Norsk
Elektroteknisk
Komite

NEK EN 50716:2023

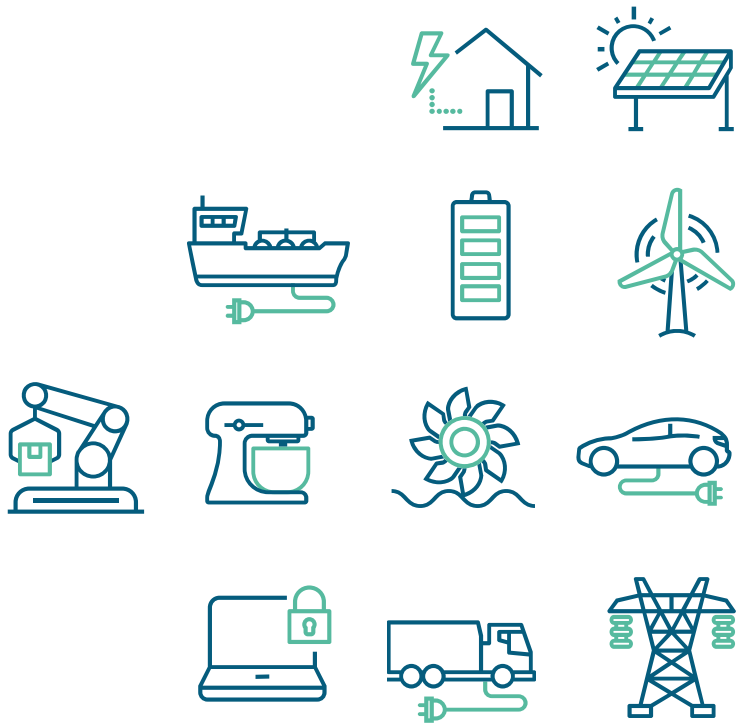
Railway Applications

Requirements for software development

Norwegian electrotechnical standard

Jernbaneapplikasjoner

Krav til programvareutvikling



Engelsk versjon

EUROPEAN STANDARD
NORME EUROPÉENNE
EUROPÄISCHE NORM

EN 50716

November 2023

ICS 35.240.60

Supersedes EN 50128:2011; EN 50128:2011/AC:2014; EN 50657:2017; EN 50128:2011/A1:2020; EN 50128:2011/A2:2020; EN 50657:2017/A1:2023

English Version

Railway Applications - Requirements for software development

Applications ferroviaires - Exigences pour le développement de logiciels

Sektorübergreifende Software-Norm für Eisenbahnen

This European Standard was approved by CENELEC on 2023-10-30. CENELEC members are bound to comply with the CEN/CENELEC Internal Regulations which stipulate the conditions for giving this European Standard the status of a national standard without any alteration.

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

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

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

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

CEN-CENELEC Management Centre: Rue de la Science 23, B-1040 Brussels

Ref. No. EN 50716:2023 E

Contents

Page

List of figures

List of tables

European foreword

This document (EN 50716:2023) has been prepared by CLC/TC 9X “Electrical and electronic applications for railways”.

The following dates are fixed:

- latest date by which this document has to be implemented at national level by publication of an identical national standard or by endorsement (dop) 2024-10-30
- latest date by which the national standards conflicting with this document have to be withdrawn (dow) 2026-10-30

This document supersedes EN 50128:2011 and EN 50657:2017 and all of their amendments and corrigenda (if any).

EN 50716:2023 includes the following significant technical changes with respect to EN 50128:2011 and EN 50657:2017:

— Better alignment with EN 50126-1:2017 and EN 50126-2:2017, including definitions, has been made;

— Clause 5: requirements have been re-written to simplify readability (while keeping existing options for organization broadly unchanged);

— Annex A has been updated to have a better alignment with lifecycle phases;

— In informative Annex C, new clause C.1 has been added with additional guidance on lifecycle models;

— In informative Annex C, new clause C.2 has been added with guidance on modelling for software development;

— Additional guidance provided for software components of different software integrity levels;

— Requirements on programming languages have been generalized.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. CENELEC shall not be held responsible for identifying any or all such patent rights;

This document is read in conjunction with EN 50126-1 “*Railway applications – The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS): Basic requirements and generic process*” [1] and EN 50126-2 “*Railway applications – The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS): Systems Approach to Safety*” [2].

For railway related fixed installations (electric traction power control and supply) EN 50562 *“Railway applications - Fixed installations - Process, protective measures and demonstration of safety for electric traction systems*” [20] is applicable.

This document has been prepared under a standardization request addressed to CENELEC by the European Commission. The Standing Committee of the EFTA States subsequently approves these requests for its Member States.

For the relationship with EU Legislation, see informative Annex ZZ, which is an integral part of this document.

Any feedback and questions on this document should be directed to the users' national committee. A complete listing of these bodies can be found on the CENELEC website.

Introduction

This document concentrates on the methods which need to be used in order to provide software which meets the demands for software integrity.

This document provides a set of requirements for the development, deployment and maintenance of any software intended for railway applications. It defines requirements concerning organizational structure, the relationship between organizations and division of responsibility involved in the development, deployment and maintenance activities. Criteria for the qualification and expertise of personnel are also provided in this document.

The key concept of this document is that of levels of software integrity. This document addresses five software integrity levels where Basic Integrity is the lowest and 4 the highest one. The higher the risk resulting from software failure, the higher the software integrity level will be.

NOTE 1 The concept of Basic Integrity used in this document was first introduced in the EN 50126 series ([1] [2]).

This document identifies techniques and measures for the five levels of software integrity. The required techniques and measures for Basic Integrity and for the safety integrity levels 1 to 4 are shown in the normative tables of Annex A. The required techniques for level 1 are the same as for level 2, and the required techniques for level 3 are the same as for level 4. This document does not give guidance on which level of software integrity is appropriate for a given risk. This decision will depend upon many factors including the nature of the application, the extent to which other systems carry out safety-related functions and social and economic factors.

It is within the scope of EN 50126-1 and EN 50126-2 to define the process of specifying the safety-related functions allocated to software.

This document specifies those measures necessary to achieve these requirements.

The EN 50126 series ([1] [2]) requires that a systematic approach is taken to:

a) identify hazards, assessing risks and arriving at decisions based on risk criteria,

b) identify the necessary risk reduction to meet the risk acceptance criteria,

c) define the overall system safety requirements for the safeguards necessary to achieve the required risk reduction,

d) select a suitable system architecture,

e) plan, monitor and control the technical and managerial activities necessary to translate the system safety requirements into a safety-related system of a validated safety integrity level.

As decomposition of the specification into a design comprising safety-related systems and components takes place, further allocation of safety integrity levels is performed. Ultimately this leads to the required software integrity levels.

The current state-of-the-art is such that neither the application of quality assurance methods (so-called fault avoiding measures and fault detecting measures) nor the application of software fault tolerant approaches can guarantee the absolute safety of the software. There is no known way to prove the absence of faults in reasonably complex safety-related software, especially the absence of specification and design faults.

The principles applied in developing high integrity software include, but are not restricted to

— top-down design methods,

— modularity,

— verification of each phase of the development lifecycle,

— verified components and component libraries,

— clear documentation and traceability,

— auditable documents,

— validation,

— assessment,

— configuration management and change control,

— appropriate consideration of organization and personnel competency issues.

At the system level, the allocation of system requirements to software functions takes place. This includes the definition of the required software integrity level for the functions. The successive functional steps in the application of this document are shown in Figure 1 and are as follows:

f) define the Software Requirements Specification and in parallel consider the software architecture. The software architecture is where the strategy is developed for the software and the software integrity level (7.2 and 7.3);

g) design, implement and test the software according to the Software Quality Assurance Plan, software integrity level and the software lifecycle (7.4 and 7.5);

h) integrate the software on the target hardware and verify functionality (7.6);

i) validate and deploy the software (7.7 and 9.1);

j) software maintenance, if required during operational life (9.2).

A number of assurance activities run across the software development. These include testing (6.1), verification (6.2), validation (6.3), assessment (6.4), quality assurance (6.5) and modification and change control (6.6).

Requirements are given for support tools (6.7) and for systems which are configured by application data (Clause 8).

Requirements are also given for the independence of roles and the competence of staff involved in software development (5.1, 5.2 and Annex B).

This document does not mandate the use of a particular software development lifecycle. However, illustrative lifecycle and documentation sets are given in 5.3, 7.1, and in C.1.

Tables have been formulated ranking various techniques/measures against the software integrity levels 1 to 4 and for Basic Integrity. The tables are in Annex A. Cross-referenced from the tables is a bibliography giving a brief description of each technique/measure with references to further sources of information. The bibliography of techniques is in Annex D.

NOTE 2 Some entries within this document have been intentionally left blank. This ensures that the numbering is, as far as reasonably practicable and where not impacting readability, unchanged with respect to EN 50128 and EN 50657. This is meant to facilitate transition and adoption of this document, where relevant.

This document does not specify the requirements for the development, implementation, maintenance and/or operation of security policies or security services needed to meet cybersecurity requirements that may be needed by the safety-related system. Cyber attacks can affect not only the operation but also the functional safety of a system. For cybersecurity, appropriate standards should be applied.

NOTE 3 ISO/IEC and CEN/CENELEC publications that address cybersecurity in depth are [3], [4], [5] and [17].

It may be necessary to balance between measures against systematic errors and measures against security threats. An example is the need for fast security updates of software arising from security threats, whereas if such software is safety related, it should be thoroughly developed, tested, validated and approved before any update.

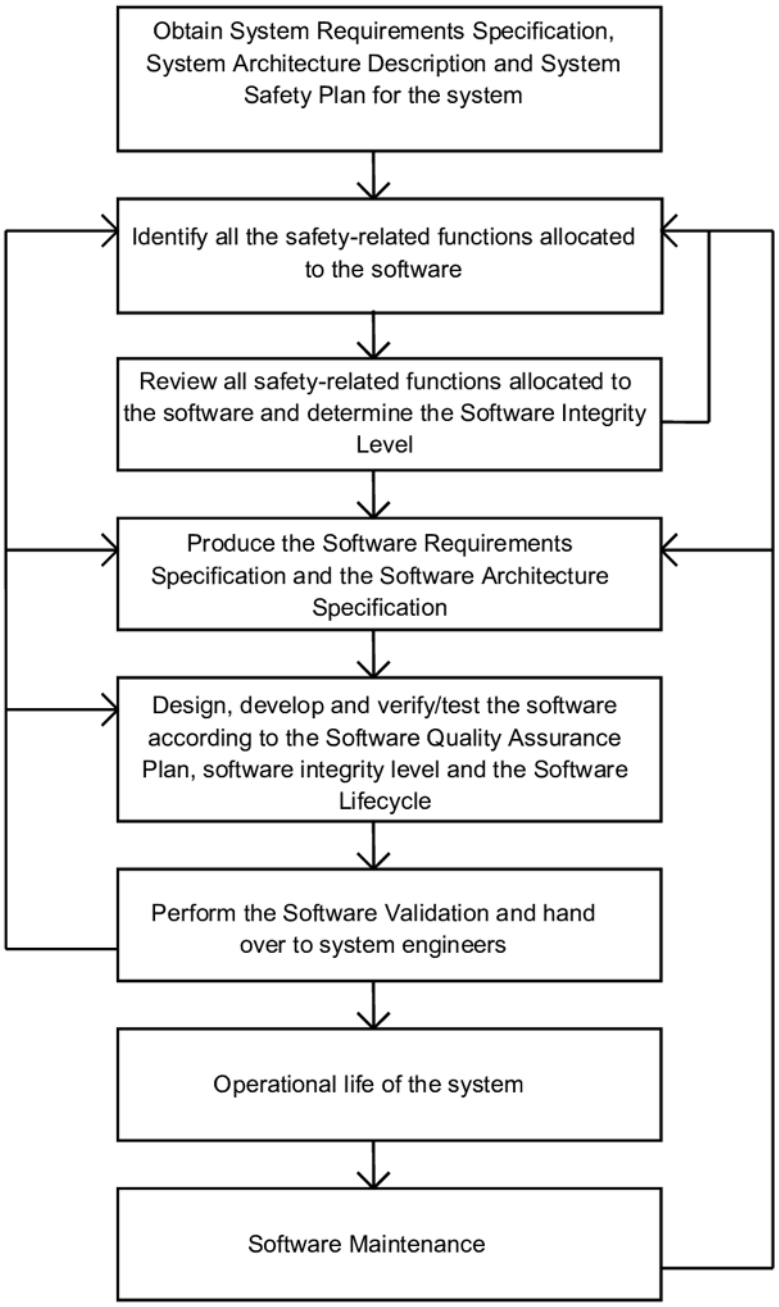


Figure 1 — Illustrative Software Route Map

1 Scope

1.1 This document specifies the process and technical requirements for the development of software for programmable electronic systems for use in:

— control, command for signalling applications,

— applications on-board of rolling stock.

This document is not intended to be applied in the area of electric traction power supply (fixed installations) or for power supply and control of conventional applications, e.g. station power supply for offices, shops. These applications are typically covered by standards for energy distribution and/or non-railway sectors and/or local legal frameworks.

1.2 This document is applicable exclusively to software and the interaction between software and the system of which it is part.

1.3 Intentionally left blank

1.4 This document applies to software as per subclause 1.1 of this document used in railway systems, including:

— application programming,

— operating systems,

— support tools,

— firmware.

Application programming comprises high level programming, low level programming and special purpose programming (for example: programmable logic controller ladder logic).

1.5 This document also addresses the use of pre-existing software (as defined in 3.1.16) and tools. Such software can be used if the specific requirements in 7.3.4.7 and 6.5.4.16 on pre-existing software and for tools in 6.7 are fulfilled.

1.6 Intentionally left blank

1.7 This document considers that modern application design often makes use of software that is suitable as a basis for various applications. Such software is then configured by application data for producing the executable software for the application.

1.8 Intentionally left blank

1.9 This document is not intended to be retrospective. It therefore applies primarily to new developments and only applies in its entirety to existing systems if these are subjected to major modifications. For minor changes, only 9.2 applies. However, application of this document during upgrades and maintenance of existing software is advisable.

1.10 For the development of User Programmable Integrated Circuits (e.g. field programmable gate arrays (FPGA) and complex programmable logic devices (CPLD)) guidance is provided in EN 50129:2018 Annex F for safety related functions and in EN 50155:2017 for non-safety related functions. Software running on softcore processors of User Programmable Integrated Circuits is within the scope of this document.

2 Normative references

There are no normative references in this document.

3 Terms, definitions and abbreviations

3.1 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

3.1.1
assessment
<of software> process of analysis to determine whether software meets the specified requirements and to form a judgement as to whether the software is fit for its intended purpose

Note 1 to entry: Safety assessment is focused on but not limited to the safety properties of software.

3.1.2
assessor
entity that carries out an assessment and, by extension, the role carried out by this entity

Note 1 to entry: This is a software specific role and should not be confused with different types of assessors specified in other standards.

[SOURCE: IEC 60050-821:2017, 821-12-05, modified – Added clarification “and, by extension, the role carried out by this entity”. The Note 1 to entry has been added for software management and organization requirements.]

3.1.3
commercial off-the-shelf software
COTS software
software defined by market-driven need, commercially available and whose fitness for purpose has been deemed acceptable by a broad spectrum of commercial users

[SOURCE: IEC 60050-821:2017, 821-12-08, modified – Preferred term and synonym have been swapped.]

3.1.4
software component
constituent part of software which has well-defined interfaces and behaviour with respect to the software architecture and design

Note 1 to entry: A software component fulfils the following criteria:

— it is designed according to “Components” (see Table A.20);

— it covers a specific subset of software requirements;

— it is clearly identified and has an independent version inside the configuration management system or is a part of a collection of components (e.g. subsystems) which have an independent version.

3.1.5
configuration manager
entity that is responsible for implementing and carrying out the processes for the configuration and change management (for documents, software and related tools) and, by extension, the role carried out by this entity

3.1.6
designer
entity that analyses and transforms specified requirements into acceptable design solutions and, by extension, the role carried out by this entity

3.1.7
entity
person, group or organization who fulfils a role

3.1.8
error
<in software> defect, mistake or inaccuracy in the development process which could result in a deviation from the intended performance or behaviour of the software

3.1.9
failure
<of an item> loss of ability to perform as required

Note 1 to entry: “Failure” is an event, as distinguished from “fault”, which is a state.

[SOURCE: IEC 60050-821:2017, 821-11-19, modified – The notes 1 and 2 have been omitted. A new note 1 to entry has been added]

3.1.10
fault
abnormal condition that could lead to an error in a system

Note 1 to entry: Any fault in software is systematic, while a hardware fault can also be random

[SOURCE: IEC 60050-821:2017, 821-11-20, modified – The note 1 to entry has been modified.]

3.1.11
firmware
software stored in read-only memory or in semi-permanent storage such as flash memory, in a way that is functionally independent of application software

3.1.12
generic software
software which can be used for a variety of installations purely by the provision of application-specific data

[SOURCE: IEC 60050-821:2017, 821-12-26, modified – “or algorithms, or both” has been removed.]

3.1.13
implementer
<of software> entity that transforms specified designs into their realization and, by extension, the role carried out by this entity

[SOURCE: IEC 60050-821:2017, 821-12-30, modified – “physical realization” has been replaced with “realization and, by extension, the role carried out by this entity”.]

3.1.14
integration
<of software> process of assembling software items or software with hardware items, according to the architectural and design specification

3.1.15
maintainability
<of software> capability of the software to be modified; to correct faults, improve performance or other attributes, or adapt it to a different environment

[SOURCE: IEC 60050-821:2017, 821-12-62, modified – “software” has been moved to the specific use. “a software” has been replaced with “the software”. “to adapt” has been replaced with “adapt”.]

3.1.16
pre-existing software
software developed prior to the application currently in question

Note 1 to entry: This includes commercial off-the-shelf software, open-source software and software previously developed but not in accordance with this document or any version of EN 50128 or EN 50657.

[SOURCE: EN 50126-1:2017, 3.43, modified – “all” has been removed. The end of the definition “is classed as pre-existing software including commercial off-the-shelf software, open-source software and software previously developed but not in accordance with this European Standard” has been moved to the Note 1 to entry and adapted to software context.]

3.1.17
programmable logic controller
solid-state control system which has a user programmable memory for storage of instructions to implement specific functions

[SOURCE: IEC 60050-821:2017, 821-12-41]

3.1.18
project management
administrative and/or technical conduct of a project, including RAMS aspects

[SOURCE: EN 50126-1:2017, 3.46]

3.1.19
project manager
entity that carries out project management and, by extension, the role carried out by this entity

3.1.20
robustness
ability of an item to detect and handle abnormal situations

3.1.21
requirements manager
entity that carries out requirements management and, by extension, the role carried out by this entity

3.1.22
requirements management
process of eliciting, documenting, analysing, prioritizing and agreeing on requirements and then controlling change and communicating to relevant stakeholders

3.1.23
risk
<for railway RAMS> combination of expected frequency of loss and the expected degree of severity of that loss

[SOURCE: EN 50126-1:2017, 3.57]

3.1.24
safety
freedom from unacceptable risk

[SOURCE: IEC 60050-903:2013, 903-01-19]

3.1.25
safety authority
body responsible for delivering the authorization for the operation of the safety-related system

[SOURCE: IEC 60050-821:2017, 821-12-52]

3.1.26
safety integrity level
one of a number of defined discrete levels for specifying the safety integrity requirements for safety-related functions to be allocated to the safety-related systems

Note 1 to entry: Safety Integrity Level with the highest figure has the highest level of safety integrity.

Note 2 to entry: It is not possible to allocate a Safety Integrity Level to safety-related processes or other measures.

[SOURCE: EN 50126-1:2017, 3.70]

3.1.27
safety-related
carries responsibility for safety

[SOURCE: IEC 60050-821:2017, 821-01-73]

3.1.28
safety-related software
software which performs safety-related functions

Note 1 to entry: Software is called safety-related if at least one of its properties is used in the safety argument for the system in which it is applied. These properties can be of functional or non-functional nature.

[SOURCE: IEC 60050-821:2017, 821-12-60, modified – “safety functions” has been replaced with “safety-related functions”. The note 1 to entry has been added.]

3.1.29
software
programs, procedures, rules, documentation and data of an information processing system

EXAMPLE Requirements and design specifications; source code listings, check lists and comments; “Help” text and messages for display at the computer/human interface; instructions for installation and operation; and support guides for hardware and software maintenance.

Note 1 to entry: Software is an intellectual creation that is independent of the medium upon which it is recorded.

Note 2 to entry: Software requires hardware devices to execute programs, and to store and transmit data.

Note 3 to entry: Types of software include firmware, system software, and application software.

[SOURCE: IEC 60050-192:2015, 192-01-07]

3.1.30
software baseline
complete and consistent set of source code, executable files, configuration files, installation scripts and documentation that are needed for a software release

Note 1 to entry: Information about compilers, operating systems, pre-existing software and dependent tools is stored as part of the software baseline. This will enable the organization to reproduce defined versions and be the input for future releases at enhancements or upgrades in the maintenance phase.

3.1.31
software deployment
transferring, installing and activating a deliverable software baseline

3.1.32
software integrity level
classification which determines the techniques and measures that have to be applied to software

Note 1 to entry: This includes Basic Integrity and the safety integrity levels 1 to 4.

3.1.33
software lifecycle
activities occurring during a period of time that starts when software is conceived and ends when the software is decommissioned

3.1.34
software maintenance
modification for the purposes of software fault removal, adaptation to a new environment, or improvement of performance

[SOURCE: IEC 60050-192:2015, 192-06-15, modified – The notes to entry have been omitted.]

3.1.35
tester
entity that carries out testing and, by extension, the role carried out by this entity

3.1.36
testing
<of software> process of executing software under controlled conditions as to ascertain its behaviour and performance compared to the corresponding requirements specification

3.1.37
tool class T1
tool which is not used to generate outputs which can directly or indirectly contribute to the executable code (including data) of the software

Note 1 to entry: T1 examples include: a text editor or a requirement or design support tool with no automatic code generation capabilities.

3.1.38
tool class T2
tool which is used to support the test or verification of the design or executable code (including data), where errors in the tool can fail to reveal defects but cannot directly create errors in the executable software (including data)

Note 1 to entry: T2 examples include: a test harness generator; a test coverage measurement tool; a static analysis tool

3.1.39
tool class T3
tool which is used to generate outputs which can directly or indirectly contribute to the executable code (including data) of the system

Note 1 to entry: T3 examples include: a source code compiler, a data compiler, a tool to change set-points during system operation; a compiler that incorporates an executable run-time package into the executable code

3.1.40
traceability
degree to which a relationship can be established between two or more products of a development process, especially those having a predecessor/successor or master/subordinate relationship to one another

3.1.41
validation
confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled

Note 1 to entry: The term “validated” is used to designate the corresponding status.

Note 2 to entry: The use conditions for validation can be real or simulated.

Note 3 to entry: In design and development, validation concerns the process of examining an item to determine conformity with user needs.

Note 4 to entry: Validation is normally performed during the final stage of development, under defined operating conditions, although it can also be performed in earlier stages.

Note 5 to entry: Multiple validations can be carried out if there are different intended uses.

[SOURCE: IEC 60050-192:2015, 192-01-18]

3.1.42
validator
entity that is responsible for the validation and, by extension, the role carried out by this entity

[SOURCE: IEC 60050-821:2017, 821-12-73, modified – “and, by extension, the role carried out by this entity” has been added.]

3.1.43
verification
confirmation, through the provision of objective evidence, that specified requirements have been fulfilled

Note 1 to entry: The term “verified” is used to designate the corresponding status.

Note 2 to entry: Design verification is the application of tests and appraisals to assess conformity of a design to the specified requirement.

Note 3 to entry: Verification is conducted at various lifecycle phases of development, examining the system and its constituents to determine conformity to the requirements specified at the beginning of that lifecycle phase.

[SOURCE: IEC 60050-192:2015, 192-01-17, modified – The note 3 to entry has been modified]

3.1.44
verifier
entity that is responsible for one or more verification activities and, by extension, the role carried out by this entity

[SOURCE: IEC 60050-821:2017, 821-12-75, modified – “and, by extension, the role carried out by this entity” has been added.]

3.2 Abbreviations

For the purposes of this document, the following abbreviations apply.

ASR	Assessor
COTS	Commercial off-the-shelf
DES	Designer
HR	Highly Recommended
IMP	Implementer
M	Mandatory
NR	Not Recommended
PM	Project Manager
R	Recommended
RAMS	Reliability, Availability, Maintainability and Safety
RQM	Requirements Manager
SIL	Safety Integrity Level
TST	Tester
V&V	Verification and Validation
VAL	Validator
VER	Verifier

4 Software integrity levels conformance

4.1 The allocation of system functions to software, as well as software interfaces, shall be identified in the system documentation. The system in which the software is embedded shall be fully defined with respect to the following:

— functions and interfaces;

— application conditions;

— configuration or architecture of the system;

— hazards to be controlled;

— safety integrity requirements;

— apportionment of requirements and allocation of SIL or Basic Integrity to software and hardware;

— timing constraints.

NOTE 1 The allocation of safety integrity requirements can lead to different integrity levels for well-separated software and hardware parts of a subsystem. This allocation depends on the contribution of the software and hardware parts of the subsystem to the safety-related functions and on the mechanisms for the failure mitigation including the separation of functions with different integrity levels (see EN 50126-2:2017 [2]).

NOTE 2 This document uses the term System Requirements Specification to address the complete set of system requirements including the safety requirements.

4.2 The software integrity shall be specified as one of five levels, from Basic Integrity (the lowest) to SIL 4 (the highest).

4.3 The required software integrity level shall be decided and assessed at system level, on the basis of the system safety integrity level and the level of risk associated with the use of the software in the system.

4.4 At least the Basic Integrity requirements of this document shall be fulfilled for the software part of functions that have a safety impact below SIL 1. For non-safety related functions a software quality assurance process shall be used. This may be achieved by applying the Basic Integrity requirements from this software standard, ISO/IEC/IEEE 90003 [19], or alternative codes of practice.

4.5 To conform to this document it shall be shown that each of the requirements has been satisfied to the software integrity level defined and therefore the objective of the subclause in question has been met.

4.6 Where a requirement is qualified by the words “to the extent required by the software integrity level”, this indicates that a range of techniques and measures shall be used to satisfy that requirement.

4.7 Where 4.6 is applied, tables from normative Annex A shall be used to assist in the selection of techniques and measures appropriate to the software integrity level. The selection shall be documented in the Software Quality Assurance Plan or in another document referenced by the Software Quality Assurance Plan. Guidance to these techniques is given in the informative Annex D.

4.8 If a technique or measure which is ranked as highly recommended (HR) in the tables is not used, then the rationale for using alternative techniques shall be detailed and recorded either in the Software Quality Assurance Plan or in another document referenced by the Software Quality Assurance Plan. This is not necessary if an approved combination of techniques given in the corresponding table is used. The selected techniques shall be demonstrated to have been applied correctly.

4.9 If a technique or measure is proposed to be used that is not contained in the tables then its effectiveness and suitability in meeting the particular requirement and overall objective of the subclause shall be justified and recorded in either the Software Quality Assurance Plan or in another document referenced by the Software Quality Assurance Plan.

4.10 Compliance with the requirements of a particular subclause and their respective techniques and measures detailed in the tables shall be verified by the inspection of documents. Where appropriate, other objective evidence, auditing and the witnessing of tests shall also be taken into account.

4.11 Software developed according to any version of EN 50128 or EN 50657 may be considered as compliant and not subject to the requirements on pre-existing software.

NOTE This ensures continuity in the application of these standards, i.e. software that was developed in accordance with the previous software standards can still be re-used for new projects.

5 Software management and organization

5.1 Organization and independence of roles

5.1.1 Objective

To ensure that all personnel who have responsibilities for the software are organized so as to provide sufficient independence and empowerment to achieve the following objectives:

a) To reduce the probability of people in different roles suffering from the same misconceptions or making the same mistakes by ensuring independence between roles.

b) To ensure that people in roles which involve making judgements about the acceptability of a product or process from the point of view of safety should not be influenced by pressure from their peers or supervisors, or by considerations of commercial gain.

5.1.2 Requirements

5.1.2.1 As a minimum, the supplier shall implement requirements dealing with the organization and management of the personnel and responsibilities.

These requirements can be met with the related parts of EN ISO 9001 [18].

5.1.2.2 Intentionally left blank

5.1.2.3 The personnel assigned to the roles involved in the development or maintenance of the software shall be named and recorded.

5.1.2.4 An Assessor shall be appointed for SIL 1 to SIL 4. The Assessor may be appointed by the supplier or by the customer.

5.1.2.5 The Assessor may be part of any stakeholder organization (e.g. supplier, customer or third-party organisations).

5.1.2.6 The Assessor shall be independent from the project team and shall be a different entity, organizationally independent, from those undertaking other roles in the project.

NOTE The project team is the set of roles comprising typically of: project manager, configuration manager, requirements manager, designer, implementer, tester, verifier and validator. The Assessor is considered as a role external to the project team.

5.1.2.7 The Assessor shall be given authority to perform the assessment of the software.

5.1.2.8 The Validator shall give agreement or disagreement for the software release.

5.1.2.9 Throughout the Software Lifecycle, the assignment of roles to persons shall be in accordance with 5.1.2.10 to 5.1.2.12 (see Figure 2 for an illustration of the organizational structure).

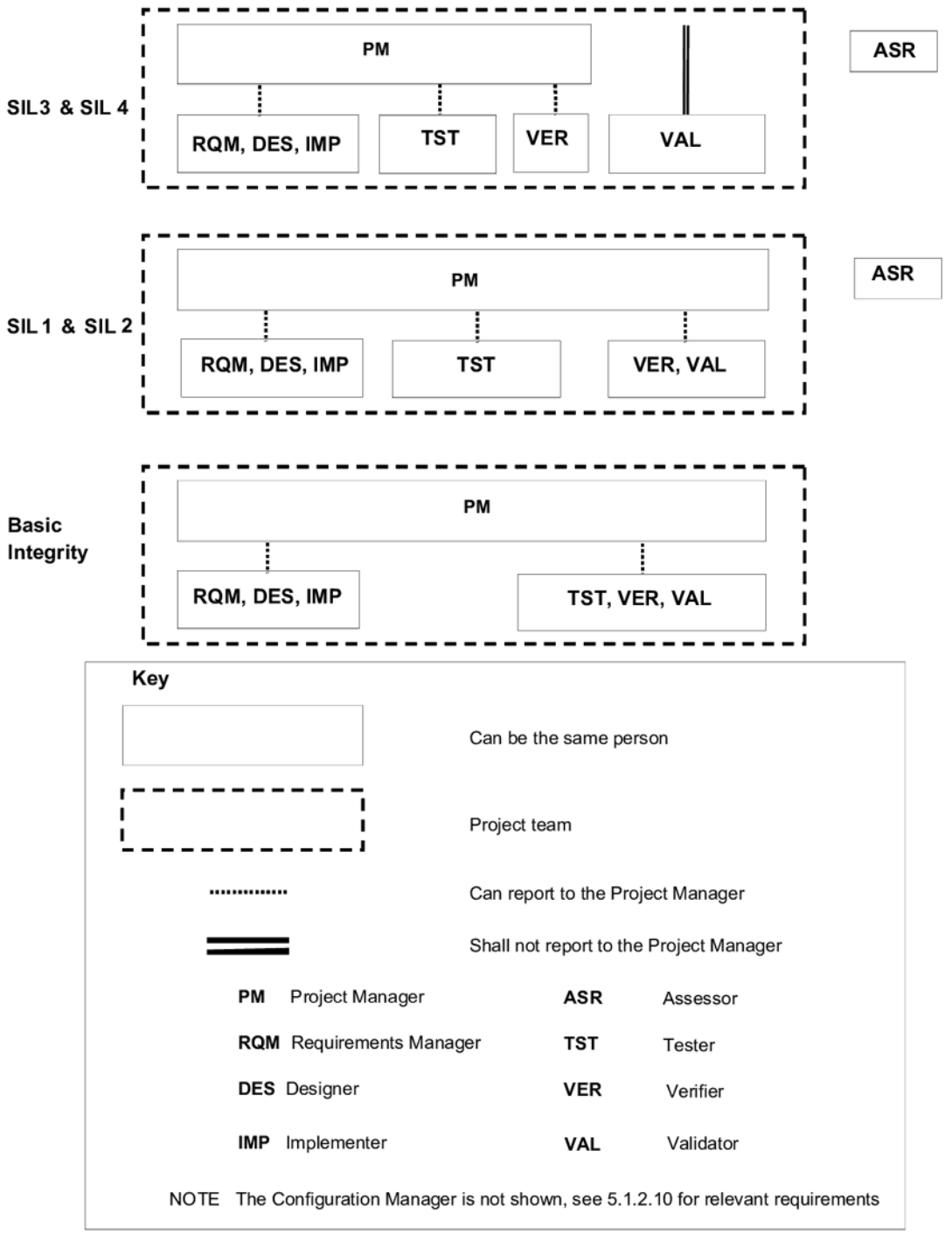


Figure 2 — Illustration of the organizational structure

5.1.2.10 The following provisions apply for Basic Integrity, SIL 1, SIL 2, SIL 3 and SIL 4:

a) A role may be shared by more than one person and, when not explicitly forbidden, a person may fulfil more than one role.

NOTE Role names (e.g. Requirements Manager, Designer, Implementer, Tester, Verifier, Validator) indicate generically one or more members of the related entity.

b) A Validator or Verifier shall not be a Requirements Manager, Designer or Implementer.

c) A Requirements Manager, Designer or Implementer shall neither report to, nor manage or supervise, a Validator and vice versa.

d) A Validator shall not be the Project Manager.

e) A Configuration Manager shall not be a Validator.

f) The roles of the Verifier and the Validator shall be assigned to entities defined at the project level and shall remain unchanged throughout the development project. Any change of personnel shall not jeopardize the activities and shall be documented and justified.

NOTE The intention of this requirement is to ensure continuous and consistent verification and validation activities throughout the project. Obviously, a change in personnel cannot be fully excluded.

5.1.2.11 The following provisions apply for SIL 1, SIL 2, SIL 3 and SIL 4:

a) If a Validator performs Tester activities, another Validator shall review related test activities and documents.

b) If a Verifier performs Tester activities, another Verifier or Validator shall review related test activities and documents.

c) A Requirements Manager, Designer or Implementer for a software component shall not be a Tester for the same software component. However, a Requirements Manager, Designer or Implementer for a software component may be Tester for a different software component. This principle may also be applied at higher system levels (i.e. for Software Integration Testing, Software/Hardware Integration Testing, and Overall Software Testing), if the necessary independence between design/implementation and testing is ensured and demonstrated.

NOTE Tests can be specified and executed by either a Tester or other parties such as the Requirements Manager, Designer or Implementer if the provisions in 6.1.4.1 are complied with.

5.1.2.12 The following provisions apply for SIL 3 and SIL 4:

a) A Validator shall not report to the Project Manager.

NOTE The objective is that Validators' decisions are not influenced by the Project Manager. However, Validators inform the Project Manager about their decisions.

b) If a Validator performs Verifier activities, another Validator shall review related verification activities and documents.

5.2 Personnel competence and responsibilities

5.2.1 Objectives

To ensure that all personnel who have responsibilities for the software are competent, empowered and capable of fulfilling their responsibilities by demonstrating the ability to perform relevant tasks correctly, efficiently and consistently to a high quality and under varying conditions.

5.2.2 Requirements

5.2.2.1 The key competencies required for each role in the software development are defined in Annex B. If additional experience, capabilities or qualifications are required for a role in the software lifecycle, these shall be defined in the Software Quality Assurance Plan.

5.2.2.2 Documented evidence of personnel competence, including technical knowledge, qualifications, relevant experience and appropriate training, shall be maintained by the supplier's organization in order to demonstrate appropriate safety organization.

5.2.2.3 The organization shall maintain procedures to manage the competence of personnel to suit appropriate roles in accordance to applicable quality standards.

5.2.2.4 Once it has been proved to the satisfaction of an Assessor or by a certification that competence has been demonstrated for all personnel appointed in various roles, each individual will need to show continuous maintenance and, if needed, development of competence. This could be demonstrated by keeping a logbook showing the activity is being regularly carried out correctly, and that additional training is being undertaken in accordance with the Quality Assurance System. EN ISO 9001:2015 and ISO/IEC/IEEE 90003:2018, 7.2 “Competence” may be used.

5.2.2.5 Responsibilities shall be compliant with the requirements defined in Annex B (Tables B.1 to B.9).

5.3 Lifecycle issues and documentation

5.3.1 Objectives

5.3.1.1 To structure the development of the software into defined phases and activities.

5.3.1.2 To record all information pertinent to the software throughout the lifecycle of the software.

5.3.2 Requirements

5.3.2.1 A lifecycle model for the development of software shall be selected. It shall be detailed in the Software Quality Assurance Plan in accordance with 6.5.

5.3.2.2 The lifecycle model shall take into account the possibility of iterations within and between phases.

NOTE 1 Further guidance regarding lifecycle models, in particular considering iterative development, is provided in C.1.

NOTE 2 Guidance on usage of modelling within the development of software is provided in C.2.

5.3.2.3 Quality Assurance procedures shall run in parallel with lifecycle activities and use the same terminology.

5.3.2.4 The Software Quality Assurance Plan, Software Verification Plan, Software Validation Plan and Software Configuration Management Plan shall be drawn up at the start of the project and maintained throughout the software development lifecycle.

5.3.2.5 All activities to be performed during a phase shall be defined and planned prior to the commencement of the phase.

5.3.2.6 All documents shall be structured to allow continued expansion in parallel with the development process.

5.3.2.7 For each document, traceability shall be provided in terms of a unique reference number and a defined and documented relationship with other documents.

5.3.2.8 Each term, acronym or abbreviation shall have the same meaning in every document. If, for historical reasons, this is not possible, the different meanings shall be listed and the references given.

5.3.2.9 Except for documents of the pre-existing software (see 7.3.4.7), each document shall be written according to the following rules:

— it shall contain or implement all applicable conditions and requirements of the preceding document with which it has a hierarchical relationship;

— it shall not contradict the preceding document.

5.3.2.10 Each item or concept shall be referred to by the same name or description in every document.

5.3.2.11 The contents of all documents shall be recorded in a form appropriate for manipulation, processing and storage.

5.3.2.12 When documents which are produced by independent roles are combined into a single document, the relation to the parts produced by any independent role shall be traced within the document.

5.3.2.13 Documents may be combined or divided in accordance with 5.3.2.12. Some development steps may be combined, divided or, when justified, eliminated, at the discretion of the Project Manager and with the agreement of the Validator. Documents can be generated from databases or modelling tools. When generating documents from databases or modelling tools the traceability between document version and database version shall be preserved.

NOTE If the database is no longer available the generated documents are directly used as the basis for software maintenance.

5.3.2.14 It shall be established that any lifecycle or documentation structure adopted meets all the objectives and requirements of this document.

6 Software assurance

6.1 Software testing

6.1.1 Objective

The objective of software testing is to ascertain the behaviour or performance of software against the corresponding test specification to the extent achievable by the selected test coverage.

6.1.2 Input documents

1) All necessary System, Hardware and Software Documentation as specified in the Software Verification Plan.

6.1.3 Output documents

1) Overall Software Test Specification

2) Overall Software Test Report

3) Software Integration Test Specification

4) Software Integration Test Report

5) Software/Hardware Integration Test Specification

6) Software/Hardware Integration Test Report

7) Software Component Test Specification

8) Software Component Test Report

6.1.4 Requirements

6.1.4.1 Tests specified and/or executed by other parties such as the Requirements Manager, Designer or Implementer, if fully documented and complying with the following requirements, may be accepted by the Verifier and/or the Validator. The rationale for using test evidence provided by those other parties shall be documented in the verification and/or validation reports.

6.1.4.2 Measurement equipment used for testing shall be calibrated appropriately. Any tools, hardware or software, used for testing shall be shown to be suitable for the purpose.

6.1.4.3 Software testing shall be documented by a Test Specification and a Test Report, as defined in the following.

6.1.4.4 Each Test Specification shall document the following:

a) test objectives;

b) test cases, test data and expected results;

c) types of tests to be performed;

d) test environment, tools, configuration and programs;

e) test criteria on which the completion of the test will be judged;

f) the criteria and degree of test coverage to be achieved;

g) the roles and responsibilities of the personnel involved in the test process;

h) the requirements which are covered by the test specification;

i) the selection and utilization of the software test equipment.

6.1.4.5 A Test Report shall be produced as follows:

a) the Test Report shall mention the Tester names, state the test results and whether the test objectives and test criteria of the Test Specification have been met. Failures shall be documented and summarized;

b) test cases and their results shall be recorded, preferably in a machine-readable form for subsequent analysis;

c) tests shall be repeatable and, if practicable, be performed by automatic means;

d) test scripts for automatic test execution shall be verified against the Test Specification;

e) the identity and configuration of all items involved (hardware used, software used, equipment used, equipment calibration, as well as version information of the test specification) shall be documented;

f) an evaluation of the test coverage and test completion shall be provided and any deviations noted.

6.2 Software verification

6.2.1 Objective

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 fulfil the requirements and plans with respect to completeness, correctness and consistency. These activities are managed by the Verifier.

6.2.2 Input documents

1) All necessary System, Hardware and Software Documentation.

6.2.3 Output documents

1) Software Verification Plan

2) Software Verification Report(s)

3) Software Planning Verification Report

6.2.4 Requirements

6.2.4.1 Verification shall be documented by at least a Software Verification Plan and one or more (process-related) Verification Reports.

6.2.4.2 A Software Verification Plan shall be written, under the responsibility of the Verifier, on the basis of the necessary documentation.

Requirements from 6.2.4.3 to 6.2.4.9 refer to the Software Verification Plan.

6.2.4.3 The Software Verification Plan shall describe the activities to be performed to ensure proper verification and that particular design or other verification needs are suitably provided for.

6.2.4.4 During development (and depending upon the size of the system) the plan may be sub-divided into a number of child documents, as the detailed needs of verification become clearer.

6.2.4.5 The Software Verification Plan shall document all the criteria, techniques and tools to be used in the verification process. The Software Verification Plan shall include techniques and measures chosen from Table A.5, Table A.6 and Table A.7. The selected combination shall be justified as a set satisfying 4.8, 4.9 and 4.10.

6.2.4.6 The Software Verification Plan shall describe the activities to be performed to ensure correctness and consistency with respect to the input to that phase. These include (but are not necessarily limited to) reviewing and testing.

6.2.4.7 In each development phase it shall be shown that the functional, performance and safety requirements are met.

6.2.4.8 The results of each verification shall be retained in a format defined or referenced in the Software Verification Plan.

6.2.4.9 The Software Verification Plan shall address the following:

a) the selection of verification strategies and techniques (to avoid undue complexity in the assessment of the verification and testing, preference shall be given to the selection of techniques which are in themselves readily analysable);

b) the selection and documentation of verification activities;

c) the evaluation of verification results gained;

d) the evaluation of the safety and robustness requirements;

e) the roles and responsibilities of the personnel involved in the verification process;

f) the degree of the functional based test coverage required to be achieved;

g) the structure and content of each verification step, especially for the software requirements verification (7.2.4.22), software architecture and design verification (7.3.4.41, 7.3.4.42), software components verification (7.4.4.13), software source code verification (7.5.4.10) and software integration verification (7.6.4.13) in a way that facilitates review against the Software Verification Plan.

6.2.4.10 A Software Planning Verification Report shall be written, under the responsibility of the Verifier, on the basis of the input documents from 6.2.2.

The requirement in 6.2.4.11 refers to the Software Planning Verification Report.

6.2.4.11 Once the software plans have been established (Software Quality Assurance Plan, Software Configuration Management Plan, Software Verification Plan, Software Validation Plan, and Software Maintenance Plan) verification shall confirm

a) that the software plans meet the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17 as well as the specific requirements in 6.2.4.3 to 6.2.4.9, 6.5.4.4 to 6.5.4.6, 6.3.4.4 to 6.3.4.6 and 9.2.4.6,

b) the internal consistency of the software plans,

c) the coherency of the software plans.

The results shall be recorded in a Software Planning Verification Report.

6.2.4.12 Any Software Verification Reports shall be written, under the responsibility of the Verifier, on the basis of the input documents. These reports can be partitioned for clarity and convenience, and shall follow the Software Verification Plan. The requirement in 6.2.4.13 refers to the Software Verification Reports.

6.2.4.13 Each Software Verification Report shall document the following:

a) the identity and configuration of the items verified, as well as the Verifier names;

b) items which do not conform to the specifications;

c) components, data, structures and algorithms poorly adapted to the problem;

d) detected errors or deficiencies;

e) the fulfilment of, or deviation from, the Software Verification Plan (in the event of deviation the Verification Report shall explain whether the deviation is critical or not);

f) the correct use of the chosen techniques and measures;

g) assumptions if any;

h) a summary of the verification results.

6.3 Software validation

6.3.1 Objective

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 integrity level, fulfils 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 software integrity level, and to evaluate the safety criticality of all anomalies and non-conformities based on the results of reviews, analyses and tests.

6.3.2 Input documents

All system, hardware and software documentation as specified in this document.

6.3.3 Output documents

1) Software Validation Plan

2) Software Validation Report

6.3.4 Requirements

6.3.4.1 The Software Validation activities shall be developed and performed, with their results evaluated, by a Validator with an appropriate level of independence as defined in 5.1.

6.3.4.2 Validation shall be documented with, at least, a Software Validation Plan and a Software Validation Report, as defined in the following.

6.3.4.3 A Software Validation Plan shall be written, under the responsibility of the Validator, on the basis of the input documents.

Requirements from 6.3.4.4 to 6.3.4.6 refer to the Software Validation Plan.

6.3.4.4 The Software Validation Plan shall include a summary justifying the validation strategy chosen. The justification shall include consideration, according to the required software integrity level, of

a) manual or automated techniques or both,

b) static or dynamic techniques or both,

c) analytical or statistical techniques or both,

d) testing in a real or simulated environment or both,

e) justification for the involvement in any other role of any person fulfilling a validator role.

6.3.4.5 The Software Validation Plan shall identify the steps necessary to demonstrate the adequacy of any Software Specification in fulfilling the requirements set out in the System Requirements Specification.

6.3.4.6 The Software Validation Plan shall identify the steps necessary to demonstrate the adequacy of the Overall Software Test Specification as a test against the Software Requirements Specification.

6.3.4.7 A Software Validation Report shall be written, under the responsibility of the Validator, on the basis of the input documents.

Requirements from 6.3.4.8 to 6.3.4.11 refer to the Software Validation Report.

6.3.4.8 The results of the validation shall be documented in the Software Validation Report.

6.3.4.9 The Validator shall check that the verification process is complete.

6.3.4.10 The Software Validation Report shall fully state the software baseline that has been validated.

6.3.4.11 The Validation Report shall clearly identify any known deficiencies in the software and the impact these may have on the use of the software.

6.3.4.12 The Validator shall be empowered to require or perform additional audits, reviews, analyses and tests.

6.3.4.13 The software shall only be released for operation after authorization by the Validator.

6.3.4.14 Simulation and modelling may be used to supplement the validation process.

6.4 Software assessment

6.4.1 Objective

6.4.1.1 To evaluate the extent to which the lifecycle processes and their outputs are such that the software meets the specified requirements for SIL 1 to SIL 4 and to form a judgement as to whether the software is fit for its intended purpose.

6.4.1.2 For Basic Integrity, requirements of this standard shall be fulfilled but no assessment will be required.

NOTE For Basic Integrity, the requirements of subclauses 6.4.2 to 6.4.4 do not need to be fulfilled.

6.4.2 Input documents

1) System Requirements Specification (see 4.1, NOTE 2)

2) Software Requirements Specification

3) All other documents necessary to carry out the assessment process.

6.4.3 Output documents

1) Software Assessment Plan

2) Software Assessment Report

6.4.4 Requirements

6.4.4.1 The assessment of the software shall be carried out by an Assessor who is independent as described in 5.1.2.6 and 5.1.2.7.

6.4.4.2 Software with a Software Assessment Report from another Assessor does not have to be an object of a new assessment. The Assessor shall check that the software is fit for its intended use within the intended environment, and that the former assessment stated the software has achieved a safety integrity level at least equal to the required level.

6.4.4.3 The Assessor shall have access to all project-related documentation throughout the development process.

6.4.4.4 A Software Assessment Plan shall be written, under the responsibility of the Assessor, on the basis of the input documents from 6.4.2. Where appropriate, an existing documented generic Software Assessment Plan or procedure may be used. The requirement in 6.4.4.5 refers to the Software Assessment Plan.

6.4.4.5 The Software Assessment Plan shall address the following topics:

a) scope of the assessment;

b) activities throughout the assessment process and their sequential link to engineering activities;

c) documents to be taken into consideration;

d) statements on pass/fail criteria and the way to deal with non-conformance cases;

e) requirements with regard to content and form of the Software Assessment Report.

6.4.4.6 The Assessor shall assess that the software of the system is fit for its intended purpose and responds correctly to safety issues derived from the System Requirements Specification.

6.4.4.7 The Assessor shall assess if an appropriate set of techniques from Annex A, suitable for the intended lifecycle phase, has been selected and applied in accordance to the required software integrity level.

Moreover, the Assessor shall consider the extent to which each technique from Annex A is applied, i.e. whether it is applied to all or to only part of the software, and also look for evidence that it is properly applied.

6.4.4.8 The Assessor shall assess the configuration and change management system and the evidences on its use and application.

6.4.4.9 The Assessor shall evaluate the competency of the project staff according to Annex B and shall assess the organization for the software development according to 5.1.

6.4.4.10 For any software addressing safety-related application conditions, the Assessor shall check for noted deviations, non-compliances to requirements and recorded non-conformities if these have

an impact on safety, and make a judgment whether the justification from the project is acceptable. The result shall be stated in the Software Assessment Report.

NOTE Safety-related application conditions can be requirements for software development (including development of application data).

6.4.4.11 The Assessor shall assess the verification and validation activities and the supporting evidence.

6.4.4.12 The Assessor shall agree the scope and contents of the Software Validation Plan. This agreement shall also make a statement concerning the presence of the Assessor during testing.

6.4.4.13 The Assessor may carry out audits and inspections (e.g. witnessing tests) throughout the entire development process. The Assessor may ask for additional verification and validation work.

NOTE It is of advantage to involve the Assessor early in the project.

6.4.4.14 A Software Assessment Report shall be written under the responsibility of the Assessor. Requirements from 6.4.4.15 to 6.4.4.17 refer to the Software Assessment Report.

6.4.4.15 The Software Assessment Report shall meet the requirements of the Software Assessment Plan and provide a conclusion and recommendations.

6.4.4.16 The Assessor shall record his/her activities as a consistent base for the Software Assessment Report. These shall be summarized in the Software Assessment Report.

6.4.4.17 The Assessor shall identify and evaluate any non-conformity with the requirements of this document and judge the impact on the final result. These non-conformities and their judgments shall be listed in the Software Assessment Report.

6.5 Software quality assurance

6.5.1 Objectives

6.5.1.1 To identify, monitor and control all those activities, both technical and managerial, which are necessary to ensure that the software achieves the quality required. This is necessary to provide the required qualitative defence against systematic faults and to ensure that an audit trail can be established to allow verification and validation activities to be undertaken effectively.

6.5.1.2 To provide evidence that the above activities have been carried out.

6.5.2 Input documents

All the documents available at each stage of the lifecycle.

6.5.3 Output documents

1) Software Quality Assurance Plan

2) Software Configuration Management Plan, if not available at system level

6.5.4 Requirements

6.5.4.1 All the plans shall be issued at the beginning of the project and updated during the lifecycle.

6.5.4.2 The organizations taking part in the software development shall implement and use a Quality Assurance System, to support the requirements of this document.

6.5.4.3 A Software Quality Assurance Plan shall be written on the basis of the input documents from 6.5.2.

The requirements from 6.5.4.4 to 6.5.4.6 refer to the Software Quality Assurance Plan.

6.5.4.4 A Software Quality Assurance Plan shall be written and shall be specific to the project. It shall implement the requirements in 6.5.4.5.

6.5.4.5 As a minimum, the following items shall be specified or referenced in the Software Quality Assurance Plan.

a) Definition of the lifecycle model:

   1) activities and elementary tasks consistent with the plans, e.g. Safety Plan, that have been established at the System level;

   2) entry and exit criteria of each activity;

   3) inputs and outputs of each activity;

   4) major quality activities;

   5) the entity responsible for each activity.

b) Documentation structure.

c) Documentation control:

   1) roles involved for writing, checking and approval;

   2) scope of distribution;

   3) archiving.

d) Tracking and tracing of deviations.

e) Techniques and measures chosen from Table A.2, Table A.3, Table A.4 and Table A.9. The selected combination shall be justified as a set satisfying 4.8, 4.9 and 4.10.

Some of the Software Quality Assurance Plan required information may be contained in other documents, such as a separate Software Configuration Management Plan, a Software Maintenance Plan, a Software Verification plan and a Software Validation Plan. The subclauses of the Software Quality Assurance Plan shall reference the documents in which the information is contained. In any case the content of each subclause of the Software Quality Assurance Plan shall be specified either directly or by reference to another document.

The referenced documents shall be reviewed in order to ensure they provide all the required information and that they fully address the requirements of this standard.

6.5.4.6 Quality assurance activities, actions, documents, etc. required by all normative subclauses of this document shall be specified or referenced in the Software Quality Assurance Plan and tailored to the specific project.

6.5.4.7 Intentionally left blank

6.5.4.8 Intentionally left blank

6.5.4.9 Each planning document shall have a paragraph specifying details about its own updating throughout the project: frequency, responsibility, method.

6.5.4.10 Each software document and deliverable shall be placed under configuration control from the time of its first release.

6.5.4.11 Changes to all items under Configuration Management Control shall be authorized and recorded.

6.5.4.12 In addition to software development, the Configuration Management System shall also cover the software development environment used during the full lifecycle.

This extension, necessary for the reproducibility of the development and for the maintenance activities, shall include all the tools, translators, data and test files, parameterization files, and supporting hardware platforms.

6.5.4.13 The supplier shall establish, document and maintain procedures for control of the external suppliers, including

— methods and relevant records to ensure that software provided by external suppliers adheres to established requirements. Previously developed software shall be compliant with the required software integrity level and dependability. New software shall be developed and maintained in conformity with the Software Quality Assurance Plan of the supplier or with a specific Software Quality Assurance Plan prepared by the external supplier in accordance with the Software Quality Assurance Plan of the supplier,

— methods and relevant records to ensure that the requirements provided to the external supplier are adequate and complete.

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

6.5.4.15 Within the context of this document, and to a degree appropriate to the specified software integrity level, traceability 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 tests (component, integration, overall test) and analyses that verify them.

Traceability shall be the subject of configuration management.

6.5.4.16 In special cases, e.g. pre-existing software or prototyped software, traceability may be established after the implementation and/or documentation of the code, but prior to verification/validation. In these cases, it shall be shown that verification/validation is as effective as it would have been with traceability over all phases.

6.5.4.17 Objects of requirements, design or implementation that cannot be adequately traced shall be demonstrated to have no bearing upon the safety or integrity of the system.

6.5.4.18 Where different parts of a specific software development are being undertaken by separate and distinct organizations there shall be a clear definition of the tasks being undertaken by each organization. The documentation produced by each organization shall provide sufficient references to the documentation of the other participating organizations for an understanding of the overall software development. The referenced documentation shall be available. The sub-contracting organization remains responsible for compliance with this document. The sub-contractor organization shall apply the requirements of this standard to the extent of the software development activities in which it takes part.

The complexity of the development should not obscure the tasks and responsibilities of the contributing organizations. Where more than one distinct organization contributes to the development of the software, a description of the overall software development including a definition of the organizational interfaces between the involved parties should be contained in the Software Quality Assurance Plan.

EXAMPLE An organization allocates the development of some of the software components to an external supplier. The external supplier produces the software components along with design and design verification documentation. The external supplier in this example does not have the facilities to extensively test the software in its target environment. The integration, overall testing and validation will be then carried out by the allocating organization. In this case the test reports and validation documentation are provided by the coordinating organization. However, the external supplier in this example is requested to contribute to the specification of the tests. Whatever the arrangement, it is understood by all parties what their responsibilities and tasks are. In this

example the contribution of each party is specified in the Software Quality Assurance Plan of the coordinating organization. The external supplier in this example will reference the relevant documentation, including the Software Quality Assurance Plan of the coordinating organization, in their own Software Quality Assurance Plan.

6.6 Modification and change control

6.6.1 Objectives

6.6.1.1 To ensure that the software performs as required, preserving the software integrity and dependability when modifying the software.

6.6.1.2 These objectives are managed by the Configuration Manager.

6.6.2 Input documents

1) Software Quality Assurance Plan

2) Software Configuration Management Plan

3) All relevant design, development and analysis documentation

4) Change Requests

5) Change impact analysis and authorization

6.6.3 Output documents

1) All changed input documents

2) Software Change records (see 9.2.4.10)

3) New Configuration records

6.6.4 Requirements

6.6.4.1 The Change Management Process shall define at least the following aspects:

a) the documentation needed for problem reporting and/or corrective actions, with the aim of giving feedback to the responsible management;

b) analysis of the information collected in the problem reports to identify its causes;

c) the practices to be followed for reporting, tracking and resolving problems identified during the development phase and during software maintenance;

d) the specific organizational responsibilities with regard to development and software maintenance;

e) how to apply controls to ensure that corrective actions are taken and that they are effective;

f) impact analysis of the effect of the changes on the software component under development or already delivered;

g) impact analysis shall state the re-verification, re-validation and re-assessment necessary for the change;

h) where multiple changes are applied, the impact analysis shall consider the cumulative impact;

NOTE Several changes might cumulatively require a complete re-test.

i) authorization before implementation.

6.6.4.2 All changes shall initiate a return to an appropriate phase of the lifecycle. All subsequent phases shall then be carried out in accordance with the procedures specified for the specific phases in accordance with the requirements in this document.

6.7 Support tools and languages

6.7.1 Objectives

The objective is to provide evidence that potential failures of tools do not adversely affect the integrated toolset output in a safety related manner that is undetected by technical and/or organizational measures outside the tool. To this end, software tools are categorized into three classes namely, T1, T2 and T3 respectively (see definitions in 3.1).

When tools are being used as a replacement for manual operations, the evidence of the integrity of tools output can be adduced by the same process steps as if the output was done in manual operation. These process steps might be replaced by alternative methods if an argumentation on the integrity of tools output is given and the software integrity level is not decreased by the replacement.

Tools that are used as part of the system (i.e. contributing to its intended functions) after deployment for operation in the final environment of application shall be considered as software and not subject to the requirements of 6.7.

6.7.2 Input documents

Tools specification or manual.

6.7.3 Output documents

Tools validation report (when needed, see 6.7.4.4).

6.7.4 Requirements

6.7.4.1 Software tools shall be selected as a part of the software development activities.

Appropriate tools to support the development of software should be used in order to increase the integrity of the software by reducing the likelihood of introducing or not detecting faults during the development. Examples of tools relevant to the phases of the software development lifecycle include

a) transformation or translation tools that convert a software or design representation (e.g. text or a diagram) from one abstraction level to another: design refinement tools, compilers, assemblers, linkers, binders, loaders and code generation tools,

b) verification and validation tools such as static code analysers, test coverage monitors, theorem proving assistants, simulators and model checkers,

c) diagnostic tools used to maintain and monitor the software under operating conditions,

d) infrastructure tools such as development support systems,

e) configuration control tools such as version control tools,

f) application data tools that produce or maintain data which are required to define parameters and to instantiate system functions e.g. function parameters, instrument ranges, alarm and trip levels, output states to be adopted at failure, geographical layout.

The selected tools should be able to cooperate. In this context, tools cooperate if the outputs from one tool have suitable content and format for automatic input to a subsequent tool, thus minimizing the possibility of introducing human error in the reworking of intermediate results.

Tools shall be selected and demonstrated to be compatible with the needs of the application.

The availability of suitable tools to supply the services that are necessary over the whole lifetime of the software shall be considered.

6.7.4.2 The selection of the tools in classes T2 used for SIL 1 to SIL 4 and T3 used for SIL 1 to SIL 4 shall be justified (see 7.3.4.12). The justification shall include the identification of potential failures which can be injected into the tools output and the measures to avoid or handle such failures.

6.7.4.3 All tools in classes T2 used for SIL 1 to SIL 4 and T3 shall have a specification or manual which clearly defines the behaviour of the tool and any instructions or constraints on its use.

6.7.4.4 For each tool in class T3 used for SIL 1 to SIL 4, evidence shall be available that the output of the tool conforms to the specification of the output or failures in the output are detected. Evidence shall be based on one or more of the following techniques.

a) a suitable combination of history of successful use, for which the following conditions apply:

   — It has been confirmed and periodically checked by analysis, examination and provision of objective evidence that the tool is suitable for the purpose for which it is to be used.

   — There is sufficient qualitative evidence from previous use to show that the output of the tool can be trusted, including a suitable combination of documented history of successful use (project and applications realized, list of anomalies, identification of successive versions).

b) tool validation as specified in 6.7.4.5,

c) tool diversity, which is typically obtained by using one of the possibilities below:

   — Use of two diverse tools to perform the same function, with comparison of their outputs.

   — Use one tool to perform the function and feed its output into a diverse tool which regenerates the input data provided to the first tool, with comparison of the original and regenerated input data.

   — Use one tool to perform the function and subject its output to verification of results by an independent tool, which is diverse from the first tool because it works in accordance with different principles, for example a rule-checking tool to confirm that the output conforms to all relevant rules (e.g. rules for the positioning of balises).

d) compliance with the software integrity levels derived from the risk analysis of the process and procedures including the tools,

e) other appropriate methods for avoiding or handling failures introduced by tools (e.g. verification of tool output).

NOTE 1 As an example the use of a non-trusted compiler can be justified as follows.

The object code produced by the compiler has been subjected to a combination of tests, checks and analyses which are capable of ensuring the correctness of the code to the extent that it is consistent with the target software integrity level. In particular, the following applies to all tests, checks and analyses.

— Testing has been shown to have a sufficiently high coverage of the implemented code. If there is any code unreachable by testing, it has been shown by checks or analyses that the function concerned is executed correctly when the code is reached on the target.

— Checks and analyses have been applied to the object code and shown to be capable of detecting the types of errors which might result from a defect in the compiler.

— No more translation with the compiler has taken place after testing, checking and analysis.

— If further compilation or translation is carried out, all tests, checks and analyses will be repeated.

NOTE 2 The evidence listed for T3 can also be used for T2 tools in judging the correctness of their results. The need for such judgement can arise based on the results of the tool justification as described in subclause 6.7.4.2.

6.7.4.5 If tool validation is selected as evidence to fulfil 6.7.4.4, the results of tool validation shall be documented covering the following results:

a) a record of the validation activities;

b) the version of the tool manual being used;

c) the tool functions being validated;

d) tools and equipment used to perform the tool validation;

e) the results of the validation activity; the documented results of validation shall state either that the software has passed the validation or the reasons for its failure;

f) test cases and their results for subsequent analysis;

g) discrepancies between expected and actual results.

6.7.4.6 Intentionally left blank

6.7.4.7 Intentionally left blank

6.7.4.8 Intentionally left blank

6.7.4.9 For each tool in class T3 used for SIL 1 to SIL 4, where automatic code generation or similar automatic translation takes place, the suitability of the automatic translator for safety-related software development shall be evaluated at the point in the development lifecycle where development support tools are selected.

6.7.4.10 Configuration management shall ensure that for tools in classes T2 used for SIL 1 to SIL 4 and T3, only justified versions are used.

6.7.4.11 For tools in classes T2 used for SIL 1 to SIL 4, and T3, each new version of a tool that is used shall be justified. This justification may rely on evidence provided for an earlier version if sufficient evidence is provided that

a) the functional differences (if any) will not affect tool compatibility with the rest of the toolset,

b) the new version is unlikely to contain significant new, unknown faults.

NOTE Evidence that the new version is unlikely to contain significant new unknown faults can be based on a credible identification of the changes made, and on an analysis of the verification and validation actions performed.

6.7.4.12 The relation between the tool classes and the applicable subclauses is defined within Table 1.

Table 1 — Relation between tool class and applicable subclauses

Tool class	Applicable requirements for SIL 1 to SIL 4	Applicable requirements for Basic Integrity
T1	6.7.4.1	6.7.4.1
T2	6.7.4.1, 6.7.4.2, 6.7.4.3, 6.7.4.10, 6.7.4.11	6.7.4.1
T3	6.7.4.1, 6.7.4.2, 6.7.4.3, 6.7.4.4, 6.7.4.5, 6.7.4.9, 6.7.4.10, 6.7.4.11	6.7.4.1, 6.7.4.3, 6.7.4.10, 6.7.4.11

7 Software development

7.1 Lifecycle and documentation for software

7.1.1 Objectives

To provide a description of the software itself, from the higher levels of abstraction down to the detailed refinements, in order to create a frame for the demonstration of the achieved safety as well as for future maintenance actions.

7.1.2 Requirements

7.1.2.1 To the extent required by the software integrity level, the requirements related to documents listed in Table A.1 shall be fulfilled.

7.1.2.2 The sequence of deliverable documents as they are described in Table A.1 reflects an ideal linear waterfall model. This model is however not intended to be a reference in the sense of schedule and linkage of activities, as it would usually be difficult to achieve a strict compliance in practice. Phases can overlap but verification and validation activities shall demonstrate the consistency of inputs and outputs (documents and software) within and between the phases.

However, the main purpose of the documentation foreseen is to provide a description of the software itself, from the higher levels of abstraction down to the detailed refinements, in order to create a frame for the demonstration of the achieved safety as well as for future maintenance actions.

7.2 Software requirements

7.2.1 Objectives

7.2.1.1 To describe a complete set of requirements for the software meeting all System and Safety Requirements and provide a comprehensive set of documents for each subsequent phase.

7.2.1.2 To describe the Overall Software Test Specification.

7.2.2 Input documents

1) System Requirements Specification

2) System Architecture Description

3) External Interface Specifications (e.g. Software/Software Interface Specification, Software/Hardware Interface Specification)

4) Software Quality Assurance Plan

5) Software Validation Plan

7.2.3 Output documents

1) Software Requirements Specification

2) Overall Software Test Specification

3) Software Requirements Verification Report

7.2.4 Requirements

7.2.4.1 A Software Requirements Specification shall be written, under the responsibility of the Requirements Manager, on the basis of the input documents from 7.2.2.

The requirements from 7.2.4.2 to 7.2.4.15 refer to the Software Requirements Specification.

7.2.4.2 The Software Requirements Specification shall express the required properties of the software being developed. These properties shall include

a) functionality (including capacity and response time performance),

b) robustness and maintainability,

c) safety (including safety-related functions and their associated safety integrity levels),

d) efficiency,

e) usability,

f) portability.

7.2.4.3 The software integrity level shall be derived as defined in Clause 4 and recorded in the Software Requirements Specification.

7.2.4.4 The Software Requirements Specification shall be expressed and structured in such a way that it is

a) complete, clear, precise, unequivocal, verifiable, maintainable and feasible,

b) traceable back to all the input documents.

7.2.4.5 The Software Requirements Specification shall include modes of expression and descriptions which are understandable to the responsible personnel involved in the lifecycle of the software.

7.2.4.6 The Software Requirements Specification shall identify and document or reference all interfaces with any other system, either within or outside the equipment under control, including operators, wherever a direct connection exists or is planned.

7.2.4.7 All relevant modes of operation shall be detailed in the Software Requirements Specification.

7.2.4.8 All relevant modes of behaviour of the programmable electronics, in particular failure behaviour, shall be documented or referenced (e.g. system level documentation) in the Software Requirements Specification.

7.2.4.9 Any constraints between the hardware and the software shall be documented or referenced (e.g. system level documentation) in the Software Requirements Specification.

7.2.4.10 To the extent required by the description of system documentation, the Software Requirements Specification shall consider the software self-checking and the hardware checking by the software. Software self-checking consists of the detection and reporting by the software of its own failures and errors.

7.2.4.11 The Software Requirements Specification shall include requirements for the periodic testing of functions to the extent required by the safety requirements.

7.2.4.12 The Software Requirements Specification shall include requirements to enable all functions to be testable during overall system operation to the extent required by the System Requirements Specification.

7.2.4.13 All functions to be performed by the software shall be clearly identified in the Software Requirements Specification, including the required software integrity level.

7.2.4.14 The Software Requirements Specification shall explicitly identify any functions that can be configured by application data. If the Software Requirements Specification defines any such functions, the software is considered “configurable by application data”.

7.2.4.15 The Software Requirements Specification shall be supported by techniques and measures from Table A.2, as chosen in the Software Quality Assurance Plan (see 6.5.4.5).

7.2.4.16 An Overall Software Test Specification shall be written, under the responsibility of the Tester, on the basis of the Software Requirements Specification.

The requirements from 7.2.4.17 to 7.2.4.19 refer to the Overall Software Test Specification.

7.2.4.17 The Overall Software Test Specification shall be a description of the tests to be performed on the completed software.

7.2.4.18 To the extent required by the software integrity level, the Overall Software Test Specification shall apply techniques and measures from Table A.7, as chosen in the Software Verification Plan (see 6.2.4.5).

7.2.4.19 The Overall Software Test Specification shall identify for each required function the test cases including

a) the required input signals with their sequences and their values,

b) the anticipated output signals with their sequences and their values,

c) the test success criteria, including performance and quality aspects.

d) evidence that the software behaves in an appropriate manner when the interface is subjected to inputs which are out of specification (only necessary if not performed during software integration test, see 7.3.4.32).

7.2.4.20 A Software Requirements Verification Report shall be written, under the responsibility of the Verifier, on the basis of the System Requirements Specification, Software Requirements Specification, Overall Software Test Specification and Software Quality Assurance Plan.

Requirements from 7.2.4.21 to 7.2.4.22 refer to the Software Requirements Verification Report.

7.2.4.21 The Software Requirements Verification Report shall be written in accordance with the generic requirements established for all the Verification Reports (see 6.2.4.13).

7.2.4.22 Once the Software Requirements Specification has been established, verification shall address

a) the adequacy of the Software Requirements Specification in fulfilling the requirements set out in the System Requirements Specification and the Software Quality Assurance Plan,

b) that the Software Requirements Specification meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17 as well as the specific requirements in 7.2.4.2 to 7.2.4.15,

c) the adequacy of the Overall Software Test Specification as a test against the Software Requirements Specification,

d) the definition of any additional activity in order to demonstrate the correct coverage of not testable requirements,

e) the internal consistency of the Software Requirements Specification,

f) the adequacy of the Software Requirements Specification in fulfilling or taking into account the constraints between hardware and software.

The results shall be recorded in a Software Requirements Verification Report.

7.3 Architecture and design

7.3.1 Objectives

7.3.1.1 To develop a software architecture that achieves the requirements of the software.

7.3.1.2 To identify and evaluate the significance of hardware/software interactions for safety.

7.3.1.3 To choose a design method if one has not been previously defined.

7.3.1.4 To design software of a defined software integrity level from the input documents.

7.3.1.5 To ensure that the resultant system and its software will be readily testable from the outset. As verification and test will be a critical element in the validation, particular consideration shall be given to verification and test needs throughout the implementation.

7.3.2 Input documents

1) Software Requirements Specification

7.3.3 Output documents

1) Software Architecture Specification

2) Software Design Specification

3) Software Interface Specifications

4) Software Integration Test Specification

5) Software/Hardware Integration Test Specification

6) Software Architecture and Design Verification Report

7.3.4 Requirements

7.3.4.1 To the extent required by the software integrity level (see Table A.1) a Software Architecture Specification shall be written, under the responsibility of the Designer, on the basis of the Software Requirements Specification.

Requirements from 7.3.4.2 to 7.3.4.14 refer to the Software Architecture Specification.

7.3.4.2 The proposed software architecture shall be established and detailed in the Software Architecture Specification.

7.3.4.3 The Software Architecture Specification shall consider the feasibility of achieving the Software Requirements Specification at the required software integrity level.

NOTE The goal is to minimize the size and complexity of the safety part of the application.

7.3.4.4 The Software Architecture Specification shall identify, analyse and detail the significance of all hardware/software interactions.

7.3.4.5 The Software Architecture Specification shall identify all software components and for these components identify

a) whether these components are new or existing,

b) whether these components have been previously validated and if so their validation conditions,

c) the software integrity level of the component.

7.3.4.6 Software components shall

a) cover a defined subset of software requirements,

b) be clearly identified and independently versioned inside the configuration management system.

7.3.4.7 The use of pre-existing software shall be subject to the following restrictions.

a) For all software integrity levels the following information shall clearly be identified and documented:

— the requirements that the pre-existing software is intended to fulfil;

— the assumptions about the environment of the pre-existing software;

— interfaces with other parts of the software.

b) For all software integrity levels the pre-existing software shall be included in the validation process of the whole software.

c) For software integrity levels SIL 3 or SIL 4, the following precautions shall be taken:

— an analysis of possible failures of the pre-existing software and their consequences on the whole software shall be carried out;

— a strategy shall be defined to detect failures of the pre-existing software and to protect the system from these failures;

— the verification and validation process shall ensure

1) that the pre-existing software fulfils the allocated requirements,

2) that failures of the pre-existing software are detected and the system where the pre-existing software is integrated into is protected from these failures,

3) that the assumptions about the environment of the pre-existing software are fulfilled.

d) The pre-existing software shall be accompanied by a sufficiently precise (e.g. limited to the used functions) and complete description (i.e. functions, constraints and evidence). The description shall include hardware and/or software constraints of which the Designer and the Tester shall be aware and take into consideration during application. In particular it is the means for informing the Designer and the Tester of what the software was designed for, its properties, behaviour and characteristics.

Statistical evidence may be used in the validation strategy of the pre-existing software.

7.3.4.8 The use of existing verified software components developed according to this document in the design is to be preferred wherever possible.

7.3.4.9 Where the software consists of components of different software integrity levels then all of the software components shall be managed as belonging to the highest of these levels unless there is evidence of non-interference between the higher software integrity level components and the lower software integrity level components. Software parts related to the control of non-interference shall be managed as belonging to the highest of these integrity levels. This evidence shall be recorded in the Software Architecture Specification.

NOTE Non-interference means freedom from adverse interference with the safety requirements to be implemented by the software under consideration.

Evidence of non-interference should address both spatial and temporal domains in relation with the safety requirements to be implemented by the software under consideration:

— Spatial: the safety related data used by one component should not be changed by another component with different software integrity level.

— Temporal: one software component should not cause another component with different software integrity level to function unsafely by taking up too much of the available processor execution time, or by unduly using a shared resource of some kind.

Such evidence should demonstrate the control of sources of interference including, at least:

— Timing and execution (i.e. respect of timing, respect of program sequence);

— Data access (i.e. protection against read/write violations);

— Exchange of information (i.e. integrity, authenticity, sequence and timeliness of exchanged data);

— Shared use of Hardware resources (i.e. CPU, hardware peripherals, memories);

— Capability to trigger relevant safe state in time when an adverse interference is detected should be demonstrated.

The set of measures taken to control the sources of interference should be analysed to justify their completeness.

7.3.4.10 The Software Architecture Specification shall describe the strategy for the software development to the extent required by the software integrity level (see Table A.3). The Software Architecture Specification shall be expressed and structured in such a way that it is

a) complete, consistent, clear, precise, unequivocal, verifiable, testable, maintainable and feasible,

b) traceable back to the Software Requirements Specification.

7.3.4.11 Measures for handling faults shall be included in the Software Architecture Specification in order to achieve the balance between the fault avoidance and fault handling strategies.

7.3.4.12 The Software Architecture Specification shall justify that the techniques, measures and tools chosen form a set which satisfies the Software Requirements Specification at the required software integrity level.

7.3.4.13 A rigid separation between the generic software and the application data shall be enforced, i.e. it shall be possible to recompile and update either the software or the application data without needing to update the other, unless there has been a change to the defined interface between the software and the application data. Likewise, the applications specific data shall be separated from the application-generic data.

7.3.4.14 The Software Architecture Specification shall apply techniques and measures from Table A.3, as chosen in the Software Quality Assurance Plan (see 6.5.4.5).

NOTE one of the goals to be considered within architecture definition is to balance size and complexity

7.3.4.15 Intentionally left blank

7.3.4.16 Prototyping may be used in any phase to elicit requirements or to obtain a more detailed view on requirements and their consequences.

7.3.4.17 Code from a prototype may be used in the target system only if it is demonstrated that the code and its development and documentation fulfils the requirements of this document.

7.3.4.18 A Software Interface Specification for all Interfaces between the components of the software and the boundary of the overall software shall be written, under the responsibility of the Designer, on the basis of the Software Requirements Specification and the Software Architecture Specification.

For software that is configurable by application data, the Software Interface Specification shall address the application data.

For Basic Integrity software, the software interface specification is only required for the boundary of the overall software.

The requirement in 7.3.4.19 refers to the Software Interface Specification.

7.3.4.19 The description of interfaces shall address

a) pre/post conditions,

b) definition and description of all boundary values for all specified data,

c) behaviour when the boundary value is exceeded,

d) behaviour when the value is at the boundary,

e) for time-critical input and output data:

1) time constraints and requirements for correct operation,

2) management of exceptions.

f) allocated memory for the interface buffers and the mechanisms to detect that the memory cannot be allocated or all buffers are full, where applicable,

g) existence of synchronization mechanisms between functions (see e).

All data from and to the interfaces shall be defined for the whole range of values defined by the type of the data, including the ranges which are not used when processed by the functions:

h) definition and description of all equivalence classes for all specified data and each software function using them,

i) definition of unused or forbidden equivalence classes.

NOTE The data type includes the following:

1) input parameters and output results of functions and/or procedures;

2) data specified in telegrams or communication packets;

3) data from the hardware;

4) application data for configurable software.

7.3.4.20 To the extent required by the software integrity level (see Table A.1), a Software Design Specification shall be written, under the responsibility of the Designer, on the basis of the Software Requirements Specification, the Software Architecture Specification and the Software Interface Specification.

Requirements from 7.3.4.21 to 7.3.4.24 refer to the Software Design Specification.

7.3.4.21 The input documents shall be available, although not necessarily finalized, prior to the start of the design process.

7.3.4.22 The Software Design Specification shall describe the software design based on a decomposition into components with each component having a Software Component Design Specification and a Software Component Test Specification.

7.3.4.23 The Software Design Specification shall address

a) software components traced back to software architecture and their software integrity level,

b) interfaces of software components with the environment,

c) interfaces between the software components,

d) data structures,

e) allocation and tracing of requirements on components,

f) main algorithms and sequencing,

g) error reporting mechanisms.

For software that is configurable by application data, the Software Design Specification shall address the interfaces between the software and the application data, to the extent that these interfaces have not already been considered in the Software Architecture Specification and Software Interface Specification.

The software shall be designed to detect corrupted application data where this is feasible.

7.3.4.24 The Software Design Specification shall apply to the extent required by the software integrity level techniques and measures from Table A.4, as chosen in the Software Quality Assurance Plan (see 6.5.4.5).

7.3.4.25 A suitable programming language shall be selected regarding the criteria in Table A.15.

NOTE The term programming language is not limited to textual languages but also includes diagrammatical or modelling languages.

7.3.4.26 Coding standards shall be developed and specify

a) good programming practice, as defined by Table A.12,

b) measures to avoid or detect errors which can be made during application of the language and are not detectable during the verification (see 7.5 and 7.6). Such failures are derived by analysis over all features of the language,

c) procedures for source code documentation.

7.3.4.27 The selection of a coding standard shall be justified to the extent required by the software integrity level (see Table A.4 and Table A.12).

7.3.4.28 The coding standards shall be used for the development of all software and be referenced in the Software Quality Assurance Plan.

7.3.4.29 In accordance with the required software integrity level the design method chosen shall possess features that facilitate

a) abstraction, modularity and other features which control complexity,

b) the clear and precise expression of

   1) functionality,

   2) information flow between components,

   3) sequencing and time related information,

   4) concurrency,

   5) data structure and properties,

c) human comprehension,

d) verification and validation,

e) software maintenance.

7.3.4.30 A Software Integration Test Specification shall be written, under the responsibility of the Tester on the basis of the Software Requirements Specification, the Software Architecture Specification, the Software Design Specification and the Software Interface Specifications.

The requirements from 7.3.4.31 to 7.3.4.33 refer to the Software Integration Test Specification.

7.3.4.31 The Software Integration Test Specification shall be written in accordance with the generic requirements established for a Test Specification (see 6.1.4.4).

7.3.4.32 The Software Integration Test Specification shall address the following:

a) it shall be shown that each software component provides the specified interfaces for the other components by executing the components together;

b) it shall be shown that the software behaves in an appropriate manner when the interface is subjected to inputs which are out of specification;

c) the required input data with their sequences and their values shall be the basis of the test cases;

d) the anticipated output data with their sequences and their values shall be the basis of the test cases;

e) it shall be shown which results of the component test (see 7.5.4.5 and 7.5.4.7) are intended to be reused for the software integration test.

7.3.4.33 The Software Integration Test Specification shall apply techniques and measures from Table A.6, as chosen in the Software Verification Plan (see 6.2.4.5).

7.3.4.34 To the extent required by the software integrity level (see Table A.1), a Software/Hardware Integration Test Specification shall be written, under the responsibility of the Tester, on the basis of the system documentation, the Software Requirements Specification, the Software Architecture Specification and the Software Design Specification.

The requirements from 7.3.4.35 to 7.3.4.40 refer to the Software/Hardware Integration Test Specification.

7.3.4.35 A Software/Hardware Integration Test Specification should be created early in the development lifecycle, in order that integration testing may be properly directed and that particular design or other integration needs may be suitably provided for. Depending upon the size of the system, the Software/Hardware Integration Test Specification may be subdivided during development into a number of child documents and be naturally added to, as the hardware and software designs evolve and the detailed needs of integration test become clearer.

7.3.4.36 The Software/Hardware Integration Test Specification shall distinguish between those activities which can be carried out by the supplier on his premises and those that require access to the user's site.

7.3.4.37 To the extent required by the software integrity level (see Table A.1), the Software/Hardware Integration Test Specification shall address the following:

a) it shall be shown that the software runs in a proper way on the hardware using the hardware via the specified hardware interfaces;

b) it shall be shown that the software can handle hardware faults as required;

c) the required timing and performance shall be demonstrated;

d) the required input data with their sequences and their values shall be the basis of the test cases;

e) the anticipated output data with their sequences and their values shall be the basis of the test cases;

f) it shall be shown which results of the component test (see 7.5.4.5) and of the software integration test (see 7.6.4.3) are intended to be reused for the software/hardware integration test.

7.3.4.38 The Software/Hardware Integration Test Specification shall document the following:

a) test cases and test data;

b) types of tests to be performed;

c) test environment including tools, support software and configuration description;

d) test criteria on which the completion of the test will be judged.

7.3.4.39 The Software/Hardware Integration Test Specification shall be written in accordance with the generic requirements established for a Test Specification (see 6.1.4.4).

7.3.4.40 The Software/Hardware Integration Test Specification shall apply to the extent required by the software integrity level techniques and measures from Table A.6, as chosen in the Software Verification Plan (see 6.2.4.5).

7.3.4.41 To the extent required by the software integrity level (see Table A.1), a Software Architecture and Design Verification Report shall be written, under the responsibility of the Verifier, on the basis of the Software Requirements Specification, Software Architecture Specification, Software Design Specification, Software Interface Specifications, Software Integration Test Specification and Software/Hardware Integration Test Specification.

The requirements from 7.3.4.42 to 7.3.4.44 refer to the Software Architecture and Design Verification Report.

7.3.4.42 The Software Architecture and Design Verification Report shall be written in accordance with the generic requirements established for a Verification Report (see 6.2.4.13).

7.3.4.43 After the Software Architecture, Interface and Design Specifications have been established, verification shall address

a) the internal consistency of the Software Architecture, Interface and Design Specifications,

b) the adequacy of the Software Architecture, Interface and Design Specifications in fulfilling the Software Requirements Specification with respect to consistency and completeness,

c) that the Software Architecture Specification meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.16 as well as the specific requirements in 7.3.4.1 to 7.3.4.14,

d) that the Software Interface Specification meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.16 as well as the specific requirements in 7.3.4.18 to 7.3.4.19,

e) that the Software Design Specification meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.16 as well as the specific requirements in 7.3.4.20 to 7.3.4.24,

f) the adequacy of the Software Architecture Specification and the Software Design Specification in taking into account the hardware and software constraints.

The results shall be recorded in a Software Architecture and Design Verification Report.

7.3.4.44 After the Software Integration and Software/Hardware Integration Test Specifications have been established, verification shall address

a) that the Software Integration Test Specification meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.16, as well as the specific requirements in 7.3.4.30 to 7.3.4.33,

b) that the Software/Hardware Integration Test Specification meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.16, as well as the specific requirements in 7.3.4.34 to 7.3.4.40.

The results shall be recorded in a Software Architecture and Design Verification Report.

7.4 Component design

7.4.1 Objectives

7.4.1.1 To develop a software component design that achieves the requirements of the Software Design Specification to the extent required by the software integrity level (see Table A.1 and Table A.20).

7.4.1.2 To develop a software component test specification that achieves the requirements of the Software Component Design Specification to the extent required by the software integrity level (see Table A.1).

7.4.2 Input documents

1) Software Design Specification

7.4.3 Output documents

1) Software Component Design Specification

2) Software Component Test Specification

3) Software Component Design Verification Report

7.4.4 Requirements

7.4.4.1 For each component, a Software Component Design Specification shall be written to the extent required by the software integrity level (see Table A.1), under the responsibility of the Designer, on the basis of the Software Design Specification.

Requirements from 7.4.4.2 to 7.4.4.6 refer to the Software Component Design Specification.

7.4.4.2 For each software component, the following information shall be available

— author,

— configuration history, and

— short description.

The configuration history shall include a precise identification of the current and all previous versions of the component, specifying the version, date and author, and a description of the changes made from the previous version.

7.4.4.3 The Software Component Design Specification shall address

a) identification of all lowest software units (e.g. subroutines, methods, procedures) traced back to the upper level,

b) their detailed interfaces with the environment and other components with detailed inputs and outputs,

c) their software integrity level without any further apportionment within the component itself,

d) detailed algorithms and data structures.

Each Software Component Design Specification shall be self consistent and allow transforming into code of the corresponding components.

7.4.4.4 Each Software Component Design Specification shall be readable, understandable and testable.

7.4.4.5 The size and complexity of each developed Software Component shall be balanced.

7.4.4.6 To the extent required by the software integrity level the Software Component Design Specification shall apply techniques and measures from Table A.4, as chosen in the Software Quality Assurance Plan (see 6.5.4.5).

7.4.4.7 To the extent required by the software integrity level, a Software Component Test Specification shall be written for each component, under the responsibility of the Tester, on the basis of the Software Component Design Specification and the Software Design Specification (see Table A.1).

The requirements from 7.4.4.8 to 7.4.4.10 refer to the Software Component Test Specification.

7.4.4.8 The Software Component Test Specification shall be written in accordance with the generic requirements established for a Test Specification (see 6.1.4.4).

7.4.4.9 A Software Component Test Specification shall be produced against which the component shall be tested. These tests shall show that each component performs its intended function. The Software Component Test Specification shall define and justify the required criteria and degree of test coverage to the extent required by the software integrity level (see Table A.5 and Table A.21). Tests shall be designed so as to fulfil three objectives:

a) to confirm that the component performs its intended functions;

b) to check how the internal parts of the component interact to carry out the intended functions;

c) to confirm that all parts of the component are tested.

7.4.4.10 The Software Component Test Specification shall apply techniques and measures from Table A.5, as chosen in the Software Verification Plan (see 6.2.4.5).

7.4.4.11 To the extent required by the software integrity level, a Software Component Design Verification Report shall be written, under the responsibility of the Verifier, on the basis of the Software Design Specification, Software Component Design Specification and Software Component Test Specification (see Table A.1).

Requirements from 7.4.4.12 to 7.4.4.13 refer to the Software Component Design Verification Report.

7.4.4.12 The Software Component Design Verification Report shall be written in accordance with the generic requirements established for a Verification Report (see 6.2.4.13).

7.4.4.13 After each Software Component Design Specification has been established, verification shall address

a) the adequacy of the Software Component Design Specification in fulfilling the Software Design Specification,

b) that the Software Component Design Specification meets general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17, as well as the specific requirements in 7.4.4.1 to 7.4.4.6,

c) the adequacy of the Software Component Test Specification as a set of test cases for the Software Component Design Specification,

d) that the Software Component Test Specification meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17, as well as the specific requirements in 7.4.4.7 to 7.4.4.10,

e) the decomposition of the Software Design Specification into software components and the Software Component Design Specification with reference to

    1) feasibility of the performance required,

2) testability for further verification, and

3) maintainability to permit further evolution.

The results shall be recorded in a Software Component Design Verification Report.

7.5 Component implementation and testing

7.5.1 Objectives

To achieve software which is analysable, testable, verifiable and maintainable. Component testing is also included in this phase.

7.5.2 Input documents

1) Software Component Design Specification

2) Software Component Test Specification

7.5.3 Output documents

1) Software Source Code and supporting documentation

2) Software Component Test Report

3) Software Source Code Verification Report

7.5.4 Requirements

7.5.4.1 The Software Source Code shall be written under the responsibility of the Implementer on the basis of the Software Component Design Specification.

Requirements from 7.5.4.2 to 7.5.4.4 refer to the software source code.

NOTE The term Source Code also applies to other inputs (e.g. models) for the generation (compilation) to executable code. Source Code is what the Implementer implements.

7.5.4.2 The size and complexity of the developed source code shall be balanced.

7.5.4.3 The Software Source Code shall be readable, understandable and testable.

7.5.4.4 The Software Source Code shall be placed under configuration control before the commencement of documented testing.

7.5.4.5 To the extent required by the software integrity level (see Table A.1), a Software Component Test Report shall be written, under the responsibility of the Tester, on the basis of the Software Component Test Specification and the Software Source Code.

Requirements from 7.5.4.6 to 7.5.4.7 refer to the Software Component Test Report.

7.5.4.6 The Software Component Test Report shall be written in accordance with the generic requirements established for a Test Report (see 6.1.4.5).

7.5.4.7 The Software Component Test Report shall include the following features.

a) A statement of the test results and whether each component has met the requirements of its Software Component Design Specification.

b) A statement of test coverage shall be provided for each component, showing that the required degree of test coverage has been achieved for all required criteria.

7.5.4.8 To the extent required by the software integrity level (see Table A.1), a Software Source Code Verification Report shall be written, under the responsibility of the Verifier, on the basis of the

Software Component Design Specification, the Software Component Test Specification and the Software Source Code.

Requirements from 7.5.4.9 to 7.5.4.10 refer to the Software Source Code Verification Report.

7.5.4.9 The Software Source Code Verification Report shall be written in accordance with the generic requirements established for a Verification Report (see 6.2.4.13).

7.5.4.10 After the Software Source Code and the Software Component Test Report have been established, verification shall address

a) the adequacy of the Software Source Code as an implementation of the Software Component Design Specification,

b) determining the correct application of the coding standards,

c) that the Software Source Code meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17, as well as the specific requirements in 7.5.4.1 to 7.5.4.4,

d) the adequacy of the Software Component Test Report as a record of the tests carried out in accordance with the Software Component Test Specification.

The results shall be recorded in a Software Source Code Verification Report.

7.6 Integration

7.6.1 Objectives

7.6.1.1 To carry out integration activities, software integration testing and software/hardware integration testing.

7.6.1.2 To demonstrate that the software and the hardware interact correctly to perform their intended functions.

7.6.2 Input documents

1) Software/Hardware Integration Test Specification

2) Software Integration Test Specification

3) Software Source Code and supporting documentation

7.6.3 Output documents

1) Software Integration Test Report

2) Software/Hardware Integration Test Report

3) Software Integration Verification Report

7.6.4 Requirements

7.6.4.1 The integration of software components shall be the process of progressively combining individual and previously tested components into a composite whole in order that the components interfaces and the assembled software may be adequately proven prior to system integration and system test.

7.6.4.2 During software/hardware integration any modification or change to the integrated system shall be subject to an impact study which shall identify all components impacted and the necessary reverification activities.

7.6.4.3 A Software Integration Test Report shall be written, under the responsibility of the Tester, on the basis of the Software Integration Test Specification.

Requirements from 7.6.4.4 to 7.6.4.6 refer to the Software Integration Test Report.

7.6.4.4 The Software Integration Test Report shall be written in accordance with the generic requirements established for a Test Report (see 6.1.4.5).

7.6.4.5 A Software Integration Test Report shall be produced as follows:

a) a Software Integration Test Report shall be produced stating the test results and whether the objectives and criteria of the Software Integration Test Specification have been met. If there is a failure, the circumstances for the failure shall be recorded;

b) test cases and their results shall be recorded, preferably in machine readable form for subsequent analysis;

c) tests shall be repeatable and, if practicable, be performed by automatic means;

d) the Software Integration Test Report shall document the identity and configuration of all the items involved.

7.6.4.6 The Software Integration Test Report shall demonstrate the correct use of the chosen techniques and measures from Table A.6 as a set satisfying 4.8 and 4.9.

7.6.4.7 To the extent required by the software integrity level (see Table A.1), a Software/Hardware Integration Test Report shall be written, under the responsibility of the Tester, on the basis of the Software/Hardware Integration Test Specification.

Requirements from 7.6.4.8 to 7.6.4.10 refer to the Software/Hardware Integration Test Report.

7.6.4.8 The Software/Hardware Integration Test Report shall be written in accordance with the generic requirements established for a Test Report (see 6.1.4.5).

7.6.4.9 A Software/Hardware Integration Test Report shall be produced as follows:

a) the Software /Hardware Integration Test Report shall state the test results and whether the objectives and criteria of the Software/Hardware Integration Test Specification have been met. If there is a failure, the circumstances of the failure shall be recorded;

b) test cases and their results shall be recorded, preferably in a machine-readable form for subsequent analysis;

c) the Software/Hardware Integration Test Report shall document the identity and configuration of all items involved.

7.6.4.10 The Software/Hardware Integration Test Report shall demonstrate the correct use of the chosen techniques and measures from Table A.6 as a set satisfying 4.8 and 4.9.

7.6.4.11 To the extent required by the software integrity level (see Table A.1), a Software Integration Verification Report shall be written, under the responsibility of the Verifier, on the basis of the Software and Software/Hardware Integration Test Specifications and the corresponding test reports.

Requirements from 7.6.4.12 to 7.6.4.13 refer to the Software Integration Verification Report.

7.6.4.12 The Software Integration Verification Report shall be written in accordance with the generic requirements established for a Verification Report (see 6.2.4.13).

7.6.4.13 After the Software Integration Test Report and the Software/Hardware Integration Test Report have been established, verification shall address

a) the adequacy of the Software Integration Test Report as a record of the tests carried out in accordance with the Software Integration Test Specification,

b) whether the Software Integration Test Report meets the requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17, as well as the specific requirements in 7.6.4.3 to 7.6.4.6,

c) the adequacy of the Software/Hardware Integration Test Report as a record of the tests carried out in accordance with the Software/Hardware Integration Test Specification,

d) whether the Software/Hardware Integration Test Report meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17, as well as the specific requirements in 7.6.4.7 to 7.6.4.10.

7.7 Overall software testing / Final validation

7.7.1 Objectives

To analyse and test the integrated software and hardware to ensure compliance with the Software Requirements Specification with particular emphasis on the functional and safety aspects according to the software integrity level and to check whether it is fit for its intended application.

7.7.2 Input documents

1) Software Requirements Specification

2) Overall Software Test Specification

3) Software Verification Plan

4) Software Validation Plan

5) All Hardware and Software Documentation including intermediate verification results

6) System Requirements Specification

7.7.3 Output documents

1) Overall Software Test Report

2) Software Validation Report

3) Release Note

7.7.4 Requirements

7.7.4.1 An Overall Software Test Report shall be written, under the responsibility of the Tester, on the basis of the Overall Software Test Specification.

Requirements from 7.7.4.2 to 7.7.4.4 refer to the Overall Software Test Report.

7.7.4.2 The Overall Software Test Report shall be written in accordance with the generic requirements established for a Test Report (see 6.1.4.5).

7.7.4.3 The Validator shall specify and perform supplementary tests on his discretion or have them performed by the Tester.

7.7.4.4 The results of all tests and analyses shall be recorded in an Overall Software Test Report.

7.7.4.5 The software shall be exercised either by connection to real items of hardware or actual systems with which it would interface in operation, or by simulation of input signals and loads driven by outputs. It shall be exercised under conditions present during normal operation, anticipated occurrences and undesired conditions requiring system action. Where simulated inputs or loads are used it shall be shown that these do not differ statically or dynamically from the inputs and loads

encountered in operation. Where there are differences, the consequences shall be analysed in order to evaluate what kind of tests have to be defined and performed in the real environment and which are still suitable to be performed on the test bench.

NOTE Simulated inputs or loads might replace inputs or loads which are only present at system level or in faulty modes.

7.7.4.6 A Software Validation Report shall be written, under the responsibility of the Validator, on the basis of the Software Validation Plan.

Requirements from 7.7.4.7 to 7.7.4.11 refer to the Software Validation Report.

7.7.4.7 The Software Validation Report shall be written in accordance with the generic requirements established for the Validation Report (see 6.3.4.7 to 6.3.4.11).

7.7.4.8 Once integration is finished and overall software testing and analysis are complete, a Software Validation Report shall be produced as follows:

a) it shall state whether the objectives and criteria of the Software Validation Plan have been met. Deviations to the plan shall be recorded and justified;

b) it shall give a summary statement on the tests results and whether the whole software on its target machine fulfils the requirements set out in the Software Requirements Specification;

c) an evaluation of the test coverage on the requirements of the Software Requirements Specification shall be provided;

d) an evaluation of other verification activities in accordance to the Software Verification Plan and Report shall be done together with a check that requirements tracing is fully performed and covered;

e) if the Validator produces own test cases not given to the Tester the Software Validation Report shall document these in accordance with 6.3.4.7 to 6.3.4.11.

7.7.4.9 The Software Validation Report shall contain the confirmation that each combination of techniques or measures selected according to Annex A is appropriate to the defined software integrity level. It shall contain an evaluation of the overall effectiveness of the combination of techniques and measures adopted, taking account of the size and complexity of the software produced and the actual results of testing, verification and validation activities.

7.7.4.10 The following shall be addressed in the Software Validation Report:

a) documentation of the identity and configuration of the software;

b) statement of appropriate identification of technical support software and equipment;

c) statement of appropriate identification of simulation models used;

d) statement about the adequacy of the Overall Software Test Specification;

e) confirmation of the correctness, consistency and adequacy of release notes for the software under consideration;

f) collection and keeping track of any deviations found;

g) review and evaluation of all deviations in terms of risk (impact);

h) a statement that the project has performed appropriate handling of corrective actions in accordance with the change management process and procedures and with a clear identification of any discrepancies found;

i) statement of each restriction given by the deviations in a traceable way;

j) a conclusion whether the software is fit for its intended application, taking into account the application conditions and constraints.

7.7.4.11 Any discrepancies found, including detected errors and non-compliances with this document or with any of the software requirements or plans, as well as constraints and limitations, shall be clearly identified in a separate subclause of the Software Validation Report, evaluated regarding the software integrity level and included in any Release Note which accompanies the delivered software.

7.7.4.12 A Release Note which accompanies the delivered software shall include:

a) information of compatibility between software and hardware;

b) all restrictions and conditions for using the software. These are derived from:

— the detected errors,

— non-compliances with the requirements of this document,

— degree of fulfilment of the requirements,

— degree of fulfilment of any plan.

8 Development of application data: systems configured by application data

8.1 Objectives

8.1.1 A characteristic feature in many railway systems is the need to design each installation to meet the individual requirements for a specific application. A system configured by application data allows approved software to be customised with the individual requirements for each specific application.

NOTE A generic software can also be configured by application software which determine the desired response of the system according to its inputs. The requirements for the development of application software are the same as for the development of software as described in Clauses 4 to 7 and Clause 9.

The objectives for the development of application data are the correct deriving of the data from the given installation and the check of the intended behaviour. For SIL 1 to SIL 4, this is followed by an assessment of the development process used for that application data.

A typical example is a system whose software is configured to each specific installation by instantiation and interconnection of configuration data. For instance, the signalling principles of an interlocking system (e.g. signal management, point management) or configuration of a door control unit (e.g. duration of warning sounds, time before closure) may be implemented by a set of application data.

Application data typically take the form of parameter values or descriptions (e.g. identity, type, location) of external objects.

The customisation of systems through configurability gives the designer different degrees of control over the detailed software functionality.

8.1.2 The procedures and the tools used for the development of application data shall be appropriate to the system safety integrity level as determined by the function for which they are developed.

8.1.3 The subclauses below describe the requirements for the initial development of a configurable system and for the subsequent development of each set of application-specific data.

8.2 Input documents

1) Software Requirements Specification

2) Software Architecture Specification

3) Application conditions of the software and support tools

4) User documentation of the software and support tools

8.3 Output documents

1) Application Preparation Plan

2) Application Requirements Specification

3) Application Architecture and Design

4) Application Test Specification

5) Application Test Report

6) Application Preparation Verification Report

7) Application Data

8) Application Data Verification Report

9) Application Integration Test Specification

10) Application Integration Test Report

11) Application Release Note

8.4 Requirements

8.4.1 Application development process

8.4.1.1 To the extent required by the software integrity level (see Table A.1), an Application Preparation Plan shall be written, under the responsibility of the Designer, on the basis of the input documents from 8.2.

The requirements from 8.4.1.2 to 8.4.1.11 refer to the Application Preparation Plan.

8.4.1.2 An Application Preparation Plan shall be produced in order to define and detail the application development process, including all the activities, deliverables and roles in charge of them.

NOTE The Application Preparation Plan contains the definition of an application development process that can be re-used for a class of specific applications (i.e. for a generic application).

8.4.1.3 The Application Preparation Plan shall define a documentation structure for the application preparation process.

8.4.1.4 The Application Preparation Plan shall choose, to the extent required by the software integrity level, techniques and measures from Table A.11. The selected combination shall be justified as a set satisfying 4.8 and 4.9.

8.4.1.5 The Application Preparation Plan shall specify the procedures and support tools (with their classification based on 6.7) to be used in the application development process.

8.4.1.6 The Application Preparation Plan shall include verification and validation activities to ensure that the application data are complete, correct and compatible with each other and with the generic application, and to provide evidence that the application conditions of the generic application are met. These verification and validation activities and evidence can be replaced by verification and validation

performed on the tools that produce the application data. The results are gathered together in the respective verification and the test reports (8.4.1.12, 8.4.2.4, 8.4.4.2, 8.4.4.7, 8.4.5.4).

8.4.1.7 The Application Preparation Plan shall include verification and validation activities to ensure that the support tools and the software are compatible with each other and with the specific application, and to provide evidence that their application conditions are met.

8.4.1.8 For safety-related software a failure analysis shall be carried out covering the application development process, including the support tools and procedures, in order to validate the Application Preparation Plan and to meet the required safety integrity level.

8.4.1.9 The Application Preparation Plan shall specify the requirements for the independence between staff carrying out verification, validation and preparation tasks according to 5.1.

NOTE Data preparation activities are carried out by Application Designers.

8.4.1.10 The Application Preparation Plan shall define a tool class for any hardware or software tools used in the application preparation lifecycle.

8.4.1.11 Where possible, the Application Preparation Plan shall call for notations for specifying requirements and design which are familiar to applications engineers. Where new notations are introduced, the necessary user documentation shall be provided, as well as training where appropriate.

8.4.1.12 An Application Preparation Verification Report shall be written, under the responsibility of the Verifier, on the basis of the input documents from 8.2.

The requirement in 8.4.1.13 refers to the Application Preparation Verification Report.

8.4.1.13 Once the Application Preparation Plan has been established, verification shall confirm

a) that the Application Preparation Plan meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17 as well as the specific requirements in 8.4.1.2 to 8.4.1.11,

b) the internal consistency of the Application Preparation Plan.

The results shall be recorded in the Application Preparation Verification Report.

8.4.1.14 The implementation of the Application Preparation Plan shall be verified and validated for each specific application.

8.4.2 Application requirements specification

8.4.2.1 An Application Requirements Specification shall be written, under the responsibility of the Requirements Manager, on the basis of the input documents from 8.2.

The requirements from 8.4.2.2 to 8.4.2.3 refer to the Application Requirements Specification.

8.4.2.2 The requirements for the specific application shall include the requirements which are specific to the data of the installation under consideration, as well as a recap or reference to the application conditions of the software and the support tools, and the standards with which the application data shall comply.

8.4.2.3 The requirements related to the application data processed by the software of the system shall be specified at this stage.

8.4.2.4 An Application Data Verification Report shall be written, under the responsibility of the Verifier, on the basis of the input documents from 8.2.

The requirements in 8.4.2.5, 8.4.4.4, 8.4.4.8 and 8.4.5.5 refer to the Application Data Verification Report.

8.4.2.5 Once the Application Requirements Specification has been established, verification shall address

a) that the Application Requirements Specification meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17 as well as the specific requirements in 8.4.2.2 to 8.4.2.3,

b) the internal consistency of the Application Requirements Specification.

The results shall be recorded in the Application Data Verification Report.

8.4.3 Architecture and Design

The quantity and type of the hardware and software components to be used in the specific application shall be specified. The location of components and application data in the specific application architecture shall be defined. The application data processed by the software shall be designed at this stage.

The results of these activities shall be recorded in the output document Application Architecture and Design.

8.4.4 Application data production

8.4.4.1 Intentionally left blank

8.4.4.2 An Application Test Report shall be written, under the responsibility of the Tester, on the basis of the Application Test Specification (see 8.4.4.5).

The requirement in 8.4.4.3 refers to the Application Test Report.

8.4.4.3 The Application Test Report shall document the correct and complete execution of the tests defined in Application Test Specification.

8.4.4.4 To the extent required by the software integrity level (see Table A.1), the Application Data Verification Report shall

a) document every activity performed to ensure correctness and completeness of application data and their coherency with application principles and specific application architecture,

b) evaluate compatibility of data with conditions from the generic application.

8.4.4.5 An Application Test Specification shall be written, under the responsibility of the Tester, on the basis of the input documents from 8.2.

The requirement in 8.4.4.6 refers to the Application Test Specification.

8.4.4.6 The Application Test Specification shall specify tests to be carried out at intermediate or final stage of application data preparation, in order to ensure

a) coherency and completeness of application data with respect to application principles,

b) coherency and completeness of application data with respect to specific application architecture.

8.4.4.7 Intentionally left blank

8.4.4.8 Once the Application Test Specification has been established, verification shall address

a) that the Application Test Specification meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17 as well as the specific requirements in 8.4.4.6,

b) the internal consistency of the Application Test Specification.

The results shall be recorded in the Application Data Verification Report.

8.4.5 Application integration and testing

8.4.5.1 For some systems the application data can be integrated with the hardware and software for a factory test before installation on the target system. This may not be necessary where a sufficient degree of confidence can be obtained by other means. The application shall then be installed on the target system, and integration tests within the complete installation shall be carried out. Finally the target system shall be commissioned as a fully operational system, and the overall testing of the target system in the complete installation shall be carried out. The Application Integration Test Report shall document the correct and complete execution of tests defined in the Application Test Specification. The Application Data Verification Report shall check the completeness and correctness of tests performed on the complete installation.

8.4.5.2 An Application Integration Test Specification shall be written, under the responsibility of the Tester, on the basis of the input documents from 8.2.

The requirement in 8.4.5.3 refers to the Application Integration Test Specification.

8.4.5.3 The Application Integration Test Specification shall specify tests to be carried out to ensure

a) correct integration of application data on hardware and software, if needed,

b) correct integration of application data with complete installation.

NOTE While the Application Data Production testing activities are typically carried out on a simulated environment, the Application Integration and Testing ones are carried out in a more representative environment, e.g. complete installation / on-site.

8.4.5.4 Intentionally left blank

8.4.5.5 Once the Application Integration Test Specification has been established, verification shall address that the Application Test Specification meets the specific requirements in 8.4.5.3.

The results shall be recorded in the Application Data Verification Report.

8.4.5.6 An Application Integration Test Report shall be written, under the responsibility of the Tester, on the basis of the Application Integration Test Specification (see 8.4.5.2).

The requirement in 8.4.5.7 refers to the Application Test Report.

8.4.5.7 The Application Test Report shall document the correct and complete execution of the tests defined in Application Integration Test Specification.

8.4.6 Application validation and assessment

Validation and assessment activities shall audit the performance of each stage of the life-cycle. Assessment is not required for Basic Integrity.

8.4.7 Application preparation procedures and tools

8.4.7.1 Intentionally left blank

8.4.7.2 Intentionally left blank

8.4.7.3 All application data and associated documentation for each specific application shall be subject to the software deployment requirements as specified in 9.1.

8.4.7.4 All application data and associated documentation shall be subject to the software maintenance requirements specified in 9.2.

8.4.7.5 All application data and associated documentation shall be placed under configuration management according to the requirements specified in 6.5 and 6.7. The configuration management of application data can be separate from the software part.

8.4.7.6 The Application Preparation Verification Report shall demonstrate the coverage and enforcement of the application conditions of the software and support tools.

8.4.7.7 An Application Release Note which accompanies the delivered application data shall be written, under the responsibility of the Designer, on the basis of the input documents from 8.2.

The requirement in 8.4.7.8 refers to the Application Release Note.

8.4.7.8 An Application Release Note shall be provided that defines

a) the application conditions which shall be adhered to,

b) information of compatibility among software components (including application data) and between software and hardware,

c) all restrictions in using the software (see 7.7.4.12).

8.4.7.9 Once the Application Release Note has been established, verification shall address

a) that the Application Release Note meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17 as well as the specific requirement in 8.4.7.8,

b) the internal consistency of the Application Release Note.

The results shall be recorded in the Application Preparation Verification Report.

9 Software deployment and maintenance

9.1 Software deployment

9.1.1 Objective

To ensure that the software performs as required, preserving the required software integrity level and dependability when it is deployed in the final environment of application.

9.1.2 Input documents

All design, development and analysis documents relevant to the deployment.

9.1.3 Output documents

1) Software Deployment Manual

2) Deployment Records

3) Deployment Verification Report

9.1.4 Requirements

9.1.4.1 The deployment shall be carried out under the responsibility of the Project Manager.

9.1.4.2 Before delivering a software release, the software baseline shall be recorded and kept traceable under configuration management control, including pre-existing and previously developed software.

9.1.4.3 The software release shall be reproducible throughout the baseline lifecycle.

9.1.4.4 Intentionally left blank

9.1.4.5 Intentionally left blank

9.1.4.6 A Software Deployment Manual shall be written on the basis of the input documents from 9.1.2.

The requirement in 9.1.4.7 refers to the Software Deployment Manual.

9.1.4.7 The Software Deployment Manual shall define procedures in order to correctly identify and install a software release.

9.1.4.8 In case of incremental deployment (i.e. deployment of single components), it is highly recommended for SIL 3 and SIL 4, and recommended for SIL 1 and SIL 2, that the software is designed to include facilities which ensure that activation of incompatible versions of software components is excluded.

9.1.4.9 Configuration management shall ensure that no harm results from the co-presence of different versions of the same software components where it cannot be avoided.

The change control procedures shall ensure that any amendment to the generic software may only be installed after it has been established that either the revised software is compatible with the original application data or the application data have been revised.

9.1.4.10 A rollback procedure (i.e. capability to return to the previous release) shall be available when installing a new software release.

9.1.4.11 The software shall have embedded self-identification mechanisms, allowing its identification at the loading process and after loading into the target. The self-identification mechanism should indicate version information for the software and any configuration data as well as the product identity.

The data within the code, containing the information about the software release, is recommended to be protected through coding (see Table A.3 “Error Detecting Codes”).

9.1.4.12 A Deployment Record shall be written on the basis of the input documents from 9.1.2.

The requirement in 9.1.4.13 refers to the Deployment Record.

9.1.4.13 A Deployment Record shall give evidence that intended software has been loaded, by inspection of the embedded self-identification mechanisms (see 9.1.4.11). This record shall be stored among the delivered system related documents like other verifications and is part of the commissioning and acceptance.

9.1.4.14 The deployed software shall be traceable to delivered installations.

NOTE This is of special importance when critical faults are discovered and need to be corrected in more than one installation.

9.1.4.15 Diagnostic information shall be provided by the software, as part of fault monitoring.

9.1.4.16 A Deployment Verification Report shall be written, under the responsibility of the Verifier, on the basis of the input documents from 9.1.2.

Requirements from 9.1.4.17 to 9.1.4.19 refer to the Deployment Verification Report.

9.1.4.17 Once the Software Deployment Manual has been established, verification shall address

a) that the Software Deployment Manual meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17 as well as the specific requirements in 9.1.4.7,

b) the internal consistency of the Software Deployment Manual.

9.1.4.18 Once the Deployment Record has been established, verification shall address

a) that the Deployment Record meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17 as well as the specific requirements in 9.1.4.13,

b) the internal consistency of the Deployment Record.

9.1.4.19 Intentionally left blank

9.1.4.20 Measures shall be included in the software package to prevent or detect errors occurring during storage, transfer, transmission or duplication of executable code or data. The executable code

is recommended to be coded (see Table A.3 “Error Detecting Codes”) as part of checking the integrity of the code in the load process.

9.2 Software maintenance

9.2.1 Objective

To ensure that the software performs as required, preserving the required software integrity level and dependability when making corrections, enhancements or adaptations to the software itself. See also 6.6 “Modification and change control”.

9.2.2 Input documents

All relevant design, development and analysis documents

9.2.3 Output documents

1) Software Maintenance Plan

2) Software Change Records

3) Software Maintenance Records

4) Software Maintenance Verification Report

9.2.4 Requirements

9.2.4.1 Although this document is not intended to be retrospective, applying primarily to new developments and only applying in its entirety to existing software if these are subjected to major modifications, this subclause 9.2 concerning software maintenance applies to all changes, even those of a minor nature.

9.2.4.2 For any software integrity level, the supplier shall, before starting work on any change, decide whether the modifications are to be considered as major or minor in order to decide if the existing maintenance methods for the system are adequate. The decision shall be justified and recorded by the supplier and submitted to the validator. For SIL 1 to SIL 4, it shall also be submitted to the Assessor’s evaluation. The Assessor shall record the evaluation and its result in the Software Assessment Report.

9.2.4.3 Maintenance should be carried out under consideration of the guidelines on software maintenance contained in ISO/IEC/IEEE 90003:2018 [19], 8.5.1.7.

9.2.4.4 Maintainability shall be designed as an inherent aspect of the software, in particular by following the requirements of 7.3, 7.4 and 7.5.

9.2.4.5 A Software Maintenance Plan shall be written on the basis of the input documents from 9.2.2.

The requirement 9.2.4.6 refers to the Software Maintenance Plan.

9.2.4.6 Procedures for the maintenance of software shall be established and recorded in the Software Maintenance Plan. These procedures shall also address

a) control of error reporting, error logs, maintenance records, change authorization and software/system configuration and the techniques and measures in Table A.10,

b) verification, validation and, for SIL 1 to SIL 4, assessment of any modification,

c) definition of the Authority which approves the changed software, and

d) authorization for the modification.

9.2.4.7 A Software Maintenance Record shall be written on the basis of the input documents from 9.2.2.

The requirement in 9.2.4.8 refers to the Software Maintenance Record.

9.2.4.8 A Software Maintenance Record shall be established for each software item before its first release, and it shall be maintained. The Maintenance Record shall include

a) references to all the Software Change Records for that software item,

b) change impact evaluation,

c) test cases, including revalidation and regression testing data, and

d) software configuration history.

9.2.4.9 A Software Change Record shall be written on the basis of the input documents from 9.2.2.

The requirement in 9.2.4.10 refers to the Software Change Record.

9.2.4.10 A Software Change Record shall be established for each maintenance activity. This record shall include

a) the modification or change request, version, nature of fault, required change and source for change,

b) an analysis of the impact of the maintenance activity on the overall system, including hardware, software, human interaction and the environment and possible interactions,

c) the detailed specification of the modification or change carried out, and

d) revalidation, regression testing and re-assessment of the modification or change to the extent required by the software integrity level. The responsibility for revalidation can vary from project to project, according to the software integrity level. Also the impact of the modification or change on the process of revalidation can be confined to different system levels (only changed components, all identified affected components, the complete system). Therefore the Software Validation Plan shall address both problems, according to the software integrity level. The degree of independence of revalidation shall be the same as that for validation.

9.2.4.11 A Software Maintenance Verification Report shall be written, under the responsibility of the Verifier, on the basis of the input documents from 9.2.2.

Requirements from 9.2.4.12 to 9.2.4.14 refer to the Software Maintenance Verification Report.

9.2.4.12 Once the Software Maintenance Plan has been established, verification shall address

a) that the Software Maintenance Plan meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17 as well as the specific requirements in 9.2.4.6,

b) the internal consistency of the Software Maintenance Plan.

9.2.4.13 Once the Software Maintenance Record has been established, verification shall address

a) that the Software Maintenance Record meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17 as well as the specific requirements in 9.2.4.8,

b) the internal consistency of the Software Maintenance Record.

9.2.4.14 Once the Software Change Record has been established, verification shall address

a) that the Software Change Record meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17 as well as the specific requirements in 9.2.4.10,

b) the internal consistency of the Software Change Record.

9.2.4.15 The maintenance activities shall be carried out following the Software Maintenance Plan to the extent required by the software integrity level.

9.2.4.16 The techniques and measures from Table A.10 shall be chosen. The selected combination shall be justified as a set satisfying 4.8 and 4.9.

9.2.4.17 Maintenance shall be performed with at least the same level of expertise, tools, documentation, planning and management as for the initial development of the software. This shall apply also to configuration management, change control, document control, and independence of involved parties.

9.2.4.18 External supplier control, problem reporting and corrective actions shall be managed with the same criteria specified in the relevant paragraphs of the Software Quality Assurance (6.5) as for new software development.

9.2.4.19 For each reported problem or enhancement a safety impact analysis shall be made.

9.2.4.20 For software under maintenance, mitigation actions proportionate to the identified risk shall be taken in order to ensure the overall integrity of the system whilst the reported problems are investigated and corrected.

Annex A
(normative)

Criteria for the selection of techniques and measures

A.1 General

The clauses of this document are associated within this annex to the clause tables (see A.2, Table A.1 to Table A.11) to illustrate the means of achieving conformance. There exist lower level tables as well, the detailed tables (see A.3, Table A.12 to Table A.21), which expand upon certain entries in the clause tables. The detailed tables only apply if the measure or technique was selected from the clause table (e.g. Table A.13 only applies if dynamic testing was selected when applying Table A.5, Table A.6 or Table A.7). There also exists an informative Annex D which is referred to from the clause tables.

Some techniques or measures appear in more than one table in this annex because they are applicable to more than one lifecycle phase. The recommendation for the usage of the techniques or measures may be different for different lifecycle phases.

With each technique or measure in the tables there is a requirement for each software integrity level. In this version of the document, the requirements for software integrity levels 1 and 2 are the same for each technique. Similarly, each technique has the same requirements at software integrity levels 3 and 4. These requirements can be

'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,

'-' this symbol means that the technique or measure has no recommendation for or against being used,

'NR' this symbol 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.

The combination of techniques or measures are to be stated in the Software Quality Assurance Plan or in another document referenced by the Software Quality Assurance Plan with one or more techniques or measures being selected unless the notes attached to the table makes other requirements. These notes can include reference to approved techniques or approved combinations of techniques. If such techniques or combinations of techniques, including all respective mandatory techniques, are used, then the Assessor shall accept them as valid and shall only be concerned that they have been correctly applied. If a different set of techniques is used and can be justified, then the Assessor may find this acceptable.

NOTE The following examples provide further guidance on the application of the clause tables (see A.2) and the detailed tables (see A.3).

— Table A.7 defines HR for “Dynamic Analysis and Testing” for Basic Integrity and refers to Table A.13. But Table A.13 only contains “R” or “-”methods. This means it is highly recommended to apply dynamic analysis and testing but the user of the standard is free to apply the techniques from Table A.13 or any other appropriate technique (see also Clause 4.9).

— Table A.5 has no requirement for “Static Analysis” for Basic Integrity and refers to Table A.19. But Table A.19 contains a “HR” for “Walkthroughs/Design Reviews, This means it is not required to do static analysis but if the user of the standard nevertheless decides to do so, then the HR in Table A.19 applies.

— Table A.7 defines “M” for Performance Testing for SIL 3 and SIL 4, referring to Table A.18. But Table A.18 contains only “HR” methods. This means it is mandatory to do performance testing but it is allowed to apply a technique that is not mentioned in Table A.18, if the rationale for using alternative techniques can be given.

A.2 Clauses tables

Table A.1 — Lifecycle issues and documentation (5.3)

DOCUMENTATION	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
Planning					
1. Software Quality Assurance Plan	HR	HR	HR	HR	HR
2. Software Planning Verification Report	R	HR	HR	HR	HR
3. Software Configuration Management Plan	HR	HR	HR	HR	HR
4. Software Verification Plan	HR	HR	HR	HR	HR
5. Software Validation Plan	HR	HR	HR	HR	HR
Software requirements					
6. Software Requirements Specification	HR	HR	HR	HR	HR
7. Overall Software Test Specification	HR	HR	HR	HR	HR
8. Software Requirements Verification Report	R	HR	HR	HR	HR
Architecture and design					
9. Software Architecture Specification	R	HR	HR	HR	HR
10. Software Design Specification	R	HR	HR	HR	HR
11. Software Interface Specifications	HR	HR	HR	HR	HR
12. Software Integration Test Specification	R	HR	HR	HR	HR
13. Software/Hardware Integration Test Specification	R	HR	HR	HR	HR
14. Software Architecture and Design Verification Report	R	HR	HR	HR	HR
Component Design					
15. Software Component Design Specification	-	HR	HR	HR	HR
16. Software Component Test Specification	-	HR	HR	HR	HR
17. Software Component Design Verification Report	-	HR	HR	HR	HR
Component Implementation and Testing					
18. Software Source Code and supporting documentation	HR	HR	HR	HR	HR
19. Software Component Test Report	-	HR	HR	HR	HR
20. Software Source Code Verification Report	-	HR	HR	HR	HR

DOCUMENTATION	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
Integration					
21. Software Integration Test Report	R	HR	HR	HR	HR
22. Software/Hardware Integration Test Report	R	HR	HR	HR	HR
23. Software Integration Verification Report	R	HR	HR	HR	HR
Overall Software Testing / Final Validation					
24. Overall Software Test Report	HR	HR	HR	HR	HR
25. Software Validation Report	HR	HR	HR	HR	HR
26. Tools Validation Report	-	HR	HR	HR	HR
27. Release Note	HR	HR	HR	HR	HR
Systems configured by application data					
28. Application Requirements Specification	HR	HR	HR	HR	HR
29. Application Preparation Plan	R	HR	HR	HR	HR
30. Application Test Specification	HR	HR	HR	HR	HR
31. Application Architecture and Design	R	HR	HR	HR	HR
32. Application Preparation Verification Report	-	HR	HR	HR	HR
33. Application Test Report	HR	HR	HR	HR	HR
34. Source Code of Application Data	HR	HR	HR	HR	HR
35. Application Data Verification Report	R	HR	HR	HR	HR
36. Application Integration Test Specification	HR	HR	HR	HR	HR
37. Application Integration Test Report	HR	HR	HR	HR	HR
38. Application Release Note	HR	HR	HR	HR	HR
Software deployment					
39. Software Deployment Manual	R	HR	HR	HR	HR
40. Deployment Records	R	HR	HR	HR	HR
41. Deployment Verification Report	R	HR	HR	HR	HR
Software maintenance					
42. Software Maintenance Plan	R	HR	HR	HR	HR
43. Software Change Records	HR	HR	HR	HR	HR
44. Software Maintenance Records	R	HR	HR	HR	HR
45. Software Maintenance Verification Report	R	HR	HR	HR	HR
Software assessment					
46. Software Assessment Plan	-	HR	HR	HR	HR
47. Software Assessment Report	-	HR	HR	HR	HR
For Basic Integrity, the content of the software verification plan can be limited to 6.2.4.9 a), b), c), d), f). For Basic Integrity the software interface specification is only highly recommended for the boundary of the overall software (see also 7.3.4.18).					

DOCUMENTATION	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
NOTE 1 According to 5.3.2.12 and 5.3.2.13, documents can be combined differently. NOTE 2 Documents 29, 30 and 31 being HR or R depends on the importance defined in the process and where the verification takes place, e.g. data can only be needed to be verified but tested in the system domain while more functional properties need both test and verification. In this case HR has been marked but can be optional R. NOTE 3 Software Architecture Specification and Software Design Specification for Basic Integrity is important to establish especially for long-term maintenance, selection can consider this (e.g. adding these information within Software Requirements specification).					

Table A.2 — Software Requirements Specification (7.2)

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Modelling	Table A.17	R	R	R	HR	HR
2. Structured methodology	D.52	R	R	R	HR	HR
3. Decision Tables	D.13	R	R	R	HR	HR
Requirements: 1) The Software Requirements Specification shall include a description of the problem in natural language and any necessary formal or semiformal notation. 2) The table reflects additional requirements for defining the specification clearly and precisely. One or more of these techniques shall be selected to satisfy the Software Integrity Level being used.						

Table A.3 — Software architecture (7.3)

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Defensive Programming	D.14	-	HR	HR	HR	HR
2. Fault Detection and Diagnosis	D.26	-	R	R	HR	HR
3. Error Correcting Codes	D.19	-	-	-	-	-
4. Error Detecting Codes	D.19	-	R	R	HR	HR
5. Failure Assertion Programming	D.24	-	R	R	HR	HR
6. Safety Bag Techniques	D.47	-	R	R	R	R
7. Diverse Programming	D.16	-	R	R	HR	HR
8. Recovery Block	D.44	-	R	R	R	R
9. Backward Recovery	D.5	-	NR	NR	NR	NR
10. Forward Recovery	D.30	-	NR	NR	NR	NR
11. Retry Fault Recovery Mechanisms	D.46	-	R	R	R	R
12. Memorising Executed Cases	D.36	-	R	R	HR	HR
13. Artificial Intelligence and Machine Learning	C.3	-	NR	NR	NR	NR
14. Dynamic Reconfiguration	D.17	-	NR	NR	NR	NR
15. Software Error Effect Analysis	D.25	-	R	R	HR	HR
16. Graceful Degradation	D.31	-	R	R	HR	HR
17. Fully Defined Interface	D.38	HR	HR	HR	M	M

18. Modelling	Table A.17	R	R	R	HR	HR
19. Structured Methodology	D.52	R	HR	HR	HR	HR
Requirements: 1) Approved combinations of techniques for software integrity levels 3 and 4 are as follows: a) 1, 7, 17, 19 and one from 4, 5, 12 or 18; b) 1, 4, 17, 19 and one from 2, 5, 12, 15 or 18. 2) Approved combinations of techniques for software integrity levels 1 and 2 are as follows: 1, 17, 19 and one from 2, 4, 5, 7, 12, 15 or 18. 3) Some of these issues may be defined at the system level. 4) Error detecting codes may be used in accordance with the requirements of EN 50159. 5) For Basic Integrity, fully defined interfaces are only highly recommended at the boundary of the overall software.						
NOTE 1 Technique/measure 17 is for external interfaces. NOTE 2 The techniques/measures are general and independent from processing technology (e.g. instruction set, parallelism/pipelining, cache levels, number of cores)						

Table A.4 — Software design and implementation (7.3, 7.4 and 7.5)

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Modelling	Table A.17	R	HR	HR	HR	HR
2. Structured methodology	D.52	R	HR	HR	HR	HR
3. Modular Approach	D.38	HR	M	M	M	M
4. Components	Table A.20	HR	HR	HR	HR	HR
5. Design and Coding Standards	Table A.12	HR	HR	HR	M	M
6. Analysable Programs	D.2	HR	HR	HR	HR	HR
7. Structured Programming	D.53	R	HR	HR	HR	HR
8. Suitable Programming Languages	Table A.15	R	HR	HR	HR	HR
Requirements: 1) An approved combination of techniques for software integrity levels 3 and 4 is 1, 3, 4, 5. 2) An approved combination of techniques for software integrity levels 1 and 2 is 2, 3, 4, 5 and one from 7 or 8.						

Table A.5 — Software component analysis and testing (6.2 and 7.4)

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Formal Proof	D.29	-	R	R	HR	HR
2. Static Analysis	Table A.19	-	HR	HR	HR	HR
3. Dynamic Analysis and Testing	Table A.13	HR	HR	HR	M	M
4. Metrics	D.37	-	R	R	R	R
5. Test Coverage for code	Table A.21	-	HR	HR	HR	HR
6. Performance Testing	Table A.18	-	HR	HR	HR	HR
7. Interface Testing	D.34	HR	HR	HR	HR	HR
Requirements: 1) For software integrity levels 3 and 4, the approved combinations of techniques are: 1, 3 and 5; or: 2, 3 and 5. 2) For software integrity levels 1 and 2, the approved combinations of techniques are: 2 and 5; or: 3 and 5. 3) For Basic Integrity, interface testing is only highly recommended for the boundary of the overall software.						

Table A.6 — Software integration analysis and testing (7.3 and 7.6)

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Dynamic Analysis and Testing	Table A.13	HR	HR	HR	HR	HR
2. Performance Testing	Table A.18	-	R	R	HR	HR

Table A.7 — Overall software analysis and testing (6.2 and 7.2)

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Performance Testing	Table A.18	-	HR	HR	M	M
2. Dynamic Analysis and Testing	Table A.13	HR	HR	HR	M	M
3. Modelling	Table A.17	-	R	R	R	R
NOTE At the overall software level the “Dynamic Analysis and Testing” technique is based on Software Requirements Specification (Functional) and is applied on the whole integrated software (Black-box).						

Table A.8 — Intentionally left blank

Table A.9 — Software quality assurance (6.5)

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Compliant with EN ISO 9001:2015	7.1	R	HR	HR	HR	HR
2. Compliant with ISO/IEC/IEEE 90003:2018	7.1	R	R	R	R	R
3. Company Quality System	7.1	M	M	M	M	M
4. Software Configuration Management	D.48	M	M	M	M	M
5. Checklists	D.7	R	HR	HR	HR	HR
6. Traceability	D.58	R	HR	HR	M	M
7. Data Recording and Analysis	D.12	R	HR	HR	M	M
Requirement: 1) This table shall be applied to different roles and all phases.						

Table A.10 — Software maintenance (9.2)

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Impact Analysis	D.32	R	HR	HR	M	M
2. Data Recording and Analysis	D.12	HR	HR	HR	M	M

Table A.11 — Data preparation techniques (8.4)

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Tabular Specification Methods	D.68	R	R	R	R	R
2. Domain specific language	D.71	R	R	R	R	R
3. Simulation	D.42	R	HR	HR	HR	HR
4. Functional testing	Table A.13	M	M	M	M	M
5. Checklists	D.7	R	HR	HR	M	M
6. Fagan inspection	D.23	-	R	R	R	R
7. Formal design reviews	D.56	R	HR	HR	HR	HR
8. Formal proof of correctness (of data)	D.29	-	-	-	HR	HR
9. Walkthrough	D.56	R	R	R	HR	HR
Requirements: 1) For Safety Integrity Level 1 and 2 an approved combination of techniques is 1 and 4. 2) For Safety Integrity Level 3 and 4 the approved combinations of techniques are 1, 4, 5, 7 or 2, 3, 4, 5, 6.						
NOTE The description of the reference D.29 is on programs while technique 8 in this context applies to formal proof of the correctness of data.						

A.3 Detailed tables

Table A.12 — Coding standards

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Coding Standard	D.15	HR	HR	HR	M	M
2. Coding Style Guide	D.15	HR	HR	HR	HR	HR
3. Limited size and complexity of Functions, Subroutines and Methods	D.38	HR	HR	HR	HR	HR
4. Entry/Exit Point strategy for Functions, Subroutines and Methods	D.38	R	HR	HR	HR	HR
5. Defined control of Global Variables	D.38	HR	HR	HR	M	M
Requirements: 1) It is accepted that technique 3 may not be possible for complex components for which further modularization logically is not sensible as complexity depends on the problem.						

Table A.13 — Dynamic analysis and testing

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Test Case Execution from Error Guessing	D.20	R	R	R	R	R
2. Test Case Execution from Error Seeding	D.21	-	R	R	R	R
3. Structure-Based Testing	D.50	-	R	R	HR	HR
4. Test Case Execution from Cause Consequence Diagrams	D.6	-	-	-	R	R
5. Prototyping / Animation	D.43	-	-	-	R	R
6. Test Case Execution from Boundary Value Analysis	D.4	-	HR	HR	HR	HR
7. Equivalence Classes and Input Partition Testing	D.18	R	HR	HR	HR	HR
8. Process Simulation	D.42	R	R	R	R	R
Requirement: 1) The analysis for the test cases is at the relevant level and is based on the specification and/or the specification and the code. 2) The completeness of the simulation will depend upon the extent of the software integrity level, complexity and application.						
NOTE 1 Techniques/measures 1 and 2 are experience-based testing techniques/measures. NOTE 2 Techniques/measures 4, 6 and 7 are specification-based testing techniques/measures (Functional / Black-box testing). NOTE 3 Technique/Measure 3 is typically used at component level to complement Techniques/Measures 6 and 7 in order to achieve the required test coverage.						

Table A.14 — Intentionally left blank

Table A.15 — Suitable Programming Languages

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Documented and defined operational semantics	D.54	R	HR	HR	M	M
2. Supports modular approaches	D.54, D.38	R	HR	HR	HR	HR
3. Supports commenting	D.54	R	HR	HR	HR	HR
4. Strong type system and checking	D.54	R	HR	HR	HR	HR
5. Supports static analysis	D.54, Table A.19	R	HR	HR	HR	HR
6. Supports Testing	D.54	R	HR	HR	HR	HR
7. Supports Test Coverage Metrics	D.54, Table A.21	R	HR	HR	HR	HR
Requirements: 1) The selection of the languages shall be based on the requirements given in 7.3.						
NOTE For information on assessing the suitability of a programming language see entry in D.54 ‘Suitable Programming Languages’						

Table A.16 — Intentionally left blank

Table A.17 — Modelling

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Data Modelling	D.65	R	HR	HR	HR	HR
2. Data Flow Diagrams	D.11	-	HR	HR	HR	HR
3. Control Flow Diagrams	D.66	R	HR	HR	HR	HR
4. Finite State Machines or State Transition Diagrams	D.27	-	HR	HR	HR	HR
5. Petri Nets	D.55	-	HR	HR	HR	HR
6. Decision/Truth Tables	D.13	R	HR	HR	HR	HR
7. Formal Methods	D.28	-	HR	HR	HR	HR
8. Performance Modelling	D.39	-	HR	HR	HR	HR
9. Prototyping/Animation	D.43	-	R	R	R	R
10. Structure Diagrams	D.51	-	HR	HR	HR	HR
11. Sequence Diagrams	D.67	R	HR	HR	HR	HR
12. Cause Consequence Diagrams	D.6	R	R	R	R	R
13. Event Tree Diagrams	D.22	-	R	R	R	R
Requirements: 1) For SIL 1-4, modelling guidance shall be defined and used. 2) For SIL 1-4, at least one of the HR techniques shall be chosen.						

Table A.18 — Performance testing

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

Table A.19 — Static analysis

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Control Flow Analysis	D.8	-	HR	HR	HR	HR
2. Data Flow Analysis	D.10	-	HR	HR	HR	HR
3. Software Error Effect Analysis	D.25	-	R	R	HR	HR
4. Walkthroughs/Design Reviews	D.56	HR	HR	HR	HR	HR
Requirements: 1) Only Design Reviews are Highly Recommended in technique/measure 4 for Basic Integrity.						

Table A.20 — Components

TECHNIQUE/MEASURE	Ref	Basic Integrity	SIL 1	SIL 2	SIL 3	SIL 4
1. Information Hiding / Encapsulation	D.33	R	HR	HR	HR	HR
2. Parameter Number Limit	D.38	R	R	R	R	R
3. Fully Defined Interface	D.38	R	HR	HR	M	M
Requirement: 1) Information encapsulation are only highly recommended if there is no general strategy for data access.						
NOTE Technique/measure 3 is for Internal Interfaces.						

Table A.21 — Test coverage for code

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

Requirements:

1) For SIL 1 to SIL 4, a quantified measure of coverage shall be developed for the test undertaken. This can support the judgment on the confidence gained in testing and the necessity for additional techniques.

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

— 2 and 3; or

— 2 and 4; or

— 5

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

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

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

NOTE 1 Statement coverage is automatically achieved by items 2 to 5.

NOTE 2 The test coverage criteria in this table are used for structure-based (code-based, white box) testing. Techniques/measures for functional (specification-based, black box) testing are given in Table A.13.

NOTE 3 A high percentage of coverage is usually difficult to achieve. The use of test case execution from boundary values (D.4) and equivalence classes and input partition testing (D.18) can enable a sufficient coverage to be achieved with a smaller number of tests.

NOTE 4 The difference between 2 and 3 depends in practice on the level of the programming language and the use of compound conditions. When single conditions are used only, for example as a result of compilation, 2 and 3 are considered identical.

NOTE 5 The coverage objective for the different software components can be also achieved while executing Dynamic Analysis and Testing within Software Integration Analysis and Testing or Overall Software Analysis and Testing phases.

Annex B
(normative)

Key software roles and responsibilities

Table B.1 — Requirements Manager role specification

Role: Requirements Manager
Responsibilities: 1. shall be responsible for specifying the software requirements 2. shall actively manage the Software Requirements Specification 3. shall establish and maintain traceability to and from system level requirements 4. shall ensure the specifications and software requirements are under change and configuration management including state, version and authorization status 5. shall ensure consistency and completeness in the Software Requirements Specification (with reference to user requirements and final environment of application) 6. shall develop and maintain the software requirements documents
Key competencies: 1. shall be competent in requirements engineering 2. shall be experienced in application's domain 3. shall be experienced in safety attributes of application's domain (only for safety-related functions) 4. shall understand the overall role of the system and the environment of application 5. shall understand analytical techniques and outcomes 6. shall understand applicable regulations

Table B.2 — Designer role specification

Role: Designer
Responsibilities: 1. shall transform specified software requirements into acceptable solutions 2. shall own the architecture and downstream solutions 3. shall define or select the design methods and supporting tools 4. shall apply appropriate design principles and standards 5. shall develop component specifications where appropriate 6. shall maintain traceability to and from the specified software requirements 7. shall develop and maintain the design documentation 8. shall ensure design documents are under change and configuration control
Key competencies: 1. shall be competent in engineering appropriate to the application area 2. shall be competent in safety design principles (only for safety-related functions) 3. shall be competent in design analysis and design test methodologies 4. shall be able to work within design constraints in a given environment 5. shall be competent in the problem domain 6. shall understand all the constraints imposed by the hardware platform, the operating system and the interfacing systems 7. shall be competent in application engineering for development of application data

Table B.3 — Implementer role specification

Role: Implementer
Responsibilities: 1. shall transform the design solutions into data/source code/other design representations 2. shall transform source code into executable code/other design representation 3. shall apply safety design principles (only for safety-related functions) 4. shall apply specified data preparation/coding standards 5. shall carry out analysis to verify the intermediate outcome 6. shall integrate software on the target machine 7. shall manage the integration process using the software baselines 8. shall develop and maintain the implementation documents comprising the applied methods, data types, and listings 9. shall maintain traceability to and from design 10. shall maintain the generated or modified data/code under change and configuration control
Key competencies: 1. shall be competent in engineering appropriate to the application area 2. shall be competent in the implementation language and supporting tools 3. shall be capable of applying the specified coding standards and programming styles 4. shall understand all the constraints imposed by the hardware platform, the operating system and the interfacing systems 5. shall be competent in various integration approaches/methodologies and be able to identify the most suitable method or combination of methods in a given context

Table B.4 — Tester role specification

Role: Tester
Responsibilities: 1. shall ensure that test activities are planned 2. shall develop the test specification (objectives and cases) 3. shall ensure traceability of test objectives against the specified software requirements and of test cases against the specified test objectives 4. shall ensure that the planned tests are implemented and specified tests are carried out 5. shall identify deviations from expected results and record them in test reports 6. shall communicate deviations with relevant change management body for evaluation and decision 7. shall document the test results and relevant observations in test reports 8. shall select the software test equipment
Key competencies: 1. shall be competent in the domain where testing is carried out e.g. software requirements, relevant programming languages, software interfaces, operating systems, data, platform, code. 2. shall be competent in various test and verification approaches/methodologies and be able to identify the most suitable method in a given context 3. shall be capable of deriving test cases from given specifications 4. shall have analytical thinking ability and good observation skills tending towards the system level perspective

Table B.5 — Verifier role specification

Role: Verifier
Responsibilities: 1. shall develop a Software Verification Plan (which may include quality issues) stating what needs verification and what type of process (e.g. review, analysis) and test is required as evidence 2. shall check the adequacy (completeness, consistency, correctness, relevance and traceability) of the documented evidence from review, integration and testing with the specified verification objectives 3. shall identify anomalies, evaluate these in risk (impact) terms, record and communicate these to relevant change management body for evaluation and decision 4. shall manage the verification process (review, integration and testing) and ensure independence of activities as required 5. shall develop and maintain records on the verification activities 6. shall develop a Verification Report stating the outcome of the verification activities
Key competencies: 1. shall be competent in the domain where verification is carried out e.g. software requirements, data, code. 2. shall be competent in various verification approaches/methodologies and be able to identify the most suitable method or combination of methods in a given context 3. shall be capable of deriving the types of verification from given specifications 4. shall have analytical thinking ability and good observation skills

Table B.6 — Validator role specification

Role: Validator
Responsibilities: 1. shall develop an understanding of the software's behaviour within the intended system and environment of application 2. shall develop a validation plan and specify the essential tasks and activities for software validation and, for SIL 1 to SIL 4, agree this plan with the assessor 3. shall review the software requirements against the intended environment/use 4. shall review the evidences of all software development activities to ensure the software fulfils all software requirements 5. shall evaluate the conformity of the software process and the developed software against the requirements of this document including the assigned software integrity level 6. shall review the completeness, correctness, consistency and adequacy of the verification and testing 7. shall check the completeness, correctness, consistency and adequacy of test cases and executed tests 8. shall ensure all validation plan activities are carried out 9. shall review and classify all deviations in terms of risk (impact), records and submits to the body responsible for Change Management and decision making 10. shall give a recommendation on the suitability of the software for intended use and indicate any application constraints as appropriate 11. shall capture deviations from the validation plan 12. shall carry out audits, inspections or reviews on the overall project (as instantiations of the generic development process) as appropriate in various phases of development 13. shall review that developed solutions are traceable to the software requirements 14. shall develop a validation report 15. shall give agreement/disagreement for the release of the software
Key competencies: 1. shall be competent in the domain where validation is carried out 2. shall be experienced in safety attributes of application's domain (only for safety-related functions) 3. shall be competent in various validation approaches/methodologies and be able to identify the most suitable method or combination of methods in a given context 4. shall be capable of deriving the types of validation evidence required from given specifications bearing in mind the intended application 5. shall be capable of combining different sources and types of evidence and synthesize an overall view about fitness for purpose or constraints and limitations of the application 6. shall have analytical thinking ability and good observation skills 7. shall have an understanding of the overall software including its application environment

Table B.7 — Assessor role specification

Role: Assessor
Responsibilities: 1. shall develop a system understanding of the software within the intended environment of application 2. shall develop an assessment plan 3. shall evaluate the conformity of the software process and the developed software against the requirements of this document including the assigned SIL 4. shall evaluate the competency of project staff and organization for the software development 5. shall evaluate the verification and validation activities and the supporting evidence 6. shall evaluate the quality management systems adopted for the software development 7. shall evaluate the configuration and change management system and the evidence of its use and application 8. shall identify and evaluate in terms of risk (impact) any deviations from the software requirements in the assessment report 9. shall ensure that the assessment plan is implemented 10. shall carry out safety audits and inspections on the overall development process as appropriate at various phases of development 11. shall give a professional view on the fitness of the developed software for its intended use detailing any constraints, application conditions and observations for risk control as appropriate 12. shall develop an assessment report and maintain records on the assessment process
Key competencies: 1. shall be competent in the domain/technologies where assessment is carried out 2. shall have / strive to continually gain sufficient levels of experience in the safety principles and the application of the principles within the application domain 3. shall be competent to check that a suitable method or combination of methods in a given context have been applied 4. shall be competent in understanding the relevant safety, human resource, technical and quality management processes in fulfilling the requirements of this document 5. shall be competent in assessment approaches/methodologies 6. shall have analytical thinking ability and good observation skills 7. shall be capable of combining different sources and types of evidence and synthesize an overall view about fitness for purpose or constraints and limitations on application 8. shall have overall software understanding and perspective including understanding the application environment 9. shall be able to judge the adequacy of all development processes (like quality management, configuration management, validation and verification processes)

Table B.8 — Project Manager role specification

Role: Project Manager
Responsibilities: 1. shall ensure that the quality management system and independency of roles according to 5.1 are in place for the project and progress is checked against the plans 2. shall allocate sufficient number of competent resources in the project to carry out the essential tasks including safety activities, bearing in mind the requisite independence of roles 3. shall ensure that a suitable validator has been appointed for the project as defined in this document 4. shall be responsible for the delivery and deployment of the software and ensure that safety requirements from the stakeholders are also fulfilled and delivered 5. shall allow sufficient time for the proper implementation and fulfilment of safety tasks 6. shall endorse partial and complete safety deliverables from the development process 7. shall ensure that sufficient records and traceability is maintained in safety related decision making
Key competencies: 1. shall understand quality, competencies, organizational and management requirements of this document 2. shall understand the requirements of the safety process (only for safety-related software) 3. shall be able to weigh different options and understand the impact on safety performance of a decision or selected options 4. shall understand the requirements of the software development process 5. shall understand the requirements of other relevant standards

Table B.9 — Configuration Manager role specification

Role: Configuration Manager
Responsibilities: 1. shall be responsible for the software configuration management plan 2. shall actively manage the configuration management system (including modification and change control) 3. shall establish that all software components are clearly identified and independently versioned inside the configuration management system 4. shall ensure that the Release Notes are published which include incompatible versions of software components, if any
Key competencies: 1. shall be competent in software configuration management (including modification and change control)

Annex C
(informative)

Guidance on software development

C.1 Lifecycle model examples

C.1.1 General remarks

This annex provides additional guidance on creating lifecycle models, especially with respect to possible phase sequences.

A lifecycle model provides a structure for the activities performed on the software during its lifetime. It does so by defining:

— the activities to be performed and their grouping into phases;

— the inputs for specific activities and the outputs to be produced as the result of the specific activities;

— the sequence and dependencies of the activities and phases.

This document defines requirements for the way how activities are performed on the software, and for outputs of these activities. These requirements in 6.4, 7.2 to 7.7, Clauses 8 and 9 are organized in a way that represents groups of activities that conceptually belong together, as well as dependencies between the activities. In this respect, these (sub-)clauses represent abstract phase definitions for a lifecycle model.

The selection and sequencing of applicable activities and phases is explicitly left to the specific software project which is required to produce appropriate plans as defined in 5.3 (the planning phase at the beginning of every project).

A well-defined lifecycle model is also essential for related disciplines such as project management. For example, the completion of specific verification or validation tasks can be used by organizations as decision gates to understand and manage the uncertainties and risks of development to keep control of cost and schedule. However, these aspects are not in the scope of this Annex.

C.1.2 Linear lifecycle models

Linear lifecycle models arrange the applicable activities according to their dependencies in a sequential fashion where every subsequent phase depends on the output of the preceding phase. The planning is done up front for all phases of the project and updated during the development phases as/if needed.

Figure C.1 and Figure C.2 show a visualization of an example lifecycle model for the development, deployment and maintenance of generic software. For the purposes of this annex, the only difference between “Waterfall” and “V” models is that the “V” model shows dependencies of a given phase to all relevant preceding phases more explicitly, while the “Waterfall” model explicitly visualizes the dependency on the preceding phase only.

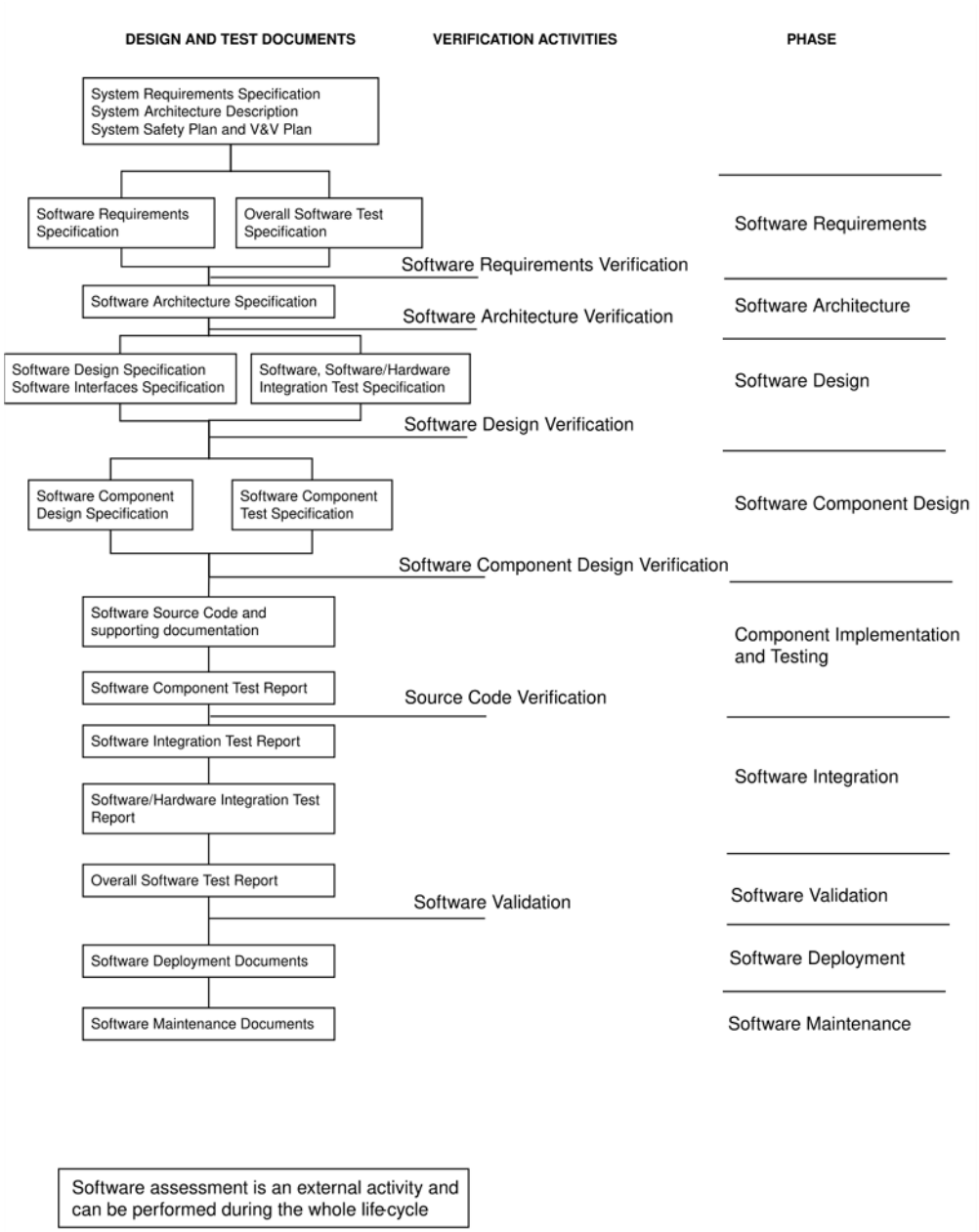


Figure C.1 — Linear lifecycle model example 1 (Waterfall)

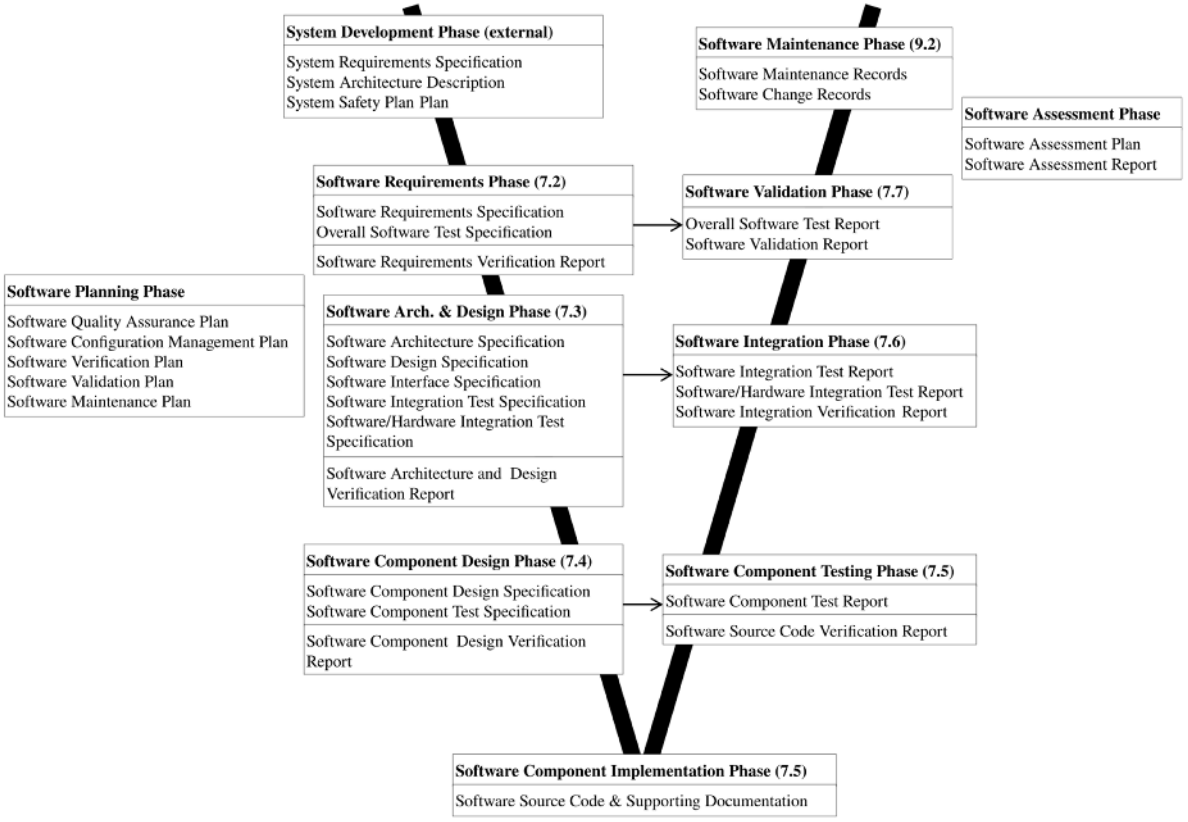


Figure C.2 — Linear lifecycle model example 2 (V model)

Linear lifecycle models implicitly include feedback loops as any change occurring during the execution of the project may trigger a return to one of the preceding phases. For example:

— a review of the Software Architecture might uncover a missing requirement in the Software Requirements Specification

— the results of Component Test might uncover a deficiency in the Component Implementation, Component Design or Software Architecture phase, such as an implementation error or a missing test case in a test specification

— in the most extreme case, a change in the system requirements that occurs during the Overall Software Testing phase might trigger a reexecution of every phase in the project starting with the Software Requirements Specification

However, such repetitions always occur as a result of an unplanned event which triggers an update to the overall project plan.

To model the dependencies between phases, the normative sections of this document implicitly assume a linear lifecycle model (see also 7.1). This reflects the structure and dependencies of activities and their results once software development is completed, which is independent from the realization process used to achieve validated software and documentation.

C.1.3 Iterative lifecycle models

In iterative lifecycle models, activities from one or more of the phases are executed repeatedly to progressively cover the project scope.

NOTE 1 No reference is made within this Annex to specific development frameworks supporting iterative lifecycle models. Guidance is provided on how to tailor the activities to be executed iteratively while meeting the

objectives and requirements of this document, the development framework (e.g. method, practices, tools) to be used is a project choice.

Iterative lifecycle models use the same definitions for phases and corresponding activities. However, the project scope is divided into smaller packages and sequences of one or more abstract phases are performed repeatedly on the subdivisions of the project until the overall goal is achieved.

The general relationship of this document to iterative lifecycle models can be summarized as follows:

— the final phase outputs to be delivered are the same as if they would have been produced in a linear lifecycle, subject to the same design, verification and validation requirements

— whenever outputs (or partial outputs) from a previous iteration are reused in a later iteration, ensure that the reused items haven't been invalidated by any changes that may have occurred in the meantime

In contrast to the linear lifecycle approaches, the repetitions of phases and activities do not only occur as a result of some unplanned event, but are explicitly intended and planned upfront as part of the regular project planning.

Figure C.3 shows an exemplary lifecycle model where the architecture and design, component design, testing and implementation, and integration phases are repeated three times.

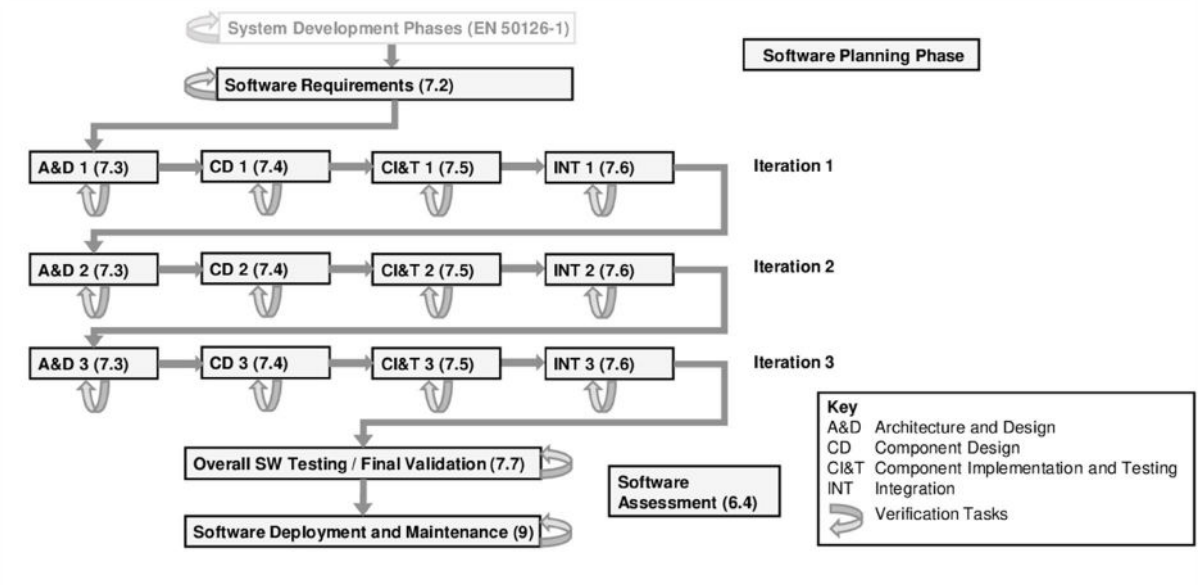


Figure C.3 — Iterative lifecycle model example 1

In detail, the depicted lifecycle model describes the following sequence of activities:

— The software requirements are produced based on the system requirements according to the rules of the subclause 7.2.

— Once the software requirements are completed, the first iteration starts. A portion of the software requirements is derived into the architecture and design, some components are designed, implemented, and integrated, along with appropriate verification measures according to the requirements of subclauses 7.3 to 7.6. A preliminary version of each required output is produced.

— The second and the third iteration follow, implementing further subsets of software requirements. The appropriate outputs are updated, and the impact of the introduced changes on already-available parts is considered.

NOTE 2 The examples shown are limited to two or three iterations for explanatory purposes. Any number of iterations can be defined for a given development, depending on needs of a given project.

The three iterations in this example are not necessarily of the same size (in terms of number of requirements or components), same duration or same complexity. In a typical new software development, the first iteration is likely to focus on architecture groundwork and be rather limited in integration activities. Conversely, the third iteration is likely to only have minor architectural extensions and be more intensive in terms of integration.

Phase activities can also be arranged for parallel execution, Figure C.4 shows an example of a lifecycle model where architecture and design, component design and component implementation and testing are performed in parallel in a second iteration for different components. In this example the second iteration of integration activities is common for the outputs of the parallel branches. However, some integration activities might be performed in parallel where related results are not impacted by concurrent activities being carried out.

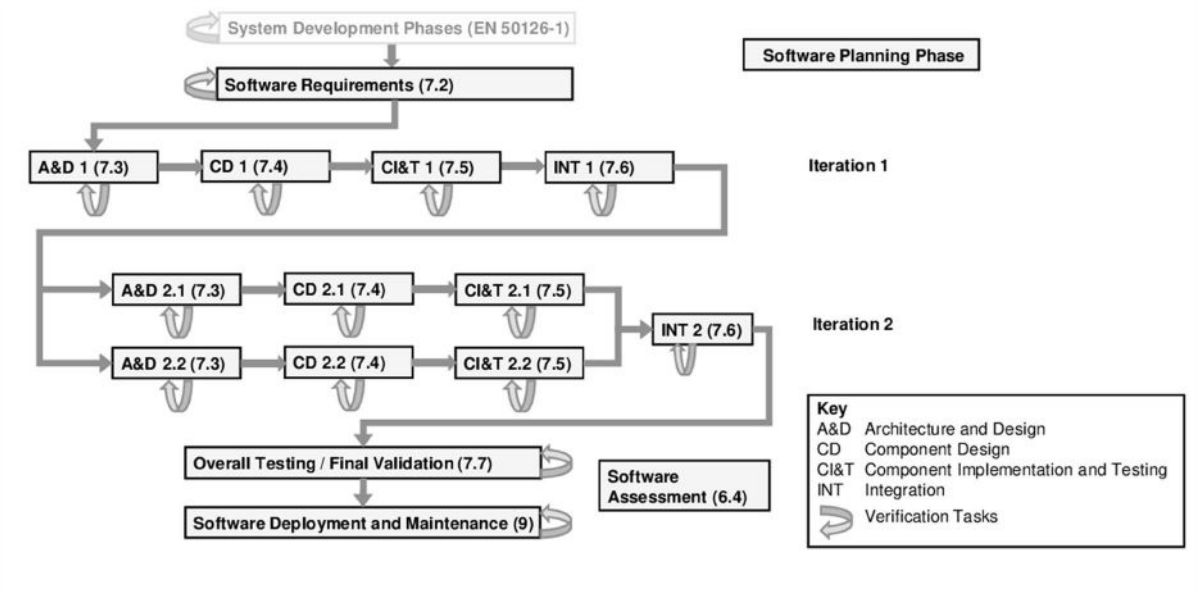


Figure C.4 — Iterative lifecycle model example 2

It is up to the project to define the focus of each iteration and define the sequencing and/or parallel execution of specific activities to achieve its goals. Regardless of the choices on the lifecycle, the outputs after the last iteration have the same content and evidence as if everything would have been produced within a linear lifecycle model.

Figure C.5 provides an example of an iterative development of a Software Architecture Specification and related Verification Report.

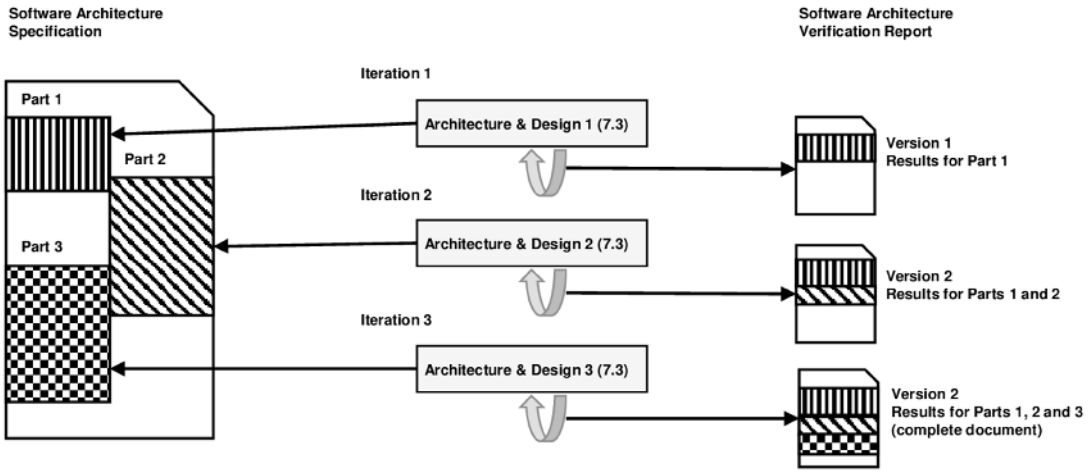


Figure C.5 — Example for iterative development of a work product

Each iteration extends or modifies parts of the documents. The affected parts of the documents may overlap between iterations. In each iteration, a version of the appropriate verification report is produced. Every version of the verification report considers not only the part that is primarily developed in the current iteration but the complete document available to date according to the criteria defined in the normative part of this document (7.3 for the Software Architecture Specification).

NOTE 3 In a well-planned lifecycle model, the activities and the documents are structured in such a way that the additional work necessary to adapt already-existing parts of the documents is minimized (see also 6.5).

Figure C.6 shows a related example that extends the iterative approach to more phases.

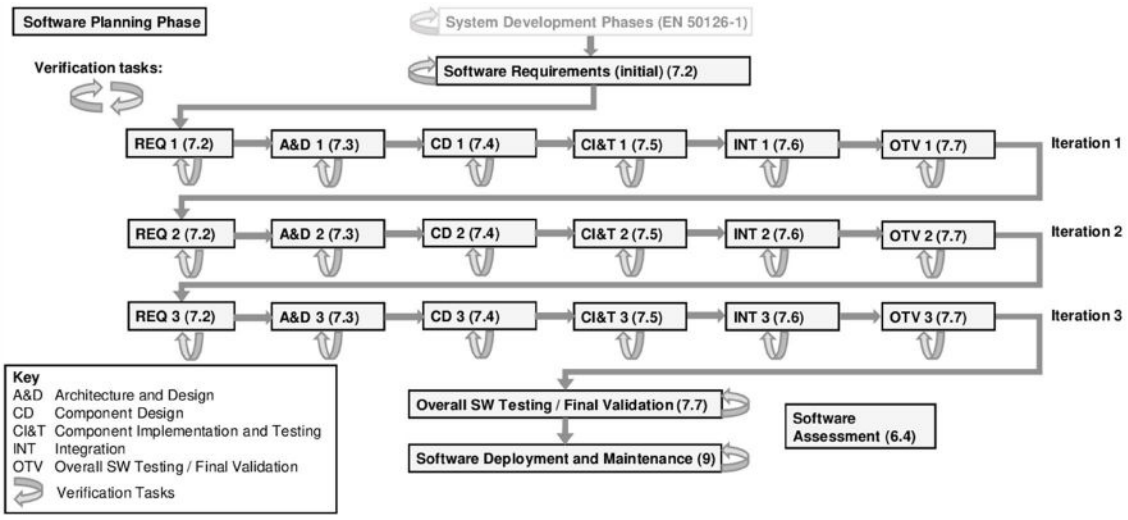


Figure C.6 — Iterative lifecycle model example 3

In the depicted lifecycle model, the Software Requirements Specification is not finalized before the start of the first iteration. Instead, it is only produced and verified to a degree of completion that is sufficient to commence the architecture and design work and the following phases in Iteration 1.

Subsequent iterations extend the Software Requirements Specification up to completion (analysing the impact on previous states of the requirement specifications, including the initial one) and provide input to the architecture and design, as well as subsequent phases. The advantage of this modified lifecycle model is that it reduces the lead time to start of further development activities. It also enables early feedback from later phases on the Software Requirements phase. For example, insights from

developing the software architecture could be considered when working on the software requirements in the Iteration 2. On the other hand, there is an increased risk that a late requirement will affect the architecture in a fundamental way, requiring major changes to nearly-completed work products in the worst case. Estimating and managing these risks and opportunities is a key factor influencing the choice of a lifecycle model.

A similar approach applies to the Overall Software Testing / Final Validation. Instead of waiting for the final software version to be ready to be tested and validated, some activities of this phase are performed at the end of each iteration. As with every other iterative activity, the impact of intermediate changes on existing results from previous iterations are then considered.

Some validation activities are not depending on specific outputs but on the software and the corresponding documentation as a whole. Therefore, performing some validation activities within the iterations generally does not remove the need of a final Overall Software Testing / Final Validation campaign. Similar concepts and constraints would apply to Software Assessment.

C.2 Modelling

C.2.1 Modelling – General

Nowadays, software is ubiquitous in railway. It provides more services to the operator by implementing more functions and by increasing the complexity of existing functions. At the same time, the time schedule is more challenging while software needs to comply with the same or more demanding requirements.

Development only based on requirements in natural language has shown its limitations to cope with these constraints. To improve the situation, formal and semiformal notation can be used in the Software Requirements Specification in addition to natural language (see Requirement 1 in Table A.2). This is a first step toward modelling. Its application to all phases of the development can to be considered further.

This Annex explains possible usage benefits of modelling approaches during development at software level as well as specific aspects to consider. It also provides guidance for a possible application of modelling in software development in compliance with the requirements of the standard.

C.2.2 Modelling – Definition

A model is a logical representation aimed at developing, understanding, communicating, or explaining aspects of a system, entity, or process.

Models are developed using a modelling notation which can be graphical and/or textual and which can be formal (i.e. with an underlying mathematical definition) or semi-formal (i.e. structured notation but incomplete semantics). Modelling notations are typically based on concepts such as classes, block diagrams, control diagrams, or state charts.

NOTE Formal modelling notation include the formal methods listed in D.28. Semi-formal modelling notation includes Unified Modelling Language (UML), Systems Modelling Language (SysML), Domain specific languages (D.71) and Finite State Machines/State Transition Diagrams (D.27).

Modelling notations with well-defined syntax and semantics allow to decrease workload, complexity, project risks and costs through:

— efficient support of variability and variants

— easier evaluation of alternative scenario

— early discovery of defects

— automatic code and document generation

— enhanced verification means (e.g. static analyser, model checker, proof engine)

— reduced tests effort by use of model consistency checks

— built-in simulation capabilities

— integration with support tools (e.g. traceability)

— reuse of pre-existing designs in different context (e.g. different hardware platforms)

— increased consistency leading to improved change management and maintainability

Models themselves may consist of various hierarchical refinement levels or references to refined models at a lower hierarchical level (e.g. hierarchical structure with black box and white box views for each model element).

Models may be used in most phases of software development to specify the software, to represent its abstract or detailed design, to implement the software components, to verify and integrate them, to validate the software etc.

The following sections elaborate on all this, particularly on:

— the usage of modelling on the development lifecycle

— how modelling can contribute to software assurance, especially on verification and testing activities

— the constraints on modelling tools: model editors, automatic code generators, proof engines, model checkers etc.

— the impact of modelling on the different phases of software development

C.2.3 Modelling – Lifecycle issues and documentation

The use of modelling techniques is not supposed to have an impact on the selection of the lifecycle model and the documentation structure for the development of the software. However, models can better integrate phases together and minimize overlaps between them mainly because one given model may contain the deliverables of several phases of the development, reducing duplication and improving consistency. Models can also reduce the need for certain activities, particularly when using formal proofs. On the other hand, models are not easily viewed, edited and printed without dedicated tools. Most lifecycle activities have to be done in dedicated tools and not in standard office ones (e.g. word processors, spreadsheet editors). Even if document generation is possible, a lot of the information contained in the models, the metadata associated to the elements they contain, the capacity to easily navigate between and within models etc. may be lost in the process. It has therefore to be ensured that generated documents (i.e. extracted information) have the appropriate content so that related activities can be performed (e.g. traceability, analysis).

C.2.4 Modelling – Software assurance

C.2.4.1 General

Software assurance largely benefits from modelling, mainly because of the higher formalism of the modelling languages.

C.2.4.2 Software testing

Models supporting testing activities can be used as the comparison basis for testing (i.e. back-to-back), to generate test cases and to simulate the environment of the software under test (i.e. model-in-the-loop).

C.2.4.3 Software verification

As any other documented information, models need to be verified to ensure that:

— The requirements for readability and traceability are met (see 5.3.2.7 to 5.3.2.10 and 6.5.4.14 to 6.5.4.17).

— The requirements set out in the input documents are fulfilled (see 6.2).

— The external constraints are correctly taken into account (see 7.2.4.9).

— The selected techniques and measures are correctly applied (see 4.10).

— The models are complete and consistent (see 6.2).

Models can also efficiently support verification activities. For example, the modelling language can be used to express functional, design or safety properties and then prove that those properties hold on the models (requiring, of course, that the model to verify is formally defined). Verification models can also be defined by the Verifier with an appropriate level of independence as defined in 5.1 to prove that specification, design or implementation models actually behave as expected (model-checking, proof of properties etc.).

C.2.4.4 Software validation

Models need to be validated regarding completeness, consistency, adequacy, correctness and fitness for purpose.

C.2.5 Modelling – Support tools and languages

All tools used when modelling need to comply with the requirements of 6.7. For instance, model editors generally belong to class T1; provers and model-checkers generally belong to class T2; code generators generally belong to class T3. Appropriate techniques, such as diverse redundant tools, can be used to limit the evidence to be provided for each individual tool.

C.2.6 Modelling – Software development (Lifecycle and documentation)

C.2.6.1 General

According to EN ISO 9000 [16], a document is *information* and the medium on which it is contained. Information is not limited to natural language and medium is not limited to paper. Thus, models stored in databases are actual documents. Table A.1 can then be applied as is.

C.2.6.2 Software requirements

The software specification needs to include a description of the problem in natural language and any necessary formal or semiformal notation complying with the techniques of Table A.2. Modelling techniques of Table A.17 can all be used as modelling notations.

C.2.6.3 Architecture and design

Most of the modelling techniques of Table A.17 can be used for the architecture and design of the software, particularly time Petri nets, control flow diagrams, and finite state machines.

Techniques of Table A.4 related to suitable programming languages are either not applicable or need to be adapted (see Table C.1).

Table C.1 — Architecture and design typical adaptation for modelling

TECHNIQUES / MEASURES OF TABLE A.4	TYPICAL ADAPTATION FOR MODELLING
1. Modelling	applicable as is
2. Structured methodology	applicable as is
3. Modular approach	A module is an element of organization of the source code to improve its understanding and separate the concerns. If the source code is not automatically generated, if it's modified after generation or if it's the input of manual analysis, this technique remains applicable.
4. Component	applicable as is
5. Design and coding standards	The design standard shall cover the modelling notation (i.e. it shall encourage good modelling practices and avoid poorly-defined features of the modelling language) and the structure, organization and hierarchy of the model. The coding standard is fully applicable when the source code is not automatically generated, when it's modified after generation or when it's input of manual analysis. Otherwise, the coding standard may be present as part of a validated translator, particularly the rules contributing to the avoidance of poorly-defined features of the programming language.
6. Analysable programs	If analysis are performed on the source code, the technique is applicable as-is. If all or parts of the analysis are made on the models, the models also need to be analysable.
7. Structured programming	This technique is used to limit the structural complexity of the source code. An equivalent technique need then to be similarly applied on models. The technique is also applicable to the source code if specific analysis are done on it.
8. Suitable Programming language	Criteria for programming languages are also applicable to modelling notations. If the source code is automatically generated but neither modified after generation nor the input of subsequent analysis and depending on the guarantees provided by the generator, some of the criteria can be useless.

C.2.6.4 Component design

Most of the modelling techniques of Table A.17 can also be used for software component design. As for architecture and design, depending on the context, techniques of Table A.4 related to suitable programming languages need to be adapted (see above).

C.2.6.5 Component implementation and testing

When using models at software component level, the subclauses applicable to the source code can be contextualized in similar subclauses applicable to models (see Table C.2).

Table C.2 — Component implementation and testing typical adaptation for modelling

SUBCLAUSES	TYPICAL ADAPTATION FOR MODELLING
7.5.4.2 The size and complexity of the developed source code shall be balanced.	7.5.4.2 The size and complexity of the developed model shall be balanced.
7.5.4.3 The Software Source Code shall be readable, understandable and testable.	7.5.4.3 The model shall be readable, understandable and testable.
7.5.4.4 The Software Source Code shall be placed under configuration control before the commencement of documented testing.	7.5.4.4 The model shall be placed under configuration control before the commencement of documented testing.
7.5.4.10 After the Software Source Code and the Software Component Test Report have been established, verification shall address a) the adequacy of the Software Source Code as an implementation of the Software Component Design Specification, b) the correct use of the chosen techniques and measures from Table A.4 as a set satisfying 4.8 and 4.9, c) determining the correct application of the coding standards, d) that the Software Source Code meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17, as well as the specific requirements in 7.5.4.1 to 7.5.4.4, e) the adequacy of the Software Component Test Report as a record of the tests carried out in accordance with the Software Component Test Specification	7.5.4.10 After the model and the Software Component Test Report have been established, verification shall address a) the adequacy of the model as an implementation of the Software Component Design Specification, b) the correct use of the chosen techniques and measures from Table A.4 as a set satisfying 4.8 and 4.9, c) determining the correct application of the modelling standards, d) that the model meets the general requirements for readability and traceability in 5.3.2.7 to 5.3.2.10 and in 6.5.4.14 to 6.5.4.17, as well as the specific requirements in 7.5.4.1 to 7.5.4.4, e) the adequacy of the Software Component Test Report as a record of the tests carried out in accordance with the Software Component Test Specification

If the source code is automatically generated and neither modified after generation nor input of analysis, it is no longer necessary to apply the subclauses listed in the table above, as 6.7 would apply for the automatic code generation tool.

As for architecture and design, techniques of Table A.12 related to coding standards need to be transposed to the modelling notations used (see Table C.3).

Table C.3 — Coding standards techniques / measures typical adaptation for modelling

TECHNIQUES / MEASURES OF TABLE A.12	TYPICAL ADAPTATION FOR MODELLING
1. Coding Standard	Modelling Standard
2. Coding Style Guide	Modelling Style Guide
3. Limited size and complexity of Functions, Subroutines and Methods	Limited size and complexity of model parts
4. Entry/Exit Point strategy for Functions, Subroutines and Methods	Not applicable since models generally don’t have the concept of unconditional jump
5. Defined control of Global Variables	Defined control of input and output data in model parts (e.g. signals, information flows)

Techniques for software component analysis and testing of Table A.5 are directly applicable as-is to modelling except test coverage for code. Indeed, techniques of Table A.21 are very specific to imperative programming languages. Alternative coverage criteria need then to be defined depending on the modelling notation used with the objective to cover all parts of the model (e.g. components, interfaces, data flow, control flow) with the same level as with programming languages.

C.2.6.6 Integration

Techniques for software integration analysis and testing of Table A.6 are all directly as-is applicable to modelling.

C.2.6.7 Overall software testing / Final validation

Techniques for overall software analysis and testing of Table A.7 are all directly as-is applicable to modelling.

C.3 Artificial Intelligence and Machine Learning

C.3.1 General

The term Artificial Intelligence (AI) covers a wide range of disciplines, research fields, applications, and techniques. Machine Learning (ML) is the most relevant of these areas for software which is within the scope of the current standard.

C.3.2 Usage of AI and ML within EN 50716

The lifecycle processes of EN 50716 take system requirements, decomposes them into software requirements and from them the software architecture is developed and elaborated into components which are then implemented. Progressive verification activities throughout this process of traceable, hierarchical decomposition play a major part in ensuring the integrity of the software.

ML implements a software structure which “learns” its behaviour through the application of training data. System requirements are not decomposed so as to be traceable to software architecture and components, and the training data are likely to be an incomplete representation of all possible input states. Consequently, it is not easy to demonstrate the coverage achieved by testing or other verification methods.

ML is still a developing field, especially where techniques for verification and validation are concerned. The main steps in a machine learning process can be summarized as follows:

— Build a training data set;

— Build a model to perform the task (usually a program with initially undetermined parameters);

— Design an algorithm to train the model to perform the task (i.e. by modifying its parameters to improve its performance).

Building the model and designing and implementing the algorithm are conventional software engineering activities. If the ML application is required to meet a specific SIL, then the software of the model and the algorithm would need to be developed in accordance with the relevant requirements of EN 50716.

The trained software produced by machine learning needs to be executed by an electronic system if it is to perform a function within the system, and this system will need its own software in order to provide a platform for the trained software. The platform software is likely to be developed according to relevant requirements of EN 50716.

The overall software will need to achieve an integrity level consistent with the requirements of the function to be performed. It follows that machine learning will not displace the techniques and measures used to develop high integrity systems and software, but these techniques and measures will need to be supplemented and enhanced in order to ensure the integrity of the machine learning process and the trained software which it produces.

C.3.3 Challenges to the use of AI and ML within EN 50716

Many of the successful applications of AI/ML reported to date are concerned with identifying patterns and connections within very large data sets (“Big Data”), e.g. predicting the 3D shape of proteins. The

integration of AI/ML components into products and industrial processes is currently limited to industries that do not have requirements for rigorous software verification.

In the context of applications in the scope of EN 50716, there are four challenges of machine learning which are hard in the sense that it is not yet possible to be confident that they can be solved either by existing techniques or by techniques currently under development. These are:

1) Ensuring that the training data are sufficiently complete and accurate: ML is of value in applications where requirements cannot be fully specified using representations such as truth tables and Boolean logic. The training data has to include enough cases to make it sufficiently unlikely that a situation not represented in the training data will actually occur in practice. The effectiveness of the training data depends on the statistical correlation between the data and reality.

2) Verifying the trained software: because the structure of the trained software is not visible/analysable it is not possible to know what coverage is achieved by testing, and it is also not possible to supplement testing by other means such as static analysis.

3) Validation of approximated functionality: the chief advantage of most of the ML methods lies in the generalization of their intended functions into areas of application not specifically defined in beforehand. Precisely this advantage turns into a challenge when it comes to the aspect of validation, namely the check if the final functionality - which has been statistically approximated by the Machine Learning algorithm - “has learned the correct function”.

4) Adversarial attacks and missing causality: so-called “adversarial attacks”, e.g. small deviations in the picture to be recognized by a ML algorithm where even small changes in the picture can completely change the categorization of the object. This is among others due to the fact that a ML algorithm does not provide causal relations between input and output, but merely statistical correlations.

These challenges have to be overcome if AI/ML techniques are to be included as viable techniques for software development within EN 50716. A significant amount of research is currently being undertaken into verification and validation of AI/ML applications, but any form of generally-recognized good practice has yet to emerge.

Annex D
(informative)

Bibliography of techniques

D.1 Intentionally left blank

D.2 Analysable programs

Aim

To design a program in a way that program analysis is easily feasible.

Description

The intention is to produce programs which are easy to analyse using static analysis methods. In order to achieve this, the rules of structured programming should be followed, for instance:

a) the component control flow should be composed of structured constructs, that is sequences, iterations and selection;

b) the components should be small;

c) the number of possible paths through a component is small;

d) the individual program parts shall be designed so that they are decoupled as far as possible;

e) the relation between the input and output parameters should be as simple as possible;

f) complex calculations should not be used as the basis of branching and loop decisions;

g) branch and loop decisions should be simply related to the component input parameters;

h) boundaries between different types of mappings shall be simple.

D.3 Avalanche/stress testing

Aim

To burden the test object with an exceptionally high workload in order to show that the test object would stand normal workloads easily.

Description

There are a variety of test conditions which can be applied for avalanche/stress testing. Some of these test conditions are listed below:

a) if working in a polling mode then the test object gets many more input changes per time unit than under normal conditions;

b) if working on demands then the number of demands per time unit to the test object is increased beyond normal conditions;

c) if the size of a database plays an important role then it is increased beyond normal conditions;

d) influential devices are tuned to their maximum speed or lowest speed as appropriate;

e) for the extreme cases, all influential factors, as far as is possible, are put to the boundary conditions at the same time.

Under these test conditions the time behaviour of the test object can be evaluated. The influence of load changes can be observed. The correct dimension of internal buffers or dynamic variables, stacks etc can be checked.

D.4 Boundary value analysis

Aim

To remove software errors occurring at parameter limits or boundaries.

Description

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:

a) zero divisor;

b) non-printing control characters;

c) empty stack or list element;

d) null matrix;

e) 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.5 Backward recovery

Aim

To provide correct functional operation in the presence of one or more faults.

Description

If a fault has been detected, the system is reset to an earlier internal state, the consistency of which has been proven before. This method implies saving of the internal state frequently at so-called well-defined checkpoints. This may be done globally (for the complete database) or incrementally (changes only between checkpoints). Then the system needs to compensate for the changes which have taken place in the meantime by using journaling (audit trail of actions), compensation (all effects of these changes are nullified) or external (manual) interaction.

D.6 Cause consequence diagrams

Aim

To model, in a diagrammatic form, the sequence of events that can develop in a system as a consequence of combinations of basic events.

Description

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.7 Checklists

Aim

To provide a stimulus to critical appraisal of all aspects of the system rather than to lay down specific requirements.

Description

A set of questions to be completed by the person performing 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

Aim

To detect poor and potentially incorrect program structures.

Description

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

a) inaccessible code, for instance, unconditional jumps which leaves blocks of code unreachable,

b) 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.9 Intentionally left blank

D.10 Data flow analysis

Aim

To detect poor and potentially incorrect program structures.

Description

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

a) variables that are read before they are written. This is very likely to be an error, and is certainly bad programming practice,

b) variables that are written more than once without being read. This could indicate omitted code,

c) 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 computerized tool where the intended data flows are defined using a structured comment that can be read by the tool.

D.11 Data flow diagrams

Aim

To describe the data flow through a program in a diagrammatic form.

Description

Data Flow Diagrams document how data input is transformed to output, with each stage in the diagram representing a distinct transformation.

The basic components of a data flow diagram include

a) functions, represented by bubbles,

b) data flows, represented by arrows,

c) data stores, represented by open boxes,

d) input/output, represented by special kinds of boxes.

Data flow diagrams describe how an input is transformed to an output. They do not, and should not, include control information or sequencing information. Each bubble can be considered as a stand alone black box which, as soon as its inputs are available, transforms them to its outputs.

One of the principle advantages of data flow diagrams is that they show transformations without making any assumptions about how these transformations are implemented.

The preparation of data flow diagrams is best approached by considering system inputs and working towards system outputs. Each bubble shall represent a distinct transformation – its output should, in some way, be different from its input. There are no rules for determining the overall structure of the diagram and constructing a data flow diagram is one of the creative aspects of system design. Like all design, it is an iterative process with early attempts refined in stages to produce the final diagram.

D.12 Data recording and analysis

Aim

To facilitate software process improvement by recording, validating and analysing relevant data from individual projects and persons. The relevance of the data are determined by the strategic goals of the organization. The goals may be directed towards the evaluation of a particular software development method relative to the claims for it, e.g. with respect to defect prevention effectiveness.

Description

Data recording and analysis constitutes an essential part of software process improvement. The recording of valid data represents an important part of learning more about the software development process and to evaluate alternative software development methods.

Detailed records are maintained during a project, both on a project and individual basis. For instance, an engineer would be required to keep records which could include

a) effort expanded on individual components,

b) testing performed on each component,

c) decisions and their rationale,

d) achievement of project milestones,

e) problems and their solutions.

During and at the conclusion of the project these records can be analysed to establish a wide variety of information. In particular data recording is very important for the maintenance of computer systems as the rationale for certain decisions made during the development project is not always known by the maintenance engineers.

Due to poor planning, data recording often tends to be over-volumed and out of focus. This can be avoided by following the principle that data recording should be driven by goals, questions, and metrics of relevance to what is strategically important to the organization.

In order to achieve the desired accuracy, the data recording and validation process should proceed concurrently with the development, e.g. as part of the configuration control process.

D.13 Decision tables and truth tables

Aim

To provide a clear and coherent specification and analysis of complex logical combinations and relationships.

Description

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.14 Defensive programming

Aim

To produce programs which detect anomalous control flow, data flow or data values during their execution and react to these in a predetermined and acceptable manner.

Description

Many techniques can be used during programming to check for control or data anomalies. These can be applied systematically throughout the programming of a system to decrease the likelihood of erroneous data processing.

Two overlapping areas of defensive techniques can be identified. Intrinsic error-safe software is designed to accommodate its own design shortcomings. These shortcomings may be due to plain error of design or coding, or to erroneous requirements. The following lists some of the defensive techniques:

a) variables should be range checked;

b) where possible, values should be checked for plausibility;

c) parameters to procedures should be type, dimension and range checked at procedure entry.

These first three recommendations help to ensure that the numbers manipulated by the program are reasonable, both in terms of the program function and physical significance of the variables.

Read-only and read-write parameters should be separated and their access checked. Functions should treat all parameters as read-only. Literal constants should not be write-accessible. This helps detect accidental overwriting or mistaken use of variables.

Error tolerant software is designed to 'expect' failures in its own environment or use outside nominal or expected conditions, and behave in a predefined manner. Techniques include the following:

a) input variables and intermediate variables with physical significance should be checked for plausibility;

b) the effect of output variables should be checked, preferably by direct Observation of associated system state changes;

c) the software should check its configuration. This could include both the existence and accessibility of expected hardware and also that the software itself is complete. This is particularly important for maintaining integrity after maintenance procedures.

Some of the defensive programming techniques such as control flow sequence checking, also cope with external failures.

D.15 Coding standards and style guide

Aim

To ensure a uniform layout of the design documents and the produced code, enforce consistent programming and to enforce a standard design method which avoids errors.

Description

Coding Standards are rules and restrictions on a given programming language to avoid potential faults which can be made when using that language.

Coding standard content should include

a) scope and base standard when available,

NOTE For domain specific languages base standards might not be available.

b) procedure for changing the coding standard,

c) analysis of the potential faults and recommended treatment,

d) restrictions to avoid the faults.

Features which make verification difficult and therefore should be avoided in order to ensure availability and reliability are:

a) unconditional jumps excluding subroutine calls;

b) recursion;

c) pointers, heaps or any type of dynamic variables or objects;

d) interrupt handling at source code level;

e) multiple entries or exits of loops, blocks or subprograms;

f) implicit variable initialization or declaration;

g) variant records and equivalence; and

h) procedural parameters.

The following features aim at improving availability and reliability:

i) All variables used are also defined (Definition of variables before usage) and initialized.

j) exception handling

k) error detection and recovery through:

    1) plausibility check on data input,

    2) proper check of loop conditions,

    3) identification and correct handling of constraints,

    4) errors that “can’t” happen can be recovered from:

        i) monitoring indexes out of range,

        ii) monitoring pointers out of range;

    5) watchdog to detect:

        i) Failure of timers,

        ii) Infinite loops,

        iii) Blocking.

l) information redundancy, check sum / bits on transmitted data to detect failures during transmission;

m) information hiding:

    1) object orientation techniques;

    2) abstract data types;

n) functional responses to missing or incorrect data.

Style guidelines deal with issues such as formatting and naming conventions, and although it can be highly subjective, more than anything style affects the readability of your code. The establishment of a common and consistent style for a project will facilitate understanding and maintenance of code developed by more than one programmer, and will make it easier for several people to cooperate in the development of the same program.

D.16 Diverse programming

Aim

Detect and mask residual software design faults during execution of a program, in order to prevent safety critical failures of the system, and to continue operation for high reliability.

Description

In diverse programming a given program specification is implemented N times in different ways. The same input values are given to the N versions, and the results produced by the N versions are compared. If the result is considered to be valid, the result is transmitted to the computer outputs.

The N versions can run in parallel on separate computers, alternatively all versions can be run on the same computer and the results subjected to an internal vote. Different voting strategies can be used on the N versions depending on the application requirements.

a) If the system has a safe state, then it is feasible to demand complete agreement (all N agree) otherwise a fail-safe output value is used. For simple trip systems the vote can be biased in the safe direction. In this case the safe action would be to trip if either version demanded a trip. This approach typically uses only two versions (N = 2).

b) For systems with no safe state, majority voting strategies can be employed. For cases where there is no collective agreement, probabilistic approaches can be used in order to maximize the chance of selecting the correct value, for example, taking the middle value, temporary freezing of outputs until agreement returns, etc.

This technique does not eliminate residual software design faults, but it provides a measure to detect and mask before they can affect safety.

Unfortunately, experiments and analytical studies show that N-version programming is not always as effective as desired. Even if different algorithms are used, diverse software versions too often fail on the same inputs.

Two alternatives to N-version programming are design diversity and functional diversity. Design diversity involves the use of multiple components, each designed in a different way but implementing the same function. Functional diversity involves solving the same problem in functionally different ways. Irrespective of the approach, no effective method to assess the level of diversity is currently available.

D.17 Dynamic reconfiguration

Aim

To maintain system functionality despite an internal fault.

Description

The logical architecture of the system shall be such that it can be mapped onto a subset of the available resources of the system. The architecture needs to be capable of detecting a failure in a physical resource and then remapping the logical architecture back onto the restricted resources left functioning. Although the concept is more traditionally restricted to recovery from failed hardware units, it is also applicable to failed software units if there is sufficient 'run-time redundancy' to allow a software re-try or if there is sufficient redundant data to isolate the failure.

Although traditionally applied to hardware, this technique is being developed for application to software and, thus, the total system. It shall be considered at the first system design stage.

D.18 Equivalence classes and input partition testing

Aim

To test the software adequately using a minimum of test data. The test data are obtained by selecting the partitions of the input domain required to exercise the software.

Description

This testing strategy is based on the equivalence relation of the inputs, which determines a partition of the input domain.

Test cases are selected with the aim of covering all subsets of this partition. At least one test case is taken from each equivalence class.

There are two basic possibilities for input partitioning which are

a) equivalence classes may be defined on the specification. The interpretation of the specification may be either input oriented, for example the values selected are treated in the same way or output oriented, for example the set of values leading to the same functional result, and

b) equivalence classes may be defined on the internal structure of the program. In this case the equivalence class results are determined from static analysis of the program, for example the set of values leading to the same path being executed.

D.19 Error detecting and correcting codes

Aim

To detect and correct errors in sensitive information.

Description

For an information of n bits, a coded block of k bits is generated which enables errors to be detected and corrected. Different types of code include:

a) hamming codes;

b) cyclic codes;

c) polynomial codes;

d) hash codes;

e) cryptographic codes.

D.20 Error guessing

Aim

To remove common programming errors.

Description

Testing experience and intuition combined with knowledge and curiosity about the system under test may add some uncategorized 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.21 Error seeding

Aim

To ascertain whether a set of test cases is adequate.

Description

Some known error types are inserted in the program, and the program is executed with the test cases under test conditions. If only some of the seeded errors are found, the test case set is not adequate. The ratio of found seeded errors to the total number of seeded errors is an estimate of the ratio of found real errors to total number errors. This gives a possibility of estimating the number of remaining errors and thereby the remaining test effort.

$$\frac{\text{Found seeded errors}}{\text{Total number of seeded errors}} = \frac{\text{Found real errors}}{\text{Total number of real errors}}$$

The detection of all the seeded errors may indicate either that the test case set is adequate, or that the seeded errors were too easy to find. The limitations of the method are that, in order to obtain any usable results, the error types as well as the seeding positions shall reflect the statistical distribution of real errors.

If error seeding is used, the location of all errors shall be recorded, and the Validator shall ensure that all seeded errors have been removed before the software release.

D.22 Event tree analysis

Aim

To model, in a diagrammatic form, the sequence of events that can develop in a system after an initiating event, and thereby indicate how serious consequences can occur.

Description

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.

D.23 Fagan inspections

Aim

To reveal errors in all phases of the program development.

Description

A 'formal' audit on quality assurance documents aimed at finding errors and omissions. The inspection process consists of five phases; Planning, Preparation, Inspection, Rework and Follow up. Each of these phases has its own separate objective. The complete system development (specification, design, coding and testing) shall be inspected.

D.24 Failure assertion programming

Aim

To detect residual faults during execution of a software program.

Description

The assertion programming method follows the idea of checking a pre-condition (before a sequence of statements is executed, the initial conditions are checked for validity) and a post-condition (results are checked after the execution of a sequence of statements). If either the pre-condition or the post-condition is not fulfilled, the processing stops with an error.

For example,

assert <pre-condition>

action 1;

:

:

action x;

assert <post-condition>

D.25 SEEA – Software Error Effect Analysis

Aim

To identify software components, their criticality; to propose means for detecting software errors and enhancing software robustness; to evaluate the amount of validation needed on the various software components.

Description

The analysis is done in three phases.

a) 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.

b) Software error analysis:

   The result of this phase is a table listing the following information:

   1) component name;

   2) error considered;

   3) consequences of the error at the module level;

   4) consequences at the system level;

   5) violated safety criterion;

   6) error criticality;

   7) proposed error detection means;

   8) violated criterion if the detection means is implemented;

9) residual criticality if the detection means is implemented.

c) Synthesis:

The synthesis identifies the remaining unsafe scenarios and the validation effort needed given the criticality of each module.

SEEA, being an in-depth analysis carried out by an independent team, is a powerful bug-finding method.

D.26 Fault detection and diagnosis

Aim

To detect faults in a system, which might lead to a failure, thus providing the basis for countermeasures in order to minimize the consequences of failures.

Description

Fault detection is the process of checking a system for erroneous states (which are caused, as explained before, by a fault within the (sub)system to be checked). The primary goal of fault detection is to inhibit the effect of wrong results. A system which delivers either correct results, or no results at all, is called “self checking”.

Fault detection is based on the principles of redundancy (mainly to detect hardware faults) and diversity (software faults). Some sort of voting is needed to decide on the correctness of results. Special methods applicable are assertion programming, N-version programming and the safety bag technique and on hardware level by introducing e.g. sensors, control loops, error checking codes.

Fault detection may be achieved by checks in the value domain or in the time domain on different levels, especially on the physical (e.g. temperature, voltage), logical (error detecting codes), functional (assertions) or external level (plausibility checks). The results of these checks may be stored and associated with the data affected to allow failure tracking.

Complex systems are composed of subsystems. The efficiency of fault detection, diagnosis and fault compensation depends on the complexity of the interactions among the subsystems, which influences the propagation of faults.

Fault diagnosis isolates the smallest subsystem that may be identified. Smaller subsystems allow a more detailed diagnosis of faults (identification of erroneous states).

D.27 Finite state machines/state transition diagrams

Aim

To define or implement the control structure of a system.

Description

Many systems can be defined in terms of their states, their inputs, and their actions. Thus when in state S1, on receiving input I a system might carry out action A and move to state S2. By defining a system's actions for every input in every state we can define a system completely. The resulting model of the system is called a Finite State Machine (FSM). It is often drawn as a so-called state transition diagram showing how the system moves from one state to another, or as a matrix in which the dimensions are state and input and the matrix cells contain the action and new state resulting from receipt of the input in the given state.

Where a system is complicated or has a natural structure this can be reflected in a layered Finite State Machine.

A specification or design expressed as an Finite State Machine can be checked for completeness (the system shall have an action and new state for every input in every state), for consistency (only one state change is defined for each state/input pair) and reachability (whether or not it is possible to get

from one state to another by any sequence of inputs). These are important properties for critical systems and they can be checked. Tools to support these checks are easily written. Algorithms also exist that allow the automatic generation of test cases for verifying a Finite State Machine implementation or for animating a Finite State Machine model.

Several extensions of basic FSMs have been devised to improve the description of complex system behaviour. So called statecharts add hierarchy, composition (parallelism), inter-level transitions, history states, etc. A particularly useful feature is the nesting of internal states and transitions, giving the possibility to reveal or conceal the internal states at need. Statecharts are part of UML (Unified Modelling Language), and as a result supported by many commercial tools.

D.28 Formal methods

Aim

"Formal Methods" refer to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems.

Description

"Mathematically rigorous" means that the specifications used in formal methods are well-formed statements in a mathematical logic and that the formal verifications are rigorous deductions in that logic (i.e. each step follows from a rule of inference and hence can be checked by a mechanical process.) The value of formal methods is that they provide a means to symbolically examine the entire state space of a digital design (whether hardware or software) and establish a correctness or safety property that is true for all possible inputs. However, this is rarely done in practice today (except for the critical components of safety critical systems) because of the enormous complexity of real systems.

Several approaches are used to overcome the astronomically-sized state spaces associated with real systems:

a) apply formal methods to requirements and high-level designs where most of the details are abstracted away;

b) apply formal methods to only the most critical components;

c) analyse models of software and hardware where variables are made discrete and ranges drastically reduced;

d) analyse system models in a hierarchical manner that enables "divide and conquer";

e) automate as much of the verification as possible.

Although the use of mathematical logic is a unifying theme across the discipline of formal methods, there is no single best "formal method". Each application domain requires different modelling methods and different proof approaches. Furthermore, even within a particular application domain, different phases of the lifecycle may be best served by different tools and techniques. For example a theorem prover might be best used to analyse the correctness of a register transfer level description of a Fast Fourier Transform circuit, whereas algebraic derivational methods might best be used to analyse the correctness of the design refinements into a gate-level design.

Examples of Formal Methods (also used for Formal Proof) are: CSP, CCS, HOL, LOTOS, OBJ, Temporal Logic, VDM, Z Method, B Method and Model Checking.

D.29 Formal proof

Aim

Using theoretical and mathematical models and rules it is possible to prove the correctness of a program or model without executing it.

Description

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.

Examples of Formal Methods (also used for Formal Proof) are: CSP, CCS, HOL, LOTOS, OBJ, Temporal Logic, VDM, Z Method, B Method and Model Checking.

D.30 Forward recovery

Aim

To provide correct functional operation in the presence of one or more faults.

Description

If a fault has been detected, the current state of the system is manipulated to obtain a state, which will be consistent some time later. This concept is especially suited for real-time systems with a small database and fast rate of change of the internal state. It is assumed, that at least part of the system state may be imposed onto the environment, and only part of the system states are influenced (forced) by the environment.

D.31 Graceful degradation

Aim

To maintain the more critical system functions available despite failures by dropping the less critical functions.

Description

This technique gives priorities to the various functions to be carried out by the system. The design then ensures that should there be insufficient resources to carry out all the system functions, then the higher priority functions are carried out in preference to the lower ones. For example, error and event logging functions may have lower priority than system control functions, in which case system control would continue if the hardware associated with error logging were to fail.

Another example would be a signalling system where in the event of loss of communication with the control centre the local lineside equipment automatically sets the available routes for the direction taken by the highest priority traffic. This would be a graceful degradation because trains on the priority routes would be able to pass through the area affected by the loss of communication with the control centre, but other movements, such as shunting movements, would not be possible.

D.32 Impact analysis

Aim

To identify the effect that a change or an enhancement to a software will have on other components in that software as well as to other systems.

Description

Prior to a modification or enhancement being performed on the software, an analysis shall be undertaken to identify the impact of the modification or enhancement on the software and to also identify the affected software systems and components.

After the analysis has been completed, a decision is required concerning the reverification of the software system. This depends on the number of components affected, the criticality of the affected components and the nature of the change. The possible decisions are:

a) only the changed components to be reverified;

b) all identified affected components are reverified; and

c) the complete system is reverified.

D.33 Information hiding / encapsulation

Aim

To increase the robustness and maintainability of software.

Description

Data that is globally accessible to all software components can be accidentally or incorrectly modified by any of these components. Any changes to these data structures may require detailed examination of the code and extensive modifications.

Information hiding is a general approach for minimizing these difficulties. The key data structures are 'hidden' and can only be manipulated through a defined set of access procedures. This allows the internal structures to be modified or further procedures to be added without affecting the functional behaviour of the remaining software. For example, a named directory might have access procedures Insert, Delete and Find. The access procedures and internal data structures could be re-written (e.g. to use a different look-up method or to store the names on a hard disk) without affecting the logical behaviour of the remaining software using these procedures.

This concept of an abstract data type is directly supported in a number of programming languages, but the basic principle can be applied whatever programming language is used.

D.34 Interface testing

Aim

To demonstrate that interfaces of subprograms do not contain any errors or any errors that lead to failures in a particular application of the software or to detect all errors that may be relevant.

Description

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

a) all interface variables at their extreme positions,

b) all interface variable individually at their extreme values with other interface variables at normal values,

c) all values of the domain of each interface variable with other interface variables at normal values,

d) all values of all variables in combination (this may only be feasible for small interfaces),

e) 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.35 Intentionally left blank

D.36 Memorizing executed cases

Aim

To force the software to fail safe if it executes an unlicensed path.

Description

During licensing a record is made of all relevant details of each program execution. During normal operation each program execution is compared with the set of the licensed executions. If it differs, a safety action is taken.

The execution record can be the sequence of the individual decision-to-decision paths (DDpaths) or the sequence of the individual accesses to arrays, records or volumes, or both.

Different methods of storing execution paths are possible. Hash-coding methods can be used to map the execution sequence onto a single large number or sequence of numbers. During normal operation the execution path value shall be checked against the stored cases before any output operation occurs.

Since the possible combinations of decision-to-decision paths during one program is very large, it may not be feasible to treat programs as a whole. In this case, the technique can be applied at component level.

D.37 Metrics

Aim

To predict the attributes of programs from properties of the software itself rather than from its development or test history.

Description

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:

a) 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;

b) number of ways to activate a certain component (accessibility): the more a component can be accessed, the more likely it is to be debugged;

c) 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;

d) 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.38 Modular approach

Aim

Decomposition of a software into small comprehensible parts in order to limit the complexity of the software.

Description

A Modular Approach or modularization contains several rules for the design, coding and maintenance phases of a software project. These rules vary according to the design method employed during design. Most methods contain the following rules:

a) a module/component shall have a single well-defined task or function to fulfil;

b) connections between modules/components shall be limited and strictly defined, coherence in one module/component shall be strong;

c) collections of subprograms shall be built providing several levels of modules/components;

d) subprograms shall have a single entry and a single exit only;

e) modules/components shall communicate with other modules/components via their interfaces. Where global or common variables are used they shall be well structured, access shall be controlled and their use shall be justified in each instance;

f) all module/component interfaces shall be fully documented;

g) any modules/components interface shall contain the minimum number of parameters necessary for the modules/components function; and

h) a suitable restriction of parameter number shall be specified, typically 5.

D.39 Performance modelling

Aim

To ensure that the working capacity of the system is sufficient to meet the specified requirements.

Description

The requirements specification includes throughput and response requirements for specific functions, perhaps combined with constraints on the use of total system resources. The proposed system design is compared against the stated requirements by

a) defining a model of the system processes, and their interactions,

b) identifying the use of resources by each process, for example, processor time, communications bandwidth, storage devices, etc),

c) identifying the distribution of demands placed upon the system under average and worst-case conditions,

d) computing the mean and worst-case throughput and response times for the individual system functions.

For simple systems, an analytic solution may be possible whilst for more complex systems, some form of simulation is required to obtain accurate results.

Before detailed modelling, a simpler 'resource budget' check can be used which sums the resources requirements of all the processes. If the requirements exceed designed system capacity, the design is infeasible. Even if the design passes this check, performance modelling may show that excessive delays and response times occur due to resource starvation. To avoid this situation engineers often design systems to use some fraction (e.g. 50 %) of the total resources so that the probability of resource starvation is reduced.

D.40 Performance requirements

Aim

To establish that the performance requirements of a software have been satisfied.

Description

An analysis is performed of both the system and the Software Requirements Specifications to identify all general and specific, explicit and implicit performance requirements.

Each performance requirement is examined in turn to determine

a) the success criteria to be obtained,

b) whether a measure against the success criteria can be obtained,

c) the potential accuracy of such measurements,

d) the project stages at which the measurements can be estimated, and

e) the project stages at which the measurements can be made.

The practicability of each performance requirement is then analysed in order to obtain a list of performance requirements, success criteria and potential measurements. The main objectives are:

a) each performance requirement is associated with at least one measurement;

b) where possible, accurate and efficient measurements are selected which can be used as early in the development process as possible;

c) essential and optional performance requirements and success criteria are identified and

d) where possible, advantage shall be taken of the possibility of using a single measurement for more than one performance requirement.

D.41 Intentionally left blank

D.42 Process simulation

Aim

To test the function of a software, together with its interface to the outside world, without allowing it to modify the real world in any way.

Description

The creation of a system, for testing purposes only, which mimics the behaviour of the system to be controlled by the system under test.

The simulation may be software only or a combination of software and hardware. It shall

i) provide all the inputs of the system under test which will exist when the system is installed,

ii) respond to outputs from the system in a way which faithfully represents the controlled equipment,

iii) have provision for operator inputs to provide any perturbations with which the system under test is required to cope.

When software is being tested, the simulation may be a simulation of the target hardware with its inputs and outputs.

D.43 Prototyping / animation

Aim

To check the feasibility of implementing the system against the given constraints. To communicate the specifier's interpretation of the system to the customer, in order to locate misunderstandings.

Description

A sub-set of system functions, constraints, and performance requirements are selected. A prototype is built using high level tools. At this stage, constraints such as the target computer, implementation language, program size, maintainability and robustness need not be considered. The prototype is evaluated against the customer's criteria and the system requirements may be modified in the light of this evaluation.

D.44 Recovery block

Aim

To increase the likelihood of the program performing its intended function.

Description

Several different program sections are written, often independently, each of which is intended to perform the same desired function. The final program is constructed from these sections. The first section, called the primary, is executed first. This is followed by an acceptance test of the result it calculates. If the test is passed then the result is accepted and passed on to subsequent parts of the system. If it fails, any side effects of the first are reset and the second section, called the first alternative, is executed. This too is followed by an acceptance test and is treated as in the first case. A second, third or even more alternatives can be provided if desired.

D.45 Response timing and memory constraints

Aim

To ensure that the system will meet its temporal and memory requirements.

Description

The requirements specification for the system and the software includes memory and response requirements for specific functions, perhaps combined with constraints on the use of total system resources. An analysis is performed which will identify the distribution demands under average and worst-case conditions. This analysis requires estimates of the resource usage and elapsed time of each system function. These estimates can be obtained in several ways, for example, comparison with an existing system or the prototyping and bench-marking of time critical systems.

D.46 Re-try fault recovery mechanisms

Aim

To attempt functional recovery from a detected fault condition by re-try mechanisms.

Description

In the event of a detected fault or error condition, attempts are made to recover the situation by re-executing the same code. Recovery by re-try can be as complete as a re-boot and a re-start procedure or a small re-scheduling and re-starting task, after a software time-out or a task watchdog action. Re-try techniques are commonly used in communication fault or error recovery and re-try conditions could be flagged from a communication protocol error (e.g. check sum) or from a communication acknowledgement response time-out.

D.47 Safety bag

Aim

To protect against residual specification and implementation faults in software which adversely affect safety.

Description

A safety bag is an external monitor, implemented on an independent computer to a different specification. This safety bag is solely concerned with ensuring the main computer performs safe, not necessarily correct, actions. The safety bag continuously monitors the main computer. The safety bag prevents the system from entering an unsafe state. In addition if it detects that the main computer is entering a potentially hazardous state, the system needs to be brought back to a safe state either by the safety bag or the main computer.

D.48 Software configuration management

Aim

Software Configuration management aims to ensure the consistency of groups of development deliverables as those deliverables change. Configuration Management, in general, applies to both hardware and software development.

Description

Software Configuration Management is a technique used throughout development. In essence, it requires the recording of the production of every version of every “significant” deliverable and of every relationship between different versions of the different deliverables. The resulting records allow the Designer to determine the effect on other deliverables of a change to one deliverable. In particular, systems or subsystems can be reliably re-built from consistent sets of component versions.

D.49 Intentionally left blank

D.50 Structure based testing

Aim

To apply tests which exercise certain subsets of the program structure.

Description

Based on an analysis of the program a set of input data are chosen such that a large fraction of selected program elements are exercised. The program elements exercised can vary depending on the level of rigour required:

a) Statements: this is the least rigorous test since it is possible to execute all code statements without exercising both branches of a conditional statement;

b) Branches: both sides of every branch should be checked. This may be impractical for some types of defensive code;

c) Compound conditions: every condition in a compound conditional branch (i.e. linked by AND/OR) is exercised;

d) LCSAJ (Linear Code Sequence And Jump): a linear code sequence and jump is any linear sequence of code statements including conditional jumps terminated by a jump. Many potential sub-paths will be infeasible due to constraints on the input data imposed by the execution of earlier code.

e) Data flow: the execution paths are selected on the basis of data usage for example a path where the same variable is both written and read.

f) Call graph: a program is composed of subroutines which may be invoked from other subroutines. The call graph is the tree of subroutine invocations in the program. Tests are designed to cover all invocations in the tree.

g) Entire path: execute all possible paths through the code. Complete testing is normally infeasible due to the very large number of potential paths.

D.51 Structure diagrams

Aim

To show the structure of a program diagrammatically.

Description

Structure Diagrams are a notation which complements Data Flow Diagrams. They describe the programming system and a hierarchy of parts and display this graphically, as a tree. They document how elements of a data flow diagram can be implemented as a hierarchy of program units.

A structure chart shows relationships between program units without including any information about the order of activation of these units. They are drawn using the following three symbols:

a) a rectangle annotated with the name of the unit;

b) an arrow connecting these rectangles;

c) a circled arrow, annotated with the name of data passed to and from elements in the structure chart. Normally, the circled arrow is drawn parallel to the arrow connecting the rectangles in the chart.

From any non-trivial data flow diagram, it is possible to derive a number of different structure charts.

Structure charts derived from data flow diagrams represent a first level structure of the system, where each box on the structure chart represents a bubble in the data flow diagram. Naturally, deeper levels can be described using the same technique.

D.52 Structured methodology

Aim

The main aim of Structured Methodologies is to promote the quality of software development by focusing attention on the early parts of the lifecycle. The methods aim to achieve this through both precise and intuitive procedures and notations (assisted by computers) to identify the existence of requirements and implementation features in a logical order and a structured manner.

Description

A range of Structured Methodologies exist. Some such as Structured Systems Analysis and Design Methodology (SSADM) are designed for traditional data-processing and transaction processing functions, while others (Modular Approach to Software Construction, Operation and Test (MASCOT), Jackson System Development Method (JSD), real-time Yourdon) are more oriented to process-control and real-time applications (which tend to be more safety-critical).

Structured Methods are essentially “thought tools” for systematically perceiving and partitioning a problem or system. Their main features are

a) a logical order of thought, breaking a large problem into manageable stages,

b) identification of total system, including the environment as well as the required system,

c) decomposition of data and function in the required system,

d) checklists, i.e. lists of the sort of things that need definition,

e) low intellectual overhead – simple, intuitive, pragmatic.

The supporting notations tend to be precise for identifying problem and system entities (e.g. processes and data flows), but the processing functions performed by these entities tend to be expressed using informal notations. However some methods do make partial use of (mathematically) formal notations (for example JSD makes use of regular expressions: Yourdon, SOM (Service Oriented Modelling) and SDL (Specification and Description Language) utilize finite state machines). This precision not only reduces the scope for misunderstanding, it provides scope for automatic processing.

Another benefit of structured notation is their visibility, enabling a specification or design to be checked intuitively by a user, against his powerful but unstated knowledge.

D.53 Structured programming

Aim

To design and implement the software component in a way which makes practical the analysis of the software component. This analysis should be capable of discovering all significant component behaviour.

Description

The software component should contain the minimum of structural complexity. Complicated branching should be avoided. Loop constraints and branching should (where possible) be simply related to input parameters. The software component should be divided into appropriately small modules, and the interaction of these modules should be explicit. Features of the programming language which encourage the above approach should be used in preference to other features which are (allegedly) more efficient, except where efficiency takes absolute priority (e.g. some safety-critical systems).

D.54 Suitable programming languages

Aim

To support the requirements of this document as much as possible, in particular, defensive programming, strong typing, structured programming and possibly assertions. The programming language chosen should lead to easily verifiable code with a minimum of effort and facilitate program development, verification and maintenance.

Description

Basic Language Requirements

A major purpose of the software development process is to avoid systematic failures i.e. errors in software. In order to systematically avoid software errors, the selection of a programming language needs to form a consistent and suitable set with style guide, analysis tools, compiler as well as testing methods and test tools.

Hence, the software development process is a multi-layered approach for avoiding systematic software errors. Depending on the capabilities of the programming language software errors can be detected rather early by compiler or analysis tools. Where analysis tools and compiler are not sufficient a coding standard and style guide (see D.15) can be used to avoid software errors. The coding standard and style guide are used during programming and checked. Finally, the remaining errors will be found during testing and verification.

Based on the activities required by this software standard a programming language should provide

— Defined Operational Semantics,

  The language should be fully and unambiguously defined. I.e. for each language element it should be exactly defined what will be executed.

— Support modular approaches

  i.e. components, interfaces, scopes (see Tables A.3, A.4 and D.38)

— Support commenting

  e.g. for traceability (see Table A.5)

— Strong type system and checking (see Table A.4 and Table A.15)

The language provides automatic type checking also for user-defined types. Typical examples of languages supporting strong type systems and checking are Pascal, Ada and Modula-2.

— Support static analysis (see Table A.19)

e.g. Boundary Value Analysis, Control Flow Analysis, Data Flow Analysis, Design Reviews

— Support Testing (see Tables A.13, A.18)

  — Interfaces

  — Boundary Value Analysis, Equivalence Classes and Input Partition

  — Structure-Based Testing

  — Performance Modelling: Avalanche/Stress Testing, Response Timing and Memory Constraints, Performance Requirements

— Support Test Coverage Metrics (see Table A.21)

e.g. Statement, Branch, Compound Condition, Data Flow, Path

In addition to the already referenced features the language should provide for

a) block structure,

b) translation time checking,

c) run time type and array bound checking, and

d) parameter checking.

The language should encourage

a) the use of small and manageable components,

b) restriction of access to data in defined components,

c) definition of variable sub-ranges, and

d) any other type of error limiting constructs.

In addition, language justification should be performed and based upon:

a) support of portability of source code to different operating systems or availability of compiler for different operating systems respectively,

b) support of different design methodologies as appropriate for the application e.g. object orientation, state machine, tasking,

c) compliance to standards (e.g. ANSI),

d) availability of certified compiler,

e) no support of, e.g.:

  1) dynamic memory allocation,

  2) unconditional jumps,

3) recursion.

It is desirable that the language is supported by a suitable translator, appropriate libraries of pre-existing components, a debugger and tools for both version control and development.

Low level languages, in particular assembly languages, present problems due to their machine oriented nature.

EN 61131-3 provides definitions and guidance on programming languages for programmable controllers.

Analysis of the Programming Language

Besides defining the functional behaviour of a software a programming language may also provide additional syntax e.g. keywords to enable or facilitate further static analysis to detect typical software errors. However, a programming language may also provide language elements (e.g. pointers in C) which are prone to software errors. Hence, as part of the language selection process the language should be analysed. The results should be used to define the complementing measures in the software development process (e.g. define coding standard and style guide) as outlined above.

The analysis regarding features and vulnerabilities should consider the following aspects:

— Type - Concept / Conversion / Number representation

— Memory Access / Handling

— Concurrency (Threads, Processes, Synchronization)

— Syntax constructs (Operators, Expressions, Precedence)

— Libraries / Runtime Environment

— Features
(Naming, Syntactic sugar, Generics, Meta-Programming, Macros)

Additional guidance may be found in ISO/IEC TR 24772 series.

D.55 Petri nets

Aim

To model relevant aspects of the system behaviour and to assess and possibly improve safety and operational requirements through analysis and re-design.

Description

Petri nets belong to a class of graph theoretic models which are suitable for representing information and control flow in systems exhibiting concurrency and asynchronous behaviour.

A Petri net is a network of places and transitions. The places may be 'marked' or 'unmarked'. A transition is 'enabled' when all the input places to it are marked. When enabled, it is permitted (but not obliged) to 'fire'. If it fires, the input marks are removed, and each output place from the transition is marked instead.

The potential hazards are represented as particular states (markings) in the model. Extended Petri nets allow timing features of the system to be modelled. Although 'classical' Petri nets concentrate on control flow aspects, several extensions have been proposed to incorporate dataflow into the model.

D.56 Walkthroughs / design reviews

Aim

To detect errors in some product of the development process as soon and as economically as possible.

Description

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 are then manually traced through the logic of the program.

D.57 Intentionally left blank

D.58 Traceability

Aim

The objective of Traceability is to ensure that all requirements can be shown to have been properly met and that no untraceable material has been introduced.

Description

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.

D.59 Intentionally left blank

D.60 Intentionally left blank

D.61 Intentionally left blank

D.62 Intentionally left blank

D.63 Intentionally left blank

D.64 Intentionally left blank

D.65 Data modelling

Aim

Creating a data model

Description

Data modelling in computer science is the process of creating a data model by applying formal data model descriptions using data modelling techniques.

A data model in software engineering is an abstract model that describes how data are represented and accessed. Data models formally define data objects and relationships among data objects for a domain of interest. Some typical applications of database models include supporting the development of databases and enabling the exchange of data for a particular area of interest. Data models are specified in a data modelling language.

D.66 Control flow diagram/control flow graph

Aim

Describing the behaviour of a system in a diagrammatic way

Description

In computer science, a control flow diagram or a control flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. Each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

The CFG is essential to many compiler optimizations and static analysis tools.

Reachability is another graph property useful in optimization. If a block/subgraph is not connected from the subgraph containing the entry block, that block is unreachable during any execution, and so is unreachable code; it can be safely removed. If the exit block is unreachable from the entry block, it indicates an infinite loop. Again, dead code and some infinite loops are possible even if the programmer didn't explicitly code that way: optimizations like constant propagation and constant folding followed by jump threading could collapse multiple basic blocks into one, cause edges to be removed from a CFG, etc., thus possibly disconnecting parts of the graph.

terminology

these terms are commonly used when discussing control flow graphs

entry block

block through which all control flow enters the graph

exit block

block through which all control flow leaves the graph

back edge

an edge that points to an ancestor in a depth-first (DFS) traversal of the graph

critical edge

an edge which is neither the only edge leaving its source block, nor the only edge entering its destination block. These edges should be split (a new block should be created in the middle of the edge) in order to insert computations on the edge

abnormal edge

an edge whose destination is unknown. These edges tend to inhibit optimization. Exception handling constructs can produce them

impossible edge

(also known as a fake edge) an edge which has been added to the graph solely to preserve the property that the exit block postdominates all blocks. It cannot ever be traversed

dominator

block M dominates block N if every path from the entry that reaches block N needs to pass through block M. The entry block dominates all blocks

postdominator

block M postdominates block N if every path from N to the exit needs to pass through block M. The exit block postdominates all blocks

immediate dominator

block M immediately dominates block N if M dominates N, and there is no intervening block P such that M dominates P and P dominates N. In other words, M is the last dominator on any path from entry to N. Each block has a unique immediate dominator, if it has any at all

immediate postdominator

analogous to immediate dominator

dominator tree

an ancillary data structure depicting the dominator relationships. There is an arc from Block M to Block N if M is an immediate dominator of N. This graph is a tree, since each block has a unique immediate dominator. This tree is rooted at the entry block

postdominator tree

analogous to dominator tree. This tree is rooted at the exit block

loop header

sometimes called the entry point of the loop, a dominator that is the target of a loop-forming back edge. Dominates all blocks in the loop body

loop pre-header

suppose block M is a dominator with several incoming edges, some of them being back edges (so M is a loop header). It is advantageous to several optimization passes to break M up into two blocks Mpre and Mloop. The contents of M and back edges are moved to Mloop, the rest of the edges are moved to point into Mpre, and a new edge from Mpre to Mloop is inserted (so that Mpre is the immediate dominator of Mloop). In the beginning, Mpre would be empty, but passes like loop-invariant code motion could populate it. Mpre is called the loop pre-header, and Mloop would be the loop header

D.67 Sequence diagram

Aim

Describing the interaction between processes or components in a diagrammatic way.

Description

A sequence diagram is a kind of interaction diagram, that shows how processes or components operate one with another and in what order.

D.68 Tabular specification methods

Aim

The aim is to provide a standardized and well-structured means of defining the data driven functions of a system.

Description

Tabular notations such as signalling control tables are a well-established method of documenting the installation specific requirements for a railway signalling system.

The technique is suitable where the types of relationships between elements of the system are standardized.

Advantage: The format of the table and the possible entries in each field can serve as a checklist during verification.

D.69 Intentionally left blank

D.70 Intentionally left blank

D.71 Domain specific languages

Aim

To represent software programs and related artefacts in a language tailored to a particular domain.

Description

A domain specific language (DSL) is a programming, specification or modelling language created specifically to solve problems in a particular application domain or problem domain, or with a particular technique. The language is based on concepts and features relevant to this domain. Domain specific languages are also known as special-purpose languages, in contrast to general-purpose programming languages or modelling languages like Java and UML.

One of the important benefits of domain specific languages is the possibility to represent and solve problems within a particular domain without the need for knowledge about general programming, specification or modelling. As a consequence, programs, specifications or models can be produced at a higher level, possibly by the end-user. By providing constructs tailored to this domain, and possibly means for automated code generation, a DSL generally also increases the productivity of the designer and the quality of the resulting product. The code generation is typically implemented as an application generator using the DSL as input.

Annex ZZ
(informative)

Relationship between this European Standard and the Essential Requirements of EU Directive (EU) 2016/797 aimed to be covered

This European Standard has been prepared under a Commission's standardization request "M/483 Mandate to CEN and CENELEC for Standardisation in the field of interoperability of the rail system" to provide one voluntary means of conforming to (parts of) Essential Requirements of Directive (EU) 2016/797 of the European Parliament and of the Council of 11 May 2016 on interoperability of the rail system (recast) as specified in the relevant technical specifications for interoperability (TSI).

Once this standard is cited in the Official Journal of the European Union under that Directive, compliance with the normative clauses of this standard given in Table ZZ.1 for Control-Command and Signalling and Table ZZ.2 for locomotive and passenger RST confers, within the limits of the scope of this standard, a presumption of conformity with the corresponding Essential Requirements of that Directive as specified in the technical specifications for interoperability (TSI), and associated EFTA regulations.

Table ZZ.1 — Correspondence between this European Standard, Commission Regulation 2016/919 concerning the technical specification for interoperability relating to the 'control-command and signalling' subsystems of the rail system in the European Union* and Directive (EU) 2016/797

Essential Requirements of Directive (EU) 2016/797	Clauses of the Annex to the Technical Specification for Interoperability (TSI)	Clause/ subclauses of this European Standard	Comments
Section 3 of the Annex to the TSI indicates the correspondence between the TSI clauses and the Essential Requirements of Directive (EU) 2016/797	4.2. Functional and technical specifications of the Subsystems	Clause 4, Clause 5, Clause 6, Clause 7, Clause 8, Clause 9, Annex A, Annex B	This European Standard defines requirements for development of the Subsystems (software aspects)
	4.5.1. Responsibility of the manufacturer of equipment	9.2	Maintenance requirements and procedures necessary for achieving essential requirements (application conditions, restrictions, version compatibility)

*** As amended by Commission Implementing Regulation (EU) 2019/776, Commission Implementing Regulation (EU) 2020/387 and Commission Implementing Regulation (EU) 2020/420**

NOTE The Technical Specification for Interoperability (TSI) can refer to other clauses of this standard making the application of those clauses mandatory. Possible references to such clauses are found in Tables A.3 and A.4 of Annex A to the TSI.

Table ZZ.2 — Correspondence between this European Standard, Commission Regulation (EU) N° 1302/2014 concerning the technical specification for interoperability relating to the 'rolling stock — locomotives and passenger rolling stock' subsystem of the rail system in the European Union* and Directive (EU) 2016/797

Essential Requirements of Directive (EU) 2016/797	Clauses of the Annex to the Technical Specification for Interoperability (TSI)	Clause/ subclauses of this European Standard	Comments
Section 3 of the Annex to the TSI indicates the correspondence between the TSI clauses and the Essential Requirements of Directive (EU) 2016/797	4.2.1.3 (4) Safety aspects	Clause 4, Clause 5, Clause 6, Clause 7, Clause 8, Clause 9, Annex A, Annex B	This European standard defines requirements for development of the Subsystems (software aspects)
	4.5. (5) Maintenance rules	9.2	Maintenance requirements and procedures necessary for achieving essential requirements (application conditions, restrictions, version compatibility)
	6.2.3.5 Conformity assessment for safety requirements	6.4	Requirements in this European Standard support the conformity assessment (software aspects)

*** As amended by Commission Regulation (EU) 2016/919, Commission Implementing Regulation (EU) 2018/868, Commission Implementing Regulation (EU) 2019/776 and Commission Implementing Regulation (EU) 2020/387.**

NOTE The Technical Specification for Interoperability (TSI) can refer to other clauses of this standard making the application of those clauses mandatory. Possible references to such clauses are found in the Appendix J to the TSI.

WARNING 1 — Presumption of conformity stays valid only as long as a reference to this European Standard is maintained in the list published in the Official Journal of the European Union. Users of this standard should consult frequently the latest list published in the Official Journal of the European Union.

WARNING 2 — Other Union legislation may be applicable to the products falling within the scope of this standard.

Bibliography

[1] EN 50126-1:2017, *Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS) - Part 1: Generic RAMS Process*

[2] EN 50126-2:2017, *Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS) - Part 2: Systems Approach to Safety*

[3] EN ISO/IEC 27000:2020, *Information technology - Security techniques - Information security management systems - Overview and vocabulary (ISO/IEC 27000:2018)*

[4] ISO/IEC/TR 19791:2010, *Information technology — Security techniques — Security assessment of operational systems*

[5] IEC 62443 (series), *Industrial communication networks - Network and system security*

[6] IEC 60050-903:2013, *International Electrotechnical Vocabulary - Part 903: Risk assessment*

[7] EN 50155:2017, *Railway applications - Rolling stock - Electronic equipment*

[8] EN 50159, *Railway applications - Communication, signalling and processing systems - Safety-related communication in transmission systems*

[9] EN 61131-3, *Programmable controllers - Part 3: Programming languages (IEC 61131-3)*

[10] EN 61508-2, *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems (IEC 61508-2)*

[11] ISO/IEC Guide 51:2014, *Safety aspects — Guidelines for their inclusion in standards*

[12] IEC 60050-821:2017, *International Electrotechnical Vocabulary - Part 821: Signalling and security apparatus for railways*

[13] EN 50128:2011,1 *Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*

[14] EN 50657:2017, *Railways Applications - Rolling stock applications - Software on Board Rolling Stock*

[15] EN 50129:2018, *Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling*

[16] EN ISO 9000, *Quality management systems - Fundamentals and vocabulary (ISO 9000:2015)*

[17] CLC/TS 50701, *Railway applications - Cybersecurity*

[18] EN ISO 9001:2015, *Quality management systems - Requirements (ISO 9001:2015)*

[19] ISO/IEC/IEEE 90003:2018, *Software engineering — Guidelines for the application of ISO 9001:2015 to computer software*

[1] As impacted by EN 50128:2011/AC:2014, EN 50128:2011/A1:2020 and EN 50128:2011/A2:2020.

[20] EN 50562:2018, *Railway applications - Fixed installations - Process, protective measures and demonstration of safety for electric traction systems*

Norsk
Elektroteknisk
Komite

The Norwegian National Committee of
The International Electrotechnical Commission, IEC
The European Committee for Electrotechnical
Standardization, CENELEC

www.nek.no