INTERNATIONAL STANDARD

ISO 19014-4

First edition
2020-07

Earth-moving machinery — Functional safety —

Part 4: Design and evaluation of software and data transmission for safety-related parts of the control system

Engins de terrassement — Sécurité fonctionnelle —

Partie 4: Conception et évaluation du logiciel et de la transmission des données pour les parties relatives à la sécurité du système de commande

Reference number
ISO 19014-4:2020(E)

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

Published in Switzerland

Contents

Page

Foreword

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

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

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

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

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

This document was prepared by ISO/TC 127, *Earthmoving machinery*, Subcommittee SC 2, *Safety, ergonomics and general requirements,* in collaboration with the European Committee for Standardization (CEN) Technical Committee CEN/TC 151, *Construction equipment and building material machines - Safety,* in accordance with the Agreement on technical cooperation between ISO and CEN (Vienna Agreement).

This first edition of ISO 19014-4, together with other parts in the ISO 19014 series, cancels and replaces ISO 15998:2008 and ISO/TS 15998-2:2012, which have been technically revised.

The main changes compared to the previous documents are as follows:

— additional requirements for software development,

— requirements for software-based parametrization development,

— requirements for transmission of safety related messages on a communication bus, and

— requirements for software validation and verification of machine performance levels.

A list of all parts in the ISO 19014 series can be found on the ISO website.

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

Introduction

This document addresses systems comprising any combination of electrical, electronic, and programmable electronic components [electrical/electronic/programmable electronic systems (E/E/PES)] used for functional safety in earth-moving machinery.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The machinery concerned and the extent to which hazards, hazardous situations, or hazardous events are covered are indicated in the Scope of this document.

When requirements of this type-C standard are different from those which are stated in type-A or type-B standards, the requirements of this type-C standard take precedence over the requirements of the other standards for machines that have been designed and built according to the requirements of this type-C standard.

Earth-moving machinery — Functional safety —

Part 4: Design and evaluation of software and data transmission for safety-related parts of the control system

1 Scope

This document specifies general principles for software development and signal transmission requirements of safety-related parts of machine-control systems (MCS) in earth-moving machinery (EMM) and its equipment, as defined in ISO 6165. In addition, this document addresses the significant hazards as defined in ISO 12100 related to the software embedded within the machine control system. The significant hazards being addressed are the incorrect machine control system output responses from machine control system inputs.

Cyber security is out of the scope of this document.

NOTE For guidance on cybersecurity, see an appropriate security standard.

This document is not applicable to EMM manufactured before the date of its publication.

2 Normative references

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

ISO 6750-1, *Earth-moving machinery — Operator's manual — Part 1: Contents and format*

ISO 12100:2010, *Safety of machinery — General principles for design — Risk assessment and risk reduction*

ISO 13849-1, *Safety of machinery — Safety-related parts of control systems — Part 1: General principles for design*

ISO 19014-1, *Earth-moving machinery — Functional safety — Part 1: Methodology to determine safety-related parts of the control system and performance requirements*

ISO 19014-2:—[1)], *Earth-moving machinery — Functional safety — Part 2: Design and evaluation of hardware and architecture requirements for safety-related parts of the control system*

3 Terms and definitions

For the purposes of this document, the terms and definitions in ISO 12100, ISO 19014-1, ISO 13849-1 and the following apply.

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

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

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

1) Under preparation. Stage at the time of publication: ISO/DIS 19014-2:2020.

3.1
bus system
subsystem used in an electronic control system for the transmission of *messages* (3.6)

Note 1 to entry: The bus system consists of the system unit (sources and sinks of information), a transmission path/transmission medium (e.g. electrical lines, fiber-optical lines, radio frequency transmission) and the interface between message source/sink and bus electronics (e.g. protocol application specific integrated circuit, transceivers).

3.2
encapsulated bus system
bus system (3.1) comprising a fixed number or a predetermined maximum number of bus participants connected to each other through a transmission medium with well-defined and fixed performance/characteristics

3.3
failure of peer communication
situation in which the communication peer is not available

3.4
unintended message repetition
situation in which the same *message* (3.6) is unintentionally sent again

3.5
message repetition
situation in which the same *message* (3.6) is intentionally sent again

Note 1 to entry: This technique of resending the same message addresses failures such as message loss (3.10).

3.6
message
electronic transmission of data

Note 1 to entry: Transmitted data can include user data, address, or identifier data and data to ensure transmission integrity.

3.7
ECU
electronic control unit
electronic device (electronic programmable controller) used in a control system on earth-moving machinery

[SOURCE: ISO 22448:2010, 3.3, modified — The admitted terms "ECM" and "electronic control module" have been removed.]

3.8
reaction time
time from the detection of a safety-related event until the initiation of a safety reaction

3.9
artifact
work products that are produced and used during a project to capture and convey information

3.10
message loss
unintended deletion of a *message* (3.6) due to a fault of a bus participant

3.11
incorrect sequence
unintended modification of the sequence of *messages* (3.6) due to a fault of a bus participant

Note 1 to entry: *Bus systems* (3.1) can contain elements with stored messages (first-in, first-out (FIFOs), etc.) that can modify the correct sequence.

3.12
message falsification
unintended modification of *messages* (3.6) due to an error of a bus participant or due to errors on the transmission channel

3.13
message retardation
unintended delay or prevention of the safety function, caused by an overload of the transmission path by normal data exchange or by sending incorrect *messages* (3.6)

3.14
alive counter
accounting component initialised with “0” when the object to be monitored is created or restored

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

3.15
black box testing
testing of an object that does not require knowledge of its internal structure or its concrete implementation

3.16
partition
resource entity allocating a portion of memory, input/output devices, and central processing unit usage to one or more *system tasks* (3.21)

Note 1 to entry: The partitions can be assigned to one or more subsystems within the microcontroller network.

3.17
software partitioning
software fault (3.26) containment method consisting of assigning resources to specific software components with the intention of avoiding the propagation of a software fault to multiple *partitions* (3.16)

3.18
software component
one or more *software modules* (3.19)

[SOURCE: ISO 26262-1:2018, 3.157, modified — The word "units" has been replaced with "modules".]

3.19
software module
independent piece of software that can be independently tested and traced to a specification

Note 1 to entry: The software module is an indivisible software component.

3.20
software partitions
runtime environment with separate system resources assigned

3.21
system task
runtime entities that are executed within the resource budget of *partitions* (3.16) and with different priorities

3.22
independence of software
exclusion of unintended interactions between software components, as well as freedom from impact on the correct operation of a software component resulting from errors of another software component

3.23
operational history
operating data about a software component or a *software module* (3.19) during its time in service

3.24
maximum cycle time
static time to access a communication bus between nodes at a bus or node level

Note 1 to entry: The application of a time-triggered protocol ensures this cycle time is not exceeded.

3.25
maximum response time
fixed time assigned to a system activity to exchange globally-synchronised *messages* (3.6) on a bus in a time-triggered architecture

3.26
software fault
incorrect step, process, or data definition in software which causes the system to produce unexpected results

3.27
impact analysis
documentation that records the understanding and implications of a proposed change

3.28
configuration management process
task of tracking and controlling changes to the *artifacts* (3.9) in the development process

3.29
constant transmission of messages
situation in which the faulty node continually transmits *messages* (3.6) that compromises the operation of the bus

3.30
blocking access to the data bus
situation in which the faulty node does not adhere to the expected patterns of use and makes excessive demands of service, thereby reducing its availability to other nodes

4 Software development

4.1 General

The main objective of the following requirements is to achieve software reliability by means of readable, understandable, testable, and maintainable software. This clause gives recommendations for the design of software and the subsequent related testing. The avoidance of software faults shall be considered during the entire software development process.

Where an existing software component has been developed to a previous standard and demonstrated through application usage and validation to reduce the risk to as low as reasonably practicable, there shall be no requirement to update the software life cycle documentation at the software module level.

Machine control software shall comply with the safety requirements of this clause. In addition, the machine control software shall be designed and developed according to the principles of ISO 12100:2010 for relevant but not significant hazards which are not dealt with by this document.

4.2 Planning

A plan shall be developed to define the relationship between the individual phases of the software development and the related artifacts.

Appropriate methods and measures from Table 3 through Table 9 shall be selected for software development according to the machine performance level required (MPLr).

The MPLr of the system may be achieved by adding, in parallel, two systems of a lower performance level. When adding in parallel (according to ISO 19014-2), the software can be developed in each system to the lower MPLr requirements. This is only allowable when there are no common cause failures between the two systems.

The suitability of the selected methods or measures to the application shall be justified and shall be made at the beginning of each planned development phase. For a particular application, the appropriate combination of methods or measures shall be stated during development planning. Methods or measures not listed in Table 3 through Table 9 may be used.

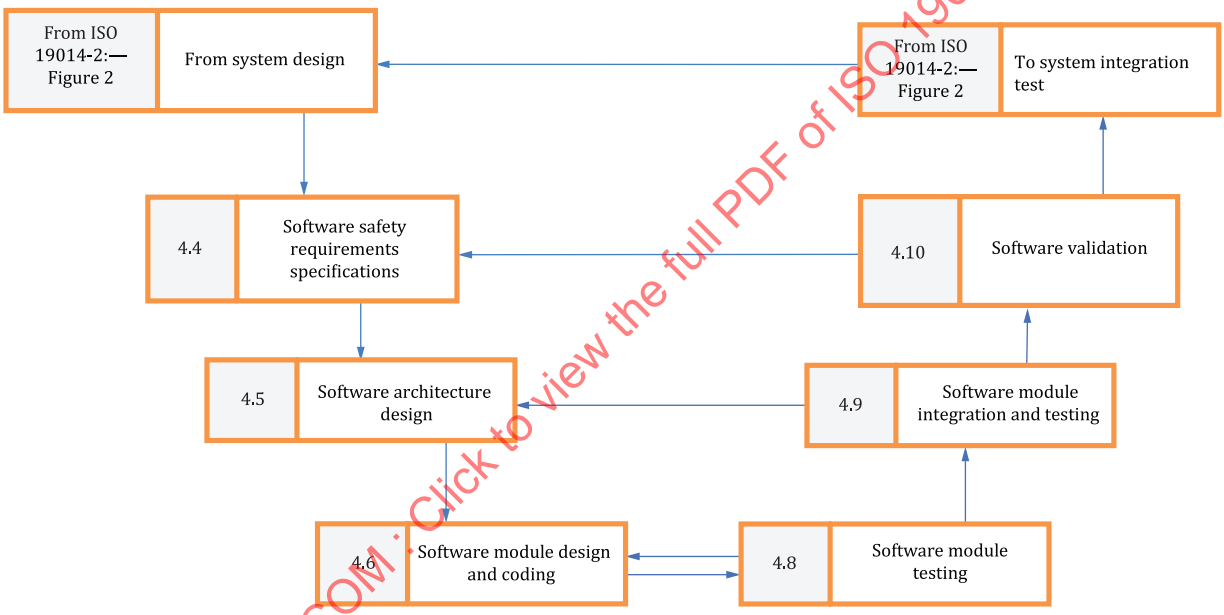


Figure 1 — Software development V-model

Figure 1 is a representation of one possible design method (V-model). Any organized, proven development process that meets the requirements of this document may be used for the software development.

When selecting methods and measures, in addition to manual coding, model-based development may be applied where the source code is automatically generated from models.

With each method or measure in the tables, there is a different level of provision for each performance level. Table 1 indicates the requirements.

Table 1 — Software safety requirements specification

Symbol	Software safety requirements specification
+	The method or measure shall be used for this MPLr. In case this method or measure is not used, the related rationale shall be documented during the safety planning phase.
o	The method or measure may be used for this MPLr.
–	The method or measure is not suitable to meet this MPLr.

Methods and measures corresponding to the respective MPLr shall be selected. Alternative or equivalent methods and measures are identified by letters after the number. At least one of the alternatives or equivalent methods and measures marked with a "+" shall be selected, in which case, providing a rationale is not required. An example of this is Table 2.

Table 2 — Example software safety requirements specification

Method/measure		MPLr = a	MPLr = b, c	MPLr = d	MPLr = e
1.a	Measure 1	+	+	–	–
1.b	Measure 2	+	+	+	+
1.c	Measure 3	+	+	+	+

In this case,

— one measure from Measure 1, Measure2 or Measure 3 shall be fulfilled for MPLr = a, b, c;

— one measure from Measure2 or Measure 3 shall be fulfilled for MPLr = d, e;

— otherwise, a rationale shall be provided about the unspecified alternative method/measure to satisfy the requirement of the standard for the specific MPLr.

Rationale or justification shall be provided if other equivalent methods or measures are used instead of the listed methods or measures.

If a software component has any impact on different safety functions with a different MPLr, then the requirements related to the highest MPLr shall apply.

If the software contains safety-related and non-safety-related components, then the overall embedded software machine performance level achieved (MPLa) shall be limited to the software component with the lowest MPLa; this requirement does not apply when adequate independence between the software components can be demonstrated in accordance with Clause 7.

When reusing a software component that is intended to be modified, an impact analysis shall be performed. An action plan shall be developed and implemented for the overall software life cycle, based on the result of the impact analysis, to ensure that the safety goals are met.

4.3 Artifacts

Once the individual phases of software development plan have been determined, the artifacts shall be defined for each phase to be carried out. Other phases and related artifacts can be added by distributing the activities and tasks. Taking into account the extent and complexity of the project, all artifacts in the individual phases shown in Figure 1 may be modified.

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

As part of the software development process, the artifacts shall be:

a) documented according to the outcomes expected from the planned phases;

b) modified as a consequence of an impact analysis, and only the impacted software shall be regression tested;

c) subject to a configuration management process.

The first artifact applicable to the process is the software development plan. The subsequent artifacts, defined by the plan, shall include:

— design specification and related verification report, for each software design phase (descending branch of the V-model in Figure 1);

— test specification and related test report, for each software (SW) testing phase (rising branch of the V-model in Figure 1);

— executable software.

4.4 Software safety requirements specification

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

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

— functions related to the detection, indication, and handling of faults by the safety-related parts of control systems (SRP/CS);

— functions related to the detection, indication, and handling of faults in the software;

— functions related to the online and offline tests of the safety functions;

NOTE 1 An online test is performed while the system being tested is in use. An offline test is performed while the system being tested is not in use.

NOTE 2 An example of an online test would be checking for faults in the steering system while driving the machine. An example of an offline test would be checking for faults in the steering system prior to allowing machine movement.

— functions that allow modifications of safety-related software parameters;

— interfaces with functions that are not safety-related;

— performance and response time;

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

Appropriate method or measures shall be selected from Table 3 to meet the specified MPLr.

Table 3 — Software safety requirements specification

	Method/measure	Reference	MPLr = a	MPLr = b, c	MPLr = d	MPLr = e
1.	Requirements specification in natural language	A.1	+	+	+	+
2.	Computer aided specification tools	A.2	o	o	o	+
3.a	Informal methods	A.3	+	+	+	–
3.b	Semi-formal methods	A.4	+	+	+	+
3.c	Formal methods	A.5	+	+	+	+
4.	Forward traceability between the system safety requirements and the software safety requirements	A.6	o	o	o	+
5.	Backward traceability between the software safety requirements and the system safety requirements		o	o	o	+
6.a	Walk-through of software safety requirements	A.7	+	+	+	–
6.b	Inspection of software safety requirements	A.8	+	+	+	+
NOTE The detailed description of these methods/measures is in Annex A.						

4.5 Software architecture design

Software architecture that describes the hierarchical structure of all the safety-related software components of each safety control system (SCS) shall be developed based on the software safety requirements. Appropriate methods or measures shall be selected from Table 4 to meet the specified MPLr.

Table 4 — Software architecture design

Method/measure		Reference	MPLr = a	MPLr = b, c	MPLr = d	MPLr = e
1.a	Informal methods	A.3	+	+	+	–
1.b	Semi-formal methods	A.4	+	+	+	+
1.c	Formal methods	A.5	+	+	+	+
2.	Computer-aided design tools	A.9	o	o	o	+
3.a	Cyclic behaviour, with guaranteed maximum cycle time	A.10	o	o	+	+
3.b	Time-triggered architecture		o	o	+	+
3.c	Event-driven, with guaranteed maximum response time		o	o	+	+
4.	Forward traceability between the software safety requirements specification and the software architecture	A.6	o	o	o	+
5.	Backward traceability between the software architecture and the software safety requirements specification		o	o	o	+
6.a	Walk-through of software architecture	A.7	+	+	+	–
6.b	Inspection of software architecture	A.8	+	+	+	+
NOTE The detailed description of these methods/measure is in Annex A.						

4.6 Software module design and coding

The objectives of this phase of software development are:

— specifying, in detail, the behaviour of the safety-related software modules that are prescribed by the software architecture;

— generating readable, testable, and maintainable software modules (e.g. manual code, model, etc.);

— verifying that the software architecture has been fully and correctly implemented.

Appropriate methods or measures shall be selected from Table 5 to meet the specified MPLr. There is no requirement to review auto-generated code.

Table 5 — Software module design and coding

Method/measure		Reference	MPLr = a	MPLr = b, c	MPLr = d	MPLr = e
1.a	Informal methods	A.3	+	+	–	–
1.b	Semi-formal methods	A.4	+	+	+	+
1.c	Formal methods	A.5	+	+	+	+
NOTE The detailed description of these methods/measures is in Annex A.						
[a] The use of trusted and verified software elements is highly recommended.						
[b] These methods or measures are not always applicable for graphical modelling notations used in model-based development.						

Table 5 *(continued)*

Method/measure		Reference	MPLr = a	MPLr = b, c	MPLr = d	MPLr = e
2.	Computer aided design tool	A.9	o	o	o	+
3.	Use of design and coding standards	A.11	o	+	+	+
4.	No unstructured control flow in programs in higher level languages[b]		o	o	+	+
5.	Limited automatic type conversion[b]		o	o	+	+
6.	Limited use of interrupts[b]		o	o	o	+
7.	Limited use of pointers[b]		o	o	o	+
8.	Limited use of recursion		o	o	o	+
9.a	Dynamic variables or objects without online check[b]	A.12	o	o	–	–
9.b	Dynamic variables or objects with online check[b]	A.13	o	o	+	+
10.	Software module size limit	A.14	+	+	+	+
11.	One entry/one exit point in subroutines and functions[b]		o	+	+	+
12.	Fully defined interface		o	+	+	+
13.	Information hiding/encapsulation		o	o	+	+
14.	Software complexity control		o	o	o	+
15.	Structured design or coding	A.15	o	+	+	+
16.	Defensive design or code	A.16	o	o	o	+
17.	Use of trusted/verified software elements[a]	A.17	o	o	o	o
18.	Forward traceability between the software safety requirements specification and the software design	A.6	o	o	o	+
19.a	Walk-through of software design, source code or both	A.7	+	+	+	–
19.b	Inspection of software design, source code or both	A.8	+	+	+	+

NOTE The detailed description of these methods/measures is in Annex A.

[a] The use of trusted and verified software elements is highly recommended.

[b] These methods or measures are not always applicable for graphical modelling notations used in model-based development.

4.7 Language and tool selection

The safety integrity of the software being developed can be directly affected by the programming language selected, the tools used during development and testing, and the use of existing, trusted, verified software modules. Appropriate methods or measures shall be selected from Table 6 to meet the specified MPLr.

Table 6 — Language and tool selection

Method/measure		Reference	MPLr = a	MPLr = b, c	MPLr = d	MPLr = e
1.	Suitable programming language	A.18	+	+	+	+
2.	Language subset support	A.19	o	o	o	+
3.a	Tools and translators with increased confidence from use or validation	A.20	o	+	+	+
3.b	Certified tools and certified translators	A.21	o	+	+	+

NOTE The detailed description of these methods/measures is in Annex A.

4.8 Software module testing

The objective of software module testing is to verify that the designed and implemented software modules correctly fulfil the software safety design. In this phase, a procedure for testing the software modules against their requirements shall be produced and the tests shall be carried out in accordance with that procedure.

For a systematic approach to software module testing, the use of appropriate tools for test management and test automation supports the work-intensive and error-prone tasks in software module testing. The availability of support tools encourages a more exhaustive approach to both normal and regression testing.

For easier verification, validation, assessment, and maintenance, all data, decisions, and rationale should be documented throughout the software project. Documentation on the software modules should include:

— testing performed;

— decisions and their rationale;

— problems and their solutions.

NOTE Data recording is important for the maintenance of computer systems as the rationale for certain decisions made during the development project is not always known by the maintenance engineers.

Appropriate methods or measures shall be selected from Table 7 to meet the specified MPLr.

Software module testing may be executed in different environments, for example:

— model-in-the-loop tests,

— software-in-the-loop tests,

— processor-in-the-loop tests,

— hardware-in-the-loop tests.

Table 7 — Software module testing

Method/measure		Reference	MPLr = a	MPLr = b, c	MPLr = d	MPLr = e
1.	Boundary value analysis	A.22	o	o	+	+
2.	Control flow analysis	A.23	o	o	+	+
3.	Data flow analysis	A.24	o	o	+	+
4.	Test case execution from boundary value analysis	A.25	o	o	o	+
5.	Functional/black box testing (including fault insertion (FI) testing)	A.26	+	+	+	+
6.a	Structure test coverage (entry points)	A.27	o	+	–	–
6.b	Structure test coverage (statements)		o	+	+	-
6.c	Structure test coverage (branches)		o	+	+	+
7.	Equivalence classes and input partition testing[a]	A.28	o	o	+	+
8.	Test case execution from model-based test case generation	A.29	o	o	o	+

NOTE The detailed description of these methods/measures is in Annex A.

[a] The tester may choose not to perform this test method, but shall perform it at the integration level, when required.

Table 7 *(continued)*

	Method/measure	Reference	MPLr = a	MPLr = b, c	MPLr = d	MPLr = e
9.	Response timings and memory constraints testing[a]	A.30	o	+	+	+
10.	Performance requirements testing[a]		o	+	+	+
11.	Avalanche/stress testing[a]		o	o	o	+
12.	SW module interface testing	A.31	o	o	o	+
13	Back-to-back comparison testing	A.32	o	o	+	+
14.	Forward traceability between the software module design and the module test specifications	A.6	o	o	o	+

NOTE The detailed description of these methods/measures is in Annex A.

[a] The tester may choose not to perform this test method, but shall perform it at the integration level, when required.

4.9 Software module integration and testing

The objectives of this phase of software development are:

— integrating the software modules into software components throughout the embedded software of the safety control system;

— verifying that the software requirements are correctly realized by the embedded software.

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

Appropriate methods/measures shall be selected from Table 8 to meet the specified MPLr. However, the tester may choose not to apply a test method or measure at the integration level, but shall apply the method or measure at the module level, when required.

Software module integration testing may be executed in different environments, for example:

— model-in-the-loop tests,

— software-in-the-loop tests,

— processor-in-the-loop tests,

— hardware-in-the-loop tests.

Table 8 — Software module integration and testing

	Method/measure	Reference	MPLr = a	MPLr = b, c	MPLr = d	MPLr = e
1.	Functional/black box testing (including fault insertion (FI) testing)	A.26	+	+	+	+
2.	Equivalence classes and input partition testing	A.28	o	o	+	+
3.	Response timings and memory constraints	A.30	o	+	+	+
4.	Performance requirements testing		o	+	+	+
5.	Avalanche/stress testing		o	o	o	+
6.	Back-to-back comparison testing	A.32	o	o	+	+
7.	Forward traceability between the software architecture design and the integration test specifications	A.6	o	o	o	+

NOTE The detailed description of these methods/measures is in Annex A.

4.10 Software validation

The objective of this phase of software development is showing that the software safety requirements are correctly realized by the embedded software.

Testing shall be the main verification method for software. Animation and modelling may be used to supplement the verification activities.

The software shall be exercised by simulation of:

— input signals present during normal operation;

— anticipated occurrences;

— undesired conditions requiring system action.

The effectiveness of the test procedures, and of any other measures used, shall be evaluated against the safety requirements specifications on conclusion of the verification process.

Appropriate methods/measures shall be selected from Table 9 to meet the specified MPLr.

Table 9 — Software validation

Method/measure		Reference	MPLr = a	MPLr = b, c	MPLr = d	MPLr = e
1.a	Machine network test	B.1	+	+	+	+
1.b	Hardware-in-the-loop test	B.2	+	+	+	+
1.c	Machine level test	B.3	+	+	+	+
2.	Forward traceability between the software safety requirements specification and software verification (including data verification) plan	A.6	o	o	o	+
3.	Backward traceability between the software verification (including data verification) plan and the software safety requirements specification		o	o	o	+
NOTE The detailed description of these methods/measures is in Annex B.						

The test method shall be carried out as specified in Annex B.

5 Software-based parameterization

5.1 General

Software-based parameterization refers to the possibility of adapting the software system to different requirements, after completion of development, by changing parameters in order to modify the functionality of the software. Software-based parameterization of safety-related parameters shall be considered part of the SRP/CS and shall be described in the software safety requirements specification.

Software-based parameters include:

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

— system parameters (e.g. value for low idle speed, engine characteristic diagrams, etc…);

— calibration data (e.g. machinery specific, limit stop for throttle setting, etc…).

5.2 Data integrity

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

— the range of valid inputs;

— data corruption before and after transmission;

— the errors from the parameter transmission process;

— the effects of incomplete parameter transmission; and

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

5.3 Software-based parameterization verification

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

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

— verification that the safety-related parameters have been checked for plausibility by use of invalid values written in the software during the configuration phase to verify its behaviour;

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

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

6 Transmission protection of safety-related messages on bus systems

This clause gives recommendations for the transmission protection of safety-related messages used in SCS and in the communication that can take place between various components (e.g. microcontrollers, intelligent sensors, intelligent actuators) within an SCS as shown in Figure 2.

NOTE 1 At the time of publication, only encapsulated bus systems in which the manufacturer has defined the number and type of bus participants (i.e. units connected to the bus) are considered. Data and address busses internal to the CPU and ECU internal devices are excluded.

Appropriate methods and measures from Table 11 shall be implemented to control the errors as shown in Table 10.

Table 10 — Control of transmission error and performance levels

Transmission errors	MPLr = a, b, c	MPLr = d,e
Failure of peer communication	YES[a]	YES
Message falsification	YES[a]	YES
Message repetition	NO	YES
Message loss	YES[a]	YES
Message insertion	NO	YES
Incorrect sequence	NO	YES
Message retardation	YES[a]	YES
Blocking access to the data bus	YES[a]	YES
Constant transmission of messages	YES[a]	YES
[a] Does not apply to category B or 1 systems since diagnostic coverage is not required		

NOTE 2 Techniques can be implemented on protocols such as SAE J1939 to address the transmission errors.

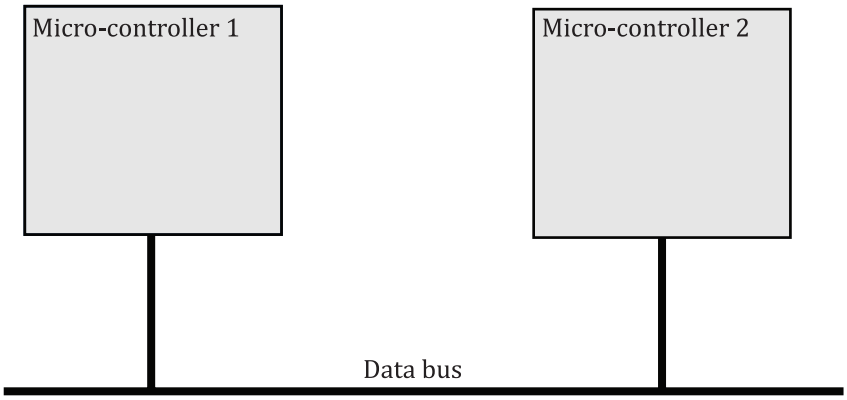


Figure 2 — Micro-controller network consisting of electronic control units on a data bus

Table 11 defines some of the methods and measures to protect against the transmission errors by excluding particular fault effects related to the communication or protection of the safety-related data in the SCS. Other methods and measures may be used to protect against errors.

Table 11 — Methods and measures within the scope of a microcontroller network

			Transmission errors (see Clause 6)								
			Failure of peer communication	Unintended message repetition	Message loss	Message insertion	Incorrect sequence	Message falsification	Message retardation	Blocking access to the data bus	Constant transmission of messages
Method/ measure	D.1	Keep alive messages	YES	NO	NO	NO	NO	NO	NO	YES	YES
	D.2	Alive counter	NO	YES	YES	YES	NO	NO	NO	NO	YES
	D.3	Cyclic redundancy check (CRC)	NO	NO	NO	NO	NO	YES	NO	NO	NO
	D.4	Sequence number	NO	YES	YES	YES	YES	NO	NO	NO	YES
	D.5	Message repetition	NO	NO	YES	NO	YES	YES	NO	NO	NO
	D.6	Watchdog	YES	NO	YES	NO	NO	NO	YES	YES	YES
	D.7	Time-triggered data bus	YES	NO	NO	YES	NO	NO	YES	NO	NO
	D.8	Bus guardian	NO	NO	NO	NO	NO	NO	NO	NO	YES
	D.9	Minislotting	NO	NO	NO	NO	NO	NO	YES	NO	YES

The fulfilment of the MPLr by the measures implemented on the network communication may be verified by means of the calculation of the data integrity assurance, as reported in Annex C. Annex D has additional information on Table 11.

7 Independence by software partitioning

7.1 General

Software partitioning is intended to aid the designer in proving independence for software components. Adequate independence of software components is guaranteed by excluding particular software fault effects violating this independence.

For that purpose, methods and measures shall be implemented with adequate effectiveness:

— to control hazards that can occur in subsystems so that they cannot affect other subsystems;

— to have adequate independence of software components by software partitioning.

In order to achieve adequate independence of software components, the system resources should be assigned to independent partitions, each representing a particular runtime environment. The use of software partitioning is not restricted to the co-existence of software of different MPL in the same runtime environment. It can also support:

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

b) co-existence of software of a different nature (in-house, third party).

For software partitions with MPLr equal to c, d, e, independence is required.

NOTE Partitions can be allocated within a single microcontroller or allocated to several microcontrollers within a microcontroller network.

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

a) several partitions within a single microcontroller;

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

The concept of software partitioning and its associated methods and measures for the independence of software components shall be taken into account when specifying the software architecture.

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

7.2 Several partitions within a single microcontroller

In order to guarantee adequate independence of software components within a single microcontroller as shown in Figure 3, correct execution of the safety-related function shall be protected against any of the following fault effects:

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

NOTE 1 Memory corruption applies to memory mapped input/output (I/O), memory registers, etc.

— blocking of partitions (due to communication deadlocks);

— wrong allocation of processor execution time;

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

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

NOTE 2 Corruption of I/O interface refers only to external devices.

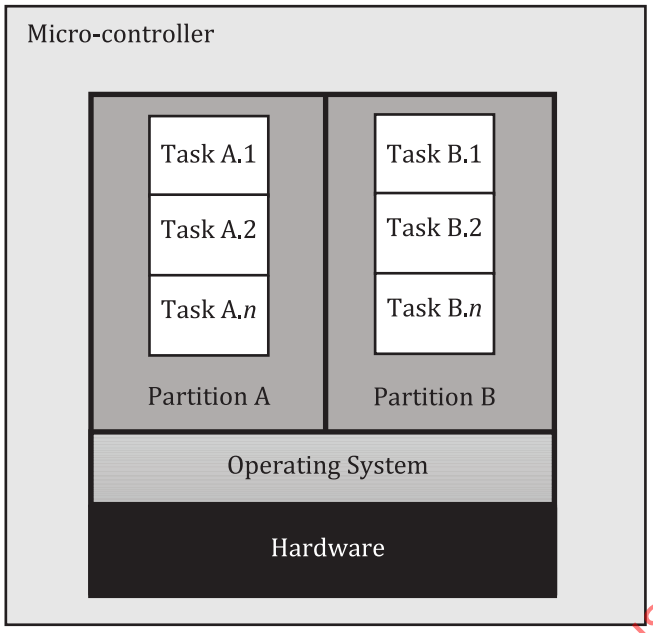


Figure 3 — Several partitions within a single microcontroller

Table 12 defines the measures so that all appropriate fault effects listed above can be handled to ensure adequate effectiveness and adequate independence of software components. Additional information is in Annex E.

Table 12 — Methods and measures within microcontroller

			Fault effect				
			Memory corruption	Blocking of partitions	Wrong processor execution time allocation	Wrong communication peer	Corruption of I/O interface
Method/ measure	E.1	Unambiguous bidirectional communication object	NO	NO	NO	YES	NO
	E.2	Strictly two unidirectional communication objects	NO	NO	NO	YES	NO
	E.3	IDs for identification, acknowledgement, or both.	NO	NO	NO	YES	NO
	E.4	Asynchronous data communication	NO	YES	NO	NO	NO
	E.5	Strict priority-based scheduling	NO	NO	YES	NO	NO
	E.6	Time slicing method	NO	NO	YES	NO	NO
	E.7	Memory protection mechanisms	YES	NO	NO	NO	YES
	E.8	Verification of safety critical data	YES	NO	NO	NO	NO
	E.9	Static analysis	YES	NO	NO	NO	NO
	E.10	Static allocation	YES	YES	YES	YES	YES

7.3 Several partitions within the scope of an ECU network

In order to guarantee adequate network communication, when the independence of software components is implemented by means of a microcontroller network, refer to Clause 6.

8 Information for use

8.1 General

Information for use shall be provided in accordance with ISO 12100:2010, 6.4.

8.2 Instruction handbook

Information for use shall be included in the machine instruction handbook (i.e., operator's manual). This information shall use ISO 6750-1 for guidance. This information can include the following:

— descriptions of symbols that are displayed to the operator and what action is required;

— descriptions of error messages that are displayed to the operator and what action is required;

— descriptions of warning messages that are displayed to the operator and what action is required;

— descriptions of calibration procedures that the operator is required to perform.

Annex A
(informative)

Description of software methods/measures

A.1 Requirements specification in natural language

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

A.2 Computer-aided specification tools

Use of these tools to facilitate automatic detection of ambiguity and completeness in semi-formal and formal specification methods should be used. The tool should produce specifications in the form of a database that can be automatically inspected to assess consistency and completeness. In general, this method supports not only the creation of the specification but also of design and other phases of the project life cycle.

A.3 Informal methods

Informal methods should provide a means of developing a description of a system at some stage in its development, i.e. specification, design or coding, typically by means of natural language, diagrams, figures, etc.

A.4 Semi-formal methods

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

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

Examples of semi-formal methods include, but are not limited to: data flow diagrams, pseudo code, modelling tools, finite state machine/state transition diagrams.

A.5 Formal methods

Formal methods provide a means of developing description of a system in a strict notation that should be subjected to mathematical analysis to detect various classes of inconsistency or incorrectness. Moreover, the description should in some cases be analysed by machine with a rigor similar to the syntax checking of source program compiler, or animated to display various aspects of the behaviour of the system described. Animation can give extra confidence that the system meets the real requirement as well as the formally specified requirement, because it improves human recognition of the specified behaviour.

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

A.6 Traceability of the safety software

In order to ensure that the software resulting from development activities meets the requirements for correct operation of the safety-related system, consistency between the software development stages is essential. Traceability between activities is a key concept that confirms:

1. that decisions made at an earlier stage are adequately implemented in later stages (forward traceability),
2. that decisions made at a later stage are required by earlier decisions (backward traceability).

Forward traceability is broadly concerned with checking that a requirement is adequately addressed in later software development stages. Forward traceability is valuable at several points in the safety software development:

— from the system safety requirements to the software safety requirements;

— from the software safety requirements specification to the software architecture;

— from the software safety requirements specification to the software design;

— from the software design specification to the module and integration test specifications;

— from the system and software design requirements for hardware/software integration to the hardware/software integration test specifications;

— from the software safety requirements specification to the software safety validation plan;

— from the software safety requirements specification to the software modification plan (including re-verification and re-validation);

— from the software design specification to the software verification (including data verification) plan;

Backward traceability is broadly concerned with checking that every implementation (interpreted in a broad context, and not confined to code implementation) decision is clearly justified by some requirement. If this justification is absent, then the implementation contains something unnecessary that adds to the complexity but not necessarily address any genuine requirement of the safety-related system. Backward traceability is valuable at several points in the safety software development:

— from the safety requirements, to the perceived safety needs;

— from the software architecture, to the software safety requirements specification;

— from the software detailed design to the software architecture;

— from the software code to the software detailed design;

— from the software safety validation plan, to the software safety requirements specification;

— from the software modification plan, to the software safety requirements specification;

— from the software verification (including data verification) plan, to the software design specification.

A.7 Walk-through

A walk-through is a systematic, informal verification used to review an aspect of the design. During a walk through, the author of an artifact provides a step-by-step report to one or more assessors. The objective is to create a common understanding of the artifact, and to identify any errors, defects, discrepancies or problems in the artifact. A walk-through is less stringent than an inspection.

A walk-through should be used to detect software faults as soon as economically possible during development. It consists of a walk-through team selecting a small set of paper test cases, representative

sets of inputs, and corresponding expected outputs for the program. The test data is then manually traced through the logic of the program.

A.8 Inspection

An inspection is a systematic, formal verification method used to review an aspect of the design. During an inspection, the artifact is checked by one or more assessors to see whether it complies with the requirements. The inspection is organized and moderated by an inspection leader. The author of the artifact participates in the inspection but cannot lead the process.

Design inspection can be performed at design level to reveal defects in the design of the software. A design inspection is a formal, documented, comprehensive, and systematic examination of the software design to evaluate the design requirements, the capability of the design to meet these requirements, identify problems, and propose solutions. Such a review is primarily intended to verify the work of the designers and should be treated as a confirmation and refining activity.

Code inspections can be performed at coding level to reveal defects in a software element. Formal inspection is a structured process to inspect software material that is carried out by peers of the person producing the material to find defects and to enable the producer to improve the material. The producer should take no part in the inspection process, other than to brief the inspectors during the familiarization stage. Formal inspections may be carried out on specific software elements produced at any phase of the software development life cycle.

Prior to the inspection taking place the inspectors should become familiar with the materials to be inspected. The inspectors' roles in the inspection process should be clearly defined. An inspection agenda should be prepared. Entry and exit criteria should be defined based on the properties required for the software element. Entry criteria are the criteria or requirements which shall be met prior to the inspection taking place. Exit criteria are the criteria or requirements which shall be met to complete a specific process.

During the inspection the findings of the inspection should be formally recorded by the moderator, whose role is to facilitate the inspection. A consensus on the findings should be reached by all inspectors. Defects should be classified as either

a) requiring rectification prior to acceptance, or

b) requiring rectification by a given time/milestone.

Defects identified should be referred to the producer for subsequent rectification after completion of the inspection. Depending on the number and scope of identified defects, the moderator may determine the necessity for further inspection of the software material.

A.9 Computer-aided design tools

This type of tool can support to carry out the design procedure more systematically by including the use of automatic elements which are made available to structure the design of the software.

Computer-aided design tools should be used during the design of both hardware and software when available and justified by the complexity of the system. The correctness of such tools should be demonstrated by specific testing, by an extensive history of satisfactory use, or by independent verification of their output for the particular safety-related system that is being designed.

Support tools should be selected for their degree of integration. In this context, tools are integrated if they work co-operatively such that the outputs from one tool have suitable content and format for

automatic input to a subsequent tool, thus minimizing the possibility of introducing human error in the reworking of intermediate results.

NOTE Integrated Development Environment (IDE) could be employed to provide comprehensive facilities to computer programmers for software development. They normally consist of a source code editor, build automation tools, and debugger. Most of them have intelligent code completion; some contain a compiler, interpreter, or both.

A.10 Safety performance in real time

The implementation of fault-tolerance into safety-critical real-time systems with predictable behaviour is achieved by means of the following solutions:

— In a cyclic behaviour with guaranteed maximum cycle time, communication between nodes is done using a time-triggered protocol class C (TTP/C) according to a static schedule, deciding when to transmit a message and whether a received message is relevant for the particular electronic module or not. Access to the bus is controlled by a cyclic time-division multiple access (TDMA) scheme derived from the global notion of time.

— In a time-triggered architecture (TTA) system, all system activities are initiated and based on the progression of a globally synchronised time-base. Each application is assigned a fixed time slot on the time-triggered bus, which contains the messages exchanged between the jobs of each application which can therefore be exchanged only according to a defined schedule.

In an event-driven system, activities triggered by arbitrary events at unpredictable times should be considered.

NOTE Other time-triggered protocols are FlexRay and TT-Ethernet (time-triggered Ethernet).

A.11 Design rules

Coding standards should be used to facilitate verifiability of the produced code. The detailed rules should be fully agreed upon before coding. These rules typically require

— details of modularization, e.g. interfaces between software module, software module sizes;

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

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

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

— layout of the code (listing);

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

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

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

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

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

A.12 Dynamic variables or objects without online check

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

Restrictions on the use of dynamic objects are needed where the dynamic behaviour cannot be accurately predicted by static analysis (i.e. in advance of the program execution), and therefore predictable program execution cannot be guaranteed. After a dynamic variable or object has been used (for example, after exiting a subroutine) the whole memory which was allocated to it should be freed.

A.13 Dynamic variables or objects with online check

Online check determines at run time whether that the existing variables, data or code are not impacted by the allocation. If allocation is not allowed (for example, if the memory at the determined address is not sufficient), appropriate action shall be taken. This is an alternative method to statically allocating all the required variables and objects.

A.14 Modular approach

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

— A software module should have a single well-defined system task or function to fulfil.

— Connections between software modules should be limited and strictly defined.

— Collections of subprograms should be built, providing several levels of software modules.

— The software module size should be restricted to a specified value (typically two to four screen sizes, but related to a pre-defined coding standard).

— Software modules should have single entry and single exit. If more than one point of entry or exit is needed, the reason for this strategy shall be documented within the software; this ensures proper execution and maintainability of the software.

— Software modules should communicate with other software modules via their interfaces. Where global or common variables are used they should be well structured, access should be controlled, and their use should be justified in each instance.

— All software module interfaces should be fully documented.

— Any software module interface should contain only those parameters necessary for its function.

— Complexity metrics should be used to predict the attributes of programs from properties of the software itself or from its development or test history. Software tools are required to evaluate most of the measures. Some of the metrics which can be applied are, e.g., graph theoretic complexity, accessibility, number of operators and operands, and number of entries and exits per software module.

— Information hiding or encapsulation should be used to prevent unintended access to data or procedures and thereby support a good program structure. Data that is globally accessible to all software elements can be accidentally or incorrectly modified by any of these elements. Any changes to these data structures can require detailed examination of the code and extensive modifications.

Information hiding is a general approach for minimising these difficulties. The key data structures are "hidden" and can only be manipulated through a defined set of access procedures. This allows the internal structures to be modified or further procedures to be added without affecting the functional behaviour of the remaining software.

A.15 Structured programming

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

The following should be carried out so as to minimize structural complexity:

a) divide the program into appropriately small software modules, ensuring all interactions are explicit;

b) compose the software module control flow using structured constructs, (i.e. sequences, iterations and selection statements);

c) keep the number of possible paths through a software module small, and the relation between the input and output parameters as simple as possible;

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

e) where possible, use input parameters as loop constraints to perform branching, and avoid using calculations as the basis of branching and loop decisions.

Features of the programming language which encourage the above approach should be used in preference to other features which are (allegedly) more efficient, except where efficiency takes absolute priority.

A.16 Defensive programming

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

— range checking the variables;

— checking values for plausibility;

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

This first set of defensive methods helps to ensure that the numbers manipulated by the program are reasonable, both in terms of the program function and physical significance of the variables.

Read-only and read-write parameters should be separated, and their access checked. Functions should treat all parameters as read-only. Literal constants should not be accessible. This helps detect accidental overwriting or mistaken use of variables. Fault tolerant software is designed to “expect” failures in its own environment or use outside nominal or expected conditions, and behave in a predefined manner. Methods include the following:

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

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

— checking by the software of its configuration, including both the existence and accessibility of expected hardware, and also that the software itself is complete; this is particularly important for maintaining integrity after maintenance procedures.

Some of the defensive programming methods, such as control flow sequence checking, also cope with external failures.

A.17 Use of trusted/verified software elements

Trusted/verified software modules and software components may be re-used in new applications. This allows the developer to take advantage of designs which have not been formally or rigorously verified, but for which considerable operational history is available, thus reducing the amount of validation required for software modules and hardware component designs in new applications.

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

— the safety-related function has at least one year or 1 000 hours of operation with no change to the specification;

— prior operating history of the software module relates to the intended purpose in the new application, thus establishing confidence in the software module's suitability for the new application;

— no failures of safety-related functions in the re-used software during the operational history.

Careful evaluation of each function should be performed since a non-safety-related function in one application can be a safety-related function in a different application

To verify that a component or software module meets the above criteria, the following procedures should be available to provide the evidence to be documented:

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

— identification and selection of a sufficient sample of users and time of use of the application (i.e. one year or 1 000 hours of operation);

— detection and registration of failures, with related corrective actions.

If alternative procedures are followed to collect evidence of suitability, a rationale should be provided to demonstrate that the goal of trusted software component or software module is still achieved.

The described approach is extendable to supplied complex electronic components (e.g. from supplier to customer), where an unmodified software is integrated with unmodified hardware, and when the previously described requirements are fulfilled and documented.

A.18 Suitable programming language

The aim is to choose a programming language that supports the requirements of this document as much as possible, in particular defensive programming, structured programming, and possibly assertions statements. The programming language chosen should lead to easily verifiable code and facilitate program development, verification, and maintenance. Widely used languages or their subsets are preferred to special-purpose languages. Low-level languages, in particular assembly languages, present problems due to their processor/platform machine-orientated nature.

A desirable language property is that its design and use should result in programs whose execution is predictable without running the code. Given a suitably defined programming language, there is a subset

which ensures that program execution is predictable. This subset cannot (in general) be statically determined, although many static constraints can assist in ensuring predictable execution.

NOTE See IEC 61508-7: 2010, Table C.1 for a list of programming languages.

A.19 Language subset support

The use of a language subset reduces the probability of introducing programming faults and increase the probability of detecting any remaining software faults. The programming language should be examined to determine programming constructs which are either error-prone or difficult to analyse (e.g. using static analysis methods). These programming constructs should then be excluded and a language subset defined. Also, it should be documented as to why the constructs used in the language subset are safe.

NOTE MISRA 'C' (Motor Industry Software Reliability Association) guidelines are an example of software development guidelines that define a language subset and, generally, aim to facilitate code safety, security, portability, and, reliability.

A.20 Tools with increased confidence from use or validation

Tools that are proven in use or that have been validated should be applied in order to avoid difficulties due to tool failures which can arise during development, verification, and maintenance of software packages. Software tools without operating history or with any serious known faults should be avoided unless there is some other assurance of correct performance. If the software tool has shown small deficiencies, the related language constructs are noted down and carefully avoided during a safety-related project.

A software tool should only be argued as having increased confidence from use if evidence is provided for the following:

a) the software tool has been used previously for the same purpose with comparable use cases, a comparable determined operating environment, and with similar functional constraints;

b) the justification for increased confidence from use is based on sufficient and adequate data (at least one project producing a software with at least one year or 1 000 hours of operation);

c) the specification of the software tool is unchanged;

d) the occurrence of malfunctions and corresponding erroneous outputs of the software tool acquired during previous developments are accumulated in a systematic way.

A software tool should only be argued as having increased confidence from validation if evidence is provided showing the validation meets the following criteria:

a) the validation measures demonstrate that the software tool complies with its specified requirements (e.g. the standard for a programming language helps to define the requirements for validating the associated compiler);

b) the malfunctions and their corresponding erroneous outputs of the software tool occurring during validation have been analysed together with information on their possible consequences and with measures to avoid or detect them;

c) the reaction of the software tool to anomalous (out of the specified type of application) operating conditions has been examined.

A.21 Certified tools and certified translators

Tools are necessary to help developers in the different phases of software development. Wherever possible, tools should be certified so that some level of confidence can be assumed regarding the correctness of the outputs.

The certification of a tool is generally carried out by an independent third party, against independently set criteria, typically national or international standards. Ideally, the tools used in all development phases (specification, design, coding, testing and validation) and those used in configuration management should be subject to certification.

It is important to note that certified tools, and certified translators are usually certified only against their respective language or process standards; they are usually not certified in any way with respect to safety.

A.22 Boundary value analysis

Boundary value analysis should be used to detect software errors occurring at parameter limits or boundaries. The input domain of the program is divided into a number of input classes (subsets of input values/combinations) according to the equivalence relation (each value of a class results in the same output value). The tests should cover the boundaries and extremes of the classes. The tests check that the boundaries in the input domain of the specification coincide with those in the program.

Normally, the boundaries for input have a direct correspondence to the boundaries for the output range. Test cases should be written to force the output to its limited values. Consider also if it is possible to specify a test case which causes the output to exceed the specification boundary values.

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

A.23 Control flow analysis

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

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

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

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

A.24 Data flow analysis

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

Data flow analysis is a static testing method that combines the information obtained from the control flow analysis with information about which variables are read or written in different portions of code. The analysis can check for the following types of variables:

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

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

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

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

A.25 Test case execution from boundary value analysis

Test case execution from boundary value analysis (see A.22) should be used to detect software errors occurring at parameter limits or boundaries.

A.26 Functional/Black box testing

Functional testing should be used to reveal failures during the specification and design phases, and to avoid failures during implementation and the integration of software and hardware.

During the functional tests, reviews should be carried out to determine whether the specified characteristics of the system have been achieved and that the system input data which adequately characterize the normally expected operation have been given. The outputs are observed and their response is compared with that given by the specification. Deviations from the specification and indications of an incomplete specification should be documented. Functional testing of electronic components designed for a multi-channel architecture usually involves the manufactured components being tested with pre-validated partner components.

In addition, it is recommended that the manufactured components be tested in combination with other partner components of the same batch, in order to reveal common mode software faults which would otherwise have remained masked.

A.27 Structure-based testing

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

In all cases, 100 % of the selected coverage metric should be the aim; if it is not possible to achieve 100 % coverage, the reasons why 100 % cannot be achieved should be documented in the test report (for example, defensive code which can only be entered if a hardware problem arises). The methods in the following list are widely supported by testing tools:

— Entry point (call graph) coverage: ensure that every subprogram (subroutine or function) has been called at least once (this is the least rigorous structural coverage measurement).

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

— Statements: ensure that all statements in the code have been executed at least once.

— Branches: both sides of every branch should be checked. This can be impractical for some types of defensive code.

A.28 Equivalence classes and input partition testing

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

This testing strategy is based on the equivalence relation of the inputs, which determines a partition of the input domain. Test cases are selected with the aim of covering all the partitions previously specified. At least one test case is taken from each equivalence class.

There are two basic possibilities for input partitioning which are

— equivalence classes derived from the specification – the interpretation of the specification can be either input orientated, for example the values selected are treated in the same way, or output orientated, for example the set of values lead to the same functional result;

— equivalence classes derived from the internal structure of the program – the equivalence class results are determined from static analysis of the program, for example the set of values leading to the same path being executed.

A.29 Test case execution from model-based test case generation

Test case execution from model-based test case generation is to facilitate efficient automatic test case generation from system models and to generate highly repeatable test suites.

Model-based testing (MBT) is an approach in which common testing tasks such as test case generation (TCG) and test results evaluation are based on a model of the system (application under test (SUT). Additionally, model-based testing can be combined with source code level test coverage measurement; functional models can be based on existing source code.

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

The model is an abstract, partial representation of the desired behaviour of the SUT. From this model, test models are derived, building an abstract test suite. Test cases are derived from this abstract test suite and executed against the system; tests can be run against the system model as well.

Model-based testing is specifically targeting recently the safety critical domain; it allows for early exposure of ambiguities in specification, and design. Model-based testing also provides the capability to automatically generate many non-repetitive efficient tests, evaluate regression test suites, assess software reliability and quality, and eases updating of test suites. provides the capability to automatically generate many non-repetitive efficient tests, to evaluate regression test suites and to assess software reliability and quality, and eases updating of test suites.

A.30 Performance testing

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

The requirements specification shall include throughput and response time requirements for safety-related functions. The requirements specification should include constraints on the use of total system resources. The proposed system design is compared with the stated requirements by:

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

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

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

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

Response timing and memory constraints testing can be performed to ensure that the system meets its temporal and memory requirements. Response time is the total time it takes from when a user/system makes a request until a response is received. When testing response time, one needs to find out how the application handles all the requests and how the response time increases/decreases with the load and

execution through time. Moreover, memory constraints testing verifies that all memory is explicitly initialized before it is used and blocked from other processes for use. An analysis is performed to determine the distribution demands under average and worst-case conditions. This analysis requires estimates of the resource usage and elapsed time of each system function. These estimates can be obtained in several ways; for example, comparison with an existing system or the prototyping and benchmarking of time critical systems.

Performance requirements testing are performed to establish demonstrable performance requirements of a software system. An analysis is performed of both the system and the software requirements specifications to specify all general and specific, explicit and implicit performance requirements. Each performance requirement is examined in turn to determine

— the success criteria to be achieved;

— whether a measure against the success criteria can be achieved;

— the potential accuracy of such measurements;

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

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

Avalanche/stress testing can be performed to burden the test object with an exceptionally high workload in order to show that the test object would stand normal workloads easily. It helps developers to determine if the system performs sufficiently if the use load goes well above the expected maximum. This kind of test is normally used to understand the upper limits of capacity within the system. There are a variety of test conditions applicable to avalanche/stress testing, including the following.

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

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

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

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

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

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

A.31 SW module interface testing

Module interface testing is used to detect errors in the interfaces of software components and modules.

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

— all interface variables at their extreme values;

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

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

— all values of all variables in combination (this is only feasible for small interfaces);

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

These tests are particularly important if the interfaces do not contain assertions that detect incorrect parameter values; they are also important after new configurations of pre-existing software components and modules have been generated.

A.32 Back-to-back comparison testing

Back-to-back comparison testing is to determine whether or not an implementation and model both produce the same outputs when given the same input; this is only applicable for model-based development.

Annex B
(normative)

Software validation test environments

B.1 Machine network testing

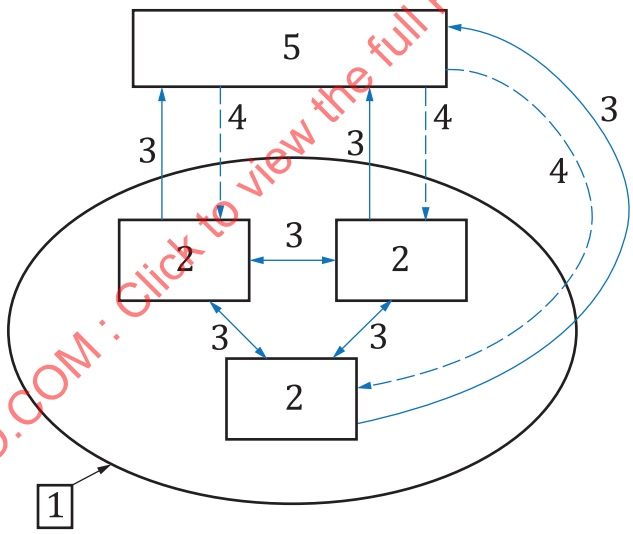
The software shall be integrated with its host microprocessor in its associated ECU; this ECU shall be integrated with the remaining ECUs that are part of the complete machine electrical system. The software shall then be tested at the interface to the ECU network, in order to demonstrate that the software performs according to specification.

The software shall be exercised as shown in Figure B.1 by simulation of

— input signals present during normal operation;

— anticipated occurrences;

— undesired conditions requiring system action.



Key

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

Figure B.1 — Machine network testing

B.2 Hardware-in-the-loop testing

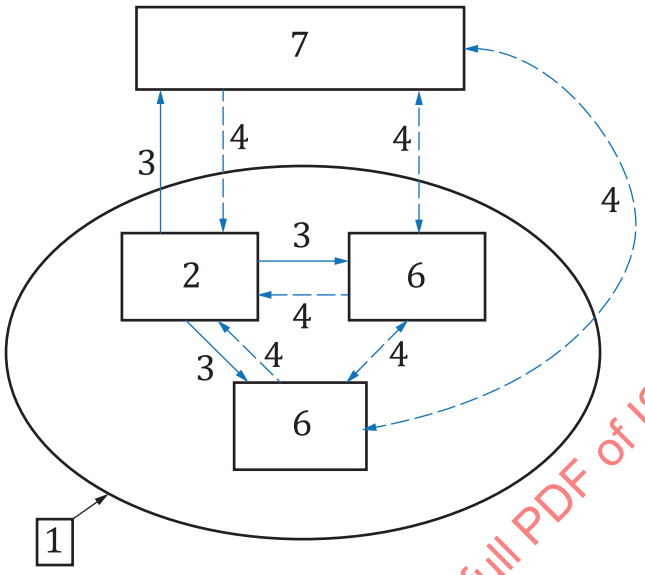
The software shall be integrated with its host microprocessor in its associated ECU, while the rest of the associated machine electrical system and its environment shall be simulated. The software shall then be tested in this simulated environment to demonstrate that the software performs according to specification.

The software shall be exercised as shown in Figure B.2 by simulation of:

— input signals present during normal operation;

— anticipated occurrences; and

— undesired conditions requiring system action.



Key

1 complete E/E/PES system
2 real ECU with real software to be tested
3 real signals to be monitored
4 signal simulated
6 ECU simulated
7 simulated environment (plant model and test vector)

Figure B.2 — Hardware-in-the-loop testing

B.3 Machine level testing

The software and the associated machine electrical system shall be integrated into the associated machine architecture. The system shall then be tested in the machine to demonstrate that the software performs according to specification.

The software shall be exercised as shown in Figure B.3 by:

— input signals present during normal operation;

— anticipated occurrences;

— undesired conditions requiring system action.