Safety Certification of Software-Intensive Systems
with Reusable Components

<table>
<thead>
<tr>
<th>Report type</th>
<th>Deliverable D2.4.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report name</td>
<td>A report describing how to combine the V&amp;V methods for system certification</td>
</tr>
<tr>
<td>Dissemination level</td>
<td>PU</td>
</tr>
<tr>
<td>Status</td>
<td>Draft</td>
</tr>
<tr>
<td>Version number</td>
<td>1.5</td>
</tr>
<tr>
<td>Date of preparation</td>
<td>2013-02-20</td>
</tr>
</tbody>
</table>
### Authors

<table>
<thead>
<tr>
<th>Authors</th>
<th>E-mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIT, Egbert Althammer</td>
<td><a href="mailto:egbert.althammer@ait.ac.at">egbert.althammer@ait.ac.at</a></td>
</tr>
<tr>
<td>AIT, Erwin Schoitsch</td>
<td><a href="mailto:erwin.schoitsch@ait.ac.at">erwin.schoitsch@ait.ac.at</a></td>
</tr>
<tr>
<td>AIT, Petr Böhm</td>
<td><a href="mailto:Petr.Boehm@ait.ac.at">Petr.Boehm@ait.ac.at</a></td>
</tr>
<tr>
<td>FBK, Stefano Tonetta</td>
<td><a href="mailto:tonettas@fbk.eu">tonettas@fbk.eu</a></td>
</tr>
<tr>
<td>GMV, Carmen Lomba</td>
<td><a href="mailto:clomba@gmv.com">clomba@gmv.com</a></td>
</tr>
<tr>
<td>GMV, Elena Alaña</td>
<td><a href="mailto:ealana@gmv.com">ealana@gmv.com</a></td>
</tr>
<tr>
<td>MDS, Emmanuel Vaumorin</td>
<td><a href="mailto:vaumorin@magillem.com">vaumorin@magillem.com</a></td>
</tr>
<tr>
<td>MDS, Tesnim Abdellatif</td>
<td><a href="mailto:tesnim@magillem.com">tesnim@magillem.com</a></td>
</tr>
<tr>
<td>UPM, Clara Benac Earle</td>
<td><a href="mailto:cbenac@babel.ls.fi.upm.es">cbenac@babel.ls.fi.upm.es</a></td>
</tr>
<tr>
<td>UPM, Elena Gómez-Martinez</td>
<td><a href="mailto:egomez@babel.ls.fi.upm.es">egomez@babel.ls.fi.upm.es</a></td>
</tr>
<tr>
<td>VTEC, Cecilia Ekelin</td>
<td><a href="mailto:cecilia.ekelin@volvo.com">cecilia.ekelin@volvo.com</a></td>
</tr>
<tr>
<td>VTEC, Olle Bridal</td>
<td><a href="mailto:olof.bridal@volvo.com">olof.bridal@volvo.com</a></td>
</tr>
<tr>
<td>Version</td>
<td>Date</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>0.1</td>
<td>2012-09-28</td>
</tr>
<tr>
<td>0.2</td>
<td>2012-10-04</td>
</tr>
<tr>
<td>0.3</td>
<td>2012-11-06</td>
</tr>
<tr>
<td>0.4</td>
<td>2012-11-15</td>
</tr>
<tr>
<td>0.5</td>
<td>2012-11-16</td>
</tr>
<tr>
<td>0.6</td>
<td>2012-11-19</td>
</tr>
<tr>
<td>0.7</td>
<td>2012-12-05</td>
</tr>
<tr>
<td></td>
<td>2012-12-13</td>
</tr>
<tr>
<td>0.8</td>
<td>2012-12-18</td>
</tr>
<tr>
<td>0.85</td>
<td>2013-01-11</td>
</tr>
<tr>
<td>0.9</td>
<td>2013-01-14</td>
</tr>
<tr>
<td>1.0</td>
<td>2013-01-21</td>
</tr>
<tr>
<td>1.1</td>
<td>2013-02-07</td>
</tr>
<tr>
<td>1.2</td>
<td>2013-02-18</td>
</tr>
<tr>
<td>1.3</td>
<td>2013-02-19</td>
</tr>
<tr>
<td>1.5</td>
<td>2013-02-20</td>
</tr>
</tbody>
</table>
### List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AADL</td>
<td>Architecture Analysis &amp; Design Language</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASIL</td>
<td>Automotive Safety Integrity Level</td>
</tr>
<tr>
<td>BVA</td>
<td>Boundary Value Analysis</td>
</tr>
<tr>
<td>CAR</td>
<td>Certification Artifact Repository</td>
</tr>
<tr>
<td>CBD</td>
<td>Component Based Development</td>
</tr>
<tr>
<td>CMMI</td>
<td>Capability Maturity Model Integration (SEI, Carnegie Mellon Univ.)</td>
</tr>
<tr>
<td>CTF</td>
<td>Certification Tool Framework</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>EPF</td>
<td>Eclipse Process Framework</td>
</tr>
<tr>
<td>FDIR</td>
<td>Fault Detection, Identification and Recovery</td>
</tr>
<tr>
<td>FMEA</td>
<td>Failure Mode and Effects Analysis</td>
</tr>
<tr>
<td>FTA</td>
<td>Fault Tree Analysis</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>IP-XACT</td>
<td>Standard Structure for Packaging, Integrating, and Reusing IP (electronic design Intellectual Property)</td>
</tr>
<tr>
<td>ISO</td>
<td>International Standard Organization</td>
</tr>
<tr>
<td>MDS</td>
<td>Magillem</td>
</tr>
<tr>
<td>MyCCM</td>
<td>&quot;Make Your Component-Container Model&quot; (Component Framework)</td>
</tr>
<tr>
<td>PAS</td>
<td>Publicly available specification (IEC pre-standard-like)</td>
</tr>
<tr>
<td>PM</td>
<td>Process Model</td>
</tr>
<tr>
<td>SIL</td>
<td>Safety Integrity Level</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>V&amp;V</td>
<td>Verification and Validation</td>
</tr>
</tbody>
</table>
Table of Contents

Authors .................................................................................................................................................. 2
Revision Chart and History Log ............................................................................................................. 3
List of Abbreviations .............................................................................................................................. 4
Table of Contents .................................................................................................................................. 5
List of Figures .......................................................................................................................................... 8
List of Tables .......................................................................................................................................... 9
1 Introduction ......................................................................................................................................... 10
2 Input from Previous pSafeCer Work ................................................................................................. 11
  2.1 Static V&V techniques ..................................................................................................................... 12
  2.1.1 Static Code analysis ..................................................................................................................... 12
  2.1.2 Inspections ................................................................................................................................ 12
  2.1.3 Walkthroughs .............................................................................................................................. 12
  2.1.4 Reviews .................................................................................................................................... 12
  2.1.5 Sizing and Schedulability analyses ............................................................................................ 13
  2.1.6 Traceability analysis ................................................................................................................... 13
  2.1.7 Formal Proofs ............................................................................................................................ 13
  2.2 V&V testing techniques .................................................................................................................. 13
  2.2.1 Testing Levels ............................................................................................................................ 14
  2.2.2 Testing Strategies ........................................................................................................................ 15
  2.2.2.1 White Box Testing (Structural-Based Testing) ....................................................................... 16
  2.2.2.1.1 Fault Injection ...................................................................................................................... 16
  2.2.2.1.2 Dynamic Analysis ............................................................................................................... 16
  2.2.2.2 Black Box Testing (Behavioural-Based Testing) .................................................................... 17
  2.2.2.2.1 Interface testing .................................................................................................................. 17
  2.2.2.2.2 Stress testing ...................................................................................................................... 17
  2.2.2.2.3 Back to Back testing .......................................................................................................... 17
  2.2.2.2.4 Design-Based functional testing ......................................................................................... 17
  2.2.2.2.5 Robustness Testing ............................................................................................................. 18
  2.2.2.3 Test Data selection .................................................................................................................. 18
  2.2.2.3.1 Test data selection from Boundary Value Analysis (BVA) .................................................. 18
  2.2.2.3.2 Test Data Selection from Equivalence partitioning and Equivalence classes testing .......... 18
  2.2.2.4 Testing Analysis ...................................................................................................................... 19
  2.3 Additional model-based testing (V&V) methods ............................................................................ 19
  2.3.1 Model Checking (EN 50128: 2011) .......................................................................................... 19
  2.3.2 Model-based Testing and Test-Case Generation (IEC 61508-3 and Part 7: 2010) ................. 20
3 Requirements for V&V methods for System Certification in Various Domains ............................... 22
  3.1 Automotive Domain ....................................................................................................................... 22
  3.1.1 Verification and validation recommendations in ISO 26262 .................................................... 24
  3.2 Avionics Domain ............................................................................................................................. 25
3.3 Railway Domain ........................................................................................................................................27
  3.3.1 Example RAIL_EN50126_TRAINTIC_000, EN50126 ........................................................................29
  3.3.2 Example RAIL_EN50128_AIT_01, EN50128 ...................................................................................29
  3.3.3 Example RAIL_EN50129_AIT_01, EN50129 ...................................................................................30

3.4 Space ..........................................................................................................................................................31

3.5 Cross Domain/Other Domains ................................................................................................................32
  3.5.1 Cross Domain/Other Domains: V&V requirements ...........................................................................33
  3.5.2 Cross Domain/Other Domain: Identified V&V requirements ..........................................................33

4 Existing V&V methods for system certification ..........................................................................................36
  4.1 Automotive Domain ..............................................................................................................................36
  4.2 Avionics Domain ....................................................................................................................................37
  4.3 Railway Domain ....................................................................................................................................39
    4.3.1 EN50128, Table A.5 Verification and Testing .................................................................................39
    4.3.2 EN50128, Table A.19 Static Analysis ...............................................................................................42
    4.3.3 EN50128, Table A.8 Software Analysis Techniques ........................................................................44

4.4 Space Domain ..........................................................................................................................................45
  4.4.1 V&V for Critical Software .................................................................................................................45
  4.4.2 V&V strategies for Critical Software ..................................................................................................45

4.5 Cross Domain/Other Domains ................................................................................................................47
  4.5.1 IEC 61508 – 3: How Techniques are described and recommended ...................................................48
  4.5.2 Time-triggered Architecture .............................................................................................................54

4.6 Comparative table of V&V methods for system certification .....................................................................57

5 Adaptation and combination of existing methods and tools to system certification .....................................59
  5.1 Adaptation and combination of OCRA contract verification methods ....................................................59
    5.1.1 Contract-based verification ................................................................................................................59
    5.1.2 Combination with fault-free analysis ................................................................................................59
    5.1.3 Combination with formal requirements validation ..........................................................................60

5.2 Adaptation for automotive domain certification ........................................................................................60

5.3 Adaptation and combination of methods and tools for avionic domain certification ...................................60
    5.3.1 Mapping Objectives ..........................................................................................................................61
      5.3.1.1 Mapping low-level requirements and software architecture ......................................................61
      5.3.1.2 Mapping resources performance description and hardware architecture ................................62
    5.3.2 Managing Versioning Issues ............................................................................................................63
      5.3.2.1 Changes detection in hardware architecture ..............................................................................63
      5.3.2.2 Changes detection in the software architecture ........................................................................64
      5.3.2.3 Changes detection in integration requirements ...........................................................................65
    5.3.3 Computing impacts ............................................................................................................................65

5.4 Adaptation of Model-Based mutation-based Testing and Test Case Generation ........................................65

6 Contribution to Overall pSafeCer Objectives ............................................................................................68

7 Conclusions ..................................................................................................................................................69

© The SafeCer Consortium
9.1 Traceability for Avionic domain

9.1.1 Software Life Cycle Development

9.1.1.1 Software Requirements Process

9.1.1.2 Software Design Process

9.1.1.3 Software Coding Process

9.1.1.4 Software Integration Process

9.1.2 Requirements Traceability Activities and Objectives

9.1.3 Software Integral Processes

9.1.4 Software Production for Airborne Systems

9.2 Further examples for Cross-Domain/Other Domain V&V techniques and measures

9.3 Time-triggered Architecture
List of Figures

Figure 1: Types of testing. .................................................................14
Figure 2: Verification lifecycle. .........................................................15
Figure 3: Testing methods: White box testing, black box testing and test data selection ..........16
Figure 4: Example of a Software Safety Lifecycle (EN 50128). ..................................................30
Figure 5: Simplified picture of domain set/superset of requirements for cross/other domain........33
Figure 6: ”Model based testing”- technique for Software Design and development in IEC 61508-3. .................................................................50
Figure 7: Rigour level and SIL ...........................................................50
Figure 8: Properties of ”Model-based Testing” as technique described in IEC 61508-3, Table C.5, for systematic integrity, software module testing and integration ........................................53
Figure 9: Combined process for “Build-for-reuse” (left part) and “Build-by-reuse” (right part)......55
Figure 10: Vendor-oriented component testing ............................................................................56
Figure 11: Validation process for adapted and customized components .......................................56
Figure 12: Mapping Low-level Requirements to Software Task components ...............................62
Figure 13: Mapping Resources Performance Description to Hardware components ..................63
Figure 14: Changes Detection in the Hardware Architecture .......................................................64
Figure 15: Changes Detection in the Software Architecture ........................................................64
Figure 16: Impacts Computation ..................................................................................................65
Figure 17: Automatic Test Case Generation ..................................................................................66
Figure 18: Combined Black-Box/White-Box Testing ...................................................................67
Figure 19: Software Development Life Cycle Overview ................................................................72
Figure 20: Integration Requirements ............................................................................................74
Figure 21: Overall Integration Traceability ...................................................................................75
Figure 22: Overall certification processes interactions .................................................................76
Figure 23: Functional and Black-Box Testing - model-based testing (MBT) referenced .................77
Figure 24: Dynamic Analysis and Testing – model-based testing (MBT) referenced .......................78
Figure 25: Detailed properties of MBT-Test Case generation and execution for functional and black-box testing ...............................................................80
Figure 26: Detailed properties of MBT-TCG and execution for functional and black-box testing....81
Figure 27: IEC 61508-3: “Time-triggered Architecture” as technique for “Software Architecture Design” ........................................................................83
List of Tables

Table 1: Use of verification methods throughout the development life cycle [Storey96] completed with some ISO 26262 recommended methods ................................................................. 11
Table 2: Testing levels ........................................................................................................................................ 14
Table 3: V&V requirements from automotive domain .................................................................................. 22
Table 4: V&V requirements from avionics domain ....................................................................................... 25
Table 5: V&V requirements from railway domain ......................................................................................... 28
Table 6: IEC 61508 Safety integrity levels – target failure measures for a safety function operating in high demand mode of operation or continuous mode of operation .................................. 34
Table 7: V&V Requirements from cross domain/other domain ................................................................. 34
Table 8: EN50128, Table A.5 Verification and Testing .............................................................................. 39
Table 9: EN50128, Table A.19 Static Analysis. .......................................................................................... 42
Table 10: EN50128, Table A.8 Software Analysis Techniques .............................................................. 44
Table 11: Verification techniques for the space domain ........................................................................... 46
Table 12: Validation techniques for the space domain ............................................................................. 46
Table 13: Verification techniques for the space domain: Functional, Design and Code level ....... 46
Table 14: Comparative table of V&V techniques for system certification ............................................. 57
1 Introduction

The certification process includes the activities of software verification, system verification and safety. In these activities components have an important role: safety assessment is concerned with how unsafe conditions may arise in the interaction among components; system and software verification examines the components structure to divide and conquer the analysis problem.

pSafeCer will define a generic component model (output of WP2.2) to be used in the integrated certification and development process (output of WP2.1). The component model will feature component contracts and will be exploited to enable a compositional safety argumentation (output of WP2.3) and a compositional verification and validation (output of WP2.4).

The main objective of this document is to describe different existing V&V methods to support system certification, and how to adapt them to the pSafeCer requirements. In the present document, we outline this adaption, which will be developed in nSafeCer project.

To lead this adaptation, we consider the requirements collected in the pSafeCer deliverable D1.0.2 and how they are certified in the different domains considering also the software development process to verify these requirements.

This document is structured as follows: The input from previous pSafeCer work that is relevant for this deliverable is presented in Section 2, with the requirements gathered in D1.0.2 presented in more detail in Section 3. Section 4 describes V&V methods for system certification in different domains, and it identifies its possible shortcomings. The adaptation and combination of existing V&V methods to system certification is outlined in Section 5. Section 4.6 provides a summary of this document, by means of comparative tables of V&V methods for system certification. The contributions to the overall pSafeCer objectives are presented in Section 6. Lastly, some conclusions and future work is discussed in Section 7.
2 Input from Previous pSafeCer Work

This document takes as input, from D1.0.2, a large amount of requirements in different application domains. Due to their size, we have considered only those related to V&V and that are applicable to our target domains.

Moreover, the existing V&V techniques, presented in D2.4.1, are also considered in order to classify the certification methods in each specific domain.

Table 1: Use of verification methods throughout the development life cycle [Storey96] completed with some ISO 26262 recommended methods

<table>
<thead>
<tr>
<th>Development cycle stage</th>
<th>Static methods</th>
<th>Dynamic methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements analysis and functional specification</td>
<td>Walkthroughs, reviews, checklists, formal validation techniques, FTA, ETA</td>
<td>Prototyping/requirements animation, Functional simulation</td>
</tr>
<tr>
<td>Top-level design</td>
<td>Walkthroughs, reviews, checklists, formal proofs, inspection</td>
<td>Design simulation</td>
</tr>
<tr>
<td>Detailed design</td>
<td>Walkthroughs, reviews, control flow analysis, data flow analysis, symbolic execution, checklists, inspection, FMEA, FDIR effectiveness analysis</td>
<td>Design simulation, prototyping</td>
</tr>
<tr>
<td>Implementation</td>
<td>Static code analysis, semantic code analysis, semi-formal and formal verification</td>
<td>Functional testing, boundary value analysis, structured-based testing, requirements-based testing, probabilistic testing, error guessing, process simulation, error seeding</td>
</tr>
<tr>
<td>Integration testing</td>
<td>Walkthroughs, design reviews, sneak circuit analysis</td>
<td>Functional testing, time and memory tests, boundary value analysis, requirements-based testing, performance testing, stress testing, probabilistic testing, error guessing, back-to-back test, resource usage test, interfaces testing, fault injection test, test of interaction/communication, field experience test, long term test, user test under real-life conditions</td>
</tr>
<tr>
<td>Complete system validation and verification.</td>
<td>Reviews</td>
<td>Functional testing, long-term tests, user tests under real-life conditions, black box testing, simulation, tests under boundary conditions, fault injection, durability tests, stress tests, highly accelerated life testing (HALT)</td>
</tr>
</tbody>
</table>

Next, some of these methods are briefly outlined. A detailed description of each method can be found in D2.4.1.
2.1 Static V&V techniques

In this section, we give an overview of static methods to verify and validate software and hardware components and their integration. The common characteristic of static methods is that the analysis is performed without the need to run the system.

2.1.1 Static Code analysis

Static code analysis is intended to identify errors in the source code before being tested and integrated. This can involve checking of compliance with coding standards, gathering of software metrics, analysis of control flow and analysis of data flow. As well, it can be applied to check the consistency of source code with a formal mathematical specification.

The use of this kind of analysis may provide graphical representations of the code (e.g. control graphs). Static analysis provides benefits such as:

- Identification of suspicious source code to remove.
- Analysis of all blocks.
- Analysis of all execution paths or other suitable metrics (e.g. branches).
- Test data ranges.
- Identify errors such as semantic ones that can lead to errors such as buffer overflows.
- Others.

This approach is quite fast and has no runtime overhead. It also has the advantage that it can cover the complete program code.

2.1.2 Inspections

Software requirements, design and implementation are examined to detect faults. Inspections are very structured and they require detailed reviews with defined roles for participants.

It is a very effective technique for finding errors. Many defects may be discovered in a single inspection. It is typically performed against a checklist.

2.1.3 Walkthroughs

Walkthroughs represent an evaluation technique in which a developer leads one or more other members of the development team through a document, and presenting/describing it. The other members ask questions and make comments about technique, style, identification of possible errors, violations of development standards, or other problems.

It is less formal than inspections and often reveals as a high-level overview.

2.1.4 Reviews

Reviews are systematic examinations of items for the purpose of assessing the results obtained at a given time in the project, by persons not themselves responsible for the project.

Whereas inspections and walkthroughs concentrate on assessing correctness, reviews seek to ascertain that tolerable levels of quality are being attained. They do not focus on discovering technical flaws but on ensuring that the design and development fully and accurately address the needs of the applications.
2.1.5 Sizing and Schedulability analyses

Sizing and Schedulability analyses represent key techniques to analyse limited but important implicit characteristics of embedded real-time critical software (e.g., predictability, resource usage, etc.).

Schedulability analysis techniques allow predicting whether the system accomplishes the deadlines it was designed for. It is a mathematical technique which takes information about timing constraints and execution times of tasks and interrupt handlers, and allows to calculate their worst-case response times, determining if they can be guaranteed to always meet their deadlines. Schedulability Analysis is a technique able to verify that all critical tasks are schedulable (i.e. that they can be executed within their deadlines) under worst-case execution time conditions.

The Schedulability Analysis is supported by tools that allow the off-line static verification of real-time systems, the simulation of their run-time behaviour and the evaluation of the worst case execution time.

2.1.6 Traceability analysis

Traceability analysis checks that every requirement has been implemented in the design and code. In addition, the design and code must have their basis in the requirements. Tests are also traced to verify that the results are in agreement with the requirements specified.

2.1.7 Formal Proofs

Formal proofs attempt to demonstrate, logically, that software is correct. They check that all input preconditions will result in defined post conditions being met.

Formal proofs do not need to target the complete partial correctness problem, it can focus on specific aspects as well (e.g., formal proof of absence of run-time errors). Indeed, formal proof is just a form of static analysis (see Section 2.1.1) for source code verification.

2.2 V&V testing techniques

Testing demonstrates that a software product satisfies its requirements. From verification and validation points’ of view:

- Validation: The process of ensuring that an artifact, typically a requirements specification or a final product, is consistent with the actual needs of the users and other stakeholders.
- Verification: The process of ensuring that a result from a system development activity fulfils its applicable requirements.

The testing procedure describes the activities to verify an element through the test execution. It covers:

1. Tests design;
2. Specification of test cases;
3. Specification of test procedures;
4. Execution of tests and record of results;
5. Analysis of results.

Figure 1 illustrates the different approaches that can be followed.
Figure 1: Types of testing.

The testing procedure has to take into account the different types of testing: testing can be applied at different levels, using various strategies, and following a specific objective.

2.2.1 Testing Levels

As previously shown, there are different testing levels. Testing shall be performed in accordance with a strategy for each testing level, see table below.

<table>
<thead>
<tr>
<th>SOFTWARE PHASE</th>
<th>TESTING LEVELS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software goal</td>
<td>Acceptance testing</td>
</tr>
<tr>
<td>Requirements</td>
<td>System testing</td>
</tr>
<tr>
<td>Design</td>
<td>Integration testing</td>
</tr>
<tr>
<td>Code</td>
<td>Unit Testing</td>
</tr>
</tbody>
</table>

- **Unit Testing**: Testing individual software components or modules.
- **Integration Testing**: Integration testing checks the interaction among units (assembled components that must have been tested and accepted previously) and verifies the implementation of the software requirements and software components within the software architecture: the process of testing units against the architectural design. It can be conducted using either a top-down or a bottom-up strategy.
- **System (or Software) Testing**: The aim is to determine if the system (or software) meets the requirements. It focuses on the functionality rather than on the implementation. Tests cases are derived from the specification. Possible techniques are black box testing and test data selection.
- **Acceptance testing**: This testing activity demonstrates that the software addresses the requirements specified by the customer.
- **Regression testing**: Tests conducted after a change is done. Automated testing tools are used for this type of testing.

Figure 2 illustrates the relation between software development phases and testing levels. The dotted lines represent testing.
Once the criticality of the software is determined, the test coverage goal for each testing level shall be agreed between the customer and the supplier. Each test must define and guarantee:

1. Test procedures and data must be adequate and traceable.
2. Tests must satisfy requirements.
3. Test coverage must be checked considering the stated goals.
4. Non-conformance and problems must be reported.
5. Independent software verification & validation (ISVV) shall be performed for highly critical software:
   a. Only applicable where the risks associated with the project justify the cost involved. The purchaser may also consider a less rigorous level of independence, e.g. an independent team in the same organization.
   b. ISVV is not considered to be merely “independent” testing of the product. The concept of ISVV includes the necessity of setting up an independent team of highly qualified staff composed of specialists from all disciplines including software product assurance. This team, independently of the development team, performs verification activities such as conducting reviews, inspections, testing and auditing.

### 2.2.2 Testing Strategies

Testing is mainly divided into:

- White box testing (also called structural techniques). To determine test cases from knowledge of the internal logic of the software. Test cases are derived from the program structure.
- Black box testing (also called behavioural techniques). To determine the software functionality as a whole. Tests are based on requirements and functionality.

But other additional testing techniques exist. For instance, testing input data range testing, such as test data selection. Figure 3 depicts these testing methods.
2.2.2.1 White Box Testing (Structural-Based Testing)

Fault Injection

Focussing on software systems, fault injection is used to evaluate dependability of computer systems. Software methods are used to produce changes at the software state level, injecting faults to corrupt the internal states of the software to test its faults tolerance. This technique is used to develop high quality reliable code and to reveal how software behaves under experimentally controlled anomalous circumstances. It is also useful to evaluate safety, simulating failures in real software environments and estimating worst-case scenarios. It improves the coverage of a test.

This form of software testing differs from traditional black-box testing. It provides ways to assess how faults propagate across the software, and if the “mitigation means” take place, as designed. Different test cases should be designed to determine the software behaviour.

Faults or errors are introduced and the effect is studied to determine the effect of a corruption. When an anomalous input is introduced but it is not propagated, the software has run correctly (fault tolerance). The software works satisfactorily under a hostile external environment.

It is useful to test the robustness and failure immunity of the design, but it should only be applied in areas where this is an important feature.

Nevertheless, fault injection is recognized as a key element in the assessment and validation of critical systems, as it is a practical approach performing stress testing, i.e. raising conditions to trigger rarely executed software operations such as error handling and recovery. Fault injection is used to assess the FDIR mechanisms of any system.

2.2.2.1.2 Dynamic Analysis

This analysis shows the system behaviour when it is executed, with execution performed according to its intended usage. In other words, the dynamic analysis carries out the offline analysis of results from online test.

The main purpose of the dynamic analysis is to determine the degree of source code coverage obtained during testing, and also to ensure that the control flows are free of problems, such as design or code elements that are unreachable or incorrectly reached.
An analysis of the code coverage obtained by testing, and the identification of code portions that have not been executed by the tests can identify paths that are not physically possible to follow, because of some conditions or constraints of which the compiler was unaware. Branches that cannot be executed should be eliminated from the source code (dead code).

In addition, dynamic analysis identifies problems due to interaction of multiple functions, interaction with other processes and the operating system itself. It exposes indications of non-functional issues: performance, scalability and reliability.

These real problems are discovered with high precision and the process is easy to automate. Dynamic analysis can be as fast as program execution. Examples of dynamic analysis techniques are:

- Decision coverage.
- Branch coverage.
- Statement coverage.

2.2.2.2 Black Box Testing (Behavioural-Based Testing)

2.2.2.2.1 Interface testing

Interface testing is used to detect faults in the specification, design and/or implementation of the interfaces of the software components. Test cases are built with data that test all module interfaces.

Several levels of detail or completeness of testing are feasible. The most important levels are tests for:

- All interface variables at their extreme values.
- All interface variables individually at their extreme values, with other interface variables at normal values.
- All values of the domain of each interface variable, with other interface variables at normal values.
- All values of all variables in combination (this is only feasible for simple interfaces).

The specified test conditions relevant to each call of each subroutine.

2.2.2.2.2 Stress testing

Stress is used to evaluate the software stability. It executes the system beyond its maximum design load; this is beyond its specifications to check how and when it fails.

This technique is useful for distributed systems. Examples of stress tests are: complex database queries, activation of asynchronous events such as interrupts and semaphore actions at the highest frequency, etc.

Most software has capacity limits, sometimes hidden, and these limits must be determined and checked. The main disadvantage is that it requires large resources.

2.2.2.2.3 Back to Back testing

This technique is based on applying the same test to different versions of the software and comparing the outputs. Discrepancies are analysed to check whether or not they indicate a fault.

2.2.2.2.4 Design-Based functional testing
Design-based functional testing is a top-down approach (Functional Design Tree).
Firstly a functional hierarchy is constructed. Then, for each function at each level, extreme, non-extreme and special value test data are identified. In addition, the test data that will generate these values are identified.

2.2.2.5 Robustness Testing

Robustness testing aims at testing the response of the software at any level to invalid inputs. It is more focused on evaluating specific dependability aspects of the software.

For instance, robustness test cases include:
- For time-related functions, such as filters, integrators and delays, test cases should be developed for arithmetic overflow protection mechanisms.
- For state transitions, test cases should be developed to provoke transitions that are not allowed by the software requirements.

2.2.2.3 Test Data selection

2.2.2.3.1 Test data selection from Boundary Value Analysis (BVA)

Boundary value analysis is a testing technique to verify unusual or extreme situations that the code should be able to handle. So, it focuses on the possible boundary inputs to identify test cases. Examples of possible inputs are:
- Negative values;
- Zero value;
- Maximum and minimum values;
- Empty files;
- Sequencing errors;
- Missing parameters;
- Full/Empty tables;
- Boundary Value Analysis complements Equivalence Partitioning.

2.2.2.3.2 Test Data Selection from Equivalence partitioning and Equivalence classes testing

This technique is intended to test the software using a minimum of test data.
Equivalent partitioning divides the input domain into a finite number of sub-domains (called equivalence classes) for the selection of test inputs. There have to be designed enough tests to cover each partition. It is assumed that the test of a representative value of each class is equivalent to a test of any other value.

Equivalent classes can be defined according to the following conditions:
- If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
- If an input condition calls for a specific value, then one valid and two invalid equivalence classes are defined.
- If an input condition specifies a member of a set, then one valid and one invalid equivalence class are defined.
• If an input condition is Boolean, then one valid and another invalid equivalence class are defined.

Equivalent partitioning used to test dependability and safety of software focuses the test cases on the limit values or ‘out of bound’ values for each class identified.

These tests cases are complemented with those of Boundary Value Analysis.

### 2.2.2.4 Testing Analysis

The purpose of testing analysis is to assure the completeness of test activities. It is usually required at the end of development. However, it may be conducted after completion of module, integration, or system testing.

The reviewer should ask questions about:

- Whether the test objectives were met.
- The nature of major anomalies.
- Check that the anomalies existed only in the product and were not caused by the test cases and procedures.

In case of “cross-domain” and “other domain” methods for certification and validation a generic approach is taken here, since the basic methods of the different domains apply in many cases. From generic standards like IEC 61508-3 (software-intensive systems, SW items) and new proposals for “Dependability Case” standards and specifications (for example from IEC TC 56), the methods for describing appropriateness of processes, measures and techniques for different SILs are taken as examples of a general approach to assessment and description of V&V methods for component qualification and system certification.

### 2.3 Additional model-based testing (V&V) methods

In this chapter, model-based techniques/measures for V&V are listed, as they are addressed in EN 50128 (June 2011) and IEC 61508-3, Ed. 2.0 (2010).

#### 2.3.1 Model Checking (EN 50128: 2011)

**Aim:**

Given a model of a system, test automatically whether this model meets a given specification.

**Description:**

Model checking is the process of checking whether a given structure is a model of a given logical formula. The concept is general and applies to all kinds of logics and suitable structures. A simple model-checking problem is testing whether a given formula in the propositional logic is satisfied by a given structure.

An important class of model checking methods have been developed to algorithmically verify formal systems. This is achieved by verifying if the structure, often derived from a hardware or software design, satisfies a formal specification, typically a temporal logic formula.

Model checking is most often applied to hardware designs. For software, because of undecidability (see Computability theory) the approach cannot be fully algorithmic; typically it may fail to prove or disprove a given property.

The structure is usually given as a source code description in an industrial hardware description language or a special-purpose language. Such a program corresponds to a finite state machine, i.e., a directed graph consisting of nodes (or vertices) and edges. A set of atomic propositions is associated with each node, typically stating which memory elements are one. The nodes represent states of a system; the edges represent possible transitions which
may alter the state, while the atomic propositions represent the basic properties that hold at a point of execution.

Formally, the problem can be stated as follows: given a desired property, expressed as a temporal logic formula \( p \), and a structure \( M \) with initial states, decide if \( M \) is finite, as it is in hardware, model checking reduces to a graph search.

2.3.2 Model-based Testing and Test-Case Generation (IEC 61508-3 and Part 7: 2010)

NOTE: This technique/measure is referenced in table A.5 (and C.5) of IEC 61508-3.

Aim:

Facilitate efficient automatic test case generation from system models and generate highly repeatable test suites.

Description:

Model-based Testing (MBT) is a black-box approach in which common testing tasks such as test case generation (TCG) and test results evaluation are based on a model of the system (application) under test (SUT). Typically, but not only, the systems data and user behaviour are modelled using Finite state machines, Markov processes (Prowell, 2005), decision tables or the like (El-Far, 2001, generalized). Additionally, model-based testing can be combined with source code level test coverage measurement, and functional models can be based on existing source code.

Model-based Testing is the automatic generation of efficient test cases/procedures using models of system requirements and specified functionality (SoftwareTech, 2009).

Since testing is very expensive, there is a huge demand for automatic test case generation tools. Therefore, model-based testing is currently a very active field of research, resulting in a large number of available TCG (Test Case Generation) tools. These tools typically extract a test suite from the behavioural part of the model, guaranteeing to meet certain coverage requirements.

The model is an abstract, partial representation of the system under test's (SUT) desired behaviour. From this model, test models are derived, building an abstract test suite. Test cases are derived from this abstract test suite and executed against the system, and tests can be run against the system model as well. MBT with TCG is based on and strongly related to use of formal methods, so recommendations are similar with respect to safety integrity levels (SIL): HR (highly recommended) for higher SILs, and not required for lower SILs.

The specific activities in general are:

- build the model (from system requirements)
- generate expected inputs
- generate expected outputs
- run tests
- compare actual outputs with expected outputs
- decide on further action (modify model, generate more tests, estimate reliability/quality of the software)

Tests can be derived with different methods and techniques for expressing models of user/system behaviour, e.g.

- by using decision tables
- by using finite state machines
- by using grammars
- by using Markov Chain models
- by using state charts
• by theorem proving
• by constraint logic programming
• by model checking
• by symbolic execution
• by using an event-flow model
• reactive system tests: parallel hierarchical finite automaton
• ..etc.

Model-based Testing is specifically targeting recently the safety critical domain. It allows for early exposure of ambiguities in specification and design, provides the capability to automatically generate many non-repetitive efficient tests, to evaluate regression test suites and to assess software reliability and quality, and eases updating of test suites.
3 Requirements for V&V methods for System Certification in Various Domains

The present document aims to describe how to combine the V&V methods for system certification. As first step, we analyse the requirements needed to certify system in each domain of those considered in pSafeCer project and collected in deliverable D1.0.2. These requirements will be used for adapting generic V&V methods.

3.1 Automotive Domain

The following table contains requirements from the automotive domain that address verification and validation. The requirements have been collected from the pSafeCer deliverable D1.0.2. Note that a few cross-domain requirements that are considered applicable to the automotive V&V are included.

<table>
<thead>
<tr>
<th>Requirement Id</th>
<th>Requirement</th>
<th>V&amp;V method</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT_ISO26262_VTEC_46</td>
<td>The automotive instantiation of the integrated certification and development process shall include safety analysis of the system design with respect to systematic faults</td>
<td>Design safety analysis</td>
</tr>
<tr>
<td>AT_ISO26262_VTEC_48</td>
<td>The CAR shall be able to store (or at least reference) and handle integration testing specifications</td>
<td>Integration testing</td>
</tr>
<tr>
<td>AT_ISO26262_VTEC_50</td>
<td>The CAR shall be able to store (or at least reference) and handle system validation reports</td>
<td>System validation</td>
</tr>
<tr>
<td>AT_ISO26262_VTEC_56</td>
<td>The automotive instantiation of the integrated certification and development process shall include verification that the hardware safety requirements are appropriate, as described in ISO26262-5, section 6.</td>
<td>Safety requirements verification</td>
</tr>
<tr>
<td>AT_ISO26262_VTEC_59</td>
<td>The automotive instantiation of the integrated certification and development process shall include hardware safety analysis as described in ISO26262-5, section 7.</td>
<td>Hardware safety analysis</td>
</tr>
<tr>
<td>AT_ISO26262_VTEC_61</td>
<td>The CAR shall be able to store (or at least reference) and handle hardware design verification reports that describe how it has been verified that the hardware design is appropriate.</td>
<td>Hardware design verification</td>
</tr>
<tr>
<td>AT_ISO26262_VTEC_71</td>
<td>The automotive instantiation of the integrated certification and development process shall include hardware integration and testing as described in ISO26262-5, section 10.</td>
<td>Hardware integration and testing</td>
</tr>
<tr>
<td>AT_ISO26262_VTEC_73</td>
<td>The integrated certification and development process shall include a planning of the software verification</td>
<td>Software verification</td>
</tr>
<tr>
<td>Requirement Id</td>
<td>Requirement</td>
<td>V&amp;V method</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>AT_ISO26262_VTEC_82</td>
<td>The automotive instantiation of the integrated certification and development process shall include verification that the software safety requirements are appropriate, as described in ISO26262-6, section 6.</td>
<td>Software safety requirements verification</td>
</tr>
<tr>
<td>AT_ISO26262_VTEC_98</td>
<td>The integrated certification and development process shall include a verification that the software architectural design is appropriate</td>
<td>Software architecture design verification</td>
</tr>
<tr>
<td>AT_ISO26262_VTEC_103</td>
<td>The integrated certification and development process shall include analytical and empirical verification of software components</td>
<td>Analytical and empirical verification</td>
</tr>
<tr>
<td>AT_ISO26262_VTEC_105</td>
<td>The integrated certification and development process shall include software integration testing to demonstrate that the software architectural design is realized by the software</td>
<td>Software integration testing</td>
</tr>
<tr>
<td>AT_ISO26262_VTEC_114</td>
<td>The integrated certification and development process shall include verification of configuration data with respect to value-within-allowed-range and compatibility with other configuration data</td>
<td>Verification configuration data (value-within-range)</td>
</tr>
<tr>
<td>AT_ISO26262_VTEC_136</td>
<td>The automotive instantiation of the integrated certification and development process shall include verification of the safety requirements with respect to their compliance with the requirements defined in ISO 26262. At least the two methods 'verification by inspection' and 'semi-formal verification' shall be covered.</td>
<td>Inspection and Semi-formal verification</td>
</tr>
<tr>
<td>AT_ISO26262_AVL_012</td>
<td>The safety arguing technique shall allow to create safety cases that are traceable to FMEA results.</td>
<td>FMEA</td>
</tr>
<tr>
<td>AT_ISO26262_AKH_02</td>
<td>CTF should include a code coverage analysis tool at level statement, branch and MC/DC level (DO-178B – ISO26262)</td>
<td>Code coverage analysis</td>
</tr>
<tr>
<td>AT_ISO26262_AKH_06</td>
<td>CTF should include tools for semi-automated consistency checking within different system safety analyses (e.g. PHA, FTA, and FMEA) (ISO26262)</td>
<td>Consistency checking</td>
</tr>
<tr>
<td>CROSS_FBK_000</td>
<td>pSafeCer shall support formal methods for properties, architecture, and behavioural specification, validation and verification.</td>
<td>Formal verification</td>
</tr>
<tr>
<td>CROSS_DO-178B_AKH_17</td>
<td>CTF should include a tool for Test execution on host, simulator, and embedded target systems (DO-178B –ISO26262)</td>
<td>Test execution</td>
</tr>
</tbody>
</table>

As can be seen the requirements are not very detailed regarding V&V methods but only mention a few classes (testing, inspection, analysis). The details are instead expected to be derived from the specific standard, typically ISO 26262.
3.1.1 Verification and validation recommendations in ISO 26262

The ISO 26262 standard for functional safety of automotive electronic systems addresses verification and validation at the following levels:

- Verification of system design
- Verification of software architectural design
- Analytical verification of software unit design and implementation
- Software unit testing
- Software integration testing
- Verification of fulfilment of software safety requirements
- Verification of hardware design
- Hardware probabilistic metrics evaluation
- Hardware integration testing
- Hardware-software integration testing
- System integration testing
- Vehicle integration testing
- In-vehicle safety validation

Specific methods to use are given with varying degree of strength (highly recommended, recommended) depending on ASIL i.e. Automotive Safety Integrity Level. In many cases, methods should be combined. An example is given below (clause 10.4.3 in Part 6 of ISO 26262), showing how software integration testing is addressed.

---

### 10.4.3 The software integration test methods listed in Table 13 shall be applied to demonstrate that both the software components and the embedded software achieve:

- compliance with the software architectural design in accordance with Clause 7;
- compliance with the specification of the hardware-software interface in accordance with ISO 26262-4:2011, Clause 7;
- the specified functionality;
- Robustness (Example: Absence of inaccessible software; effective error detection and handling) and sufficient resources to support the functionality.

<table>
<thead>
<tr>
<th>Table 13 — Methods for software integration testing Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Requirements-based testa</td>
<td>++</td>
</tr>
<tr>
<td>1b Interface test</td>
<td>++</td>
</tr>
<tr>
<td>1c Fault injection testb</td>
<td>+</td>
</tr>
<tr>
<td>1d Resource usage testcd</td>
<td>+</td>
</tr>
<tr>
<td>1e Back-to-back comparison test between model and code, if applicablee</td>
<td>+</td>
</tr>
</tbody>
</table>

*a The software requirements at the architectural level are the basis for this requirements-based test.

*b This includes injection of arbitrary faults in order to test safety mechanisms (e.g. by corrupting software or hardware components).

*c To ensure the fulfillment of requirements influenced by the hardware architectural design with sufficient tolerance, properties such as average and maximum processor performance, minimum or maximum execution times, storage usage (e.g. RAM for stack and heap, ROM for program and data) and the bandwidth of communication links (e.g. data buses) have to be determined.

*d Some aspects of the resource usage test can only be evaluated properly when the software integration tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests.

*e This method requires a model that can simulate the functionality of the software components. Here, the model and code are stimulated in the same way and results compared with each other.
The following observations, based on section 4.2 ('Interpretations of tables') of Parts 2-9 of ISO26262, are vital for the understanding of the ISO 26262 requirement example and table given above:

- A rationale has to be provided that the selected combination of methods complies with the requirement. Possibly, one valid rationale could simply be that all the listed methods are used but this is not completely clear in the standard.
- Methods listed as ‘++’ are highly recommended while methods listed as ‘+’ are only recommended. As stated in ISO 26262, highly recommended methods “should be preferred” but what this means with respect to the fulfilment of the corresponding requirement is not clear.
- The requirement is given as “The software integration test methods listed in Table 13 shall be applied...” but this seems to be in conflict with section 4.2 of Parts 2-9 of the standard that states “an appropriate combination of methods shall be applied in accordance with the ASIL indicated, independent of whether they are listed in the table or not”.
- The purpose of the distinction between ‘+’ and ‘++’ indications in ISO 26262 is not clear since a rationale has to be created anyway that the selected combination of methods is appropriate.
- The methods in the requirement we looked at are numbered as 1a, 1b, 1c etc. In ISO 26262, some requirements use the numbering scheme 1, 2, 3, etc. How to interpret the associated tables of methods is not completely clear in that case either.
- To summarize, ISO 26262 fails to define clear criteria for determining whether the quoted requirement 10.4.3 in ISO 26262-6 is fulfilled or not. This conclusion is valid for all requirements in ISO 26262 that are based on tables using ‘+’ and ‘++’ and does not just concern verification and validation.

### 3.2 Avionics Domain

The following table contains the main Avionics domain verification and validation objectives for certification. They are based on the guidance DO-178C for avionics. In the context of the work we are also going to present the requirements for traceability and re-certification. Those requirements are linked to the DO-178B certification document but are also applicable to the DO-178C certification version. For example, requirements AV_DO178B_THALES_14, AV_DO178B_THALES_16, AV_DO-178B_AKH_11 and AV_DO-178B_AKH_18 are linked to requirement AV_DO178C_MDS_05 for traceability requirements.

<table>
<thead>
<tr>
<th>Requirement Id</th>
<th>Requirement</th>
<th>V&amp;V method</th>
</tr>
</thead>
<tbody>
<tr>
<td>AV_DO178C_MDS-01</td>
<td>Software reviews and analyses of High-level requirements (Section 6.3.1 of the guidance)</td>
<td>Requirement validation</td>
</tr>
<tr>
<td>AV_DO178C_MDS-02</td>
<td>Software reviews and analyses of Low-level requirements and Software architecture (Section 6.3.2 and 6.3.3 of the guidance)</td>
<td>Verification of Outputs of Software Design Process &amp; requirement validation</td>
</tr>
<tr>
<td>Requirement Id</td>
<td>Requirement</td>
<td>V&amp;V method</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AV_DO178C_MDS-03</td>
<td>Software reviews and analyses of Source code, the outputs of the Integration Process and Parameter Data Items (Section 6.3.4, 6.3.5 and 6.6 of the guidance)</td>
<td>Verification of Outputs of Software Coding and Integration Processes</td>
</tr>
<tr>
<td>AV_DO178C_MDS-04</td>
<td>Software reviews and analyses of Software Testing (Section 6.4 of the guidance)</td>
<td>Testing of Outputs of Integration Process</td>
</tr>
<tr>
<td>AV_DO178C_MDS-05</td>
<td>Software reviews and analyses of Test coverage Analysis and Software Verification Process Traceability (Section 6.4.4 and 6.4.5 of the guidance)</td>
<td>Verification of Verification Process Results</td>
</tr>
<tr>
<td>AV_DO178C_MDS-06</td>
<td>Software Quality Process objectives include the consistency of Software plans and standards with documents and software life cycle processes and the conformity review of the software product. (Section 8.1 of the guidance)</td>
<td>Software Quality Assurance Process</td>
</tr>
<tr>
<td>AV_DO178C_MDS-07</td>
<td>Establish communication and understanding between the applicants of the certification authority through Software Life cycle. Approval of the Plan for Software aspects of certification and providing compliance substantiation (Section 9.1 of the guidance)</td>
<td>Certification Liaison Process</td>
</tr>
<tr>
<td>AV_DO178B_THALES_01</td>
<td>pSafeCer approach shall allow to reuse pre-certified software components in new software architectures (component assembly) and avoid/reduce re-certification</td>
<td>Formal Methods</td>
</tr>
<tr>
<td>AV_DO178B_THALES_02</td>
<td>pSafeCer approach (Process, CTF, CAR) shall allow incremental certification for component based software engineering and For any incremental change (functional evolution / non-functional evolution) in the code the Certification framework shall give a quantitative/visible evaluation of the impact on all the contracts and artefacts that will be impacted</td>
<td>Impact Computation</td>
</tr>
<tr>
<td>AV_DO178B_THALES_07</td>
<td>For any incremental evolution - of the software, the components, the specifications, the contracts or the platform - the CAR shall help identifying all the artefacts impacted and strictly the ones impacted by the modification, whichever configuration management tool is used by the company</td>
<td>Impact Computation</td>
</tr>
<tr>
<td>AV_DO178B_THALES_08</td>
<td>pSafeCer approach shall provide means (methodological or tooled) to reduce the re-certification of a certified component that has been modified.</td>
<td>Impact Computation</td>
</tr>
<tr>
<td>Requirement Id</td>
<td>Requirement</td>
<td>V&amp;V method</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>AV_DO178B_THALES_14</td>
<td>pSafeCer CAR shall ensure traceability between software high level requirements and system requirements</td>
<td>Traceability analysis</td>
</tr>
<tr>
<td>AV.DO178B_THALES_16</td>
<td>pSafeCer CAR and CTF shall guarantee that no source code implements an undocumented function</td>
<td>Traceability analysis</td>
</tr>
<tr>
<td>AV.DO178B_THALES_17</td>
<td>The software testing process and tools from pSafeCer shall test HW/SW integration, SW integration between different components</td>
<td>Testing of Outputs of Integration Process</td>
</tr>
<tr>
<td>AV.DO178B_TTT_019</td>
<td>The Process model (PM) shall include the considerations for the use of previously developed software according to DO178B section 12.1</td>
<td></td>
</tr>
<tr>
<td>AV.DO-178B_AKH_01</td>
<td>pSafeCer should produce enable context sensitive help (process help, best practice documents, templates or reference documents)</td>
<td>Traceability analysis &amp; Documentation Management</td>
</tr>
<tr>
<td>AV.DO-178B_AKH_11</td>
<td>pSafeCer should identify a process for isolate and describe the system requirements and collect into SRD Software Requirements Document (DO-178B, 11.9)</td>
<td>Traceability analysis &amp; Documentation Management</td>
</tr>
<tr>
<td>AV.DO-178B_AKH_18</td>
<td>CAR should include traceability maintenance support between related artefacts requirements across system decomposition levels (DO-178B – ISO26262)</td>
<td>Traceability analysis &amp; Documentation Management</td>
</tr>
</tbody>
</table>

In Appendix 9.1, we provide more details about some issues presented in the table, in particular, the traceability requirement. Thus, we also describe the software life cycle development processes and the outputs that have to be verified and traced.

Each verification or validation objective is applicable depending on the software criticality level:

- **Level A**: Anomalous behaviour lead to **catastrophic** failure condition for the aircraft, that is, it results in multiple fatalities with loss of the airplane.
- **Level B**: Anomalous behaviours lead to **hazardous** failure condition for the aircraft, that is, it would reduce the capability of the airplane with a large reduce in safety margins or functional capabilities (excessive workload, fatal injuries).
- **Level C**: Anomalous behaviour lead to **major** failure of the aircraft, that is, would reduce the capability of the airplane with a significant reduce in safety margins or functional capabilities (impairing efficiency, discomfort).
- **Level D**: Anomalous behaviours lead to **minor** failure of the aircraft, that is, there is not significant reduce of airplane safety. It entails slight reduction in safety margins or functional capabilities.
- **Level E**: Anomalous behaviours don’t lead to **any safety effect**.

### 3.3 Railway Domain

We detected the following V&V requirements related to the functional safety standards [EN50126], [EN50128] and [EN50129]:

© The SafeCer Consortium 27 (85)
<table>
<thead>
<tr>
<th>Requirement Id</th>
<th>Requirement</th>
<th>V&amp;V method</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAIL_EN50126_TRAINTIC_000</td>
<td>The integrated certification and development process shall describe all work products relevant to certification and development.</td>
<td>Documentation management</td>
</tr>
<tr>
<td>RAIL_General_UOM_05</td>
<td>The pSafeCer PM shall be general enough to cover the methods and activities required for the various safety-integrity-level / assurance levels. All SIL levels should be supported.</td>
<td>SIL specific V&amp;V methods</td>
</tr>
<tr>
<td>RAIL_EN50128_CCONTROL_26</td>
<td>The CAR and CTF shall support the user with checking the checklist-like rules for the product requirements of EN50128. Example: &quot;The Sw Reqs Spec shall include requirements for the periodic testing of functions...&quot;</td>
<td>Checklist</td>
</tr>
<tr>
<td>RAIL_EN50128_CCONTROL_36</td>
<td>The CAR and CTF shall support V&amp;V activities of EN50128. Note: there are lots of details in the standard of course, but should be no surprises and can be summarized with ensuring consistency/traceability with requirements and design, and &quot;sufficient&quot; verification activities. This seems to be covered by the basic idea of the CAR/CTF as it was presented in Paris and by the prototype shown by AdaCore.</td>
<td>Table A.5 [EN50128] – Verification and Testing</td>
</tr>
<tr>
<td>RAIL_EN50126_AIT_02</td>
<td>Risk analysis shall comply with EN 50126 (phase 3), be tool supported, and interface to the CAR to define (system) safety requirements to be taken up by the certification process (safety case baseline) and the related workflow (apportionment to components and subsystems plus definition of acceptance criteria - component based safety case!!) § 6.9.1.1: &quot;validate the total combination of sub-systems, components and external risk reduction measures comply with the RAMS requirements for the system&quot;</td>
<td>RAMS</td>
</tr>
<tr>
<td>RAIL_EN50128_AIT_01</td>
<td>Software Safety Lifecycle as detailed part of the overall safety lifecycle shall be modelled and treated explicitly (EN 50128)</td>
<td>Table A.1 [EN50128] – Lifecycle Issues and Documentation</td>
</tr>
<tr>
<td>Requirement Id</td>
<td>Requirement</td>
<td>V&amp;V method</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>RAIL_EN50129_AIT_01</td>
<td>The CTF and the related processes (controlled by the Workflow Engine) shall fulfil the requirements of EN 50129 to establish and control the Safety Case and its documents (Fig. 3)</td>
<td>Documentation management</td>
</tr>
</tbody>
</table>

3.3.1 Example RAIL_EN50126_TRAINTIC_000, EN50126

The refined requirement **RAIL_EN50126_TRAINTIC_000** states: “The integrated certification and development process shall describe all work products relevant to certification and development.”

The work products for the 14 phases of the safety lifecycle are described in the [EN 50126] Section 6 (6.1 – 6.14), subsections 2 (Input) and 4 (Deliverables), for example 6.9.2 and 6.9.4 for the phase 9 – System validation (including safety acceptance and commissioning):

6.9.2 Inputs

The input to this phase shall include all relevant information, and where appropriate, data, necessary to meet the requirement, and in particular the system requirements produced in phase 4, the Verification and Validation Plan produced in phase 4, the Commissioning Plan produced in phase 6 and the training material prepared in phase 7.

6.9.4 Deliverables

6.9.4.1 The results of this phase shall be documented, along with any assumptions and justifications made during the phase.

6.9.4.2 A record of all RAMS validation tasks undertaken within the phase, including the commissioning activity, shall be maintained.

6.9.4.3 An Application Specific Safety Case shall be produced for the system within this phase.

6.9.4.4 A record of all Acceptance Tasks undertaken within this phase shall be maintained.

6.9.4.5 The deliverables from this phase form a key input to subsequent lifecycle phases

3.3.2 Example RAIL_EN50128_AIT_01, EN50128

The refined requirement **RAIL_EN50128_AIT_01** states: “Software Safety Lifecycle as detailed part of the overall safety lifecycle shall be modelled and treated explicitly (EN 50128).”

An example for the Software Safety Lifecycle is shown in [EN 50128] in Section 5.3.2.14, and is illustrated in this document in Figure 4.
3.3.3 Example RAIL_EN50129_AIT_01, EN50129

The refined requirement RAIL_EN50129_AIT_01 states: “The CTF and the related processes (controlled by the Workflow Engine) shall fulfil the requirements of EN 50129 to establish and control the Safety Case and its documents (Fig. 3)

The requirements of EN 50129 regarding the structure of the Safety Case are as follows (see [EN 50129], Section 5.1):

*The Safety Case contains the documented safety evidence for the system/sub-system/equipment, and shall be structured as follows:*

**Part 1 Definition of System (or sub-system/equipment)**

This shall precisely define or reference the system/sub-system/equipment to which the Safety Case refers, including version numbers and modification status of all requirements, design and application documentation.

**Part 2 Quality Management Report**

This shall contain the evidence of quality management, as specified in 5.2 of this standard.

**Part 3 Safety Management Report**

This shall contain the evidence of safety management, as specified in 5.3 of this standard.

**Part 4 Technical Safety Report**
This shall contain the evidence of functional and technical Safety, as specified in 5.4 of this standard.

**Part 5 Related Safety Cases**

This shall contain references to the Safety Cases of any sub-systems or equipment on which the main Safety Case depends.

It shall also demonstrate that all the safety-related application conditions specified in each of the related sub-system/equipment Safety Cases are either fulfilled in the main Safety Case, or carried forward into the safety-related application conditions of the main Safety Case.

**Part 6 Conclusion**

This shall summarize the evidence presented in the previous parts of the Safety Case, and argue that the relevant system/sub-system/equipment is adequately safe, subject to compliance with the specified application conditions.

### 3.4 Space

Since the space domain is not considered within pSafeCer, the corresponding requirements are not determined yet. These are analysed and defined in nSafeCer. Nevertheless, the software development process for V&V processes is worth considering at this stage.

Civil avionics is analysed in section 3.2. ECSS-E-ST-40C and ECSS-Q-ST-80C standards specify the software development process in the Space Domain, and namely the verification and validation processes:

- The software verification process is intended to confirm that adequate specifications and inputs exist for every activity, and that the outputs of the activities are correct and consistent with the specification and inputs.
- The software validation process is intended to confirm that the technical specification and the requirements baseline functions and performances are correctly and completely implemented in the final product.

Both processes are executed with different degrees of independence. This degree of independence can range from the same person, or a different person in the same organization, to a person in a different organization, with varying degrees of separation. When the software verification and validation processes are executed by an organization independent of the supplier, it is called Independent Verification and Validation (ISVV). ISVV is intended to improve quality and reduce costs of a software product as well as reduce development risks by having an organization independent of the software developer's to perform the verification and validation of the specifications and code of a software product.

The V&V processes are tailored according to the software criticality, specifying the V&V strategy to be followed. Verification and validation processes for critical systems involve performing additional activities and analyses than for non-critical systems. Firstly the software is classified according to its criticality. The V&V techniques to be applied will depend on the criticality assigned: those applications with higher criticality need a stricter V&V analysis. Then, the selection of one technique or combination of different techniques will vary.

Table 6 lists software criticality categories based on the severity of the consequences of system failures:
Table 6: Software criticality categories and severity categories

<table>
<thead>
<tr>
<th>SW Category</th>
<th>Severity</th>
<th>Level</th>
<th>Dependability (ECSS-Q-ST-30)</th>
<th>Safety (ECSS-Q-ST-40C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Catastrophic</td>
<td>1</td>
<td>Failure propagation</td>
<td>Loss of life, life-threatening or permanently disabling injury or occupational illness.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Loss of an interfacing manned flight system.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Severe detrimental environmental effects.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Loss of launch site facilities.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Loss of system.</td>
</tr>
<tr>
<td>B</td>
<td>Critical</td>
<td>2</td>
<td>Loss of mission</td>
<td>Temporarily disabling but not life-threatening injury, or temporary occupational illness.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Major detrimental environmental effects.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Major damage to public or private properties.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Major damage to interfacing flight systems.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Major damage to ground facilities.</td>
</tr>
<tr>
<td>C</td>
<td>Major</td>
<td>3</td>
<td>Major mission degradation</td>
<td></td>
</tr>
<tr>
<td>D/E</td>
<td>Minor or negligible</td>
<td>4</td>
<td>Minor mission degradation or any other effect.</td>
<td></td>
</tr>
</tbody>
</table>

Therefore, the qualification process differs depending on the software criticality level. For instance:

- Independent Software Verification Validation is only required for A and B software.
- 100% source code modified condition and decision coverage is required for category A (CATASTROPHIC) but not for category B (CRITICAL).

3.5 Cross Domain/Other Domains

In order to be able to extend applicability of the SafeCer approach of incremental/compositional certification of software-intensive systems with reusable components, the “cross-domain” and “other domains” work package has been added to those addressing the focused domains automotive (including construction equipment), avionics and rail. This addresses the reuse of arguments (methodology/process) as well as of (pre-qualified) components.

“Cross Domain” covers two aspects:

- Methodology/Framework aspect: Applicability of the SafeCer approach across domains, following generic standards (and some domain-specific adaptations/add on’s if required), or a to be defined “Superset Standard” (for a set of application domains, which can only be conceptual at the moment),
- Component-multi-domain reuse: Re-use of (preferably only once) qualified/certified components, in the sense of IEC 61508 as (software-intensive) components consisting of HW and SW, or SW components (“items”) under specified conditions, across several domains to enlarge the component market.
“Other Domains” covers the aspect of applicability of the SafeCer approach (methodology, tool framework) to other domains than those addressed in pSafeCer (one example is “Healthcare/Medical Devices” in nSafeCer). This is supported by guidelines on how to achieve this goal effectively and efficiently, based on the SafeCer methodology and tool framework, and SafeCer instantiation policy according to the relevant (domain or generic) functional safety standards. This is also part of the Education & Training Use Case in nSafeCer, one of the expected outcomes of SafeCer to facilitate its broader application.

3.5.1 Cross Domain/Other Domains: V&V requirements

Depending on the domains to be covered, there are four aspects to be considered with regard to the SafeCer requirements for the V&V framework:

- Processes required by the generic and the domain specific standards have to (if possible) be mapped upon each other to identify commonalities in the process-related requirements (the generic and domain-specific process models developed already for the relevant standards should support this) and a minimal union set (minimal superset) has to be established to avoid duplicate V&V and certification/qualification effort (see Figure 5).
- Techniques required by the generic and the domain specific standards have to (if possible) be mapped upon each other to identify commonalities in the required techniques and a minimal superset has to be defined.
- For other domains, a guideline has to be developed how to select either an existing set of processes and techniques to fulfil the requirements of the other domain (easier in case an existing e.g. generic standard has to be applied because of lack of a domain standard, otherwise the “other” domain standard has to be mapped)
- Additionally, the properties of the provided tools have to be mapped versus the identified set of techniques and process requirements.

The ability to derive the V&V requirements for cross-domain or other domains is essential for a broad applicability of the SafeCer approach towards incremental component-based certification/qualification.

![Figure 5: Simplified picture of domain set/superset of requirements for cross/other domain](image)

The pragmatic way would be to compare the cross- or other domain requirements with the existing standards’ sets and look for the minimal set of enhancements/extensions necessary, i.e. to choose the “best fit” standard and extend it to the “superset”.

3.5.2 Cross Domain/Other Domain: Identified V&V requirements

V&V Requirements for Cross/Other Domain stem either from the standards to be applied or from the artefact (component, subsystem, system, software) to be qualified/certified in a certain context or out of context. It should be noted that most derived functional safety standards like ISO 26262,
IEC 61511, IEC 61131-6, and IEC 62061 apply only up to SIL 3 (Safety Integrity Level 3) of the generic functional safety standard IEC 61508 but has preference in the respective domains before the generic standard up to this SIL. For applications requiring SIL 4 (e.g. this would at the moment apply to autonomous vehicles) IEC 61508 shall be applied (for SIL definition see Table 6).

**Table 6: IEC 61508 Safety integrity levels – target failure measures for a safety function operating in high demand mode of operation or continuous mode of operation**

<table>
<thead>
<tr>
<th>Safety integrity level (SIL)</th>
<th>Average frequency of a dangerous failure of the safety function [h-1] (PFH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>$\geq 10^{-4}$ to $&lt; 10^{-3}$</td>
</tr>
<tr>
<td>3</td>
<td>$\geq 10^{-5}$ to $&lt; 10^{-4}$</td>
</tr>
<tr>
<td>2</td>
<td>$\geq 10^{-6}$ to $&lt; 10^{-5}$</td>
</tr>
<tr>
<td>1</td>
<td>$\geq 10^{-7}$ to $&lt; 10^{-6}$</td>
</tr>
</tbody>
</table>

The (main) requirements identified for “cross domain/other domain” impacting V&V are described in Table 7.

**Table 7: V&V Requirements from cross domain/other domain**

<table>
<thead>
<tr>
<th>Requirement Id</th>
<th>Requirement</th>
<th>V&amp;V method</th>
</tr>
</thead>
<tbody>
<tr>
<td>CROSS_AKH_03</td>
<td>CTF should include tools for planning, specification, design and verification</td>
<td>Methods for planning, specification, design and verification</td>
</tr>
<tr>
<td>CROSS_FBK_000</td>
<td>SafeCer shall support formal methods for properties, architecture, and behavioural specification, validation and verification.</td>
<td>Formal methods</td>
</tr>
<tr>
<td>CROSS_FBK_001</td>
<td>SafeCer shall support model-based safety analysis.</td>
<td>Safety analysis</td>
</tr>
<tr>
<td>CROSS_AIT_011</td>
<td>The CTF and the Workflow Engine shall be adaptable to show the conformance to meet the product-specific requirements as addressed e.g. in IEC 61131, IEC 60601 or IEC 6151 and other related standards</td>
<td>V&amp;V methods described by the respective standards</td>
</tr>
<tr>
<td>CROSS_AIT_012</td>
<td>The CTF (and the CAR and Workflow Engine) shall cover the initial phases (concept, risk/hazard analysis, safety function allocation) in a consistent manner and allow to derive the required information for the component model, component property definition (contract conformance or adaptation?) and safety case building, with the related tool support (automatic and with manual (human) interaction where required)</td>
<td>Concept, risk/hazard analysis, safety function allocation</td>
</tr>
<tr>
<td>Requirement Id</td>
<td>Requirement</td>
<td>V&amp;V method</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>CROSS_CCONTROL_038</td>
<td>In the CAR (and supported by CTF), there shall be traceability for all steps in the engineering chain.</td>
<td>Traceability analysis</td>
</tr>
<tr>
<td>CROSS_CCONTROL_047</td>
<td>The CAR shall produce a report telling which activities are allocated on which tool with the respect to the safety standard and level</td>
<td>Documentation management</td>
</tr>
<tr>
<td>AT_DO178B_THALES_15</td>
<td>SafeCer CAR and CTF shall ensure that the source code is complete with respect to the low level requirements and that every low level requirements has been developed into the source code</td>
<td>Software validation</td>
</tr>
<tr>
<td>AT_DO178B_THALES_18</td>
<td>SafeCer CTF shall cover both requirements and structural tests as per section 6 from the DO178B</td>
<td>V&amp;V methods of section 6 from the DO178B</td>
</tr>
<tr>
<td>CROSS_DO-178B_AKH_17</td>
<td>CTF should include a tool for Test execution on host, simulator, and embedded target systems (DO-178B – ISO26262)</td>
<td>HW and SW testing</td>
</tr>
</tbody>
</table>
4 Existing V&V methods for system certification

This section describes existing V&V methods for system certification in different domains: automotive, avionics, railway, space and cross domain/other domains.

Moreover, it tries to identify shortcomings of existing V&V methods for system certification.

4.1 Automotive Domain

According to the waterfall model or the V model for system development, verification and validation is performed after the system design and implementation. That is, analysis activities performed during design to check the soundness of different design choices and to identify potentials for improvement of the design are normally not considered V&V. However, in the context of this deliverable such activities are included in the scope of V&V. For this wider scope, techniques that are used in the automotive domain include:

- **FMEA** (Failure Modes and Effects Analysis) to identify, analyse and mitigate possible faults. Potential faults are systematically identified and investigated in terms of their causes and probability of occurrence, their effects and the associated severity, as well as the possibility of detecting the errors resulting from the faults. The purpose of the FMEA is at least two-fold: the dependability of an existing or proposed design is assessed and potential improvement areas are identified. FMEA is a well-known method and is therefore not described in detail here.

- **FTA** (Fault Tree Analysis) to identify and mitigate possible faults, based on a given set of system-level failures potentially resulting from those faults. The purpose of performing fault tree analysis is quite similar to that of an FMEA. However, instead of an inductive bottom-up analysis from faults to their effects, a deductive top-down approach is taken in which the system-level failures are traced to their potential causes in terms of faults. FMEA and FTA both have their strong and weak points so ideally they should be used in combination to ensure that the analysis covers as much as possible. FTA is a well-known method and is therefore not described in detail here.

- **Design reviews** to check that the design is reasonable and conforms to requirements. The review should follow a formalized procedure and the findings should be documented in a formal review protocol. The details on how the review is to be performed, whom to involve as reviewers, etc. depend on the scope of the review. For example, a review of a large system is a much bigger undertaking than the review of a small component.

- **Simulation** to verify behaviour of algorithms before implementation. Much design work is today performed using tools like Matlab/Simulink and various UML tools, enabling simulation to be carried out. In the same way as for testing, simulation is used to verify that a design fulfils specified requirements and to validate that the behaviour conforms to unstated expectations. The detailed procedure for performing simulations depends heavily on the nature of the target system to be simulated and on how its requirements are expressed. Examples range from simple ‘one-shot’ simulations to verify that the simulation responds as specified in a given situation to long term simulations of perhaps millions of test cases.

- **Timing analysis** to verify that end-to-end timing and other temporal properties are satisfactory. This includes establishing the worst-case execution-time (WCET) of tasks, determining worst-case forwarding-delay (WCFD) of messages and performing schedulability analysis of the tasks in the computation nodes. These analyses can be done analytically or though simulations and measurements. Analytical approaches use models of the system and then apply different techniques such as abstract/symbolic interpretation, response-time analysis or search algorithms to find the worst case. A downside of these approaches is that they often are time-consuming and over-pessimistic. Therefore, if the worst-case calculation does not have to be safe, alternative solutions could be useful. E.g.
simulating different system scenarios and recording the worst-case seen. The same approach can be used in test rigs to measure the worst-case.

- **Bus load analysis** to verify that communication busses are not overloaded. This is done as part of timing analysis but can also be done as a separate activity where timing is not explicitly addressed. It is known that the probability of meeting the timing requirements decrease with the utilization of the bus. It is also not flexible to have a too high bus load since there will be no room for additional messages. The analysis is analytically done taking into account the message sizes and transmission patterns.

- **Back-to-back testing** to verify that the implementation corresponds to the simulated behaviour. Any deviation is investigated in terms of why it has occurred and what its consequences will be. When necessary, actions to rectify the situation are identified.

- **Requirements-based testing** (applied at software, hardware, electronic control unit, system or vehicle level, as described in the following bullet item) to verify that the requirements at the corresponding abstraction level are fulfilled. Such testing can be performed in many different ways, depending on how the requirement under consideration is expressed. However, the typical case is when the requirement states what the system shall do in a particular situation. This situation (in terms of system state, environmental state and system inputs) is then created and it is checked that the system behaves as stated in the requirement.

- **Testing** at different levels to verify compliance with stated requirements and to validate compliance with unstated expectations:
  - **Software testing** to verify that the implemented software behaves as specified and expected when executing. Such testing can cover a software unit or the complete software for a control unit, or anything in between. The testing is performed in a general purpose computer. If the tests are performed in the target hardware they are instead referred to as **control unit testing** (see below).
  - **Hardware testing** to verify that hardware behaves as specified and expected.
  - **Control unit testing** to verify that each electronic control unit, including its embedded software, behaves as specified and expected. Such testing is typically performed by the control unit supplier.
  - **System testing** in test rigs to verify that a system behaves as specified and expected. The system can be a single control unit with its associated sensors and actuators, or several such units interconnected with each other or even the whole electrical system of a vehicle. Such testing is typically performed by the vehicle manufacturer. Compared to testing in a vehicle, rig testing offers the following benefits:
    - Easy access to ECU I/Os
    - Facilitates fault injection testing
    - Enables automated testing, thereby speeding up the testing
    - Enables system tests to be carried out before the vehicle has been produced
  - **In-vehicle testing** to verify that the vehicle behaves as specified and expected. Such testing is typically performed by the vehicle manufacturer. Some tests can be performed in a garage while others can only be made when driving on a road.

Much of the V&V activities are based on testing and therefore appear rather late in the development cycle. Earlier analysis and testing based on models would potentially eliminate many defects. The use of models would also increase the possibility to use formal verification techniques.

### 4.2 Avionics Domain

The development of verification and validation tools, methods and techniques that advance safety assurance and certification of complex avionics is an important issue since it represents more than 50% of software development costs.
According to the DO-178C guidance for software development of avionics, the purpose of the software verification process is to detect and report errors that may have been introduced during the software development. It includes the verification of source code, low-level requirements and software architecture. The degree of test coverage and the robustness of testing are also important issues for verification and validation in the certification processes. Techniques that are used and explored in the avionics domain include the following:

- **Testing** is the process of exercising a system or system component to verify that it satisfies specified requirements and to detect errors. It includes the elaboration of test cases, a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. It also includes test procedures that are detailed instructions for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases.

- **Simulation** uses simulators or emulators which are devices or computer programs or systems used during software verification. They accept the same inputs and produce the same output as a given system, using object code that is derived from the original object code.

- **Traceability** is an association between items, such as between process outputs, between an output and its originating process, or between requirements and its implementation. Existing tools include Reqtify\(^1\) or Doors\(^2\). In the DO-178C guidance it is required to verify the traceability. Software development process traceability activities include trace data showing bidirectional association between, for example:
  - System requirements allocated to software and high-level requirements.
  - High-level requirements and low-level requirements.
  - Low-level requirements and source code.

Moreover, system, high-level, and low-level requirements should be traced to corresponding test cases.

- **Formal methods** are a key issue since their use presents one of the main novelties in the DO-178C compared to the DO-178B. They describe notations and analytical methods used to construct, develop, and reason about mathematical modes of system behaviour. A formal method is a formal analysis carried out on a formal model. A model is formal if it has non-ambiguous, mathematically defined syntax and semantics.
  1. There exist **formal modelling languages**, such as the SCADE language, that are usually dedicated to a specific type of application. For example, synchronous languages for reactive systems and formal notations for the expression of properties, such as temporal logics.
  2. There are **analysis techniques** such as Model checking, that explores all possible behaviours of a formal model to determine the satisfaction of a given property (timing, deadlock freedom and so on). Analysis techniques also include abstract interpretation and deductive methods.
  3. Formal methods are used in **industrial systems** such as aiT for WCET computation, Stackanalyzer\(^3\) for stack consumption computation and Caveat for verification of compliance of low level requirements in replacement of unit testing. These methods are used for achieving some DO-178B verification objectives.

- **Coverage analysis** is the process of determining the degree to which a proposed verification process activity satisfies its objectives.

\(^1\) [http://www.3ds.com/products/catia/portfolio/geensoft/reqtify/](http://www.3ds.com/products/catia/portfolio/geensoft/reqtify/)


\(^3\) [http://www.absint.com/stackanalyzer/index.htm](http://www.absint.com/stackanalyzer/index.htm)
Structural coverage analysis is an evaluation of code structure, including interfaces, exercised during requirements-based testing.

4.3 Railway Domain

In sections 6.2 and 6.3 of the standard [EN50128] the objectives of verification and validation are described:

6.2.1.1 The objective of software verification is to examine and arrive at a judgment based on evidence that output items (process, documentation, software or application) of a specific development phase fulfill the requirements and plans with respect to completeness, correctness and consistency. These activities are managed by the verifier.

6.3.1.1 The objective of software validation is to demonstrate that the processes and their outputs are such that the software is of the defined software safety Integrity level, fulfills the software requirements and is fit for its Intended application. This activity is performed by the validator.

6.3.1.2 The main validation activities are to demonstrate by analysis and/or testing that all the software requirements are specified, implemented, tested and fulfilled as required by the applicable SIL, and to evaluate the safety criticality of all anomalies and non-conformities based on the results of reviews, analyses and tests.

In this section two examples from the [EN50128] are described:

- Table A.5 – Verification and Testing (6.2)
- Table A.8 – Software Analysis Techniques (6.3)

4.3.1 EN50128, Table A.5 Verification and Testing

The Table A.5 in [EN50128] – Verification and Testing – contains the recommended V&V methods for verification and testing (see Table 8). To better understand the applied V&V methods the information in the column “Ref” is used which contains references to detailed tables and explanatory sections. A selection of them (those marked in grey) is detailed below. The table further contains the recommendation for each SIL. The meaning of the symbols (the first occurrence is marked in grey) is also described in the following.

Table 8: EN50128, Table A.5 Verification and Testing

<table>
<thead>
<tr>
<th>TECHNIQUE/MEASURE</th>
<th>Ref</th>
<th>SIL 0</th>
<th>SIL 1</th>
<th>SIL 2</th>
<th>SIL 3</th>
<th>SIL 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Formal Proof</td>
<td>D.29</td>
<td></td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>2. Static Analysis</td>
<td>Table A.19</td>
<td></td>
<td></td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>3. Dynamic Analysis and Testing</td>
<td>Table A.13</td>
<td></td>
<td></td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>4. Metrics</td>
<td>D.37</td>
<td></td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>5. Traceability</td>
<td>D.58</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>6. Software Error Effect Analysis</td>
<td>D.25</td>
<td></td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>7. Test Coverage for code</td>
<td>Table A.21</td>
<td></td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>8. Functional/ Black-box Testing</td>
<td>Table A.14</td>
<td></td>
<td>HR</td>
<td>HR</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>9. Performance Testing</td>
<td>Table A.18</td>
<td></td>
<td></td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>10. Interface Testing</td>
<td>D.34</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
</tbody>
</table>
Requirements:

1) For software Safety Integrity Levels 3 and 4, the approved combination of techniques is 3, 5, 7, 8 and one from 1, 2 or 6.
2) For software Safety Integrity Levels 1 and 2, the approved combination of techniques is 5 together with one from 2, 3 or 8.

NOTE 1 Techniques/measures 1, 2, 4, 5, 6 and 7 are for verification activities.
NOTE 2 Techniques/measures 3, 8, 9 and 10 are for testing activities.

The symbols for the recommendations have the following meaning:

- **M** this symbol means that the use of a technique is Mandatory
- **HR** this symbol means that the technique or measure is Highly Recommended for this safety integrity level. If this technique or measure is not used then the rationale for using alternative techniques shall be detailed in the Software Quality Assurance Plan or in another document referenced by the Software Quality Assurance Plan,
- **R** this symbol means that the technique or measure is Recommended for this safety integrity level. This is a lower level of recommendation than an 'HR' and such techniques can be combined to form part of a package,
- **NR** this symbol (not used in this table) means that the technique or measure is positively Not Recommended for this safety integrity level. If this technique or measure is used then the rationale behind using it shall be detailed in the Software Quality Assurance Plan or in another document referenced by the Software Quality Assurance Plan.
- This symbol means that the technique or measure has no recommendation for or against being used

The explanatory sections and detailed tables are the following:

**D.25 Software Error Effect Analysis**

The analysis is done in three phases.

- **Vital software components identification**
  Determination of the depth of the analysis (at the level of a single instruction line, a group of instructions, a component, etc.) needed for each software component, from its specification.

- **Software error analysis**
  - The result of this phase is a table listing the following Information:
    - component name;
    - error considered;
    - consequences of the error at the module level;
    - consequences at the system level;
    - violated safety criterion;
    - error criticality;
    - proposed error detection means;
    - violated criterion if the detection means is implemented;
    - Residual criticality if the detection means is implemented.

- **Synthesis**
  The synthesis identifies the remaining unsafe scenarios and the validation effort needed given the criticality of each module.
D.29 Formal Proof

A number of assertions are stated at various locations in the program, and they are used as pre and post conditions to various paths in the program. The proof consists of showing that the program transfers the preconditions into the post-conditions according to a set of logical rules, and that the program terminates.

D.34 Interface Testing

Several levels of detail or completeness of testing are feasible. The most important levels are testing

- all Interface variables at their extreme positions,
- all interface variable individually at their extreme values with other interface variables at normal values,
- all values of the domain of each Interface variable with other Interface variables at normal values,
- all values of all variables in combination (this may only be feasible for small interfaces),
- The specified test conditions relevant to each call of each subroutine.

These tests are particularly important if their interfaces do not contain assertions that detect incorrect parameter values. They are also important after new configurations of pre-existing subprograms have been generated.

D.37 Metrics

These models evaluate some structural properties of the software and relate this to a desired attribute such as complexity. Software tools are required to evaluate most of the measures. Some of the metrics which can be applied are given below:

- Graph Theoretic Complexity: this measure can be applied early in the lifecycle to assess trade-offs, and is based on the complexity of the program control graph, represented by its cyclomatic number;
- number of ways to activate a certain component (accessibility): the more a component can be accessed, the more likely it is to be debugged;
- Halstead complexity measures: this measure computes the program length by counting the number of operators and operands. It provides a measure of complexity and estimates development resources;
- Number of entries and exits per component: minimizing the number of entry/exit Points is a key feature of structured design and programming techniques.

D.58 Traceability

Traceability to requirements shall be an important consideration in the validation of a system and means shall be provided to allow this to be demonstrated throughout all phases of the lifecycle.

Traceability shall be considered applicable to both functional and non-functional requirements and shall particularly address:

a) traceability of requirements to the design or other objects which fulfil them,

b) traceability of design objects to the implementation objects which instantiate them,

c) traceability of requirements and design objects to the operational and maintenance objects required to be applied in the safe and proper use of the system,

d) traceability of requirements, design, implementation, operation and maintenance objects, to the verification and test plans and specifications which will determine their acceptability,
e) Traceability of verification and test plans and specifications to the test or other reports which record the results of their application.

Where requirements, design or other objects are instantiated as a number of separate documents, traceability shall be maintained within the document structures and in a hierarchical manner.

The output of the Traceability process shall be the subject of formal Configuration Management.

4.3.2 EN50128, Table A.19 Static Analysis

The Table A.19 in [EN50128] – Static analysis – contains the recommended V&V methods for Static analysis (see Table Table 9). To better understand the applied V&V methods the information in the column “Ref” is used which contains references to detailed tables and explanatory sections. A selection of them (those marked in grey) is detailed below. The table further contains the recommendation for each SIL. The meaning of the symbols (the first occurrence is marked in grey) is also described in the following.

Table 9: EN50128, Table A.19 Static Analysis.

<table>
<thead>
<tr>
<th>TECHNIQUE/MEASURE</th>
<th>Ref</th>
<th>SIL 0</th>
<th>SIL 1</th>
<th>SIL 2</th>
<th>SIL 3</th>
<th>SIL 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Boundary Value Analysis</td>
<td>D.4</td>
<td>-</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>2. Checklists</td>
<td>D.7</td>
<td>-</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>3. Control Flow Analysis</td>
<td>D.8</td>
<td>-</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>4. Data Flow Analysis</td>
<td>D.10</td>
<td>-</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>5. Error Guessing</td>
<td>D.20</td>
<td>-</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>6. Walkthroughs/Design Reviews</td>
<td>D.56</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
</tbody>
</table>

D.4 Boundary Value Analysis

The Input domain of the program is divided into a number of Input classes. The tests should cover the boundaries and extremes of the classes. The tests check that the boundaries in the input domain of the specification coincide with those in the program. The use of the value zero, in a direct as well as in an indirect translation, is often error-prone and demands special attention:

- zero divisor;
- non-printing control characters;
- empty stack or list element;
- null matrix;
- Zero table entry.

Normally the boundaries for input have a direct correspondence to the boundaries for the output range. Test cases should be written to force the output to its limited values. Consider also, if it is possible to specify a test case which causes output to exceed the specification boundary values.

If output is a sequence of data, for example a printed table, special attention should be paid to the first and the last elements and to lists containing none, 1 and 2 elements.

D.7 Checklist

A set of questions to be completed by the person performing is the checklist. Many of the questions are of a general nature and the Assessor shall interpret them as seems most appropriate to the particular system being assessed.
To accommodate wide variations in software and systems being validated, most checklists contain questions which are applicable to many types of system. As a result there may well be questions in the checklist being used which are not relevant to the system being dealt with and which should be ignored. Equally there may be a need, for a particular system, to supplement the standard checklist with questions specifically directed at the system being dealt with.

In any case it should be clear that the use of checklists depends critically on the expertise and judgement of the engineer selecting and applying the checklist. As a result the decisions taken by the engineer, with regard to the checklist(s) selected, and any additional or superfluous questions, should be fully documented and justified. The objective is to ensure that the application of the checklists can be reviewed and that the same results will be achieved unless different criteria are used.

The object in completing a checklist is to be as concise as possible. When extensive justification is necessary this should be done by reference to additional documentation. Pass, Fail and Inconclusive, or some similar restricted set of tokens should be used to record the results for each question. This conciseness greatly simplifies the process of reaching an overall conclusion as to the results of the checklist assessment.

D.8 Control Flow Analysis

Control Flow Analysis identifies suspect areas of code which do not follow good programming practice.

The program is analysed to form a directed graph which can be analysed for

- inaccessible code, for instance, unconditional Jumps which leaves blocks of code unreachable,
- Knotted code, which is well structured code whose control graph is reducible by successive graph reductions to a single node. Poorly structured code can only be reduced to a knot composed of several nodes.

D.10 Data Flow Analysis

Data Flow Analysis combines the information obtained from the control flow analysis with information about which variables are read or written in different portions of code. The analysis can check for

- Variables that are read before they are written. This is very likely to be an error, and is certainly bad programming practice,
- Variables that are written more than once without being read. This could indicate omitted code,
- Variables that are written but never read. This could indicate redundant code

There is an extension of data flow analysis known as information flow analysis, where the actual data flows (both within and between procedures) are compared with the design intent. This is normally implemented with a computerised tool where the intended data flows are defined using a structured comment that can be read by the tool.

D.20 Error Guessing

Testing experience and intuition combined with knowledge and curiosity about the system under test may add some uncategorised test cases to the designed test case set. Special values or combinations of values may be error-prone. Some interesting test cases may be derived from inspection checklists. It may also be considered whether the system IS robust enough. Can the
buttons be pushed on the front-panel too fast or too often? What happens if two buttons are pushed simultaneously?

**D.56 Walkthroughs / Design Reviews**

IEC/TC 56, have published a Guide on Formal Design Reviews, which includes a general description of formal design reviews, their objectives, details of the various design review types, the composition of a design review team and their associated duties and responsibilities. The IEC document also provides general guidelines for planning and conducting formal design reviews, as well as specific details concerning the role of independent specialists within a design review team.

The IEC recommend that a “formal design review shall be conducted for all new products/processes, new applications, and revisions to existing products and manufacturing processes which affect the function, performance, safety, reliability, ability to inspect maintainability, availability, ability to cost, and other characteristics affecting the end product/process, users or bystanders”.

A code walk through consists of a walk through team selecting a small set of paper test cases, representative sets of inputs and corresponding expected outputs for the program. The test data is then manually traced through the logic of the program.

**4.3.3 EN50128, Table A.8 Software Analysis Techniques**

The Table A.8 in [EN50128] – Software Analysis Techniques – contains the recommended software analysis techniques (see Table 10). The explanatory sections (marked in grey) which have not been described in section 4.3.1 are described below.

<table>
<thead>
<tr>
<th>TECHNIQUE/MEASURE</th>
<th>Ref</th>
<th>SIL 0</th>
<th>SIL 1</th>
<th>SIL 2</th>
<th>SIL 3</th>
<th>SIL 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Static Software Analysis</td>
<td>Table A.19</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td></td>
</tr>
<tr>
<td>2. Dynamic Software Analysis</td>
<td>Table A.13T A.14</td>
<td>-</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>3. Cause Consequence Diagrams</td>
<td>D.6</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>4. Event Tree Analysis</td>
<td>D.22</td>
<td>-</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>5. Software Error Effect Analysis</td>
<td>D.25</td>
<td>-</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
</tbody>
</table>

Requirement:
1) One or more of these techniques shall be selected to satisfy the Software Safety Integrity Level being used.

**D.6 Cause Consequence Diagrams**

It can be regarded as a combination of fault-tree and event-tree analysis. Starting from a critical event, a cause-consequence graph is traced backwards and forwards. In the backwards direction it is equivalent to a fault tree with the critical event as the given top event. In the forward direction the possible consequences arising from an event are identified. The graph can contain vertex symbols which describe the conditions for propagation along different branches from the vertex. Time delays can also be included. These conditions can also be described with fault trees. The lines of
propagation can be combined with logical symbols, to make the diagram more compact. A set of standard symbols are defined for use in cause consequence diagrams. The diagrams can be used to compute the probability of occurrence of certain critical consequences.

**D.13 Decision Tables**

These related methods use two dimensional tables to concisely describe logical relationships between Boolean program variables.

The conciseness and tabular nature of both methods make them appropriate as a means of analysing complex logical combinations expressed in code.

Both methods are potentially executable if used as specifications.

**D.22 Event Tree Analysis**

On the top of the diagram is written the sequence conditions that are relevant in the development following the initiating event which is the target of the analysis. Starting under the initiating event, one draws a line to the first condition in the sequence. There the diagram branches off into a 'yes' and a 'no' branch, describing how the future developments depend on the condition for each of these branches, one continues to the next condition in a similar way. Not all conditions are, however, relevant for all branches. One continues to the end of the sequence, and each branch of the tree constructed in this way represents a possible consequence. The event tree can be used to compute the probability of the various consequences based on the probability and number of conditions in the sequence.

**4.4 Space Domain**

**4.4.1 V&V for Critical Software**

Verification and Validation activities must be exhaustive when critical software is being developed. They must be performed along the whole life cycle, being applied at each stage of the software process.

The verification of the implementation of dependability and safety is limited due to the fact of the huge range of different scenarios, input combinations for the functions to test and characteristics to verify. Thus the dependability and safety of software should be verified and assessed using a process independent from that used for the verification of functionality.

V&V are specially related to fault removal and fault tolerance mechanisms, which must be applied earlier on the software life cycle.

**4.4.2 V&V strategies for Critical Software**

Verification methods are mainly divided into four groups:

- Analysis. Verification through a theoretical or empirical evaluation.
- Review-of-design. Verification by validation of records, by evidence of validated design documents, when approved design reports/technical descriptions/engineering drawings unambiguously show that the requirement is met.
- Inspection. Verification through visual determination of physical characteristics.
- Test. Empirical verification method that proves that under certain conditions, the specific inputs cause the expected results.
Splitting up the previous verification methods, different verification techniques are identified. Here are some of them:

**Table 11: Verification techniques for the space domain**

<table>
<thead>
<tr>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Code Analysis</td>
</tr>
<tr>
<td>Inspection</td>
</tr>
<tr>
<td>Walkthroughs</td>
</tr>
<tr>
<td>Sizing and Schedulability analysis</td>
</tr>
<tr>
<td>Traceability analysis</td>
</tr>
<tr>
<td>Formal proof of correctness</td>
</tr>
<tr>
<td>Reviews and buddy checks</td>
</tr>
<tr>
<td>White and black box testing</td>
</tr>
<tr>
<td>Functional and performance testing</td>
</tr>
<tr>
<td>Code dynamic analysis</td>
</tr>
</tbody>
</table>

A full conjunction of static and dynamic verification provides a full V&V coverage. They are complementary techniques.

Static verification is linked to the analysis of the static software representation to discover problems and prevent possible faults, whereas dynamic verification is concerned with exercising and observing the software behaviour (testing). Testing is used for those requirements that need to be verified measuring the product performance and the behaviour under simulated environments.

Validation can employ different techniques. Table 12 collects some of them:

**Table 12: Validation techniques for the space domain**

<table>
<thead>
<tr>
<th>Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Specifications and Test results:</td>
</tr>
<tr>
<td>Integration validation or testing</td>
</tr>
<tr>
<td>Functional validation or testing</td>
</tr>
<tr>
<td>User acceptance validation or testing</td>
</tr>
<tr>
<td>Code validation or testing</td>
</tr>
<tr>
<td>Formal methods</td>
</tr>
</tbody>
</table>

The only validation technique for non-functional requirements is software testing. The software has to be executed to check its behaviour: if the software meets the requirements specified. For instance, inspections cannot check non-functional characteristics (e.g., performance) and inspections only check the conformance with a specification.

The techniques collected in Table 11 and Table 12 shall be applied along the whole software life cycle. However, each technique focuses on specific parts of the software life-cycle (different levels). So, depending on the level analysed, one of them have to be selected.

Moreover the aforementioned techniques can be combined to provide a better and more complete V&V result. Next table specifies the phases in which each technique is usually employed:

**Table 13: Verification techniques for the space domain: Functional, Design and Code level**
The verification techniques can be used at different stages of the software life-cycle and they purpose differ. Here are summarized the main objectives to be achieved at each level:

- **Functional Level:** The objective of the functional requirements verification is to determine if the requirements are adequate:
  - Check that requirements are consistent.
  - Check that requirements are satisfied.
  - Check that requirements are tested.
  - Functional validation.

- **Design Level:** The design must be consistent with the requirements, and the design itself must be adequate. As an example, the following must be verified:
  - Consistency with the requirements of the software component.
  - Consistency between software components.
  - Traceability from requirements to the software items.
  - Correct design with respect to the requirements and interfaces. The design can be derived from requirements.

- **Code Level:** Code verification activity must guarantee that the implementation is adequate and consistent with the design. These are some activities done during the verification process:
  - Code can be derived from design or software requirements.
  - Code inspection.
  - Detect and eliminate errors.
  - Verify coding practices, model to code translation, static and dynamic analysis, etc.

### 4.5 Cross Domain/Other Domains

In case of “cross-domain” and “other domains” methods for certification and validation a generic approach is taken here, since the basic methods of the different domains apply in many cases. From generic standards like IEC 61508-3 (software-intensive systems, SW items) and new proposals for “Dependability Case” standards and specifications (IEC TC 56), the methods how to describe appropriateness of processes, measures and techniques for different SILs are taken as examples for a general approach to assessment and description of V&V methods for component qualification and system certification.

“Cross Domain” covers two aspects, as already described in section 3.5:

- Applicability of the SafeCer approach across different domains,
- Re-use of (preferably only once) qualified/certified components (HW+SW) or SW items (in IEC 61508 terminology) under specified conditions across several domains,

“Other Domain” covers the aspect of applicability of the SafeCer approach (methodology, tool framework) to other domains than addressed in pSafeCer (one example is “Healthcare/Medical Devices” in nSafeCer), supported by guidelines and the SafeCer instantiation policy according to the relevant (domain or generic) functional safety standards. This is also part of the Education & Training Use Case in nSafeCer.

An economic “side effect” is that this approach (pSafeCer Certification Tool Framework and its instantiations comprising integration of adapted tools) opens a broader market for tool vendors also, not only for component suppliers.

The “Cross Domain” as well as the “Other Domains” approach requires that methods (techniques), processes, and tools used for system certification can be selected from a “Reference Technology Platform” (to use this ARTEMIS term).

As already outlined in the combined deliverable D 2.1.4&D4.1.2, some standards follow rather a process-oriented approach, some rather a product and technology oriented approach, or a mixed approach; besides these differences, some standards define quite detailed techniques/measures to achieve certain safety goals (SIL levels in IEC 61508), others define rather goals (DO 178B/C), to which the user can map his techniques and tools.

### 4.5.1 IEC 61508 – 3: How Techniques are described and recommended

Existing V&V techniques and methods for system certification and their rigour level required to achieve the desired SIL are referenced in the mandatory and informative tables of IEC 61508-3. A state-of-the-art collection of detailed descriptions of available methods and techniques is contained in part 7 of IEC 61508, the references to the related detailed descriptions in Part 7, Annexes B and C, are given in column “Ref” of tables A.x and B.x of part 3.

IEC 61508 Part 7 (informative) provides in 3 Annexes A, B and C an overview of techniques and measures for E/E/PE safety-related systems for

- control of random hardware failures (see IEC 61508-2) (Annex A)
- avoidance of systematic failures (see IEC 61508-2 and IEC 61508-3) (Annex B)
- achieving software safety integrity (see IEC 61508-3)(Annex C),

Annex C, the most relevant one in context of V&V methods and techniques, consists of more than 50 pages of detailed descriptions (about 100 techniques and methods covering the whole SW life cycle, among these a considerable number of techniques and methods for analysis and testing). These will not be repeated here, but the approach taken will be explained in an examples, namely “Model-based Testing”. In Appendix 10.2, another example will be provided on “Model-based Test Case Generation”. These examples are chosen, because the method and technology is further developed in SafeCer (see 5.4).

The following tables describe the parts of IEC 61508-3 Ed. 2.0 Software Requirements (2010) which are relevant with respect to validation and verification of software parts of E/E/PE (electrical/electronic/programmable electronic) safety related systems.

The standard IEC 61508-3 consists of the textual description of the software requirements throughout the system safety lifecycle, of the normative Annex A with tables A1 to A10 “Guide to selection of techniques and measures”, the informative Annex B with detailed tables B.1 to B.9 and the associated informative Annex C “Properties for software systematic capability”. Further Annexes D to G will not be considered here.

Each normative table A.x contains a selection of techniques/measures to fulfil software (safety relevant) requirements within a certain phase of the safety life cycle or of support tools. For the V&V part relevant tables:
• A.2 software architecture design,
• A.5 software module testing and integration,
• A.9 software verification

are primarily considered here. Each technique/measure has a reference to details in tables B.x or to a detailed description in Part 7 of IEC 61508. Details relevant for V&V are in Tables

• B.2 “Dynamic analysis and testing” and
• B.3 “Functional and black box testing”.

In both tables A or B is provided a recommendation “R” (recommended), “HR” (highly recommended), or, in some cases, “NR” (not recommended) respectively “---” (neutral, no recommendation provided, neither positive nor negative) for each technique/measure and for each of the SILs (Safety Integrity Level) SIL1 to SIL 4) (one per line in the table). Appropriate techniques/measures shall be selected according to the safety integrity level. If alternative or equivalent techniques/measures are possible, a letter is following the number (e.g. in table A.2, 11a to 11d are equivalent, or 13a to 13c, and only one of them has to be satisfied, see 9.2). Any deviation from this concept has to be argued separately and proven to satisfy the general requirements particularly of chapter 7 of this standard.

The reference tables for “Model-based testing” are shown under Table A.5 in Figure 6 and Table C.5 in Figure 8. Further references are in Figure 23 (Table B.3, see chapter 9.2) and Figure 24.

Table A.5 – Software design and development – software module testing and integration

<table>
<thead>
<tr>
<th>Technique/Measure *</th>
<th>Ref.</th>
<th>SIL1</th>
<th>SIL2</th>
<th>SIL3</th>
<th>SIL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Probabilistic testing</td>
<td>C.5.1</td>
<td>---</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>2 Dynamic analysis and testing</td>
<td>B.6.5 Table B.2</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>3 Data recording and analysis</td>
<td>C.5.2</td>
<td>No recommendation provided, neither positive nor negative</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 Functional and black box testing</td>
<td>B.5.1 B.5.2 Table B.3</td>
<td>No recommendation provided, neither positive nor negative</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 Performance testing</td>
<td>Table B.6</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>6 Model based testing</td>
<td>C.5.27</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>7 Interface testing</td>
<td>C.5.3</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>8 Test management and automation tools</td>
<td>C.4.7</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>9 Forward traceability between the software design specification and the module and integration test specifications</td>
<td>C.2.11</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>10 Formal verification</td>
<td>C.5.12</td>
<td>---</td>
<td>---</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>
NOTE 1  Software module and integration testing are verification activities (see Table B.9).

NOTE 2  See Table C.5.

NOTE 3  Technique 9. Formal verification may reduce the amount and extent of module and integration testing required.

NOTE 4  The references (which are informative, not normative) “B.x.x.x”, “C.x.x.x” in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

*Appropriate techniques/measures shall be selected according to the safety integrity level.

Figure 6: “Model based testing”- technique for Software Design and development in IEC 61508-3.

The ranking of techniques and measures is linked to the concept of “effectiveness” as used in IEC 61508-2. Since a large number of factors influence effectiveness and since there is no general rule (algorithm) how to best combine techniques and measures in all applications, Annex C provides some guidance and a rationale how to achieve software systematic capability.

Tables C.x define 4 to 8 desirable properties which should be achieved with a certain rigour, e.g. for C.12 “Detailed properties – dynamic analysis and testing”:

- Completeness of testing and integration with respect to the software design specification
- Correctness of testing and integration with respect to the software design specification (successful completion)
- Repeatability
- Precisely defined testing configuration

The rigour is defined as:

R1: without objective acceptance criteria, or with limited objective acceptance criteria, e.g., black-box testing based on judgement, field trials.

R2: with objective acceptance criteria that can give a high level of confidence that the required property is achieved (exceptions to be identified & justified); e.g., test or analysis techniques with coverage metrics, coverage of checklists

R3: with objective, systematic reasoning that the required property is achieved, e.g. formal proof, demonstrated adherence to architectural constraints that guarantee the property.

- this technique is not relevant to this property

<table>
<thead>
<tr>
<th>SIL</th>
<th>Rigour R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 or 2</td>
<td>R1</td>
</tr>
<tr>
<td>3</td>
<td>R2 where available</td>
</tr>
<tr>
<td>4</td>
<td>Highest rigour available</td>
</tr>
</tbody>
</table>

Figure 7: Rigour level and SIL
**Table C.5 – Properties for systematic integrity – Software design and development – software module testing and integration**

*(See 7.4.7 and 7.4.8. Referenced by Table A.5)*

<table>
<thead>
<tr>
<th>Technique/Measure</th>
<th>Completeness of testing and integration with respect to the software design specification</th>
<th>Correctness of testing and integration with respect to the software design specification (successful completion)</th>
<th>Repeatability</th>
<th>Precisely defined testing configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R1 (R2 if operational profile coverage targets are defined, justified and met)</td>
<td>R1 (R2 if required outputs are defined, justified and met)</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>R1 (R2 if structural coverage targets are defined, justified and met)</td>
<td>R1 (R2 if required outputs are defined, justified and met)</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>3</td>
<td>_</td>
<td>R1 Promotes consistency in testing procedures</td>
<td>R2</td>
<td>If Fault records/test Logs include details of software baseline</td>
</tr>
<tr>
<td>4</td>
<td>R1 (R2 if operational profile coverage targets are defined, justified and met)</td>
<td>R1 (R2 if required outputs are defined, justified and met)</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>5</td>
<td>_</td>
<td>R1 (R2 if required outputs are defined, justified and met)</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Technique/Measure</td>
<td>Properties</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Completeness of testing and integration with respect to the software design specification</td>
<td>Correctness of testing and integration with respect to the software design specification (successful completion)</td>
<td>Repeatability</td>
<td>Precisely defined testing configuration</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Model based testing</td>
<td>R2 MBT allows early exposure of ambiguities in specification and design, the MBT process starts with requirements R3 If rigorous reasoning is applied to modelling, and TCG (Test Case Generation) is used</td>
<td>R2 Evaluation of results and regression test suites is a key benefit of MBT R3 If rigorous modelling approach is applied, then objective evidence of coverage is possible</td>
<td>R3 MBT (with TCG) aims at automatic execution of generated tests</td>
</tr>
<tr>
<td>7</td>
<td>Interface testing</td>
<td>–</td>
<td>R1 (R2 if required outputs are defined, justified and met)</td>
<td>–</td>
</tr>
<tr>
<td>8</td>
<td>Test management and automation tools</td>
<td>R1 (R2 if test coverage targets are defined, justified and met)</td>
<td>–</td>
<td>R1 Automation promotes consistency</td>
</tr>
<tr>
<td>9</td>
<td>Forward traceability between the software safety requirements specification and the module and integration test specifications</td>
<td>R1 Confidence that the test specification addresses the software safety requirements</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Technique/Measure</td>
<td>Properties</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Completeness of testing and integration with respect to the software design specification</td>
<td>Correctness of testing and integration with respect to the software design specification (successful completion)</td>
<td>Repeatability</td>
<td>Precisely defined testing configuration</td>
</tr>
<tr>
<td>10 Formal verification</td>
<td>R3 If rigorous reasoning is applied to construction of test cases to show that all aspects of design have been exercised</td>
<td>R3 Gives objective evidence of meeting all of the software safety requirements</td>
<td>R1 If support tools unavailable R2 If tool supported</td>
<td>_</td>
</tr>
</tbody>
</table>

Figure 8: Properties of “Model-based Testing” as technique described in IEC 61508-3, Table C.5, for systematic integrity, software module testing and integration.

Finally, the Technique/Measure is described, as indicated as “Ref” in Tables A.5 and B.2 and B.3, in IEC 61508-Part 7 under C.5.27:

C.5.27 Model based testing (Test case generation)

NOTE This technique/measure is referenced in table A.5 (and C.5) of IEC 61508-3.

Aim: To facilitate efficient automatic test case generation from system models and to generate highly repeatable test suites.

Description:

Model-based Testing (MBT) is a black-box approach in which common testing tasks such as test case generation (TCG) and test results evaluation are based on a model of the system (application) under test (SUT). Typically, but not only, the systems data and user behaviour are modelled using Finite state machines, Markov processes, decision tables or the like (El-Far, 2001, generalized). Additionally, model-based testing can be combined with source code level test coverage measurement, and functional models can be based on existing source code.

Model-based Testing is the automatic generation of efficient test cases/procedures using models of system requirements and specified functionality (SoftwareTech, 2009).

Since testing is very expensive, there is a huge demand for automatic test case generation tools. Therefore, model-based testing is currently a very active field of research, resulting in a large number of available TCG (Test Case Generation) tools. These tools typically extract a test suite from the behavioural part of the model, guaranteeing to meet certain coverage requirements.

The model is an abstract, partial representation of the system under test's (SUT) desired behaviour. From this model, test models are derived, building an abstract test suite. Test
cases are derived from this abstract test suite and executed against the system, and tests can be run against the system model as well. MBT with TCG is based on and strongly related to use of formal methods, so recommendations are similar with respect to safety integrity levels (SIL): HR (highly recommended) for higher SILs, and not required for lower SILs.

The specific activities in general are:

- build the model (from system requirements)
- generate expected inputs
- generate expected outputs
- run tests
- compare actual outputs with expected outputs
- decided on further action (modify model, generate more tests, estimate reliability/quality of the software)

Tests can be derived with different methods and techniques for expressing models of user/system behaviour, e.g.

- by using decision tables
- by using finite state machines
- by using grammars
- by using Markov Chain models
- by using state charts
- by theorem proving
- by constraint logic programming
- by model checking
- by symbolic execution
- by using an event-flow model
- reactive system tests: parallel hierarchical finite automaton
- ..etc.

Model-based Testing is specifically targeting recently the safety critical domain. It allows for early exposure of ambiguities in specification and design, provides the capability to automatically generate many non-repetitive efficient tests, to evaluate regression test suites and to assess software reliability and quality, and eases updating of test suites.

A thorough overview is provided by ElFar (2001) and SoftwareTech 2009 (see references), other details and domain specific issues are discussed in the other references.

### 4.5.2 Time-triggered Architecture

IEC/PAS 62814 is an evolving Public Specification (not a mandatory standard, but guidance, and considered a pre-standard) currently being developed by IEC TC 56 (Dependability). It promotes re-usability driven software development. It focuses on requirements on functionality and tests of software products containing reusable components in a totally domain-independent manner, thus addressing “Cross-Domain” and “Other Domain” issues. It addresses two aspects of components reuse:

- “Build-for-reuse” (planned production of reusable components) and
- “Build-by-reuse” (planned production of systems using reusable components)

It defines processes for both and a combined process (see Figure 9). The overall development process is as usual consisting of typical phases of planning, requirements elicitation, analysis & design, specification implementation, code & documentation, testing, which is a slightly simplified waterfall model. Typical new processes are the compliance check of available reusable components with build-for-reuse component requirements on the one hand, and of reusable components properties with system requirements of the system to build by reuse on the other
A second pair of processes is the pre-store process with iterative reusability characteristics, tests of the components developed for reuse and the pre-use process before a reusable component is sent as qualified for the specific reuse to the system to be built.

**Figure 9: Combined process for “Build-for-reuse” (left part) and “Build-by-reuse” (right part)**

The document gives pragmatic, generic guidance on:
- Validation, re-validation and reliability of software reuse
- Dependability and reuse aspects of software/hardware interaction
- Software reuse assurance
- Build-for-reuse: validation and qualification of components to be reused.
- Build-by-reuse: validation and qualification of the receiving system.

Annex C describes, besides basic concepts about software component testing, a validation process for reusable software components, distinguishing between vendor-oriented (see Figure 10) and user – oriented component testing, and distinguishing between a validation process for completely reused and for adapted and customized components (see Figure 11).
The PAS defines Maturity-Levels 0-4 for a component level process (like CMMI: ad-hoc, standard, managed, certified, systematic component testing). Interesting is the short chapter on “Emerging techniques for component integration”, classifying the techniques into
• Component-based interaction approach (the model represents three types: API-based, event-based and message-based interactions),
• UML based approach (UML-based models (component collaboration and sequence diagrams) used as system-level test model to support component integration and re-integration, and system regression testing,
• Hybrid approach (graphic representation model to present white-box and black-box information from specification and implementation respectively (for test case identification and reuse).

Annex E contains useful checklists for reuse of software (e.g. for “qualified as new” components of IEC 62309).

4.6 Comparative table of V&V methods for system certification

Results of the previous sections are collected and analysed to generate a comparative, which permits us to compare different V&V methods and techniques for system certification.

Table 14 summarizes all the V&V methods categorized according to the classification proposed in D2.4.1

<table>
<thead>
<tr>
<th>Method/Technique</th>
<th>Automotive</th>
<th>Avionic</th>
<th>Railway</th>
<th>Space</th>
<th>Cross domain/Other domains ¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walkthroughs and inspections</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Safety Assessment: FTA, FMEA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Requirement validation</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Sizing and Schedulability analysis</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Static analysis</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Traceability analysis</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Formal proof of correctness: model checking and formal verification</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Reviews and buddy checks</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Model-Based testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>White Box:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>• Fault Injection</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>• Dynamic Analysis</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

⁰ Cross domain/other domains information is based primarily on IEC 61508 requirements, partially on IEC 60601 (medical devices). The recommendation level (R, HR) of techniques and measures (tables A of IEC 61508, normative; tables B, informative) and the rigour (effectiveness) of techniques and measures (R1-R3) (tables C of IEC 61508, informative) differ depending on SIL and the context in which they are applied.
<table>
<thead>
<tr>
<th>Method/Technique</th>
<th>Automotive</th>
<th>Avionic</th>
<th>Railway</th>
<th>Space</th>
<th>Cross domain/Other domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black box testing:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Interface testing</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Stress Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Back to Back testing</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Design Based functional testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Robustness Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statistical Testing</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional and performance testing</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code dynamic analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other testing methods: stress test, control unit, etc.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

As it can be observed in Table 14, safety and critical systems are very thorough to verify and validate software system. Therefore, all different domains use several of these methods.
5 Adaptation and combination of existing methods and tools to system certification

This section describes the adaptation and combination of existing V&V methods to system certification.

5.1 Adaptation and combination of OCRA contract verification methods

Contract-based verification is a formal verification technique and therefore is one of the verification methods that in Section 2 have been referred to as “formal proof” or “formal verification”. It can be applied in different domains. Moreover, from the point of view of system certification, described in Section 2, contract-based verification focuses on the “verification of system design” and “verification of software architectural design”. In order to be adapted to the system certification process, contract-based verification should therefore be combined with the other required V&V methods such as design/software/hardware safety analysis, safety requirements validation, hardware/software verification and testing.

This section shows how OCRA contract-based system verification can be combined with system fault-tree analysis and formal requirements validation. Adaptations and combinations with further techniques such as test case generation and software verification will be developed in nSafeCer.

5.1.1 Contract-based verification

Contract-based verification is a compositional verification approach where each component is enriched with contracts that specify the guarantees provided by the component together with the assumptions on the environment for a correct use of the component. The approach is compositional because the correctness of a composite component is obtained by composing the arguments of correctness of the subcomponents without using the internals of the subcomponents. This reduces the cost of verification but also allows for reusing or substituting a component. In fact, since the internals of components are not used in the derivation of the correctness of their composition, the implementation of a component can be reused or substituted as far as it satisfies the component contracts.

In this setting, the system is described as a composite component. Its subcomponents are iteratively decomposed in other components until defining the basic/primitive components. In the compositional approach only these primitive components are associated with an implementation, for example a state machine, describing the concrete behaviours of the components. The system implementation consists of the composition of the implementations of the primitive components according to the composition constraints defined by the system architecture. If the implementations of the primitive components are correct and the contract refinement of the system architecture is correct, then also the system is guaranteed to be correct.

5.1.2 Combination with fault-tree analysis

Fault-tree analysis is a validation technique applied to a system description in order to check the dependency of system properties on the occurrences of faults. In particular, given a system implementation, a set of faults events, and a top-level failure condition, the fault-tree analysis produces a tree structure that shows how the top-level condition depends on different sets of occurrences of fault events.

We can combine fault-tree analysis with the contract-based verification by considering the implementation of primitive components taking into account the occurrence of fault events and considering as top-level failure condition the negation of the guarantee of a system contract.

In order to adapt the fault-tree analysis to the contract-based specification of the system architecture, we have to combine the implementations of the primitive components building the monolithic implementation of the system. We can then pass this implementation to the fault-tree analysis procedure, which generates the fault-tree.
A more integrated combination will be investigated in nSafeCer, in order to integrate the automatic generation of models extended with faults, and to produce the fault-tree in a compositional way, avoiding the construction of the monolithic system implementation.

5.1.3 Combination with formal requirements validation

Formal requirements validation is a validation technique applied to the formalization of requirements in order to check if the formal requirement specifies the right function or constraint. In particular, given a set of requirements and their formalization into a formal property specification language, the validation performs a series of checks in order to see if the specification contains some errors such as inconsistent requirements, wrong values, variables, or conditions, missing assumptions, over-constrained requirements. The specification of these checks is either automatically generated based on the structure of the formal property or manually specified by means of additional scenario properties.

We can combine formal requirements validation with the contract-based verification by considering the formalization of requirements in contracts. We can apply the validation not only to system contracts but also to any component contract. In fact, while the refinement verification guarantees that the contracts of the subcomponents are a correct refinement of the parent component, the contract may be still inconsistent or strong enough to remove desired behaviours.

In order to adapt the requirements validation to the contract-based specification of requirements, we apply the validation checks to every contract in the system architecture, both to the assumption assertion and the implication “assumption implies guarantee”.

5.2 Adaptation for automotive domain certification

For the automotive domain, the real-time properties of the embedded systems are very important. Timing analysis is quite challenging since components in a system typically will share resources such as processing node and network. This means that, to be able to ensure that a component that has been analysed in isolation maintains the same behaviour inside a system, the same isolation must be present in the system. This may be hard to achieve and typically also introduces pessimism in the times. For example, in a WCET analysis it will have to be assumed that no data is available in the cache while in a particular system most of the data will be in the cache. Moreover, WCET analysis is not meaningful without a particular target which means that a range of WCET values would have to be provided with a component. Thus, for CBD it is a challenge to find the correct level of providing timing information such that it is useful in a reuse context. This challenge will be addressed in nSafeCer by adapting existing timing analysis methods to the specifics associated with component reuse.

5.3 Adaptation and combination of methods and tools for avionic domain certification

Existing verification and validation methods and tools have to be combined in order to ensure safety and certification of complex avionics, all over the development levels.

In this purpose, the mapping tools developed by Magillem (pSafeCer partner, MDS acronym) enable the management of the mapping and versioning of different types of data from the development processes of complex systems. We will extend in nSafeCer its features by enabling the computation of the impacted data and verification points when changes occur in systems components or requirements or tests etc... Therefore, the MDS tools are combined with existing tools of traceability management and testing or verification in order to fulfil the safety, certification and re-certification objectives.
5.3.1 Mapping Objectives

Providing guarantee that software requirements are correctly implemented by the software life cycle processes is a key issue in systems certification. In that purpose, it is necessary to take into account different parameters. We will reason in the context of avionics certification based on the DO-178C guidance, but the method is easily generalized to other contexts and domains.

On one hand, software requirements are strongly linked to the software architecture, that is, the way software tasks are partitioned. In particular, each low-level requirement is specific to one or more specific software task components. On the other hand, software requirements implementability depends on the target platform performances and its architecture. Indeed, software tasks are mapped to hardware components following integration requirements. Therefore, each software task component and its low-level requirement are dependable of the hardware component and its resources performance.

Low-level requirements are correctly implemented when the needed resources are available. We have to find a traceability management between SW requirements and dependencies on HW. For example, the execution time of a software application depends on three factors amongst others:

- The performance of a HW accelerator.
- The memories size.
- The number of processors.

Before the validation and verification steps, we need to interconnect models, data and descriptions of different kind and coming from different sources. We need to map, on one hand, low-requirements and the software architecture, and on the other hand, to map resources performance description and the hardware platform architecture. Finally, based on the integration requirements, a mapping is built between software tasks and hardware components. Thus, we obtain a complete information flow between requirements and resources performances.

We can use validation and verification techniques to verify that the requirements are well implemented. We can build graphs based on exploration techniques to evaluate the conditions of implementability. These mapping mechanisms are also very relevant when it comes to re-certify a system due to changes either in software architecture or hardware architecture and integration requirements.

Instead of re-certifying the whole system, we can compute the data impacted by the changes and the corresponding impacted verification points.

5.3.1.1 Mapping low-level requirements and software architecture

The MDS mapping tool aims to build the correspondence between software architectures and low-level requirements of a given application.

The MDS mapping tool takes as inputs:

- The low-level requirements describing different kind of constraints, such as:
  - Timing constraints.
  - Memory size allocations.
  - Partitioning.
- The software architecture.

Each input has a specific document format. For example, low-level requirements might be provided in an Excel document or can even have a graphical representation. The software architecture might be an AADL model or a MyCCM representation form.

The MDS tool builds a tree graph representation of each input data following users provided mapping rules; it computes the resulting graph of the software architecture annotated with low-level requirements. Mapping rules indicate the kind of relation between software components and
low-level requirements. The resulting graph can be visualized as a table with entries, in which, for each task component corresponds one or more low-level requirements (see Figure 12). The table is the representation we use to reduce complexity in the following figures.

5.3.1.2 Mapping resources performance description and hardware architecture

The MDS mapping tool aims to build the correspondence between a given hardware architecture and the resources performance description of the target platform.

The MDS mapping tool takes as inputs:

- The hardware platform structure.
- The resources performances description, such as:
  - The memory overhead.
  - The clocks frequency.
  - Execution times.

Each input can also have a specific document format. The hardware platform structure can be modelled following the IP-XACT standard. The resources performances description can either be identified as IP-XACT parameters or vendor extensions, or specified in other document formats.

The MDS tool builds a tree graph for each input data following users provided mapping rules; it computes the annotated graph of the hardware architecture with the resources performances annotations. The resulting graph can be visualized as a table with entries, in which each hardware component has resource performances associated (see Figure 13).
5.3.2 Managing Versioning Issues

The first presented feature of the MDS tools is to build and manage the mapping between different kinds of data. An essential issue is to keep the coherency of the mapping between given input data, when modifications are done on one of the inputs. The second feature is then to handle versioning issues by keeping the coherency of the initial mapping.

5.3.2.1 Changes detection in hardware architecture

The MDS tool manages the tracing of differences between two versions of a platform:

- Versions of components (parameters, configurations, etc.) used in the platform. That is, when a change occurs in a hardware component.
- Architectural description of the platform (number and choices of processors, accelerators, size of memories, buses architecture and arbitration schemas, etc.)
- Performance of the hardware in regards with software execution (clock frequencies, timing characteristics, etc.).

In Figure 14, we give an example showing the introduction of a new version of a given processor “process1” into “process1-v2”. The MDS mapping tool identifies the change in the hardware architecture (step1) and updates the initial stored mapping (step2).
### 5.3.2.2 Changes Detection in the Software Architecture

The MDS tool also manages the tracing of differences between two versions of given software architecture:

- When it comes to changes in components versions (parameters, configurations, etc.).
- Changes in the partitioning.

In Figure 15, we give an example showing the introduction of a new version of a given software component “coding” into “coding-v2”. The MDS mapping tool identifies the change in the software architecture (step1) and updates the initial stored mapping (step2).
5.3.2.3 Changes detection in integration requirements

The MDS tool manages the integration requirements traceability. If a change occurs in an integration requirement, it detects the impacted components of the change. Therefore, it also gives the corresponding resources performances and low-level requirements that may be impacted.

5.3.3 Computing impacts

When a change occurs in the software or the hardware architecture, or in the integration requirements, it is a key issue to compute the impacts of the changes on the rest of the software development and verification processes.

The MDS mapping tools enable the management of the mapping and versioning of different types of data. We extend its features by enabling the computation of the impacted data and verification points when changes occur.

In the Figure 16, we give the example showing the introduction of a new version of a given processor “process1” into “process1-v2”. The MDS mapping tool updates the corresponding mapping (step1) then looks in the software requirements trace for the impacted data (step2). In the example, software component “coding” with its requirement “LR1” are directly impacted due to an integration requirement. The MDS mapping tool outputs all the impacted data (step3). It then has to draw a list of the impacted verification to review (step4).

![Figure 16: Impacts Computation](image)

5.4 Adaptation of Model-Based mutation-based Testing and Test Case Generation

AIT, together with a few other partners, developed in the context of MOGENTES, an FP7 STREP “Model-Based Generation of Tests for Dependable Embedded Systems” a tool now called MoMuT (Model Mutation Based Test Analysis and Test Design) for UML models. MoMuT enables efficient automated test design by model-driven test case generation, basically fault-mutation driven. The current version derives its output from UML State Machine Diagrams and, considering the methods described in chapter 2 and 6, aims at black-box testing and is requirements drive.
Adaptations planned:
- Other languages: SCADE under development
- White box testing support
- URN envisioned

![Diagram of Test Case Generation]

**Figure 17: Automatic Test Case Generation**

The TCG (Test Case Generation) principles:
- UML Test Model (represents requirements)
- Mutated Model (represents faulty implementation)
- Test conformance relation (covers under specification)
- Counter example search by (bounded) model checking
- Counter example to conformance is a test case exposing at least the introduced fault/mutation

Additive Mutation Based TCG provides additional features:
- Reuse of legacy test cases
- Impact analysis: identify TCs to rerun
- Long and short TCs – for regression testing and debugging

The outcome is an efficient test suite with (a) maximum mutation coverage and (b) minimum test effort.

The new approach aims at a combined Black-Box/White-Box TCG to achieve the advantages of both methods in an efficient manner:
With respect to SafeCer, the following adaptations/extensions are planned (and required) to fulfill the SafeCer concept of compositional approach to V&V and certification and to include the V&V of contract based composition of systems:

- Extend component test cases to system test cases
  - Given: MB test cases for a subsystem + a model for subsystem and system
  - Derive test cases for the system, covering as many as possible of the subsystem
- Compositional Testing
  - Test individual components for conformance to their individual spec.
  - Infer the overall system conforms to overall spec., without explicitly testing it.

Features which already exist to fulfill SafeCer requirements for certifiability and reuse of arguments (test results (outputs)) are:

- Traceability
  - (Requirements -) Model Elements – Mutants – Test Cases (– Test Result)
- TCG for changed requirements - recertification
  - Reuses as many TCs as possible
  - Highlights changed areas.

Figure 18: Combined Black-Box/White-Box Testing
6 Contribution to Overall pSafeCer Objectives

The main pSafeCer challenges are to reduce the cost of qualification, certification and verification and to provide a framework for compositional development and certification of systems.

Therefore, the main considered issue is to increase efficiency and reuse in development and certification of safety-relevant embedded systems by providing process and technology that enable composable qualification and certification, i.e. qualification/certification of systems/subsystems based on reuse of already established arguments for and properties of their parts.

In this context, the main objective of this document is to describe different existing V&V methods to support system certification, and how to adapt them to the pSafeCer requirements. In the present document, we outline this adaption, which will be developed in nSafeCer project.
7 Conclusions

Verification and validation is a crucial part of the development life cycle of an embedded system. The results of the V & V process are an important component in the safety case, which is heavily used to support the certification process.

According to [TA], the pSafeCer WP2.4 is responsible for adapting and extend verification and validation techniques to the certification components and their composition to the chosen languages.

Therefore, this deliverable D2.4.3 aims to describe different existing V&V methods to support system certification, how to adapt them to the pSafeCer requirements. To lead this adaptation, we analyse the current requirements in each domain and how they are validated and verified in the certification process. As main conclusion, we can observe that safety and critical systems are very thorough to verify and validate software system.

Adaptations and combinations with further techniques such as test case generation and software verification will be developed in nSafeCer.
8 References

<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>[D1.0.2v2.2]</td>
<td>pSafeCer WP1 D1.0.2: 111121_Detailed_Requirements_pSafeCerV2_2.xlc</td>
</tr>
<tr>
<td>[D2.4.1v1.2]</td>
<td>pSafeCer WP2 D2.4.1: A report describing the existing verification &amp; validation methods for product-based certification of components and systems</td>
</tr>
<tr>
<td>[DO-178C]</td>
<td>Software Considerations in Airborne Systems and Equipment Certification</td>
</tr>
<tr>
<td>[ECSS-E-ST-40C]</td>
<td>European Cooperation for Space Standardization</td>
</tr>
<tr>
<td>[ECSS-Q-ST-80C]</td>
<td>European Cooperation for Space Standardization</td>
</tr>
<tr>
<td>[EN 50126]</td>
<td>CENELEC European Standard EN50126-1:1999</td>
</tr>
<tr>
<td>[EN 50128]</td>
<td>CENELEC European Standard EN50128-1:2001</td>
</tr>
</tbody>
</table>
Appendices

9.1 Traceability for Avionic domain

In this section a detailed description of the traceability requirement for avionic domain is provided. Thus, we also describe the software life cycle development processes and the outputs that have to be verified and traced.

9.1.1 Software Life Cycle Development

This section describes the software life cycle. It includes the following processes:

- The software planning process defines and coordinates the activities of the software development processes and the integral process for a project.
- The software development processes produce the software product.
- The integral processes ensure the correctness and control, and confidence in the software life cycle processes and their outputs. They are performed concurrently with the planning and development processes through the software life cycle.

The software development processes produce the software product with respect to software requirements that are provided by the system life cycle. Figure 19 is a representation of the software development processes and the produced outputs. Each step is explained in the subsections below.

![Figure 19: Software Development Life Cycle Overview](image)

9.1.1.1 Software Requirements Process

The software requirements process uses inputs from the system life cycle process, including system requirements, hardware interface and system architecture, in order to develop high-level requirements and define derived high-level requirements. These high-level requirements include functional, performance, interface and safety-related requirements.

The objectives of software requirements are the following:
Each system requirement allocated to software should be specified in high-level requirements.

High-level requirements should conform to standards, be verifiable and consistent. They should be stated in quantitative terms with tolerances where applicable.

They should not describe design or verification details.

Ambiguities, inconsistencies, undefined conditions and incorrectness should be identified in the software requirements.

Derived high-level requirements are also development. They are provided to the system processes (including the system safety assessment process) for verification.

The process output is represented by the Software Requirements Data, including the High-level requirements and the derived high-level requirements.

9.1.1.2 Software Design Process

The high-level requirements are refined through one or more iterations in the software design process to develop the software architecture and the low-level requirements. The latters are used to implement source code.

The activities for this process have to guarantee some properties such as:

- Low-level requirements and software architecture have to be traceable, verifiable and consistent.
- Using additional data in case of partitioning or other architectural means.
- Defining interfaces between software components (data flow, control flow).

Derived low-level requirements are also developed and provided to the system processes, including the system safety assessment process for verification.

The process output is represented by the Design Description which includes the software architecture, the low-level requirements and the derived low-level requirements.

9.1.1.3 Software Coding Process

In the software coding process, the source code is implemented from the software architecture and the low-level requirements. Its output is the Source Code.

The activities for this process have to guarantee some properties such as:

- The source code should implement the low-level requirements and conform to the software architecture and the software code standards.
- The inadequate or incorrect inputs should be provided to the software requirements, design and planning processes.
  The use of auto-code generators should implement the constraints defined in the planning process.

9.1.1.4 Software Integration Process

The integration process develops the integrated system or equipment by loading the executable object code into the target hardware. It consists of software integration and software/hardware integration. Therefore, the process takes into account integration requirements.

The inputs of the process are the software architecture and the source code.

The outputs of the integration are the Object Code, the executable object code, a parameter data item file and the compiling, linking and loading data.

9.1.2 Requirements Traceability Activities and Objectives
Software development process traceability activities include trace data showing bidirectional association between:

- System requirements allocated to software and high-level requirements.
- High-level requirements and low-level requirements.
- Low-level requirements and Source code.

It is also important to have a traceability process between the software development process activities and integration requirements. Integration requirements give the mapping between software tasks and hardware components, in order to load correctly the executable code in the hardware platform. It ensures a correct referencing in the requirements using object names from specification documents (see example Figure 20).

---

**Figure 20: Integration Requirements**

Figure 21 gives the overall traceability management between software development requirements and HW/SW integration requirements.

Traceability between software development requirements enables the verification of the following points:

- The system requirements allocated to software are actually implemented.
- The verification of the complete implementation of the high-level requirements.
- The verification that no source code implements an undocumented function.
- The verification of the complete implementation of the low-level requirements.

The traceability process also enables a visibility of the derived high-level requirements which are not directly traceable to system requirements and the visibility of the derived low-level requirements which are not directly traceable to high-level requirements and to the architectural design decisions.

These objectives are part of the certification process.
9.1.3 Software Integral Processes

The Integral processes main objectives are detecting and reporting errors that may have been introduced during software development process in high-level requirements, low-level requirements and, source code and the integration processes.

Thus, the integral processes ensure the correctness and control, and confidence in the software life cycle processes and their outputs. They are performed concurrently with the planning and development processes through the software life cycle.

It includes the following processes for software:

- Verification
- Quality assurance
- Configuration management
- Certification liaison

Verification is a technical assessment of the outputs of the software planning process, software development process and the software verification process itself.

The software verification process is applied as defined in the software verification plan.

Verification in this context is a combination of reviews, analyses and tests.

9.1.4 Software Production for Airborne Systems

The DO-178C provides guidance for the software production for airborne systems and equipment. It gives guidance for satisfying certification requirements, so as the production performs their intended function with a level of confidence in safety that complies with airworthiness requirements. The guidance also gives the main objectives for software life cycle processes and describes the activities for achieving those objectives.

Mainly it provides the following (see Figure 22 for the representation of the overall certification processes interactions):

---

**Figure 21: Overall Integration Traceability**
The description of the relationship between the system life cycle processes, the hardware life cycle processes and the software life cycle processes. Indeed, the software life cycle interactions with the system and hardware life cycles are necessary for the software and hardware integration and verification processes. The software life cycle needs the system description and hardware definition in order to produce software which is compatible, in requirements, design and verification activities. The system safety assessment process is responsible for relating the errors in the software life cycle with the system failure condition(s) and the severity of that failure conditions by analysing the derived requirements provided by the software life cycle.

The description of the software life cycle processes. It gives the objectives for software life cycle processes and activities that provide a means for satisfying those objectives.

It produces data showing that the objectives have been satisfied. Therefore, it provides the list of software life cycle data for certification and the planned activities that have been performed for that purpose.

The description of software life cycle data used in the development, validation and verification for software development. It is indicated the objectives they have to satisfy and the corresponding activities and results to be done in the verification process.

**Figure 22: Overall certification processes interactions**

### 9.2 Further examples for Cross-Domain/Other Domain V&V techniques and measures

In section 0 Model-based Testing (Heimdahl 2005, Jacky 2008, Paradkar 2005, Utting 2007) was chosen as one of the methods and techniques relevant for certification of safety-critical systems to
be positioned in context of achieving a certain SIL according to IEC 61508-3. This is now continued for Model-based Test Case Generation and Execution. The relevant tables are

- B.3 “Functional and black-box testing” (technique/measure 2).
- B.2 “Dynamic analysis and testing” (technique/measure 4) and

The related properties and rigor levels can be derived from

- C.13 “Functional and Black-Box testing” (technique/measure 2)
- C.12 “Detailed properties – Dynamic analysis and testing” (technique/measure 4) and

In IEC 61508, Part 7, the technique/measure is addressed as C.5.27. (see section 5.4).

Relevant fields are marked in yellow.

**Table B.3 – Functional and black-box testing**

(Referenced by Tables A.5, A.6 and A.7)

<table>
<thead>
<tr>
<th>Technique/Measure</th>
<th>Ref</th>
<th>SIL1</th>
<th>SIL2</th>
<th>SIL3</th>
<th>SIL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Test case execution from cause consequence diagrams</td>
<td>B.6.6.2</td>
<td>---</td>
<td>---</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>2 Test case execution from model-based test case generation</td>
<td>C.5.27</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>3 Prototyping/animation</td>
<td>C.5.17</td>
<td>---</td>
<td>---</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>4 Equivalence classes and input partition testing, including boundary value analysis</td>
<td>C.5.7 C.5.4</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>5 Process simulation</td>
<td>C.5.18</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

**NOTE 1** The analysis for the test cases is at the software system level and is based on the specification only.

**NOTE 2** The completeness of the simulation will depend upon the safety integrity level, complexity and application.

**NOTE 3** See Table C.13.

**NOTE 4** The references (which are informative, not normative) “B.x.x.x”, “C.x.x.x” in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

*Appropriate techniques/measures shall be selected according to the safety integrity level.

**Figure 23: Functional and Black-Box Testing - model-based testing (MBT) referenced**
Table B.2 – Dynamic analysis and testing
(Referenced by Tables A.5 and A.9)

<table>
<thead>
<tr>
<th>Technique/Measure *</th>
<th>Ref</th>
<th>SIL1</th>
<th>SIL2</th>
<th>SIL3</th>
<th>SIL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Test case execution from boundary value analysis</td>
<td>C.5.4</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td></td>
</tr>
<tr>
<td>2 Test case execution from error guessing</td>
<td>C.5.5</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>3 Test case execution from error seeding</td>
<td>C.5.6</td>
<td>---</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>4 Test case execution from model-based test case generation</td>
<td>C.5.27</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>5 Performance modelling</td>
<td>C.5.20</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>6 Equivalence classes and input partition testing</td>
<td>C.5.7</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>7a Structural test coverage (entry points) 100% **</td>
<td>C.5.8</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>7b Structural test coverage (statements) 100% **</td>
<td>C.5.8</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>7c Structural test coverage (branches) 100% **</td>
<td>C.5.8</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>7d Structural test coverage (conditions, MC/DC) 100% **</td>
<td>C.5.8</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
</tbody>
</table>

NOTE 1 The analysis for the test cases is at the subsystem level and is based on the specification and/or the specification and the code.

NOTE 2 See Table C.12.

NOTE 3 The references (which are informative, not normative) “B.x.x.x”, “C.x.x.x” in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

*Appropriate techniques/measures shall be selected according to the safety integrity level.

**Where 100% coverage cannot be achieved (e.g. statement coverage of defensive code), an appropriate explanation should be given.

Figure 24: Dynamic Analysis and Testing – model-based testing (MBT) referenced
<table>
<thead>
<tr>
<th>Technique/Measure</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Completeness of testing, integration and validation with respect to the design specifications</td>
</tr>
<tr>
<td>1 Test case execution from cause consequence diagrams</td>
<td>R1</td>
</tr>
<tr>
<td>2 Test case execution from model-based test case generation</td>
<td>R2 MBT Model-based Testing is the automatic generation of efficient test cases/procedures using models of system requirements and specified functionality, it facilitates early error disclosure and understanding of consequences of specified requirements R3 If rigorous reasoning is applied to modelling, and TCG is used</td>
</tr>
<tr>
<td>3 Prototyping/animation</td>
<td>--</td>
</tr>
<tr>
<td>Technique/Measure</td>
<td>Properties</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Completeness of testing, integration and validation with respect to the design specifications</strong></td>
<td>Correctness of testing, integration and validation with respect to the design specifications (successful completion)</td>
</tr>
<tr>
<td>Repeatability</td>
<td>Precisely defined testing, integration and validation configuration</td>
</tr>
<tr>
<td><strong>Equivalence classes and input partition testing, boundary analysis</strong></td>
<td>R1 (If the input data profile is well defined and is manageably simple in structure)</td>
</tr>
<tr>
<td><strong>Process simulation</strong></td>
<td>_</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 25: Detailed properties of MBT-Test Case generation and execution for functional and black-box testing.

Table C.12 – Detailed properties – Dynamic analysis and testing (Referenced by Table B.2)

<table>
<thead>
<tr>
<th>Technique/Measure</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Completeness of testing and verification with respect to the software design specifications</strong></td>
<td>Correctness of testing and verification with respect to the software design specifications (successful completion)</td>
</tr>
<tr>
<td>Repeatability</td>
<td>Precisely defined testing and verification configuration</td>
</tr>
<tr>
<td><strong>Test case execution from boundary value analysis</strong></td>
<td>_</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Technique/Measure</td>
<td>Properties</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Completeness of testing and verification with respect to the software design specifications</strong></td>
<td>Correctness of testing and verification with respect to the software design specifications (successful completion)</td>
</tr>
<tr>
<td><strong>Repeatability</strong></td>
<td>Precisely defined testing and verification configuration</td>
</tr>
<tr>
<td>2 Test case execution from error guessing</td>
<td>R1</td>
</tr>
<tr>
<td>3 Test case execution from error seeding</td>
<td>R1</td>
</tr>
<tr>
<td>4 Test case execution from model-based test case generation</td>
<td>R2 Evaluation of results and regression test suites are a key benefit of MBT, if further facilitates understanding of consequences of specified requirements R3 MBT (with TCG) aims at automatic execution of generated tests R2 MBT is automation driven, testing configuration has to be precisely defined; execution of the generated tests is similar to black box testing with the possibility to be combined with source code level coverage measurement</td>
</tr>
<tr>
<td>5 Performance modelling</td>
<td>R1 (R2 if objective performance requirements)</td>
</tr>
<tr>
<td>6 Equivalence classes and input partition testing</td>
<td>R1 (If the partitions plausibly contain no non-linearities i.e. all members of a class are truly equivalent)</td>
</tr>
<tr>
<td>7 Structure-based testing</td>
<td>R1 (R2 if objective structural coverage targets)</td>
</tr>
</tbody>
</table>

Figure 26: Detailed properties of MBT-TCG and execution for functional and black-box testing.
9.3 Time-triggered Architecture

"Time-triggered Architecture (Scheidler 1998, Kopetz 2002, and Rushby 2002) is one of the techniques highly recommended for higher SILs for Software Architecture Design, see Figure 27. This is another example how appropriate techniques/measure may be derived from IEC 61508 tables.

In this example, it is made clear that for SIL 3 three methods/techniques 13a-c are highly recommended (HR) and therefore one has to be selected (if not argued otherwise), for SIL 4 one of the techniques/measure 13a or 13b has to be selected if not argued otherwise.

<table>
<thead>
<tr>
<th>Technique/Measure *</th>
<th>Ref.</th>
<th>SIL1</th>
<th>SIL2</th>
<th>SIL3</th>
<th>SIL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture and design feature</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Fault detection</td>
<td>C.3.1</td>
<td>---</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>2 Error detecting codes</td>
<td>C.3.2</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>3a Failure assertion programming</td>
<td>C.3.3</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>3b Diverse monitor techniques (with independence between the monitor and the monitored function in the same computer)</td>
<td>C.3.4</td>
<td>---</td>
<td>R</td>
<td>R</td>
<td>----</td>
</tr>
<tr>
<td>3c Diverse monitor techniques (with separation between the monitor computer and the monitored computer)</td>
<td>C.3.4</td>
<td>---</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>3d Diverse redundancy, implementing the same software safety requirements specification</td>
<td>C.3.5</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>R</td>
</tr>
<tr>
<td>3e Functionally diverse redundancy, implementing different software safety requirements specification</td>
<td>C.3.5</td>
<td>---</td>
<td>---</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>3f Backward recovery</td>
<td>C.3.6</td>
<td>R</td>
<td>R</td>
<td>---</td>
<td>NR</td>
</tr>
<tr>
<td>3g Stateless software design (or limited state design)</td>
<td>C.2.12</td>
<td>---</td>
<td>---</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>4a Re-try fault recovery mechanisms</td>
<td>C.3.7</td>
<td>R</td>
<td>R</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>4b Graceful degradation</td>
<td>C.3.8</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>5 Artificial intelligence - fault correction</td>
<td>C.3.9</td>
<td>---</td>
<td>NR</td>
<td>NR</td>
<td>NR</td>
</tr>
<tr>
<td>6 Dynamic reconfiguration</td>
<td>C.3.10</td>
<td>---</td>
<td>NR</td>
<td>NR</td>
<td>NR</td>
</tr>
<tr>
<td>7 Modular approach</td>
<td>Table B.9</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>8 Use of trusted/verified software elements (if available)</td>
<td>C.2.10</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>9 Forward traceability between the software safety requirements specification and software architecture</td>
<td>C.2.11</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>10 Backward traceability between the software safety requirements specification and software architecture</td>
<td>C.2.11</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>11a Structured diagrammatic methods **</td>
<td>C.2.1</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td></td>
<td>Methods</td>
<td>Table/Ref.</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>---</td>
<td>------------------------------------------------------------------------</td>
<td>------------</td>
<td>---</td>
<td>---</td>
<td>----</td>
</tr>
<tr>
<td>11b</td>
<td>Semi-formal methods **</td>
<td>Table B.7</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>11c</td>
<td>Formal design and refinement methods **</td>
<td>B.2.2, C.2.4</td>
<td>---</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>11d</td>
<td>Automatic software generation</td>
<td>C.4.6</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>12</td>
<td>Computer-aided specification and design tools</td>
<td>B.2.4</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>13a</td>
<td>Cyclic behaviour, with guaranteed maximum cycle time</td>
<td>C.3.11</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>13b</td>
<td>Time-triggered architecture</td>
<td>C.3.11</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>13c</td>
<td>Event-driven, with guaranteed maximum response time</td>
<td>C.3.11</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>14</td>
<td>Static resource allocation</td>
<td>C.2.6.3</td>
<td>-</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>15</td>
<td>Static synchronisation of access to shared resources</td>
<td>C.2.6.3</td>
<td>-</td>
<td>-</td>
<td>R</td>
</tr>
</tbody>
</table>

**NOTE 1** Some of the methods given in Table A.2 are about design concepts, others are about how the design is represented.

**NOTE 2** The measures in this table concerning fault tolerance (control of failures) should be considered with the requirements for architecture and control of failures for the hardware of the programmable electronics in IEC 61508-2.

**NOTE 3** See Table C.2.

**NOTE 4** The group 13 measures apply only to systems & software with safety timing requirements.

**NOTE 5** Measure 14. The use of dynamic objects (for example on the execution stack or on a heap) may impose requirements on both available memory and also execution time. Measure 14 does not need to be applied if a compiler is used which ensures a) that sufficient memory for all dynamic variables and objects will be allocated before runtime, or which guarantees that in case of memory allocation error, a safe state is achieved; b) that response times meet the requirements.

**NOTE 6** Measure 4a. Re-try fault recovery is often appropriate at any SIL but a limit should be set on the number of retries.

**NOTE 7** The references (which are informative, not normative) “B.x.x.x”, “C.x.x.x” in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

*Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. It is intended the only one of the alternate or equivalent techniques/measures should be satisfied. The choice of alternative technique should be justified in accordance with the properties, given in Annex C, desirable in the particular application.

**Group 11, “Structured methods”. Use measure 11a only if 11b is not suited to the domain for SIL3+4.*

*Figure 27: IEC 61508-3: “Time-triggered Architecture” as technique for “Software Architecture Design”.*
In IEC 61508-Part 7, it is described as Technique/Measure C.3.11:

**C.3.11 Safety and Performance in real time: Time-Triggered Architecture**

NOTE 1 This technique/measure is referenced in table A.2 of IEC 61508-3.

**Aim:**

Composability and transparent implementation of fault-tolerance into safety-critical real-time systems with predictable behaviour.

**Description:**

In a Time-Triggered Architecture (TTA) system, all system activities are initiated and based on the progression of a globally synchronised time-base. Each application is assigned a fixed time slot on the time-triggered bus, which contains the messages exchanged between the jobs of each application which can therefore be exchanged only according to a defined schedule. In event-driven systems, system activities are triggered by arbitrary events at unpredictable points in time. The key advantages of a TTA are (see reference Scheidler et. al., 1998):

- composability, which greatly reduces the effort required for testing and certifying the system;
- transparent implementation of fault-tolerance, which makes the architecture highly recommendable for safety-critical applications;
- Provision of a globally synchronised time-base, which facilitates the design of distributed real-time systems.

Communication between nodes is done using the Time-Triggered Protocol TTP/C (see Kopetz et. al., 1996) according to a static schedule, deciding when to transmit a message and whether a received message is relevant for the particular electronic module or not. Access to the bus is controlled by a cyclic time-division multiple access (TDMA) schema derived from the global notion of time.

The TTP/C protocol guarantees (see Rushby, 2002) four basic services (core services) in a network of TTA nodes (see reference Kopetz and Bauer, 2002):

- Deterministic and timely message transport: Transport of messages from the output port of the sending element to the input ports of the receiving elements within an a priori known time bound. A fault-tolerant transport service is offered by a time-triggered communication service that is available via the temporal firewall interface which eliminates control error propagation by design and minimises coupling between elements. The timely transport of messages with minimal latency and jitter is crucial for the achievement of control stability in real-time applications.
- Fault-tolerant Clock Synchronization: The communication controller generates a fault-tolerant synchronised global time base (with a precision within a few clock ticks) that is provided to the host subsystem.
- Consistent Diagnosis of Failing Nodes (Membership Service): The communication controller informs every SRU (“smallest replaceable unit”) about the state of every other SRU in a cluster with a latency of less than one TDMA round.
- Strong Fault Isolation: A maliciously faulty host subsystem (including its software) can produce erroneous data outputs, but can never interfere in any other way with the
correct operation of the rest of a TTP/C cluster. Fail silence in the temporal domain is guaranteed by the time-triggered behaviour of the communication controller.

Note 2 Other time-triggered protocols are FlexRay and TT-Ethernet (time-triggered Ethernet).