INTERNATIONAL STANDARD

ISO 26262-6

Second edition
2018-12

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 26262-6:2018(E)

COPYRIGHT PROTECTED DOCUMENT

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

Published in Switzerland

Contents

Page

Foreword

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

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

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

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

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

This document was prepared by Technical Committee ISO/TC 22, *Road vehicles Subcommittee*, SC 32, *Electrical and electronic components and general system aspects*.

This edition of ISO 26262 series of standards cancels and replaces the edition ISO 26262:2011 series of standards, which has been technically revised and includes the following main changes:

— requirements for trucks, buses, trailers and semi-trailers;

— extension of the vocabulary;

— more detailed objectives;

— objective oriented confirmation measures;

— management of safety anomalies;

— references to cyber-security;

— updated target values for hardware architecture metrics;

— guidance on model based development and software safety analysis;

— evaluation of hardware elements;

— additional guidance on dependent failure analysis;

— guidance on fault tolerance, safety related special characteristics and software tools;

— guidance for semiconductors;

— requirements for motorcycles; and

— general restructuring of all parts for improved clarity.

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

A list of all parts in the ISO 26262 series can be found on the ISO website.

Introduction

The ISO 26262 series of standards is the adaptation of IEC 61508 series of standards to address the sector specific needs 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 in the development of road vehicles. Development and integration of automotive functionalities strengthen the need for functional safety and the need to provide evidence that functional 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, these being considered within the scope of functional safety. ISO 26262 series of standards includes guidance to mitigate these risks by providing appropriate requirements and processes.

To achieve functional safety, the ISO 26262 series of standards:

a) provides a reference for the automotive safety lifecycle and supports the tailoring of the activities to be performed during the lifecycle phases, i.e., development, production, operation, service and decommissioning;

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

c) uses ASILs to specify which of the requirements of ISO 26262 are applicable to avoid unreasonable residual risk;

d) provides requirements for functional safety management, design, implementation, verification, validation and confirmation measures; and

e) provides requirements for relations between customers and suppliers.

The ISO 26262 series of standards is concerned with functional safety of E/E systems that is achieved through safety measures including safety mechanisms. It also provides a framework within which safety-related systems based on other technologies (e.g. mechanical, hydraulic and pneumatic) can be considered.

The achievement of 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 the management processes.

Safety is intertwined with common function-oriented and quality-oriented activities and work products. The ISO 26262 series of standards addresses the safety-related aspects of these activities and work products.

Figure 1 shows the overall structure of the ISO 26262 series of standards. The ISO 26262 series of standards 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 among ISO 26262-3, ISO 26262-4, ISO 26262-5, ISO 26262-6 and ISO 26262-7;

— for motorcycles:

  — ISO 26262-12:2018, Clause 8 supports ISO 26262-3;

  — ISO 26262-12:2018, Clauses 9 and 10 support ISO 26262-4;

— 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 ISO 26262-2:2018, Clause 6.

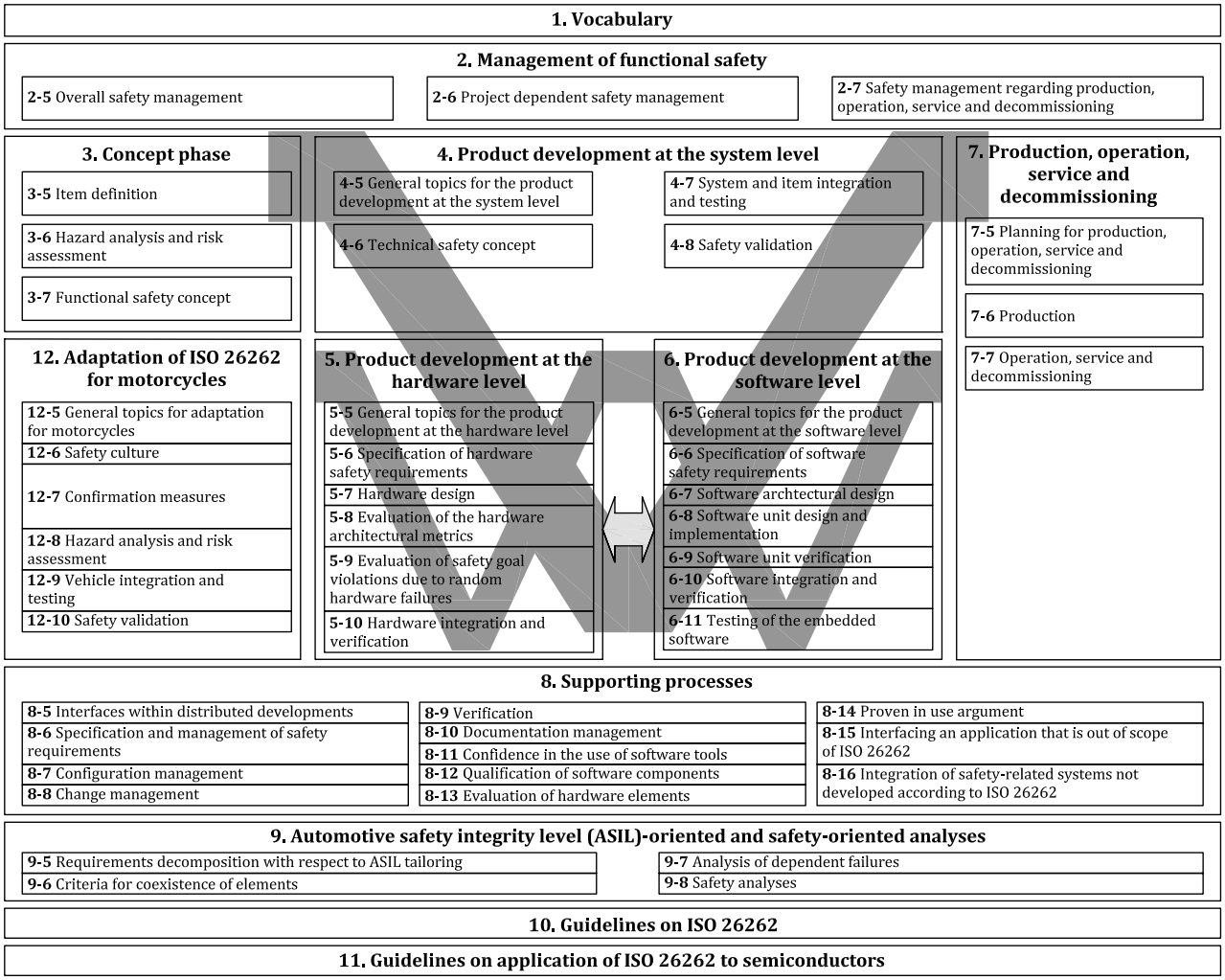


Figure 1 — Overview of the ISO 26262 series of standards

Road vehicles — Functional safety —

Part 6:
Product development at the software level

1 Scope

This document 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 road vehicles, excluding mopeds. This document does not address unique E/E systems in special vehicles such as E/E systems designed for drivers with disabilities.

NOTE Other dedicated application-specific safety standards exist and can complement the ISO 26262 series of standards or vice versa.

Systems and their components released for production, or systems and their components already under development prior to the publication date of this document, are exempted from the scope of this edition. This document addresses alterations to existing systems and their components released for production prior to the publication of this document by tailoring the safety lifecycle depending on the alteration. This document addresses integration of existing systems not developed according to this document and systems developed according to this document by tailoring the safety lifecycle.

This document addresses possible hazards caused by malfunctioning behaviour of safety-related E/E 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 safety-related E/E systems.

This document describes a framework for functional safety to assist the development of safety-related E/E systems. This framework is intended to be used to integrate functional safety activities into a company-specific development framework. Some requirements have a clear technical focus to implement functional safety into a product; others address the development process and can therefore be seen as process requirements in order to demonstrate the capability of an organization with respect to functional safety.

This document does not address the nominal performance of E/E systems.

This document specifies the requirements for product development at the software level for automotive applications, including the following:

— general topics for product development at the software level;

— specification of the software safety requirements;

— software architectural design;

— software unit design and implementation;

— software unit verification;

— software integration and verification; and

— testing of the embedded software.

It also specifies requirements associated with the use of configurable software.

Annex A provides an overview on objectives, prerequisites and work products of this document.

2 Normative references

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

ISO 26262-1, *Road Vehicles — Functional Safety — Part 1: Vocabulary*

ISO 26262-2:2018, *Road Vehicles — Functional Safety — Part 2: Management of functional safety*

ISO 26262-3:2018, *Road vehicles — Functional safety — Part 3: Concept phase*

ISO 26262-4:2018, *Road vehicles — Functional safety — Part 4: Product development at the system level*

ISO 26262-5:2018, *Road vehicles — Functional safety — Part 5: Product development at the hardware level*

ISO 26262-7:2018, *Road vehicles — Functional safety — Part 7: Production, operation, service and decommissioning*

ISO 26262-8:2018, *Road vehicles — Functional safety — Part 8: Supporting processes*

ISO 26262-9:2018, *Road vehicles — Functional safety — Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses*

3 Terms and definitions

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

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

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

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

4 Requirements for compliance

4.1 Purpose

This clause describes how:

a) to achieve compliance with the ISO 26262 series of standards;

b) to interpret the tables used in the ISO 26262 series of standards; and

c) to interpret the applicability of each clause, depending on the relevant ASIL(s).

4.2 General requirements

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

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

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

Informative content, including notes and examples, 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 the ISO 26262 series of standards 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.

4.3 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 listed highly recommended and recommended methods in accordance with the ASIL apply. It is allowed to substitute a highly recommended or recommended method by others not listed in the table, in this case, a rationale shall be given describing why these comply with the corresponding requirement. If a rationale can be given to comply with the corresponding requirement without choosing all entries, a further rationale for omitted methods is not necessary.

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 or even a selected single method 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; and

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

4.4 ASIL-dependent requirements and recommendations

The requirements or recommendations of each sub-clause shall be met 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 26262-9:2018, Clause 5, the ASIL resulting from the decomposition shall be met.

If an ASIL is given in parentheses in the ISO 26262 series of standards, the corresponding sub-clause 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.

4.5 Adaptation for motorcycles

For items or elements of motorcycles for which requirements of ISO 26262-12 are applicable, the requirements of ISO 26262-12 supersede the corresponding requirements in this document. Requirements of ISO 26262-2 that are superseded by ISO 26262-12 are defined in Part 12.

4.6 Adaptation for trucks, buses, trailers and semi-trailers

Content that is intended to be unique for trucks, buses, trailers and semi-trailers (T&B) is indicated as such.

5 General topics for the product development at the software level

5.1 Objectives

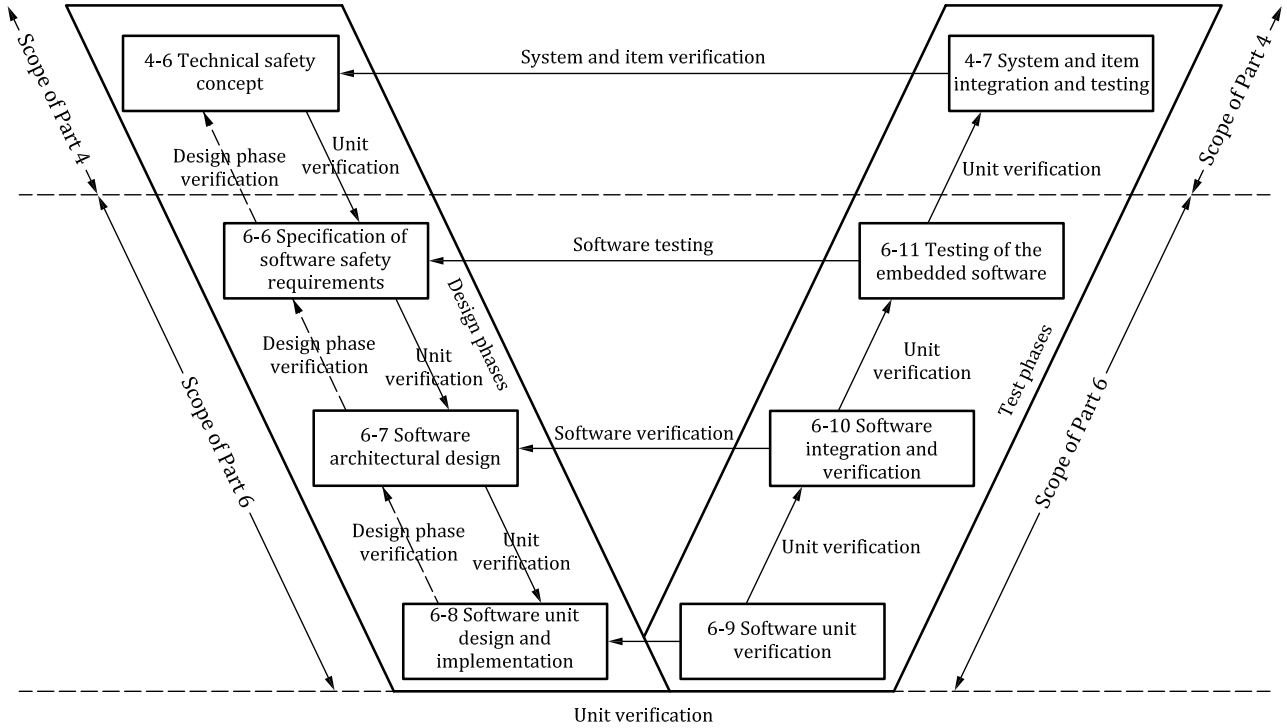
The objectives of this clause are:

a) to ensure a suitable and consistent software development process; and

b) to ensure a suitable software development environment.

5.2 General

The reference phase model for the development of software is given in Figure 2. Details concerning the treatment of configurable software are provided in Annex C.



NOTE Within the figure, the specific clauses of each part of the ISO 26262 series of standards 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 ISO 26262-4:2018, Clause 7.

Figure 2 — Reference phase model for the product development at the software level

NOTE 1 Development approaches or methods from agile software development can also be suitable for the development of safety-related software, but if the safety activities are tailored in this manner, ISO 26262-2:2018 6.4.5 is considered. However, agile approaches and methods cannot be used to omit safety measures or ignore the fundamental documentation, process or safety integrity of product rigour required for the achievement of functional safety.

EXAMPLE 1 Test Driven Development can be used to improve quality and testability of requirements.

EXAMPLE 2 Continuous integration based on an automated build system can support consistency of sub-phases and facilitate regression tests. Such a build system typically performs code generation, compiling and linking, static code analysis, documentation generation, testing and packaging. It allows, subject to tool chain and tool configuration, repeatable and, after changes, comparable production of software, documentation and test results.

NOTE 2 Cybersecurity can also be considered when developing the embedded software of a particular item, see ISO 26262-2:2018, 5.4.2.3. In order to be able to develop software, specific topics are addressed in this clause concerning the modelling, design and/or programming languages to be used, and the application of guidelines and tools.

NOTE 3 Tools used for software development can include tools other than software tools.

EXAMPLE 3 Tools used for testing phases.

5.3 Inputs to this clause

5.3.1 Prerequisites

The following information shall be available:

— (none)

5.3.2 Further supporting information

The following information can be considered:

— qualified software tools available (see ISO 26262-8:2018, Clause 11);

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

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

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

5.4 Requirements and recommendations

5.4.1 When developing the software of an item, software development processes and software development environments shall be used which:

a) are suitable for developing safety-related embedded software, including methods, guidelines, languages and tools;

b) support consistency across the sub-phases of the software development lifecycle and the respective work products; and

c) are compatible with the system and hardware development phases regarding required interaction and consistency of exchange of information.

NOTE 1 The sequencing of phases, tasks and activities, including iteration steps, for the software of an item intends 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).

NOTE 2 The software tool criteria evaluation report (see ISO 26262-8:2018, 11.5.1) or the software tool qualification report (see ISO 26262-8:2018, 11.5.2) can provide input to the tool usage.

5.4.2 The criteria that shall be considered when selecting a design, modelling or programming language are:

a) an unambiguous and comprehensible definition;

EXAMPLE Unambiguous definition of syntax and semantics or restriction to configuration of the development environment.

b) suitability for specifying and managing safety requirements according to ISO 26262-8:2018 Clause 6, if modelling is used for requirements engineering and management;

c) support the achievement of modularity, abstraction and encapsulation; and

d) support the use of structured constructs.

NOTE 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. However using assembly languages can require a suitable application or tailoring of all software development phases (e.g. requirements of Clause 8).

5.4.3 Criteria for suitable modelling, design or programming languages (see 5.4.2) that are not sufficiently addressed by the language itself shall be covered by the corresponding guidelines, or by the development environment, considering the topics listed in Table 1.

EXAMPLE 1 MISRA C (see Reference [3]) is a coding guideline for the programming language C and includes guidance for automatically generated code.

EXAMPLE 2 In the case of model based development with automatic code generation, guidelines can be applied at the model level as well as the code level. Appropriate modelling style guides including the MISRA AC series can be considered. Style guides for commercial tools are also possible guidelines.

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

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[d]	+	+	++	++
1e	Use of well-trusted design principles[e]	+	+	++	++
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+	++	++	++
1h	Use of naming conventions	++	++	++	++
1i	Concurrency aspects[f]	+	+	+	+

[a] An appropriate compromise of this topic with other requirements of this document may be required.

[b] The objectives of topic 1b include:

— 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 topic 1c is to impose principles of strong typing where these are not inherent in the language.

[d] Examples of defensive implementation techniques:

— Verify the divisor before a division operation (different from zero or in a specific range).

— Check an identifier passed by parameter to verify that the calling function is the intended caller.

— Use the “default” in switch cases to detect an error.

[e] Verification of the validity of the underlying assumptions, boundaries and conditions of application may be required.

[f] Concurrency of processes or tasks is not limited to executing software in a multi-core or multi-processor runtime environment.

5.5 Work products

5.5.1 Documentation of the software development environment resulting from requirements 5.4.1 to 5.4.3 and C.4.1 to C.4.11.

6 Specification of software safety requirements

6.1 Objectives

The objectives of this sub-phase are:

a) to specify or refine the software safety requirements which are derived from the technical safety concept and the system architectural design specification;

b) to define the safety-related functionalities and properties of the software required for the implementation;

c) to refine the requirements of the hardware-software interface initiated in ISO 26262-4:2018, Clause 6; and

d) to verify that the software safety requirements and the hardware-software interface requirements are suitable for software development and are consistent with the technical safety concept and the system architectural design specification.

6.2 General

The technical safety requirements are refined and allocated to hardware and software during the system architectural design phase given in ISO 26262-4:2018, Clause 6. The specification of the software safety requirements considers in particular 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 requirements specification in accordance with ISO 26262-4:2018, 6.5.1;

— technical safety concept in accordance with ISO 26262-4:2018, 6.5.2;

— system architectural design specification in accordance with ISO 26262-4:2018, 6.5.3;

— hardware-software interface (HSI) specification in accordance with ISO 26262-4:2018, 6.5.4; and

— documentation of the software development environment in accordance with 5.5.1.

6.3.2 Further supporting information

The following information can be considered:

— hardware design specification (see ISO 26262-5:2018, 7.5.1); and

— specification of non-safety-related functions and properties of the software (from an external source).

6.4 Requirements and recommendations

6.4.1 The software safety requirements shall be derived considering the required safety-related functionalities and properties of the software, whose failures could lead to the violation of a technical safety requirement allocated to software.

NOTE 1 The software safety requirements are either derived directly from the technical safety requirements allocated to software or are requirements for software functions and properties that, if not fulfilled, could lead to a violation of the technical safety requirements allocated to software.

EXAMPLE 1 Safety-related functionality of the software can be:

— functions that enable the safe execution of a nominal function;

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

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

— self-test or monitoring functions related to the detection, indication and mitigation of failures in the operating system, basic software or the application software itself;

— functions related to on-board and off-board tests during production, operation, service and decommissioning;

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

— functions related to performance or time-critical operations.

EXAMPLE 2 Safety-related properties include robustness against erroneous inputs, independence or freedom from interference between different functionalities, or fault tolerance capabilities of the software.

NOTE 2 Safety-oriented analyses (see 7.4.10 or 7.4.11) can be used to identify additional software safety requirements or provide evidence for their achievement.

6.4.2 Specification of the software safety requirements derived from the technical safety requirements, the technical safety concept and the system architectural design in accordance with ISO 26262-4:2018, 6.4.1 and 6.4.3 shall consider:

a) the specification and management of safety requirements in accordance with ISO 26262-8:2018, 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 and each transition between the operating modes of the vehicle, the system, or the hardware, having an impact on the software.

EXAMPLE 4 Operating modes include shut-off or sleep, initialization, normal operation, degraded and advanced modes for testing or flash programming.

6.4.3 If ASIL decomposition is applied to the software safety requirements, ISO 26262-9:2018, Clause 5, shall be complied with.

6.4.4 The hardware-software interface specification initiated in ISO 26262-4:2018, Clause 6, shall be refined sufficiently to allow for the correct control and usage of the hardware by the software, 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, a specification of these functions and their properties in accordance with the applied quality management system shall be available.

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

6.4.7 The software safety requirements and the refined requirements of the hardware-software interface specification shall be verified in accordance with ISO 26262-8:2018, Clauses 6 and 9, to provide evidence for their:

a) suitability for software development;

b) compliance and consistency with the technical safety requirements;

c) compliance with the system design; and

d) 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 (HSI) specification (refined) resulting from requirement 6.4.4.

NOTE This work product refers to the same work product as given in ISO 26262-5:2018, 6.5.2.

6.5.3 Software verification report resulting from requirements 6.4.6 and 6.4.7.

7 Software architectural design

7.1 Objectives

The objectives of this sub-phase are:

a) to develop a software architectural design that satisfies the software safety requirements and the other software requirements;

b) to verify that the software architectural design is suitable to satisfy the software safety requirements with the required ASIL; and

c) to support the implementation and verification of the software.

7.2 General

The software architectural design represents the software architectural elements and their interactions in a hierarchical structure. Static aspects, such as interfaces between the 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. The software architecture for each microcontroller is also addressed by this sub-clause.

A software architectural design is able to satisfy both the software safety requirements as well as the other software requirements. Hence, in this sub-phase, safety-related and non-safety-related software requirements are handled within one development process.

The software architectural design provides the means to implement both the software requirements and the software safety requirements with the required ASIL and to manage the complexity of the detailed design and the implementation of the software.

7.3 Inputs to this clause

7.3.1 Prerequisites

The following information shall be available:

— documentation of the software development environment in accordance with 5.5.1;

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

— software safety requirements specification in accordance with 6.5.1.

7.3.2 Further supporting information

The following information can be considered:

— technical safety concept (see ISO 26262-4:2018, 6.5.2);

— system architectural design specification (see ISO 26262-4:2018, 6.5.3);

— qualified software components available (see ISO 26262-8:2018, Clause 12); and

— specification of non-safety-related functions and properties of the software and other software requirements in accordance with 6.4.5 (from an external source).

7.4 Requirements and recommendations

7.4.1 To avoid systematic faults in the software architectural design and in the subsequent development activities, the description of the software architectural design shall address the following characteristics supported by notations for software architectural design as listed in Table 2:

a) comprehensibility;

b) consistency;

c) simplicity;

d) verifiability;

e) modularity;

f) abstraction;

NOTE Abstraction can be supported by using hierarchical structures, grouping schemes or views to cover static, dynamic or deployment aspects of an architectural design.

g) encapsulation; and

h) maintainability.

Table 2 — Notations for software architectural design

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

[a] Natural language can complement the use of notations for example where some topics are more readily expressed in natural language or providing explanation and rationale for decisions captured in the notation.

[b] Semi-formal notations can include pseudocode or modelling with UML®, SysML®, Simulink® or Stateflow®.

NOTE UML®, SysML®, Simulink® and Stateflow® are examples of suitable products available commercially. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of these products.

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 systematic faults, the software architectural design shall exhibit the following characteristics by use of the principles listed in Table 3:

a) comprehensibility;

b) consistency;

c) simplicity;

d) verifiability;

e) modularity;

f) encapsulation; and

g) maintainability.

Table 3 — Principles for software architectural design

Principles		ASIL			
		A	B	C	D
1a	Appropriate hierarchical structure of the software components	++	++	++	++
1b	Restricted size and complexity of software components[a]	++	++	++	++
1c	Restricted size of interfaces[a]	+	+	+	++
1d	Strong cohesion within each software component[b]	+	++	++	++
1e	Loose coupling between software components[b,c]	+	++	++	++
1f	Appropriate scheduling properties	++	++	++	++
1g	Restricted use of interrupts[a,d]	+	+	+	++
1h	Appropriate spatial isolation of the software components	+	+	+	++
1i	Appropriate management of shared resources[e]	++	++	++	++

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

[b] Principles 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] Principle 1e addresses the management of dependencies between software components.

[d] Principle 1g can include minimizing the number, or using interrupts with a clear priority, in order to achieve determinism.

[e] Principle 1i applies for shared hardware resources as well as shared software resources in the case of coexistence. Such resource management can be implemented in software or hardware and includes safety mechanisms and/or process measures that prevent conflicting access to shared resources as well as mechanisms that detect and handle conflicting access to shared resources.

NOTE 1 An appropriate compromise between the principles listed in Table 3 can be necessary since the principles are not mutually exclusive.

NOTE 2 Indicators for high complexity can be:

— highly branched control or data flow;

— excessive number of requirements allocated to single design elements;

— excessive number of interfaces of one design element or interactions between design elements;

— complex types or excessive number of parameters;

— excessive number of global variables;

— difficulty in providing evidence for suitability and completeness of error detection and handling;

— difficulty in achieving the required test coverage; or

— comprehensibility only to a few experts or only to project participants.

NOTE 3 These properties and principles also apply to software routines (e.g. service routines for interrupt handling).

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

7.4.5 The software architectural design shall describe:

a) the static design aspects of the software architectural elements; and

NOTE 1 Static design aspects address:

— the software structure including its hierarchical levels;

— the data types and their characteristics;

— the external interfaces of the software components;

— the external interfaces of the embedded software;

— the global variables; and

— the constraints including the scope of the architecture and external dependencies.

NOTE 2 In the case of model-based development, modelling the structure can be an inherent part of the overall modelling activities. The structure modelled can be dependent on the selected modelling language.

b) the dynamic design aspects of the software architectural elements.

NOTE 3 Dynamic design aspects address:

— the functional chain of events and behaviour;

— the logical sequence of data processing;

— the control flow and concurrency of processes;

— the data flow through interfaces and global variables; and

— the temporal constraints.

NOTE 4 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 5 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 The software safety requirements shall be hierarchically allocated to the software components down to software units. As a result, each software component shall be developed in compliance with the highest ASIL of any of the requirements allocated to it.

NOTE During this process of design development and requirement allocation, splitting or further refinement of software safety requirements in accordance with Clause 6 can be necessary.

7.4.7 If a pre-existing software architectural element is used without modifications in order to meet the assigned safety requirements without being developed according to the ISO 26262 series of standards, then it shall be qualified in accordance with ISO 26262-8:2018, Clause 12.

NOTE 1 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.

NOTE 2 The suitability for reuse of software elements developed according to the ISO 26262 series of standards is performed during the verification of the software architectural design.

7.4.8 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 26262-9:2018, Clause 6.

7.4.9 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 4.4);

EXAMPLE Hardware feature such as a memory protection unit.

c) the element of the software that implements the software partitioning is developed in compliance with the highest ASIL assigned to any requirement of the software partitions; and

NOTE 4 In general the operating system provides or supports software partitioning.

d) evidence for the effectiveness of the software partitioning is generated during software integration and verification (in accordance with Clause 10).

7.4.10 Safety-oriented analysis shall be carried out at the software architectural level in accordance with ISO 26262-9:2018, Clause 8, in order to:

— provide evidence for the suitability of the software to provide the specified safety-related functions and properties as required by the respective ASIL;

NOTE 1 Safety-related properties include independence and freedom from interference requirements.

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

— support the specification and verify the effectiveness of the safety measures.

NOTE 2 Safety measures include safety mechanisms that have been derived from safety-oriented analyses and can cover issues associated with random hardware failures as well as software faults.

NOTE 3 See Annex E for additional information about the application of safety-oriented analysis at the software architectural level and the selection of appropriate safety measures.

7.4.11 If the implementation of software safety requirements relies on freedom from interference or sufficient independence between software components, dependent failures and their effects shall be analysed in accordance with ISO 26262-9:2018, Clause 7.

NOTE See Annex E and ISO 26262-9:2018, Annex C for additional information about the application of analyses of dependent failures at the software architectural level.

7.4.12 Depending on the results of the safety-oriented analyses at the software architectural level in accordance with 7.4.10 or 7.4.11, safety mechanisms for error detection and error handling shall be applied.

NOTE 1 Annex E provides guidance for deciding if safety mechanisms are required for a failure mode.

NOTE 2 Safety mechanisms for error detection can include:

— Range checks of input and output data;

— Plausibility check (e.g. using a reference model of the desired behaviour, assertion checks, or comparing signals from different sources);

— Detection of data errors (e.g. error detecting codes and multiple data storage);

— Monitoring of program execution by an external element such as an ASIC or another software element performing a watchdog function. Monitoring can be logical or temporal monitoring or both;

— Temporal monitoring of program execution;

— Diverse redundancy in the design; or

— Access violation control mechanisms implemented in software or hardware concerned with granting or denying access to safety-related shared resources.

NOTE 3 Safety mechanisms for error handling can include:

— Deactivation in order to achieve and maintain a safe state;

— Static recovery mechanism (e.g. recovery blocks, backward recovery, forward recovery and recovery through repetition);

— Graceful degradation by prioritizing functions to minimize the adverse effects of potential failures on functional safety;

— Homogenous redundancy in the design, which focuses primarily on controlling the effects of transient faults or random faults in the hardware on which a similar software is executed (e.g. temporal redundant execution of software);

— Diverse redundancy in the design which implies dissimilar software in each parallel path and focuses primarily on the prevention or control of systematic faults in the software;

— Correcting codes for data; or

— Access permission management implemented in software or hardware concerned with granting or denying access to safety-related shared resources.

NOTE 4 The software safety mechanisms (including general robustness mechanisms) can be reviewed at the system level to analyse the potential impact on the system behaviour and the consistency with the technical safety requirements.

7.4.13 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.

7.4.14 The software architectural design shall be verified in accordance with ISO 26262-8:2018, Clause 9 and by using the software architectural design verification methods listed in Table 4 to provide evidence that the following objectives are achieved:

a) the software architectural design is suitable to satisfy the software requirements with the required ASIL;

b) the review or investigation of the software architectural design provides evidence for the suitability of the design to satisfy the software requirements with the required ASIL;

c) compatibility with the target environment; and

NOTE Target environment is the environment in which the software is executed. This can include the operating system and the basic software, in addition to the target hardware and its resources as specified in 7.4.13.

d) adherence to design guidelines.

Table 4 — 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 behaviour of the design	+	+	+	++
1d	Prototype generation	o	o	+	++
1e	Formal verification	o	o	+	+
1f	Control flow analysis[b]	+	+	++	++
1g	Data flow analysis[b]	+	+	++	++
1h	Scheduling analysis	+	+	++	++

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

[b] Control and data flow analysis can 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 7.4.13.

7.5.2 Safety analysis report resulting from requirement 7.4.10.

7.5.3 Dependent failures analysis report resulting from requirement 7.4.11.

7.5.4 Software verification report resulting from requirement 7.4.14.

8 Software unit design and implementation

8.1 Objectives

The objectives of this sub-phase are:

a) to develop a software unit design in accordance with the software architectural design, the design criteria and the allocated software requirements which supports the implementation and verification of the software unit; and

b) to implement the software units as specified.

8.2 General

Based on the software architectural design, the detailed design of the software units is developed. The detailed design can be represented in the form of a model.

The implementation at the source code level can be manually or automatically generated from the design in accordance with the software development environment.

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

8.3 Inputs to this clause

8.3.1 Prerequisites

The following information shall be available:

— documentation of the software development environment in accordance with 5.5.1;

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

— software architectural design specification in accordance with 7.5.1;

— software safety requirements specification in accordance with 6.5.1;

— configuration data in accordance with C.5.3, if applicable; and

— calibration data in accordance with C.5.4, if applicable.

8.3.2 Further supporting information

The following information can be considered:

— technical safety concept (see ISO 26262-4:2018, 6.5.2);

— system architectural design specification (see ISO 26262-4:2018, 6.5.3);

— specification of non-safety-related functions and properties of the software (from an external source);

— safety analysis report (see 7.5.2); and

— dependent failure analysis report (see 7.5.3).

8.4 Requirements and recommendations

8.4.1 The requirements of this sub-clause shall be complied with if the software unit is a safety-related element.

NOTE "Safety-related" means that the unit implements safety requirements, or that the criteria for coexistence (see ISO 26262-9:2018, Clause 6) of the unit with other units are not satisfied.

8.4.2 The software unit design and implementation shall:

a) be suitable to satisfy the software requirements allocated to the software unit with the required ASIL;

b) be consistent with the software architectural design specification; and

c) be consistent with the hardware-software interface specification, if applicable.

EXAMPLE Consistency and integrity of the interfaces to other software units; correctness, accuracy and timeliness of input or output data.

8.4.3 To avoid systematic faults and to ensure that the software unit design achieves the following properties, the software unit design shall be described using the notations listed in Table 5.

a) consistency;

b) comprehensibility;

c) maintainability; and

d) verifiability.

Table 5 — Notations for software unit design

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

[a] Natural language can complement the use of notations for example where some topics are more readily expressed in natural language or provide an explanation and rationale for decisions captured in the notations.

EXAMPLE To avoid possible ambiguity of natural language when designing complex elements, a combination of an activity diagram with natural language can be used.

[b] Semi-formal notations can include pseudocode or modelling with UML®, SysML®, Simulink® or Stateflow®.

NOTE UML®, SysML®, Simulink® and Stateflow® are examples of suitable products available commercially. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of these products.

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.

8.4.4 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.

8.4.5 Design principles for software unit design and implementation at the source code level as listed in Table 6 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) verifiability.

Table 6 — Design principles for software unit design and implementation

Principle		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]	+	++	++	++
1c	Initialization of variables	++	++	++	++
1d	No multiple use of variable names[a]	++	++	++	++
1e	Avoid global variables or else justify their usage[a]	+	+	++	++
1f	Restricted use of pointers[a]	+	++	++	++
1g	No implicit type conversions[a]	+	++	++	++
1h	No hidden data flow or control flow	+	++	++	++
1i	No unconditional jumps[a]	++	++	++	++
1j	No recursions	+	+	++	++

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

NOTE For the C language, MISRA C (see Reference [3]) covers many of the principles listed in Table 6.

8.5 Work products

8.5.1 **Software unit design specification** resulting from requirements 8.4.2 to 8.4.5.

NOTE In the case of model-based development, the implementation model and supporting descriptive documentation, using methods listed in Tables 5 and 6, specifies the software units.

8.5.2 **Software unit implementation** resulting from requirement 8.4.5.

9 Software unit verification

9.1 Objectives

The objectives of this sub-phase are:

a) to provide evidence that the software unit design satisfies the allocated software requirements and is suitable for the implementation;

b) to verify that the defined safety measures resulting from safety-oriented analyses in accordance with 7.4.10 and 7.4.11 are properly implemented;

c) to provide evidence that the implemented software unit complies with the unit design and fulfils the allocated software requirements with the required ASIL; and

d) to provide sufficient evidence that the software unit contains neither undesired functionalities nor undesired properties regarding functional safety.

9.2 General

The software unit design and the implemented software units are verified by using an appropriate combination of verification measures such as reviews, analyses and testing.

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

9.3 Inputs to this clause

9.3.1 Prerequisites

The following information shall be available:

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

— software architectural design specification in accordance with 7.5.1;

— software unit design specification in accordance with 8.5.1;

— software unit implementation in accordance with 8.5.2;

— configuration data in accordance with C.5.3, if applicable;

— calibration data in accordance with C.5.4, if applicable;

— safety analysis report in accordance with 7.5.2; and

— documentation of the software development environment in accordance with 5.5.1.

9.3.2 Further supporting information

The following information can be considered:

— None.

9.4 Requirements and recommendations

9.4.1 The requirements of this sub-clause shall be complied with if the software unit is a safety-related element.

NOTE 1 "Safety-related element" according to ISO 26262-1 means that the unit implements safety requirements, or that the criteria for coexistence (see ISO 26262-9:2018, Clause 6) of the unit with other units are not satisfied.

NOTE 2 The requirements of this clause address safety-related software units; other software standards (see ISO 26262-2:2018, 5.4.5.1) can apply for verification of other software units.

NOTE 3 For model-based software development, the corresponding parts of the implementation model also represent objects for the verification planning. Depending on the selected software development process the verification objects can be the code derived from this model, the model itself, or both.

9.4.2 The software unit design and the implemented software unit shall be verified in accordance with ISO 26262-8:2018, Clause 9 by applying an appropriate combination of methods according to Table 7 to provide evidence for:

a) compliance with the requirements regarding the unit design and implementation in accordance with Clause 8;

NOTE 1 Software safety requirements include both functions and properties of the software.

NOTE 2 Performing verification at the model level can substitute for verification at the source code level provided that the code generation preserves the properties (e.g. sufficient confidence in the code generators used).

EXAMPLE Evidence for the effective implementation of the error detection and error handling mechanisms specified to achieve robustness of the software unit against erroneous inputs.

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

NOTE 3 In the case of model-based development, requirement b) still applies.

c) compliance with the specification of the hardware-software interface (in accordance with 6.4.4), if applicable;

d) confidence in the absence of unintended functionality and properties;

e) sufficient resources to support their functionality and properties; and

f) implementation of the safety measures resulting from the safety-oriented analyses in accordance with 7.4.10 and 7.4.11.

Table 7 — Methods for software unit verification

Methods		ASIL			
		A	B	C	D
1a	Walk-through[a]	++	+	o	o
1b	Pair-programming[a]	+	+	+	+
1c	Inspection[a]	+	++	++	++
1d	Semi-formal verification	+	+	++	++
1e	Formal verification	o	o	+	+
1f	Control flow analysis[b, c]	+	+	++	++
1g	Data flow analysis[b, c]	+	+	++	++
1h	Static code analysis[d]	++	++	++	++
1i	Static analyses based on abstract interpretation[e]	+	+	+	+
1j	Requirements-based test[f]	++	++	++	++
1k	Interface test[g]	++	++	++	++

[a] For model-based development these methods are applied at the model level, if evidence is available that justifies confidence in the code generator used.

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

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

[d] Static analyses are a collective term which includes analysis such as searching the source code text or the model for patterns matching known faults or compliance with modelling or coding guidelines.

[e] Static analyses based on abstract interpretation are a collective term for extended static analysis which includes analysis such as extending the compiler parse tree by adding semantic information which can be checked against violation of defined rules (e.g. data-type problems, uninitialized variables), control-flow graph generation and data-flow analysis (e.g. to capture faults related to race conditions and deadlocks, pointer misuses) or even meta compilation and abstract code or model interpretation.

[f] The software requirements at the unit level are the basis for this requirements-based test. These include the software unit design specification and the software safety requirements allocated to the software unit.

[g] This method can be used to provide evidence for the integrity of used and exchanged data.

[h] In the context of software unit testing, fault injection test means to modify the tested software unit (e.g. introduce faults into the software) for the purposes described in 9.4.2. Such modifications include injection of arbitrary faults (e.g. by corrupting values of variables, by introducing code mutations, or by corrupting values of CPU registers).

[i] Some aspects of the resource usage evaluation can only be performed properly when the software unit tests are executed on the target environment or if the emulator for the target processor adequately supports resource usage tests.

[j] 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.

EXAMPLE In the case of model-based design results of non-floating-point operations can be compared.

Table 7 *(continued)*

Methods		ASIL			
		A	B	C	D
1l	Fault injection test[h]	+	+	+	++
1m	Resource usage evaluation[i]	+	+	+	++
1n	Back-to-back comparison test between model and code, if applicable[j]	+	+	++	++

[a] For model-based development these methods are applied at the model level, if evidence is available that justifies confidence in the code generator used.

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

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

[d] Static analyses are a collective term which includes analysis such as searching the source code text or the model for patterns matching known faults or compliance with modelling or coding guidelines.

[e] Static analyses based on abstract interpretation are a collective term for extended static analysis which includes analysis such as extending the compiler parse tree by adding semantic information which can be checked against violation of defined rules (e.g. data-type problems, uninitialized variables), control-flow graph generation and data-flow analysis (e.g. to capture faults related to race conditions and deadlocks, pointer misuses) or even meta compilation and abstract code or model interpretation.

[f] The software requirements at the unit level are the basis for this requirements-based test. These include the software unit design specification and the software safety requirements allocated to the software unit.

[g] This method can be used to provide evidence for the integrity of used and exchanged data.

[h] In the context of software unit testing, fault injection test means to modify the tested software unit (e.g. introduce faults into the software) for the purposes described in 9.4.2. Such modifications include injection of arbitrary faults (e.g. by corrupting values of variables, by introducing code mutations, or by corrupting values of CPU registers).

[i] Some aspects of the resource usage evaluation can only be performed properly when the software unit tests are executed on the target environment or if the emulator for the target processor adequately supports resource usage tests.

[j] 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.

EXAMPLE In the case of model-based design results of non-floating-point operations can be compared.

9.4.3 To enable the specification of appropriate test cases for the software unit testing in accordance with 9.4.2, test cases shall be derived using the methods as listed in Table 8.

Table 8 — 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 based on knowledge or experience[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.

9.4.4 To evaluate the completeness of verification and to provide evidence that the objectives for unit testing are adequately achieved, the coverage of requirements at the software unit level shall be determined and the structural coverage shall be measured in accordance with the metrics as listed in Table 9. If the achieved structural coverage is considered insufficient, either additional test cases shall be specified or a rationale based on other methods shall be provided.

NOTE 1 No target value or a low target value for structural coverage without a rationale is considered insufficient.

EXAMPLE 1 Analysis of structural coverage can reveal shortcomings in requirements-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).

EXAMPLE 3 A rationale can be based on the state of the art.

Table 9 — 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 2 The structural coverage can be determined by the use of appropriate software tools.

NOTE 3 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.

EXAMPLE 4 The analysis of structural coverage performed at the model level can replace the source code coverage metrics if it is shown to be equivalent, with rationales based on evidence that the coverage is representative of the code level.

NOTE 4 If instrumented code is used to determine the degree of structural coverage, it can be necessary to provide evidence that the instrumentation has no effect on the test results. This can be done by repeating representative test cases with non-instrumented code.

9.4.5 The test environment for software unit testing shall be suitable for achieving the objectives of the unit testing considering the target environment. If the software unit testing is not carried out in the target environment, the differences in the source and object code, as well as 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; or

— 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 specification resulting from requirements 9.4.2 to 9.4.5.

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

10 Software integration and verification

10.1 Objectives

The objectives of this sub-phase are:

a) to define the integration steps and integrate the software elements until the embedded software is fully integrated;

b) to verify that the defined safety measures resulting from safety analyses at the software architectural level are properly implemented;

c) to provide evidence that the integrated software units and software components fulfil their requirements according to the software architectural design; and

d) to provide sufficient evidence that the integrated software contains neither undesired functionalities nor undesired properties regarding functional safety.

10.2 General

In this sub-phase, the particular integration levels and the interfaces between the software elements are verified according to the software architectural design. The steps of the integration and verification of the software elements are related 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 (HSI) specification (refined) in accordance with 6.5.2;

— software architectural design specification in accordance with 7.5.1;

— safety analysis report in accordance with 7.5.2;

— dependent failures analysis report in accordance with 7.5.3, if applicable;

— software unit implementation in accordance with 8.5.2;

— configuration data in accordance with C.5.3, if applicable;

— calibration data in accordance with C.5.4, if applicable;

— documentation of the software development environment in accordance with 5.5.1; and

— software verification specification in accordance with 9.5.1.

10.3.2 Further supporting information

The following information can be considered:

— qualified software components available (see ISO 26262-8:2018, Clause 12).

10.4 Requirements and recommendations

10.4.1 The software integration approach shall be defined and 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 dependencies with the achievement of the verification objectives provided in 10.4.2;

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

c) 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 (see Annex B).

10.4.2 The software integration shall be verified in accordance with ISO 26262-8:2018, Clause 9 by an appropriate combination of methods according to Table 10 to provide evidence that the hierarchically integrated software units, the software components and the integrated embedded software achieve:

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

b) compliance with the hardware-software interface specification in accordance with 6.5.2;

c) the specified functionality;

d) the specified properties;

EXAMPLE Reliability due to absence of inaccessible software, robustness against erroneous inputs, dependability due to effective error detection and handling.

e) sufficient resources to support the functionality; and

f) effectiveness of the safety measures resulting from the safety-oriented analysis in accordance with 7.4.10 and 7.4.11, if applicable.

NOTE 1 The safety measures can include software partitioning.

Table 10 — Methods for verification of software integration

Methods		ASIL			
		A	B	C	D
1a	Requirements-based test[a]	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[b]	+	+	++	++
1d	Resource usage evaluation[c, d]	++	++	++	++
1e	Back-to-back comparison test between model and code, if applicable[e]	+	+	++	++
1f	Verification of the control flow and data flow	+	+	++	++
1g	Static code analysis[f]	++	++	++	++
1h	Static analyses based on abstract interpretation[g]	+	+	+	+

[a] The software requirements allocated to the architectural elements are the basis for this requirements-based test.

[b] In the context of software integration testing, fault injection test means to introduce faults into the software for the purposes described in 10.4.3 and in particular to test the correctness of hardware-software interface related to safety mechanisms. This includes injection of arbitrary faults in order to test safety mechanisms (e.g. by corrupting software interfaces). Fault injection can also be used to verify freedom from interference.

[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 evaluation can only be performed properly when the software integration tests are executed on the target environment or if the emulator for the target processor adequately 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.

[f] Static analyses are a collective term which includes analysis such as architectural analyses, analyses of resource consumption and searching the source code text or the model for patterns matching known faults or compliance with modelling or coding guidelines, if not already verified at the unit level.

[g] Static analyses based on abstract interpretation are a collective term for extended static analysis which also includes analysis such as extending the compiler parse tree by adding semantic information which can be checked against violation of defined rules (e.g. data-type problems, uninitialized variables), control-flow graph generation and data-flow analysis (e.g. to capture faults related to race conditions and deadlocks, pointer misuses) or even meta compilation and abstract code or model interpretation, if not already verified at the unit level.

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

10.4.3 To enable the specification of appropriate test cases for the software integration test methods selected in accordance with 10.4.2, test cases shall be derived using the methods as listed in Table 11.

Table 11 — 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 based on knowledge or experience[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.

10.4.4 To evaluate the completeness of verification and to provide evidence that the test objectives for integration testing are adequately achieved, 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 based on other methods shall be provided.

10.4.5 This sub-clause applies to ASIL (A), (B), C and D, in accordance with 4.4: To evaluate the completeness of test cases and to provide evidence that the test objectives for integration testing are adequately achieved, the structural coverage shall be evaluated in accordance with the methods as listed in Table 12. If the achieved structural coverage is considered insufficient, either additional test cases shall be specified or a rationale based on other methods 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 12 — Structural coverage 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 sub-programs or functions in the software (for definition see IEC 61508-7:2010, C.5.8).

[b] Method 1b refers to the percentage of executed software sub-programs or function with respect to each implemented call of these sub-programs or functions in the software.

NOTE The evidence can be provided by implementing an appropriate software integration and test strategy.

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

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

10.4.6 It shall be verified that the embedded software that is to be included as part of a production release in accordance with ISO 26262-2:2018, 6.4.13 contains all the specified functions and properties 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 26262-8:2018, Clause 8).

10.4.7 The test environment for software integration testing shall be suitable for achieving the objectives of the integration testing considering 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; or

— hardware-in-the-loop tests.

10.5 Work products

10.5.1 Software verification specification (refined) resulting from requirements 10.4.2 to 10.4.7.

10.5.2 Embedded software resulting from requirement 10.4.1.

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

11 Testing of the embedded software

11.1 Objective

The objectives of this sub-phase are to provide evidence that the embedded software:

a) fulfils the safety-related requirements when executed in the target environment; and

b) contains neither undesired functionalities nor undesired properties regarding functional safety.

11.2 General

The purpose of this activity is to provide evidence that the embedded software fulfils its requirements in the target environment. This evidence can be provided by using suitable results of other verification activities.

NOTE It can be good practice to perform the testing of the embedded software by a different person than the one who implemented the software.

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;

— software safety requirements specification in accordance with 6.5.1;

— embedded software in accordance with 10.5.2;

— calibration data in accordance with C.5.4, if applicable;

— documentation of the software development environment in accordance with 5.5.1; and

— software verification specification (refined) in accordance with 10.5.1.

11.3.2 Further supporting information

The following information can be considered:

— technical safety concept (see ISO 26262-4:2018, 6.5.2);

— system architectural design specification (see ISO 26262-4:2018, 6.5.3);

— integration and test strategy (see ISO 26262-4:2018, 7.5.1); and

— integration and test report (see ISO 26262-4:2018, 7.5.2).

11.4 Requirements and recommendations

11.4.1 To verify that the embedded software fulfils the software safety requirements in the target environment, tests shall be conducted in suitable test environments as listed in Table 13 and in accordance with ISO 26262-8:2018, Clause 9.

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

Table 13 — Test environments for conducting the software testing

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.

11.4.2 The testing of the embedded software shall be performed using methods as listed in Table 14 to provide evidence that the embedded software fulfils the software requirements as required by their respective ASIL.

Table 14 — Methods for tests of the embedded software

Methods		ASIL			
		A	B	C	D
1a	Requirements-based test	++	++	++	++
1b	Fault injection test[a]	+	+	+	++

[a] In the context of software testing, fault injection test means to introduce faults into the software by means of e.g. corrupting calibration parameters.

11.4.3 To enable the specification of appropriate test cases for the software testing in accordance with 11.4.2, test cases shall be derived using the methods as listed in Table 15.

Table 15 — Methods for deriving test cases for the test of the embedded software

Methods		ASIL			
		A	B	C	D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes	+	++	++	++
1c	Analysis of boundary values	+	+	++	++
1d	Error guessing based on knowledge or experience	+	+	++	++
1e	Analysis of functional dependencies	+	+	++	++
1f	Analysis of operational use cases[a]	+	++	++	++

[a] Examples for operational use cases for software can include software update in the field, starting the nominal application only if the integrity of the software is ensured by bootloader, safety-related behaviour of the embedded software in different operational modes such as start-up, diagnosis, degraded, power-down (going to sleep), power-up (waking up), calibration, functions for mode synchronization between different ECUs or end-of-line-specific test bench mode for safeguarding production personnel.

11.4.4 The results of the testing of the embedded software shall be evaluated with regard to:

a) compliance with the expected results; and

b) coverage of the software safety requirements.

NOTE This includes the coverage of the configuration and calibration ranges. See Annex C.

11.5 Work products

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

11.5.2 Software verification report (refined) resulting from requirements 11.4.1 to 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 General topics for the product development at the software level	The objectives of this Clause are: a) to ensure a suitable and consistent software development process; and b) to ensure a suitable software development environment.	(none)	5.5.1 Documentation of the software development environment
6 Specification of software safety requirements	The objectives of this sub-phase are: a) to specify or refine the software safety requirements which are derived from the technical safety concept and the system architectural design specification; b) to define the safety-related functionalities and properties of the software required for the implementation; c) to refine the requirements of the hardware-software interface initiated in ISO 26262-4:2018, Clause 6; and d) to verify that the software safety requirements and the hardware-software interface requirements are suitable for software development and are consistent with the technical safety concept and the system architectural design specification	Technical safety requirements specification (see ISO 26262-4:2018, 6.5.1) Technical safety concept (see ISO 26262-4:2018, 6.5.2) System architectural design specification (see ISO 26262-4:2018, 6.5.3) Hardware-software interface (HSI) specification (see ISO 26262-4:2018, 6.5.4) Documentation of the software development environment (see 5.5.1)	6.5.1 Software safety requirements specification 6.5.2 Hardware-software interface (HSI) specification (refined) 6.5.3 Software verification report

Table A.1 *(continued)*

Clause	Objectives	Prerequisites	Work products
7 Software architectural design	The objectives of this sub-phase are: a) to develop a software architectural design that satisfies the software safety requirements and the other software requirements; b) to verify that the software architectural design is suitable to satisfy the software safety requirements with the required ASIL; and c) to support the implementation and verification of the software.	Documentation of the software development environment (see 5.5.1) Hardware-software interface (HSI) specification (refined) (see 6.5.2) Software safety requirements specification (see 6.5.1)	7.5.1 Software architectural design specification 7.5.2 Safety analysis report 7.5.3 Dependent failures analysis report 7.5.4 Software verification report
8 Software unit design and implementation	The objectives of this sub-phase are: a) to develop a software unit design in accordance with the software architectural design, the design criteria and the associated software requirements which supports the implementation and verification of the software unit; and b) to implement the software units as specified.	Documentation of the software development environment (see 5.5.1) Hardware-software interface (HSI) specification (refined) (see 6.5.2) Software architectural design specification (see 7.5.1) Software safety requirements specification (see 6.5.1) Configuration data (see C.5.3), if applicable Calibration data (see C.5.4), if applicable	8.5.1 Software unit design specification 8.5.2 Software unit implementation

Table A.1 *(continued)*

Clause	Objectives	Prerequisites	Work products
9 Software unit verification	The objectives of this sub-phase are: a) to provide evidence that the software unit design satisfies the allocated software requirements and is suitable for the implementation; b) to verify that the defined safety measures resulting from safety analyses in accordance with 7.4.10 and 7.4.11 are properly implemented; c) to provide evidence that the implemented software unit complies with the unit design and fulfils the allocated software requirements with the required ASIL; and d) to provide sufficient evidence that the software unit contains neither undesired functionalities nor undesired properties regarding functional safety.	Hardware-software interface (HSI) specification (refined) (see 6.5.2) Software architectural design specification (see 7.5.1) Software unit design specification (see 8.5.1) Software unit implementation (see 8.5.2) Configuration data (see C.5.3), if applicable Calibration data (see C.5.4), if applicable Safety analysis report (see 7.5.2) Documentation of the software development environment (see 5.5.1)	9.5.1 Software verification specification 9.5.2 Software verification report (refined)

Table A.1 *(continued)*

Clause	Objectives	Prerequisites	Work products
10 Software integration and verification	The objectives of this sub-phase are: a) to define the integration steps and integrate the software elements until the embedded software is fully integrated; b) to verify that the defined safety measures resulting from safety analyses at the software architectural level are properly implemented; c) to provide evidence that the integrated software units and software components fulfil their requirements according to the software architectural design; and d) to provide sufficient evidence that the integrated software contains neither undesired functionalities nor undesired properties regarding functional safety.	Hardware-software interface (HSI) specification (refined) (6.5.2) Software architectural design specification (see 7.5.1) Safety analysis report (see 7.5.2) Dependent failures analysis report (see 7.5.3), if applicable Software unit implementation (see 8.5.2) Configuration data (see C.5.3), if applicable Calibration data (see C.5.4), if applicable Documentation of the development environment (see 5.5.1) Software verification specification (see 9.5.1)	10.5.1 Software verification specification (refined) 10.5.2 Embedded software 10.5.3 Software verification report (refined)
11 Testing of the embedded software	The objectives of this sub-phase are to provide evidence that the embedded software: a) fulfils the software safety requirements when executed in the target environment; and b) contains neither undesired functionalities nor undesired properties regarding functional safety.	Software architectural design specification (see 7.5.1) Software safety requirements specification (see 6.5.1) Embedded software (see 10.5.2) Calibration data (see C.5.4), if applicable Documentation of the software development environment (see 5.5.1) Software verification specification (refined) (see 10.5.1)	11.5.1 Software verification specification (refined) 11.5.2 Software verification report (refined)

Table A.1 *(continued)*

Clause	Objectives	Prerequisites	Work products
Annex C Software configuration	The objectives of software configuration are: a) to enable controlled changes in the behaviour of the software for different applications; b) to provide evidence that the configuration data and the calibration data fulfil the requirements with the required ASIL; and c) to provide evidence that the application-specific embedded software and its calibration data are suitable for release for production.	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 Configuration data C.5.4 Calibration data C.5.6 Verification specification (refined) C.5.7 Verification report (refined) C.5.8 Software architectural design specification (refined) C.5.9 Documentation of the software development environment (refined)

Annex B
(informative)

Model-based development approaches

B.1 Objectives

This Annex explains possible usage benefits and potential issues of model-based development approaches (MBD) during development at the software level.

NOTE This Annex does not imply that the model-based development approaches mentioned are restricted to software development only.

B.2 General

This sub-clause provides general information that is common to various use cases of MBD (e.g. requirements, design, verification/testing). Sub-clause B.3 provides specific considerations based on exemplary use cases.

B.2.1 Introduction

Models may be used to represent information or views of information required during the development phase as well as for the abstract design and implementation of one or more software elements or the software's environment. Models themselves may consist of various hierarchical refinement levels or references to refined models at a lower hierarchical level (e.g. hierarchical structure with black box and white box views for each model element). A model is developed using a modelling notation.

Modelling notations may be graphical and/or textual. They may be formal (e.g. notation with an underlying mathematical definition) or semi-formal (e.g. structured notation with incompletely defined semantics). Modelling notations may be international standards (e.g. UML®) or company-specific. They are typically based on concepts such as classes, block diagrams, control diagrams, or state charts (expressing states and transition between states). In this Annex, only models and modelling notations are considered which incorporate a defined syntax and semantics adequate for their intended use (i.e. illustrative figures without such definitions are not considered as models). Beside the specific reasons for using MBD (e.g. simulation or code generation), an adequately defined syntax and semantics is a basis for the achievement of criteria such as comprehensibility, unambiguity, correctness, consistency and verifiability of information or work products described by models especially when different parties are collaborating.

In addition to the modelling notation itself, modelling guidelines and/or suitable tools are means to achieve adequately defined syntax, semantics and compliance with modelling guidelines.

EXAMPLE As part of guidelines, naming conventions or style guides support the achievement of criteria such as "comprehensibility" while the selection of language subsets excluding ambiguous constructs supports both "comprehensibility" and the correct transformation and execution of the model. In case of simulation or transformation of models, suitable simulation engines or code generators can supplement the required semantics such as a deterministic order of execution or transformation sufficient knowledge on the detailed behaviour of the simulation engines or code generators provided.

In this Annex the following use cases of model-based development approaches are considered further:

— Specification of software safety requirements (Clause 6 and ISO 26262-8:2018, Clause 6);

— Development of the software architectural design (Clause 7);

— Design and implementation of software units, with or without automated code generation (Clause 8);

— Design, implementation and integration of software components, including automated code generation from software component models (Clauses 8, 9 and 10); and

— Verification (static and/or dynamic) (Clauses 9, 10 and 11).

B.2.2 General suitability aspects

The suitability of the selected model-based development approach for an intended application, including the modelling techniques, notations, languages or tools, can be evaluated based on aspects such as:

— achievement of criteria provided by the ISO 26262 series of standards for the corresponding development phase (e.g. comprehensibility, completeness, verifiability, unambiguity, accuracy) with the selected approach(es);

— knowledge and experiences with the selected approach(es);

— adequacy of the definition of syntax and semantics;

— adequacy for the application domain (e.g. real-time behaviour, data structures, generation of software for a fixed-point versus floating-point microcontroller);

— role of the model in the software lifecycle (e.g. development of safety requirements, expressing static or dynamic aspects of architectural designs);

— support for managing complexity by enforcing the principles of abstraction, encapsulation, hierarchy and modularity;

— collaboration model between internal/external development partners (e.g. consistency of data during data exchange);

— required versus available confidence in software tools (see ISO 26262-8:2018, Clause 11); or

— qualification of software components (e.g. software libraries) used as part of the selected MBD approach or included into the MBD environment (see ISO 26262-8:2018, Clause 12).

B.2.3 Potential impact of MBD on the software lifecycle

A model may represent more than one work product of this document (e.g. requirements, architectural design, detailed design and model-based software integration with code generation). In comparison to a traditional development process where lifecycle data are separated, a stronger coalescence of the phases “Software safety requirements”, “Software architectural design”, “Software unit design and implementation” etc. may occur. The potential benefits of this approach (e.g. continuity, information sharing across the software life cycle, consistency) are appealing, but this approach may also introduce issues causing systematic faults (see B.3).

B.3 Specific considerations for the example software development use cases

This section provides considerations related to benefits and potential issues based on example use cases.

B.3.1 Specification of software safety requirements (Clause 6 and ISO 26262-8:2018, Clause 6)

Models can be used to capture the functionality, behaviour, design or properties of the software to be realized at safety requirements level (e.g. open/closed loop controls, controlled systems, states and transitions, monitoring functions or independence properties) and to enable verification and validation of requirements by simulation or formal methods.

Models can be suitable means to achieve the characteristics of safety requirements and are a means to satisfy requirements to use semi-formal or formal notations as stated in ISO 26262-8:2018, Clause 6.

EXAMPLE When specifying a state machine using a model based approach, the corresponding model captures the required states and transitions in an intuitive, unambiguous and comprehensible way. The fact that such a model represents more than one atomic requirement compared with a textual specification of the same state machine is acceptable. An adequate rationale would be that for this purpose completeness, verifiability and internal consistency of the specification can be achieved and maintained better by the model without duplication of information.

However be aware that by using requirement models it is possible to miss safety requirements that cannot be expressed by the selected model-based development approach. For this reason, ISO 26262-8:2018, Clause 6 emphasizes an appropriate combination of textual requirements and other notations.

EXAMPLE Requirements related to the robustness of the software functionality against missing or erroneous inputs

B.3.2 Development of the software architectural design (Clause 7)

Models can be used to capture the software architecture according to several perspectives, in particular:

a) static aspects (e.g. components, their interfaces and data flows, component attributes); and

b) dynamic aspects (e.g. functional chain of events, scheduling, message passing).

Models together with the modelling guidelines can be a suitable means to achieve the properties of and principles for safety-related architectural designs and a means to satisfy requirements for semi-formal or formal notations as stated in Clause 7.

However, be aware that one modelling approach might not express all of the required aspects (e.g. models targeting the implementation of software functions often do not describe the overall architectural design including elements of the basic software or operating system) and that a model may not be fully understandable without further textual explanations (e.g. by different collaborating parties).

B.3.3 Design and implementation of software units (Clauses 8 and 9)

Models, together with the modelling and coding guidelines, can be used to develop the detailed design and the software unit itself at a higher level of abstraction. By using MBD, executable code can be generated directly from the model by code generation.

MBD enables automated consistency checks (e.g. data typing, correct definition of data before use) and verification by formal methods, simulation, static analyses or model-in-the-loop/software-in-the-loop testing of the software units. Depending on the confidence in the code generator (see ISO 26262-8:2018, Clause 11) automated code generation from the model may lower the risk of coding errors compared to manual coding.

However, be aware that, for example:

— Design aspects that cannot be expressed by the modelling notations might be missed if not documented in some other appropriate form. Even unit testing (e.g. using back-to-back-testing) might not reveal this if all tests are only derived from such an “incomplete” model.

— Safety-related code properties such as robustness might not be implemented if only the intended function is modelled. Code properties often are achieved by an appropriate combination of the model properties and code generation properties (e.g. if the division is generated “as is” by the code generator, then avoidance of division by zero will be checked at model level. If the code generator handles the division by zero with a robust design coding pattern, division by zero analysis in the model may be alleviated).

— Faults can be introduced by incorrect addressing or handling of numerical accuracy aspects (e.g. if the model is just inherited from control engineering). For instance issues resulting from different precision properties of the control engineer's computer environment and the target environment.

— Faults can be introduced when creating discrete models of the software behaviour from continuous models (e.g. effect of discretization of values and/or time, execution time of the calculation, handling concurrent tasks).

— Simulation or verification results (e.g. results of consistency checks) may be tool-dependent, because different simulation engines or model checkers may deviate regarding the implemented semantics (e.g. for algorithmic aspects not addressed by the semantics of the modelling language).

— The resource usage in case of auto-coding (execution time and memory consumption) may be different than in case of manual coding. In addition, the means for resource optimization may be more limited.

— In case of auto-coding, verifiability of the generated source code equivalence to manual code may be more difficult to achieve.

B.3.4 Design, implementation and integration of software components, including automated code generation from software component models (Clauses 8, 9 and 10)

Models, together with the modelling and coding guidelines, can be used to design, implement and integrate software at a higher level of abstraction. Thus, software components or even the embedded software can be implemented by code generation from the integrated model.

This approach implies a strong coalescence of the different software development phases, and, therefore, the benefits versus potential issues of such an approach need to be evaluated carefully (see hints provided above) including the implementation of appropriate safety measures (e.g. using separate test models for deriving additional test cases).

B.3.5 Verification (static and/or dynamic) (Clauses 9, 10 and 11)

Models can serve as a means for verifying work products (e.g. design model, code). Such a model is called a "verification model". Verification models can be used, for example, as the basis for back-to-back testing, generation of test cases or for expressing safety-related properties as a reference for formal verification (this requires that the model to verify is formal too).

Models can also serve as a means to enable or support verification activities (e.g. plant model needed for the hardware-in-the-loop testing of closed-loop controls by simulating the environment of the device under test).

The use of models for verification purposes may enable the early and efficient detection of faults/failures in work products (including software), more efficient test case generation, highly automated testing or even formal verification techniques.

However, be aware that, for example:

— Without evidence for the suitability of a model used for verification purposes, the validity of related results might be questionable (e.g. inadequate plant models may lead to incorrect test results).

— For valid results, the maturity of the verification model matches the target maturity of the verification object.

— Test cases generated from the same model which is also used for code generation cannot serve as the only source for verifying both the model itself and the code generated from it.

Annex C
(normative)

Software configuration

C.1 Objectives

The objectives of software configuration are:

a) to enable controlled changes in the behaviour of the software for different applications;

b) to provide evidence that the configuration data and the calibration data fulfil the requirements with the required ASIL; and

c) to provide evidence that the application-specific embedded software and its calibration data are suitable for release for production.

C.2 General

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

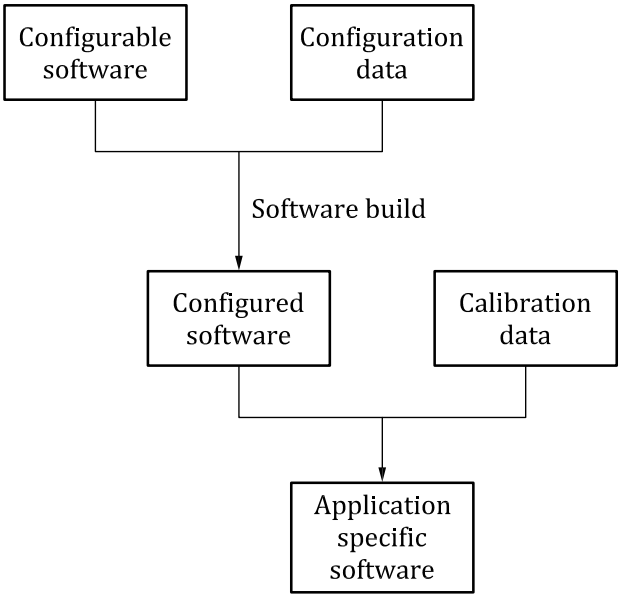


Figure C.1 — Creating application specific software

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 The configuration data and their specification shall be verified in accordance with ISO 26262-8:2018, Clause 9 to provide evidence that:

a) the configuration data complies with the software architectural design specification in accordance with 7.5.1 and with the software unit design specification in accordance with 8.5.1;

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

c) 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 Clause 9, Clause 10, Clause 11 and ISO 26262-4:2018, Clause 7).

C.4.3 The ASIL of the configuration data shall equal the highest ASIL of the configurable software that it is applied to.

C.4.4 Configurable software shall be verified in accordance with ISO 26262-8:2018, Clause 9 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).

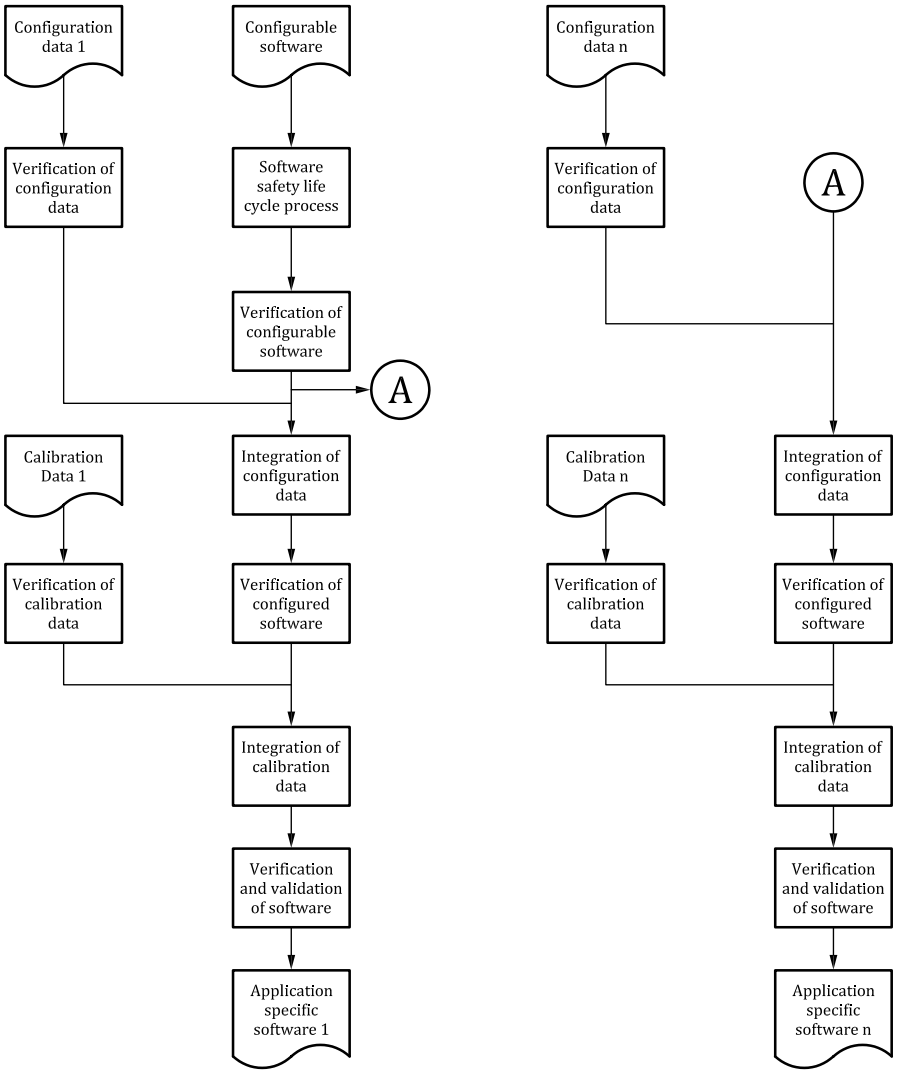


Figure C.2 — Variants of the reference phase model for software development with configurable software and different configuration data and different calibration data

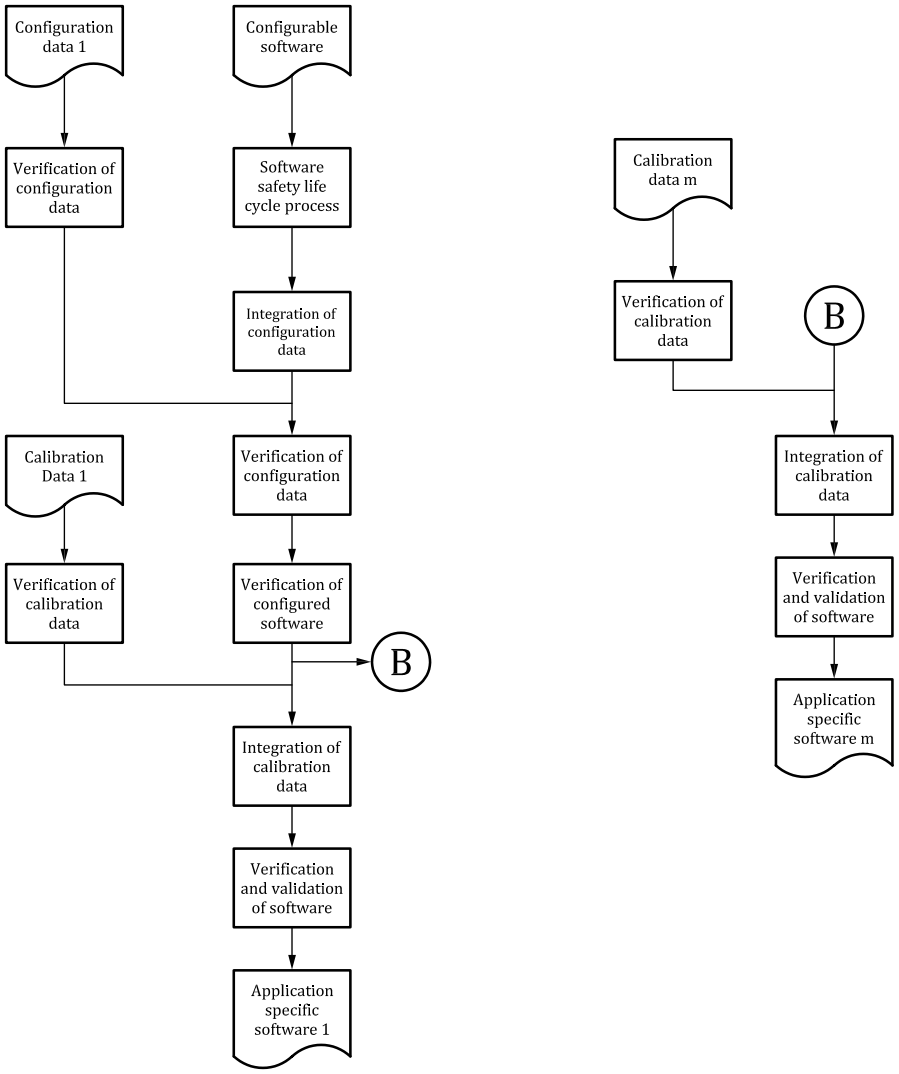


Figure C.3 — Variants of the reference phase model for software development with configurable software with the same configuration data and different calibration data

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 1 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 2 Configuration data can have an impact on the configured software that uses the calibration data.

C.4.7 The ASIL of the calibration data shall equal the highest ASIL of the software safety requirements it can violate.

C.4.8 The specification of the calibration data shall be verified in accordance with ISO 26262-8:2018, Clause 9 to provide evidence that:

a) the specified calibration data are suitable and comply with the software safety requirements in accordance with 6.5.1;

b) the specified calibration data are compliant with the software architectural design specification in accordance with 7.5.1 and with the software unit design specification in accordance with 8.5.1; and

c) the specified calibration data are consistent and compatible with the specification of other calibration data to prevent unintended impact.

C.4.9 The calibration data which are released for production shall be verified in accordance with ISO 26262-8:2018, Clause 9 to provide evidence that:

a) the released calibration data comply with their specification (see C.4.6); and

b) the calibrated, application-specific variant of the embedded software provides the specified safety-related functionalities and properties.

NOTE Verification of calibration data can also be performed at the system level.

C.4.10 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

Mechanisms		ASIL			
		A	B	C	D
1a	Plausibility checks on calibration data	++	++	++	++
1b	Redundant storage and comparison of calibration data	+	+	+	++
1c	Calibration data checks using error detecting codes[a]	+	+	+	++
[a] Error detecting codes may also be implemented in the hardware in accordance with ISO 26262-5.					

C.4.11 The generation and application of calibration data shall be specified regarding:

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 to C.4.4.

C.5.2 **Calibration data specification** resulting from requirement C.4.6 to C.4.8.

C.5.3 **Configuration data** resulting from requirement C.4.2 to C.4.4.

C.5.4 Calibration data resulting from requirement C.4.7 to C.4.10.

C.5.5 Verification specification (refined) resulting from requirements C.4.2, C.4.4, C.4.5, C.4.7, C.4.8, C.4.10 and C.4.11.

C.5.6 Verification report (refined) resulting from requirements C.4.2, C.4.4, C.4.5, C.4.7, C.4.8, C.4.10 and C.4.11.

C.5.7 Software architectural design specification (refined) resulting from requirement C.4.10.

C.5.8 Documentation of the software development environment (refined) resulting from requirements C.4.1 to C.4.11.

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; or

— 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;

— inconsistent data (e.g. due to update during data fetch);

— stack overflow or underflow; or

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

EXAMPLE 1 Safety measures 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.

EXAMPLE 2 Appropriate verification methods can be considered as detailed safety analysis to identify critical memories that protection mechanisms are used for. Results of static and semantic code analysis, control flow and data flow analysis can provide evidence for demonstrating freedom from interference.

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; or

— 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 buses, 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 buses, mini-slotting 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.

EXAMPLE 4 Appropriate verification methods can be considered as detailed safety analysis to identify critical data exchange that protection mechanisms are used for. Results of static and semantic code analysis, control flow and data flow analysis can provide evidence for demonstrating freedom from interference.

Annex E
(informative)

Application of safety analyses and analyses of dependent failures at the software architectural level

E.1 Objectives

This Annex explains the possible application of safety analyses and dependent failure analyses at the software architectural level. The examples provided in this Annex are intended to support the understanding of these analyses and provide guidance on their application.

E.2 General

E.2.1 Scope and purpose of the analyses

The ability of the embedded software to provide the specified functions, behaviour and properties with the integrity as required by the assigned ASIL is examined or confirmed by applying safety analyses or dependent failure analyses at the software architectural level.

The suitability of embedded software to provide the specified functions and properties with the integrity required by the assigned ASIL is examined by:

— identifying possible design weaknesses, conditions, faults or failures that could induce causal chains leading to the violation of safety requirements (e.g. using an inductive or deductive method); and

— analysing the consequences of possible faults, failures or causal chains on the functions and properties required for the software architectural elements.

The achieved level of independence or freedom from interference between the relevant software architectural elements is examined by analysing the software architectural design with respect to dependent failures, i.e.:

— possible single events, faults or failures that may cause a malfunctioning behaviour of more than one of the software elements which require independence from each other (e.g. cascading and/or common cause failures including common mode failures); and

— possible single events, faults or failures that may propagate from one software element to another inducing causal chains leading to the violation of safety requirements (e.g. cascading failures).

Achievement of independence or freedom from interference between the software architectural elements can be required because of:

— the application of an ASIL decomposition at the software level (see 6.4.3);

— the implementation of software safety requirements (see 7.4.11), for example, to provide evidence for effectiveness of the software safety mechanisms such as independence between the monitored element and the monitor; or

— required coexistence of the software architectural elements (see 7.4.6, 7.4.8 and 7.4.9 and ISO 26262-9:2018, Clause 6).

The role of safety analyses and analyses of dependent failures at software architectural level supports both design specification and design verification activities.

Such analysis may also reveal incompleteness or inconsistencies with respect to given safety requirements.

The results of safety analyses and analyses of dependent failures at software architectural level is also a basis for

— the specification and implementation of effective safety mechanisms in the product; or

— the determination of appropriate development-time safety measures, which are suitable either to prevent or to detect and control the relevant faults or failures identified during these analyses.

Figure E.1 illustrates the role of the safety-oriented analyses at software architectural level during software development and the basic relationships or interactions between software safety requirements, software architectural design and the safety plan.

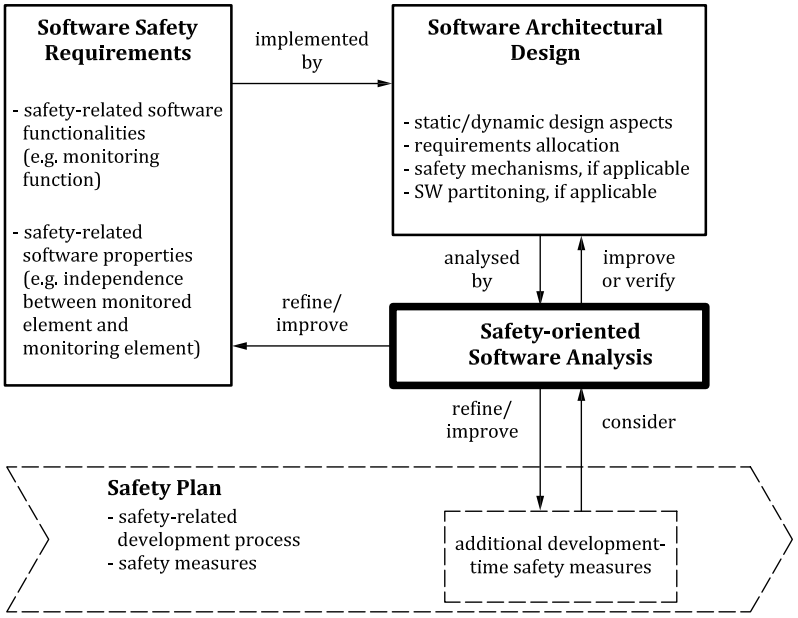
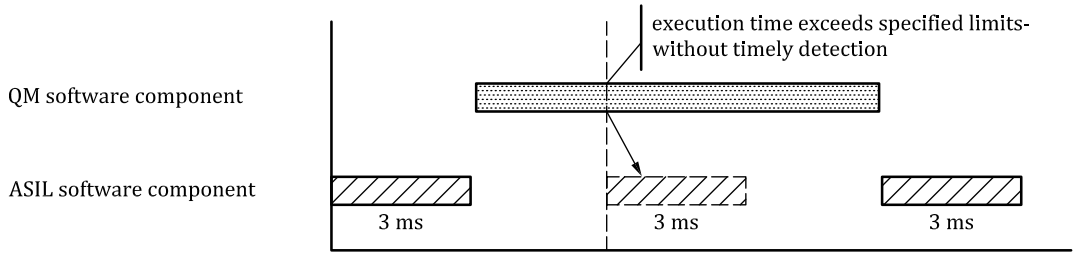


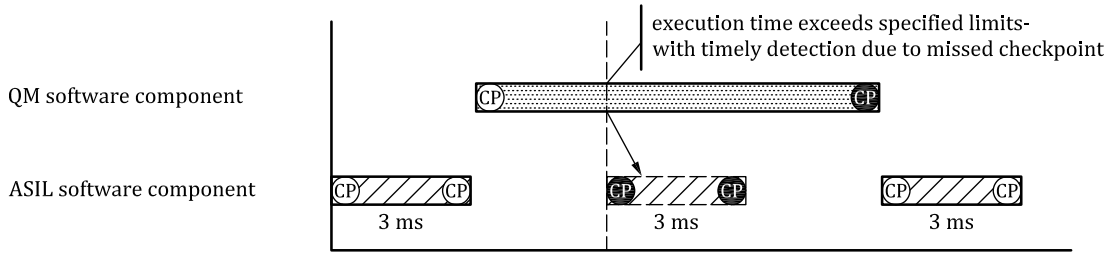
Figure E.1 — Illustration of the role of safety-oriented software analyses during development

EXAMPLE Figure E.2 illustrates interference caused by a conflicting use of a shared resource (e.g. a shared processing element). As shown in Figure E.2, a QM software element interferes and prevents the timely execution of the ASIL software element. Such interference could also occur between software components with different ASIL values. The figure illustrates the effects on the software execution without and with implementation of freedom from interference mechanisms. By introducing “check points” into software and implementing a timeout monitoring for the check points, the timing interference can be detected and a suitable reaction initiated.

Without freedom from interference mechanism:



With freedom from interference mechanism:



Legend

executed software component

blocked software component

interference

CP "passed" programme flow monitoring checkpoint

CP "failed" programme flow monitoring checkpoint

Figure E.2 — Example of timing interference leading to a cascading failure

Safety analyses and dependent failure analyses are intended to be applied at the software architecture level. Code level safety analyses (e.g. with respect to runtime errors such as divide by zero, access beyond array index boundary) are neither required nor considered necessary for the following reasons:

— The residual risk regarding these types of faults can be considered as sufficiently low provided that the software development approach as described in this document including the selection and application of suitable methods, approaches and development principles such as modelling and programming guidelines, design and implementation rules, model and/or code reviews, semantic or static analysis, unit and integration tests is applied carefully; and

— These types of faults in the implementation of the respective architectural elements usually induce the same malfunctioning behaviour of the element at its interfaces as analysed at architectural level. Therefore the analyses at the higher level already provide the evidence that the selected technical safety mechanisms are effective or that the development-time safety measures are sufficient to ensure an adequate argument.

E.2.2 Relationship between safety analyses and dependent failure analyses at the system, hardware and software level

Safety analyses and dependent failure analyses are applied during the multiple phases of the development lifecycle. These analyses can be related to each other.

In a separation of concern approach initiated at the system level, it is usually the responsibility of hardware development to ensure that the processing unit has a sufficiently low residual risk regarding hardware faults. Therefore, the software safety analysis and software dependent failure analysis at software architectural level can be performed without considering hardware faults.

However, under some hardware conditions (e.g. insufficient diagnostic coverage regarding random hardware faults of the processing element including transient faults), it may be appropriate to consider the negative influence of hardware faults. In such cases, specific analyses regarding the specific hardware/software interaction are performed considering for example:

— the specific software mapping to the different processing units of multi-core systems;

— the detailed design of the processing element on which the software is executed regarding the static or dynamic aspects of the software architectural design; or

— the achievable diagnostic coverage of the selected software safety mechanisms regarding software-relevant hardware faults.

NOTE Analysis of a specific hardware/software interaction can be supported by fault injection techniques (see also ISO 26262-11:2018, 4.8).

E.2.3 Safety analyses in a distributed software development (including SEooC development of software elements)

Embedded software and its software elements are often developed as a distributed development by more than one organization (including OEM, Tier 1, Tier 2, silicon vendor or software vendor) and even out of context (e.g. basic software, hardware-related software, COTS software or operating systems developed as an SEooC).

Important aspects for meaningful safety-oriented analyses in distributed software developments are:

— definition and agreement about the scope of the analyses and the procedures or methods used for performing the analyses either separately or jointly (including exchange of information/ documentation or confirmation of assumptions);

— definition and agreement regarding the overall responsibility for the software safety analyses process and common rules for the participating organizations (e.g. high level components, access rules between them);

— definition and agreement about the interfaces when using a modular approach for the analyses (e.g. agreed software fault models at the interfaces between the different scopes, the approach used or the exchanged information/documents); and

— definition and agreement about the verification of the analyses.

E.3 Topics for performing analyses

The software architectural design provides high level rules and decisions (e.g. functional safety classification of components, access rules between components) that supports the implementation and to argue the achievement of functional safety.

Common topics to achieve meaningful analyses are the determination of:

— the goals for the analyses – see E.3.1;

— the scope and boundaries – see E.3.2;

— the method applied to conduct the analyses – see E.3.3; and

— the means or criteria to support the meaningfulness, objectivity and rigour of the analyses – see E.3.4.

E.3.1 Determination of the goals for the analyses

The goals are provided by Clause 7 and ISO 26262-9:2018 Clauses 7 and 8.

EXAMPLE Violation of the safety-related functionalities or properties assigned to the software within the scope of the analyses such as robustness or deterministic execution.

E.3.2 Determination of the scope and boundaries regarding the specific architectural design to be analysed

Depending on the goals of the analyses, the scope is determined setting the focus on the relevant parts of the architectural design.

The scope of the analyses can be influenced by:

— the relevant scope of delivery and the respective responsibilities as defined in the DIA;

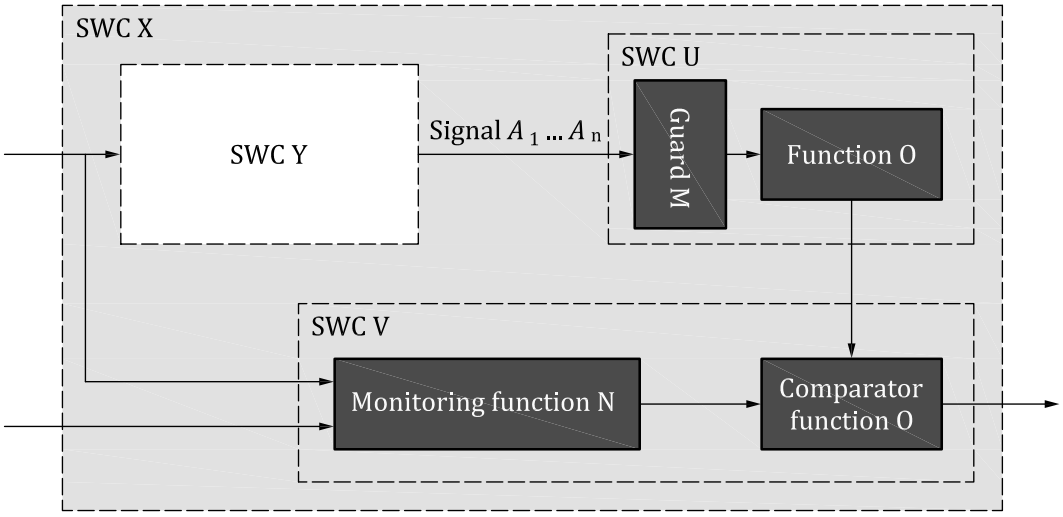
— the specific goals of the analyses;

— architectural strategies supported by “good” design principles (see 7.4.3); or

— properties required from the architectural design resulting from higher level safety concepts in respect of the achievement of freedom from interference or sufficient independence.

EXAMPLE 1 Appropriate end-to-end data protection mechanisms can be used as an argument that the basic software can be treated as a “black box” when considering the exchange of safety-related data with external senders or receivers during a safety analyses.

EXAMPLE 2 Dependencies of software functions with operating modes such as a safety-related function which is only “active” in a mode where the other “critical” function is safely deactivated by a safe mode switching function.



Legend

software component (SWC) which is in the scope of the analyses

software component (SWC) with allocated safety requirements

software component (SWC) which is excluded from the scope of the analyses based on an argument

functional interaction (e.g. data flow)

Figure E.3 — Scope of the analyses with respect to the safety argumentation

EXAMPLE 3 Figure E.3 shows software to be analysed where according to the safety concept the safety-related functionalities are implemented only in SWC U and controlled by a monitoring function implemented in SWC V. Determination of the scope of the analyses on the architecture clarifies the required safety arguments and supports the further steps shown in the following sections. In this example, SWC Y can be exempted from a further detailed analysis only if the analyses of SWC X, SWC V and SWC U are able to confirm the assumed freedom from interference (e.g. no weaknesses in the Guard M regarding cascading failures with respect to erroneous signals A_1 to A_n).

E.3.3 Determination of the method applied to conduct the analyses

Due to the specific nature of software (e.g. no random faults due to wear out or ageing and lack of a mature probabilistic method), methods established for such analyses at the system or hardware level often cannot be transferred to software without modifications, or will not deliver results that are as meaningful.

Common factors of safety analyses methods are that they usually consist of a means to enforce a structured approach, to store the gained knowledge, and to draw conclusions.

EXAMPLE Syntax and semantics to describe faults and their dependencies in a fault tree or the function net or list of elements or Risk Priority Number used in an FMEA.

The basis for the safety analyses and dependent failure analyses is the software architectural design describing the static aspects (e.g. expressed by a block diagram showing the functional elements and their interfaces/relationships) as well as the dynamic aspects (e.g. expressed by sequence, timing or scheduling diagrams or state charts) of the related software.

The safety analyses and dependent failure analyses according to the software architectural design can follow functional and/or processing chains considering both static and dynamic aspects. During such analyses, software fault models or questions such as “which specific fault or failure originating from or impacting this software component can lead to the violation of an assigned safety requirement” can be used to identify safety-related weaknesses in the design.

Refinements of software architectural design can support the safety analyses and dependent failure analyses for the adequate level of details in order to identify lower levels of faults or failures.

The level of detail required during such an analyses is related to:

— adequate selection of the scope during refinements of the architecture design;

— properties that support achievement of the specific goals of the analyses;

— an argument based on the architectural strategy supported by “good” design principles (see 7.4.3); or

— an argument resulting from sufficient independence including achieved freedom from interference.

E.3.4 Aspects to support the meaningfulness, objectivity and rigour of the analyses

E.3.4.1 Coupling factor classes for DFA in accordance with ISO 26262-9:2018

In ISO 26262–9:2018, Annex C, coupling factor classes with corresponding software-level examples are defined that can be used to improve the results of a dependent failure analysis performed on functional or processing chains of the software that are required to be independent.

E.3.4.2 Use content of Annex D

Annex D provides software fault models for consideration when analysing interference between software elements, for example with respect to:

— timing and execution;

— memory; or

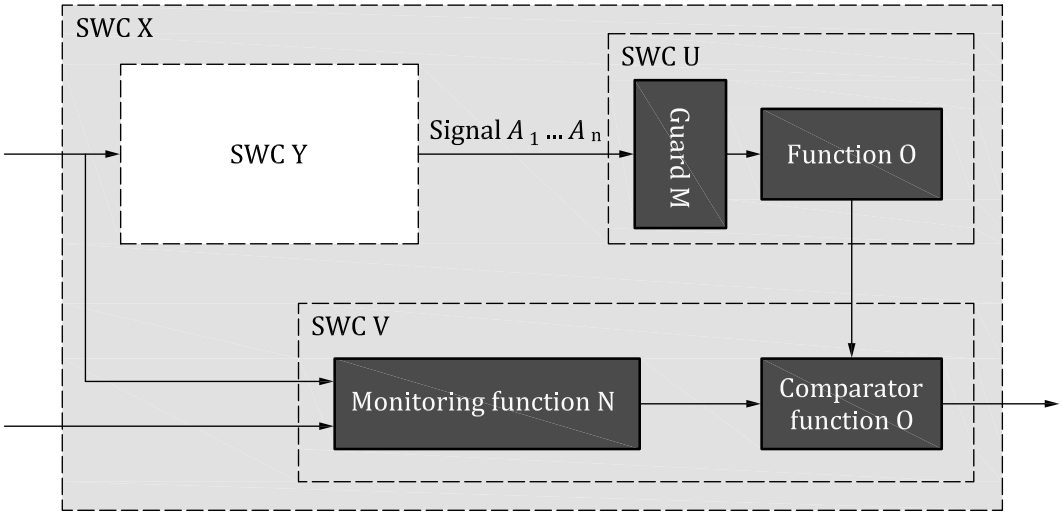
— exchange of information.

E.3.4.3 Analyses supported by using guide words

Guide words can be used to systematically examine the possible deviations from a specific design intent and to determine their possible consequences. Guide words are used to generate questions for the examination of a specific functionality or property of the architectural element during the safety-oriented analyses. When using guide words, the safety-oriented analyses of the specific functions or properties for each design element are repeated with each guide word, until the predetermined guide words have all been examined. Thus, guide words are a means to conduct such analyses systematically and to support the argument for completeness.

Data flow or similar diagrams are often used to describe the functional aspects of the software architectural design (e.g. functional or processing chains). Figure E.4 shows interaction of the software components Y, U and V and the related functional or processing chain (e.g. based on data flow).

Guide words can be used to identify weaknesses, faults, and failures. The selection of suitable guide words depends on the characteristics of the examined functions, behaviour, properties, interfaces or exchanged data.



Legend

software component (SWC) which is in the scope of the analyses

software component (SWC) with allocated safety requirements

software component (SWC) which is excluded from the scope of the analyses based on an argument

functional interaction (e.g. data flow)

Figure E.4 — Block diagram of software architectural design

Using the design intent as a basis, the guide words are derived by combining design attributes, the related guide words and their interpretation, in order to achieve a common understanding for the analyses. Table E.1 shows examples for the selection of guide words applied to the design in Figure E.4.

Table E.1 — Example for the selection of guide words related to software execution, scheduling or communication

Examined functionality or property	Examples for guide words	Interpretation	Additional references
signal flow A_1 to A_n	After / Late	Signal too late or out of sequence.	Annex D describes related considerations: D.2.4 Delay of information D.2.4 Incorrect sequence of information D.2.4 Blocking access to communication channel D.2.2 Blocking of execution D.2.2 Deadlocks D.2.2 Livelocks D.2.2 Incorrect allocation of execution time D.2.2 Incorrect synchronization between software components
	Before / Early	Signal too early or out of sequence.	Annex D describes related considerations: D.2.4 Incorrect sequence of information D.2.2 Incorrect synchronization between software components
	No	No signal	Annex D describes related considerations: D.2.2 Blocking of execution D.2.2 Deadlocks D.2.2 Livelocks D.2.2 Incorrect synchronization between software components
Signal value A_1 to A_n	More	Signal value exceeds permitted range	
	Less	Signal value falls bellows the permitted range	

During the analysis, guide words can be applied using a deductive or inductive approach. During the analyses, a further refinement of the guide words is possible if required (e.g. to better match specific design aspects).

During the analyses, the guide words are used according to the selected method (e.g. following function or processing chains) to generate the questions that will examine the design and reveal possible weaknesses and their consequences. An example for a guide word supported analysis of the signals A_1 to A_n is shown in Table E.2.

Table E.2 — Example for a guide word supported analysis

Example for guide word	Interpretation	Consequence	Safety measure	Required activity
More	Signal value exceeds permitted range	Function O will generate erroneous output	Guard M limits signal A_1 to A_n to the permitted maximum range	Implement Guard M
Less	Signal value falls below the permitted range	Function O will generate erroneous output	Guard M limits signal A_1 to A_n to the permitted minimum range	Implement Guard M
Other than	Values of signals A_1 to A_n are inconsistent	Function O will generate erroneous output	Guard M checks signal A_1 to A_n for consistency using physical dependencies	Implement Guard M

E.4 Mitigation strategy for weaknesses identified during safety and dependent failure analyses

The results of safety analyses and dependent failure analyses at the software architectural level allow the selection of adequate safety measures to prevent or to detect and control faults and failures with the potential to violate safety requirements or safety goals.

A safety measures determination strategy can be based on the following considerations:

— the criticality of each identified fault, based on whether it can violate a safety goal or safety requirement allocated to architectural elements, and their assigned ASIL;

— whether the architectural design can be improved to eliminate identified weaknesses or to reduce the criticality of the identified fault;

— the effectiveness of determined development-time safety measures and whether they can sufficiently mitigate the identified fault based on its criticality with corresponding rationale (e.g. rationale for the effectiveness of defined verification activities considering the software fault model used during the analyses);

— the effectiveness of determined safety mechanisms to mitigate the identified critical fault(s) for which development-time measures are considered insufficient; and

— the complexity of the software architectural design (e.g. number of interfaces and software components) or the software components (e.g. number of assigned requirements or size of the component).

EXAMPLE For more complex software, an argument based solely on development-time measures is less suitable.

According to the strategy above, it is not always necessary to implement safety mechanisms which are able to prevent or detect and control such faults or failures at runtime.

Bibliography

[1] ISO/IEC 12207:2008, *Systems and software engineering — Software life cycle processes*

[2] IEC 61508:2010, *(all parts), Functional safety of electrical/electronic/programmable electronic safety-related systems*

[3] MISRA C:2012, Guidelines for the use of the C language in critical systems, ISBN 978-1-906400-10-1, MIRA, March 2013

[4] MISRA AC GMG. Generic modelling design and style guidelines, ISBN 978-906400-06-4, MIRA, May 2009

[5] ISO/IEC/IEEE 29119:2013, *(all parts), Software and systems engineering — Software testing*

[6] Bieman J.M., Dreilinger D., Lin L. "Using fault injection to increase software test coverage," in Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on, vol., no., pp.166-174, 30 Oct-2 Nov 1996 doi: 10.1109/ISSRE.1996.558776

[7] Jia Y., Merayo M., Harman M. 2015) Introduction to the special issue on Mutation Testing. *Softw. Test. Verif. Reliab.,* **25**: 461–463

[8] ISO 26262-12:2018, *Road Vehicles — Functional safety — Part 12: Adaptation of ISO 26262 for Motorcycles*

ICS 43.040.10

Price based on 57 pages