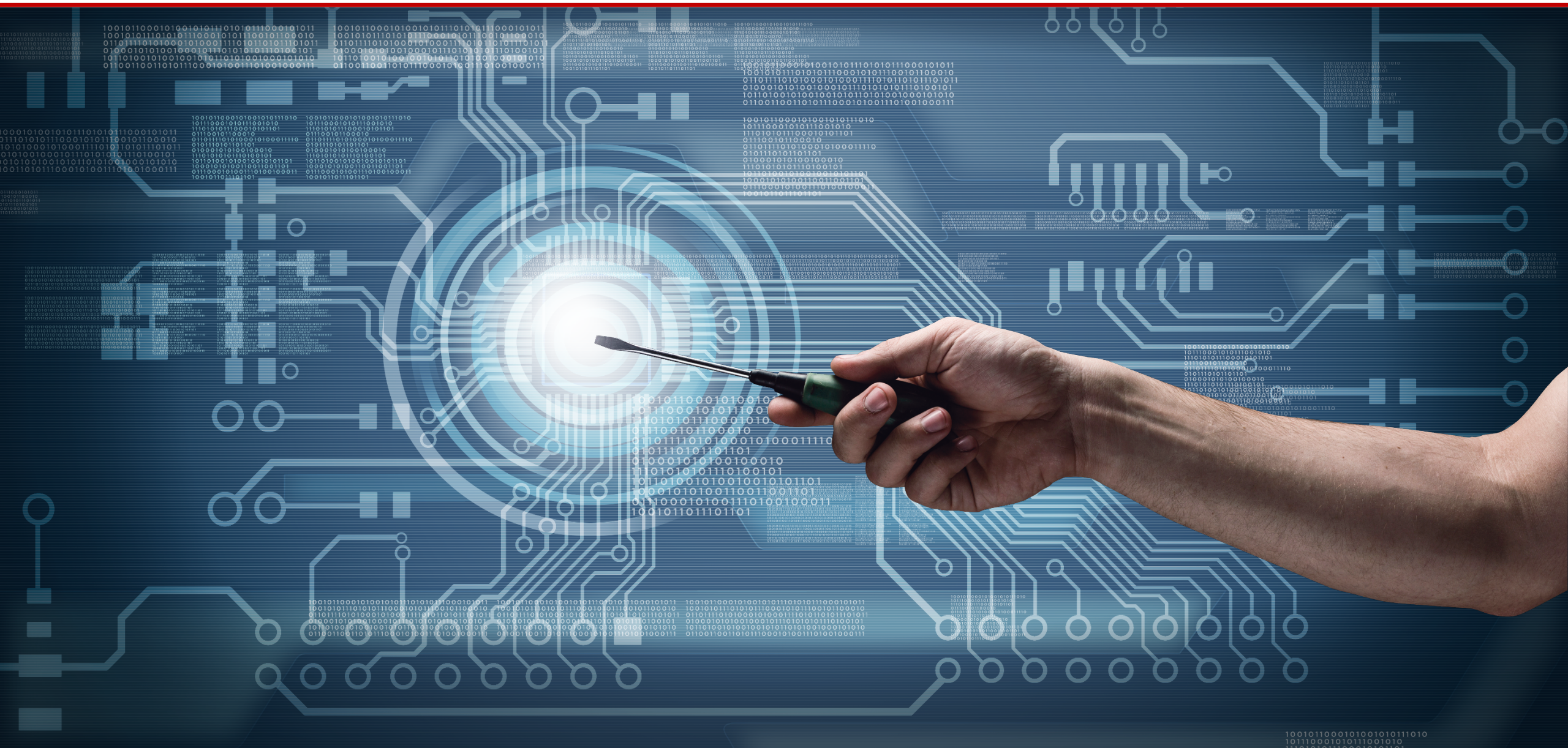
Testing Methods by Safety Standard

Sector						
Testing Methods	ISO 26262	DO-178B/C	IEC 62304	IEC 61508	EN 50128	IEC 60880
Static Analysis	✓	✓	✓	✓	✓	✓



Part 6:
Product Development at the software level
☐ Tables 10 - 15

Dynamic Testing



ISO 26262 Tables 10 & 13 – Methods for software unit & integration testing

Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a. Requirement-based test	++	++	++	++
1b. Interface test	++	++	++	++
1c. Fault injection test*	+	+	+	++
1d. Resource usage test	+	+	+	++
1e. Back-to-back comparison test between model and code (if applicable)	+	+	++	++

* This includes injection of arbitrary faults

(e.g. by corrupting values of variables, by introducing code mutations, or by corrupting values of CPU registers).

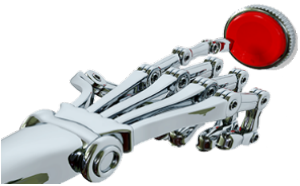
9.4.3 / 10.4.3

“The software unit / integration test methods listed in Table 10/13 shall be applied to demonstrate that both the software components and the embedded software achieve:

...

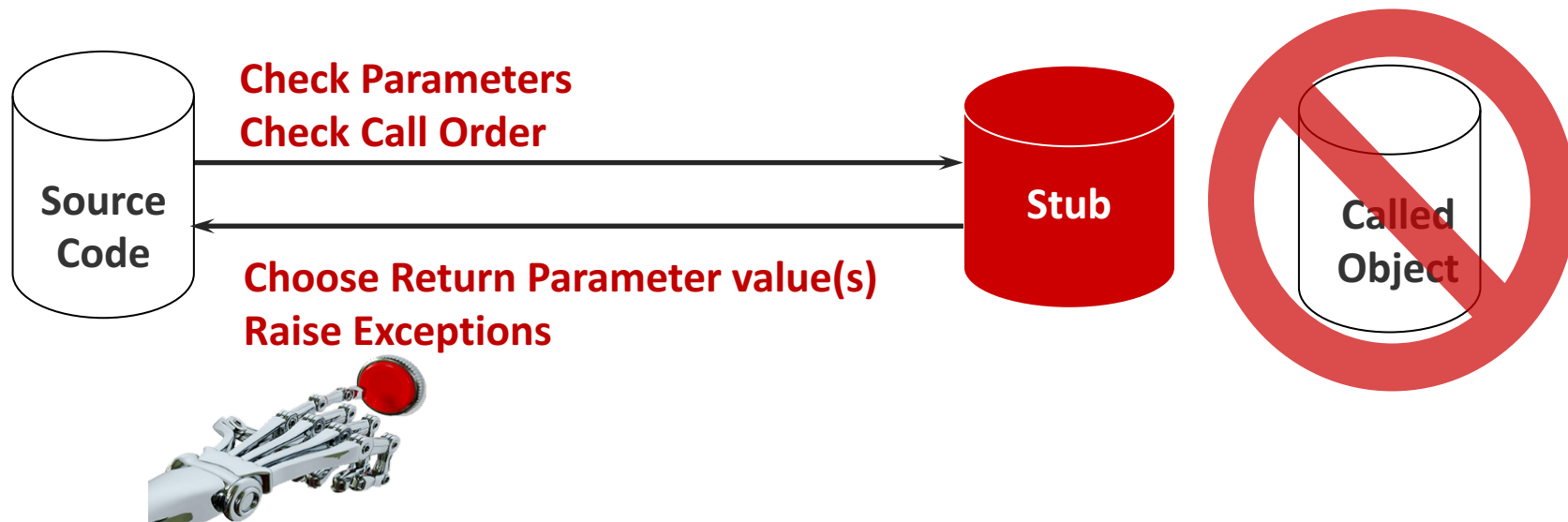
d) robustness;

EXAMPLE Absence of inaccessible software; effective error detection and handling.”



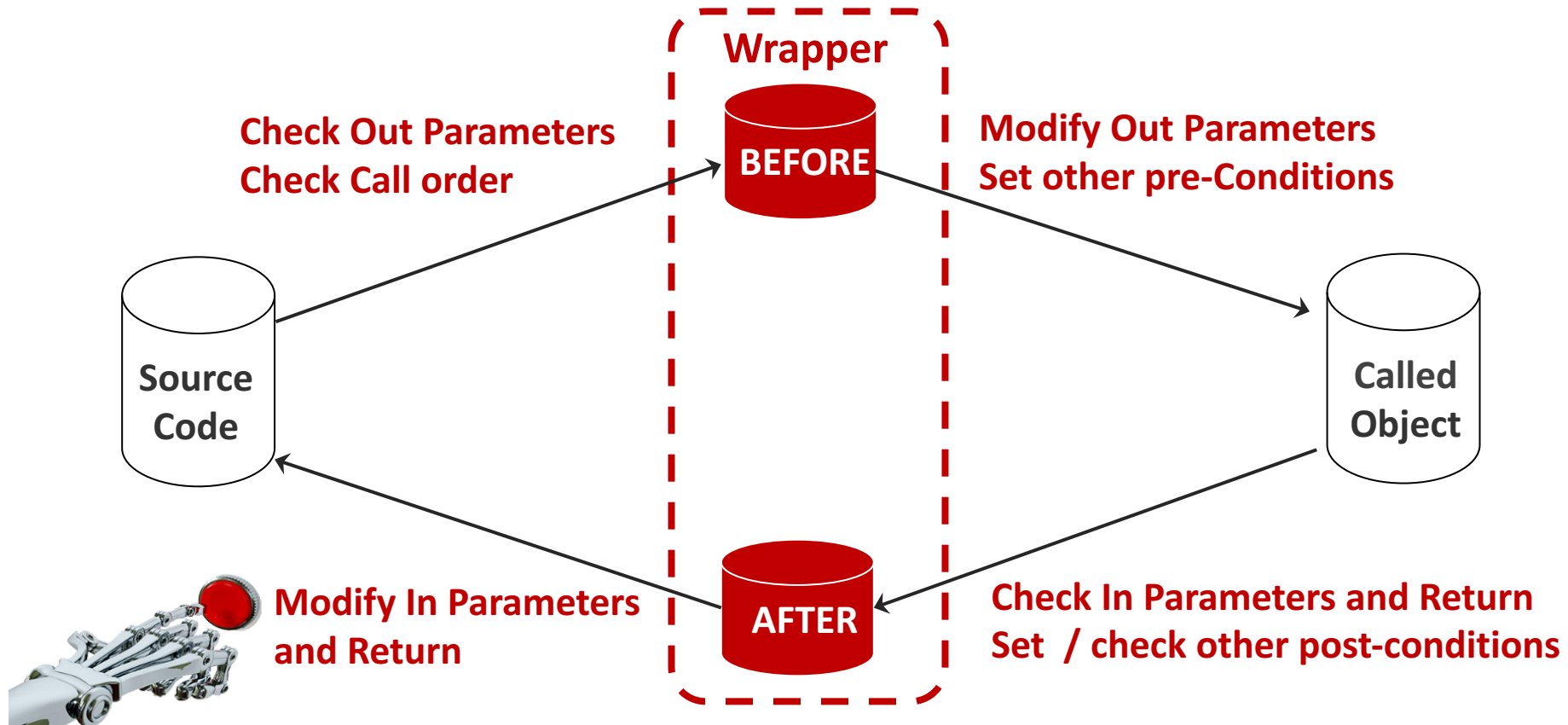
Interface Simulation

- ✓ A function/method inside test script with programmable instances
- ✓ Stub, (mock, fake, etc) replaces called object interface at link time with dummy code



Interface Interception

- ✓ A function/method inside test script with programmable instances
- ✓ Wrapper intercepts specific calls to and uses real called objects at run time



Where to control

Integration Test Entry-point

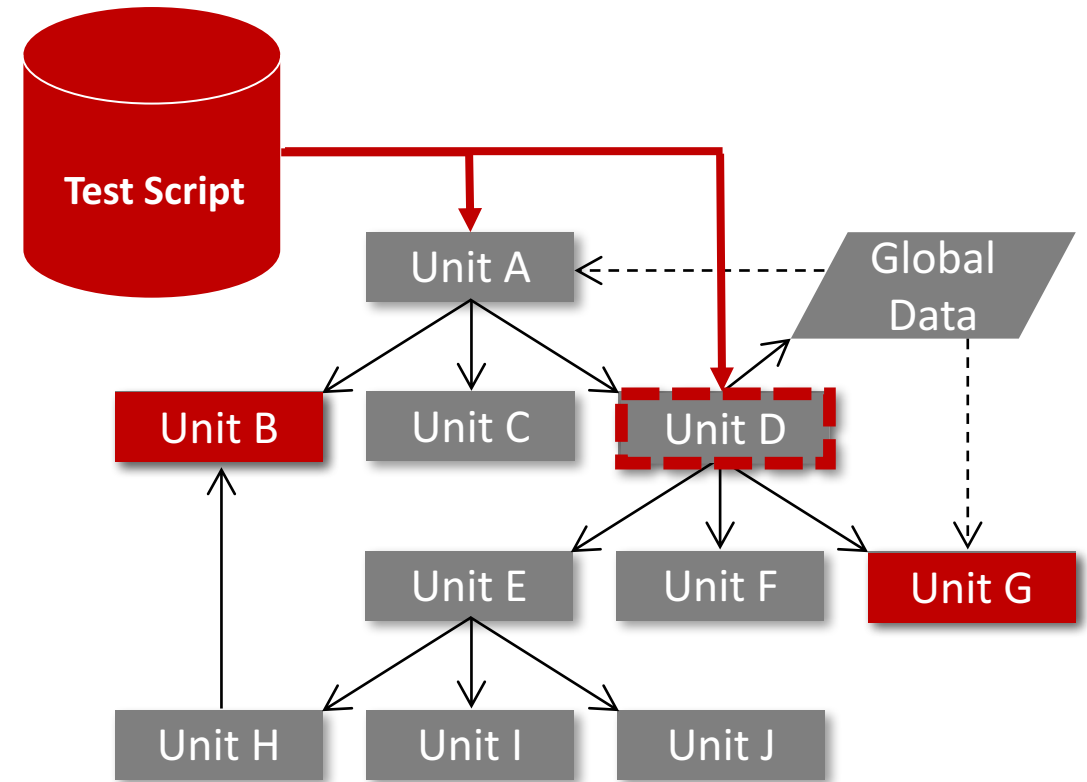
- ✓ Where to start driving from the test?

Simulation

- ✓ Stubs for calls to items not integrated

Interception

- ✓ Wrappers for specific calls to integrated items



Top-Down / Bottom-Up / As-One

What order can they be tested in?

What dependencies does that test bring?

Top-Down

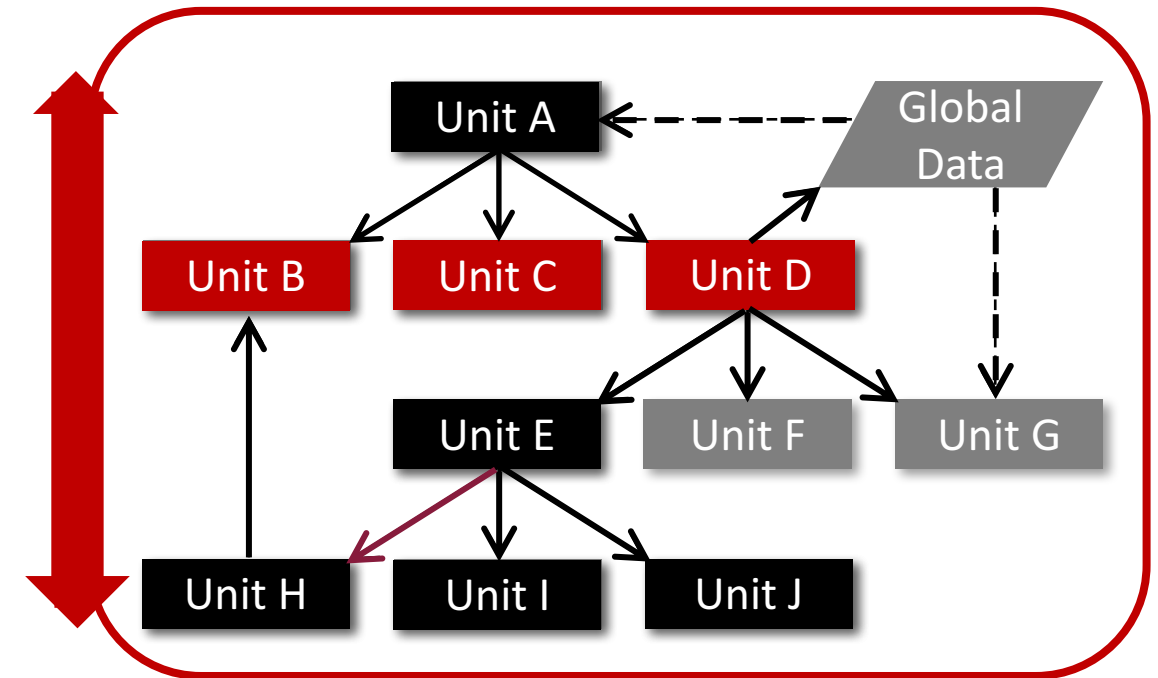
- ✓ Parents tested first & sub-units simulated then used to call sub-units

Bottom-Up

- ✓ Test script drives child-units then tested sub-units used in higher tests

As-One

- ✓ Wrapping Interceptions can check all interactions in any order



ISO 26262 Dynamic Analysis

ISO 26262 Tables 12 & 15 – Structural Coverage Metrics at the software unit & architecture levels

Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a. Statement coverage	++	++	+	+
1b. Branch coverage	+	++	++	++
1c. MC/DC Modified Condition/Decision Coverage)	+	+	+	++
1a. Function coverage	+	+	++	++
1b. Call Coverage	+	+	++	++

9.4.5

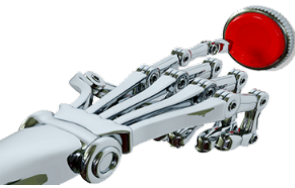
“To evaluate the completeness of test cases and to demonstrate that there is no unintended functionality, the coverage of requirements at the software unit level shall be determined and the structural coverage shall be measured in accordance with the metrics listed in Table 12. If the achieved structural coverage is considered insufficient, either additional test cases shall be specified or a rationale shall be provided.”

10.4.5

“To evaluate the completeness of tests and to obtain confidence that there is no unintended functionality, the coverage of requirements at the software architectural level by test cases shall be determined. If necessary, additional test cases shall be specified or a rationale shall be provided.”

10.4.6

“To evaluate the completeness of test cases and to obtain confidence that there is no unintended functionality, the structural coverage shall be measured in accordance with the metrics listed in Table 15. If the achieved structural coverage is considered insufficient, either additional test cases shall be specified or a rationale shall be provided.”



Pin-point Analysis

Simple Standards Compliance

- ✓ Rule-Sets for safety standards and SILs
- ✓ Metrics:
 - Entry- Points
 - Decisions
 - Relational Operators
 - Conditions (inc. masking + unique cause MC/DC)
 - Statements
 - Call Returns
 - Loops

Standalone or integrated with test suite

Powerful drill-down views, filters and reports

Automatic test case coverage optimization

The screenshot displays the QA Systems Coverage Results Explorer interface. The top pane shows a project tree with the following structure:

- C:\
 - workspace\
 - my_source_code\
 - roll_check\
 - roll_check.c
 - values\
 - values_check.c

The bottom pane shows the detailed coverage for the file `values_check.c`. It includes a code editor with the following code snippet:

```

if (high_value_check(first, second))
{
    strcpy(warning_message, "TOO HIGH");
}
else
{
    if (low_value_check(first, second))
    {
        strcpy(warning_message, "TOO LOW");
    }
}

```

Below the code editor, there are two summary tables:

Routine: char *values_check(int ,int)
Statement: if (high value check(first, second)) { strcpy(warni ...

	func	stmt
low	1	1
OK	1	1
Total	2	2

Decision: if (high_value_check(first, second))
Outcomes: 1/2 outcomes executed

	True	False
low	0	1
OK	0	1
Total	0	2

At the bottom, there is a 'Test Case Filter (2/2)' section showing '44M of 79M'.

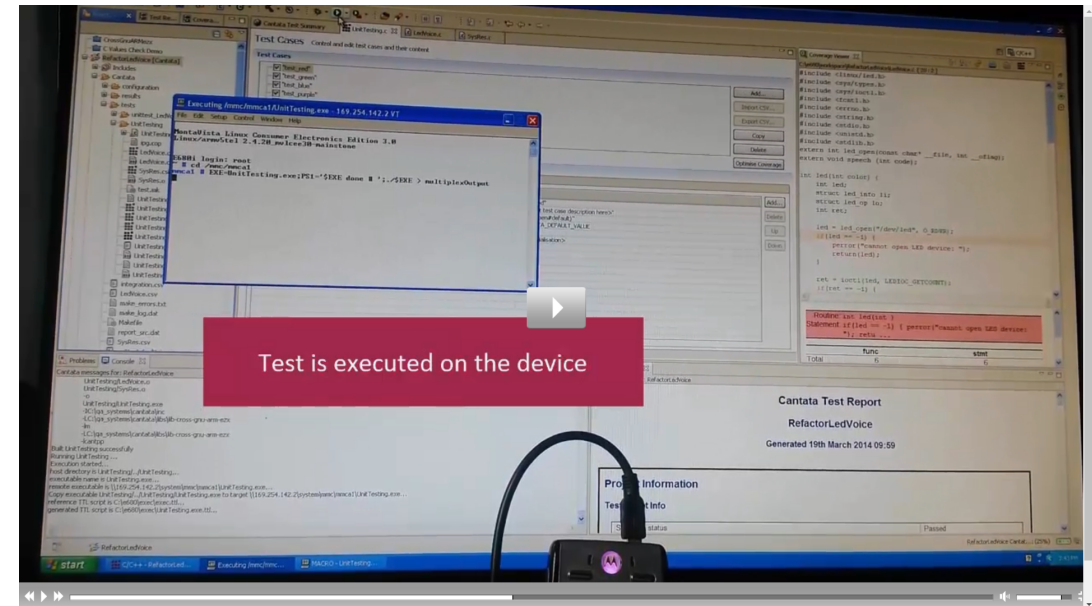
Unit Test HiL

Example controlling colours of an LED

Low-level calls to read / write operations on the LED port

Automatically intercepting return from LED to modify the call behavior at run time (HiL)

Injects faulty 'error conditions' back to controlling function to achieve desire code coverage



Use of Fault Injection to achieve Code Coverage

- ✓ Wrapping interception to inject fault
- ✓ Code coverage of HW errors becomes achievable with HiL

CANTATA

ISO 26262 Tables 11 & 14 – Methods for deriving test cases for software unit & integration testing

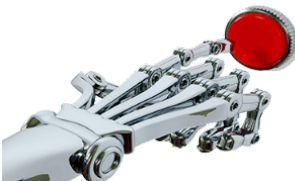


Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a. Analysis of requirements	++	++	++	++
1b. Generation and analysis of equivalence classes	+	++	++	++
1c. Analysis of boundary values	+	++	++	++
1d. Error guessing	+	+	+	+

9.4.3 / 10.4.3
 “The software unit / integration test methods listed in Table 10/13 shall be applied to demonstrate that both the software components and the embedded software achieve:

- a) compliance with the software unit / architectural design specification
- b) compliance with the specification of the hardware-software interface
- c) the specified functionality”

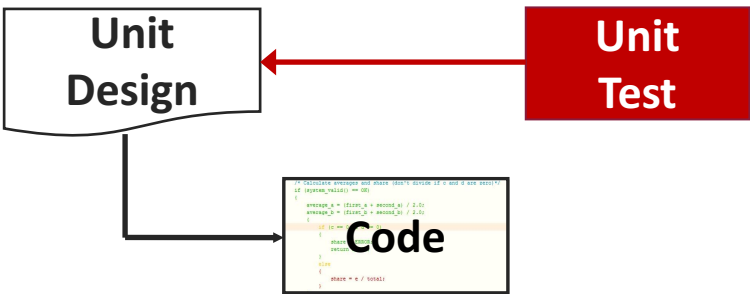
ISO 26262 Tables 12 & 15 – Structural Coverage Metrics at the software unit & architecture levels



Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a. Statement coverage	++	++	+	+
1b. Branch coverage	+	++	++	++
1c. MC/DC Modified Condition/Decision Coverage)	+	+	+	++

Example Requirements Specification

Req	Description
✓ LLRq_01	system_valid() shall be called to determine if input parameters are currently valid to use in calculations
✓ LLRq_02	when system_valid() returns OK [0] checked_status [GD int] shall be set to OK [0]
✓ LLRq_03	when system_valid() returns ERROR [3], checked_status [GD int] shall be set to ERROR [3]
✓ LLRq_04	when result of params e/(c+d) = zero, share [GD double] shall be set to ERROR [3]
✓ LLRq_05	when result of params e/(c+d) ≠ zero, share [GD double] shall be set to result value
✓ LLRq_06	warning level ERROR [3] shall be returned when check_status [GD int] is set to ERROR [3]
✓ LLRq_07	warning level ERROR [3] shall be returned when share [GD double] is set to ERROR [3]
✓ LLRq_08	warning level OK [0] shall be returned unless either check_status [GD int] or share [GD double] are set to ERROR [3]
✓ LLRq_09	warning level TOO_LOW [1] shall be returned when average of params first_a & second_a <11.25
✓ LLRq_10	warning level TOO_LOW [1] shall be returned when average of params first_b & second_b <13.25
✓ LLRq_11	warning level TOO_HIGH [2] shall be returned when average of params first_a & second_a >15.25
✓ LLRq_12	warning level TOO_HIGH [2] shall be returned when average of params first_b & second_b >18.25



Implementation

values_check.c

● int values_check()
 ● S low_value_check()
 ● S high_value_check()

● S average_a

● S average_b

values_check.h

● int checked_status

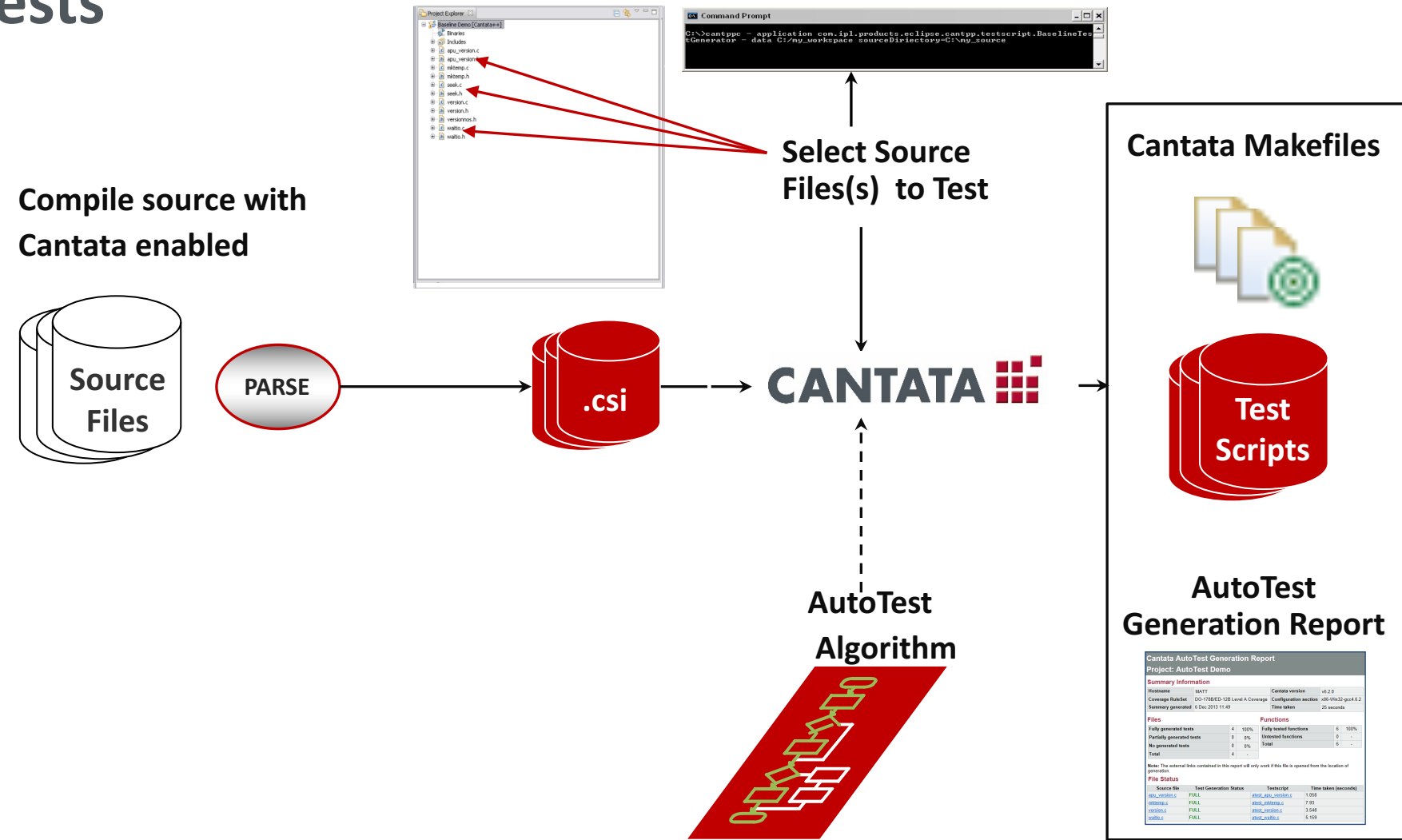
● double share

Testing Objectives

- ✓ Test cases which verify requirements
- ✓ Test cases which are traced to requirements (**requirements coverage**)
- ✓ Test cases which exercise all the software (**code coverage**)

Automatic Test Generation

Generate Tests from Code



Cantata AutoTest Generation Report
Project: AutoTest Demo

Summary Information			
Hostname:	WAT7	Cantata version:	v1.2.0
Coverage Rule Set:	COV-178865-128 Level A Coverage	Configuration section:	v10.0fx10.qcov.0.2
Statement generated:	6 Dec 2015 11:48	Time taken:	25 seconds
Files		Functions	
Fully generated tests	4 100%	Fully tested functions	6 100%
Partially generated tests	0 0%	Uncovered functions	0 0%
No generated tests	0 0%	Total	6 100%
Total	4 100%		

Note: The internal tests contained in this report will only work if this file is opened from the location of generation.

Source File	Test Generation Status	Testscript	Time taken (seconds)
src\api\api.c	FAIL	src\api\api001.c	1.003
src\api\api.h	FAIL	src\api\api002.c	7.193
src\api\api.c	FAIL	src\api\api003.c	3.648
src\api\api.h	FAIL	src\api\api004.c	5.199

Map Tests to Requirements

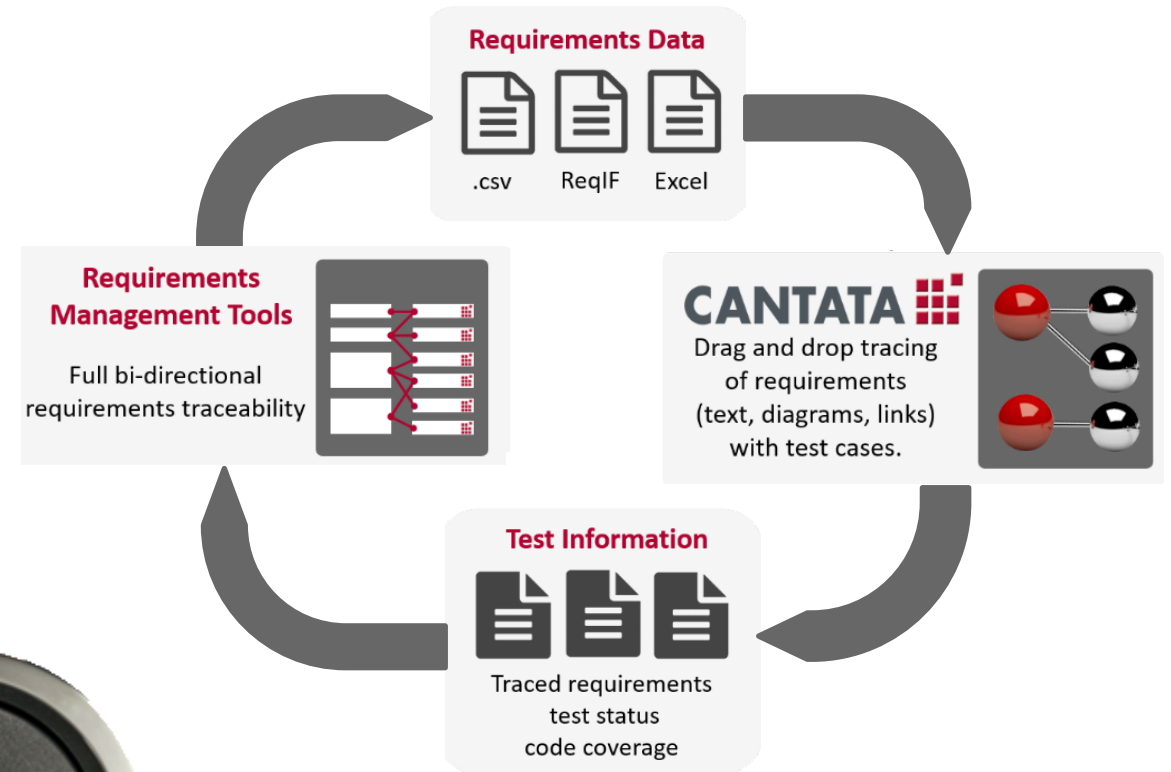
Bi-Directional Requirements Traceability

- ✓ Imports requirements data to Cantata server
Text, images & hyperlinks
- ✓ Drag & drop assignment
- ✓ Controlled export with results
status and coverage

Integrated with Requirements Tools



© All Copyright and Trademarks of their respective owners are acknowledged



Static Analysis



ISO 26262 – Implementation Principles

ISO 26262 Table 8 – Design principles for software unit design and implementation

Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a. One entry and one exit point in subprograms and functions	++	++	++	++
1b. No dynamic objects or variables, or else online test during their creation	+	++	++	++
1c. Initialization of variables	++	++	++	++
1d. No multiple use of variable names	+	++	++	++
1e. Avoid global variables or else justify their usage	+	+	++	++
1f. Limited use of pointers	0	+	+	++
1g. No implicit type conversions	+	++	++	++
1h. No hidden data flow or control flow	+	++	++	++
1i. No unconditional jumps	++	++	++	++
1j. No recursions	+	+	++	++

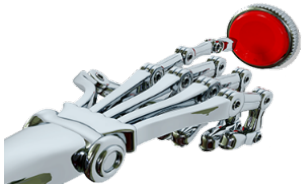
8.4.4

“The software unit design and implementation at the source code level as listed in Table 8 shall be applied to achieve the following properties:

...

c) correctness of data flow and control flow between and within the software units;”

NOTE For the C language, MISRA C covers many of the methods listed in Table 8.



MISRA – Motor Industry Software Reliability Association

MISRA C

- ✓ 1998 - Guidelines for the use of the C language in vehicle based software
 - ✓ MISRA C:1998 (MISRA C1)
- ✓ 2004 - MISRA C:2004 Guidelines for the use of the C language in critical systems
 - ✓ MISRA C:2004 (MISRA C2)
- ✓ 2013 - MISRA C:2012 Guidelines for the use of the C language in critical systems
 - ✓ MISRA C:2012 (MISRA C3)
 - ✓ 159 rules of which 138 are statically enforceable

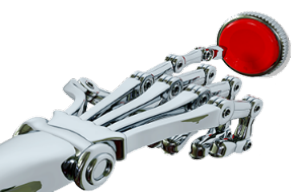
MISRA C++

- ✓ 2008 - Guidelines for the use of the C++ language in critical systems
 - ✓ 228 rules of which 219 are statically enforceable

ISO 26262 Static Analysis Verification Methods

ISO 26262 Table 9 – Methods for the verification of software unit design and implementation

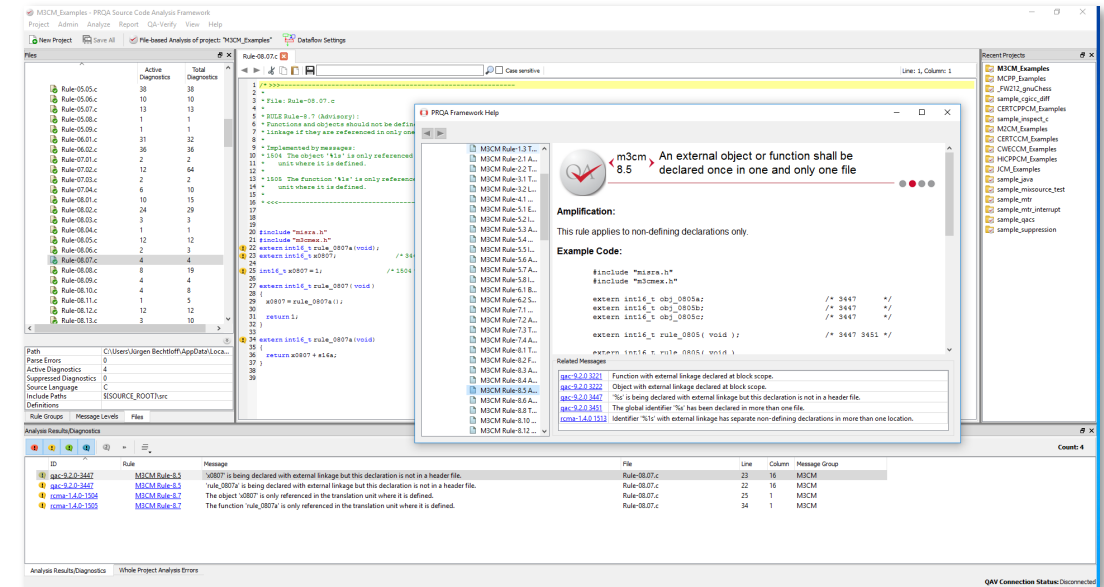
Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a. Walk-through	++	+	0	0
1b. Inspection	+	++	++	++
1c. Semi-formal verification	+	+	++	++
1d. Formal verification	0	0	+	+
1e. Control flow analysis	+	+	++	++
1f. Data flow analysis	+	+	++	++
1g. Static code analysis	+	++	++	++
1h. Semantic code analysis	+	+	+	+



8.4.5
 “The software unit design and implementation shall be verified in accordance with Clause 9, and by applying the verification methods listed in Table 9 to demonstrate:
 ...
 d) the compliance of the source code with the coding guidelines (see 5.5.3); and
 e) the compatibility of the software unit implementations with the target hardware

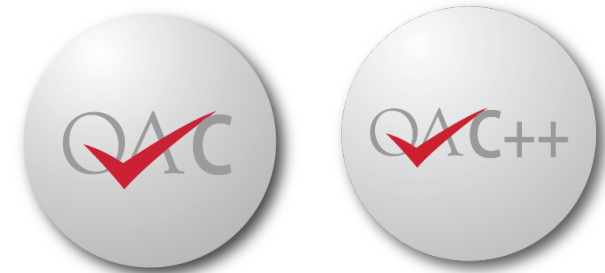
Static Analysis of C Code

- Select rule configuration
- Select level of data flow analysis
- Synchronize project to import source files
- Select and analyze source file(s)
- Review results
- Uninitialized variables and others



Use of Static Analysis to detect uninitialized variable

- ✓ Identify root cause of violation
- ✓ MISRA compliance rules and learning



Bosch Sustained Quality Improvement

Static Analysis

- ✓ Combination of
 - ✓ Code review
 - ✓ Compiler
 - ✓ QA-C
- ✓ Verification of
 - ✓ Functional requirements
 - ✓ **Coding guidelines**
 - ✓ Lexical correctness
 - ✓ Syntax
 - ✓ Semantics
 - ✓ **Complexity**

Dynamic Testing

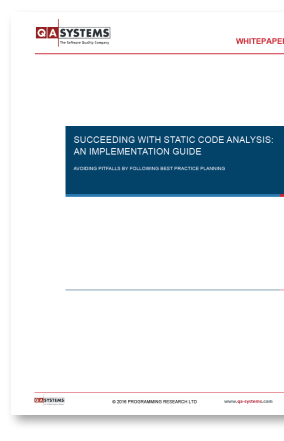
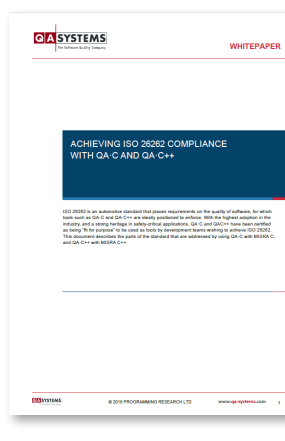
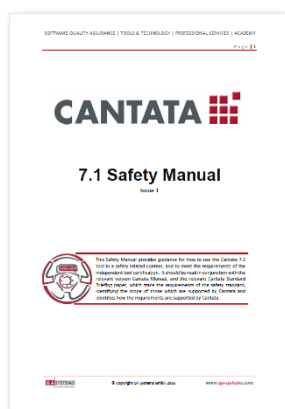
- ✓ Combination of
 - ✓ Functional test tool
 - ✓ **Cantata**
- ✓ Verification of
 - ✓ **Functional Requirements**
 - ✓ **Interfaces**
 - ✓ **Code Coverage**
- ✓ AutoTest generated test cases
 - ✓ **Verification of low level requirements**
 - ✓ Reduced effort for functional tests

Continuous Integration & Testing

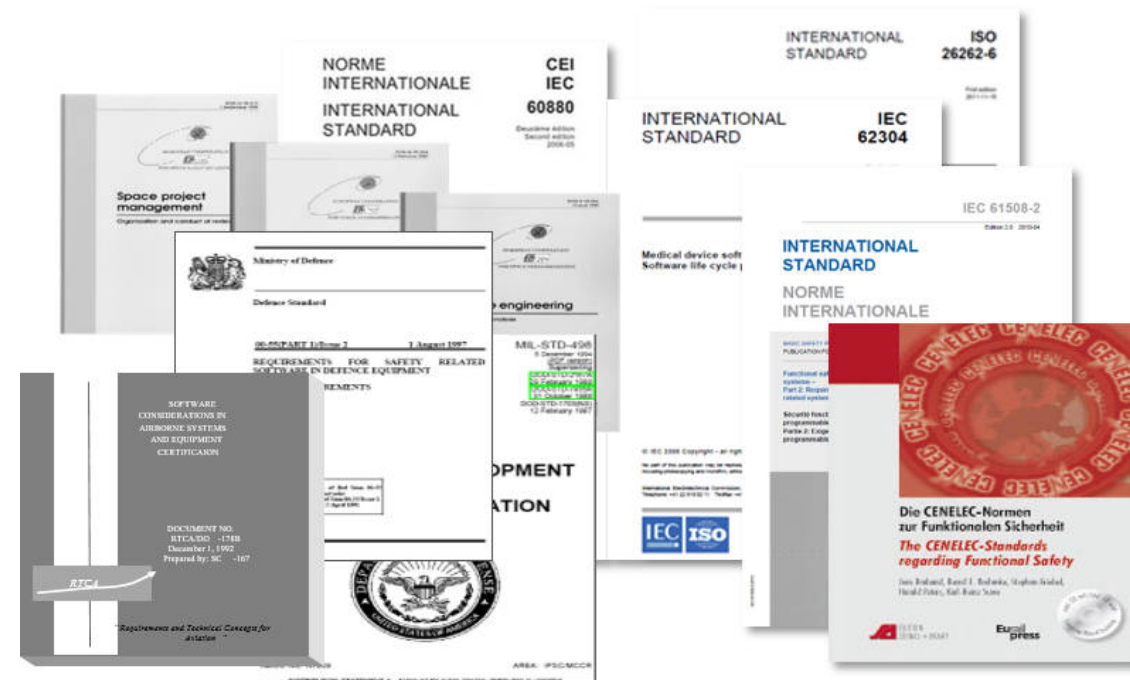
- ✓ Static Analysis & Dynamic Testing
- ✓ Continuous Testing Process

ISO 26262 Tool Certification Kits

Cantata and QA-C/C++ have been certified by SGS-TÜV SAAR GmbH, an independent third party certification body for functional safety, accredited by Deutsche Akkreditierungsstelle GmbH (DAkKS)



For all the main software safety standards...



Questions?

Thank you

Wolfram Kusterer
Sales Manager

wolfram.kusterer@qa-systems.de