INTERNATIONAL STANDARD

ISO 25119-3

Second edition
2018-10

Tractors and machinery for agriculture and forestry — Safety-related parts of control systems —

Part 3: Series development, hardware and software

Tracteurs et matériels agricoles et forestiers — Parties des systèmes de commande relatives à la sécurité —

Partie 3: Développement en série, matériels et logiciels

Reference number
ISO 25119-3:2018(E)

COPYRIGHT PROTECTED DOCUMENT

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/TC 23, *Tractors and machinery for agriculture and forestry*, Subcommittee SC 19, *Agricultural electronics*.

This second edition cancels and replaces the first edition (ISO 25119-3:2010), which has been technically revised. The main changes compared to the previous edition are as follows:

— the introduction has been modified to add specific information on safety standards;

— the prerequisites of functional safety have been specified;

— Clause 5 has been revised to:

  — specify the prerequisites of functional safety, and

  — simplify the general requirements of technical safety concepts;

— additional instructions have been added throughout the document to verify consistency of test specifications and reports;

— Annex B has been changed to a normative annex;

— the document has been editorially revised.

A list of all parts in the ISO 25119 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

ISO 25119 (all parts) sets out an approach to the assessment, design and verification, for all safety life cycle activities, of safety-related parts comprising electrical and/or electronic and/or programmable electronic systems (E/E/PES) on tractors used in agriculture and forestry, and on self-propelled ride-on machines and mounted, semi-mounted and trailed machines used in agriculture. It is also applicable to mobile municipal equipment.

A prerequisite to the application of ISO 25119 (all parts) is the completion of a suitable hazard identification and risk analysis (e.g. ISO 12100) for the entire machine. As a result, an E/E/PES is frequently assigned to provide safety-related functions that create safety-related parts of control systems (SRP/CS). These can consist of hardware or software, can be separate or integrated parts of a control system, and can either perform solely safety-related functions or form part of an operational function.

In general, the designer (and to some extent, the user) will combine the design and validation of these SRP/CS as part of the risk assessment. The objective is to reduce the risk associated with a given hazard (or hazardous situation) under all conditions of use of the machine. This can be achieved by applying various measures (both SRP/CS and non-SRP/CS) with the end result of achieving a safe condition.

ISO 25119 (all parts) allocates the ability of safety-related parts to perform a safety-related function under foreseeable conditions into five performance levels. The performance level of a controlled channel depends on several factors, including system structure (category), the extent of fault detection mechanisms (diagnostic coverage), the reliability of components (mean time to dangerous failure, common-cause failure), design processes, operating stress, environmental conditions and operation procedures. Three types of failures that can cause E/E/PES malfunctions leading to potential hazardous situations are considered: systematic, common-cause and random.

In order to guide the designer during design, verification, and to facilitate the assessment of the achieved performance level, ISO 25119 (all parts) defines an approach based on a classification of architecture with different design features and specific behaviour in case of a fault.

The performance levels and categories can be applied to the control systems of all kinds of mobile machines: from simple systems (e.g. auxiliary valves) to complex systems (e.g. steer by wire), as well as the control systems of protective equipment (e.g. interlocking devices, pressure sensitive devices).

ISO 25119 (all parts) adopts a risk-based approach for the determination of the risks, while providing a means of specifying the required performance level for the safety-related functions to be implemented by E/E/PES safety-related channels. It gives requirements for the whole safety life cycle of E/E/PES (design, validation, production, operation, maintenance, decommissioning), necessary for achieving the required functional safety for E/E/PES that are linked to the performance levels.

The structure of safety standards in the field of machinery is as follows.

a) Type-A standards (basic safety standards) give basic concepts, principles for design and general aspects that can be applied to machinery.

b) Type-B standards (generic safety standards) deal with one or more safety aspect(s), or one or more type(s) of safeguards that can be used across a wide range of machinery:

   — type-B1 standards on particular safety aspects (e.g. safety distances, surface temperature, noise);

   — type-B2 standards on safeguards (e.g. two-hand controls, interlocking devices, pressure sensitive devices, guards).

c) Type-C standards (machinery safety standards) deal with detailed safety requirements for a particular machine or group of machines.

This document is a type-B1 standard as stated in ISO 12100.

This document is of relevance, in particular, for the following stakeholder groups representing the market players with regard to machinery safety:

— machine manufacturers (small, medium and large enterprises);

— health and safety bodies (regulators, accident prevention organizations, market surveillance, etc.).

Others can be affected by the level of machinery safety achieved with the means of the document by the above-mentioned stakeholder groups:

— machine users/employers (small, medium and large enterprises);

— machine users/employees (e.g. trade unions, organizations for people with special needs);

— service providers, e.g. for maintenance (small, medium and large enterprises);

— consumers (in case of machinery intended for use by consumers).

The above-mentioned stakeholder groups have been given the possibility to participate at the drafting process of this document.

In addition, this document is intended for standardization bodies elaborating type-C standards.

The requirements of this document can be supplemented or modified by a type-C standard.

For machines which are covered by the scope of a type-C standard and which have been designed and built according to the requirements of that standard, the requirements of that type-C standard take precedence.

Tractors and machinery for agriculture and forestry — Safety-related parts of control systems —

Part 3: Series development, hardware and software

1 Scope

This document sets out general principles for the design and development of safety-related parts of control systems (SRP/CS) on tractors used in agriculture and forestry and on self-propelled ride-on machines and mounted, semi-mounted and trailed machines used in agriculture. It can also be applied to mobile municipal equipment (e.g. street-sweeping machines).

This document is not applicable to:

— aircraft and air-cushion vehicles used in agriculture;

— lawn and garden equipment.

This document specifies the characteristics and categories required of SRP/CS for carrying out their safety-related functions. It does not identify performance levels for specific applications.

NOTE 1 Machine specific type-C standards can specify performance levels (AgPL) for safety-related functions in machines within their scope. Otherwise, the specification of AgPL is the responsibility of the manufacturer.

This document is applicable to the safety-related parts of electrical/electronic/programmable electronic systems (E/E/PES), as these relate to mechatronic systems. It covers the possible hazards caused by malfunctioning behaviour of E/E/PES safety-related systems, including interaction of these systems. It does not address hazards related to electric shock, fire, smoke, heat, radiation, toxicity, flammability, reactivity, corrosion, release of energy, and similar hazards, unless directly caused by malfunctioning behaviour of E/E/PES safety-related systems. It also covers malfunctioning behaviour of E/E/PES safety-related systems involved in protective measures, safeguards, or safety-related functions in response to non-E/E/PES hazards.

Examples included within the scope of this document:

— SRP/CS's limiting current flow in electric hybrids to prevent insulation failure/shock hazards;

— electromagnetic interference with the SRP/CS;

— SRP/CS's designed to prevent fire.

Examples not included in the scope of this document:

— insulation failure due to friction that leads to electric shock hazards;

— nominal electromagnetic radiation impacting nearby machine control systems;

— corrosion causing electric cables to overheat.

This document is not applicable to non-E/E/PES systems (e.g. hydraulic, mechanic or pneumatic).

NOTE 2 See also ISO 12100 for design principles related to the safety of machinery.

This document is not applicable to safety related parts of control systems manufactured before the date of its publication.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 25119-1, *Tractors and machinery for agriculture and forestry — Safety-related parts of control systems — Part 1: General principles for design and development*

ISO 25119-2:2018, *Tractors and machinery for agriculture and forestry — Safety-related parts of control systems — Part 2: concept phase*

ISO 25119-4:2018, *Tractors and machinery for agriculture and forestry — Safety-related parts of control systems — Part 4: Production, operation, modification and supporting processes*

3 Terms and definitions

For the purposes of this document, the terms and definitions in ISO 25119-1 apply.

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

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

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

4 Abbreviated terms

For the purposes of this document, the following abbreviated terms apply.

AgPL	agricultural performance level
$AgPL_r$	required agricultural performance level
CAD	computer-aided design
Cat	hardware category
CCF	common-cause failure
DC	diagnostic coverage
DC_{avg}	average diagnostic coverage
ECU	electronic control unit
ETA	event tree analysis
E/E/PES	electrical/electronic/programmable electronic systems
EMC	electromagnetic compatibility
FMEA	failure mode and effects analysis
FSM	functional safety management
FTA	fault tree analysis
HARA	hazard analysis and risk assessment
HIL	hardware in the loop

MTTF	mean time to failure
$MTTF_D$	mean time to dangerous failure
$MTTF_{DC}$	mean time to dangerous failure for each channel
PES	programmable electronic system
QM	quality measures
RAM	random-access memory
SOP	start of production
SRL	software requirement level
SRP	safety-related parts
SRP/CS	safety-related parts of control systems
UML	unified modelling language

5 System design

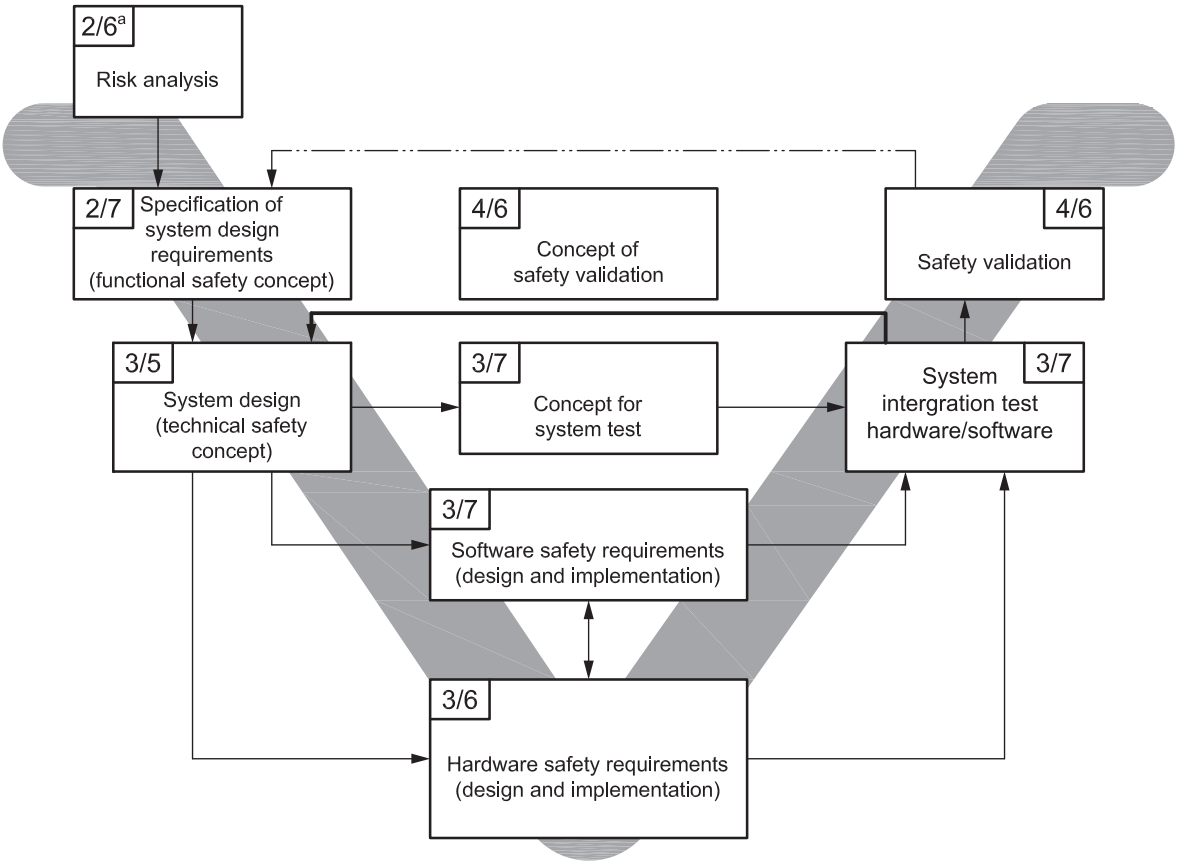
5.1 Objectives

The objective is to define a system design for producing a detail design that fulfils the safety requirements for the entire safety-related system.

5.2 General

Safety requirements constitute all requirements aimed at achieving and ensuring functional safety. During the safety life cycle, safety requirements are detailed and specified in ever greater detail at hierarchical levels. The different levels for safety requirements are illustrated in Figure 1. For the overall representation of the procedure for developing safety requirements, see also 5.4. In order to support management of safety requirements, the use of suitable tools for requirements management is recommended.

Annex A provides guidance for an example agenda of the assessment of functional safety associated with an AgPL = e.



Key

result
verification
validation

[a] The first of two numbers separated by a slash refers to the respective part of ISO 25119, and the second refers to the clause in that document: i.e. 2/6 is ISO 25119-2:2018/Clause 6, 3/5 is ISO 25119-3:2018/Clause 5, and so on.

Figure 1 — Structuring of safety requirements

5.3 Prerequisites

The functional safety concept (ISO 25119-2:2018, Clause 7) is the prerequisite.

5.4 Requirements

5.4.1 Structuring safety requirements

During the functional safety concept phase, functional safety requirements are created to describe the basic functioning of the safety-related system through which the safety goals are to be fulfilled. The basic allocation of functional safety requirements to the system architecture is specified by the technical safety concept specification in the form of technical safety requirements. This system architecture comprises both hardware and software.

The hardware safety requirements refine and solidify the technical safety requirements. Clause 6 describes how to specify the hardware requirements in detail.

The software safety requirements are derived from the requirements of the technical safety concept and the underlying hardware. The requirements for the software defined in Clause 7 shall be taken into account.

This clause specifies the approach to be used in the specification of the technical safety concept requirements during system design, thereby providing a basis for error-free system design.

5.4.2 Technical safety concept

5.4.2.1 General requirements of technical safety concept

The technical safety concept document includes the technical safety requirements for the system.

Each technical safety requirement shall be associated (e.g. by cross-reference) with higher-level safety requirements, which may be:

— other technical safety requirements;

— functional safety requirements.

NOTE Traceability can be greatly facilitated by the use of suitable requirement management tools.

The implementation of each technical safety concept requirement shall take account of feasibility, unambiguousness, consistency and completeness.

The technical safety concept shall take the following into account:

a) all safety goals and functional safety requirements;

b) all relevant norms, standards and statutory regulations;

c) the relevant results from safety analysis tools (FMEA, FTA, etc.); the safety analysis provides iterative support for the technical safety concept during system development.

The completeness of the technical safety concept increases iteratively during system design. To ensure completeness:

1) the version of the technical safety concept and the version of the relevant underlying sources shall be specified;

2) the requirements from change management (see ISO 25119-4:2018, Clause 11) shall be met and, for this reason, the technical safety requirements shall be structured and formulated to provide support for a modification process;

3) the technical safety requirements shall be reviewed (see ISO 25119-4:2018, Clause 6).

The technical safety concept shall consider all phases of the life cycle (including production, customer operation, servicing and decommissioning).

5.4.2.2 Specification of the technical safety concept

5.4.2.2.1 General

The technical safety concept shall include hardware and software safety requirements sufficient for the design of the SRP/CS, and shall be determined in accordance with 5.4.2.1.

5.4.2.2.2 States and times

The behaviour of the SRP/CS, its components and their interfaces shall be specified for all relevant operating states, including

— start-up,

— normal operation,

— shut-down,

— restart after reset, and

— reasonably foreseeable unusual operating states (e.g. degraded operating states).

In particular, failure behaviour and the required reaction shall be described exactly. Additional emergency operation functions may be included.

The technical safety concept shall specify a safe state for each functional safety requirement, the transition to the safe state, and the maintenance of the safe state. In particular, it shall be specified whether shutting off the SRP/CS immediately represents a safe state, or if a safe state can only be attained by a controlled shut down.

The technical safety concept shall specify for each functional safety requirement the maximum time that may elapse between the occurrence of an error and the attainment of a safe state (response time). All response times for subsystems and sub-functions shall be specified in the technical safety concept.

If no safe state can be achieved by a direct shut down, a time shall be defined during which a special emergency operation function has to be sustained for all subsystems and sub-functions. This emergency operation function shall be documented in the technical safety concept.

5.4.2.2.3 Safety architecture, interfaces and marginal conditions

The safety architecture and its sub-components shall be described. In particular, the technical measures shall be specified. The technical safety concept shall separately describe the following components (as applicable):

— sensor system, separate for each physical parameter recorded;

— miscellaneous digital and analogue input and output units;

— processing, separate for each arithmetic unit/discrete logical unit;

— actuator system, separate for each actuator;

— displays, separate for each indicator unit;

— miscellaneous electromechanical components;

— signal transmission between components;

— signal transmission from/to systems external to the SRP/CS;

— power supply.

The interfaces between the components of the SRP/CS, interfaces to other systems and functions in the machine, as well as user interfaces, shall be specified.

Limitations and marginal conditions of the SRP/CS shall be specified. This applies in particular to extreme values for all ambient conditions in all phases of the life cycle.

5.5 Work products

The following work product shall be applicable to system design:

a) technical safety concept specification.

6 Hardware

6.1 Objectives

The objective is to define acceptable hardware architectures for SRP/CS.

6.2 General

Improving the hardware structure of the SRP/CS can provide measures for avoiding, detecting or tolerating faults. Practical measures can include redundancy, diversity and monitoring.

In general, the following fault criteria shall be taken into account.

— If, as a consequence of a fault, further components fail, the first fault and all following faults are considered to be a single fault.

— Two or more separate faults having a common cause are regarded as a single fault (known as common cause failure).

— The simultaneous occurrence of two independent faults is highly unlikely and therefore does not need to be considered.

Iterations between the hardware and software development may be required in order to complete the necessary hardware testing.

6.3 Prerequisites

The prerequisite is the parts of the functional safety concept to be realized by the hardware (ISO 25119-2:2018, Clause 7).

The prerequisite for the “hardware system integration” and the “hardware safety validation” shown in Figure 2 for hardware that is controlled by software is the operating software itself.

6.4 Requirements

The hardware development process shall begin at the system level where safety-related functions and associated requirements are identified (see Figure 2).

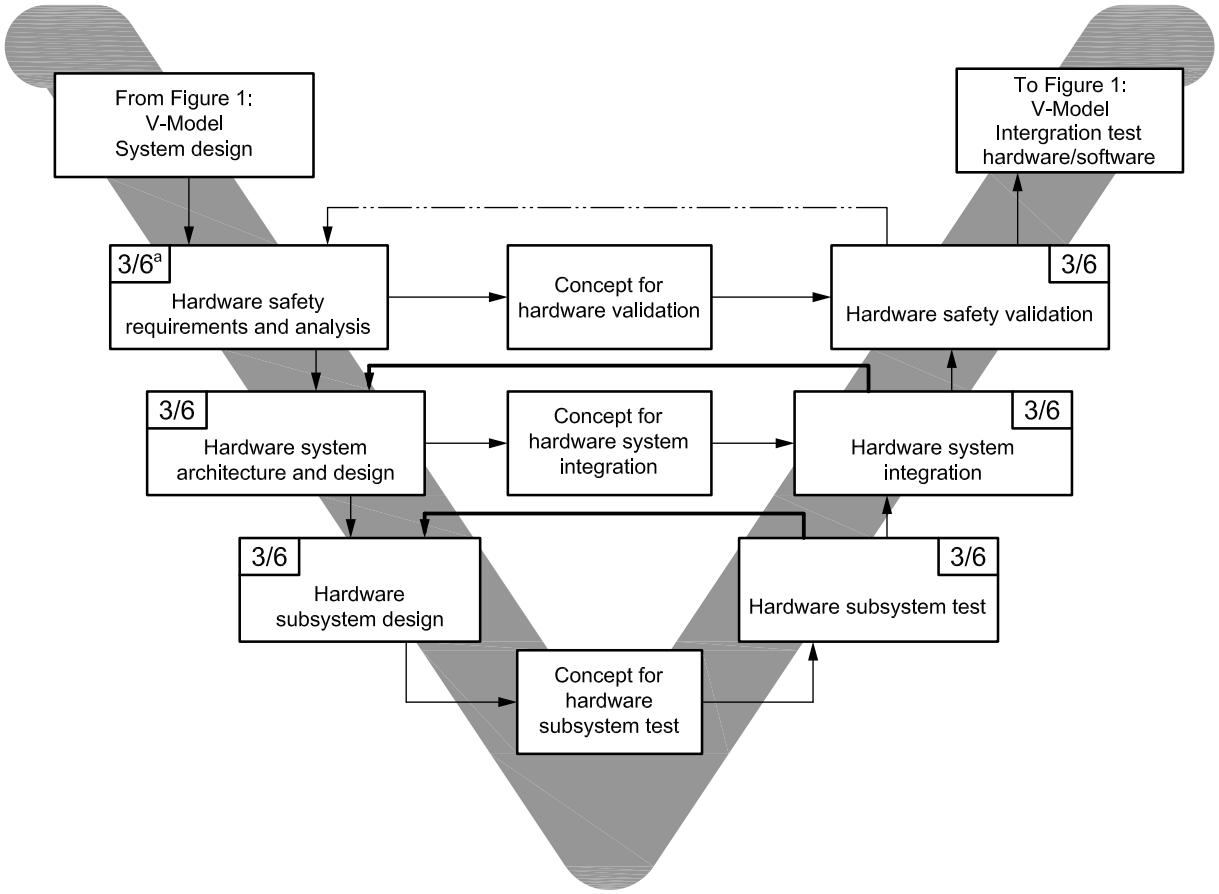
The functional safety concept shall be used to identify the performance level (AgPL) for each system safety-related function (ISO 25119-2).

Selection of hardware categories, $MTTF_{DC}$, DC and SRL shall be made such that the resultant AgPL of the individual or combination SRP/CS meets or exceeds all $AgPL_r$’s of the assigned functional safety requirements.

The system may be broken down into subsystems for easier development.

Each phase of the development cycle shall be verified.

The system integration hardware/software test shown in Figure 1 shall be performed using previously tested hardware and software components.



Key

result
verification
validation

a The first of two numbers separated by a slash refers to this document and the second refers to Clause 6.

Figure 2 — Hardware development V-model

The procedure for the hardware architectural design is as follows.

a) Select a hardware category taking into account the available $MTTF_{DC}$, SRL and DC (see ISO 25119-2:2018, Figure 2).

b) Identify the component operating environment and stress level.

c) Select components.

d) Calculate and verify that the $MTTF_{DC}$ meets the required level (see ISO 25119-2:2018, 7.3.3).

e) Determine and verify that the DC meets the required level (see ISO 25119-2:2018, 7.3.4).

f) Consider CCF (see ISO 25119-2:2018, Annex D).

g) Consider systematic failures (see ISO 25119-2:2018, Annex E).

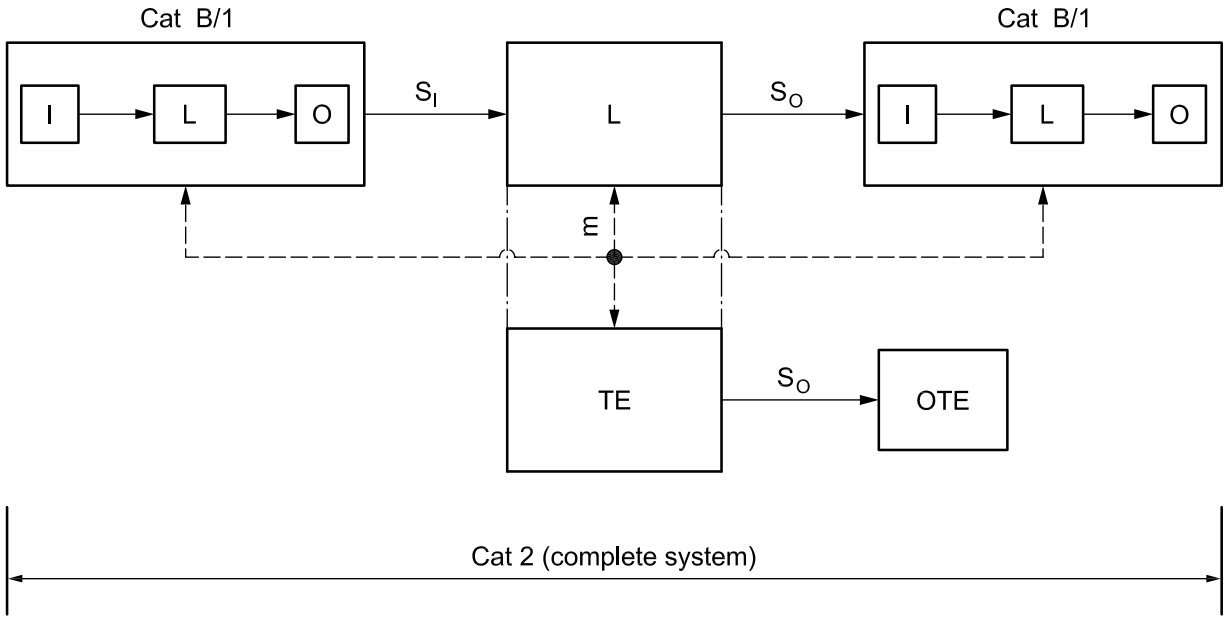
h) Consider other safety-related functions (see ISO 25119-2:2018, Annex F).

Iteration could be required for the above steps.

6.5 Hardware categories

The safety-related parts of control systems shall be designed in accordance with the requirements of one or more of the five categories specified in ISO 25119-2:2018, Annex A taking into account ISO 25119-2:2018, Annexes H, I and J.

When a safety-related function is realized by an integrated combination of multiple hardware categories, the resulting overall hardware category is determined by the characteristics of the overall system (see Figure 3).



Key

I	input device (e.g. sensor)	S_I	interconnecting signal input
L	Logic	S_O	interconnecting signal output
O	output device (e.g. actuator)	m	monitoring
TE	test equipment	Cat	hardware category
OTE	output of test equipment		

Figure 3 — Integrated system example for hardware category 2

To determine the overall SRL, see 7.3.4.7.

6.6 Work products

The following work products shall be applicable to hardware design:

a) hardware safety validation test plan;

b) hardware safety validation test specification;

c) hardware safety validation test results;

d) hardware system integration test plan/hardware subsystem test plan;

e) hardware system integration test specification;

f) hardware system integration test results/hardware subsystem test results;

g) hardware architectural design that fulfils the required category and AgPL;

h) hardware safety requirements.

7 Software

7.1 Software development planning

7.1.1 Objectives

The objective is to determine and plan the individual phases of software development. This includes the process of software development itself, which is described in this clause, as well as the necessary supporting processes described in ISO 25119-4:2018, Clause 11.

7.1.2 General

Figure 4 illustrates the process for developing software. In the following paragraphs and tables, each box in the diagram is explained in detail.

Appropriate techniques/measures shall be selected according to the required SRL. Given the large number of factors that affect software safety integrity, it is not possible to give an algorithm for combining the techniques and measures that are correct for any given application. For a particular application, the appropriate combination of techniques or measures shall be stated during software safety planning, with appropriate techniques or measures being selected according to the requirements in 7.1.4.

7.1.3 Prerequisites

The prerequisites in this phase are

— the required SRL as determined by the AgPL, $MTTF_{DC}$, DC and category from the functional safety concept,

— the system verification plan,

— the technical safety concept, and

— the functional safety requirements and safety goals.

7.1.4 Requirements

7.1.4.1 Phase determination

For the software development process, it shall be determined how phases of software development (see Figure 4) are to be carried out. The extent and complexity of the project shall be taken into account. The phases may be carried out according to Figure 4, without modification, or individual phases may be combined, if all work products of the combined phases are generated.

NOTE It is common to combine individual phases if the method used makes it difficult to clearly distinguish between the phases. For example, the design of the software architecture and the software implementation can be generated successively with the same computer-aided development tool, as is done in the model-based development process.

Other phases may be added by distributing the activities and tasks.

EXAMPLE The application of data can be inserted as a separate phase before the safety validation of the electronic control unit. The safety validation of the ECU can be conducted differently depending on the distribution of the functions — as a test of particular ECU or as a test of the combined control network. It could be conducted at the test location of the component systems or on the laboratory test vehicle.

7.1.4.2 Process flexibility

Activities and tasks may be moved from one phase to another.

7.1.4.3 Process timetable

A timetable shall be set up showing the relationship between the individual phases of the software development and the product development process including the integration steps at machine level.

7.1.4.4 Applicability

After the software safety requirement specification has been completed according to Table 1, it shall be determined which software safety requirements shall be applied to which process activities.

7.1.4.5 Supporting processes

The following supporting processes shall be planned as part of the software development process:

a) work product documentation according to ISO 25119-4:2018, Clause 13;

b) software change management according to ISO 25119-4:2018, Clause 11;

c) work product configuration management.

NOTE Supporting process b) includes a strategy for dealing with the different branches of the software that result from changes, including the merging of these branches.

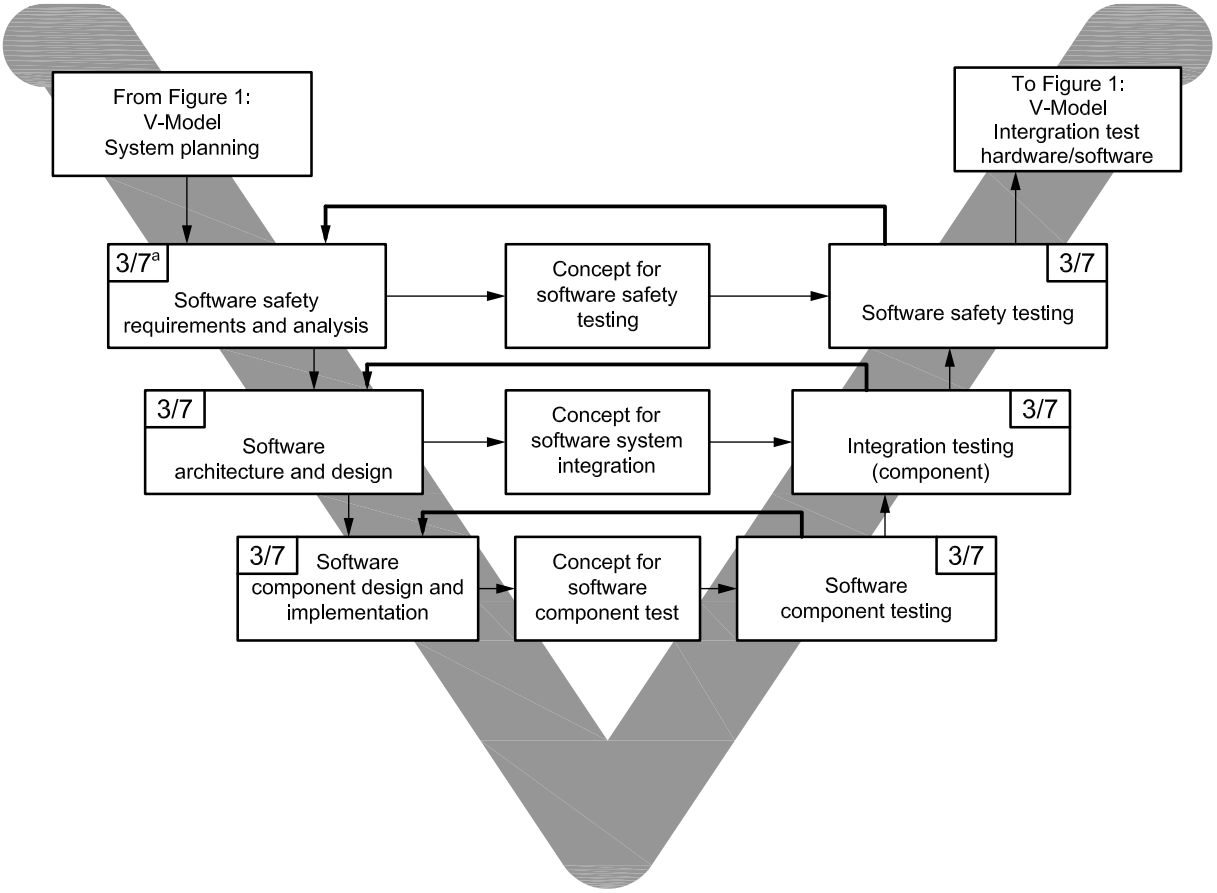
7.1.4.6 Phases of software development

For each phase of software development, the selection of the appropriate development methods and measures, the corresponding tools, and the guidelines for the implementation of the methods, measures and tools shall be carried out according to the SRL (e.g. cyclomatic complexity, test coverage, defect rates).

These selections shall be justified with regard to the appropriateness to the application area, and shall be made at the beginning of each development phase.

When selecting methods and measures, it needs to be kept in mind that, in addition to manual coding, model-based development may be applied in which the source code or the object code is automatically generated from models.

NOTE The selection of coordinated methods and measures offers the possibility of reducing the complexity of software development.



Key

→ result

← verification

a The first of the two numbers separated by a slash refers to this document and the second refers to Clause 7.

Figure 4 — Software development V-model

7.1.4.7 Using the tables

For every development method and measure, Tables 1 to 6 present an entry for each of the four SRL, using either the symbol "+", "o", or "x":

— + the method or measure shall be used for this SRL, unless there is reason not to, in which case that reason shall be documented during the planning phase;

— o there is no recommendation for or against the use of this method or measure for this SRL;

— x the method or measure is not suitable to meet this SRL.

Methods and measures corresponding to the respective SRL shall be selected. Alternative or equivalent methods and measures are identified by letters after the number. At least one of the alternative or equivalent methods and measures marked with a "+" shall be selected.

If a special method or measure is not listed in the tables, this does not mean that such a method or measure may not be used. If an unlisted method or measure is substituted for one listed in the table, it shall be one that has an equivalent or higher value.

7.1.5 Work products

The work product applicable to this phase is the software project plan resulting from 7.1.1 to 7.1.4.6 (see also ISO 25119-4:2018, Clause 6).

7.2 Software safety requirements specification

7.2.1 Objectives

The first objective is to derive the software safety requirements, including the SRL, from the technical safety requirements.

The second objective is to verify that the software safety requirements are consistent with the technical safety concept.

7.2.2 General

The software safety requirements specification shall be derived from the requirements of the technical safety concept of the system, and labelled as software safety requirements. At least the following shall be taken into account:

a) adequate implementation of the technical safety concept in the software;

b) system configuration and architecture;

c) design of the E/E/PES system hardware;

d) response times of the safety-related functions;

e) external interfaces, such as communication;

f) physical requirements and environmental conditions as far as they affect the software;

g) requirements for safe software modification.

Iterations are required between the hardware and software development of the system. During the process of further specifying and detailing the software safety requirements and the software architecture, there can be repercussions on the hardware architecture. For this reason, there shall be close cooperation between the hardware development and the software development.

7.2.3 Prerequisites

The following are the prerequisites for the software safety requirements specification:

— software project plan according to 7.1.1 to 7.1.4.6;

— technical safety concept according to 5.4.2;

— hardware categories according to 6.5.

7.2.4 Requirements

7.2.4.1 Software safety requirements specification methods

The software safety requirements specification shall be in accordance with Table 1.

Table 1 — Software safety requirements specification

Technique/measure[a]	Subclause	SRL = B	SRL = 1	SRL = 2	SRL = 3
1 Requirements specification in natural language	7.2.4.1.1	+	+	+	+
2a Informal methods [a, b]	7.2.4.1.2	+	+	x	x
2b Semi-formal methods [b]	7.2.4.1.3	+	+	+	+
2c Formal methods [b]	7.2.4.1.4	+	+	+	+
3 Computer-aided specification tools	7.2.4.1.5	o	o	+	+
4a Inspection of software safety requirements[a]	7.2.4.7	+	+	+	+
4b Walk-through of software safety requirements	7.2.4.7	+	+	x	x

NOTE 1 Ways of modelling which possess a complete syntax definition and a complete semantic definition with calculus are summarized in item 2c. Formal methods admit to formal verification and automatic test case generation. Examples include state machines connected to formal verification.

NOTE 2 Ways of modelling which possess a complete syntax definition and a semantic definition without calculus are summarized in item 2b. Examples include structured analysis/design and graphic ways of modelling, such as UML class diagrams or block diagrams.

See 7.1.4.7 for instructions on the use of Tables 1 to 6.

[a] Appropriate techniques/measures shall be selected according to the SRL. Alternative or equivalent techniques/measures are indicated by a letter following the number. Only one of the techniques/measures need be satisfied.

[b] In case of model-based development with code generation, the methods and measures for software architectural design have to be applied to the functional model, which will serve as the basis for code generation.

7.2.4.1.1 Requirements specification in natural language

7.2.4.1.1.1 Aim

The specification requirements shall be introduced in natural language (i.e. ordinary spoken and written).

7.2.4.1.1.2 Description

The software safety requirements specification shall include a description of the problem in natural language, and if necessary, further informal methods, such as figures and diagrams.

7.2.4.1.2 Informal methods

7.2.4.1.2.1 Aim

To express a complete concept, a natural language shall be used.

7.2.4.1.2.2 Description

Informal methods shall provide a means of developing a description of a system at an appropriate stage in its development, e.g. specification, design or construction, typically by means of natural language, diagrams, figures.

7.2.4.1.3 Semi-formal methods

7.2.4.1.3.1 Aim

Semi-formal methods shall express a concept, specification, design, or construction unambiguously and consistently, so that mistakes and omissions can be detected.

7.2.4.1.3.2 Description

Semi-formal methods for software implementation shall be used to provide a means of developing a description of a system at an appropriate stage in its development, e.g. specification, design or construction.

EXAMPLE Data flow diagrams, finite state machines/state transition diagrams.

The description shall, when appropriate, be analysed by machine or animated to display various aspects of the system behaviour. Animation gives extra confidence that the system meets the requirements.

7.2.4.1.4 Formal methods

7.2.4.1.4.1 Aim

The development of software in a way that is based on mathematics shall include formal concept, specification, design and construction techniques.

7.2.4.1.4.2 Description

Formal methods shall provide a means of developing a description of a system at an appropriate stage in its specification, design or implementation. The resulting description is in a strict notation that shall be subjected to mathematical analysis to detect various classes of inconsistency or incorrectness. Moreover, the description shall, when appropriate, be analysed by machine with a rigour similar to the syntax checking of a source program by a compiler, or animated to display various aspects of the behaviour of the system described. Animation gives extra confidence that the system meets the formally specified requirement, because it improves human recognition of the specified behaviour.

A formal method shall generally offer a notation (normally some form of discrete mathematics being used), a technique for deriving a description in that notation, and various forms of analysis for checking a description for different correctness properties.

7.2.4.1.5 Computer-aided specification tools

7.2.4.1.5.1 Aim

To facilitate automatic detection of ambiguity and completeness, formal specification techniques shall be used.

7.2.4.1.5.2 Description

The technique shall produce specifications in an electronic format that facilitates inspection to assess consistency and completeness. In general, the technique supports not only the creation of the specification but also of the design and other phases of the project life cycle.

7.2.4.2 Non-safety–related functions

If functions additional to the safety-related functions are carried out by the E/E/PES, these shall be specified, or reference shall be made to their specifications.

If the requirements specification includes both the non-safety requirements and the safety requirements, the latter shall be clearly identified as such.

7.2.4.3 Level of detail

The software safety requirements specification shall contain sufficient detail to implement the safety-related function in the software.

7.2.4.4 Consistency

Bi-directional traceability between software safety requirements and technical safety requirements shall be realized.

7.2.4.5 Hardware and software co-dependency

The software safety requirements specification shall specify the safety-related dependencies between the hardware and the software, if relevant.

7.2.4.6 Software safety requirements specification

The software safety requirements specification shall describe software safety requirements for the following, if relevant.

— Functions that enable the system to achieve or maintain a safe state.

— Functions related to the detection, indication and handling of faults in the ECU, sensors, actuators and communication system.

— Functions related to the detection, indication and handling of faults in the software itself (self-supervision of the software).

NOTE 1 This includes both the self-monitoring of the software in the operating system, and an application-specific self-monitoring of the software aimed at detecting systematic faults in the application software.

— Functions related to the online and offline tests of the safety-related functions.

NOTE 2 Self-testing can be necessary during operation and when the vehicle is started.

NOTE 3 This refers in particular to the testability of the safety-related functions in customer service or through other E/E/PES systems.

— Functions that allow modifications of the software to be carried out safely.

— Interfaces with functions that are not safety-related.

— Performance and reaction time.

— Interfaces between the software and the hardware of the electronic control unit.

NOTE 4 The interfaces also include programming and configuration.

The software safety requirements specification shall contain:

a) the SRL for each of the functions listed above;

b) the acceptance criteria for the software safety validation of the software safety requirements.

7.2.4.7 Software safety requirements verification

The software safety requirements shall be examined to determine if they comply with the requirements given in 7.2.4.1 to 7.2.4.6. The software safety requirements shall also be reviewed to verify whether they are consistent with the technical safety concept. The test specification and test report defined in Table 6 shall also be reviewed to verify whether they are consistent with the software safety requirement. The software developers shall participate in the verification activities. The verification methods may be either of two types: inspection or walk-through (as defined in ISO 25119-1) as appropriate.

7.2.5 Work products

The following work products are applicable to this phase:

a) software safety requirements specification according to 7.2.4.1 and 7.2.4.3 to 7.2.4.6;

b) non-safety-related software requirements specification according to 7.2.4.2;

c) verification report on the software safety requirements specification resulting from 7.2.4.7.

7.3 Software architecture design

7.3.1 Objectives

The objective of the software architecture is the structuring of all software requirements using software components that determine how the software shall be implemented. The software architecture shall ensure that all software safety requirements are fulfilled by the software components allocated to them.

7.3.2 General

The software architecture is a representation of all software components and their interactions with one another in a hierarchical structure. The static aspects, such as the interfaces and data paths of all the software components, as well as the dynamic aspects, such as process sequences and time behaviour, shall be described.

7.3.3 Prerequisites

A sufficient degree of maturity in the software safety requirements specification is the prerequisite to starting development of the software architecture.

7.3.4 Requirements

7.3.4.1 Software architecture design methods

The software architecture shall be developed in accordance with Table 2.

Table 2 — Software architecture design

Technique/measure[a]		Subclause	SRL = B	SRL = 1	SRL = 2	SRL = 3
1a	Informal methods[a]	7.2.4.1.2	+	+	x	x
1b	Semi-formal methods	7.2.4.1.3	+	+	+	+
1c	Formal methods	7.2.4.1.4	+	+	+	+
2	Computer-aided specification tools	7.2.4.1.5	o	o	+	+
3a	Inspection of software architecture[a]	7.3.4.6	+	+	+	+
3b	Walk-through of software architecture	7.3.4.6	+	+	x	x

See 7.1.4.7 for instructions on the use of Tables 1 to 6.

[a] Appropriate techniques/measures shall be selected according to the SRL. Alternative or equivalent techniques/measures are indicated by a letter following the number. Only one of the techniques/measures needs to be satisfied.

7.3.4.2 Design method characteristics

The selected design method shall have characteristics supporting:

a) abstraction, modularity, encapsulation and other characteristics that make complexity manageable;

b) the description of:

- functionality;
- the information flow between the software components;
- process control and information regarding time;
- time limitations;
- concurrent processes, if relevant;
- data structures and their characteristics;
- assumptions in the design and their dependencies;

c) the understanding of the developers and others involved;

d) capability for software modification;

e) testing.

7.3.4.3 Software architecture structure

A software architecture that describes the hierarchical structure, based on the software safety requirements, of all the safety-related software components shall be developed.

NOTE At the top level of the software architecture, there is usually a separation into basic software (e.g. board support packages, operating systems, hardware abstraction layer) and application software.

7.3.4.4 Level of detail

The hierarchical structure of the software architecture shall end with the software components at the lowest level.

When developing the software architecture, the extent of the safety-related software components should be kept as small as possible.

7.3.4.5 Software architecture traceability

Bi-directional traceability between software architecture and software safety requirements shall be realized.

7.3.4.6 Software architecture verification

The software architecture shall be verified. It shall be examined whether the designed architecture satisfies the software safety requirements. The test specification and test report defined in Table 5 shall also be reviewed to verify whether they are consistent with the software architecture. The software developers shall participate in the verification activities. The verification method may be either of two types: inspection or walk-through (as defined in ISO 25119-1) and as appropriate (see Table 2).

7.3.4.7 Combination of safety-related software components

If the embedded software has to implement software components with different SRL or safety-related and non-safety-related software components, then the overall SRL shall be limited to the software component implemented with the lowest SRL, unless adequate independence between the software components can be demonstrated according to the needs of the control system. SRL 2 or 3 shall be in accordance with Annex B.

7.3.5 Work products

The following work products are applicable to this phase:

a) software architecture according to 7.3.4.1 to 7.3.4.5;

b) a software architecture verification report resulting from 7.3.4.6.

7.4 Software component design and implementation

7.4.1 Objectives

The first objective is to specify in detail the behaviour of the safety-related software components that are prescribed by the software architecture.

The second objective is to generate a readable, testable and maintainable source (code, model, etc.) suitable to be translated into object code.

The third objective is to verify that the software architecture has been fully and correctly implemented.

7.4.2 General

7.4.3 Prerequisites

The following are the prerequisites for software component design and implementation:

— software project plan (see 7.1.1 to 7.1.4.6);

— software requirements [see 7.2.5 a) and b)];

— software architecture (see 7.3.4.1 to 7.3.4.5);

— software verification plan (see ISO 25119-4:2018, Clause 6).

7.4.4 Requirements

7.4.4.1 Software component design and implementation methods

Software shall be designed and developed in accordance with Table 3.

Table 3 — Software design and development — Support tools and programming language

Technique/measure[a]		Subclause	SRL = B	SRL = 1	SRL = 2	SRL = 3
1 Tools and programming language						
1.1	Suitable programming language	7.4.4.1.1	+	+	+	+
1.2	Strongly typed programming language	7.4.4.1.2	o	+	+	+
1.3	Language subset	7.4.4.1.3	o	+	+	+
1.4	Tools and translators: increased confidence from use	7.4.4.1.4	o	+	+	+
1.5	Use of trusted/verified software components (if available)	7.4.4.1.5	o	o	+	+
2 Design methods						
2.1a	Informal methods[a]	7.2.4.1.2	+	+	x	x
2.1b	Semi-formal methods	7.2.4.1.3	+	+	+	+
2.1c	Formal methods	7.2.4.1.4	+	+	+	+
2.2	Defensive programming	7.4.4.1.6	o	o	o	+
2.3	Structured programming	7.4.4.1.7	o	+	+	+
2.4	Modular approach					
2.4.1	Software component size limit	7.4.4.1.8	o	+	+	+
2.4.2	Software complexity control	7.4.4.1.9	o	o	o	+
2.4.3	Information hiding/encapsulation	7.4.4.1.10	o	o	+	+
2.4.4	One entry/one exit point in subroutines and functions	7.4.4.1.8	o	+	+	+
2.4.5	Fully defined interface	7.4.4.1.8	o	+	+	+
2.5	Library of trusted/verified software components	7.4.4.1.11	+	+	+	+
2.6	Computer-aided design tools	7.4.4.1.12	o	o	o	+
3 Design and coding standard						
3.1	Use of coding standard	7.4.4.1.13	o	+	+	+
3.2a	No dynamic variables or objects	7.4.4.1.14	o	o	o	+
3.2b	Online checking of the creation of dynamic variables	7.4.4.1.15	o	o	o	+
3.3.	Limited use of interrupts	7.4.4.1.16	o	o	o	+
3.4	Defined use of pointers	7.4.4.1.17	o	o	o	+
3.5	Limited use of recursion	7.4.4.1.18	o	o	o	+
4 Design and coding verification						
4a	Inspection of software design and/or source code[a]	7.4.4.2	+	+	+	+
4b	Walk-through of software design and/or source code	7.4.4.2	+	+	x	x

See 7.1.4.7 for instructions on the use of Tables 1 to 6..

[a] Appropriate techniques/measures shall be selected according to the SRL. Alternative or equivalent techniques/measures are indicated by a letter following the number. Only one of the techniques/measures needs to be satisfied.

7.4.4.1.1 Suitable programming language

7.4.4.1.1.1 Aim

The aim is to choose a programming language to support the requirements of ISO 25119 (all parts) so far as practicable, in particular defensive programming, structured programming and possibly assertions. The programming language chosen shall lead to easily verifiable code, and facilitate program development, verification and maintenance.

7.4.4.1.1.2 Description

The language shall be fully and unambiguously defined. The language shall be user- or problem-orientated rather than processor/platform machine-orientated. Widely used languages (e.g. C, C++, Ada) or their subsets are preferred to special-purpose languages.

In addition to the already referenced features, the language or subsets shall provide for:

— block structure;

— translation time type checking.

The language shall support:

— the use of small and manageable software components (see 7.4.4.1.8);

— restriction of access to data in specific software components;

— definition of variable sub-ranges;

— any other type of error-limiting constructs.

If safe operation of the system is dependent upon real-time constraints, then the language shall also provide for exception/interrupt handling. It is desirable that the language be supported by a suitable translator (e.g. compiler, assembler, linker and locator), appropriate libraries of pre-existing software components, a debugger, and tools for both version control and development. The following features which make verification difficult shall be avoided:

a) unconditional jumps, excluding subroutine calls;

b) recursion;

c) overuse of dynamic variables or objects, heaps, or pointers;

d) handling of interrupts at application level;

e) multiple entries or exits of loops, blocks or subprograms;

f) implicit variable initialisation or declaration;

g) variant records and equivalence;

h) procedural parameters.

NOTE Low-level languages, in particular assembly languages, present problems due to their processor/platform machine-orientated nature. A desirable language property is that the design and use result in programs whose execution is predictable. Given a suitably defined programming language, there is a subset which ensures that program execution is predictable. This subset cannot (in general) be statically determined, although many static constraints can assist in ensuring predictable execution. This would typically require a demonstration that array indices are within bounds, and that numeric overflow cannot arise, for example.

7.4.4.1.2 Strongly typed programming language or guideline checking

7.4.4.1.2.1 Aim

The probability of faults shall be reduced by using a language or programming practice which permits a high level of checking by the translator or static analysis tool.

7.4.4.1.2.2 Description

When a strongly typed programming language is translated or statically analysed, many checks need to be made on how variable types are used (for example, in procedure calls and external data access). Translation shall fail and produce an error message for any usage that does not conform to predefined rules.

NOTE Such languages usually allow user-defined data types to be defined from the basic language data types (such as integer, real). These types can then be used in exactly the same way as the basic type. Strict checks are imposed to ensure the correct type is used. These checks are imposed over the whole program, even if this is built from separately translated units. The checks also ensure that the number and type of procedure arguments match, even when referenced from separately translated software components.

7.4.4.1.3 Language subset

7.4.4.1.3.1 Aim

The use of a language subset shall reduce the probability of introducing programming faults and increase the probability of detecting any remaining faults.

7.4.4.1.3.2 Description

The programming language shall be examined to determine programming constructs which are either undefined, error-prone or difficult to analyse (e.g. using static analysis methods). These programming constructs shall then be excluded and a language subset defined. Also, it shall be documented as to why the constructs used in the language subset are safe.

7.4.4.1.4 Tools and translators — Increased confidence from use

7.4.4.1.4.1 Aim

Tools and translators which are proven in use shall be applied, in order to avoid any difficulties due to translator failures which can arise during development, verification and maintenance of software packages.

7.4.4.1.4.2 Description

A translator shall be used where there has been no evidence of improper performance over a substantial number of prior projects. Translators without operating experience or with any serious known faults shall be avoided unless there is some other assurance of correct performance. If the translator has shown small deficiencies, the related language constructs are identified, and carefully avoided during a safety-related project.

NOTE 1 This description is based on experience from many projects. It has been shown that immature translators are a serious handicap to software development and make a safety-related software development generally unfeasible.

NOTE 2 It is recognized that no method currently exists to prove the correctness for all tool or translator parts.

7.4.4.1.5 Use of trusted/verified software components (if available)

7.4.4.1.5.1 Aim

Trusted/verified software components shall be used to avoid the need for software components and hardware component designs to be extensively revalidated or redesigned for each new application. This allows the developer to take advantage of designs which have not been formally or rigorously verified, but for which considerable operational history is available.

7.4.4.1.5.2 Description

This measure shall verify that the software components are sufficiently free from systematic design faults and/or operational failures. Only in rare cases will the employment of trusted software components (i.e. those which are proven in use) be sufficient as the sole measure to ensure that the necessary SRL is achieved. For complex software components with many possible functions (e.g. operating system), it is essential to establish which functions are actually sufficiently proven in use. For example, where a self-test routine is provided to detect hardware faults, but no hardware failure occurs within the operating period, the self-test routine cannot be considered as being proven by use.

A software component can be sufficiently trusted if it is already verified to the required SRL, or if it fulfils the following criteria:

— unchanged specification impacting safety-related function (e.g. changes to decision points and lookup tables are acceptable provided that, through verification, the safety-related function was not impacted);

— at least one year of service history;

— all of the operating experience of the software component shall relate to known demand profiles, ensuring that increased operating experience (e.g. in systems in different use applications) leads to an increased knowledge of the behaviour of the software component;

— no failures of safety-related functions.

NOTE 1 Failure of a non-safety-related function in one application can be a failure of a safety-related function in another application, and vice versa.

To enable verification that a component or software component fulfils the above criteria, the following shall be documented:

a) exact identification of each system and its components, including version numbers (for software and hardware) used in the verification process;

b) identification of statistically significant sample of users and time of application use (e.g. years of service);

c) procedure for the selection of the statistically significant sample;

d) procedures for detecting and registering failures, and for removing faults.

NOTE 2 The application of this method is possible to entire SW systems (also provided by suppliers) if the required criteria are fulfilled and the required evidences are documented.

7.4.4.1.6 Defensive programming

7.4.4.1.6.1 Aim

Defensive programming shall be used to produce programs which detect anomalous control flow, data flow or data values during their execution, and which react to these in a predetermined and acceptable manner.

7.4.4.1.6.2 Description

Many techniques can be used during programming to check for control or data anomalies. The techniques used shall be applied systematically throughout the programming of a system to decrease the likelihood of erroneous data processing. Intrinsic error-safe software is designed to accommodate its own design shortcomings. These shortcomings can be due to mistakes in design or coding, or to erroneous requirements. There are two overlapping areas of defensive techniques. The first set of defensive techniques includes the following:

— range checking the variables;

— checking values for plausibility;

— type, dimension and range checking parameters of procedures at procedure entry.

This first set of defensive techniques helps 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 shall be separated, and their access checked. Functions shall treat all parameters as read-only. Literal constants shall not be "write" accessible. This helps detect accidental overwriting or mistaken use of variables. Fault tolerant software shall be designed to "expect" failures in its own environment or use outside nominal or expected conditions, and behave in a predefined manner. The second set of defensive techniques includes the following:

— checking input variables and intermediate variables with physical significance for plausibility;

— checking the effect of output variables, preferably by direct observation of associated system state changes;

— checking by the software of its configuration, including both the existence and accessibility of expected hardware, and also that the software itself is complete — 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.

7.4.4.1.7 Structured programming

7.4.4.1.7.1 Aim

Structured programming shall be used to design and implement the program such that it is practical to analyse without being executed.

7.4.4.1.7.2 Description

The following shall be carried out so as to minimize structural complexity.

a) Divide the program into appropriately small software components, ensuring they are decoupled as far as possible and all interactions are explicit.

b) Compose the software component control flow using structured constructs, i.e. sequences, iterations and selection.

c) Keep the number of possible paths through a software component small, and the relation between the input and output parameters as simple as possible.

d) Avoid complicated branching. In particular, avoid unconditional jumps (go-to) in higher-level languages.

e) Where possible, relate loop constraints and branching to input parameters.

f) Avoid using complex calculations as the basis of branching and loop decisions. Features of the programming language which encourage the above approach shall be used in preference to other features which are (allegedly) more efficient, except where efficiency takes absolute priority.

7.4.4.1.8 Modular approach

7.4.4.1.8.1 Aim

A modular approach shall be used for a decomposition of the software system into small comprehensible parts, in order to manage the complexity of the system.

7.4.4.1.8.2 Description

A modular approach (modularisation) presupposes a number of rules for the design, coding and maintenance phases of a software project. These rules vary according to the design method employed. For the methods of this document, the following apply.

— A software component shall have a single well-defined task or function to fulfil (applicable to items 2.4.1 and 2.4.4 of Table 3);

— Connections between software components shall be limited and strictly defined; coherence in one software component shall be strong (applicable to item 2.4.5 of Table 3);

— Collections of subprograms shall be built, providing several levels of software components (applicable to item 2.4.5 of Table 3);

— Software component size shall be restricted to a specified value defined in the coding standard (applicable to item 2.4.1 of Table 3);

— Software components shall have a single entry and a single exit (applicable to item 2.4.4 of Table 3);

— Software components shall communicate with other software 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 (applicable to item 2.4.5 of Table 3);

— All software component interfaces shall be fully documented (applicable to item 2.4.5 of Table 3);

— Any software component interface shall contain only those parameters necessary for its function (applicable to item 2.4.5 of Table 3).

7.4.4.1.9 Complexity metrics

7.4.4.1.9.1 Aim

Complexity metrics shall be used to predict the attributes of programs from properties of the software itself or from its development or test history.

7.4.4.1.9.2 Description

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

— graph theoretic complexity – this measure can be applied early in the lifecycle to assess trade-offs, and is based on the complexity of the program control graph, represented by its cyclomatic number;

— number of ways to activate a certain software component (accessibility) – the more a software component can be accessed, the more likely it is to be debugged;

— Halstead type metrics science – this measure computes the program length by counting the number of operators and operands; it provides a measure of complexity and size that forms a baseline for comparison when estimating future development resources;

— number of entries and exits per software component – minimizing the number of entry/exit points is a key feature of structured design and programming techniques.

7.4.4.1.10 Information hiding/encapsulation

7.4.4.1.10.1 Aim

Information hiding/encapsulation shall be used to prevent unintended access to data or procedures and thereby support a good program structure.

7.4.4.1.10.2 Description

Data that is globally accessible to all software elements can be accidentally or incorrectly modified by any of these elements. Any changes to these data structures may require detailed examination of the code and extensive modifications.

Information hiding is a general approach which shall be applied as appropriate 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 name directory might have access procedures “insert”, “delete” and “find”. The access procedures and internal data structures could be re-written (for example 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.

In this connection, the concept of abstract data types should be used. If direct support is not provided, then it may be necessary to check that the abstraction has not been inadvertently broken.

7.4.4.1.11 Library of trusted/verified software components

7.4.4.1.11.1 Aim

A library of trusted/verified software components shall be used to avoid the need for extensive revalidation or redesign for each new application. This method allows the developer to reuse designs which have not been formally or rigorously validated, but for which considerable operational history is available (also see 7.4.4.1.5).

7.4.4.1.11.2 Description

In order to be well-designed and structured, E/E/PES shall be composed of hardware components and software components which are clearly distinct, and which interact with one another in clearly specified ways.

NOTE 1 E/E/PES designed for differing applications occasionally contain a number of software components which are the same or very similar. Building up a library of such generally applicable software components allows many of the resources necessary for validating the designs to be shared by more than one application.

NOTE 2 Furthermore, the use of such software components in multiple applications provides empirical evidence of successful operational use. This empirical evidence justifiably enhances the trust which users are likely to have in the software components.

7.4.4.1.12 Computer-aided design tools

7.4.4.1.12.1 Aim

CAD tools shall be used to carry out the design procedure more systematically, and to include appropriate automatic construction elements which are already available and tested.

7.4.4.1.12.2 Description

CAD tools shall be used during the design of both hardware and software when available and justified by the complexity of the system. The correctness of such tools shall be demonstrated by specific testing, by an extensive history of satisfactory use, or by independent verification of their output for the applicable safety-related system.

7.4.4.1.13 Use of coding standards

7.4.4.1.13.1 Aim

Coding standards shall be used to facilitate verifiability of the produced code.

7.4.4.1.13.2 Description

The detailed rules shall be fully agreed upon before coding. These rules typically require

— details of modularisation, e.g. interface shapes, software component sizes,

— use of encapsulation, inheritance (restricted in depth) and polymorphism, in the case of object-orientated languages,

— limited use or avoidance of certain language constructs such as “go-to”, “equivalence”, dynamic objects, dynamic data, dynamic data structures, recursion, automatic type conversion, pointers and exits,

— restrictions on interrupts enabled during the execution of safety-critical code,

— layout of the code (listing), and

— no unconditional jumps (for example “go-to”) in programs in higher-level languages.

These rules enable ease of software component testing, verification, assessment and maintenance. Therefore, they shall take into account available tools in particular analysers.

7.4.4.1.14 Design and coding standards — No dynamic variables or objects

7.4.4.1.14.1 Aim

Design and coding standards shall exclude dynamic variables or objects to be avoided, such as

— unwanted or undetected overlay of memory, and

— bottlenecks of resources during (safety-related) run-time.

7.4.4.1.14.2 Description

For the purposes of this measure, dynamic variables and dynamic objects are those variables and objects that have their memory allocated and absolute addresses determined at run-time. The value of allocated memory and its address depends on the state of the system at the moment of allocation, which means that it cannot be checked by the compiler or any other off-line tool.

Because the number of dynamic variables and objects, and the existing free memory space for allocating new dynamic variables or objects, depends on the state of the system at the moment of allocation, it is possible for faults to occur when allocating or using the variables or objects. For example, when the amount of free memory at the location allocated by the system is insufficient, the memory contents of another variable can be inadvertently overwritten. If dynamic variables or objects are not used, these faults are avoided.

7.4.4.1.15 Online checking during creation of dynamic variables or dynamic objects

7.4.4.1.15.1 Aim

Online checking during creation of dynamic variables or dynamic objects shall be used to check that the memory to be allocated to dynamic variables and objects is free before allocation takes place, ensuring that the allocation of dynamic variables and objects during run-time does not impact existing variables, data or code.

7.4.4.1.15.2 Description

In the case of this measure, dynamic variables are those variables that have their memory allocated and absolute addresses determined at run-time (variables in this sense are also the attributes of object instances). By means of hardware or software, the memory is checked to ensure it is free before a dynamic variable or object is allocated to it (for example, to avoid stack overflow). If allocation is not allowed (for example if the memory at the determined address is not sufficient), appropriate action shall be taken. After a dynamic variable or object has been used (for example, after exiting a subroutine) the whole memory which was allocated to it shall be freed.

7.4.4.1.16 Design and coding standards — Limited use of interrupts

7.4.4.1.16.1 Aim

The software developer shall limit the use of interrupts, in order to keep the software verifiable and testable.

7.4.4.1.16.2 Description

The use of interrupts shall be restricted. Interrupts may be used if they simplify the system. Software handling of interrupts shall be inhibited during execution of critical software. If interrupts are used, then parts not able to be interrupted shall have a specified maximum computation time, so that the maximum time for which an interrupt is inhibited can be calculated. Interrupt usage and inhibiting shall be thoroughly documented.

7.4.4.1.17 Design and coding standards — Defined use of pointers

7.4.4.1.17.1 Aim

Defined use of pointers shall be used to avoid the problems caused by accessing data without first checking range and type of the pointer, to support modular testing and verification of software, and to limit the consequence of failures.

7.4.4.1.17.2 Description

In the basic and application software, pointer arithmetic shall be used at source code level only if the pointer data type and value range (to ensure that the pointer reference is within the correct address space) are checked.

7.4.4.1.18 Design and coding standards — Limited use of recursion

7.4.4.1.18.1 Aim

Limited use of recursion shall be employed to avoid unverifiable and unstable use of subroutine calls.

7.4.4.1.18.2 Description

If recursion is used, clear criteria shall be established on the allowed depth of recursion.

7.4.4.2 Software component design and coding verification

The software component design and its coding shall be verified. It shall be examined whether the design and coding fulfil the software safety requirements. The test specification and test report defined in Table 6 shall also be reviewed to verify whether they are consistent with the software component design and its coding. The software developers shall participate in the verification activities. The verification methods may be either of two types: inspection or walk-through (as defined in ISO 25119-1). Verification of automatically generated code may not be necessary.

7.4.5 Work products

The following work products are applicable to this phase:

a) detailed design of the software according to 7.4.4.1;

b) software according to 7.4.4.1;

c) software component design and coding verification report resulting from 7.4.4.2.

7.5 Software component testing

7.5.1 Objectives

The objective of the software component test is to verify that the designed and coded software components correctly implement the software requirements.

7.5.2 General

In this phase, a procedure for testing the software components against their requirements shall be established, and the tests carried out in accordance with that procedure.

7.5.3 Prerequisites

The following are the prerequisites for software component testing:

— software project plan (see 7.1.1 to 7.1.4.6);

— software requirements [see 7.2.5 a) and b)];

— software verification plan (see ISO 25119-4:2018, Clause 6);

— software components according to 7.4.4.1.

7.5.4 Requirements

7.5.4.1 Software component testing methods

The software component testing shall be in accordance with Table 4. Software component test plans shall document the techniques/measures selected to verify the target SRL. Software component test

specifications shall describe the procedures to utilize when executing these techniques/measures. Software component test reports shall record tests conducted and test results.

Table 4 — Software component testing

	Technique/measure[a]	Subclause	SRL = B	SRL = 1	SRL = 2	SRL = 3
1	Static analysis					
	1.1 Boundary value analysis	7.5.4.1.1	+	+	+	+
	1.2 Checklists	7.5.4.1.2	o	o	o	o
	1.3 Control flow analysis	7.5.4.1.3	o	o	+	+
	1.4 Data flow analysis	7.5.4.1.4	o	o	+	+
2	Dynamic analysis and testing	7.5.4.1.5				
	2.1 Test case execution from boundary value analysis	7.5.4.1.6	o	o	o	+
	2.2a Structure test coverage (entry points)	7.5.4.1.7	o	o	+	x
	2.2b Structure test coverage (statements)	7.5.4.1.7	o	o	+	+
	2.2c Structural test coverage (branches)	7.5.4.1.7	o	o	+	+
3	Unit testing					
	3.1 Equivalence classes and input partition testing	7.5.4.1.8	o	o	+	+
	3.2 Boundary value analysis	7.5.4.1.1	o	o	+	+
	3.3 Test case execution from model-based test case generation	7.5.4.1.9	o	o	o	+
4	Performance testing	7.5.4.1.10				
	4.1 Response timings and memory constraints	7.5.4.1.11	o	+	+	+
	4.2 Performance requirements testing	7.5.4.1.12	o	+	+	+
	4.3 Avalanche/stress testing	7.5.4.1.13	o	o	o	+
5	Interface testing	7.5.4.1.14	o	o	o	+

See 7.1.4.7 for instructions on the use of Tables 1 to 6.

[a] Appropriate techniques/measures shall be selected according to the SRL. Alternative or equivalent techniques/measures are indicated by a letter following the number. Only one of the techniques/measures needs to be satisfied.

7.5.4.1.1 Boundary value analysis

7.5.4.1.1.1 Aim

Boundary value analysis shall be used to detect software errors occurring at parameter limits or boundaries.

7.5.4.1.1.2 Description

The input domain of the program is divided into a number of input classes according to the equivalence relation (see 7.5.4.1.5). The tests shall cover the boundaries and extremes of the classes. The tests shall 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 shall receive special attention to the following:

— zero divisor;

— blank ASCII characters;

— empty stack or list element;

— full matrix;

— zero table entry.

Normally, the boundaries for input have a direct correspondence to the boundaries for the output range. Test cases shall be written to force the output to its limited values and when practicable. a test case which causes the output to exceed the specification boundary values shall be specified.

If the output is a sequence of data (e.g. a printed table) special attention shall be paid to the first and the last elements and to lists containing no elements, one element and two elements.

7.5.4.1.2 Checklists

7.5.4.1.2.1 Aim

Checklists may be used to draw attention to, and manage critical appraisal of, all important aspects of the system by safety life cycle phase, ensuring comprehensive coverage without laying down exact requirements.

7.5.4.1.2.2 Description

A checklist is a set of questions to be answered 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.

Checklists may be used for all phases of the overall E/E/PES software safety life cycle, and are particularly useful as a tool to aid in the functional safety assessment.

NOTE 1 To accommodate wide variations in the systems being validated, most checklists contain questions which are applicable to many types of systems. As a result, there can be questions in the checklists being used which are not relevant to the system being dealt with and which need to be ignored. Equally, there might 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 needs to be clear that the use of checklists depends on the expertise and judgement of the engineer selecting and applying the checklist.

Decisions taken by the engineer, with regard to the checklist(s) selected, and any additional or superfluous questions, shall be fully documented and justified.

NOTE 2 The objective is to ensure that the application of the checklists can be reviewed, and that repeatable results will be achieved when the same criteria are used. The object in completing a checklist is to be as concise as possible.

When extensive justification is necessary, this may be done by reference to additional documentation.

Pass, fail and inconclusive, or some similar restricted set of responses, may be used to document the results for each question.

NOTE 3 This conciseness greatly simplifies the procedure of reaching an overall conclusion as to the results of the checklist assessment.

7.5.4.1.3 Static analysis — Control flow analysis

7.5.4.1.3.1 Aim

Control flow analysis shall be used to detect poor and potentially incorrect program structures.

7.5.4.1.3.2 Description

Control flow analysis is a static testing technique for finding suspect areas of code that do not follow good programming practice. The program analysed produces a directed graph which may be further analysed for:

— inaccessible code, e.g. unconditional jumps which leave blocks of code unreachable;

— knotted code, where — in contrast to well-structured code with a control graph reducible by successive graph reductions to a single node — poorly structured code can only be reduced to a knot composed of several nodes.

7.5.4.1.4 Static analysis — Data flow analysis

7.5.4.1.4.1 Aim

Data flow analysis shall be used to detect poor and potentially incorrect program structures.

7.5.4.1.4.2 Description

Data flow analysis is a static testing technique that 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 the following types of variables:

— those that may be read before they are assigned a value, which can be avoided by always assigning a value when declaring a new variable;

— those written more than once without being read, which could indicate omitted code;

— those written but never read, which could indicate redundant code.

A data flow anomaly does not always directly correspond to a program fault; however, if anomalies are avoided, the code is less likely to contain faults.

7.5.4.1.5 Dynamic analysis and testing

7.5.4.1.5.1 Aim

Dynamic analysis and testing shall be used to detect specification failures by inspection of the dynamic behaviour of a prototype at an advanced state of completion.

7.5.4.1.5.2 Description

The dynamic analysis of a safety-related system is carried out by subjecting a near-operational prototype of the safety-related system to input data which is typical of the intended operating environment. The analysis shall be considered satisfactory if the observed behaviour of the safety-related system conforms to the required behaviour. Any failure of the safety-related system shall be corrected and the new operational version shall then be re-analysed.

7.5.4.1.6 Test case execution from boundary value analysis

7.5.4.1.6.1 Aim

Test case execution from boundary value analysis shall be used to detect software errors occurring at parameter limits or boundaries.

7.5.4.1.6.2 Description

Boundary value analysis is defined in section 7.5.4.1.1.

7.5.4.1.7 Structure-based testing

7.5.4.1.7.1 Aim

Structure-based testing shall be used to apply tests which exercise certain subsets of the program structure.

7.5.4.1.7.2 Description

Based on analysis of the program, a set of input data are chosen so that a target percentage of the program code is exercised. This target shall be as large as practical, pre-specified and documented. Measures of code coverage vary as follows, depending upon the level of rigour required. This testing is performed in conjunction with the unit testing described in Table 4 and provides guidance for the expected unit testing coverage.

— **Entry point (call graph) coverage**

  Ensure that every subprogram (subroutine or function) has been called at least once (this is the least rigorous structural coverage measurement).

  In object-oriented languages, there can be several subprograms of the same name which apply to different variants of a polymorphic type (overriding subprograms) which can be invoked by dynamic dispatching. In these cases every such overriding subprogram shall be tested.

— **Statements**

  This is the least rigorous test since it is possible to execute all code statements without exercising both branches of a conditional statement.

— **Branches**

  Both sides of every branch shall be checked. This may be impractical for some types of defensive code.

7.5.4.1.8 Equivalence classes and input partition testing

7.5.4.1.8.1 Aim

Equivalence classes and input partition testing shall be used to test the software adequately using a minimum of test data. The test data shall be obtained by selecting the partitions of the input domain required to exercise the software.

7.5.4.1.8.2 Description

This testing strategy shall be 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 the partitions previously specified. At least one test case is taken from each equivalence class.

There are two basic possibilities for input partitioning:

— equivalence classes derived from the specification — the interpretation of the specification may be either input orientated (e.g. the values selected are treated in the same way) or output orientated (e.g. the set of values lead to the same functional result);

— equivalence classes derived from the internal structure of the program — the equivalence class results are determined from static analysis of the program (e.g. the set of values leading to the same path being executed).

7.5.4.1.9 Test case execution from model-based test case generation

7.5.4.1.9.1 Aim

Automatic test case generation from system models shall be used to facilitate efficient automatic test case execution and to generate highly repeatable test execution suites.

7.5.4.1.9.2 Description

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

Model-based testing is the automatic generation of efficient test cases/procedures using models of system requirements and specified functionality.

NOTE 1 Since testing is very expensive; there is a huge demand for automatic test case generation tools. Therefore, model-based testing is currently a very active field of research, resulting in a large number of available test case generation tools. These tools typically extract a test suite from the behavioural part of the model, guaranteeing to meet certain coverage requirements.

The model is an abstract, partial representation of the desired behaviour of the system under test. From this model, test models are derived, building an abstract test suite. Test cases are derived from this abstract test suite and executed against the system, and tests can be run against the system model as well. Model-based testing with test case generation is based on and strongly related to use of formal methods.

The specific activities in general shall be:

— build the model (from system requirements);

— generate expected inputs;

— generate expected outputs;

— run tests;

— compare actual outputs with expected outputs;

— decide on further action (modify model, generate more tests, estimate reliability/quality of the software).

Tests may be derived with different methods and techniques for expressing models of user/system behaviour, such as:

— by using decision tables;

— by using finite state machines;

— by using grammars;

— by using Markov chain models;

— by using state charts;

— by theorem proving;

— by constraint logic programming;

— by model checking;

— by symbolic execution;

— by using an event-flow model;

— reactive system tests: parallel hierarchical finite automaton.

NOTE 2 Recently, model-based testing is specifically targeting the safety critical domain. It allows for early exposure of ambiguities in specification and design, provides the capability to automatically generate many non-repetitive efficient tests, to evaluate regression test suites in order to assess software reliability and quality, and eases updating of test suites.

7.5.4.1.10 Performance testing — Resource budget analysis

7.5.4.1.10.1 Aim

Performance testing shall be used to ensure that the working capacity of the system is sufficient to meet the specified requirements.

Component level performance testing may be combined with the software integration performance testing.

7.5.4.1.10.2 Description

The requirements specification shall include throughput and response time requirements for specific functions, perhaps combined with constraints on the use of total system resources. The proposed system design shall be compared against the stated requirements by:

— producing a model of the system processes and their interactions;

— determining the use of resources by each process (processor time, communications bandwidth, storage devices, etc.);

— determining the distribution of demands placed upon the system under average and worst-case conditions;

— computing the mean and worst-case throughput and response times for the individual system functions.

7.5.4.1.11 Performance testing — Response time and memory constraints

7.5.4.1.11.1 Aim

Response time and memory constraints shall be used to ensure that the system will meet its temporal and memory requirements.

7.5.4.1.11.2 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 shall be performed to determine 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 (e.g. comparison with an existing system, or the prototyping and benchmarking of time-critical systems).

NOTE For SRL 1, response time and memory constraint testing can be done at the component or integration levels as practicable

7.5.4.1.12 Performance testing — Performance requirements

7.5.4.1.12.1 Aim

Testing shall be established to demonstrate the performance requirements of a software system.

7.5.4.1.12.2 Description

An analysis is performed on both the system and the software requirements specifications to specify all general/specific and explicit/implicit performance requirements.

Each performance requirement shall be examined to determine

— that the success criteria is obtained,

— whether a failure to meet the success criteria is obtained,

— the potential accuracy of such measurements,

— the project stages at which the measurements can be estimated, and

— the project stages at which the measurements can be made.

The practicability of each performance requirement shall be analysed in order to obtain a list of performance requirements, success criteria and potential measurements. The main objectives shall be as follows.

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 as possible.

c) Essential and optional performance requirements and success criteria are specified.

d) Where possible, advantage shall be taken of using a single measurement for more than one performance requirement provided that this will not reduce the validity of the measurement.

7.5.4.1.13 Performance testing — Avalanche/stress testing

7.5.4.1.13.1 Aim

Avalanche/stress testing shall be used to burden the test object with an exceptionally high workload in order to show that the test object would stand normal workloads easily.

7.5.4.1.13.2 Description

There are a variety of test conditions applicable to avalanche/stress testing, including the following.

— If working in a polling mode, then the test object gets many more input changes per time unit than under normal conditions.

— If working on demands, then the number of demands per time unit to the test object is increased beyond normal conditions.

— If the size of a database plays an important role, then it is increased beyond normal conditions.

— Influential devices are tuned to their maximum speed or lowest speed respectively.

— For the extreme cases, all influential factors, as far as possible, are put to the boundary conditions at the same time.

Under these test conditions, the time behaviour of the test object shall be evaluated, the influence of load changes observed, and the correct dimension of internal buffers or dynamic variables, stacks, etc., can be checked.

7.5.4.1.14 Interface testing

7.5.4.1.14.1 Aim

Interface testing shall be used to detect errors in the interfaces of subprograms.

7.5.4.1.14.2 Description

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

— all interface variables at their extreme values,

— all interface variables individually at their extreme values, with other interface variables at normal values,

— all values of the domain of each interface variable, with other interface variables at normal values,

— all values of all variables in combination (only feasible for small interfaces), and

— the specified test conditions relevant to each call of each subroutine.

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

The errors detected during this phase shall be eliminated. For each modification, an impact analysis shall be performed. All modifications which have an impact on the work products of any previous phase shall initiate a return to that phase of the software safety life cycle. All subsequent phases shall then be carried out in accordance with the respective parts of the ISO 25119 series.

7.5.4.2 Elimination of defects

The errors detected during this phase shall be eliminated. For each modification, an impact analysis shall be performed. All modifications which have an impact on the work products of any previous phase shall initiate a return to that phase of the software safety life cycle. All subsequent phases shall then be carried out in accordance with the respective parts of the ISO 25119 series.

7.5.5 Work products

The following work products are applicable to this phase:

a) software component test plan resulting from 7.5.4.1;

b) software component test specification in accordance with 7.5.4.1;

c) software component test report resulting from the performance of the tests.

7.6 Software integration and testing

7.6.1 Objectives

The first objective of software integration and testing is to integrate the software units step by step into software components up to the entire embedded software of the ECU.

NOTE The embedded software of the ECU can consist of either safety-related or non-safety-related software components.

The second objective is to verify that the software requirements are correctly realized by the embedded software at the integration level.

7.6.2 General

In this phase, the particular integration levels are tested against the software requirements. The interfaces between the software components are also tested. The steps of the integration and the tests of the software components shall directly correspond to the hierarchical software architecture.

7.6.3 Prerequisites

The following are the prerequisites for software integration and testing:

— software project plan (see 7.1.1 to 7.1.4.6);

— software requirements [see 7.2.5 a) and b)];

— software architecture (see 7.3.4.1 to 7.3.4.5);

— software verification plan (see ISO 25119-4:2018, Clause 6);

— tested software components according to 7.4.4.1.

7.6.4 Requirements

7.6.4.1 Software integration and test plan

A plan shall be developed for the software integration and tests that shall include at least the following:

a) a software integration strategy;

b) planning of the software integration tests.

The software integration strategy and software test plan should be developed during the software architecture and design phase.

7.6.4.2 Software integration strategy

The software integration strategy shall describe at least the following:

a) the steps to be taken for integrating the individual software components hierarchically;

b) functional dependencies that are relevant to the software integration.

Software integration testing of the components and the software safety testing may be integrated into one phase.

NOTE 1 In the case of model-based development, the software integration might be replaced with integration at the model level.

NOTE 2 Depending on the constraints, the software might be integrated in a host environment, a target-like environment (e.g. an evaluation board) or the target environment (the ECU).

7.6.4.3 Software integration test procedures

Appropriate test procedures shall be developed in the planning of the software integration tests.

NOTE The software integration tests always combine different procedures, because there is no test procedure that covers all the aspects that have to be taken into account equally well.

7.6.4.4 Software integration test methods

A software integration test shall be conducted in accordance with Table 5. Hardware may be used to facilitate the software integration testing, but the testing is focused on testing the software (see Figure 5).

Table 5 — Software Integration testing (component)

	Technique/measure[a]	Subclause	SRL = B	SRL = 1	SRL = 2	SRL = 3
1	Functional or black-box testing	7.6.4.4.1	+	+	+	+
2	Equivalence classes and input partition testing	7.5.4.1.8	o	o	+	+
3	Performance testing					
	3.1a Resource budget analysis	7.5.4.1.10	o	+	x	x
	3.1b Response timings and memory constraints	7.5.4.1.11	o	+	+	+
	3.2 Performance requirements testing	7.5.4.1.12	o	o	+	+
	3.3 Avalanche/stress testing	7.5.4.1.13	o	o	o	+

See 7.1.4.7 for instructions on the use of Tables 1 to 6.

[a] Appropriate techniques/measures shall be selected according to the SRL. Alternative or equivalent techniques/measures are indicated by a letter following the number. Only one of the techniques/measures needs to be satisfied.

7.6.4.4.1 Functional testing

7.6.4.4.1.1 Aim

Functional testing shall be used to reveal failures during the specification and design phases, and to find failures introduced during implementation and the integration of software with the hardware.

7.6.4.4.1.2 Description

During the functional tests, testing results shall be reviewed to determine whether the specified characteristics of the system have been achieved and that the system input data which adequately characterize the normally expected operation have been given. The outputs are observed and their response is compared with that given by the specification. Deviations from the specification and indications of an incomplete specification shall be documented.

Functional testing of electronic components designed for a multi-channel architecture usually involves the manufactured components being tested with pre-validated partner components. In addition and if practicable, manufactured components shall be tested in combination with other partner components of the same batch, in order to reveal common mode faults which otherwise might have remained masked.

7.6.4.5 Elimination of defects

The errors detected during this phase shall be eliminated. For each modification, an impact analysis shall be performed. All modifications which have an impact on the work products of any previous phase shall initiate a return to that phase of the software safety life cycle. All subsequent phases shall then be carried out in accordance with the respective parts of the ISO 25119 series.

7.6.5 Work products

The following work products shall be achieved during this phase:

a) software integration test plan, with a software integration strategy resulting from 7.6.4.1 to 7.6.4.3;

b) software integration test specification as required by 7.6.4.3;

c) software integration test report according to 7.6.4.1.

7.7 Software safety testing

7.7.1 Objectives

The objective of the software safety testing is to show that the software requirements are correctly realized by the embedded software.

The software safety testing is a part of the safety validation of the E/E/PES system (see ISO 25119-4:2018, Clause 6). During the planning of the safety validation of the complete E/E/PES system, it shall be determined which safety goals may be tested at E/E/PES system level, and which may be tested at the software level. In the simplest case, all of the safety goals are covered by the safety validation of the E/E/PES system with the software taken into account, so that no separate software safety validation is necessary.

7.7.2 General

7.7.3 Prerequisites

The following are the prerequisites for software safety testing:

— software project plan (see 7.1.1 to 7.1.4.6);

— software requirements [see 7.2.5 a) and b)];

— software architecture (see 7.3.4.1 to 7.3.4.5);

— software verification plan (see ISO 25119-4:2018, Clause 6);

— integrated software;

— ECU.

7.7.4 Requirements

7.7.4.1 Software safety testing methods

Testing shall be the main verification method for software; animation and modelling may be used to supplement the verification activities; adequate measures/techniques shall be selected according to Table 6.

Table 6 — Software safety testing

Technique/measure[a]	Subclause	SRL = B	SRL = 1	SRL = 2	SRL = 3
1 Tests of software safety requirements					
1.1a Tests within the ECU network[a]	7.7.4.1.1	o	+	+	x
1.1b Hardware-in-the-loop tests	7.7.4.1.2	o	+	+	+
1.1c Tests in the test machine	7.7.4.1.3	o	+	+	+

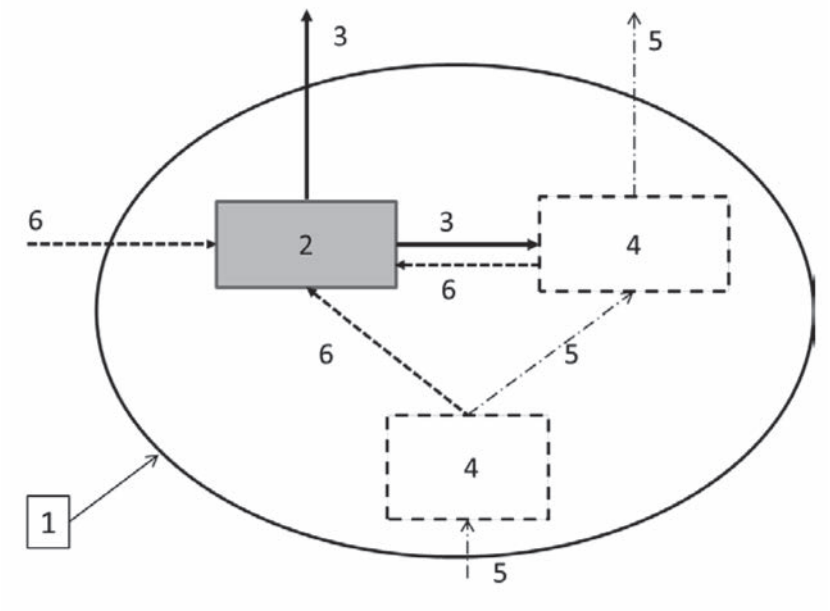
See 7.1.4.7 for instructions on the use of Tables 1 to 6.

NOTE Measures in points 1.1a, 1.1b and 1.1c represent test environments.

[a] Appropriate techniques/measures shall be selected according to the SRL. Alternative or equivalent techniques/measures are indicated by a letter following the number. Only one of the techniques/measures needs to be satisfied.

The software shall be integrated with its host microprocessor in its associated ECU. Inputs from the rest of the system may be simulated while the state of the software and the results are accessed/monitored by means of a dedicated test interface.

The following example in Figure 5 depicts how test interfaces are used for testing.



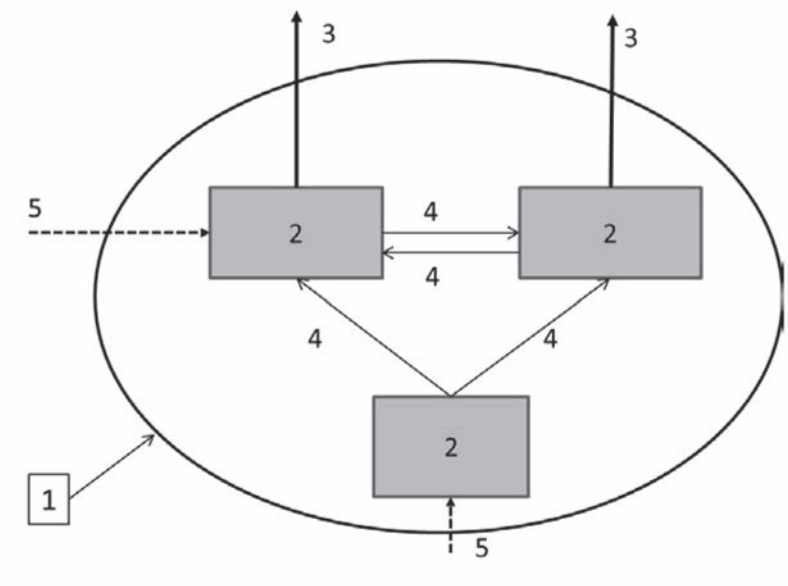
Key

1 complete E/E/PES system
2 real ECU with real software to be tested
3 test interface
4 ECU and software not present
5 signal not present
6 signal simulated

Figure 5 — Test interface of one ECU of an E/EPES system

7.7.4.1.1 Tests within the electronic control unit network

The software shall be integrated with its host microprocessor in its associated ECU, and this ECU shall be integrated with the remaining ECUs that are part of the complete E/E/PES system. The software shall then be tested at the interface to the ECU network, in order to demonstrate that the software behaves according to specification, see Figure 6.



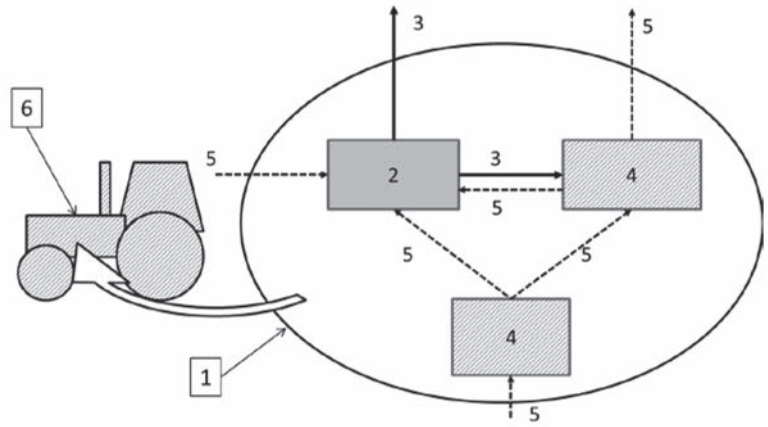
Key

1 complete E/E/PES system
2 real ECU with real software to be tested
3 test interface
4 real signals to be monitored
5 signal simulated

Figure 6 — Test within the electronic control unit network of an E/EPES system

7.7.4.1.2 Hardware-in-the-loop tests

The software shall be integrated with its host microprocessor in its associated ECU, while the rest of the associated E/E/PES system and its environment shall be simulated or physically present. The software shall then be tested in this simulated environment to demonstrate that the software behaves according to specification, see Figure 7.



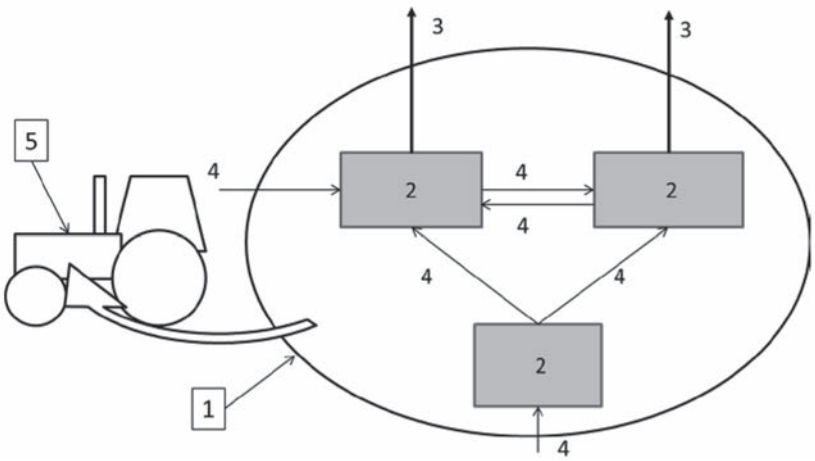
Key

1 complete E/E/PES system
2 real ECU with real software to be tested
3 result to be monitored
4 ECU simulated
5 signal simulated
6 simulated environment

Figure 7 — Hardware-in-the-loop test of one ECU of an E/EPES system

7.7.4.1.3 Tests in the machine

The software and the associated E/E/PES system shall be integrated into the associated machine architecture. The system shall then be tested in the machine to demonstrate that the software behaves according to specification, see Figure 8. This testing may be performed during the system integration hardware/software test (see Figure 1).



Key

1 complete E/E/PES system
2 real ECU with real software to be tested
3 result to be monitored
4 real signals to be monitored
5 real machine

Figure 8 — Test in the machine of an E/EPES system

7.7.4.2 Extent of tests

The software shall be exercised by simulation of

— input signals present during normal operation,

— foreseeable occurrences, and

— undesired conditions requiring system action.

7.7.4.3 Software safety requirements validation

The effectiveness of the test procedures, and of any other measures used, shall be evaluated against the safety concept on conclusion of the verification process, in order to validate the software safety requirements.

7.7.4.4 Documentation

The supplier and/or developer shall make the documented results of the software safety validation and all pertinent documentation available to the system developer to enable him to meet the requirements of ISO 25119-4.

7.7.4.5 Elimination of defects

Errors or defects detected during this phase shall be eliminated. For each modification, an impact analysis shall be performed. All modifications which have an impact on the work products of any previous phase shall initiate a return to that phase of the software safety life cycle. All subsequent phases shall then be carried out in accordance with the respective parts of the ISO 25119 series.

7.7.5 Work products

The following work products shall be achieved during this phase:

a) software safety test plan resulting from 7.7.4.1 to 7.7.4.4;

b) software safety test specification resulting from 7.7.4.1 and 7.7.4.2;

c) software safety test report according to 7.7.4.3.

7.8 Software-based parameterisation

7.8.1 Objective

Software-based parameterisation refers to the possibility of adapting the software system to different requirements, after completion of development, by changing parameters in order to modify the functionality of the software.

The objective is to derive the requirements for safety-related parameters.

7.8.2 General

Software-based parameterisation of safety-related parameters shall be considered as safety-related aspects of SRP/CS design to be described in the software safety requirement specification. Software-based parameters include

— variant coding (e.g. country code, left-hand/right-hand steering),

— parameters (e.g. value for low idle speed, engine characteristic diagrams), and

— calibration data (e.g. vehicle specific, limit stop for throttle setting).

7.8.3 Prerequisites

The following are the prerequisites for software-based parameterisation:

— software project plan (see 7.1.1 to 7.1.4.6);

— software requirements [see 7.2.5 a) and b)];

— software architecture (see 7.3.4.1 to 7.3.4.5);

— software verification plan (see ISO 25119-4:2018, Clause 6);

— tested software components according to 7.4.4.1.

7.8.4 Requirements

7.8.4.1 Data integrity

The integrity of data used for parameterisation shall be maintained, and unauthorized modifications shall be prevented. This shall be achieved by applying measures to control:

a) the range of valid inputs;

b) data corruption before and after transmission, including:

   — checking configuration data for a valid range;

   — performing plausibility checks on configuration data;

   — using redundant data storage;

   — using error detecting and correcting codes;

c) the errors from the parameter transmission process;

d) the effects of incomplete parameter transmission;

e) the effects of faults and failures of hardware and software of the tool used for parameterisation.

7.8.4.2 Executable code in parameter data

Parameter data shall not contain executable code.

7.8.4.3 Configuration management

Software-based parameterisation shall be part of the version configuration management (see ISO 25119-4:2018, Clause 7).

7.8.4.4 Software-based parameterisation verification

The following verification activities shall be undertaken for software-based parameterisation:

— verification of the correct setting for each safety-related parameter (minimum, maximum and representative values);

— verification that the safety-related parameters have been checked for plausibility, by use of invalid values, etc.;

— verification that unauthorized modification of safety-related parameters is prevented;

— verification that the data/signals for parameterisation are generated and processed in such a way that faults cannot lead to a loss of the safety-related function.

7.8.5 Work products

The following work products shall be achieved during this phase:

a) verified safety-related software parameter configurations;

b) software validation plan resulting from 7.7.4.1 to 7.7.4.3;

c) software validation test specification resulting from 7.7.4.1 and 7.7.4.2;

d) software validation test report resulting from 7.7.4.3.

Annex A
(informative)

Example of agenda for assessment of functional safety at AgPL = e

A.1 Functions of system

A.1.1 Detailed presentation of the system, the components and the functional properties.

A.1.2 Overview of electrical components.

A.1.3 Hazard and risk analysis of the functions.

A.2 Hardware

A.2.1 Block diagram of the functions.

A.2.2 Layout and wiring diagram.

A.2.3 Environmental connections (interfaces).

A.3 Safety concept

A.3.1 Fundamental principles of the safety concept.

A.3.2 Definition of the “safe state(s)” or of the “degradation principle”.

A.3.3 Function of the safety concept.

A.3.4 Verification of safety-related function under fault conditions (“fault tolerance”).

A.3.5 Interaction of the safety concept with other systems/functions.

A.4 Safety analysis and safety data

A.4.1 Failure analysis (FMEA, FTA, etc.).

A.4.2 Internal monitoring functions.

A.4.3 Hardware failure rates (for different failure modes, e.g. from databases, FTA, FMEA, etc.).

A.4.4 Data for discovering internal faults (e.g. diagnostic discovery for component failure, failure recognition times, etc.).

A.4.5 Data for discovering external faults (e.g. network, sensor, switch, power supply failure, etc.).

A.5 Safety design process for phases of life cycle

A.5.1 Project management.

A.5.2 Documentation.

A.5.3 Specification phase.

A.5.4 Planning and development phase.

A.5.5 Integration phase.

A.5.6 Planning general validation/safety validation.

A.5.7 Results of general validation/safety validation.

A.6 Software development

A.6.1 Software safety concept.

A.6.2 Software structure.

A.6.3 Software test and documentation.

A.6.4 Development tools used.

A.6.5 Identification and tracking software modifications ("version control").

A.6.6 Protecting the implemented software from non-authorized modifications.

A.7 Verification and testing

A.7.1 Verification of system functions under failure-free conditions.

A.7.2 Verification of system functions under the influence of failures.

A.8 Documentation and safety documentation

A.8.1 Completeness.

A.8.2 Consistency.

A.9 Summary and assessment

Annex B
(normative)

Independence by software partitioning

B.1 Overview

In an area in which the state of the art is rapidly changing, this annex provides a means (software partitioning and its associated methods and measures) to aid the designer in proving independence for software components (see 7.3.4.7). Other means may be used with justification of required independence.

Adequate independence of software components is guaranteed by excluding particular fault effects violating this independence. For that purpose, methods and measures are necessary that should be implemented with at least medium effectiveness. For each method or measure given in this annex, a recommendation is given as to its effectiveness (see Tables B.1 and B.2):

— "high" means that it prevents the corresponding fault effect effectively;

— "medium" means that it prevents the corresponding fault effect partially;

— "none" means that it does not contribute to the prevention of the corresponding fault effect.

B.2 Terms, definitions and abbreviated terms

B.2.1 Terms and definitons

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

B.2.1.1
alive counter
accounting component initialised with "0" when the object to be monitored is created

NOTE The counter increases from time t–1 to time t as long as the object is alive. Finally, the alive counter shows the period of time for which the object has been alive within a network.

B.2.1.2
black-box test
test of a test object that does not require knowledge of its internal structure or its concrete implementation

B.2.1.3
bus guardian
independent component between a node and the bus that allows its node to transmit on the bus only when it is allowed to do so

NOTE The guardian needs to know when its node is allowed to access the bus; this is difficult to achieve in an event-triggered system, but conceptually simple in a time-triggered system.

B.2.1.4
message queue
⟨inter-process/inter-task communication⟩ procedure used for passing and exchanging data (messages) between asynchronous processes or tasks, including FIFO buffering

NOTE The queue is either managed by the operating system, or by an application. Message queuing enables synchronization and mutual exclusion.

B.2.1.5
minislotting
bus scheduling technique in which each node connected to the bus waits a certain period of time before it is permitted to access the bus again

B.2.1.6
mutual exclusion
synchronization mechanism to protect a sequence of statements that ought to appear to be executed indivisibly

B.2.1.7
partitions
resource entities, i.e. a virtual machine environment with allocated resources like quota of memory, I/O devices and CPU time

NOTE 1 Partitions are statically specified, and are created at system start-up.

NOTE 2 The programs running within a partition are always restricted to the resources allocated to the partition at system start-up by the system configuration.

B.2.1.8
partition
⟨within single micro-controller⟩ subsystem consisting of fixed assigned system resources where each partition contains one or more tasks

B.2.1.9
partition
⟨within scope of micro-controller network⟩ subsystem consisting of available memory, available CPU, and the I/O-functionality of the micro-controller

B.2.1.10
partitioning
software partitioning
fault containment technique which consists in precluding that a failure in a partition will propagate and cause other partitions to fail

NOTE 1 When applied to software, the intent of partitioning is to control the additional hazard created when a software partition shares its resources or part thereof with other partitions (e.g. processor and/or peripherals). Software partitioning is not intended to protect against the failure of each software partition but against the propagation of such failures.

NOTE 2 Software partitioning encompasses, on the one hand, space partitioning that addresses unauthorized data access and illegitimate command of peripherals assigned to other partitions. On the other hand, time partitioning that focuses on disturbance of the timing of events in other partitions (scheduling, order of execution, etc.) is also addressed.

B.2.1.11
pipe
one-way communication between two processes/tasks, including buffering according to the FIFO principle

NOTE A pipe is built on top of message queues and used like a standard I/O interface, i.e. the output of one process/ task is used as input for another.

B.2.1.12
real addressing
absolute addressing
explicit identification of a memory location or of a peripheral device

NOTE See *relative addressing* (B.2.1.14).

B.2.1.13
redundancy
multiplication, i.e. in most cases the duplication, of components of a system with the intention of increasing the reliability of the system

NOTE This action is usually taken in cases where a backup or fail-safe is necessary.

B.2.1.14
relative addressing
identification of a memory location or of a peripheral device as an offset from some other address

NOTE See *real addressing* (B.2.12).

B.2.1.15
shared memory
memory used for inter-task communication

NOTE The shared memory is a designated memory area, which is directly accessible to more than one task. Semaphores are used to avoid interference and to ensure memory integrity. This method of inter-process/inter-task communication is faster than exchanging data by operating system services.

B.2.1.16
semaphore
non-negative integer variable, whose value is decreased when entering a critical program section and incremented again when leaving this section

NOTE Semaphores are used for synchronization when several tasks access a common resource such as a common data space.

B.2.1.17
software component
implementation of one or more functions in software

NOTE A software component is a logically separable part of the software, and consists of one or more software components and/or software units. Within the software architecture, software components are realized by partitions and tasks.

B.2.1.18
software unit
smallest independent piece of software, which can be independently translated, and which can be tested with the relevant data whether it performs to specification

NOTE The software unit is an atomic software component.

B.2.1.19
software partitions
runtime environment with separate system resources assigned

B.2.1.20
system resources
all resources required by the software to operate

EXAMPLE CPU time, I/O devices, memory.

B.2.1.21
task
runtime entities which are executed within the resource budget of partitions where each task has its own stack and priority

NOTE A task is executed under the control of the scheduler according to the task priority assigned to it and the selected scheduling policy.

B.2.1.22
independence of software
exclusion of unintended interactions between software components, as well as freedom from impact on the correct operation of another software component, resulting from design and/or implementation errors of a software component

B.2.1.23
user mode
one of two execution modes of the CPU, the other being kernel mode

NOTE User mode is non-privileged, i.e. references or accesses to memory and I/O space are checked for authorization. In user mode, specific memory areas are assigned and accessible while memory areas of other processes, for example, are prohibited. In contrast, kernel mode gives privileged access and execution rights.

B.2.1.24
watchdog
timer process that, if not reset within a certain period by a software component, assumes that the software component is in error

B.2.2 Abbreviated terms

For the purposes of this annex, the following abbreviated terms apply.

CRC cyclic redundancy check

CPU central processing unit

MMU memory management unit

MPU memory protection unit

B.3 Objectives

The first objective is to control hazards that can occur in subsystems so that they cannot affect other subsystems.

NOTE Software hazards can occur due to design and implementation errors of a software component that could then disturb the correct operation of other software components sharing resources with it.

The second objective is to specify how to demonstrate adequate independence of software components by software partitioning.

B.4 General

In order to achieve adequate independence of software components, the system resources shall be assigned when practicable to independent subsystems or partitions, each representing a particular runtime environment. System resources include the CPU, memory, bus, I/O-channels and resources of the operation system such as file handles.

The use of software partitioning is not restricted to the co-existence of software of different SRL in the same runtime environment. It can also support

a) changes in a partition without re-verification of unmodified software partitions, and

b) coexistence of software of different nature (in-house, third party).

Partitioning is usually not possible without adequate support from the hardware.

In order to isolate multiple partitions in a shared resource environment, the hardware shall, when practicable, provide the operating system with the ability to restrict memory spaces, processing time, and access to I/O for each individual partition.

NOTE Partitions can be allocated within a single micro-controller or allocated to several micro-controllers within a micro-controller network.

Depending on the selected architecture, two approaches can be used:

a) several partitions within a single micro-controller;

b) several partitions within the scope of an ECU network.

B.5 Requirements

B.5.1 General requirements

B.5.1.1 SRL

That part of the software that implements the support for partitioning implementation shall have the same or higher SRL than the highest SRL associated with the software partitions.

NOTE In general, the software providing or supporting partitioning is part of the operating system.

B.5.1.2 Software architecture

The concept of software partitioning shall be taken into account when specifying the software architecture.

B.5.2 Several partitions within a single microcontroller

B.5.2.1 General

See Figure B.1.

NOTE Tasks within a partition are not independent of one another.

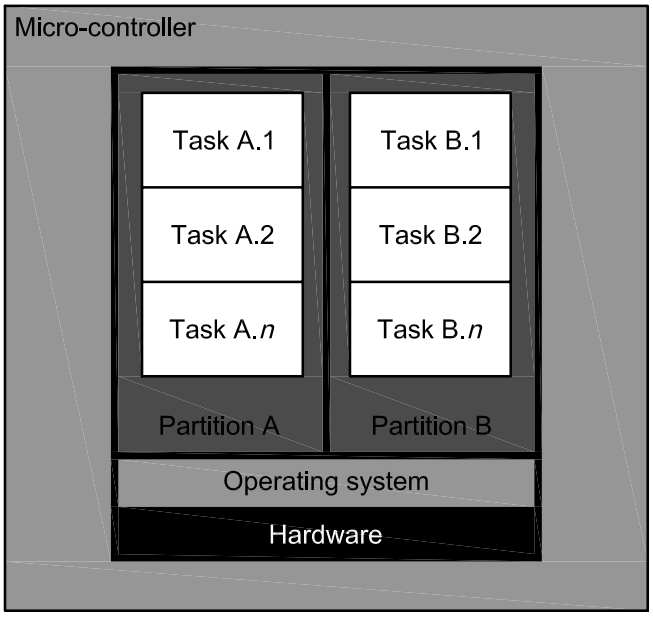


Figure B.1 — Several partitions within a single micro-controller

B.5.2.2 Software partitioning methods/measures

Each partition shall implement the methods and measures given in Table B.1 so as to prevent each of the following fault effects and ensure adequate independence of software components:

— memory corruption (unintended writing to memory of another partition);

— blocking of partitions (due to communication deadlocks);

— wrong allocation of processor execution time;

— wrong communication peer (sender sends messages to the wrong recipient or masquerades as a sender other than himself);

— corruption of I/O interface by unintended writing to an I/O interface of another partition.

B.5.2.3 Software partitioning effectiveness

The methods and measures listed in Table B.1 shall be implemented so that all appropriate fault effects can be handled at least with medium effectiveness to ensure adequate independence of software components.

Table B.1 — Methods and measures within micro-controller

Method/measure		Fault effect				
		Memory corruption	Blocking of partitions	Wrong processor execution time	Wrong communication peer	Corruption of I/O interface
Verification of communication						
1	Unambiguous bidirectional communication object[a]	None	None	None	Medium	None
2	Strictly two unidirectional communication objects[b]	None	None	None	Medium	None
3	Identifications for identification and/or acknowledgement[c]	None	None	None	High	None
4	Asynchronous data communication[d]	None	High	None	None	None
Allocation of processor execution time						
5	No priority-based scheduling[e]	None	None	Medium	None	None
6	Time slicing method[f]	None	None	High	None	None
Allocation of system resources						
7	Memory protection mechanisms[g]	High	None	None	None	None
8	Verification of safety critical data[h]	High/ medium	None	None	None	None
9	Static analysis[i]	Medium	None	None	None	None
10	Static allocation[j]	Medium	Medium	Medium	Medium	Medium

Communication objects in items 1 and 2 between partitions are, e.g. pipes, message queues, shared memory. These should not be used for synchronizing partitions.

Access shall be synchronized between both partitions using shared memory for communication e.g. by using semaphores.

Blocking read or write access should be prohibited by design when using message queues.

An MMU enables the concept of virtual address space. This prevents a task of one partition from corrupting the memory space of another task by unintended writing into that memory space, since every partition has its own address space. Usage of MMU requires support of the operating system for that feature. Provisions shall be made so that the MMU cannot be ignored. Therefore, the tasks run in a so-called user mode, and real addressing mode is not used.

[a] Exactly one bi-directional communication object is used between two partitions respectively for data exchange.

[b] Exactly two uni-directional communication objects are used between two partitions respectively for data exchange.

[c] Uses unique numbers for identifying communication peers and/or acknowledges receipt of messages by the communication peer.

[d] In using asynchronous data communication as described in this item, there is no waiting state completed by the communication itself.

[e] The partitions are considered coequally in allocating processor execution time and the same priority is assigned to all of them. Regarding the allocation of processor time in this item, there has to be some spare time/buffer in each processor cycle because of incoming interrupts.

[f] The time slicing method specifies a scheduling algorithm based on a predetermined fixed schedule, repetitive with a fixed period. Using the time slicing method, the allocation of processor execution time takes place through a static allocation table. Thus for each task, a fixed point in time is predetermined for activating the task. Usage of a time slicing method precludes priority-based scheduling.

[g] The memory protection mechanisms referred to specify processors with, for example, MMU or MPU.

[h] RAM locations containing safety critical data are verified by additional measures. This can be accomplished by, for example, using CRC or redundant storage. The effectiveness of this measure depends very heavily on the verification quality.

[i] Specifies adequate static analysis methods for reviewing pieces of code that access memory locations containing safety-related data.

[j] Resources are allocated statically during initialisation.

B.5.3 Several partitions within the scope of a micro-controller network

B.5.3.1 General

The micro-controller network can consist of several processors on a single or multiple ECU communicating via an internal processor communication or an external data bus communication (e.g. CAN).

Software components can be executed within their respective partition in their respective micro-controllers, as illustrated in Figure B.2.

Also, the micro-controller network can consist of several processors on a single ECU communicating via an internal data bus (internal processor communication). This is illustrated in Figure B.3. The following examinations apply analogously.

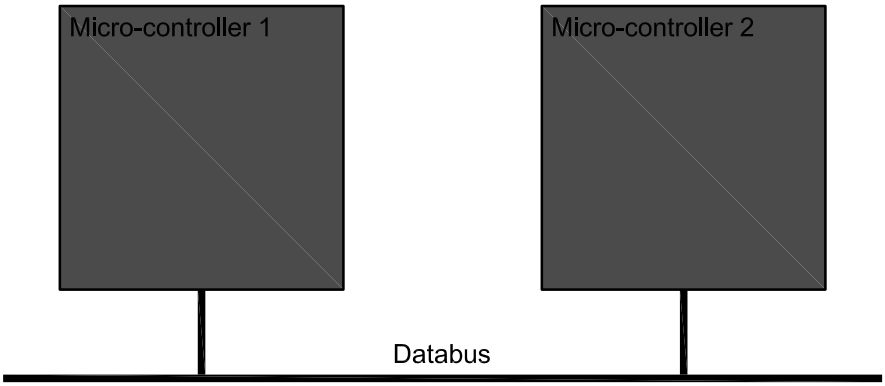


Figure B.2 — Several partitions within scope of micro-controller network

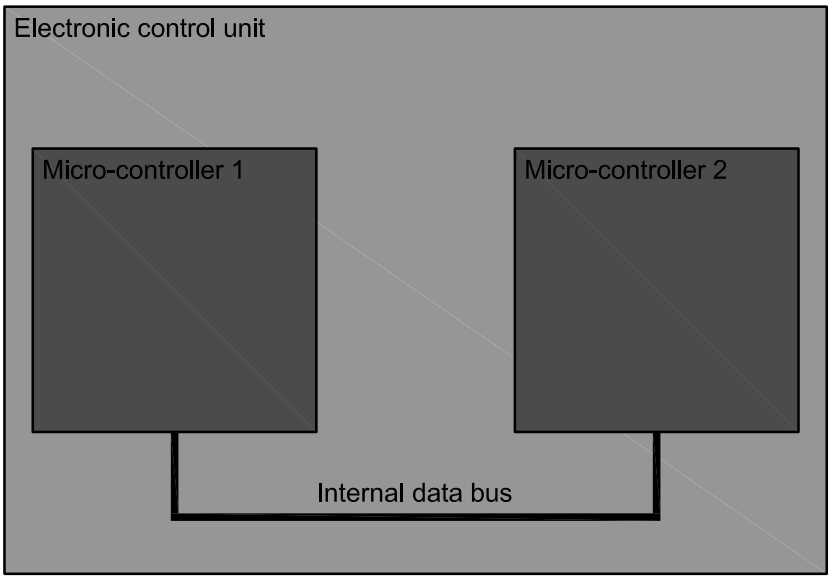


Figure B.3 — Several partitions within scope of multi-processor ECU

B.5.3.2 Methods for multi-processor partitioning

In order to guarantee adequate independence of software components within a micro-controller network, it shall be ensured that none of the following fault effects prevents the correct execution of the safety-related function:

— failure of communication peer (communication peer is not available);

— unintended message repetition (the same message is unintentionally sent again);

— message loss (message is lost during transmission);

— insertion of messages (receiver unintentionally gets an additional message, interpreted as having correct source and destination addresses);

— re-sequencing (order in which data has been sent changed during transmission, i.e. the data are not received in the same order as in which it was sent);

— message corruption (one or more data bits are changed in the message during transmission);

— message delay (the message is received correctly, but not in time);

— blocking access to data bus (a faulty node does not adhere to expected patterns of use and makes excessive demands for service, thereby reducing that available to other nodes, e.g. while wrongly waiting for non-existing data);

— constant transmission of messages, known as “babbling idiot” (a faulty node transmits constantly, thereby compromising the operation of the entire bus);

— multi-processor partitioning that involve safety related function, shall be used only if the safe state can be entered in case of a communication break or other intolerable errors.

B.5.3.3 Multi-processor partitioning effectiveness

The methods and measures listed in Table B.2 shall be implemented so that all appropriate fault effects can be handled at least with medium effectiveness to ensure adequate independence of software components in order to cover the target DC.

Table B.2 — Methods and measures within the scope of a micro-controller network

Method/ measure		Fault effect								
		Failure of communication peer	Unintended message repetition	Message loss	Insertion of messages	Re-sequencing	Message corruption and/ or masquerading	Message delay	Blocking access to data bus	Constant transmission of messages
Verification of communication										
1	Keep alive messages	High	None	None	None	None	Medium	None	None	None
2	Alive counter	None	High	Medium	Medium	None	None	None	None	Medium
3	CRC	None	None	None	None	None	High	None	None	None
4	Sequence number	None	High	High	High	High	None	None	None	Medium
5	Message repetition	None	None	High	None	Medium	Medium	None	None	None
6	Watchdog	High	None	Medium	High	None	None	High	High	High
Bus allocation										
7	Time-triggered data bus	High	High	None	High	None	None	High	None	None
8	Bus guardian	None	None	None	None	None	Medium	None	None	High
9	Minislotting	None	None	None	None	None	None	Medium	None	High
Using CRC (item 3), it shall be taken into account that the residual error rate of the CRC implemented in the bus system may not be sufficient, in which case an additional CRC at the application level is recommended. Blocking access is caused by a micro-controller connected to the bus, which permanently accesses the bus and thereby prevents other micro-controllers from getting access to the bus.										

Bibliography

[1] ISO 3600, *Tractors, machinery for agriculture and forestry, powered lawn and garden equipment — Operator's manuals — Content and format*

[2] ISO/IEC 8652, *Information technology — Programming languages — Ada*

[3] ISO 9001, *Quality management systems — Requirements*

[4] ISO/IEC 9899, *Programming languages — C*

[5] ISO 12100, *Safety of machinery — General principles for design — Risk assessment and risk reduction*

[6] ISO/IEC 14882, *Programming languages — C++*

[7] ISO 15003, *Agricultural engineering — Electrical and electronic equipment — Testing resistance to environmental conditions*

[8] IATF 16949:2016, *Quality management systems — Particular requirements for the application of ISO 9001:2008 for automotive production and relevant service part organizations*

[9] IEC 61000-4-1, *Electromagnetic compatibility (EMC) — Part 4-1: Testing and measurement techniques — Overview of IEC 61000-4 series*

[10] IEC 61496-1, *Safety of machinery — Electro-sensitive protective equipment — Part 1: General requirements and tests*

ICS 35.240.99; 65.060.01

Price based on 59 pages