INTERNATIONAL STANDARD

ISO 25119-3

First edition
2010-06-01

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 25119-3:2010(E)

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web 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.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

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.

ISO 25119-3 was prepared by Technical Committee ISO/TC 23, *Tractors and machinery for agriculture and forestry*, Subcommittee SC 19, *Agricultural electronics*.

ISO 25119 consists of the following parts, under the general title *Tractors and machinery for agriculture and forestry — Safety-related parts of control systems*:

— *Part 1: General principles for design and development*

— *Part 2: Concept phase*

— *Part 3: Series development, hardware and software*

— *Part 4: Production, operation, modification and supporting processes*

Introduction

ISO 25119 sets out an approach to the design and assessment, for all safety life cycle activities, of safety-relevant systems comprising electrical and/or electronic and/or programmable electronic components (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 municipal equipment. It covers the possible hazards caused by the functional behaviour of E/E/PES safety-related systems, as distinct from hazards arising from the E/E/PES equipment itself (electric shock, fire, nominal performance level of E/E/PES dedicated to active and passive safety, etc.).

The control system parts of the machines concerned are frequently assigned to provide the critical functions of the *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 critical 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 protective measures (both SRP/CS and non-SRP/CS) with the end result of achieving a safe condition.

ISO 25119 allocates the ability of safety-related parts to perform a critical 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 are considered: systematic, common-cause and random.

In order to guide the designer during design, and to facilitate the assessment of the achieved performance level, ISO 25119 defines an approach based on a classification of structures 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 to the control systems of protective equipment (e.g. interlocking devices, pressure sensitive devices).

ISO 25119 adopts a customer risk-based approach for the determination of the risks, while providing a means of specifying the target 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.

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

Part 3: Series development, hardware and software

1 Scope

This part of ISO 25119 provides general principles for the series development, hardware and software 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 municipal equipment (e.g. street-sweeping machines). It specifies the characteristics and categories required of SRP/CS for carrying out their safety functions.

This part of ISO 25119 is applicable to the safety-related parts of electrical/electronic/programmable electronic systems (E/E/PES). As these relate to mechatronic systems, it does not specify which safety functions or categories are to be used in a particular case.

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

2 Normative references

The following referenced documents are indispensable for the application 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 25119-1:2010, *Tractors and machinery for agriculture and forestry — Safety-related parts of control systems — Part 1: General principles for design and development*

ISO 25119-2:2010, *Tractors and machinery for agriculture and forestry — Safety-related parts of control systems — Part 2: concept phase*

ISO 25119-4:2010, *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 given in ISO 25119-1 apply.

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
EUC	equipment under control
FMEA	failure mode and effects analysis
FMECA	failure mode effects and criticality analysis
EPROM	erasable programmable read only memory
FSM	functional safety management
FTA	fault tree analysis
HAZOP	hazard and operability study
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
SRS	safety-related system
UML	unified modelling language.

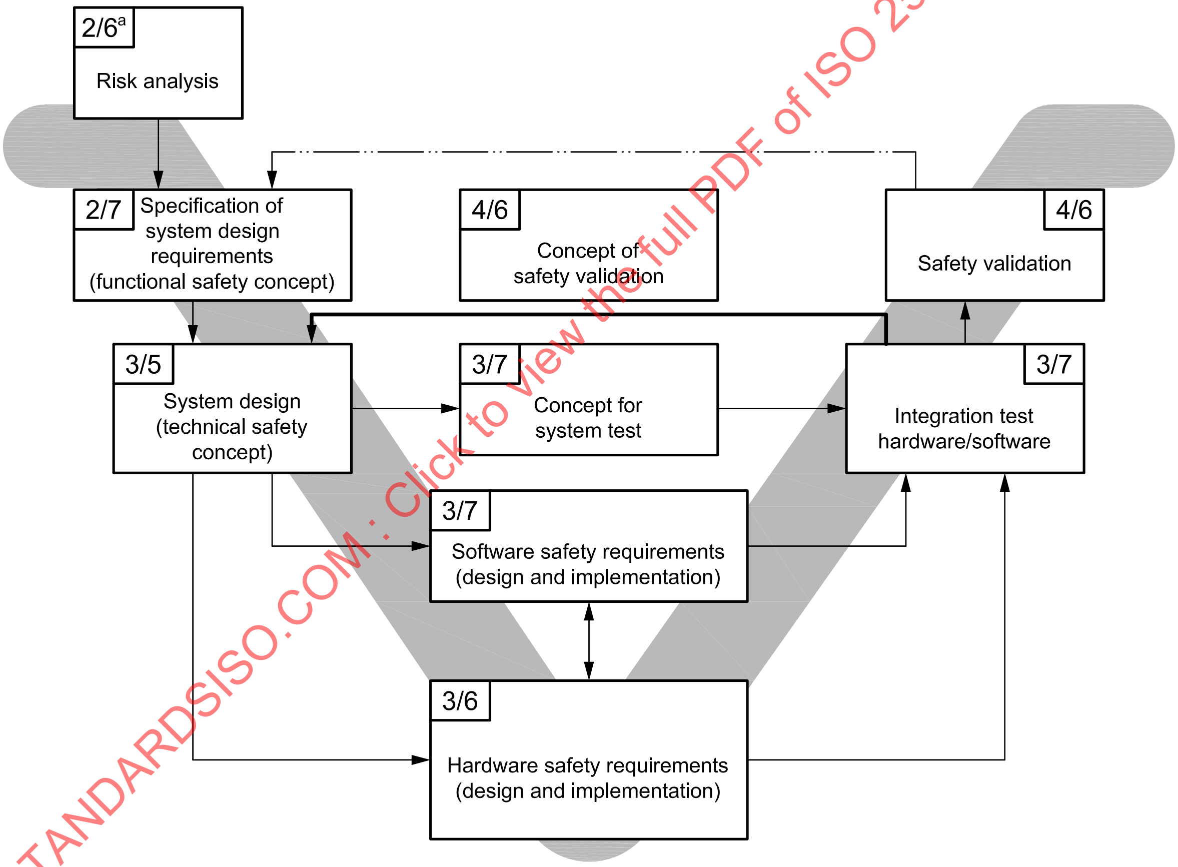
5 System design

5.1 Objectives

The objective is to define a development process for producing a 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.



2/6[a]
Risk analysis
2/7
Specification of system design requirements (functional safety concept)
4/6
Concept of safety validation
4/6
Safety validation
3/5
System design (technical safety concept)
3/7
Concept for system test
3/7
Integration test hardware/software
3/7
Software safety requirements (design and implementation)
3/6
Hardware safety requirements (design and implementation)

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 to the clause in that document: 2/6 is ISO 25119-2:2010/Clause 6, 3/5 is ISO 25119-3:2010/Clause 5, and so on.

Figure 1 — Structuring of safety requirements

5.3 Prerequisites

Before beginning system design, define the safety-related function requirements, application and operation environment.

5.4 Requirements

5.4.1 Structuring safety requirements

The functional safety concept specifies the basic functioning of the safety-related system with 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 in the form of technical safety requirements. This system architecture is comprised of both hardware and software.

The hardware safety requirements refine and solidify the requirements of the technical safety concept. 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 safety concept requirements during system design, thereby providing a basis for error-free system design.

5.4.2 Functional safety concept

5.4.2.1 General requirements of functional safety concept

Safety functions are normally identified during the system risk analysis, and the functional safety concept document includes the functional safety requirements for the system.

The implementation for each safety concept requirement shall consider the following.

- **Feasibility**

  When listing functional safety requirements, attention shall be paid to the feasibility of the requirement, considering constraints, such as available technology, as well as financial and time resources. The persons in charge of implementation shall understand and accept the technical safety requirements.

- **Unambiguousness**

  The functional safety requirements shall be formulated as precisely and unambiguously as possible.

  NOTE A functional safety requirement is unambiguously formulated when it permits only one interpretation by the anticipated readers.

- **Consistency**

  Functional safety requirements shall not be self-contradicting (internal consistency), nor shall they contradict other requirements (external consistency).

  Analyses of the requirements and comparisons between different requirements are necessary to ensure external consistency. This is a requirement management task.

- **Completeness**

  The functional safety concept shall take all relevant norms, standards and statutory regulations into account.

The functional safety concept shall take into account all relevant safety goals derived from the risk analysis according to ISO 25119-2.

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

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

2) the requirements from change management (see ISO 25119-4:2010, Clause 10) shall be met and, for this reason, the functional safety requirements shall be structured and formulated to provide support for a modification process;

3) the functional safety requirements shall be reviewed (see ISO 25119-4:2010, Clause 6).

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

5.4.2.2 Specification of the functional safety concept

This clause presents the information that is required to be specified in the functional safety concept. The functional safety concept may be derived from the machine failure scenarios evaluated during a risk analysis.

Each failure scenario description shall include the following:

- environmental conditions (moving on an ice covered road, up-hill, down-hill, weather, etc.);

- machine conditions (engine running, in-gear, standing still, etc.);

- resulting AgPL;

- safe state descriptions (engine stopped, valve off, transmission in park, continue function at reduced performance, etc.).

5.4.3 Technical safety concept

5.4.3.1 General requirements of technical safety concept

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

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

- other technical safety requirements,

- functional safety requirements, or

- safety goals and objectives.

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

Just as for the *functional* safety concept, the implementation of each technical safety concept requirement shall take account of feasibility, unambiguousness, consistency and completeness.

- **Feasibility**

  When listing technical safety requirements, attention shall be paid to the feasibility of the requirement considering constraints, such as available technology, as well as financial and time resources. Those in charge of implementation shall understand and accept the technical safety requirements.

- **Unambiguousness**

  The technical safety requirements shall be formulated as precisely and unambiguously as possible.

  NOTE A technical safety requirement is unambiguously formulated when it permits only one interpretation by the anticipated readers.

- **Consistency**

  Technical safety requirements shall not be self-contradicting (internal consistency), nor shall they contradict other requirements (external consistency).

  Analyses of the requirements and comparisons between different requirements are necessary to ensure external consistency. This is a requirement management task.

- **Completeness**

  The technical safety concept shall take the following into account:

  1) all safety objectives and functional safety requirements;

  2) all relevant norms, standards and statutory regulations;

  3) 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:

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

  5) the requirements from change management (see ISO 25119-4:2010, Clause 10) shall be met and, for this reason, the technical safety requirements shall be structured and formulated to provide support for a modification process;

  6) the technical safety requirements shall be reviewed (see ISO 25119-4:2010, Clause 6).

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

5.4.3.2 Specification of the technical safety concept

5.4.3.2.1 General

The technical safety concept shall include hardware and software safety requirements sufficient for the design of the unit of observation, and shall be determined in accordance with 5.4.3.1.

5.4.3.2.2 States and times

The behaviour of the unit of observation, its modules 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 unit of observation 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.

5.4.3.2.3 Safety architecture, interfaces and marginal conditions

The safety architecture and its sub-modules shall be described. In particular, the technical measures shall be specified. The technical safety concept shall separately describe the following modules (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 modules;
- signal transmission from/to systems external to the unit of observation;
- power supply.

The interfaces between the modules of the unit of observation, interfaces to other systems and functions in the machine, as well as user interfaces, shall be specified.

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

6 Hardware

6.1 Objectives

The objective is to define acceptable hardware architectures for safety-related control systems.

6.2 General

Improving the hardware structure of the safety-related parts of a control system can provide measures for avoiding, detecting or tolerating faults. Practical measures can include redundancy, diversity and monitoring.

In general, the following fault criteria should 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 considered highly unlikely.

6.3 Prerequisites

The prerequisite is AgPL$_r$, determined for each safety function to be realized by the hardware.

6.4 Requirements

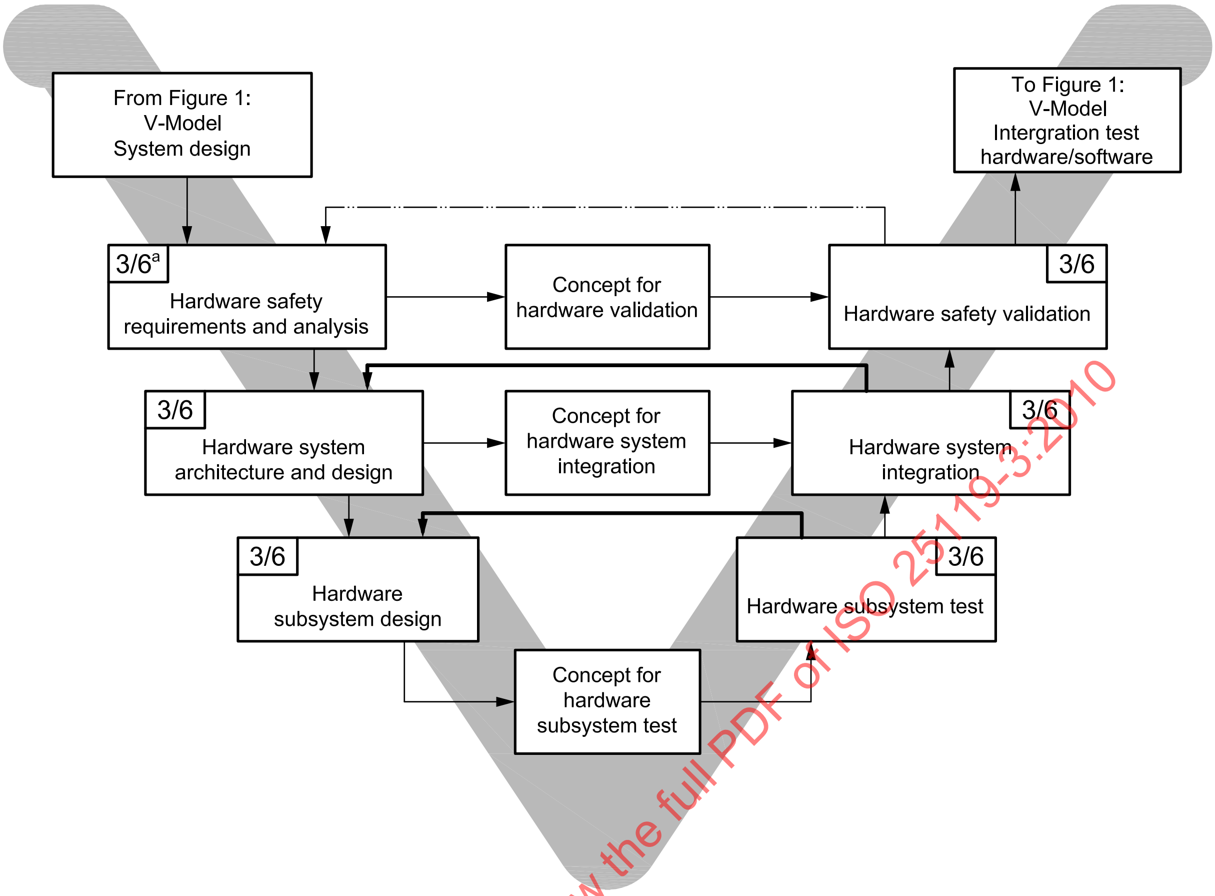
The hardware development process shall begin at the system level where safety functions and associated requirements are identified (see Figure 2).

The hardware safety analysis shall be used to identify the performance level (AgPL$_r$) for each system safety function (see ISO 25119-2).

The designer shall group functions into appropriate architectures (hardware category) with associated MTTF$_{dC}$, DC and CCF.

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

Each phase of the development cycle shall be verified.


From Figure 1:
V-Model
System design
To Figure 1:
V-Model
Intergration test
hardware/software
3/6[a]
Hardware safety requirements and analysis
Concept for hardware validation
3/6
Hardware safety validation
3/6
Hardware system architecture and design
Concept for hardware system integration
3/6
Hardware system integration
3/6
Hardware subsystem design
3/6
Hardware subsystem test
Concept for hardware subsystem test

Key

———►	result
◄———	verification
◄–··–—	validation

[a] The first of two numbers separated by a slash refers to this part of ISO 25119 and the second to Clause 6.

Figure 2 — Hardware development V-model

The design procedure for the hardware system architecture is as follows.

a) Select a hardware category (see ISO 25119-2:2010, Annex A).

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 25119-2:2010, Annex B).

e) Determine and verify that the DC meets the required level (see ISO 25119-2:2010, Annex C).

f) Consider CCF (see ISO 25119-2:2010, Annex D).

g) Consider systematic failures (see ISO 25119-2:2010, Annex E).

h) Consider other safety functions (see ISO 25119-2:2010, Annex F).

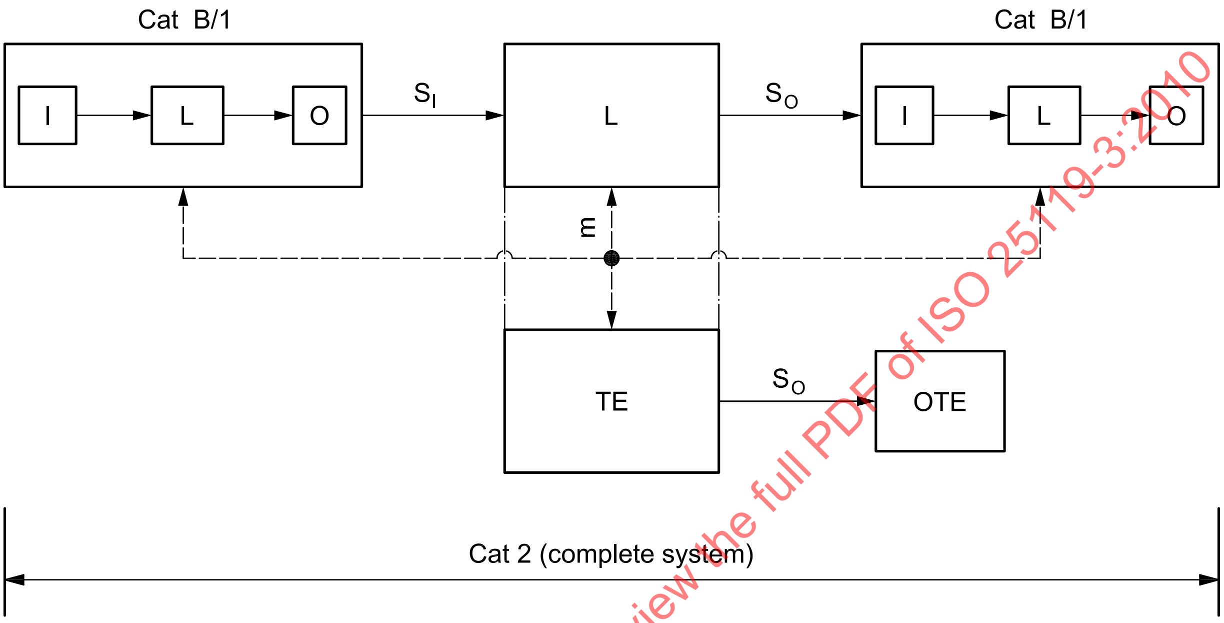
NOTE Iteration could be required for the above steps.

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 25119-2:2010, Annex A.

When a safety function is realized by an integrated combination of multiple hardware categories, the resulting safety function, AgPL, is limited by the overall hardware category: MTTF_{dC}, DC, SRL, CCF, etc. (see Figure 3).

To determine the overall SRL, see 7.3.4.7.



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 with maximum AgPL for category 2

6.6 Work products

The following work products are 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.

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 25119-4:2010, Clause 10.

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 should 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 should be stated during 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$_r$ for each safety function to be realized,
- the project plan (including system development plan),
- the system verification plan,
- the technical safety concept,
- the system design specification, and
- the safety plan.

7.1.4 Requirements

7.1.4.1 Phase determination

For the software development process, it shall be determined which 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 can be carried out according to Figure 4, without modification, or individual phases can 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 can 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 at the laboratory 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 should be determined which software safety requirements shall be applied to which integration steps.

7.1.4.5 Supporting processes

Supporting processes shall be planned and implemented as part of the software development process:

a) the work products shall be documented according to ISO 25119-4:2010, Clause 12;

b) changes to the software shall be dealt with according ISO 25119-4:2010, Clause 10;

c) the work products shall be subject to the configuration management process.

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

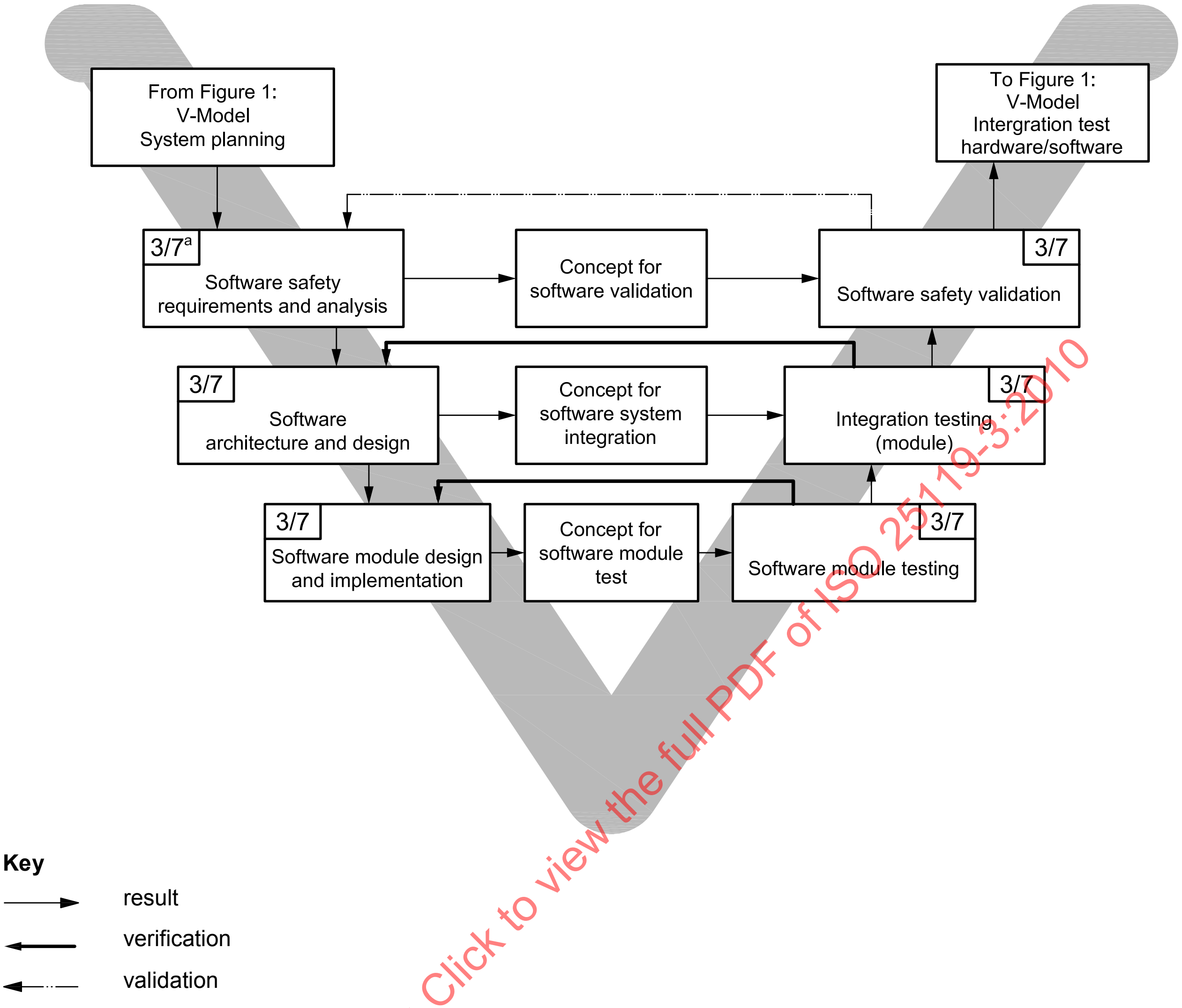
7.1.4.6 Phases of software development

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.

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 can 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.



[a] The first of the two numbers separated by a slash refers to this part of ISO 25119 and the second 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 "+" or "o":

+ 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.

In a table, an "o" may appear to the right of a "+". This means a more rigorous measure or technique is available for the same 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 7.1.4.2 to 7.1.4.4 (see also ISO 25119-4:2010, 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 should be derived from the requirements of the technical safety concept of the system, and labelled as software safety requirements. At least the following should 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 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.

NOTE 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 may be repercussions on the hardware architecture. For this reason, close cooperation between the hardware development and the software development is necessary.

7.2.3 Prerequisites

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

— software project plan according to 7.1.4.2 to 7.1.4.4;

— technical safety concept according to 5.4.3;

— 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 design methods[a]	7.2.4.1.2	+	+	o	o
2b	Semi-formal design methods	7.2.4.1.3	o	o	+	+
2c	Formal design methods	7.2.4.1.4	o	o	+	+
3	Computer-aided specification tools	7.2.4.1.5	o	o	+	+
4a	Inspection of software safety requirements[a]	ISO 25119-1:2010, 3.28	+	+	+	+
4b	Walk-through of software safety requirements	ISO 25119-1:2010, 3.56	+	+	o	o

See 7.1.4.7 for instructions on the use of this and the other tables.

[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 alternative or equivalent techniques/measures need be satisfied.

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.

NOTE 3 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 design 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 some stage in its development, i.e. specification, design or coding, typically by means of natural language, diagrams, figures, etc.

7.2.4.1.3 Semi-formal design methods

7.2.4.1.3.1 Aim

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

7.2.4.1.3.2 Description

Semi-formal methods for software design shall be used to provide a means of developing a description of a system at some stage in its development, i.e. specification, design or coding.

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

The description shall in some cases be analysed by machine or animated to display various aspects of the system behaviour. Animation shall give extra confidence that the system meets the requirements.

7.2.4.1.4 Formal design methods

7.2.4.1.4.1 Aim

The development of software in a way that is based on mathematics shall include formal design and formal coding techniques.

7.2.4.1.4.2 Description

Formal methods shall provide a means of developing a description of a system at some 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 in some cases 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 shall give extra confidence that the system meets the real requirement as well as 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 the form of a database that can be automatically inspected to assess consistency and completeness. The specification tool shall animate various aspects of the specified system for the user. 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 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 functional requirements and the safety requirements, the latter should be clearly identified as such.

7.2.4.3 Level of detail

The software safety requirements specification shall contain enough detail to implement the safety function in the software.

7.2.4.4 Consistency

The software safety requirements shall be retraceable and consistent with the specifications of the system safety requirements and the system architecture.

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 functions.

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

  NOTE 3 This refers in particular to the testability of the safety 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 requirements for the safety integrity of the software are

- the SRL for each of the functions listed above, and
- 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 examined to determine it they are consistent with the technical safety concept. 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).

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) acceptance criteria for the software safety requirements, according to 7.2.4.6;

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

7.3 Software architecture and design

7.3.1 Objectives

The objective of the software architecture is the implementation and structuring of all software requirements using software components. It should be ensured 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, should be described.

7.3.3 Prerequisites

The software architecture is only to be started after the software safety requirements specification has reached a sufficient degree of maturity.

7.3.4 Requirements

7.3.4.1 Software architecture and design methods

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

Table 2 — Software architecture and design

Technique/measure[a]		Subclause	SRL=B	SRL=1	SRL=2	SRL=3
1a	Informal design methods[a]	7.2.4.1.2	+	+	o	o
1b	Semi-formal design methods	7.2.4.1.3	o	o	+	+
1c	Formal design methods	7.2.4.1.4	o	o	+	+
2	Computer-aided specification tools	7.2.4.1.5	o	o	+	+
3	Bottom-up failure analysis	7.3.4.1.1	o	o	o	+
4	Top-down failure analysis	7.3.4.1.2	o	o	+	+
5a	Inspection of software architecture[a]	ISO 25119-1:2010, 3.28	+	+	+	+
5b	Walk-through of software architecture	ISO 25119-1:2010, 3.56	+	+	o	o

See 7.1.4.7 for instructions on the use of this and the other tables.

[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 alternative or equivalent techniques/measures need be satisfied.

7.3.4.1.1 Failure analysis — Bottom-up methods

7.3.4.1.1.1 Aim

The analysis of events or combinations of events that will lead to a hazard or serious consequence shall be supported by bottom-up methods.

7.3.4.1.1.2 Description

Starting at an event which would be the immediate cause of a hazard or serious consequence (the “bottom event”), analysis shall be carried out along a tree path. Combinations of causes are described with logical operators (and, or, etc.). Intermediate causes are analysed in the same way, and so on, back to basic events where analysis stops. The method is graphical, and a set of standardized symbols is used to draw the fault tree.

EXAMPLE FMEA, HAZOP or FMECA method.

7.3.4.1.2 Failure analysis — Top-down methods

7.3.4.1.2.1 Aim

Top-down methods shall be used to rank the criticality of components which could result in injury, damage or system degradation through single-point failures, in order to determine which components might need special attention and necessary control measures during design or operation.

7.3.4.1.2.2 Description

Criticality can be ranked in many ways. The criticality number is a function of a number of parameters, most of which have to be measured. A very simple method for criticality determination is to multiply the probability of component failure by the damage that could be generated; this method is similar to simple risk factor assessment.

NOTE These techniques are mainly intended for the analysis of hardware systems, but there have also been attempts to apply this approach to software failure analysis, examples being FTA, ETA and "cause and effect" diagrams.

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

c) the understanding of the developers and others involved,

d) capability for software modification, and

e) verification and validation.

7.3.4.3 Software architecture structure

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

NOTE At the top level of the software architecture, there is usually a separation into basic software and application software.

7.3.4.4 Level of detail

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

When developing the software architecture, the extent of the safety-related 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 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).

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 is limited to the component implemented with the lowest SRL, unless adequate independence (see Annex B) between the software components can be demonstrated.

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 module design and implementation

7.4.1 Objectives

The first objective is to specify in detail the behaviour of the safety-related software modules 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 module design and implementation:

- software project plan (see 7.1.4.2 to 7.1.4.4);
- 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 25119-4:2010, Clause 6).

7.4.4 Requirements

7.4.4.1 Software module 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 modules and components (if available)	7.4.4.1.5	o	o	+	+
2 Design methods					
2.1a Informal design methods[a]	7.2.4.1.2	+	+	o	o
2.1b Semi-formal design methods	7.2.4.1.3	o	o	+	+
2.1c Formal design methods	7.2.4.1.4	o	o	+	+
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	7.4.4.1.8	o	o	o	+
2.5 Library of trusted/verified software modules and components	7.4.4.1.9	+	+	+	+
2.6 Computer-aided design tools	7.4.4.1.10	o	o	o	+
3 Design and coding standard					
3.1 Use of coding standard	7.4.4.1.11	o	o	+	+
3.2 No dynamic variables or objects	7.4.4.1.12	o	o	o	+
3.3 Limited use of interrupts	7.4.4.1.13	o	o	o	+
3.4 Defined use of pointers	7.4.4.1.14	o	o	o	+
3.5 Limited use of recursion	7.4.4.1.15	o	o	o	+
4 Design and coding verification					
4a Inspection of software design and/or source code[a]	ISO 25119-1:2010, 3.28	+	+	+	+
4b Walk-through of software design and/or source code	ISO 25119-1:2010, 3.56	+	+	o	o

See 7.1.4.7 for instructions on the use of this and the other tables.

[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 alternative or equivalent techniques/measures need 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 as much as possible, 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 or their subsets are preferred to special-purpose languages.

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

- block structure,
- translation time checking, and
- run-time type and array bounds checking.

The language shall encourage:

- the use of small and manageable software modules,
- restriction of access to data in specific software modules,
- definition of variable sub-ranges, and
- 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, appropriate libraries of pre-existing software modules, a debugger, and tools for both version control and development. At the time of developing this part of ISO 25119, it is not clear whether object-orientated languages are to be preferred to procedural languages. The following features which make verification difficult shall be avoided:

a) unconditional jumps, excluding subroutine calls;

b) recursion;

c) pointers, heaps, or any type of dynamic variables or objects;

d) handling of interrupts at source code level;

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

f) implicit variable initialization or declaration;

g) variant records and equivalence;

h) procedural parameters.

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, etc.

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 compiler or static analysis tool.

7.4.4.1.2.2 Description

When a strongly typed programming language is compiled or statically analysed, many checks need to be made on how variable types are used (for example, in procedure calls and external data access). Compilation 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 compiled units. The checks also ensure that the number and type of procedure arguments match, even when referenced from separately compiled software modules.

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 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 noted down, 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 modules and components (if available)

7.4.4.1.5.1 Aim

Trusted/verified software modules and components shall be used to avoid the need for software modules 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 modules and components are sufficiently free from systematic design faults and/or operational failures. Only in rare cases will the employment of trusted software modules and 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 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 component or software module can be sufficiently trusted if it is already verified to the required SRL, or if it fulfils the following criteria:

- unchanged specification;
- systems in different applications;
- at least one year of service history;
- all of the operating experience of the software module shall relate to known demand profiles, ensuring that increased operating experience leads to an increased knowledge of the behaviour of the software module;
- no safety-related failures.

NOTE Failures which might not be safety-critical in one context can be safety-critical in another, and vice versa.

To enable verification that a component or software module 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);

b) identification of users and time of application;

c) operating time;

d) procedure for the selection of the user-applied systems and application cases;

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

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. There are two overlapping areas of defensive techniques. 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. Techniques include 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 is designed to “expect” failures in its own environment or use outside nominal or expected conditions, and behave in a predefined manner. Techniques include the following:

- 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 modules, ensuring they are decoupled as far as possible and all interactions are explicit.

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

c) Keep the number of possible paths through a software module 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 (modularization) 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 part of ISO 25119, the following apply.

- A software module shall have a single well-defined task or function to fulfil.
- Connections between software modules shall be limited and strictly defined; coherence in one software module shall be strong.
- Collections of subprograms shall be built, providing several levels of software modules.
- Software module size shall be restricted to a specified value, typically two to four screen sizes.
- Software modules shall have a single entry and a single exit.
- Software modules shall communicate with other software modules 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.
- All software module interfaces shall be fully documented.
- Any software module interface shall contain only those parameters necessary for its function.

7.4.4.1.9 Library of trusted/verified software modules and components

7.4.4.1.9.1 Aim

A library of trusted/verified software modules and 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.

7.4.4.1.9.2 Description

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

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

Furthermore, the use of such software modules 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 modules.

7.4.4.1.10 Computer-aided design tools

7.4.4.1.10.1 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.

7.4.4.1.10.2 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.

7.4.4.1.11 Use of coding standards

7.4.4.1.11.1 Aim

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

7.4.4.1.11.2 Description

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

- details of modularization, e.g. interface shapes, software module 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, 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 module testing, verification, assessment and maintenance. Therefore, they shall take into account available tools in particular analysers.

7.4.4.1.12 Design and coding standards — No dynamic variables or objects

7.4.4.1.12.1 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.

7.4.4.1.12.2 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.

7.4.4.1.13 Design and coding standards — Limited use of interrupts

7.4.4.1.13.1 Aim

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

7.4.4.1.13.2 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.

7.4.4.1.14 Design and coding standards — Defined use of pointers

7.4.4.1.14.1 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.

7.4.4.1.14.2 Description

In the 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.

7.4.4.1.15 Design and coding standards — Limited use of recursion

7.4.4.1.15.1 Aim

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

7.4.4.1.15.2 Description

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

7.4.4.2 Software module design and coding verification

The software module design and its coding shall be verified. It shall be examined whether the design and coding fulfil the software safety requirements. 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).

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 module design and coding verification report resulting from 7.4.4.2.

7.5 Software module testing

7.5.1 Objectives

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

7.5.2 General

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

7.5.3 Prerequisites

The following are the prerequisites for software module testing:

- software project plan (see 7.1.4.2 to 7.1.4.4);
- software requirements [see 7.2.5 a) and b)];
- software verification plan (see ISO 25119-4:2010, Clause 6);
- software modules according to 7.4.4.1.

7.5.4 Requirements

7.5.4.1 Software module testing methods

The software module testing shall be in accordance with Table 4.

Table 4 — Software module testing

Technique/measure[a]	Subclause	SRL=B	SRL=1	SRL=2	SRL=3
1 Dynamic analysis and testing	7.5.4.1.1				
1.1 Test case execution from boundary value analysis	7.5.4.1.2	o	o	o	+
1.2 Structure-based testing	7.5.4.1.3	o	o	o	+
2 Static analysis					
2.1 Boundary value analysis	7.5.4.1.4	+	+	+	+
2.2 Checklists	7.5.4.1.5	+	+	+	+
2.3 Control flow analysis	7.5.4.1.6	o	o	+	+
2.4 Data flow analysis	7.5.4.1.7	o	o	+	+
2.5 Walk-through/design reviews	7.5.4.1.8	o	o	+	+
3 Functional and black-box testing					
3.1 Equivalence classes and input partition testing	7.5.4.1.9	o	o	+	o
3.2 Boundary value analysis	7.5.4.1.4	o	o	o	+
4 Performance testing	7.5.4.1.10				
4.1 Resource budget testing	7.5.4.1.11	o	+	o	o
4.2 Response timings and memory constraints	7.5.4.1.12	o	o	+	+
4.3 Performance requirements	7.5.4.1.13	o	o	+	+
4.4 Avalanche/stress testing	7.5.4.1.14	o	o	o	+
5 Interface testing	7.5.4.1.15	o	o	o	+

See 7.1.4.7 for instructions on the use of this and the other tables.

[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 alternative or equivalent techniques/measures need be satisfied.

7.5.4.1.1 Dynamic analysis and testing

7.5.4.1.1.1 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.

7.5.4.1.1.2 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 is 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.

7.5.4.1.2 Test case execution from boundary value analysis

7.5.4.1.2.1 Aim

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

7.5.4.1.2.2 Description

The input domain of the program is divided into a number of input classes according to the equivalence relation (see 7.5.4.1.9). The tests shall cover the boundaries and extremes of the classes. The tests check that the boundaries in the input domain of the specification coincide with those in the program. The use of the value zero, in a direct as well as in an indirect translation, is often error-prone and demands special attention 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. Consider also if it is possible to specify a test case which causes the output to exceed the specification boundary values.

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.

7.5.4.1.3 Structure-based testing

7.5.4.1.3.1 Aim

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

7.5.4.1.3.2 Description

Based on analysis of the program, a set of input data is chosen so that a large (and often pre-specified target) percentage of the program code is exercised. Measures of code coverage vary as follows, depending upon the level of rigour required.

- **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.

- **Compound conditions**

  Every condition in a compound conditional branch (i.e. linked by AND/OR) is exercised.

- **Linear code sequence and jump**

  Applicable to any linear sequence of code statements, including conditional statements, terminated by a jump. Many potential sub-paths will not be feasible due to constraints on the input data imposed by the execution of earlier code.

- **Data flow**

  The execution path is selected on the basis of data usage, for example, a path where the same variable is both written and read.

- **Call graph**

  A program is composed of subroutines which can be invoked from other subroutines. The call graph is the tree of subroutine invocations in the program. Tests are designed to cover all invocations in the tree.

- **Basis path**

  One of a minimal set of finite paths from start to finish, such that all arcs are included (overlapping combinations of paths in this basis set can form any path through that part of the program). Tests of all basis paths have been shown to be efficient for locating errors.

7.5.4.1.4 Boundary value analysis

7.5.4.1.4.1 Aim

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

7.5.4.1.4.2 Description

The input domain of the program is divided into a number of input classes according to the equivalence relation (see 7.5.4.1.9). The tests shall cover the boundaries and extremes of the classes. The tests check that the boundaries in the input domain of the specification coincide with those in the program. The use of the value zero, in a direct as well as in an indirect translation, is often error-prone and demands special attention 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. Consider also if it is possible to specify a test case which causes the output to exceed the specification boundary values.

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.

7.5.4.1.5 Checklists

7.5.4.1.5.1 Aim

Checklists shall 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 the laying down of exact requirements.

7.5.4.1.5.2 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 shall 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. 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 may be a need for a particular system to supplement the standard checklist with questions specifically directed at the system being dealt with. In any case, it needs to be clear that the use of checklists depends on the expertise and judgement of the engineer selecting and applying the checklist. As a result, the decisions taken by the engineer, with regard to the checklist(s) selected, and any additional or superfluous questions, shall be fully documented and justified. The objective is to ensure that the application of the checklists can be reviewed, and that the repeatable results will be achieved, unless different criteria are used. The object in completing a checklist is to be as concise as possible. When extensive justification is necessary, this shall be done by reference to additional documentation. Pass, fail and inconclusive, or some similar restricted set of responses, shall be used to document the results for each question. This conciseness greatly simplifies the procedure of reaching an overall conclusion as to the results of the checklist assessment.

7.5.4.1.6 Static analysis — Control flow analysis

7.5.4.1.6.1 Aim

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

7.5.4.1.6.2 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 can 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.

7.5.4.1.7 Static analysis — Data flow analysis

7.5.4.1.7.1 Aim

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

7.5.4.1.7.2 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.

7.5.4.1.8 Static analysis — Walk-through/design review

7.5.4.1.8.1 Aim

A walk-through/design review shall be used to detect faults as soon as economically possible during development.

7.5.4.1.8.2 Description

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

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

7.5.4.1.9 Equivalence classes and input partition testing

7.5.4.1.9.1 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.

7.5.4.1.9.2 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).

7.5.4.1.10 Performance testing

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.

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

7.5.4.1.11 Performance testing — Resource budget testing

7.5.4.1.11.1 Aim

Resource budget testing shall be used according to the complexity of the system:

- for simple systems, an analytic solution may be sufficient;
- for more complex systems, some form of simulation may be more appropriate for obtaining accurate results.

7.5.4.1.11.2 Description

Before detailed modelling, a simpler “resource budget” check can be used which sums the resources requirements of all the processes. If the requirements exceed designed system capacity, the design is unfeasible. Even if the design passes this check, performance modelling can show that excessive delays and response times occur due to resource starvation. To avoid this situation, engineers often design systems to use some fraction (for example 50 %) of the total resources, so that the probability of resource starvation is reduced.

7.5.4.1.12 Performance testing — Response time and memory constraints

7.5.4.1.12.1 Aim

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

7.5.4.1.12.2 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 is 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).

7.5.4.1.13 Performance testing — Performance requirements

7.5.4.1.13.1 Aim

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

7.5.4.1.13.2 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 are 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 the possibility of using a single measurement for more than one performance requirement.

7.5.4.1.14 Performance testing — Avalanche/stress testing

7.5.4.1.14.1 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.

7.5.4.1.14.2 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 is possible, are put to the boundary conditions at the same time.

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

7.5.4.1.15 Interface testing

7.5.4.1.15.1 Aim

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

7.5.4.1.15.2 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 ISO 25119.

7.5.5 Work products

The following work products are applicable to this phase:

a) software module test plan resulting from 7.5.4.1;

b) software module test specification in accordance with 7.5.4.1;

c) software module 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.

7.6.2 General

In this phase, the particular integration levels are tested against the software requirements. The interfaces between the software modules and/or software components are also tested. The steps of the integration and the tests of the software components should 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 7.1.4.2 to 7.1.4.4);
- 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 25119-4:2010, Clause 6);
- tested software modules 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 modules hierarchically into software components until the entire embedded software of the ECU is integrated;

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

NOTE 1 If the hardware–software integration and the test of the embedded software on the target hardware are not already planned and carried out, this could be part of the integration strategy. This procedure can sometimes make the software integration and tests considerably easier.

NOTE 2 In the case of model-based development, the software integration might be replaced with integration at the model level and subsequent code generation for the integrated model.

NOTE 3 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 and 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 equally well all the aspects that have to be taken into account.

7.6.4.4 Software integration and test methods

A hardware and software integration test shall be conducted in accordance with Table 5.

Table 5 — Integration testing (module)

Technique/measure[a]	Subclause	SRL=B	SRL=1	SRL=2	SRL=3
1 Functional and black-box testing	7.6.4.4.1				
1.1 Equivalence classes and input partition testing	7.5.4.1.9	o	o	+	o
1.2 Boundary value analysis	7.5.4.1.4	o	o	o	+
2 Performance testing					
2.1 Resource budget analysis	7.5.4.1.1	o	+	o	o
2.2 Response timings and memory constraints	7.5.4.1.12	o	o	+	+
2.3 Performance requirements	7.5.4.1.13	o	o	+	+
2.4 Avalanche/stress testing	7.5.4.1.14	o	o	o	+

See 7.1.4.7 for instructions on the use of this and the other tables.

[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 alternative or equivalent techniques/measures need 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 avoid failures during implementation and the integration of software and hardware.

7.6.4.4.1.2 Description

During the functional tests, reviews shall be carried out 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, it is recommended that the manufactured components be tested in combination with other partner components of the same batch, in order to reveal common mode faults which would otherwise 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 ISO 25119.

7.6.5 Work products

The following work products are applicable to 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 validation

7.7.1 Objectives

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

The second objective is to provide proof that the requirements of the technical safety concept at the machine level are qualified, complete and completely achieved.

NOTE 1 The software safety validation is a part of the safety validation of the E/E/PES system (see ISO 25119-4:2010, Clause 6). During the planning of the safety validation of the complete E/E/PES system, it has to 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.

NOTE 2 The prerequisite of the software safety validation is that the hardware–software integration has already been carried out.

7.7.2 General

7.7.3 Prerequisites

The following are the prerequisites for software safety validation:

- software project plan (see 7.1.4.2 to 7.1.4.4);
- 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 25119-4:2010, Clause 6);
- integrated software;
- ECU.

7.7.4 Requirements

7.7.4.1 Software safety validation 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 validation

Technique/measure[a]	Subclause	SRL=B	SRL=1	SRL=2	SRL=3
1 Tests of software safety requirements					
1.1 Test interface	7.7.4.1.1	+	+	+	+
1.2a Tests within the ECU network[a]	7.7.4.1.2	o	+	+	o
1.2b Hardware-in-the-loop tests	7.7.4.1.3	o	o	+	+
1.2c Tests in the test machine	7.7.4.1.4	o	o	o	+

See 7.1.4.7 for instructions on the use of this and the other tables.

NOTE Measures in points 1.2a, 1.2b and 1.2c 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 alternative or equivalent techniques/measures need be satisfied.

7.7.4.1.1 Test interface

The software shall be integrated with its host microprocessor in its associated ECU. The test interface is used for determining the inner state of the ECU while testing as well as monitoring internal results.

7.7.4.1.2 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 performs according to specification.

7.7.4.1.3 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. The software shall then be tested in this simulated environment to demonstrate that the software performs according to specification.

7.7.4.1.4 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 performs according to specification.

7.7.4.2 Extent of tests

The software shall be exercised by simulation of

- input signals present during normal operation,
- anticipated 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 ISO 25119.

7.7.5 Work products

The following work products are applicable to this phase:

a) software validation plan resulting from 7.7.4.1 to 7.7.4.4;

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

c) software validation test report according to 7.7.4.3.

7.8 Software-based parameterization

7.8.1 Objective

Software-based parameterization 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 parameterization of safety-related parameters should 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 parameterization:

— software project plan (see 7.1.4.2 to 7.1.4.4);

— 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 25119-4:2010, Clause 6);
- tested software modules according to 7.4.4.1.

7.8.4 Requirements

7.8.4.1 Data integrity

The integrity of data used for parameterization 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, and
  - using error detecting and correcting codes,

c) the errors from the parameter transmission process,

d) the effects of incomplete parameter transmission, and

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

7.8.4.2 Executable code in parameter data

Parameter data shall not contain executable code.

7.8.4.3 Configuration management

Software-based parameterization shall be part of the version configuration management (see ISO 25119-4:2010, Clause 6).

7.8.4.4 Software-based parameterization verification

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

- 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 parameterization are generated and processed in such a way that faults can not lead to a loss of the safety function.

7.8.5 Work products

The following work products are applicable to this phase:

a) validated 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;

d) software validation test report resulting from 7.7.4.3.