INTERNATIONAL STANDARD

ISO 26262-6

First edition
2011-11-15

Road vehicles — Functional safety —

Part 6: **Product development at the software level**

Véhicules routiers — Sécurité fonctionnelle —

Partie 6: Développement du produit au niveau du logiciel

Reference number
ISO 26262-6:2011(E)

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 26262-6 was prepared by Technical Committee ISO/TC 22, *Road vehicles*, Subcommittee SC 3, *Electrical and electronic equipment*.

ISO 26262 consists of the following parts, under the general title *Road vehicles — Functional safety*:

— *Part 1: Vocabulary*

— *Part 2: Management of functional safety*

— *Part 3: Concept phase*

— *Part 4: Product development at the system level*

— *Part 5: Product development at the hardware level*

— *Part 6: Product development at the software level*

— *Part 7: Production and operation*

— *Part 8: Supporting processes*

— *Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses*

— *Part 10: Guideline on ISO 26262*

Introduction

ISO 26262 is the adaptation of IEC 61508 to comply with needs specific to the application sector of electrical and/or electronic (E/E) systems within road vehicles.

This adaptation applies to all activities during the safety lifecycle of safety-related systems comprised of electrical, electronic and software components.

Safety is one of the key issues of future automobile development. New functionalities not only in areas such as driver assistance, propulsion, in vehicle dynamics control and active and passive safety systems increasingly touch the domain of system safety engineering. Development and integration of these functionalities will strengthen the need for safe system development processes and the need to provide evidence that all reasonable system safety objectives are satisfied.

With the trend of increasing technological complexity, software content and mechatronic implementation, there are increasing risks from systematic failures and random hardware failures. ISO 26262 includes guidance to avoid these risks by providing appropriate requirements and processes.

System safety is achieved through a number of safety measures, which are implemented in a variety of technologies (e.g. mechanical, hydraulic, pneumatic, electrical, electronic, programmable electronic) and applied at the various levels of the development process. Although ISO 26262 is concerned with functional safety of E/E systems, it provides a framework within which safety-related systems based on other technologies can be considered. ISO 26262:

a) provides an automotive safety lifecycle (management, development, production, operation, service, decommissioning) and supports tailoring the necessary activities during these lifecycle phases;

b) provides an automotive-specific risk-based approach to determine integrity levels [Automotive Safety Integrity Levels (ASIL)];

c) uses ASILs to specify applicable requirements of ISO 26262 so as to avoid unreasonable residual risk;

d) provides requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety being achieved;

e) provides requirements for relations with suppliers.

Functional safety is influenced by the development process (including such activities as requirements specification, design, implementation, integration, verification, validation and configuration), the production and service processes and by the management processes.

Safety issues are intertwined with common function-oriented and quality-oriented development activities and work products. ISO 26262 addresses the safety-related aspects of development activities and work products.

Figure 1 shows the overall structure of this edition of ISO 26262. ISO 26262 is based upon a V-model as a reference process model for the different phases of product development. Within the figure:

— the shaded “V”s represent the interconnection between ISO 26262-3, ISO 26262-4, ISO 26262-5, ISO 26262-6 and ISO 26262-7;

— the specific clauses are indicated in the following manner: “m-n”, where “m” represents the number of the particular part and “n” indicates the number of the clause within that part.

EXAMPLE “2-6” represents Clause 6 of ISO 26262-2.

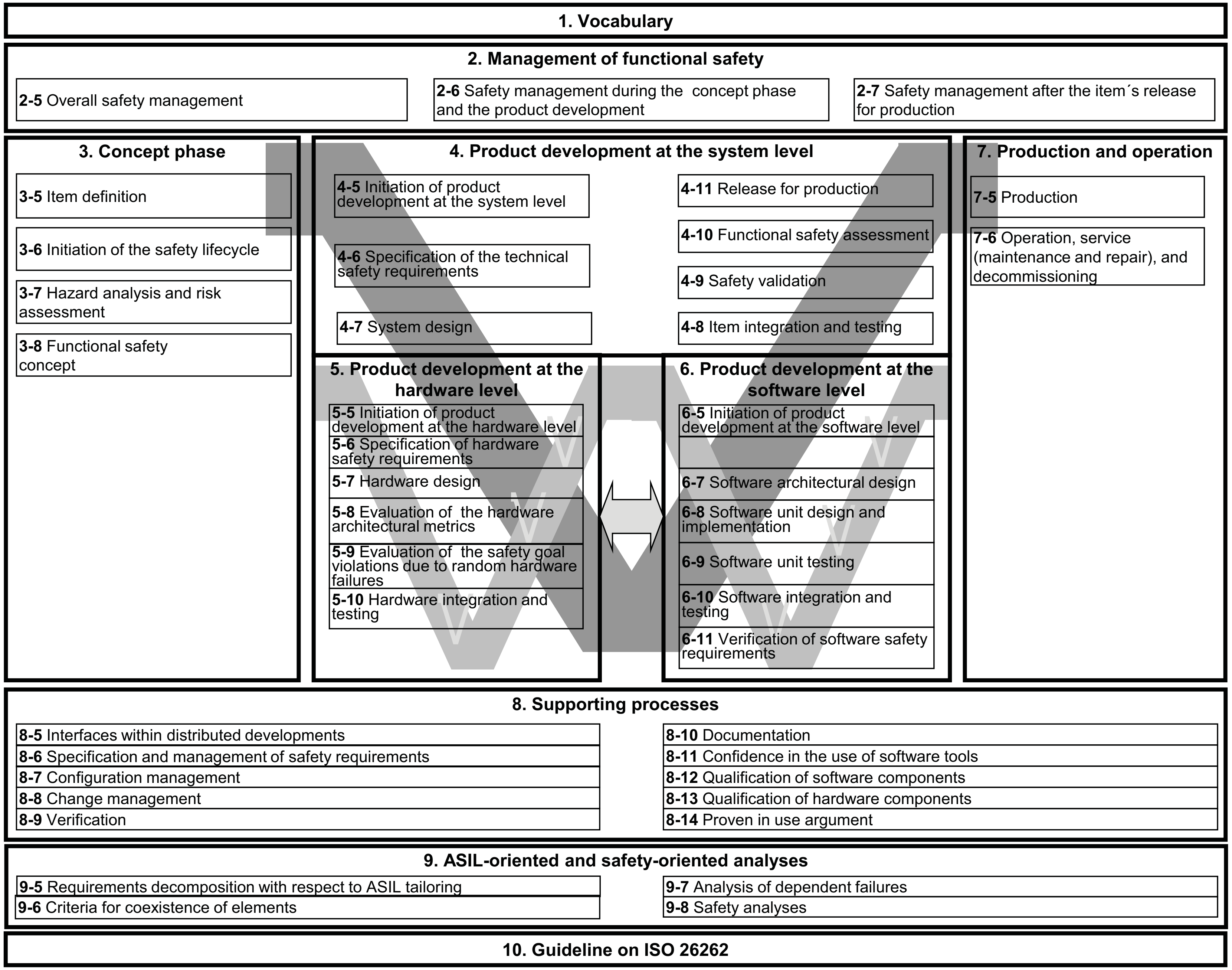

1. Vocabulary
2. Management of functional safety
2-5 Overall safety management
2-6 Safety management during the concept phase and the product development
2-7 Safety management after the item´s release for production
3. Concept phase
3-5 Item definition
3-6 Initiation of the safety lifecycle
3-7 Hazard analysis and risk assessment
3-8 Functional safety concept
4. Product development at the system level
4-5 Initiation of product development at the system level
4-6 Specification of the technical safety requirements
4-7 System design
4-11 Release for production
4-10 Functional safety assessment
4-9 Safety validation
4-8 Item integration and testing
7. Production and operation
7-5 Production
7-6 Operation, service (maintenance and repair), and decommissioning
5. Product development at the hardware level
5-5 Initiation of product development at the hardware level
5-6 Specification of hardware safety requirements
5-7 Hardware design
5-8 Evaluation of the hardware architectural metrics
5-9 Evaluation of the safety goal violations due to random hardware failures
5-10 Hardware integration and testing
6. Product development at the software level
6-5 Initiation of product development at the software level
6-7 Software architectural design
6-8 Software unit design and implementation
6-9 Software unit testing
6-10 Software integration and testing
6-11 Verification of software safety requirements
8. Supporting processes
8-5 Interfaces within distributed developments
8-6 Specification and management of safety requirements
8-7 Configuration management
8-8 Change management
8-9 Verification
8-10 Documentation
8-11 Confidence in the use of software tools
8-12 Qualification of software components
8-13 Qualification of hardware components
8-14 Proven in use argument
9. ASIL-oriented and safety-oriented analyses
9-5 Requirements decomposition with respect to ASIL tailoring
9-6 Criteria for coexistence of elements
9-7 Analysis of dependent failures
9-8 Safety analyses
10. Guideline on ISO 26262

Figure 1 — Overview of ISO 26262

Road vehicles — Functional safety —

Part 6: Product development at the software level

1 Scope

ISO 26262 is intended to be applied to safety-related systems that include one or more electrical and/or electronic (E/E) systems and that are installed in series production passenger cars with a maximum gross vehicle mass up to 3 500 kg. ISO 26262 does not address unique E/E systems in special purpose vehicles such as vehicles designed for drivers with disabilities.

Systems and their components released for production, or systems and their components already under development prior to the publication date of ISO 26262, are exempted from the scope. For further development or alterations based on systems and their components released for production prior to the publication of ISO 26262, only the modifications will be developed in accordance with ISO 26262.

ISO 26262 addresses possible hazards caused by malfunctioning behaviour of E/E safety-related systems, including interaction of these systems. It does not address hazards related to electric shock, fire, smoke, heat, radiation, toxicity, flammability, reactivity, corrosion, release of energy and similar hazards, unless directly caused by malfunctioning behaviour of E/E safety-related systems.

ISO 26262 does not address the nominal performance of E/E systems, even if dedicated functional performance standards exist for these systems (e.g. active and passive safety systems, brake systems, Adaptive Cruise Control).

This part of ISO 26262 specifies the requirements for product development at the software level for automotive applications, including the following:

— requirements for initiation of product development at the software level,

— specification of the software safety requirements,

— software architectural design,

— software unit design and implementation,

— software unit testing,

— software integration and testing, and

— verification of software safety requirements.

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 26262-1:2011, *Road vehicles — Functional safety — Part 1: Vocabulary*

ISO 26262-2:2011, *Road vehicles — Functional safety — Part 2: Management of functional safety*

ISO 26262-4:2011, *Road vehicles — Functional safety — Part 4: Product development at the system level*

ISO 26262-5:2011, *Road vehicles — Functional safety — Part 5: Product development at the hardware level*

ISO 26262-8:2011, *Road vehicles — Functional safety — Part 8: Supporting processes*

ISO 26262-9:2011, *Road vehicles — Functional safety — Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses*

3 Terms, definitions and abbreviated terms

For the purposes of this document, the terms, definitions and abbreviated terms given in ISO 26262-1:2011 apply.

4 Requirements for compliance

4.1 General requirements

When claiming compliance with ISO 26262, each requirement shall be complied with, unless one of the following applies:

a) tailoring of the safety activities in accordance with ISO 26262-2 has been planned and shows that the requirement does not apply, or

b) a rationale is available that the non-compliance is acceptable and the rationale has been assessed in accordance with ISO 26262-2.

Information marked as a “NOTE” or “EXAMPLE” is only for guidance in understanding, or for clarification of the associated requirement, and shall not be interpreted as a requirement itself or as complete or exhaustive.

The results of safety activities are given as work products. “Prerequisites” are information which shall be available as work products of a previous phase. Given that certain requirements of a clause are ASIL-dependent or may be tailored, certain work products may not be needed as prerequisites.

“Further supporting information” is information that can be considered, but which in some cases is not required by ISO 26262 as a work product of a previous phase and which may be made available by external sources that are different from the persons or organizations responsible for the functional safety activities.

4.2 Interpretations of tables

Tables are normative or informative depending on their context. The different methods listed in a table contribute to the level of confidence in achieving compliance with the corresponding requirement. Each method in a table is either

a) a consecutive entry (marked by a sequence number in the leftmost column, e.g. 1, 2, 3), or

b) an alternative entry (marked by a number followed by a letter in the leftmost column, e.g. 2a, 2b, 2c).

For consecutive entries, all methods shall be applied as recommended in accordance with the ASIL. If methods other than those listed are to be applied, a rationale shall be given that these fulfil the corresponding requirement.

For alternative entries, an appropriate combination of methods shall be applied in accordance with the ASIL indicated, independent of whether they are listed in the table or not. If methods are listed with different degrees of recommendation for an ASIL, the methods with the higher recommendation should be preferred. A rationale shall be given that the selected combination of methods complies with the corresponding requirement.

NOTE A rationale based on the methods listed in the table is sufficient. However, this does not imply a bias for or against methods not listed in the table.

For each method, the degree of recommendation to use the corresponding method depends on the ASIL and is categorized as follows:

— "++" indicates that the method is highly recommended for the identified ASIL;

— "+" indicates that the method is recommended for the identified ASIL;

— "o" indicates that the method has no recommendation for or against its usage for the identified ASIL.

4.3 ASIL-dependent requirements and recommendations

The requirements or recommendations of each subclause shall be complied with for ASIL A, B, C and D, if not stated otherwise. These requirements and recommendations refer to the ASIL of the safety goal. If ASIL decomposition has been performed at an earlier stage of development, in accordance with ISO 26262-9:2011, Clause 5, the ASIL resulting from the decomposition shall be complied with.

If an ASIL is given in parentheses in ISO 26262, the corresponding subclause shall be considered as a recommendation rather than a requirement for this ASIL. This has no link with the parenthesis notation related to ASIL decomposition.

5 Initiation of product development at the software level

5.1 Objectives

The objective of this sub-phase is to plan and initiate the functional safety activities for the sub-phases of the software development.

5.2 General

The initiation of the software development is a planning activity, where software development sub-phases and their supporting processes (see ISO 26262-8 and ISO 26262-9) are determined and planned according to the extent and complexity of the item development. The software development sub-phases and supporting processes are initiated by determining the appropriate methods in order to comply with the requirements and their respective ASIL. The methods are supported by guidelines and tools, which are determined and planned for each sub-phase and supporting process.

NOTE Tools used for software development can include tools other than software tools.

EXAMPLE Tools used for testing phases.

The planning of the software development includes the coordination with the product development at the system level (see ISO 26262-4) and the hardware level (see ISO 26262-5).

5.3 Inputs to this clause

5.3.1 Prerequisites

The following information shall be available:

— project plan (refined) in accordance with ISO 26262-4:2011, 5.5.1;

— safety plan (refined) in accordance with ISO 26262-4:2011, 5.5.2;

— technical safety concept in accordance with ISO 26262-4:2011, 7.5.1;

— system design specification in accordance with ISO 26262-4:2011, 7.5.2; and

— item integration and testing plan (refined) in accordance with ISO 26262-4:2011, 8.5.1.

5.3.2 Further supporting information

The following information can be considered:

— qualified software tools available (see ISO 26262-8:2011, Clause 11);

— qualified software components available (see ISO 26262-8:2011, Clause 12);

— design and coding guidelines for modelling and programming languages (from external source);

— guidelines for the application of methods (from external source); and

— guidelines for the application of tools (from external source).

5.4 Requirements and recommendations

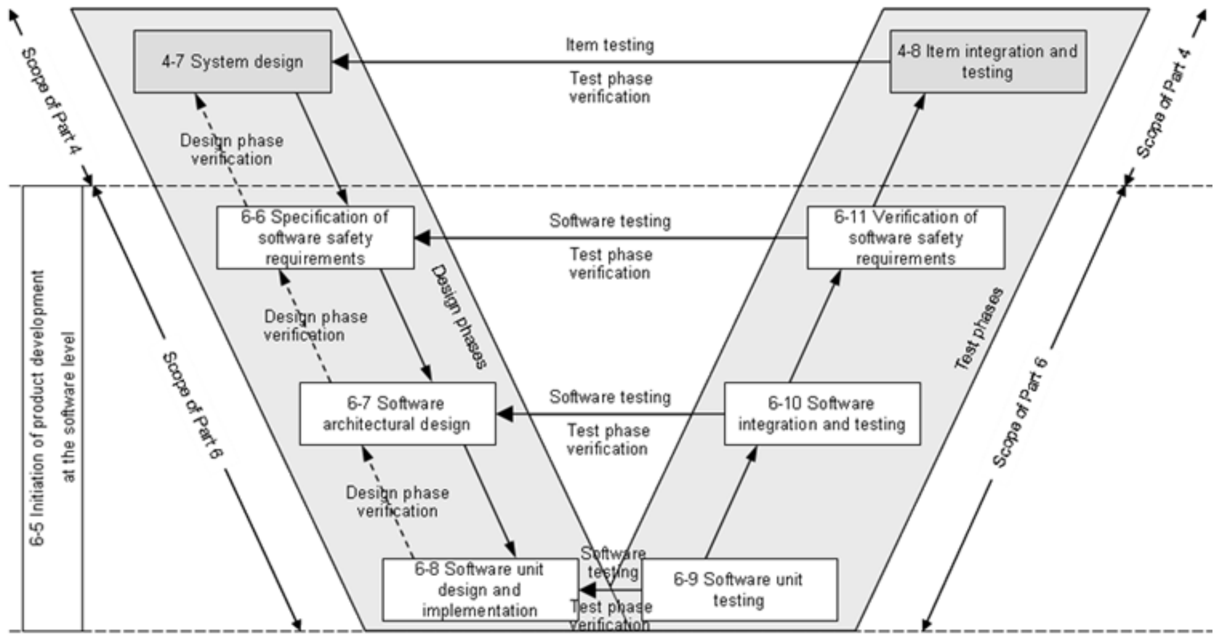
5.4.1 The activities and the determination of appropriate methods for the product development at the software level shall be planned.

5.4.2 The tailoring of the lifecycle for product development at the software level shall be performed in accordance with ISO 26262-2:2011, 6.4.5, and based on the reference phase model given in Figure 2.

5.4.3 If developing configurable software, Annex C shall be applied.

5.4.4 The software development process for the software of an item, including lifecycle phases, methods, languages and tools, shall be consistent across all the sub-phases of the software lifecycle and be compatible with the system and hardware development phases, such that the required data can be transformed correctly.

NOTE The sequencing of phases, tasks and activities, including iteration steps, for the software of an item is to ensure the consistency of the corresponding work products with the product development at the hardware level (see ISO 26262-5) and the product development at the system level (see ISO 26262-4).


4-7 System design
Item testing
Test phase verification
4-8 Item integration and testing
Scope of Part 4
Scope of Part 4
Design phase verification
6-6 Specification of software safety requirements
Software testing
Test phase verification
6-11 Verification of software safety requirements
Design phases
Test phases
6-5 Initiation of product development at the software level
Scope of Part 6
Scope of Part 6
Design phase verification
6-7 Software architectural design
Software testing
Test phase verification
6-10 Software integration and testing
Design phase verification
6-8 Software unit design and implementation
Software testing
Test phase verification
6-9 Software unit testing

NOTE Within the figure, the specific clauses of each part of ISO 26262 are indicated in the following manner: "m-n", where "m" represents the number of the part and "n" indicates the number of the clause, e.g. "4.7" represents Clause 7 of ISO 26262-4.

Figure 2 — Reference phase model for the software development

5.4.5 For each sub-phase of software development, the selection of the following, including guidelines for their application, shall be carried out:

a) methods; and

b) corresponding tools.

5.4.6 The criteria that shall be considered when selecting a suitable modelling or programming language are:

a) an unambiguous definition;

EXAMPLE Syntax and semantics of the language.

b) the support for embedded real time software and runtime error handling; and

c) the support for modularity, abstraction and structured constructs.

Criteria that are not sufficiently addressed by the language itself shall be covered by the corresponding guidelines, or by the development environment.

NOTE 1 The selected programming language (such as ADA, C, C++, Java, Assembler or a graphical modelling language) supports the topics given in 5.4.7. Programming or modelling guidelines can be used to comply with these topics.

NOTE 2 Assembly languages can be used for those parts of the software where the use of high-level programming languages is not appropriate, such as low-level software with interfaces to the hardware, interrupt handlers, or time-critical algorithms.

5.4.7 To support the correctness of the design and implementation, the design and coding guidelines for the modelling, or programming languages, shall address the topics listed in Table 1.

NOTE 1 Coding guidelines are usually different for different programming languages.

NOTE 2 Coding guidelines can be different for model-based development.

NOTE 3 Existing coding guidelines can be modified for a specific item development.

EXAMPLE MISRA C[3] and MISRA AC AGC[4] are coding guidelines for the programming language C.

Table 1 — Topics to be covered by modelling and coding guidelines

Topics		ASIL			
		A	B	C	D
1a	Enforcement of low complexity[a]	++	++	++	++
1b	Use of language subsets[b]	++	++	++	++
1c	Enforcement of strong typing[c]	++	++	++	++
1d	Use of defensive implementation techniques	o	+	++	++
1e	Use of established design principles	+	+	+	++
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+	++	++	++
1h	Use of naming conventions	++	++	++	++

[a] An appropriate compromise of this topic with other methods in this part of ISO 26262 may be required.

[b] The objectives of method 1b are

— Exclusion of ambiguously defined language constructs which may be interpreted differently by different modellers, programmers, code generators or compilers.
— Exclusion of language constructs which from experience easily lead to mistakes, for example assignments in conditions or identical naming of local and global variables.
— Exclusion of language constructs which could result in unhandled run-time errors.

[c] The objective of method 1c is to impose principles of strong typing where these are not inherent in the language.

5.5 Work products

5.5.1 Safety plan (refined) resulting from requirements 5.4.1 to 5.4.7.

5.5.2 Software verification plan resulting from requirements 5.4.1 to 5.4.5 and 5.4.7.

5.5.3 Design and coding guidelines for modelling and programming languages resulting from requirements 5.4.6 and 5.4.7.

5.5.4 Tool application guidelines resulting from requirements 5.4.5 and 5.4.6.

6 Specification of software safety requirements

6.1 Objectives

The first objective of this sub-phase is to specify the software safety requirements. They are derived from the technical safety concept and the system design specification.

The second objective is to detail the hardware-software interface requirements initiated in ISO 26262-4:2011, Clause 7.

The third objective is to verify that the software safety requirements and the hardware-software interface requirements are consistent with the technical safety concept and the system design specification.

6.2 General

The technical safety requirements are refined and allocated to hardware and software during the system design phase given in ISO 26262-4:2011, Clause 7. The specification of the software safety requirements considers constraints of the hardware and the impact of these constraints on the software. This sub-phase includes the specification of software safety requirements to support the subsequent design phases.

6.3 Inputs to this clause

6.3.1 Prerequisites

The following information shall be available:

— technical safety concept in accordance with ISO 26262-4:2011, 7.5.1;

— system design specification in accordance with ISO 26262-4:2011, 7.5.2;

— hardware-software interface specification in accordance with ISO 26262-4:2011, 7.5.3;

— safety plan (refined) in accordance with 5.5.1;

— software verification plan in accordance with 5.5.2.

6.3.2 Further supporting information

The following information can be considered:

— hardware design specification (see ISO 26262-5:2011, 7.5.1);

— guidelines for the application of methods (from external source).

6.4 Requirements and recommendations

6.4.1 The software safety requirements shall address each software-based function whose failure could lead to a violation of a technical safety requirement allocated to software.

EXAMPLE Functions whose failure could lead to a violation of a safety requirement can be:

— functions that enable the system to achieve or maintain a safe state;

— functions related to the detection, indication and handling of faults of safety-related hardware elements;

— functions related to the detection, notification and mitigation of faults in the software itself;

NOTE 1 These include both the self-monitoring of the software in the operating system and application-specific self-monitoring of the software to detect, indicate and handle systematic faults in the application software.

— functions related to on-board and off-board tests;

NOTE 2 On-board tests can be carried out by the system itself or through other systems within the vehicle network during operation and during the pre-run and post-run phase of the vehicle.

NOTE 3 Off-board tests refer to the testing of the safety-related functions or properties during production or in service.

— functions that allow modifications of the software during production and service; and

— functions related to performance or time-critical operations.

6.4.2 The specification of the software safety requirements shall be derived from the technical safety concept and the system design in accordance with ISO 26262-4:2011, 7.4.1 and 7.4.5, and shall consider:

a) the specification and management of safety requirements in accordance with ISO 26262-8:2011, Clause 6;

b) the specified system and hardware configurations;

EXAMPLE 1 Configuration parameters can include gain control, band pass frequency and clock prescaler.

c) the hardware-software interface specification;

d) the relevant requirements of the hardware design specification;

e) the timing constraints;

EXAMPLE 2 Execution or reaction time derived from the required response time at the system level.

f) the external interfaces; and

EXAMPLE 3 Communication and user interfaces.

g) each operating mode of the vehicle, the system, or the hardware, having an impact on the software.

EXAMPLE 4 Operating modes of hardware devices can include default, initialization, test, and advanced modes.

6.4.3 If ASIL decomposition is applied to the software safety requirements, ISO 26262-9:2011, Clause 5, shall be complied with.

6.4.4 The hardware-software interface specification initiated in ISO 26262-4:2011, Clause 7, shall be detailed down to a level allowing the correct control and usage of hardware, and shall describe each safety-related dependency between hardware and software.

6.4.5 If other functions in addition to those functions for which safety requirements are specified in 6.4.1 are carried out by the embedded software, these functions shall be specified, or else a reference made to their specification.

6.4.6 The verification of the software safety requirements and of the refined specification of the hardware software interface shall be planned in accordance with ISO 26262-8:2011, Clause 9.

6.4.7 The refined hardware-software interface specification shall be verified jointly by the persons responsible for the system, hardware and software development.

6.4.8 The software safety requirements and the refined hardware-software interface requirements shall be verified in accordance with ISO 26262-8:2011, Clauses 6 and 9, to show their:

a) compliance and consistency with the technical safety requirements;

b) compliance with the system design; and

c) consistency with the hardware-software interface.

6.5 Work products

6.5.1 Software safety requirements specification resulting from requirements 6.4.1 to 6.4.3 and 6.4.5.

6.5.2 Hardware-software interface specification (refined) resulting from requirement 6.4.4.

NOTE This work product refers to the same work product as given in ISO 26262-5:2011 6.5.2

6.5.3 Software verification plan (refined) resulting from requirement 6.4.6.

6.5.4 Software verification report resulting from requirements 6.4.7 and 6.4.8.

7 Software architectural design

7.1 Objectives

The first objective of this sub-phase is to develop a software architectural design that realizes the software safety requirements.

The second objective of this sub-phase is to verify the software architectural design.

7.2 General

The software architectural design represents all software components and their interactions in a hierarchical structure. Static aspects, such as interfaces and data paths between all software components, as well as dynamic aspects, such as process sequences and timing behaviour are described.

NOTE The software architectural design is not necessarily limited to one microcontroller or ECU, and is related to the technical safety concept and system design. The software architecture for each microcontroller is also addressed by this chapter.

In order to develop a software architectural design both software safety requirements as well as all non-safety-related requirements are implemented. Hence in this sub-phase safety-related and non-safety-related requirements are handled within one development process.

The software architectural design provides the means to implement the software safety requirements and to manage the complexity of the software development.

7.3 Inputs to this clause

7.3.1 Prerequisites

The following information shall be available:

— safety plan (refined) in accordance with 5.5.1;

— design and coding guidelines for modelling and programming languages in accordance with 5.5.3;

— hardware-software interface specification in accordance with ISO 26262-4:2011, 7.5.3;

— software safety requirements specification in accordance with 6.5.1;

— software verification plan (refined) in accordance with 6.5.3; and

— software verification report in accordance with 6.5.4.

7.3.2 Further supporting information

The following information can be considered:

- technical safety concept (see ISO 26262-4:2011, 7.5.1);
- system design specification (see ISO 26262-4:2011, 7.5.2);
- qualified software components available (see ISO 26262-8:2011, Clause 12);
- tool application guidelines in accordance with 5.5.4; and
- guidelines for the application of methods (from external source).

7.4 Requirements and recommendations

7.4.1 To ensure that the software architectural design captures the information necessary to allow the subsequent development activities to be performed correctly and effectively, the software architectural design shall be described with appropriate levels of abstraction by using the notations for software architectural design listed in Table 2.

Table 2 — Notations for software architectural design

Methods		ASIL			
		A	B	C	D
1a	Informal notations	++	++	+	+
1b	Semi-formal notations	+	++	++	++
1c	Formal notations	+	+	+	+

7.4.2 During the development of the software architectural design the following shall be considered:

a) the verifiability of the software architectural design;

NOTE This implies bi-directional traceability between the software architectural design and the software safety requirements.

b) the suitability for configurable software;

c) the feasibility for the design and implementation of the software units;

d) the testability of the software architecture during software integration testing; and

e) the maintainability of the software architectural design.

7.4.3 In order to avoid failures resulting from high complexity, the software architectural design shall exhibit the following properties by use of the principles listed in Table 3:

a) modularity;

b) encapsulation; and

c) simplicity.

Table 3 — Principles for software architectural design

Methods		ASIL			
		A	B	C	D
1a	Hierarchical structure of software components	++	++	++	++
1b	Restricted size of software components[a]	++	++	++	++
1c	Restricted size of interfaces[a]	+	+	+	+
1d	High cohesion within each software component[b]	+	++	++	++
1e	Restricted coupling between software components[a, b, c]	+	++	++	++
1f	Appropriate scheduling properties	++	++	++	++
1g	Restricted use of interrupts[a, d]	+	+	+	++

[a] In methods 1b, 1c, 1e and 1g "restricted" means to minimize in balance with other design considerations.

[b] Methods 1d and 1e can, for example, be achieved by separation of concerns which refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concept, goal, task, or purpose.

[c] Method 1e addresses the limitation of the external coupling of software components.

[d] Any interrupts used have to be priority-based.

NOTE An appropriate compromise between the methods listed in Table 3 can be necessary since the methods are not mutually exclusive.

7.4.4 The software architectural design shall be developed down to the level where all software units are identified.

7.4.5 The software architectural design shall describe:

a) the static design aspects of the software components; and

NOTE 1 Static design aspects address:

- the software structure including its hierarchical levels;
- the logical sequence of data processing;
- the data types and their characteristics;
- the external interfaces of the software components;
- the external interfaces of the software; and
- the constraints including the scope of the architecture and external dependencies.

NOTE 2 In the case of model-based development, modelling the structure is an inherent part of the overall modelling activities.

b) the dynamic design aspects of the software components.

NOTE 1 Dynamic design aspects address:

- the functionality and behaviour;
- the control flow and concurrency of processes;
- the data flow between the software components;
- the data flow at external interfaces; and

— the temporal constraints.

NOTE 2 To determine the dynamic behaviour (e.g. of tasks, time slices and interrupts) the different operating states (e.g. power-up, shut-down, normal operation, calibration and diagnosis) are considered.

NOTE 3 To describe the dynamic behaviour (e.g. of tasks, time slices and interrupts) the communication relationships and their allocation to the system hardware (e.g. CPU and communication channels) are specified.

7.4.6 Every safety-related software component shall be categorized as one of the following:

a) newly developed;

b) reused with modifications; or

c) reused without modifications.

7.4.7 Safety-related software components that are newly developed or reused with modifications shall be developed in accordance with ISO 26262.

NOTE In these cases ISO 26262-8:2011, Clause 12, does not apply.

7.4.8 Safety-related software components that are reused without modifications shall be qualified in accordance with ISO 26262-8:2011, Clause 12.

NOTE The use of qualified software components does not affect the applicability of Clauses 10 and 11. However, some activities described in Clauses 8 and 9 can be omitted.

7.4.9 The software safety requirements shall be allocated to the software components. As a result, each software component shall be developed in compliance with the highest ASIL of any of the requirements allocated to it.

NOTE Following this allocation, further refinement of the software safety requirements can be necessary.

7.4.10 If the embedded software has to implement software components of different ASILs, or safety-related and non-safety-related software components, then all of the embedded software shall be treated in accordance with the highest ASIL, unless the software components meet the criteria for coexistence in accordance with ISO 26262-9:2011, Clause 6.

7.4.11 If software partitioning (see Annex D) is used to implement freedom from interference between software components it shall be ensured that:

a) the shared resources are used in such a way that freedom from interference of software partitions is ensured;

NOTE 1 Tasks within a software partition are not free from interference among each other.

NOTE 2 One software partition cannot change the code or data of another software partition nor command non-shared resources of other software partitions.

NOTE 3 The service received from shared resources by one software partition cannot be affected by another software partition. This includes the performance of the resources concerned, as well as the rate, latency, jitter and duration of scheduled access to the resource.

b) the software partitioning is supported by dedicated hardware features or equivalent means (this requirement applies to ASIL D, in accordance with 4.3);

c) the part of the software that implements the software partitioning is developed in compliance with the same or an ASIL higher than the highest ASIL assigned to the requirements of the software partitions; and

NOTE In general the operating system provides or supports software partitioning.

d) the verification of the software partitioning during software integration and testing (in accordance with Clause 10) is performed.

7.4.12 An analysis of dependent failures in accordance with ISO 26262-9:2011, Clause 7, shall be carried out if the implementation of software safety requirements relies on freedom from interference or sufficient independence between software components.

7.4.13 Safety analysis shall be carried out at the software architectural level in accordance with ISO 26262-9:2011, Clause 8, in order to:

— identify or confirm the safety-related parts of the software; and

— support the specification and verify the efficiency of the safety mechanisms.

NOTE Safety mechanisms can be specified to cover both issues associated with random hardware failures as well as software faults.

7.4.14 To specify the necessary software safety mechanisms at the software architectural level, based on the results of the safety analysis in accordance with 7.4.13, mechanisms for error detection as listed in Table 4 shall be applied.

NOTE When not directly required by technical safety requirements allocated to software, the use of software safety mechanisms is reviewed at the system level to analyse the potential impact on the system behaviour.

Table 4 — Mechanisms for error detection at the software architectural level

Methods		ASIL			
		A	B	C	D
1a	Range checks of input and output data	++	++	++	++
1b	Plausibility check[a]	+	+	+	++
1c	Detection of data errors[b]	+	+	+	+
1d	External monitoring facility[c]	o	+	+	++
1e	Control flow monitoring	o	+	++	++
1f	Diverse software design	o	o	+	++

[a] Plausibility checks can include using a reference model of the desired behaviour, assertion checks, or comparing signals from different sources.

[b] Types of methods that may be used to detect data errors include error detecting codes and multiple data storage.

[c] An external monitoring facility can be, for example, an ASIC or another software element performing a watchdog function.

7.4.15 This subclause applies to ASIL (A), (B), C and D, in accordance with 4.3: to specify the necessary software safety mechanisms at the software architectural level, based on the results of the safety analysis in accordance with 7.4.13, mechanisms for error handling as listed in Table 5 shall be applied.

NOTE 1 When not directly required by technical safety requirements allocated to software, the use of software safety mechanisms is reviewed at the system level to analyse the potential impact on the system behaviour.

NOTE 2 The analysis at software architectural level of possible hazards due to hardware is described in ISO 26262-5.

Table 5 — Mechanisms for error handling at the software architectural level

Methods		ASIL			
		A	B	C	D
1a	Static recovery mechanism[a]	+	+	+	+
1b	Graceful degradation[b]	+	+	++	++
1c	Independent parallel redundancy[c]	o	o	+	++
1d	Correcting codes for data	+	+	+	+

[a] Static recovery mechanisms can include the use of recovery blocks, backward recovery, forward recovery and recovery through repetition.

[b] Graceful degradation at the software level refers to prioritizing functions to minimize the adverse effects of potential failures on functional safety.

[c] Independent parallel redundancy can be realized as dissimilar software in each parallel path.

7.4.16 If new hazards introduced by the software architectural design are not already covered by an existing safety goal, they shall be introduced and evaluated in the hazard analysis and risk assessment in accordance with the change management process in ISO 26262-8:2011, Clause 8.

NOTE Newly identified hazards, not already reflected in a safety goal, are usually non-functional hazards. If those non-functional hazards are outside the scope of this standard then it is recommended that they be annotated in the hazard analysis and risk assessment with the following statement “No ASIL is assigned to this hazard as it is not within the scope of ISO 26262.” However, an ASIL is allowed for reference purposes.

7.4.17 An upper estimation of required resources for the embedded software shall be made, including:

a) the execution time;

b) the storage space; and

EXAMPLE RAM for stacks and heaps, ROM for program and non-volatile data.

c) the communication resources.

7.4.18 The software architectural design shall be verified in accordance with ISO 26262-8:2011, Clause 9, and by using the software architectural design verification methods listed in Table 6 to demonstrate the following properties:

a) compliance with the software safety requirements;

b) compatibility with the target hardware; and

NOTE This includes the resources as specified in 7.4.17.

c) adherence to design guidelines.

Table 6 — Methods for the verification of the software architectural design

Methods		ASIL			
		A	B	C	D
1a	Walk-through of the design[a]	++	+	o	o
1b	Inspection of the design[a]	+	++	++	++
1c	Simulation of dynamic parts of the design[b]	+	+	+	++
1d	Prototype generation	o	o	+	++
1e	Formal verification	o	o	+	+
1f	Control flow analysis[c]	+	+	++	++
1g	Data flow analysis[c]	+	+	++	++

[a] In the case of model-based development these methods can be applied to the model.

[b] Method 1c requires the usage of executable models for the dynamic parts of the software architecture.

[c] Control and data flow analysis may be limited to safety-related components and their interfaces.

7.5 Work products

7.5.1 Software architectural design specification resulting from requirements 7.4.1 to 7.4.6, 7.4.9, 7.4.10, 7.4.14, 7.4.15 and 7.4.17.

7.5.2 Safety plan (refined) resulting from requirement 7.4.7.

7.5.3 Software safety requirements specification (refined) resulting from requirement 7.4.9.

7.5.4 Safety analysis report resulting from requirement 7.4.13.

7.5.5 Dependent failures analysis report resulting from requirement 7.4.12.

7.5.6 Software verification report (refined) resulting from requirement 7.4.18.

8 Software unit design and implementation

8.1 Objectives

The first objective of this sub-phase is to specify the software units in accordance with the software architectural design and the associated software safety requirements.

The second objective of this sub-phase is to implement the software units as specified.

The third objective of this sub-phase is the static verification of the design of the software units and their implementation.

8.2 General

Based on the software architectural design, the detailed design of the software units is developed. The detailed design will be implemented as a model or directly as source code, in accordance with the modelling or coding guidelines respectively. The detailed design and the implementation are statically verified before proceeding to the software unit testing phase. The implementation-related properties are achievable at the source code level if manual code development is used. If model-based development with automatic code generation is used, these properties apply to the model and need not apply to the source code.

In order to develop a single software unit design both software safety requirements as well as all non-safety-related requirements are implemented. Hence in this sub-phase safety-related and non-safety-related requirements are handled within one development process.

The implementation of the software units includes the generation of source code and the translation into object code.

8.3 Inputs to this clause

8.3.1 Prerequisites

The following information shall be available:

— design and coding guidelines for modelling and programming languages in accordance with 5.5.3;

— software verification plan (refined) in accordance with 6.5.3;

— software architectural design specification in accordance with 7.5.1;

— safety plan (refined) in accordance with 7.5.2;

— software safety requirements specification (refined) in accordance with 7.5.3; and

— software verification report (refined) in accordance with 7.5.6.

8.3.2 Further supporting information

The following information can be considered:

— technical safety concept (see ISO 26262-4:2011, 7.5.1);

— system design specification (see ISO 26262-4:2011, 7.5.2);

— tool application guidelines in accordance with 5.5.4;

— hardware-software interface specification (refined) (see 6.5.2);

— safety analysis report in accordance with 7.5.4; and

— guidelines for the application of methods (from external source).

8.4 Requirements and recommendations

8.4.1 The requirements of this subclause shall be complied with if the software unit is safety-related.

NOTE "Safety-related" means that the unit implements safety requirements, or that the criteria for coexistence (see ISO 26262-9:2011, Clause 6) of the unit with other units are not satisfied.

8.4.2 To ensure that the software unit design captures the information necessary to allow the subsequent development activities to be performed correctly and effectively, the software unit design shall be described using the notations listed in Table 7.

Table 7 — Notations for software unit design

Methods		ASIL			
		A	B	C	D
1a	Natural language	++	++	++	++
1b	Informal notations	++	++	+	+
1c	Semi-formal notations	+	++	++	++
1d	Formal notations	+	+	+	+

NOTE In the case of model-based development with automatic code generation, the methods for representing the software unit design are applied to the model which serves as the basis for the code generation.

8.4.3 The specification of the software units shall describe the functional behaviour and the internal design to the level of detail necessary for their implementation.

EXAMPLE Internal design can include constraints on the use of registers and storage of data.

8.4.4 Design principles for software unit design and implementation at the source code level as listed in Table 8 shall be applied to achieve the following properties:

a) correct order of execution of subprograms and functions within the software units, based on the software architectural design;

b) consistency of the interfaces between the software units;

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

d) simplicity;

e) readability and comprehensibility;

f) robustness;

EXAMPLE Methods to prevent implausible values, execution errors, division by zero, and errors in the data flow and control flow.

g) suitability for software modification; and

h) testability.

Table 8 — Design principles for software unit design and implementation

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

[a] Methods 1a, 1b, 1d, 1e, 1f, 1g and 1i may not be applicable for graphical modelling notations used in model-based development.

[b] Methods 1g and 1i are not applicable in assembler programming.

[c] Methods 1h and 1i reduce the potential for modelling data flow and control flow through jumps or global variables.

NOTE For the C language, MISRA C[3] covers many of the methods listed in Table 8.

8.4.5 The software unit design and implementation shall be verified in accordance with ISO 26262-8:2011 Clause 9, and by applying the verification methods listed in Table 9, to demonstrate:

a) the compliance with the hardware-software interface specification (in accordance with ISO 26262-5:2011, 6.4.10);

b) the fulfilment of the software safety requirements as allocated to the software units (in accordance with 7.4.9) through traceability;

c) the compliance of the source code with its design specification;

NOTE In the case of model-based development, requirement c) still applies.

d) the compliance of the source code with the coding guidelines (see 5.5.3); and

e) the compatibility of the software unit implementations with the target hardware.

Table 9 — Methods for the verification of software unit design and implementation

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

[a] In the case of model-based software development the software unit specification design and implementation can be verified at the model level.

[b] Methods 1e and 1f can be applied at the source code level. These methods are applicable both to manual code development and to model-based development.

[c] Methods 1e and 1f can be part of methods 1d, 1g or 1h.

[d] Method 1h is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.

NOTE Table 9 lists only static verification techniques. Dynamic verification techniques (e.g. testing techniques) are covered in Tables 10, 11 and 12.

8.5 Work products

8.5.1 Software unit design specification resulting from requirements 8.4.2 to 8.4.4.

NOTE In the case of model-based development, the implementation model and supporting descriptive documentation, using techniques listed in Table 8, specifies the software units.

8.5.2 Software unit implementation resulting from requirement 8.4.4.

8.5.3 Software verification report (refined) resulting from requirement 8.4.5.

9 Software unit testing

9.1 Objectives

The objective of this sub-phase is to demonstrate that the software units fulfil the software unit design specifications and do not contain undesired functionality.

9.2 General

A procedure for testing the software unit against the software unit design specifications is established, and the tests are carried out in accordance with this procedure.

9.3 Inputs to this clause

9.3.1 Prerequisites

The following information shall be available:

— hardware-software interface specification (refined) in accordance with 6.5.2;

— software verification plan (refined) in accordance with 6.5.3;

— safety plan (refined) in accordance with 7.5.2;

— software unit design specification in accordance with 8.5.1;

— software unit implementation in accordance with 8.5.2; and

— software verification report (refined) in accordance with 8.5.3.

9.3.2 Further supporting information

The following information can be considered:

— tool application guidelines in accordance with 5.5.4; and

— guidelines for the application of methods (from external source).

9.4 Requirements and recommendations

9.4.1 The requirements of this subclause shall be complied with if the software unit is safety-related.

NOTE "Safety-related" means that the unit implements safety requirements, or that the criteria for coexistence of the unit with other units are not satisfied.

9.4.2 Software unit testing shall be planned, specified and executed in accordance with ISO 26262-8:2011, Clause 9.

NOTE 1 Based on the definitions in ISO 26262-8:2011, Clause 9, the test objects in the software unit testing are the software units.

NOTE 2 For model-based software development, the corresponding parts of the implementation model also represent objects for the test planning. Depending on the selected software development process the test objects can be the code derived from this model or the model itself.

9.4.3 The software unit testing methods listed in Table 10 shall be applied to demonstrate that the software units achieve:

a) compliance with the software unit design specification (in accordance with Clause 8);

b) compliance with the specification of the hardware-software interface (in accordance with ISO 26262-5:2011, 6.4.10);

c) the specified functionality;

d) confidence in the absence of unintended functionality;

e) robustness; and

EXAMPLE The absence of inaccessible software, the effectiveness of error detection and error handling mechanisms.

f) sufficient resources to support their functionality.

Table 10 — Methods for software unit testing

Methods		ASIL			
		A	B	C	D
1a	Requirements-based test[a]	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[b]	+	+	+	++
1d	Resource usage test[c]	+	+	+	++
1e	Back-to-back comparison test between model and code, if applicable[d]	+	+	++	++

[a] The software requirements at the unit level are the basis for this requirements-based test.

[b] This includes injection of arbitrary faults (e.g. by corrupting values of variables, by introducing code mutations, or by corrupting values of CPU registers).

[c] Some aspects of the resource usage test can only be evaluated properly when the software unit tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests.

[d] This method requires a model that can simulate the functionality of the software units. Here, the model and code are stimulated in the same way and results compared with each other.

9.4.4 To enable the specification of appropriate test cases for the software unit testing in accordance with 9.4.3, test cases shall be derived using the methods listed in Table 11.

Table 11 — Methods for deriving test cases for software unit testing

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

[a] Equivalence classes can be identified based on the division of inputs and outputs, such that a representative test value can be selected for each class.

[b] This method applies to interfaces, values approaching and crossing the boundaries and out of range values.

[c] Error guessing tests can be based on data collected through a “lessons learned” process and expert judgment.

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

EXAMPLE 1 Analysis of structural coverage can reveal shortcomings in requirement-based test cases, inadequacies in requirements, dead code, deactivated code or unintended functionality.

EXAMPLE 2 A rationale can be given for the level of coverage achieved based on accepted dead code (e.g. code for debugging) or code segments depending on different software configurations; or code not covered can be verified using complementary methods (e.g. inspections).

Table 12 — Structural coverage metrics at the software unit level

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

NOTE 1 The structural coverage can be determined by the use of appropriate software tools.

NOTE 2 In the case of model-based development, the analysis of structural coverage can be performed at the model level using analogous structural coverage metrics for models.

NOTE 3 If instrumented code is used to determine the degree of coverage, it can be necessary to show that the instrumentation has no effect on the test results. This can be done by repeating the tests with non-instrumented code.

9.4.6 The test environment for software unit testing shall correspond as closely as possible to the target environment. If the software unit testing is not carried out in the target environment, the differences in the source and object code, and the differences between the test environment and the target environment, shall be analysed in order to specify additional tests in the target environment during the subsequent test phases.

NOTE 1 Differences between the test environment and the target environment can arise in the source code or object code, for example, due to different bit widths of data words and address words of the processors.

NOTE 2 Depending on the scope of the tests, the appropriate test environment for the execution of the software unit is used (e.g. the target processor, a processor emulator or a development system).

NOTE 3 Software unit testing can be executed in different environments, for example:

— model-in-the-loop tests;

— software-in-the-loop tests;

— processor-in-the-loop tests; and

— hardware-in-the-loop tests.

NOTE 4 For model-based development, software unit testing can be carried out at the model level followed by back-to-back comparison tests between the model and the object code. The back-to-back comparison tests are used to ensure that the behaviour of the models with regard to the test objectives is equivalent to the automatically-generated code.

9.5 Work products

9.5.1 **Software verification plan (refined)** resulting from requirements 9.4.2 to 9.4.6.

9.5.2 Software verification specification resulting from requirements 9.4.2 and 9.4.4 to 9.4.6.

9.5.3 Software verification report (refined) resulting from requirement 9.4.2.

10 Software integration and testing

10.1 Objectives

The first objective of this sub-phase is to integrate the software elements.

The second objective of this sub-phase is to demonstrate that the software architectural design is realized by the embedded software.

10.2 General

In this sub-phase, the particular integration levels and the interfaces between the software elements are tested against the software architectural design. The steps of the integration and testing of the software elements correspond directly to the hierarchical architecture of the software.

The embedded software can consist of safety-related and non-safety-related software elements.

10.3 Inputs to this clause

10.3.1 Prerequisites

The following information shall be available:

- hardware-software interface specification (refined) in accordance with 6.5.2;
- software architectural design specification in accordance with 7.5.1;
- safety plan (refined) in accordance with 7.5.2;
- software unit implementation in accordance with 8.5.2;
- software verification plan (refined) in accordance with 9.5.1;
- software verification specification in accordance with 9.5.2; and
- software verification report (refined) in accordance with 9.5.3.

10.3.2 Further supporting information

The following information can be considered:

- qualified software components available (see ISO 26262-8:2011, Clause 12);
- software tool qualification report in accordance with ISO 26262-8:2011, 11.5.2;
- tool application guidelines in accordance with 5.5.4; and
- guidelines for the application of methods (from external source).

10.4 Requirements and recommendations

10.4.1 The planning of the software integration shall describe the steps for integrating the individual software units hierarchically into software components until the embedded software is fully integrated, and shall consider:

a) the functional dependencies that are relevant for software integration; and

b) the dependencies between the software integration and the hardware-software integration.

NOTE For model-based development, the software integration can be replaced with integration at the model level and subsequent automatic code generation from the integrated model.

10.4.2 Software integration testing shall be planned, specified and executed in accordance with ISO 26262-8:2011, Clause 9.

NOTE 1 Based on the definitions in ISO 26262-8:2011, Clause 9, the software integration test objects are the software components.

NOTE 2 For model-based development, the test objects can be the models associated with the software components.

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

a) compliance with the software architectural design in accordance with Clause 7;

b) compliance with the specification of the hardware-software interface in accordance with ISO 26262-4:2011, Clause 7;

c) the specified functionality;

d) robustness; and

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

e) sufficient resources to support the functionality.

Table 13 — Methods for software integration testing

Methods		ASIL			
		A	B	C	D
1a	Requirements-based test[a]	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[b]	+	+	++	++
1d	Resource usage test[cd]	+	+	+	++
1e	Back-to-back comparison test between model and code, if applicable[e]	+	+	++	++

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

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

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

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

[e] This method requires a model that can simulate the functionality of the software components. Here, the model and code are stimulated in the same way and results compared with each other.

10.4.4 To enable the specification of appropriate test cases for the software integration test methods selected in accordance with 10.4.3, test cases shall be derived using the methods listed in Table 14.

Table 14 — Methods for deriving test cases for software integration testing

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

[a] Equivalence classes can be identified based on the division of inputs and outputs, such that a representative test value can be selected for each class.

[b] This method applies to parameters or variables, values approaching and crossing the boundaries and out of range values.

[c] Error guessing tests can be based on data collected through a “lessons learned” process and expert judgment.

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

10.4.6 This subclause applies to ASIL (A), (B), C and D, in accordance with 4.3: To evaluate the completeness of test cases and to obtain confidence that there is no unintended functionality, the structural coverage shall be measured in accordance with the metrics listed in Table 15. If the achieved structural coverage is considered insufficient, either additional test cases shall be specified or a rationale shall be provided.

EXAMPLE Analysis of structural coverage can reveal shortcomings in the requirement-based test cases, inadequacies in the requirements, dead code, deactivated code or unintended functionality.

Table 15 — Structural coverage metrics at the software architectural level

Methods		ASIL			
		A	B	C	D
1a	Function coverage[a]	+	+	++	++
1b	Call coverage[b]	+	+	++	++

[a] Method 1a refers to the percentage of executed software functions. This evidence can be achieved by an appropriate software integration strategy.

[b] Method 1b refers to the percentage of executed software function calls.

NOTE 1 The structural coverage can be determined using appropriate software tools.

NOTE 2 In the case of model-based development, software architecture testing can be performed at the model level using analogous structural coverage metrics for models.

10.4.7 It shall be verified that the embedded software that is to be included as part of a production release in accordance with ISO 26262-4:2011, Clause 11, contains all the specified functions, and only contains other unspecified functions if these functions do not impair the compliance with the software safety requirements.

EXAMPLE In this context unspecified functions include code used for debugging or instrumentation.

NOTE If deactivation of these unspecified functions can be ensured, this is an acceptable means of compliance with this requirement. Otherwise the removal of such code is a change (see ISO 26262-8:2011, Clause 8).

10.4.8 The test environment for software integration testing shall correspond as closely as possible to the target environment. If the software integration testing is not carried out in the target environment, the differences in the source and object code and the differences between the test environment and the target environment shall be analysed in order to specify additional tests in the target environment during the subsequent test phases.

NOTE 1 Differences between the test environment and the target environment can arise in the source or object code, for example, due to different bit widths of data words and address words of the processors.

NOTE 2 Depending on the scope of the tests and the hierarchical level of integration, the appropriate test environments for the execution of the software elements are used. Such test environments can be the target processor for final integration, or a processor emulator or a development system for the previous integration steps.

NOTE 3 Software integration testing can be executed in different environments, for example:

— model-in-the-loop tests;

— software-in-the-loop tests;

— processor-in-the-loop tests; and

— hardware-in-the-loop tests.

10.5 Work products

10.5.1 Software verification plan (refined) resulting from requirements 10.4.1 to 10.4.6 and 10.4.8.

10.5.2 Software verification specification (refined) resulting from requirements 10.4.1, 10.4.2, 10.4.4, 10.4.5, 10.4.7 and 10.4.8.

10.5.3 Embedded software resulting from requirement 10.4.1.

10.5.4 Software verification report (refined) resulting from requirement 10.4.2.

11 Verification of software safety requirements

11.1 Objectives

The objective of this sub-phase is to demonstrate that the embedded software fulfils the software safety requirements.

11.2 General

The purpose of the verification of the software safety requirements is to demonstrate that the embedded software satisfies its requirements in the target environment.

11.3 Inputs to this clause

11.3.1 Prerequisites

The following information shall be available:

— software architectural design specification in accordance with 7.5.1;

— safety plan (refined) in accordance with 7.5.2;

— software safety requirements specification (refined) in accordance with 7.5.3;

— software verification plan (refined) in accordance with 10.5.1;

— software verification specification (refined) in accordance with 10.5.2;

— software verification report (refined) in accordance with 10.5.4; and

— integration testing report in accordance with ISO 26262-4:2011, 8.5.3.

11.3.2 Further supporting information

The following information can be considered:

— validation plan (refined) (see ISO 26262-4:2011, 6.5.3);

— technical safety concept (see ISO 26262-4:2011, 7.5.1);

— system design specification (see ISO 26262-4:2011, 7.5.2);

— tool application guidelines in accordance with 5.5.4; and

— guidelines for the application of methods (from external source).

11.4 Requirements and recommendations

11.4.1 The verification of the software safety requirements shall be planned, specified and executed in accordance with ISO 26262-8:2011, Clause 9.

11.4.2 To verify that the embedded software fulfils the software safety requirements, tests shall be conducted in the test environments listed in Table 16.

NOTE Test cases that already exist, for example from software integration testing, can be re-used.

Table 16 — Test environments for conducting the software safety requirements verification

Methods		ASIL			
		A	B	C	D
1a	Hardware-in-the-loop	+	+	++	++
1b	Electronic control unit network environments[a]	++	++	++	++
1c	Vehicles	++	++	++	++

[a] Examples include test benches partially or fully integrating the electrical systems of a vehicle, “lab-cars” or “mule” vehicles, and “rest of the bus” simulations.

11.4.3 The testing of the implementation of the software safety requirements shall be executed on the target hardware.

11.4.4 The results of the verification of the software safety requirements shall be evaluated with regard to:

a) compliance with the expected results;

b) coverage of the software safety requirements; and

c) pass or fail criteria.

11.5 Work products

11.5.1 Software verification plan (refined) resulting from requirements 11.4.1 to 11.4.3.

11.5.2 Software verification specification (refined) resulting from requirements 11.4.1 to 11.4.3.

11.5.3 Software verification report (refined) resulting from requirements 11.4.1 and 11.4.4.

Annex A
(informative)

Overview of and workflow of management of product development at the software level

Table A.1 provides an overview of objectives, prerequisites and work products of the particular phases of the product development at the software level.

Table A.1 — Overview of product development at the software level

Clause	Objectives	Prerequisites	Work products
5 Initiation of product development at the software level	Plan and initiate the functional safety activities for the sub-phases of the software development activity.	Project plan (refined) (see ISO 26262-4:2011, 5.5.1) Safety plan (refined) (see ISO 26262-4:2011, 5.5.2) Technical safety concept (see ISO 26262-4:2011, 7.5.1) System design specification (see ISO 26262-4:2011, 7.5.2) Item integration and testing plan (refined) (see ISO 26262-4:2011, 7.5.4)	5.5.1 Safety plan (refined) 5.5.2 Software verification plan 5.5.3 Design and coding guidelines for modelling and programming languages 5.5.4 Tool application guidelines
6 Specification of software safety requirements	Specify software safety requirements. The software safety requirements are derived from the technical safety concept and the system design specification. Detail the hardware-software interface requirements. Verify that the software safety requirements and the hardware-software interface requirements are consistent with the technical safety concept and the system design specification.	Technical safety concept (see ISO 26262-4:2011, 7.5.1) System design specification (see ISO 26262-4:2011, 7.5.2) Hardware-software interface specification (see ISO 26262-4:2011, 7.5.6) Safety plan (refined) (see 5.5.1) Software verification plan (see 5.5.2)	6.5.1 Software safety requirements specification 6.5.2 Hardware-software interface specification (refined) 6.5.3 Software verification plan (refined) 6.5.4 Software verification report
7 Software architectural design	Develop a software architectural design that realizes the software safety requirements. Verify the software architectural design.	Safety plan (refined) (see 5.5.1) Design and coding guidelines for modelling and programming languages (see 5.5.3) Hardware-software interface specification (see ISO 26262-4:2011, 7.5.6) Software safety requirements specification (see 6.5.1) Software verification plan (refined) (see 6.5.3) Software verification report (refined) (see 6.5.4)	7.5.1 Software architectural design specification 7.5.2 Safety plan (refined) 7.5.3 Software safety requirements specification (refined) 7.5.4 Safety analysis report 7.5.5 Dependent failures analysis report 7.5.6 Software verification report (refined)

Table A.1 (*continued*)

Clause	Objectives	Prerequisites	Work products
8 Software unit design and implementation	Specify the software units in accordance with the software architectural design and the associated software safety requirements. Implement the software units as specified. Static verification of the software unit design and their implementation.	Design and coding guidelines for modelling and programming languages (see 5.5.3) Software verification plan (refined) (see 6.5.3) Software architectural design specification (see 7.5.1) Safety plan (refined) (see 7.5.2) Software safety requirements specification (refined) (see 7.5.3) Software verification report (refined) (see 7.5.6)	8.5.1 Software unit design specification 8.5.2 Software unit implementation 8.5.3 Software verification report (refined)
9 Software unit testing	Demonstrate that the software units fulfil the software unit design specifications and do not contain undesired functionality.	Hardware-software interface specification (refined) (see 6.5.2) Software verification plan (refined) (see 6.5.3) Safety plan (refined) (see 7.5.2) Software unit design specification (see 8.5.1) Software unit implementation (see 8.5.2) Software verification report (refined) (see 8.5.3)	9.5.1 Software verification plan (refined) 9.5.2 Software verification specification 9.5.3 Software verification report (refined)
10 Software integration and testing	Integrate the software elements. Demonstrate that the software architectural design is realized by the embedded software.	Hardware-software interface specification (refined) (6.5.2) Software architectural design specification (see 7.5.1) Safety plan (refined) (see 7.5.2) Software unit implementation (see 8.5.2) Software verification plan (refined) (see 9.5.1) Software verification specification (refined) (see 9.5.2) Software verification report (refined) (see 9.5.3)	10.5.1 Software verification plan (refined) 10.5.2 Software verification specification (refined) 10.5.3 Embedded software 10.5.4 Software verification report (refined)
11 Verification of software safety requirements	Demonstrate that the embedded software fulfils the software safety requirements.	Software architectural design specification (see 7.5.1) Safety plan (refined) (see 7.5.2) Software safety requirements specification (refined) (see 7.5.3) Software verification plan (refined) (see 10.5.1) Software verification specification (refined) (see 10.5.2) Software verification report (refined) (see 10.5.4) Integration testing report (see ISO 26262-4:2011, 8.5.2)	11.5.1 Software verification plan (refined) 11.5.2 Software verification specification (refined) 11.5.3 Software verification report (refined)

Table A.1 (*continued*)

Clause	Objectives	Prerequisites	Work products
Annex C Software configuration	Enable controlled changes in the behaviour of the software for different applications	See applicable prerequisites of the relevant phases of the safety lifecycle in which software configuration is applied.	C.5.1 Configuration data specification C.5.2 Calibration data specification C.5.3 Safety plan (refined) C.5.4 Configuration data C.5.5 Calibration data C.5.6 Software verification plan (refined) C.5.7 Verification specification C.5.8 Verification report

Annex B
(informative)

Model-based development

B.1 Objectives

This annex describes the concept of model-based development of in-vehicle software and outlines its implications on the product development at the software level.

B.2 General

Mathematical modelling, which has been extensively used in many engineering domains, is also gaining widespread use in the development of embedded software. In the automotive sector, modelling is used for the conceptual capture of the functionality to be realized (open/closed loop control, monitoring) as well as for the simulation of real physical system behaviours (vehicle environment).

Modelling is usually carried out with commercial off-the-shelf modelling and simulation software tools. They support the development and definition of system/software elements, and their connections and interfaces by semi-formal graphical models. These models employ editable, hierarchical block diagrams (e.g. control diagrams) and extended state transition diagrams (e.g. state charts). The software tools provide the necessary means of description, computation techniques and interpreters/compilers. Graphical editors permit an intuitive development and description of complex models. Hierarchically structured modularity is used in order to control complexity. A model consists of function blocks with well-defined inputs and outputs. Function blocks are connected within the block diagram by directed edges between their interfaces, which describe signal flows. With this, they represent equations in the mathematical model, which relate the interface variables of different elements. The connection lines represent causally motivated directions of action, which define the outputs of one block as the inputs of another. Other tool-specific modelling semantics also can be used to impose order of execution and timing. The hierarchy of elements can contain several levels of refinement.

Such models can be simulated, i.e. executed. During simulation the calculation causality follows the defined directions of action until the entire model has been processed. There is a range of different solvers available for solving the equations described by the model. Variable-step solvers are used primarily for modelling the vehicle and the environment. For the development of embedded software, fixed-step solvers are used, which represent a necessary prerequisite for efficient code generation.

The modelling style described is used extensively within the scope of model-based development of embedded in-vehicle software. Typically, both an executable model of the control software (e.g. a functional model) and a model of the surrounding system (e.g. a vehicle model) and its environment (e.g. an environment model) are created early in the development cycle and are simulated together. This way, it is possible to model even highly complex automotive systems with a high degree of detail at an acceptable calculation speed and to simulate their behaviour close to reality. While the vehicle/environment model is gradually replaced during the course of development by the real system and its real environment, the functional model can serve as a blueprint for the implementation of embedded software on the control unit through code generation.

One characteristic of the model-based development paradigm is the fact that the functional model not only specifies the desired function but also provides design information and finally even serves as the basis of the implementation by means of code generation. In other words, such a functional model represents specification aspects as well as design and implementation aspects. In practice, these different aspects are reflected in an evolution of the functional model from an early specification model via a design model to an implementation model and finally its automatic transformation into code (model evolution). In comparison to code-based software development with a clear separation of phases, in model-based development a stronger coalescence of the phases “Software safety requirements”, “Software architectural design” and “Software unit design and implementation” can be noted. Moreover, one and the same graphical modelling notation is used during the consecutive development stages. Verification activities can also be treated differently since models can be used as a useful source of information for the testing process (e.g. model-based testing), or can serve as the object to be verified. The seamless utilization of models facilitates highly consistent and efficient development.

NOTE The paradigm of model-based development does not depend on the existence of the model type mentioned above. Alternative models such as UML can be used.

Annex C
(normative)

Software configuration

C.1 Objectives

The objective of software configuration is to enable controlled changes in the behaviour of the software for different applications.

C.2 General description

Configurable software enables the development of application specific software using configuration and calibration data (see Figure C.1).

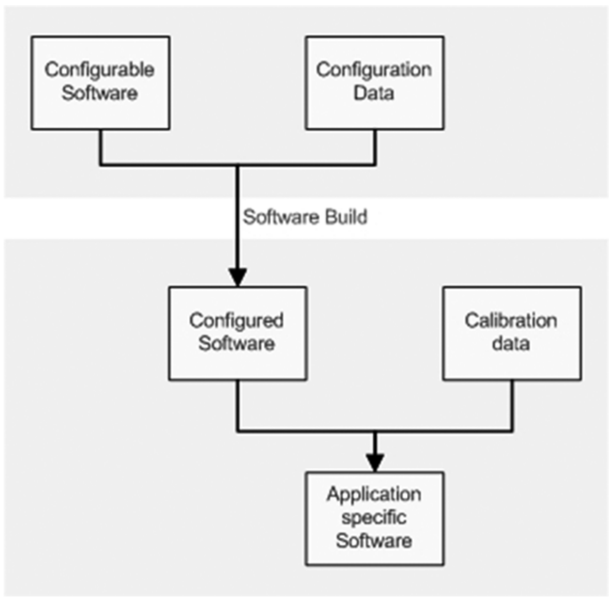


Figure C.1 — Creating application specific software

C.3 Inputs to this clause

C.3.1 Prerequisites

The prerequisites are in accordance with the relevant phases in which software configuration is applied.

C.3.2 Further supporting information

See applicable further supporting information of the relevant phases in which software configuration is applied.

C.4 Requirements and recommendations

C.4.1 The configuration data shall be specified to ensure the correct usage of the configurable software during the safety lifecycle. This shall include:

a) the valid values of the configuration data;

b) the intent and usage of the configuration data;

c) the range, scaling, units; and

d) the interdependencies between different elements of the configuration data.

C.4.2 Verification of the configuration data shall be performed to ensure:

a) the use of values within their specified range; and

b) the compatibility with the other configuration data.

NOTE The testing of the configured software is performed within the test phases of the software lifecycle [see Clauses 9 (Software unit testing), 10 (Software integration and testing), 11 (Verification of software safety requirements) and ISO 26262-4:2011, Clause 8 (Item integration and testing)].

C.4.3 The ASIL of the configuration data shall equal the highest ASIL of the configurable software by which it is used.

C.4.4 The verification of configurable software shall be planned, specified and executed in accordance with ISO 26262-8:2011, Clause 9. Configurable software shall be verified for the configuration data set that is to be used for the item development under consideration.

NOTE Only that part of the embedded software whose behaviour depends on the configuration data is verified against the configuration data set.

C.4.5 For configurable software a simplified software safety lifecycle in accordance with Figures C.2 or C.3 may be applied.

NOTE A combination of the following verification activities can achieve the complete verification of the configured software:

a) “verification of the configurable software”,

b) “verification of the configuration data”, and

c) “verification of the configured software”.

This is achieved by either

— verifying a range of admissible configuration data in a) and showing compliance to this range in b), or

— by showing compliance to the range of admissible configuration data in b) and performing c).

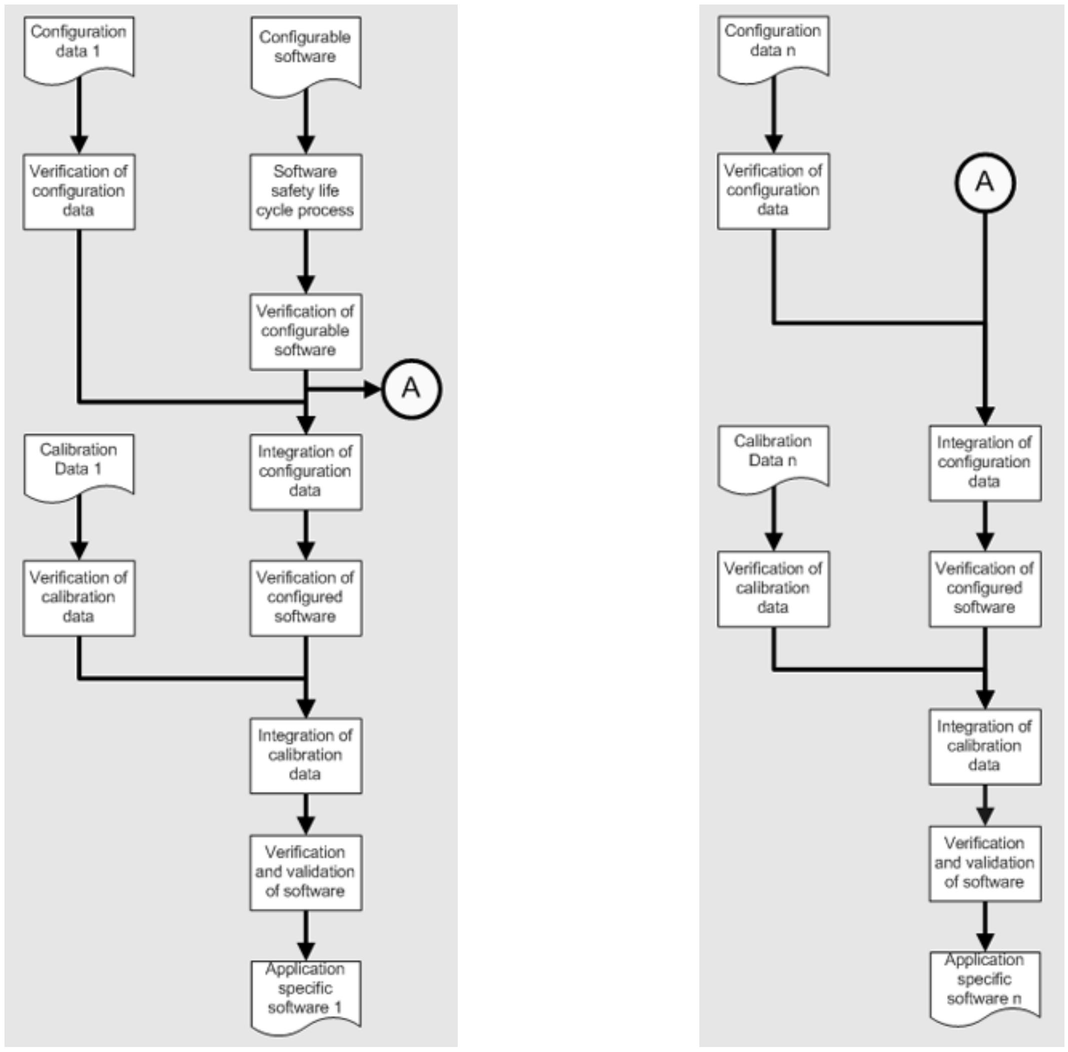

Configuration data 1
Configurable software
Verification of configuration data
Software safety life cycle process
Verification of configurable software
A
Calibration Data 1
Integration of configuration data
Verification of calibration data
Verification of configured software
Integration of calibration data
Verification and validation of software
Application specific software 1
Configuration data n
Verification of configuration data
A
Calibration Data n
Integration of configuration data
Verification of calibration data
Verification of configured software
Integration of calibration data
Verification and validation of software
Application specific software n

Figure C.2 — Variants of the reference phase model for software development with configurable software and different configuration data

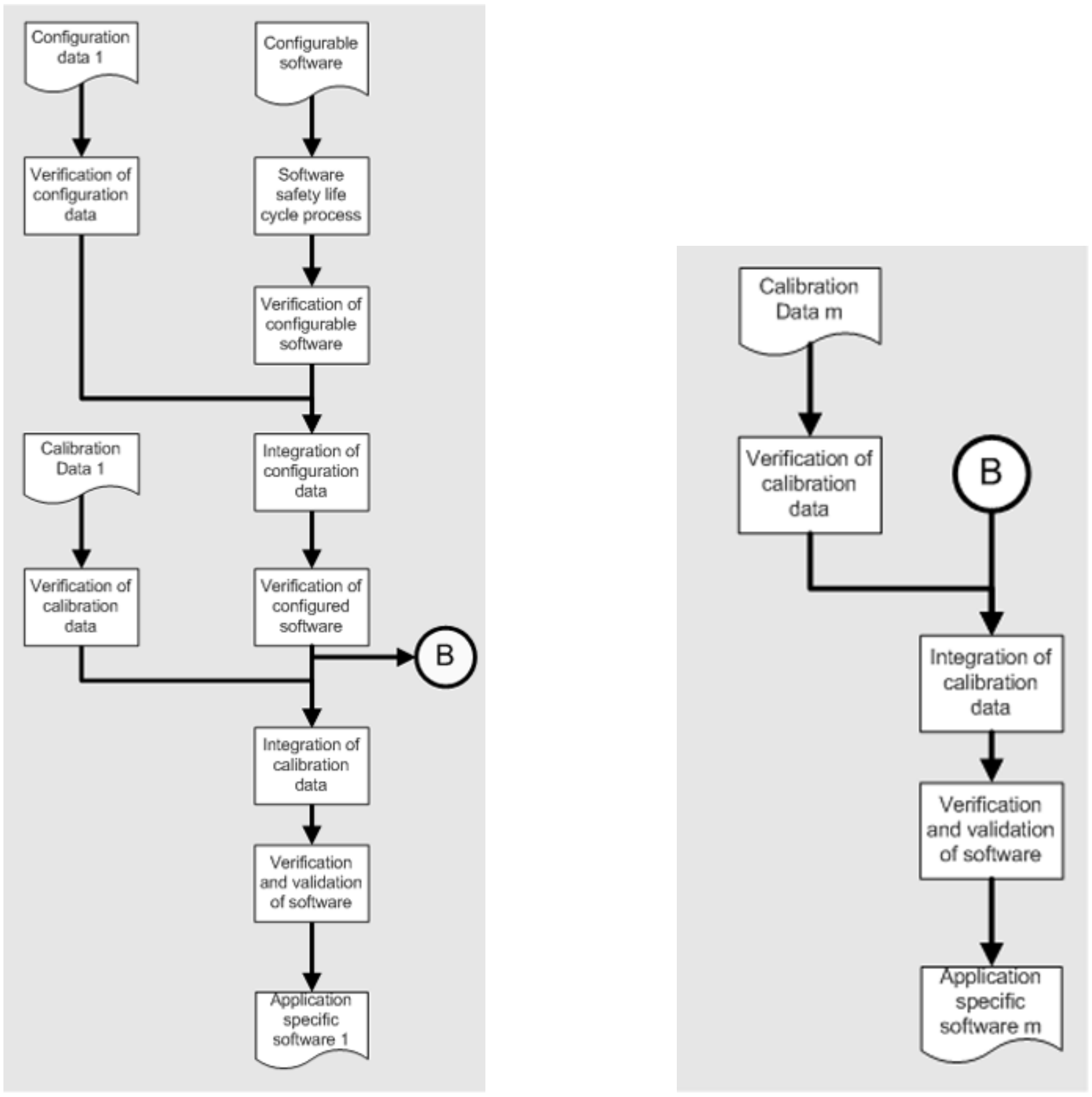

Configuration data 1
Configurable software
Verification of configuration data
Software safety life cycle process
Verification of configurable software
Calibration Data 1
Integration of configuration data
Verification of calibration data
Verification of configured software
B
Integration of calibration data
Verification and validation of software
Application specific software 1
Calibration Data m
Verification of calibration data
B
Integration of calibration data
Verification and validation of software
Application specific software m

Figure C.3 — Variants of the reference phase model for software development with configurable software and different calibration data

C.4.6 The calibration data associated with software components shall be specified to ensure the correct operation and expected performance of the configured software. This shall include:

a) the valid values of the calibration data;

b) the intent and usage of the calibration data;

c) the range, scaling and units, if applicable, with their dependence on the operating state;

d) the known interdependencies between different calibration data; and

NOTE Interdependencies can exist between calibration data within one calibration data set or between calibration data in different calibration data sets such as those applied to related functions implemented in the software of separate ECUs.

e) the known interdependencies between configuration data and calibration data.

NOTE Configuration data can have an impact on the configured software that uses the calibration data.

C.4.7 The verification of the calibration data shall be planned, specified and executed in accordance with ISO 26262-8:2011, Clause 9. The verification of calibration data shall examine whether the calibration data is within its specified boundaries.

NOTE Verification of calibration data can also be performed within application-specific software verification, or at runtime by the configurable software.

C.4.8 The ASIL of the calibration data shall equal the highest ASIL of the software safety requirements it can violate.

C.4.9 To detect unintended changes of safety-related calibration data, mechanisms for the detection of unintended changes of data as listed in Table C.1 shall be applied.

Table C.1 — Mechanisms for the detection of unintended changes of data

Methods		ASIL			
		A	B	C	D
1a	Plausibility checks on calibration data	++	++	++	++
1b	Redundant storage of calibration data	+	+	+	++
1c	Error detecting codes[a]	+	+	+	++

[a] Error detecting codes may also be implemented in the hardware in accordance with ISO 26262-5.

C.4.10 The planning of the generation and application of calibration data shall specify:

a) the procedures that shall be followed;

b) the tools for generating calibration data; and

c) the procedures for verifying calibration data.

NOTE Verification of calibration data can include checking the value ranges of calibration data or the interdependencies between different calibration data.

C.5 Work products

C.5.1 Configuration data specification resulting from requirements C.4.1 and C.4.3.

C.5.2 Calibration data specification resulting from requirement C.4.6.

C.5.3 Safety plan (refined) resulting from requirements C.4.1, C.4.4, C.4.5, C.4.9 and C.4.10.

C.5.4 Configuration data resulting from requirement C.4.3.

C.5.5 Calibration data resulting from requirement C.4.8.

C.5.6 Software verification plan (refined) resulting from requirements C.4.2, C.4.4, C.4.7 and C.4.10.

C.5.7 Verification specification resulting from requirements C.4.4 and C.4.7.

C.5.8 Verification report resulting from requirements C.4.1, C.4.4, C.4.7 and C.4.8.

Annex D
(informative)

Freedom from interference between software elements

D.1 Objectives

The objective is to provide examples of faults that can cause interference between software elements (e.g. software elements of different software partitions). Additionally, this Annex provides examples of possible mechanisms that can be considered for the prevention, or detection and mitigation of the listed faults.

NOTE The capability and effectiveness of the mechanisms used to prevent, or to detect and mitigate relevant faults is assessed during development.

D.2 General

D.2.1 Achievement of freedom from interference

To develop or evaluate the achievement of freedom from interference between software elements, the effects of the exemplary faults, and the propagation of the possible resulting failures can be considered.

D.2.2 Timing and execution

With respect to timing constraints, the effects of faults such as those listed below can be considered for the software elements executed in each software partition:

— blocking of execution;

— deadlocks;

— livelocks;

— incorrect allocation of execution time;

— incorrect synchronization between software elements.

EXAMPLE Mechanisms such as cyclic execution scheduling, fixed priority based scheduling, time triggered scheduling, monitoring of processor execution time, program sequence monitoring and arrival rate monitoring can be considered.

D.2.3 Memory

With respect to memory, the effects of faults such as those listed below can be considered for software elements executed in each software partition:

— corruption of content;

— read or write access to memory allocated to another software element.

EXAMPLE Mechanisms such as memory protection, parity bits, error-correcting code (ECC), cyclic redundancy check (CRC), redundant storage, restricted access to memory, static analysis of memory accessing software and static allocation can be used.

D.2.4 Exchange of information

With respect to the exchange of information, the causes for faults or effects of faults such as those listed below can be considered for each sender or each receiver:

— repetition of information;

— loss of information;

— delay of information;

— insertion of information;

— masquerade or incorrect addressing of information;

— incorrect sequence of information;

— corruption of information;

— asymmetric information sent from a sender to multiple receivers;

— information from a sender received by only a subset of the receivers;

— blocking access to a communication channel.

NOTE The exchange of information between elements executed in different software partitions or different ECUs includes signals, data, messages, etc.

EXAMPLE 1 Information can be exchanged using I/O-devices, data busses, etc.

EXAMPLE 2 Mechanisms such as communication protocols, information repetition, loop back of information, acknowledgement of information, appropriate configuration of I/O pins, separated point-to-point unidirectional communication objects, unambiguous bidirectional communication objects, asynchronous data communication, synchronous data communication, event-triggered data buses, event-triggered data buses with time-triggered access, time-triggered data busses, mini-slotting, bus arbitration by priority and bus arbitration by priority can be used.

EXAMPLE 3 Communication protocols can contain information such as identifiers for communication objects, keep alive messages, alive counters, sequence numbers, error detection codes and error-correcting codes.

Bibliography

[1] ISO/IEC 12207, *Systems and software engineering — Software life cycle processes*

[2] IEC 61508 (all parts), *Functional safety of electrical/electronic/programmable electronic safety-related systems*

[3] MISRA-C:2004, *Guidelines for the use of the C language in critical systems*, ISBN 978-0-9524156-2-6, MIRA, October 2004

[4] MISRA AC AGC, *Guidelines for the application of MISRA-C:2004 in the context of automatic code generation*, ISBN 978-1-906400-02-6, MIRA, November 2007

This page is intentionally blank.

This page is intentionally blank.

This page is intentionally blank.

ICS 43.040.10

Price based on 40 pages