zvei
electrifying ideas

Best Practice Guideline

Software for Safety-Related Automotive Systems

V3.0

Electro and Digital Industry Association

Content

1 Objectives of this Guideline

This guideline provides lessons-learned, experiences and best practices related to the application of ISO 26262 for the development of software. Please note that the guidelines given are of general nature and do not replace a thorough consideration of the project specific development regarding achievement of “Functional Safety” considering ISO 26262.

2 Overview

This guideline is intended to be maintained and extended. The current version addresses among other things the following aspects:

- Definition of terms used in the context of “Functional Safety” and software development.
- Guidance for safety concepts and architectures for safety-related software.
- Guidance for software safety analyses on software architecture level.
- Explanation of recommended methods listed in ISO 26262 Part 6.
- Classification and qualification of software tools used in the development of embedded software.
- Remarks on the relevance of ISO 26262.
- Guidance for using Software SEooC.
- What does compliance mean?

3 Explanation of Terms

The following explanations include terms used in this document. The explanations are intended to ease the common understanding.

Term	Description
QM software	Software that is not developed according to ISO 26262 ASIL A, to D but still the software is developed according a well-defined process (e. g. an ASPICE compliant process). QM software must not be used to realize safety-related functionalities and special consideration is needed if QM software is integrated in an ECU that realizes safety-related functionalities.
Silent software	“Silent software” is a term used to describe software that does not interfere with other software with respect to memory access (e. g. range-check of index values, verification of pointer access) under the conditions defined in the Safety Manual. “Silent” software does not fulfill specific safety-related functions.
Implicitly safe	“Implicitly safe” is a term used to describe software that is silent software with additional dedicated timing properties (e. g. with respect to execution time, deadlocks and robustness with respect to input signals) under the conditions defined in the Safety Manual. “Implicitly safe” software does not fulfill specific safety-related functions.
Safety manual	A Safety Manual describes constraints and required activities for the integration and/or usage of elements that have been developed and prequalified acc. ISO 26262 as Safety Element out of Context.
Safe, safety, explicitly safe	“Safety/safe/explicitly safe software” is a term used to describe software that fulfills specific safety- related requirements under the conditions stated in the safety manual.
“Trusted mode”, system mode, privileged mode, supervisor mode	CPU mode for executing software with full access to the hardware features of the microcontroller. Software executed in this mode poses a higher risk and should be treated as such (e. g. development according to required ASIL including the implementation of appropriate safety measures).
Safety-related functionality	A functionality that realizes safety requirements.

Table 1: Explanations of terms used in this document.

4 The Relevance of ISO 26262

ISO 26262 is the recognized standard for functional safety in Automotive. But it is not a harmonized standard and therefore not necessary for the CE mark. ISO 26262 is also not an EU or UNECE directive or regulation and is therefore no prerequisite for vehicle homologation. Nevertheless, there are many reasons for implementing ISO 26262: In the European Union all products for end users must be safe according to the General Product Safety Directive 2001/95/EC. The corresponding German law is the Produktsicherheitsgesetz and other countries have similar laws. The German Produkthaftungsgesetz pledges the car manufacturers to compensate damage if they cannot prove that their product was developed according to state-of-the-art techniques and methodologies. In cases of gross negligence or intent persons can be culpable in person. Car manufacturers and suppliers face financial compensation, loss of reputation, loss of insurance protection, and even prison in cases of unsafe products. Strictly taken, only a judge in a court case can decide if ISO 26262 **was necessary**. Until then we must assume that it **is necessary**. Full compliance with ISO 26262 is typically considered to be the **minimum necessary** to fulfil state-of-the-art in respect to functional safety. It can, however, not generally be considered to be **sufficient** for product safety.

In addition to all that, most contracts given to electronic suppliers in Automotive do explicitly call for compliance with ISO 26262. Non-compliance would therefore be a breach of contract.

The conclusion from all this is that compliance with **ISO 26262 is necessary**!

What does compliance mean?

How is compliance determined? It is based on evidence supporting confidence in the achieved functional safety. Verbal statements are not enough. And it is based on an accepted safety case and (for higher ASILs) functional safety assessment, with independence graded by ASILs. Such an assessment needs to include confirmation reviews and functional safety audits.

Relevant confirmation reviews for software development are the software parts of:

- Safety plan
- Safety analyses
- Safety concept
- Safety case Auditing aims at confirming that the safety plan and all processes specified by ISO 26262 are actually implemented and the company lives up to them.

Assessments are also based on verification reviews. Relevant for software development are review reports of:

- Software safety requirements
- Hardware-software interface requirements
- Software architectural design
- Software units
- Software integration
- Software component qualification
- Safety analyses and dependent failure analyses on software architectural level

When is compliance needed? The answer is: Always when there is a risk for human health, e. g. when testing pretest vehicles, at release for production, during operation, and during and after service and repair. In most cases compliance is not necessary for lab samples.

Who determines compliance?

- In a first step this is done by reviewers. They are typically project members other than the author, examining work products for achievement of the intended ISO 26262 work product goal.
- In a second step this is done by auditors, e. g. persons from the quality department having a functional safety specific qualification. They examine the performed activities and implementation of processes for functional safety for achievement of the process-related ISO 26262 objectives.
- Finally, this is the safety assessor. He/she examines safety planning, safety processes, safety measures, safety objectives, and the safety case for judging whether functional safety is achieved. Doing so he/she typically relies on audit and review results. The safety assessor must be a person with enough independence (graded by ASILs) from the development team. Note that no assessor qualification scheme has been established. The industry relies on experience and reputation of assessors. There is also no accreditation

scheme for assessing companies required or expected by OEMs. A best practice is involving the assessor early in the project in order to reduce the risk of late non-compliance.
- The OEM is likely to scrutinize all functional safety related prerequisites and activities very thoroughly.

How to determine compliance? What are the **criteria** for a safety assessor to determine compliance? The main criteria are the **objectives** of the clauses of ISO 26262. These objectives must be achieved.

There are objectives that are more technical and others that are more process related. Examples:
- The more technical objective of the software unit design and implementation clause is to implement software units as specified.
- The more process-related objective of the software unit verification clause is to provide evidence that the software detailed design and the implemented software units fulfill their requirements and do not contain undesired functionality.

Besides objectives ISO 26262 does contain requirements and work products. Work products are results of fulfilling requirements. Fulfilled requirements are indicators for achievement of objectives. Requirements are useful especially as a basis for process steps and for checklists. Requirements may be process-related and/or technical. Examples for how to fulfill requirements:
- Technical example: Software safety mechanisms need to include mechanisms for error detection. An example of such a mechanism is a range check of input data.
- Process example: A suitable programming language is required. A process-related practice is to define a coding guideline and to use a MISRA checker (automatically at check-in) that assures compliance with the guideline.
- Process example: Safety requirements shall be traceable. Using a suitable requirements management tool is a best practice.

An assessor needs to judge whether requirements are fulfilled and whether work products comply with their ISO 26262 requirements. Fulfilled ISO 26262 requirements are an indication for achievement of objectives.

Upon an assessment the requirements corresponding to the objectives, the state-of-the-art regarding technical solutions and the applicable engineering domain knowledge are considered. The assessor is supposed to recommend acceptance of a product development if the assessor is convinced that functional safety is achieved. The assessor will only recommend acceptance if the assessor is convinced that all objectives of ISO 26262 are fulfilled. Therefore, work on a clear, understandable and convincing **chain of argumentation** and document it in the safety case.

5 Software Safety Concepts and Architectures

5.1 Introduction

In this section different software safety concepts are depicted, and some hints are given to decide for the appropriate safety concept depending on the conditions in a specific development project.
Often many of the functionalities and properties of ECU software are not safety-related, but only a part of them. Only those software elements that contribute to the implementation of safety requirements are considered safety-related.

To implement a mix of safety-related and non-safety-related functionalities there are two fundamental design options mentioned in ISO 26262:

- Develop a design in which such a mix can coexist. This is often called “Mixed ASIL Design” and is a typical approach if the portion of safety-related functionalities is rather small or third-party or QM software needs to be integrated.
- Develop the complete ECU software in conformance with the “Maximum ASIL” assigned to any of the safety-related functions within the ECU. This is often called “Maximum ASIL Design” and the typical approach if the portion of safety-related functionalities is rather large.

Figure 1 depicts different ECU types holding software elements with and without related safety requirements and illustrates these two design patterns.

Both design options must focus on the same goal: To achieve the necessary integrity of the safety functions. The level of integrity expresses the degree of trust you can have that a software will provide the stated functions and properties as demanded under specified conditions.

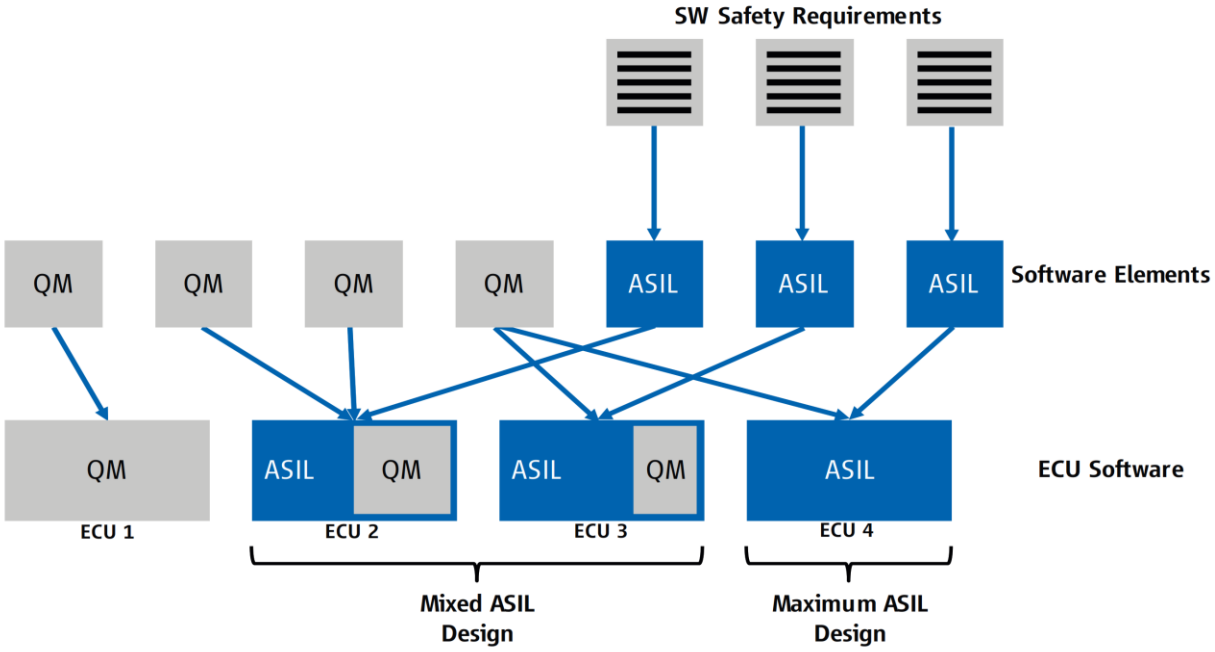

Figure 1: Mapping of software safety requirements to ECUs

Necessary integrity can be achieved in two ways:
One is to prevent that the software contains errors which lead to a malfunctioning behavior. Another is to include technical measures that are able to detect and control such a malfunctioning behavior.

In a “Mixed ASIL Design” the elements do not all have the same integrity based on their specific development goals. If they are integrated into one software without further measures, the integrity of the complete software cannot exceed that of the element with the lowest integrity, like the weakest link of a chain.

To achieve a higher degree of overall integrity one must provide evidence that the elements with a lower integrity are not able to interfere with the elements of the target ASIL which is called “achieving Freedom from Interference”. There are two principles to argue “Freedom from Interference”:

- Detect that an interference has occurred and mitigate the effects
- Prevent that an interference occurs

Detection and mitigation is sufficient if the resulting (degraded) functional behavior of the software can still ensure “Functional Safety” (e. g. achieve and maintain a safe state).

In a “Maximum ASIL Design” all elements have the same integrity. When integrating such elements, in principle the complete software has the same integrity and does not require an examination for Freedom from Interference. Nevertheless, the safety analysis at software architectural level may reveal weaknesses which have to be addressed (e. g. by technical measures) in order to achieve confidence in “Functional Safety”.

The following sections describe the two approaches in further detail. Since software architectures according to AUTOSAR are more and more used it is mentioned which contribution AUTOSAR features could provide.

5.2 “Mixed ASIL Design”

A “Mixed ASIL Design” targets the development of software elements according to QM or a lower ASIL without jeopardizing the integrity of the entire software system, which may have a higher ASIL. It may also enable the containment of errors in a partition.
This concept requires a suitable software design on application level, i. e. functional blocks must be coherent and unwanted interlinking between functional blocks (e. g. via global variables) should be avoided. It also requires a safety mechanism realizing the freedom from interference on hardware and software level which ensures that a software element with a lower ASIL cannot interfere with a software element with a higher ASIL. This mechanism must be able to either prevent that a malfunction of one element leads to the malfunction of another element, or it must be able to detect such interference and to mitigate the effects in time. This safety mechanism has to be developed according to the “Maximum ASIL” of the software safety requirements realized on this ECU.

ISO 26262 mentions different aspects of possible interferences:

1. Memory, which includes the RAM as well as the CPU registers
2. Timing and executions, which refers to blocking of execution, deadlocks and livelocks or the incorrect allocation of execution time in general
3. Communication, summarizing all possible errors that could occur in the communication between software elements both within the ECU and across ECU boundaries.

The separation between “QM or lower ASIL” and “Maximum ASIL” elements provides the following benefits:

- Development methods for “Maximum ASIL” only have to be applied for safety-related software elements (which includes the elements ensuring the freedom from interference). This allows the reuse of existing QM software (e. g. third-party software), as long as it is not safety-related.
- Propagation of failures between software elements of the same ASIL can be prevented or detected, although it is not mandated by Freedom from Interference. However, this also supports the separation of safety-related parts with high availability requirements from other parts in fail-operational architectures.
- Some failures caused by hardware defects can also be prevented or detected (e. g. timing supervision will detect a faulty clock source).

On the other hand, the following disadvantages have to be taken into account when applying the “Mixed ASIL Design”:

- The separation adds additional complexity to the software design. Especially in legacy software safety-related and non-safety-related functional blocks are often tightly coupled, which requires additional effort for a software architecture redesign.
- The safety mechanism to ensure “Freedom from interference” may result in a performance penalty during runtime (e. g. for reprogramming the MPU and context switching). To reduce these penalties to a minimum, the interaction between the software elements that are separated by freedom from interference mechanisms needs to be as low as possible.

5.3 “Maximum ASIL Design”

The “Maximum ASIL Design” has its advantages in use cases where a high share of the software provides safety-related functionality. In this approach, both the safety-related and the non-safety-related functions follow the development process of the highest ASIL in the system. For the non-safety-related software elements, the coexistence argumentation follows a process argumentation: if those software elements are developed in the same stringent way applying the same process methods as the safety- related software elements, the coexistence of the elements is possible without further technical separation measures. The only difference between the non-safety-related and the safety-related software elements is then the necessary safety analysis for the latter.

Compared to the “Mixed ASIL Design” this approach gives the following benefits:

- No additional complexity for development of a partitioning concept.
- No performance penalty due to safety mechanisms ensuring Freedom from Interference.
- Improved quality also for the non-safety-related software components which leads to a higher availability of the system.

On the other hand the following disadvantages have to be considered:

- The development effort increases since all software elements have to be developed according to the highest ASIL. For the non-safety-related part an additional safety requirement is then applied, which requires the non-interference (“silence”) with the safety-related part.
- As ASIL development does not mean that the software is error free, errors in these parts are not prevented to propagate by design.
- Inclusion of third-party software (e. g. “blackbox” software) is more difficult, as the development process of these modules is often unknown or cannot be influenced.

5.4 Mechanisms to realize freedom from interference

The following paragraphs contain suggested protection mechanisms for different kinds of fault classes in the data and control flow domain, which includes faults listed in Annex D of ISO 26262 part 6. Data faults are either related to global data, to data residing on the execution stack, or to data received by QM software components (SWCs). Additionally, hardware register faults constitute a special kind of data faults. Control flow faults are either related to timing faults or to interrupt faults. Faults due to illegal references can have an effect on either the data or the control flow domain.

Please note: The following list includes mechanisms sufficient for typical ASIL A or B projects, but it also shows additional mechanisms that can also be used for higher ASILs. Especially those mechanisms required for higher ASILs are typically supported by AUTOSAR Basic Software features.

Fault class: “Global Data Faults”

There are several options to address this fault class:

1. By partitioning the available RAM memory space in QM and ASIL parts and cyclically verifying a memory marker in between (initialized to a specific pattern), the probability to detect a relevant buffer overflow originating in QM software is increased.
2. To protect safety-related data without using an MPU, double inverse storage concepts can be employed to detect accidental overwrites by QM software by comparing the original variables to bit-inverse shadow copies upon reading or cyclically (as long as the fault tolerance time is considered). If a larger set of data is not written frequently, memory-efficient checksums can be used to detect accidental modifications of data parts. This protects against QM data pointer corruptions and QM buffer overflows, both resulting in writes to ASIL data.
3. To protect against accidental overwrites the CPU’s memory protection unit (MPU) can be used together with an allocation of tasks to separate partitions. In AUTOSAR, it is the responsibility of the Operating System to handle the MPU and thereby to ensure a proper separation between the entities. This is typically required for ASIL C and D but can also be useful or even required for lower ASILs.

Fault class: “Stack Faults”

There are several options to address this fault class:

1. By using a stack range check that checks whether the current stack pointer is in range of the allocated stack memory, the probability to detect a stack overflow or underflow by QM software modifying the stack pointer can be increased. Such a stack check can be implemented cyclically or – in most cases even better – in context of a task switch.
2. Additionally, stack overflows and underflows can be detected by checking memory markers (initialized to a specific pattern) placed above and below the allocated stack memory, which detects a subset of stack faults. This feature is also part of the AUTOSAR Operating System. Please be aware that this mechanism cannot detect stack overflows that do not overwrite the memory markers.
3. The stack can also be protected by a hardware MPU which actually prevents all stack faults. This is typically required for ASIL C and D but can also be useful or even required for lower ASILs.

Fault class: “Less Reliable QM Data Quality”

If data that is relevant to safety-related ASIL calculations is routed through QM software parts (e. g., drivers or communication stacks that process hardware input) that could corrupt data, there are several options to address this:

1. A single sporadic fault can be detected via a plausibility check. Such a plausibility check can use either values from other sources or previous values from the same source as an additional input. For instance, receiving a speed value of 0 km/h after having received one of 100 km/h in the previous CAN message 20 ms before is not plausible. Please note that the detection probability depends strongly on the assumed fault model.
2. Alternatively, and with a higher detection probability, end to end protection checksums and signal alive checks can be used. The AUTOSAR end-to-end protection modules have been specified for this purpose.

Fault class: “Hardware Register Faults”

To protect against QM software parts accidentally modifying hardware register state that is safety-related, there are several options:

1. Some microcontrollers offer locks for selected configuration registers or configurable write-once semantics, which should be used.
2. A cyclic check of the current hardware state against the expected state as held in software can be performed to detect faults as long as the fault tolerance time is considered.
3. Use a pro-active recovery mechanism that periodically rewrites the expected register states (assuming single bit flips as fault model).
4. The strongest mechanism is the protection of memory mapped registers via the MPU. Some CPUs also provide a Peripheral Protection Unit for this task. This is typically required for ASIL C and D but can also be useful or even required for lower ASILs.

Fault class: “Timing and Execution Faults”

To protect against QM software significantly delaying or even blocking ASIL software execution, there are several options:

1. Hardware or software watchdogs can be used. These should either be configured in a window mode, or they should regularly be triggered at the end of its deadline to detect delays as early as possible.
2. Depending on the scheduling scheme employed in the basic software operating system, overflows of time slices or task overruns can be detected. This is also a feature of the AUTOSAR Operating System.
3. The strongest mechanism that also detects fault in the program logic is the supervision of the program flow in combination with time stamps. This is also a feature of the AUTOSAR Watchdog Stack and is typically needed only for ASIL C and D.

Fault Class: “Interrupt Faults”

To protect against the fault that global interrupts or ASIL interrupt sources are permanently disabled by QM software parts, both properties can be checked cyclically to be enabled in an assertion.

To protect against QM Interrupt Service Routines executing at higher rate than expected, which will delay or even block the execution of ASIL ISRs, two measures can be taken:

1. If possible, from the real-time scheduling point of view, ASIL ISRs should be given a higher priority compared to QM ISRs.
2. As a monitoring measure, the arrival rate of QM ISRs can be monitored to be in range of the expected rate. This is also a feature of the AUTOSAR Operating System.

Fault class: “Illegal References”

By referencing ASIL symbols, QM software could include code that writes to protected ASIL data or executes protected ASIL functions. This misbehavior can be protected against by partitioning the software in the design phase. By explicitly denoting ASIL data and function declarations that are legal to be referenced from within QM software parts in an ASIL/ QM interface header, this design by contract can be proven in an automated way. An example approach would be to implement the interface header in a dummy module and link it to the QM software parts. The linker will then report undefined references from QM to ASIL software parts, which states an illegal interference. This proof is especially important when integrating QM third-party code, and the explicit interface can additionally be used to integrate plausibility checks when transitioning from/to QM software (see also fault class “less reliable QM data quality”).

6 Safety Analyses on Software Architectural Level

6.1 Introduction

A safety analysis on software architectural level is required by ISO 26262-6:2018 Clause 7.4.10. There is only little information on how to perform such an analysis. There are only few requirements the analysis needs to fulfill that are specified in ISO 262629:2018 Clause 8. Annex E of ISO 26262-6:2018 explains the application of such an analysis. The following section intends to give guidance on how a safety analysis on software architectural level can be performed. Moreover, the suggested methodology is visualized in an example.

ISO 26262 defines the purpose of the safety analysis on software architectural level as to:

- Provide evidence for the suitability of the software to provide the specified safety-related functions and properties with the integrity as required by the respective ASIL,
- identify or confirm the safety-related parts of the software and
- support the specification and verify the effectiveness of the safety measures.

A safety analysis on the software architectural level is intended to complement analyses on the hardware level and on the system level.

The software architectural level is defined in ISO 26262-6:2018 Clause 7.4.5. It comprises the static and dynamic aspects of the interaction of software components. The static aspects describe the hierarchy of software components, and the interfaces and dependencies between them. Usually the static aspects are modelled using e. g. a UML Class Diagram. The dynamic aspects should depict the data and control flow between software components. Usually, the assignment of functions of software components to tasks and partitions is defined. Dynamic aspects can be modelled using e. g. a UML Sequence Diagram.

The software architecture does not describe the inner processing of software units. This is part of the detailed design of a software unit. During the safety analysis on the software architectural level, only the externally visible failure modes of the inner workings of a software unit are considered. It is not the intention of the presented methodology to analyze the details of a single software unit. Code level safety analyses are not considered appropriate since the effect of faults on unit level can well be analyzed on architectural level.

The software architecture is a prerequisite for the safety analysis on software architectural level. It is typically performed in iterations: An initial software architecture is available, and the safety analysis is performed possibly leading to improvements of mechanisms and architecture. The safety analysis is then updated in turn with the software architecture. Care must be taken that assumptions and conclusions in the safety analysis are not invalidated by changes in the software architecture.

6.2 Methodology

The proposed safety analysis on software architectural level comprises the following steps:
1. Identify the software safety requirements for the elements of the software in scope.
2. Identify failure modes of the elements of the software.
3. Identify the impact on each allocated safety requirement.
4. Identify the potential causes for the failure modes.
5. Identify, analyze and improve safety measures

6.2.1 Software Safety Requirements and Elements in Scope

After all prerequisites are met, define the scope, i.e. the software safety requirements and the elements of the software architecture that are subject to analysis. Elements of the software architecture are typically software components and units, i.e. parts of software that are grouped together to achieve a defined functionality.

6.2.2 Failure Modes

The failure modes of an element of the software architecture depend on the element itself. Failure modes should be identified using guide words that have been adapted to the software elements. ISO 26262-6:2018 Table E.1 already suggests a basis for such guide words (“too late”, “too early”, “too often”, “too rare”, “not at all”, “out of sequence”, “unintended activation”, “stuck-at”, “too high”, “too low”). These kinds of guide words are helpful for data driven control applications, like many automotive applications are.

The failure modes should be described as precise as possible, e. g. instead of “Signal XYZ is too high” use “Signal XYZ is more than A”, where A is a value above which a different behavior of the software is expected. Completeness of the failure modes of a software element must be judged by the experts performing the safety analysis. The set of guide words supports achieving completeness of failure modes.

6.2.3 Impact on Safety Requirements allocated to the Software

Typically, safety analyses on system and hardware level use FMEA-like methods with a risk priority number (RPN) or similar mechanism. However, this assumes a stochastic model when things fail. This is valid for hardware, because it wears out over time. Software does not fail with a certain probability. A fault in a software element either leads to the violation of a safety requirement or it does not. As a first rule, if the failure of a software element violates a safety requirement, a measure must be implemented.

6.2.4 Potential Failure Causes

For each failure mode the causes that could lead to this failure mode must be documented. Knowing the potential cause of a failure mode helps to identify appropriate mitigations, e. g. a software implementation fault may be mitigated via a special code review (see ISO 26262-9:2018 Clause 8.4.9).

6.2.5 Safety Measures

There are typically different kinds of safety measures that aim to mitigate the failure of a software element:

- **Additional safety mechanism**
  Adding a safety mechanism is the technical solution to an issue detected during the safety analysis.
- This might comprise the creation of a new software safety requirement. Example: Add check in component XYZ to limit value DEF.
- **Requirements on the size, complexity and development process of a software element**
  Sometimes there is no adequate mechanism possible and the software element must work as specified. This is considered arguable if the element is limited in its size and complexity. No exact limits of size and complexity are provided here, since this is a decision that must be made based on corporate standards, customer collaboration and/or external assessment within a project. See also the SteeringColumnLockSwitch component in the example below.
- **Requirements on the user of the software**
  Some issues that are detected during the safety analysis on software architectural level, cannot be resolved on this level. They need to be addressed to the user of the software. The user might e. g. be the user of a software safety element out of context or the system incorporating an ECU running software.

6.2.6 Common Pitfalls

When performing a safety analysis on software architectural level there are some common pitfalls that should be avoided:

- **Safety mechanism for safety mechanism when iteratively performed**
  If the safety analysis is performed in iterations and additional safety mechanisms have been introduced in the first iterations, care must be taken:
  - Not to introduce additional safety mechanisms when analyzing existing safety mechanisms, and
  - to show what is added in the analysis for an increment.
- **Too detailed analysis**
  The safety analysis on software architectural level does not replace a detailed verification of the detailed design and code of a software component. It focuses on the interfaces between software components and units.
- **Inconsistent software architecture**
  The safety analysis on software architectural level is usually performed on a model, e. g. a UML model. It must be ensured that this model is consistent with the implemented software architecture. This consistency check is out of scope of the safety analysis.

6.2.7 Example

Figure 2 exemplarily shows a safety requirement and derived software safety requirements for a steering column lock. For this example, it is assumed that random hardware faults (incl. transient faults) are covered by adequate hardware mechanisms like lock-step CPU and memory with ECC protection.

At first the static part of the software architecture is described together with the allocated requirements (see Figure 3).

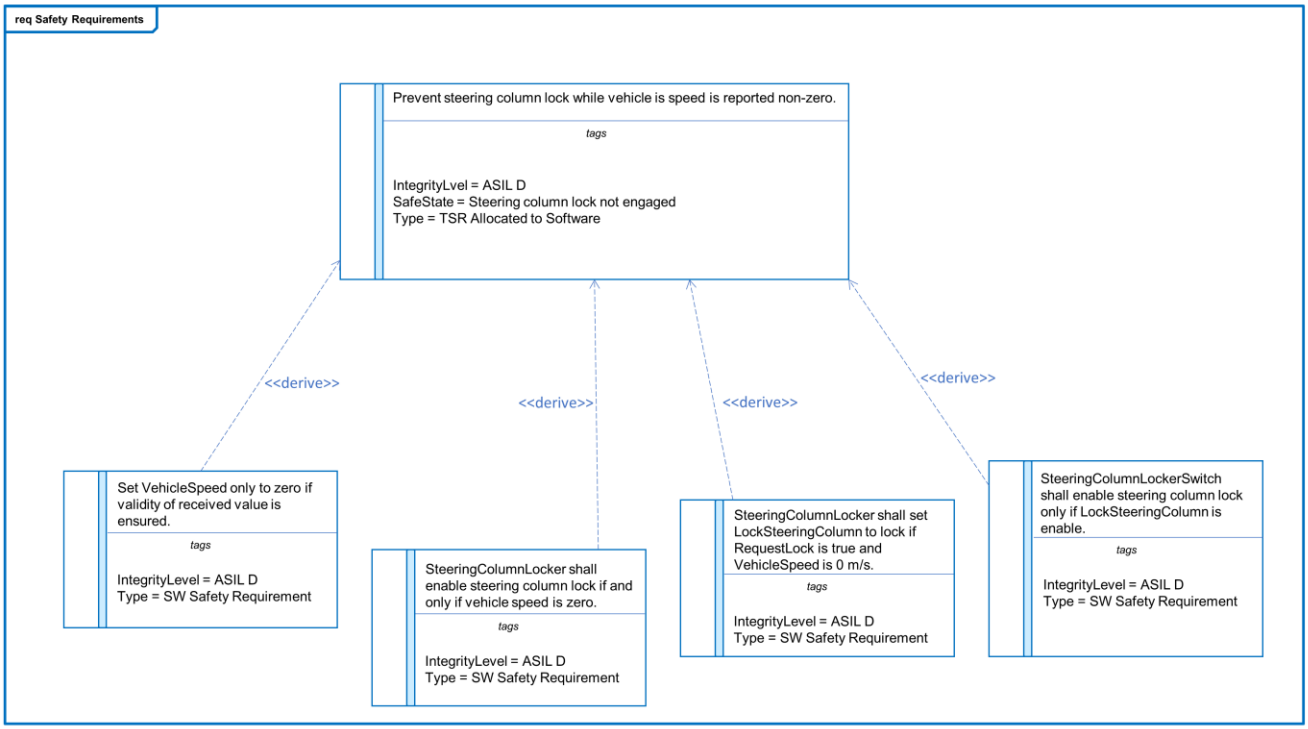

Figure 2: Safety Requirements

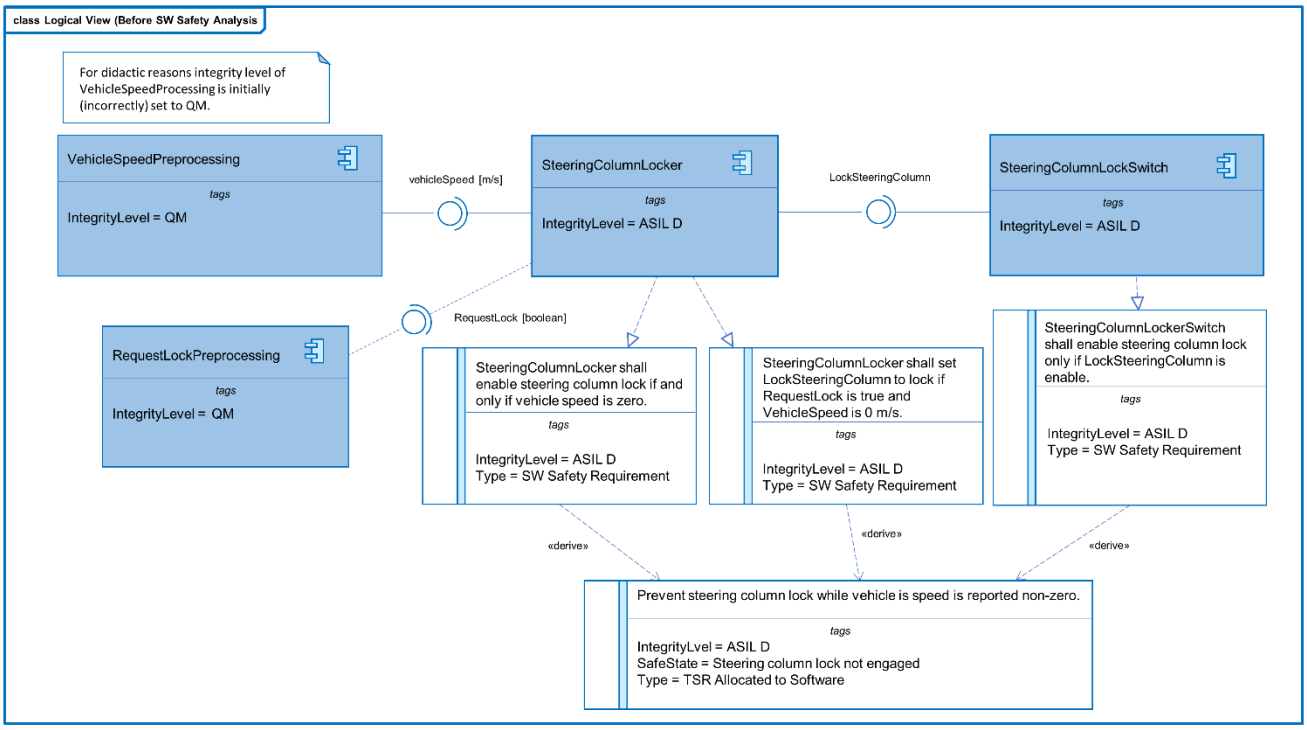

Figure 3: Logical View (Before SW Safety Analysis)

Vehicle speed is received via the in-vehicle network from a different ECU. Direct I/O by software is not depicted for simplicity reasons. However, if software has interfaces to the hardware, the Hardware-Software Interface (HSI) provides valuable input to the software safety analysis. In this case, random hardware faults must be considered in detail, e. g. bit flip of a digital I/O register value.

In a second step the dynamic parts (see Figure 4) of the software are described. For simplicity reasons the complete software is executed on a single, periodic task without any interrupts. Real systems are usually way more complex in their dynamic behavior.

As the software architecture is now complete, the safety analysis on software architecture can be started. For each element of the software architecture the failure modes are evaluated. Each failure mode is then evaluated in the context of each safety requirement (see Table 2). If a negative impact is detected, a safety measure is defined (see Table 3). It is suggested to track defined measures in the planning tool used, and only reference the item from the safety analysis. If a safety measure is a development-time measure, it should be specific for the respective failure mode, e. g. name a concrete test case that verifies that a certain failure mode does not exist.

A table-based form of documentation was chosen for this example. However, other forms may be applicable as well.

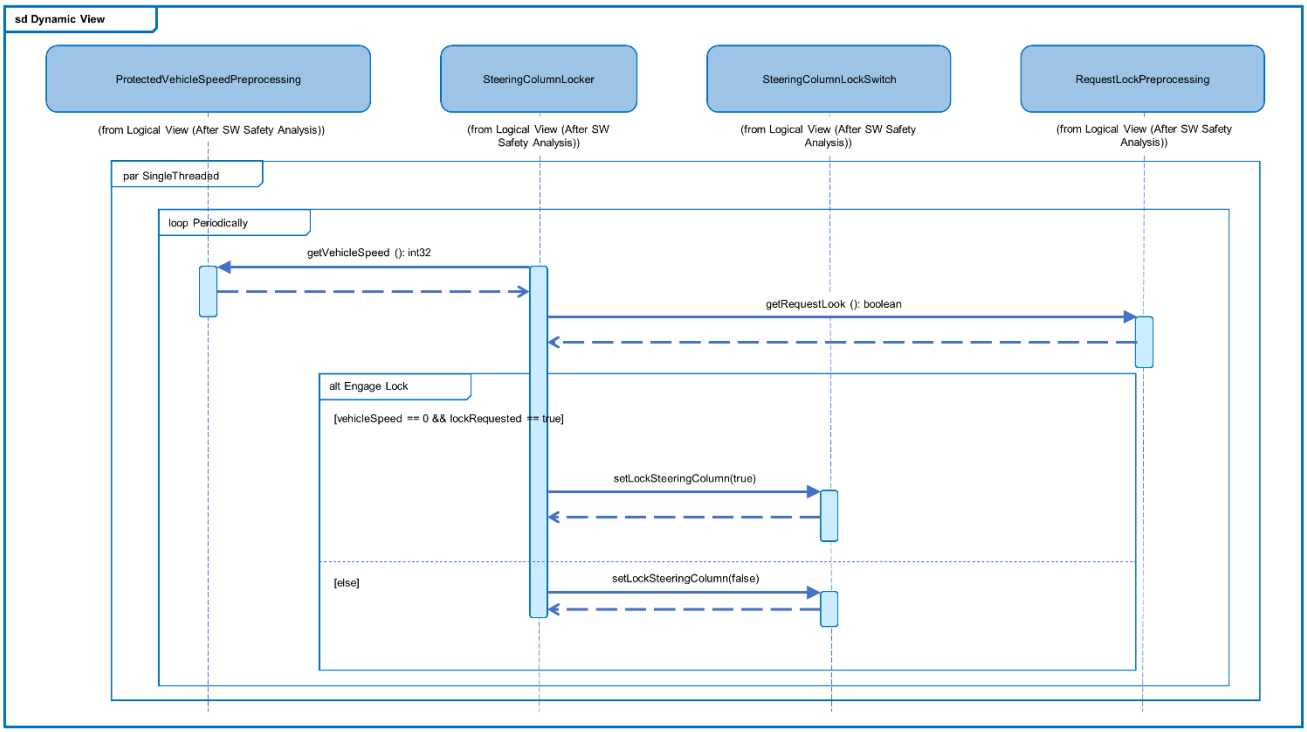

Figure 4: Dynamic View

Safety Requirement	Software Architectural Element	Failure Mode	Effect & Rationale	Impact Without Safety Measure	Potential Cause of Failure Mode	Safety Measure
SR1	RequestLockPreprocessing	RequestLock is unintendedly true	SteeringColumnLocker engages steering column lock only if vehicle is not moving and thus in a safe state	Safe	• Systematic fault in the element itself • Invalid input provided to element	-
SR1	RequestLockPreprocessing	RequestLock is unintendedly false	Steering column lock is not engaged	Safe	• Systematic fault in the element itself • Invalid input provided to element	-
SR1	VehicleSpeedProcessing	VehicleSpeed is zero even though real vehicle speed is not zero	The steering wheel lock switch is engaged even though VehicleSpeed is not zero	Unsafe	• Systematic fault in the element itself • Invalid input provided to element	SM1
SR1	VehicleSpeedProcessing	VehicleSpeed is not zero even though real vehicle speed is zero	Steering column lock switch is not engaged	Safe	• Systematic fault in the element itself • Invalid input provided to element	-
SR1	SteeringColumnLocker	LockSteeringColumn is unintendedly locked	Steering column lock switch is engaged even though VehicleSpeed is not zero	Unsafe	• Systematic fault in the element itself • Invalid input provided to element	SM2
SR1	SteeringColumnLocker	LockSteeringColumn is not locked even though intended	Steering column lock is not engaged	Safe	• Systematic fault in the element itself • Invalid input provided to element	-
SR1	SteeringColumnLockSwitch	SteeringColumnLockSwitch is unintendedly locked	Steering column lock switch is engaged even though VehicleSpeed is not zero	Unsafe	• Systematic fault in the element itself • Invalid input provided to element	SM3
SR1	SteeringColumnLockSwitch	SteeringColumnLockSwitch is not locked even though intended	Steering column lock switch is not engaged	Safe	• Systematic fault in the element itself • Invalid input provided to element	-

Table 2: Example Software Safety Analysis

ID	Safety Measure
SM1	Add new SW Safety Requirement to VehicleSpeeedProcessing: VehicleSpeedProcessing must only pass VehicleSpeed zero if end-to-end protection (incl. sequence counter and CRC) check passed. This safety mechanism must be low complex and developed and tested according ISO 26262 ASIL D requirements for this failure mode. Signal must be provided with ASIL D at input interface => feedback to system level. (This mechanism also covers random hardware faults that are not in scope of this analysis.)
SM2	SteeringColumnLocker must be low complex and developed and tested according ISO 26262 ASIL D requirements for this failure mode.
SM3	SteeringColumnLockSwitch must be low complex and developed and tested according ISO 26262 ASIL D requirements for this failure mode.

Table 3: Example Safety Measures

The safety analysis leads to a new software architecture (see Figure 5 - changes in red) implementing an additional software safety requirement “Set VehicleSpeed only to zero if validity of received value is ensured.”. Assurance here could e. g. be achieved using end to-end protection of the vehicle speed signal. The safety analysis has also confirmed the sensible allocation of the other software safety requirements.

The methodology presented above is considered to be compliant to ISO 26262-6:2018 Annex E.

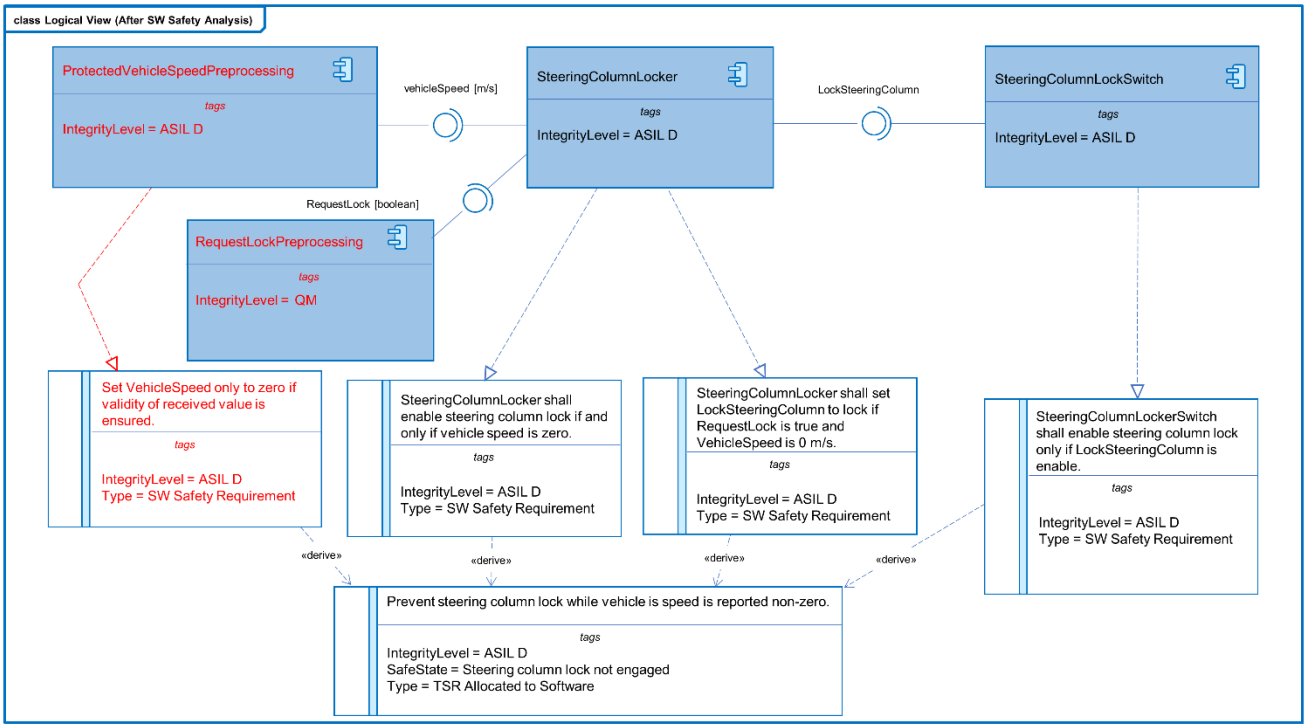

Figure 5: Logical View (After SW Safety Analysis)

7 Usage of Safety Elements out of Context (SEooC)

In software development, a safety element out of context (SEooC) is a generic software component that fulfills safety requirements of a system, although it has been developed without knowledge of the final system. Its development is based on assumptions on the use-cases it can fulfill and on the context it will be integrated into. Consequently, the incorporation of a SEooC into system development is a non-trivial task. This chapter outlines:

- What a SEooC is,
- what the properties of a SEooC are, and
- how a SEooC can be incorporated into a project.

Although hardware SEooCs are also in scope of the ISO 26262, this guideline focuses on software as SEooC. This section only discusses the usage of a SEooC, the development of a SEooC is out of scope.
The SEooC is described in ISO 26262, Part 10, Clause 9.

7.1 Definition of a SEooC

A SEooC is a stand-alone element that implements assumed safety-related functionality and that is intended to be used in different safety-related systems. It can thereby be a system, a sub-system, an array of systems or a hardware or software component. As it is developed out of context, i.e. not in the scope of the target system, it is not an item according to ISO 26262. Examples for typical SEooCs are microcontrollers or generic software components like AUTOSAR components or protocol stacks, which can provide safety-related functionality that can be reused without modification in different projects and systems.

As the developers of the SEooC do not know the precise context of the target system, they employ assumptions on the SEooC's usage and the environment in which it is going to be integrated. Therefore, safety requirements need to be assumed as input for the SEooC development. These requirements are documented and provided to the SEooC user as "assumed safety requirements".

In addition, during development, the SEooC provider may assume conditions to be fulfilled by the SEooC user. All such assumptions must be communicated to the SEooC user as well, e. g. as so called "assumptions of use". The fulfillment of the assumed safety requirements of the SEooC is only given if all the applicable assumptions of use are satisfied. The SEooC provider lists the "assumed safety requirements" and the "assumptions of use" in the safety manual of the SEooC.

The safety case for the SEooC, i.e. the argumentation and evidences for the fulfillment of the assumed safety requirements, are created by the SEooC provider.

7.2 SEooC properties

The assumed safety requirements that a SEooC provides and implements have an ASIL allocation designated from the SEooC provider. This means that the SEooC user can rely on the specified functionality up to the defined ASIL and that the SEooC provider performed the necessary verification and analysis activities with the required rigor. However, this is only valid if the SEooC's definition of the "assumed environment" matches the target system where the SEooC is going to be integrated.

The safety manual is an important collateral to the SEooC and must be carefully considered by the SEooC user. It documents the assumed safety requirements and the assumed target environment. The safety manual defines how the SEooC must be integrated into the target system and the necessary duty of care so that the assumed safety requirements are ensured.

From the target system's perspective, a SEooC may bring non-required functionality, which the SEooC provider developed according to the specified ASIL, although it is not used in the user's project. This can be seen like configurable software.

The appropriate usage of the SEooC usually requires effort on user side during integration, e. g. execution of verification measures defined in the safety manual. This might be easily missed when considering the use of a SEooC.

7.3 How to use a SEooC in a project

How can a SEooC then be integrated into the target system? The following steps are mandatory for a successful and safe integration:

- The SEooC user must verify that the SEooC's assumed safety requirements match the specific safety requirements that have been derived from the system context.
- The SEooC user also must ensure that the SEooC's assumptions of use are met.
- If both requirements cannot be achieved, the SEooC is either unsuitable for the target system or additional safety measures must be implemented.
- Further safety requirements that cannot be fulfilled by the SEooC must be addressed within the project scope.
- To ensure safe operation of the SEooC, the SEooC user must adhere to the instructions of the safety manual. Violations of these instructions must be justified within the project scope.

The flow of these steps is illustrated in Figure 6. From the system's safety goals, functional and technical safety requirements are derived. Typically, this happens in various steps, Figure 6 shows a simplified view of this process. There are usually also requirements which are not safety-related. In the end, there is a set of software safety requirements and nonsafety related software requirements. In Figure 6, the safety-related requirements are represented by the green circles and the non-safety related requirements by the blue circles. The SEooC user must match the software safety requirements to the SEooC's assumed requirements. It can happen that the SEooC provides functionality which is not needed by the system. Typically, additional safety functionality must be implemented in the software application as it is not provided by the SEooC. It is the SEooC user's responsibility to ensure that the combination of the software SEooC and the remaining application do not violate the system's safety goals.

7.4 SEooCs and deactivated code

As a SEooC is developed without any knowledge about the target systems and builds on a set of assumed requirements, it is common that a SEooC contains code and functionality that is qualified for usage in safety-related systems, but not necessarily needed by the SEooC user's requirements. There are now two contradicting viewpoints:

- From a user's perspective, this may be undesired functionality.
- From the SEooC provider's perspective, this is desired and qualified functionality.

Yet, ISO 26262 requires that this situation is dealt with: it states if during integration "[…] deactivation of these unspecified functions can be assured, this is an acceptable means of compliance with requirements." (ISO 26262 -6 Clause 10.4.6).
How can the deactivation be ensured? Removal of the deactivated code, as for example necessary for aviation projects:

- Leads to a specific version for the SEooC user's project, where certain functions are removed
- Loses the qualification aspect of the SEooC's usage in various projects with different use cases

If code removal is not an option, the SEooC user can apply the following methods, which to a large extent are in the standard repertoire of safety development methods, to ensure that deactivated code is not called:

- Control flow monitoring,
- ensure on application level that only necessary interface functions are called and that parameters passed to used functions are correct,
- use dead code elimination by the compiler e. g. to ensure that unneeded library functions are not included into the binary,
- employ coverage analysis to verify that only intended interface functions of the SEooC are used.

In general, the SEooC user must argue that the situation is known and that the benefit of using a SEooC is higher than an individualized version. It must be ensured, for example by above listed methods, that the unneeded functionality is not used by the application.

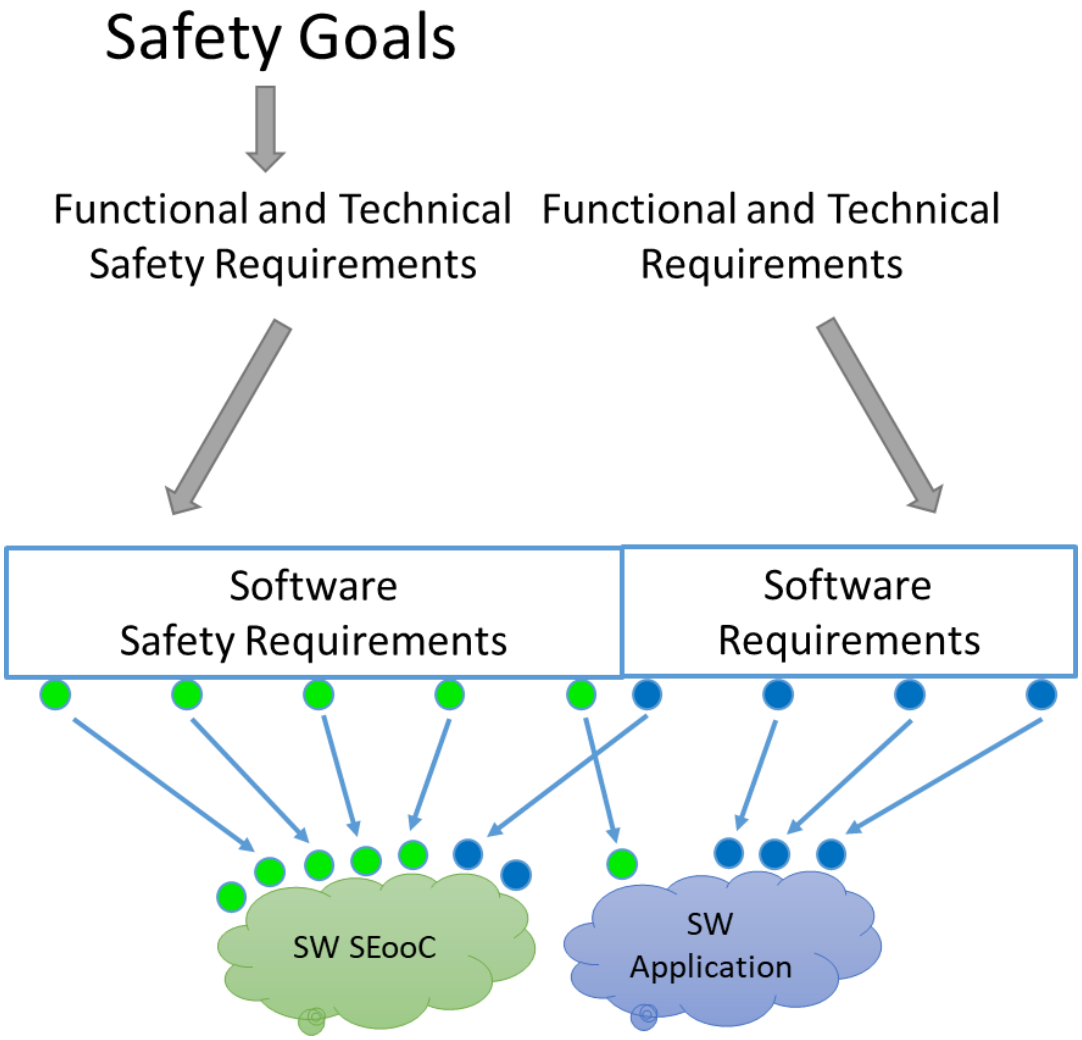

Figure 6: This figure illustrates schematically the mapping of system safety requirements to the assumed safety requirements of a software SEooC.

8 Confidence in the Use of Software Tools

8.1 Motivation

Software tools play a major role in the implementation of processes and methods used during the development of safety-related systems, software and hardware.

Using tools can be beneficial because they enable, support or automate safety-related development activities (e. g. development and management of requirements or architectural designs, code generation, analyses, testing or configuration management).

However, in case of a malfunctioning behavior such tools may also have adverse effects on the results of tool-supported development activities and thus on the “Functional Safety” achieved in the final product or its elements including software.

ISO 26262 provides an approach to achieve confidence that using software tools does not jeopardize “Functional Safety”. This approach contains:

- Determination of single tools or tool chains which are relevant for safety-related activities and identification of the used functionalities and their purpose during development.
- An analysis to determine the required confidence for each relevant software tool, based on the risks related to the used functionalities and its role in the development process (“classification“).
- Measures to qualify a software tool, if the classification indicates that this additional risk reduction is needed.

8.2 Analysis and classification of software tools

This approach can be supported by the tool vendor, e. g. by providing information such as generic analyses based on intended application use cases or test cases and test suites for tool qualification. The responsibility for using the tool in a suitable way remains with the user.

The following sections describe this approach in further detail.

The risk related to the tool functionalities used for a specific purpose during development is determined by the tool´s impact and the possibility to detect malfunctions yielding the aggregated tool confidence level (TCL):

1. The tool impact (TI) expresses the possibility that a malfunction of a particular software tool can introduce or fail to detect errors in a safety-related item or element being developed.
   - TI1: Shall be selected when there is an argument that there is no such possibility
   - TI2: Shall be selected in all other cases

2. The tool error detection (TD) expresses the confidence that due to tool-internal or tool-external measures (e. g. subsequent process activities) relevant tool malfunctions producing erroneous output can be prevented or detected.
   - TD1: High degree of confidence (that a malfunction and its corresponding erroneous output will be prevented or detected)
   - TD2, TD3: Medium or low degree of confidence

The classification depends on the usage of the tool (e. g. used functionalities) as part of the complete development process.

Figure 7 shows the approach and table gives some examples. Please note that the specific workflow embedding the tool usage has to be considered.

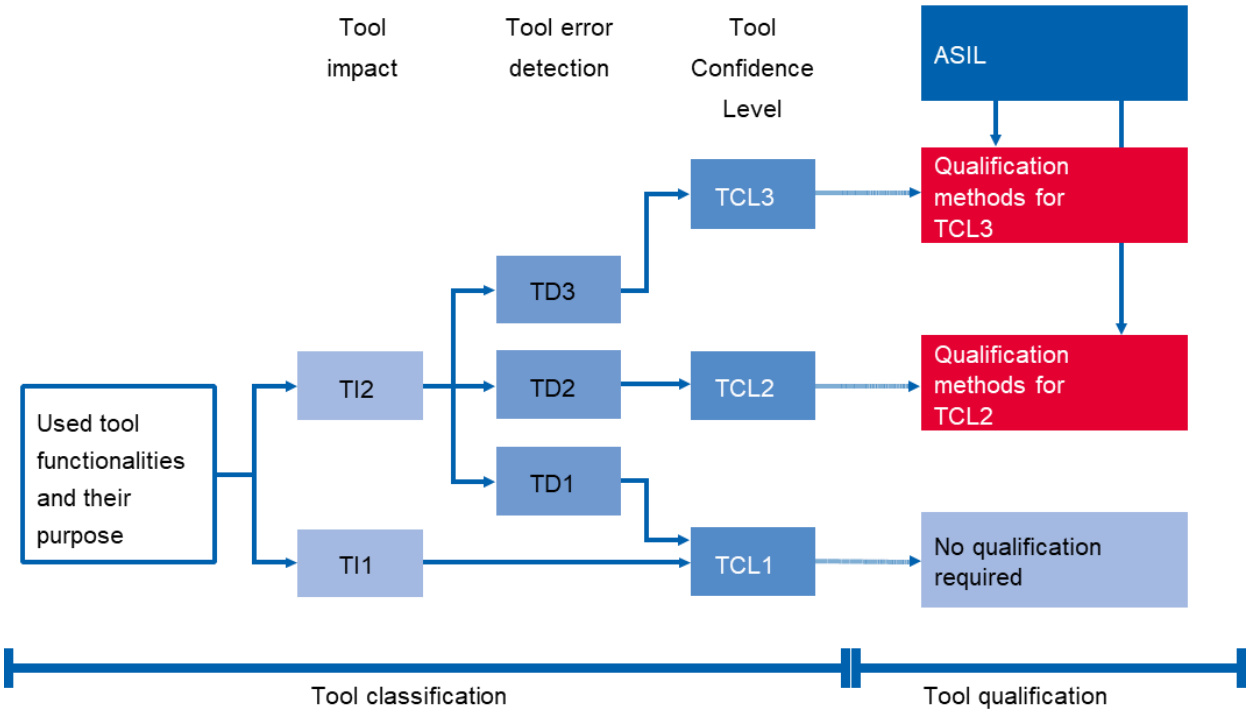

Figure 7: Classification and qualification of software tools acc. ISO 26262

Tool	Use case	Failure mode	TI	Measures to detect or prevent malfunctioning of tool		TD	Rationale	TCL	Qualification needed
C-Code generator	Generate C-Code from model	Incorrect translation from model to code	TI2	None		TD3	Errors are not detected if no systematic tests are performed.	TCL3	Yes (TCL3)
				Full verification of code with required coverage by tests, reviews and static code analysis		TD1	Errors are detected by verification.	TCL1	No
				Full verification of code with code generator specific checker tool		TD1	Errors are detected by checker tool.	TCL1	No
				Use redundant code generator and compare results		TD1	Failure of one tools will be detected by the other tool. Equal failure of both tools is unlikely	TCL1	No
Static code analysis tool	Static code analysis	False negatives with respect to specified error class (e. g. array out of bounds for a bounds checking tool)	TI2	None		TD3	Other tests do not focus on this error class	TCL3	Yes (TCL3)
Configuration management tool	Checkout specific artifact version	Checkout of wrong artifact version	TI2	Artifact checksum verified against external database		TD1	Corrupted data and wrong artifact version will be detected externally	TCL1	No
		Artifact was corrupted	TI2	Artifact checksum verified against tool internal database		TD1	Corrupted data will be detected internally	TCL1	No

Table 4: Examples for tool classification

8.3 Qualification of software tools

The resulting TCL may be reduced by improving the detection or avoidance measures (iterative tool analysis). As a consequence, alterations in the process (e. g. removal of a redundant tool in the tool chain) may invalidate the TCL argumentation.

Example: If an analysis shows that for the tool and its intended usage a TCL1 cannot be argued, there are at least two options:

- Lowering the TCL by improving the TD introducing additional detection or prevention measures into the development process (e. g. checking tool outputs) or into the tool itself.
- Performing a qualification of the tool according to the TCL for the target ASIL if lowering the TCL is not feasible or not efficient.

The quality of the documentation and the granularity of the tool analysis require an adequate level of detail so that the resulting TCL is comprehensible, and the resulting TCL can be justified (Neither a very detailed investigation nor a rough general view is helpful).

For TCL1 classified software tools no qualification measures are required at all.

For TCL2 and TCL3, tool qualification measures provide evidence that justifies confidence in a software tool for its intended use cases in the development environment. The following measures are applicable depending on the TCL and target ASIL:

- Increased confidence from use.
- Evidence for a structured tool development process.
- Tool development in compliance with a safety standard.
- Validation of the software tool.

9 Software Methods of ISO 26262:2018, Part 6

To ensure the application of a defined level of quality measures when developing safety-related software, ISO 26262 Part 6 defines objectives that need to be fulfilled for each part of the V-model. To assist in the fulfillment of the objectives, the authors of ISO 26262 have added tables with selected methods that can be applied to the artifacts. Depending on the ASIL, these methods are marked as either optional (o), recommended (+) or highly recommended (++).

Developers of safety-related software then have the task of selecting an appropriate set of measures to apply to their artifacts based on these tables. It is the developer's responsibility to identify the measures that are necessary to meet the objectives in the mandated rigidity for the target ASIL. Not all of the measures listed are suitable or appropriate for every type of software, so the developers must carefully decide which measures to apply and how.

Unfortunately, ISO 26262 Part 6 only lists the keywords of the measures without providing further details or references. In the following, we provide an explanation of all the methods described in the tables of ISO 26262-6:2018 to help developers of the safety-related software make an informed decision. We use the following fields for this purpose:

- **Method**
  identifies the method as given in ISO 26262-6:2018.
- **What is it?**
  explains the method and provides further details.
- **Reference to further details**
  gives links to other standards and documentation that provide more detailed information about the method, where possible. Note that this list is not exhaustive.
- **Examples**
  provides a list of examples how the method may be applied.
- **Benefit & Best practices**
  explains how the method can be used to improve software quality, support safety claims, and contribute to the achievement of the objectives. In addition, we may have added usage recommendations based on our experience in applying the method.

These explanations are intended to help safety managers, safety engineers and developers of safety-related software make informed decisions when defining and setting up their project. Please note that this is only informative, and that the responsibility for the proper selection and application of methods rests entirely with the developers of the safety-related software.

9.1 Table 1, Topics to be covered by modelling and coding guidelines

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Enforcement of low complexity	The implementation and code should be as simple and easy to understand as possible. Complexity describes the interactions between a number of entities. As the number of entities increases, the number of interactions between them would increase exponentially, and it would get to a point where it would be impossible to know and understand all of them.	- [9126] - [25023] - [HIS Metrics] - [def comp]	- Cyclomatic number (McCabe) - Program size (Halstead N) - HIS Source Code metrics	- Reduce complexity to a level understandable for a reviewer - The simpler the implementation, the less risky the functionality.
1b	Use of language subsets	Restrict the programming language to well-established patterns, avoid complex and error-prone language constructs. A language subset aims to improve one or more of the portability, safety and security aspects of a program.	- [MISRA] - [subset] - [61508] Part 7, C.4.2 - [AUTOSAR 839]	- MISRA C and C++ - avoid e.g. multiple inheritance in C++ - AUTOSAR	- Higher probability of fault free code - Application of established coding guidelines - Reduces the risk of tool incompatibilities - Avoid ambiguous or possibly undefined code constructs.
1c	Enforcement of strong typing	Usage of variables and programming languages with strong type definitions, i.e., the automatic conversion of	- [typing] - [61508] Part 7, C.4.1	- use explicit casts in languages like C and C++ for type conversion	- Avoidance of side effects - Avoid unexpected behavior

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
		variable types into other variable types should be avoided. Generally, a strongly typed language has stricter typing rules at compile time, which implies that errors and exceptions are more likely during compilation. Dynamically typed languages (where type checking happens at run time) can also be strongly typed. Most of these rules affect variable assignment, return values, function calling and constructor invocation.		- avoid programming languages that use weak typing (e.g. variable type is specified by the context in which the variable is accessed like in many scripting languages)	- Avoidance of complex behavior of programming languages, e.g., implicit type conversion - Enforce strong typing for all languages except Assembler - Use data types with clearly defined bit size.
1d	Use of defensive implementation techniques	Applications are suspicious of all inputs and prerequisites and are robust against violations. The implementation also handles error scenarios that could not be reached by design. Making the software behave in a predictable manner despite unexpected inputs or user actions.	- [def prog] - [61508] Part 7, C.2.5	- Plausibility checks of input values of functions - 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 - Always use the “default” in switch/case statements to detect an error	- Detection of some systematic and random faults of other elements of the design - Robustness against later design changes - Check plausibility of input values of functions at the interface of coherent software components, but not for every function - Be sure not to program too restrictive because the code may then not work correctly in some unexpected but correct constellations
1e	Use of well-trusted design principles	Usage of design principles that are well-established for the desired use-case. Well-trusted design principles are design principles that have previously been used without known safety anomalies in a comparable application.	- [61508] Part 7, C.2.10	- Hierarchical design - Modularization - Layered architecture	- Usage of design patterns that are known to work in similar scenarios - Avoid unusual designs that might be more error-prone, difficult to understand and to maintain - The use of AUTOSAR
1f	Use of unambiguous graphical representation	Usage of clearly defined graphical elements for defining a behavior or functionality in a consistent way. When using graphical representations, it must be clearly defined for all types of elements of this graphical representation what they mean.	- [UML]	- UML diagrams with a modelling guideline	- Improved comprehensibility of design documents and the desired functionality - The graphics is interpretable in the same way by different people. - The graphics is interpretable without the explanation by the author and even after some time. - Do not use too many kinds of symbols. - Make different graphics for different aspects, like e.g. for static and dynamic behavior. - Clearly state the meaning of boxes and arrows.
1g	Use of style guides	Programming style, also known as code style, is a set of rules or guidelines used for structuring and formatting the source code and the models for a computer program.	- [style] - [MISRA] - [AUTOSAR 207] - [AUTOSAR 839] - [61508] part 7, C.2.6 - [CERT C++] - [CERT C] - [UML]	- Indentation rules for programming languages - Structuring rules for source code - Guidelines for using UML elements - Style guides provided by OEMs like VW, Daimler or BMW - MISRA - AUTOSAR	- A particular programming style will help programmers read and understand source code conforming to the style and help to avoid introducing errors. - Consistency throughout the project. - Improved (code) readability. - The uniform design of source code and models makes it possible for the software to be maintained by different people over its entire life cycle. Personal preferences of programming are reduced. - Take over the style guide of your key customer - Define coding conventions
1h	Use of naming conventions	In computer programming, a naming convention is a set of rules for choosing the names to be used for identifiers which denote variables, types, functions, and other entities in source code and documentation.	- [naming] - [AUTOSAR 207]	- Hungarian notation which includes the variable type into the name - Use of component prefixes for programming languages with a global namespace - AUTOSAR	- Reduce the effort needed to read, understand and maintain source code (improved readability) - Reduce the diversity of names caused by different programmers - Take over the typical naming conventions for the programming language in use. - Define clear naming conventions for the different kinds of names
1i	Concurrency aspects	Considerations on simultaneous access of elements from various locations. Concurrent computing is a form of computing in which several computations are executed during overlapping time periods—concurrently—instead of sequentially	- [concurrent]	- Global variables that are accessed from multiple threads possibly also executed on different CPU cores - Undefined runtime behavior of multiple threads	- Prevention of race conditions - Ensure correct and consistent locking - Enforce clarity about concurrency - Consider concurrency aspects in design

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
		(one completing before the next starts).			

9.2 Table 2, Notations for software architectural design

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Natural language	Complete sentences in prose language describing the design.		- Descriptions - Explanations - Rationales	Natural language is intended in addition to other notations and enables the understanding of other notation forms.
1b	Informal notations	Description technique that does not have its syntax completely defined.	- [26262] Part 1, 3.80	- Illustrative figures and diagrams - Tables	Informal notation is used to describe design patterns in a loosely defined way to support the understanding. Often graphics are easier to understand than lengthy formulations in natural language. To improve comprehensibility, keep number of used elements low and a legend should explain the used notation.
1c	Semi-formal notations	Description technique whose syntax is completely defined but whose semantics definition can be incomplete.	- [26262] Part 1, 3.149 - [61508] Part 7, B.2.3 - [UML]	- UML - Decision tables - Constrained language	Semi-formal notation enables a common understanding by a defined description language and fosters modularity, abstraction and unambiguity. It enables consistency within the design and between design and implementation. Often semi-formal notations also provide a means for code generation.
1d	Formal notations	Description technique that has both its syntax and semantics completely defined.	- [26262] Part 1, 3.63 - [61508] Part 7, B.2.2 - [61508] Part 7, C.2.4.8 and C.2.4.9	- Z (Zed) - NuSMV - Prototype Verification System (PVS) - Vienna Development Method (VDM) - Mathematical modelling of algorithms	Formal notation enables unambiguous specification of safety requirements with the aid of a mathematical description. A formal model can be executed in a simulator or used for code generation and is a prerequisite for a mathematical proof of correctness.

9.3 Table 3, Principles for software architectural design

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Appropriate hierarchical structure of the software components	Hierarchical architecture considers the whole software as a hierarchy structure which is decomposed into logical components and/or units at different abstraction levels.	- [hierarchical]	- AUTOSAR Layered Architecture - software layers of ISO OSI network stack - Hardware abstraction layer	Hierarchical structuring improves the understandability of the architectural design. It avoids monolithic software which would be hard to comprehend. It thereby improves maintainability and verifiability.
1b	Restricted size and complexity of software components	Each software component described in the architectural design contains only a limited subset of the software's overall functionality. The components should be as simple as possible.		- one functionality, one component (single responsibility principle)	A restricted component size with low complexity allows the easy implementation of the software component. It thereby supports simplicity and comprehensibility of the software. It is also easier to verify and maintain.
1c	Restricted size of interfaces	The interfaces between software components should only contain the necessary amount of provided functions and parameters. Note that also global variables and shared memory constitute interfaces between software components.	[HIS metrics]	Number of function parameters <= 5	A restricted size of interfaces supports simplicity, verifiability, maintainability and encapsulation. It reduces the number of necessary test cases and test effort. Avoid different interfaces that provide access to the same functionality.
1d	Strong cohesion within each software component	Cohesion refers to the degree to which the elements inside a software component belong together. A software component with strong cohesion should only contain functionality for a single purpose or closely related purposes. An indicator for strong cohesion is data that is shared between its subcomponents and not needed outside. This method	[cohesion]	- shared data elements within the component - data elements are only visible within the component - component interfaces that support the same functionality	Together with loose coupling, strong cohesion fosters a clean modular structure and the encapsulation of components which reduces the possibility of error propagation. Strong cohesion improves the maintainability, robustness and reusability of components and enables a less complex software architecture.

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
		complements 1e by focusing on the inside view of a component.			
1e	Loose coupling between software components	Software components that are loosely coupled interact only via a minimum set of interfaces. A high degree of changeability inside components without impact on other components is an indicator for loose coupling. Large amounts of shared data elements between software components contradict loose coupling. This method complements 1d by focusing on the outside view of a component.		- components interact only via their specified interfaces - no shared data elements between components - no implicit scheduling dependencies between components	Together with strong cohesion, loose coupling of software components fosters a clean modular structure and encapsulation which reduces the possibility of error propagation. Dependencies between components are reduced which improves reusability and verifiability of the component and enables a less complex software architecture. A loose coupling supports maintainability by allowing internal refactoring without affecting other components.
1f	Appropriate scheduling properties	An appropriate scheduling ensures a deterministic execution of software components on the selected target hardware according to specification. This needs especially consideration of timing constraints, prioritization, preemption, CPU load, interrupts, and worst-case execution time of the executables.		- definition of appropriate scheduling sequence - selection of suitable scheduling algorithms depending on the use-case (e.g. priority-based scheduling, earliest deadline first, round robin, …)	A well-defined scheduling ensures a timely execution of a safety function within the time limits of the specification. If applicable, a simple time-sliced scheduling shows best predictability and verifiability. Event-based scheduling is possible as well, but it is more difficult to guarantee deterministic behavior.
1g	Restricted use of interrupts	Interrupts are hardware-triggered events that stop the program execution and switch to a pre-defined interrupt service routine. Usually, they are triggered by peripherals that request handling of a hardware event. To restrict the impact on the scheduling and deterministic behavior, the interrupt frequency, its execution time, and the number interrupt sources should be low.	- [61508] Part 7, C.2.6.5	- high-frequency or uncontrolled interrupt source could delay or block application	Restricted use of interrupts improves the predictability of the program execution by avoiding unexpected execution of interrupt service routines. It simplifies the program flow by reducing or avoiding concurrency problems. Interrupts need to have a clear and specified priority to ensure deterministic behavior in case multiple interrupts occur at once. Instead of interrupts, polling algorithms are an option in certain cases. If interrupts are necessary, their execution time should be low in order to return to the normal program flow as soon as possible.
1h	Appropriate spatial isolation of the software components	Software components are spatially isolated if code and private data of a software component cannot be altered by another software component. While for independent components with same ASIL a separated location of code and data in memory may be sufficient, mixed criticality systems (i.e. components with different ASIL) may need technical protection or detection mechanisms (e.g. memory protection unit, checksums).		- separation of software component's data and/or code via a memory protection unit (MPU) - checksums on critical data to detect unauthorized modification	Spatial isolation enables an argumentation for spatial freedom from interference which is a prerequisite for mixed-criticality systems. It enhances encapsulation by preventing or detecting incorrect memory accesses outside the allowed boundaries. It thereby limits a possible error propagation to other software components.
1i	Appropriate management of shared resources	Shared resources are hardware or software entities that are accessed by multiple software components. Concurrent access to shared resources must be managed e.g. by locking mechanisms or avoided e.g. by scheduling guaranteeing exclusive access.		- use of semaphores, mutexes, spinlocks or other locking mechanisms - encapsulate access to the shared resource via a dedicated component	Prevention of concurrent accesses to shared resources avoids inconsistencies, race conditions and hard to find errors. Any access to shared resources should be documented in the respective specification.

9.4 Table 4, Methods for the verification of the software architectural design

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Walk-through of the design	Documented systematic review of an artifact with at least two participants, the author and one or more reviewers. During preparation, the artifacts, the objectives and the review templates are shared to all participants. The outcome are findings that need to be addressed.	- IEEE 1028:2008, Ch.7 - [61508] Part 7, B.3.8	- peer reviews - four-eye principle - step-by-step explanation by the author	The goal of a walk-through is: - to identify anomalies regarding the compliance to the safety requirements - to verify conformance with the applied guidelines - to create a common understanding

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1b	Inspection of the design	Documented systematic review of an artifact with multiple participants following a formal procedure. The participants include an inspection leader who moderates the inspection, the author and one or preferably more expert reviewers. Each reviewer should typically focus on specific viewpoints and individually collects findings. In inspection meetings the participants consolidate the findings and define subsequent measures. During preparation, the inspection leader shares the input, the objectives and the review templates to all participants. The outcome are findings that need to be addressed.	- IEEE 1028:2008, ch.6 - [61508] Part 7, B.3.7 - [Fagan]	- verification of the artifact against pre-defined criteria by various subject matter experts	The goal of an inspection is: - to identify anomalies regarding the compliance to the safety requirements - to verify conformance with the applied guidelines - increased confidence in the review results compared to the walk-through due to the more rigid review process For intermediate versions of artifacts, it is not required that all quality criteria are fulfilled. Therefore, it may be reasonable to spend less effort for reviewing such artifacts by e.g. performing walk-throughs.
1c	Simulation of dynamic behaviour of the design	This method requires a formal model of the software that can be executed. Using this model, the specified dynamic behavior is simulated and verified.	- [61508] Part 7, B.3.6	- commercially available model-based simulation tools	The simulation of the formal model verifies it against the intended behavior as specified in the safety requirements. It allows to discover design faults early in the development phase. The simulation model may also be used for back-to-back comparison tests.
1d	Prototype generation	A prototype is a preliminary implementation of the intended software at an early development stage that not necessarily follows a rigid development process with defined quality goals. It is intended to serve first project goals like feasibility, proof of concept, user experience, requirements definition, etc.		- proof-of-concept implementations - comparison of alternative approaches	The goal of a prototype is to: - evaluate architectural design choices for suitability to fulfill the software safety and non-safety requirements - reduce risks for the final implementation of the software - allow early incorporation of e.g. customers and other stakeholders Prototypes are often used for new developments where no experience from previous solutions exists. It requires (significantly) less effort than implementation of the final software. Depending on the necessity, only certain aspects of the complete solution may be prototyped.
1e	Formal verification	This method is applicable if a formal specification of the software architectural design exists. As syntax and semantic of the specification are defined, this method is capable of verifying the correctness of the software architecture.	- [61508] Part 7, C.5.12	- execution of a formal model - model checking - mathematical proof	A formal proof is the strongest possible verification measure to identify anomalies regarding the compliance to the requirements.
1f	Control flow analysis	Control flow analysis is a technique to identify suspect control flows between architectural elements. The control flow between architectural elements must be adequate to implement the safety requirements.	- [61508] Part 7, C.5.9 - [UML]	- detect unintended recursions on architectural level - analysis of UML sequence, activity and component diagrams - incorrect invocation of a QM library from ASIL context	The control flow analysis helps to: - visualize the control flow graph to improve comprehensibility and validate correctness of the execution sequence - detect invalid mixture of ASILs in the control flow - detect architectural elements that are not accessed - detect architectural elements that should not be accessed as they could violate a safety requirement - identify unnecessarily complex control flow structures Sequence diagrams, activity diagrams and similar graphical elements displaying the sequential interaction between architectural elements support the control flow analysis. Component diagrams support detecting abandoned or incorrectly incorporated elements.
1g	Data flow analysis	Data flow analysis is a technique to identify suspect data flows between architectural elements. The data flow between architectural elements must be adequate to implement the safety requirements.	- [61508] Part 7, C.5.10 - [UML]	- analysis of incoming and outgoing data on component interfaces - incorrect usage of QM data in an ASIL context	The data flow analysis helps to: - visualize the data flow graph to improve comprehensibility and validate correctness of the data flow - detect incorrect or too low ASIL of input data - analyze the usage and modification of global data by architectural elements - detect concurrency issues in the data flow

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
					- identify unnecessarily complex data flow - detect unused data Sequence diagrams, activity diagrams and similar graphical elements can be annotated to also show the data flow supporting the data flow analysis. Class and component diagrams allow to identify the data structures and their relationship.
1h	Scheduling analysis	Scheduling analysis is a technique to identify suspect scheduling behavior of architectural elements. The scheduling between architectural elements must be adequate to implement the safety requirements. This needs especially an understanding of timing constraints, prioritization, preemption, CPU load, interrupts, access to shared resources and worst-case execution time of the executables.	- [MARTE]	- preemptive vs. non-preemptive tasks - decide polling vs. event-driven architecture - interrupt lock frequency and duration - worst-case execution time analysis - analysis of expected workload (e.g. by number of messages, interrupt frequencies)	The scheduling analysis helps to: - identify if the designed timing is feasible and prevents late architectural changes - determine if all tasks meet their deadlines in all circumstances - detect unnecessarily complex scheduling behavior - prevent high-load scenarios - detect unintended concurrent accesses It is beneficial to define target values for scheduling relevant parameters (e.g. maximum CPU load) with reasonable buffers and measure them during development. Scheduling can also be analyzed and optimized using timing analysis tools.

9.5 Table 5, Notations for software unit design

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Natural language	see Table 2			
1b	Informal notations	see Table 2			
1c	Semi-formal notations	see Table 2			
1d	Formal notations	see Table 2			

9.6 Table 6, Design principles for software unit design and implementation

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	One entry and one exit point in sub-programmes and functions	A function or subprogram should have a single entry and a single exit point (return statement).	MISRA C:2012-R15.5	Examples for multiple entry and exit points: - multiple return statements - function entry via "goto" statements or long jumps - Jump into a function in assembler code	Each function shall have exactly one entry point and one exit point. Multiple entry and exit points may lead to unreadable and unnecessarily complex code and is then consequently hard to verify and hard to maintain. Initialization and clean-up code (e.g. freeing of dynamic memory, releasing of resources) could be accidentally omitted leading to unexpected behavior of the function. Furthermore, multiple entry and exit points make debugging more difficult.
1b	No dynamic objects or variables, or else online test during their creation	Avoid creation of dynamic objects and usage of dynamic memory allocation. If it cannot be avoided, it needs to be verified at runtime that the creation was successful.	MISRA C:2012-Dir4.12 MISRA C:2012-R21.3 - [61508] part 7, C.2.6.3	- Usage of malloc()/free() or the new/delete operator - Usage of dynamic length arrays	Creation of the object or allocation of memory may fail due to insufficient free memory or fragmentation. Furthermore, usage of dynamic memory allocation can lead to unexpected behavior when it is not correctly freed, a pointer to freed memory is used or memory is accessed before values are stored to it. Besides, the management of dynamic objects

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
					adds significant complexity to the implementation.
1c	Initialization of variables	All variables shall be initialized to a defined value before first use.	MISRA C:2012-R9.1-9.5	- C90 and later: variables defined in functions or blocks are "automatic" and their initial value is undefined - Content of newly allocated dynamic memory is undefined	Avoid unpredictable behavior when accessing variables due to missing initialization. If the initialization of variables is unambiguously defined by the language standard (e.g. static variables in C90 and later, but not automatic variables), this can be considered as "initialized". Note: the initialization relies on the implementation of the compiler's startup code, if not solved differently. The corresponding code may become safety-related and the integrity of the initialization must be guaranteed.
1d	No multiple use of variable names	Commonly, programming languages support using the same variable name in different scopes or namespaces. If the scope or namespace is not obvious, behavior might be unexpected. Also, variables should not be re-used for different purposes within the same scope.	MISRA C:2012-R5.1 MISRA C:2012-R5.2 MISRA C:2012-R5.3	- same variable name used for local and global variables which hides the global variable	Use reasonable naming conventions and programming rules for a clear variable naming within the applicable scope. Avoid same variable names in hierarchical name spaces (e.g. global vs. local variables), while in parallel name spaces variable names can be reused (e.g. index variables in separate functions, variables containing the function return code). Unambiguous variable naming makes the code easier to understand and prevents developer confusion.
1e	Avoid global variables or else justify their usage	Avoid the definition of variables that are accessible throughout the entire code unless explicitly necessary for the functionality. In this case, its usage should be thoroughly documented in the design.		positive examples: - access to global variables only via defined functions - clearly designed state machine with transition functions negative examples: - global variable used as implicit parameter to a function	With global variables, the data flow throughout the code becomes less obvious, i.e. they can be seen as implicit parameters to the involved functions. Instead, pass the data as explicit parameters to functions and subroutines. In case where global functions are necessary, e.g. in state machines or as global flags, their existence and access needs to be documented in the design and necessary access precautions need to be implemented (e.g. locks to avoid resource conflicts, race conditions and reentrancy issues).
1f	Restricted use of pointers	Pointers are direct references to memory addresses which are used to flexibly access locations in memory. This includes data structures as well as executable code (function pointer). This approach is highly flexible and efficient, but also error prone.	MISRA C:2012-Dir4.8 MISRA C:2012-R7.4 MISRA C:2012-R8.13 MISRA C:2012-R11.1-11.9 MISRA C:2012-R18.1-18.8 MISRA C:2012-R22.5+22.6 - [61508] Part 7, C.2.6.6	- jump tables for functions - linked lists - pointer to peripheral	Restricted usage of pointers reduces the amount of possibly error-prone implementations. Consider alternative approaches such as arrays provide e.g. for improved type and range checking, when possible. If pointers are unavoidable, check the validity of a pointer (e.g. not NULL, within a defined boundary, etc.) and declare it as constant, if possible.
1g	No implicit type conversions	An implicit type conversion is a conversion, where the target data type is not explicitly stated in the statement but implicitly given by the type of target variable. Conversion is done differently for each programming language and may depend on compiler implementation.	MISRA C:2012-Chapter C.1	- assignment of variables to a smaller size variable - sign conversion - integer promotion in C - object type conversion in object-oriented languages	If a type conversion is required, always do explicit type conversions and be aware of the consequences (e.g. data loss, call of constructors, etc.). Provide a rationale for such cases.
1h	No hidden data flow or control flow	Hidden data flow describes the sequence in which transfer, use, and transformation of data are performed during the execution of a computer program without being explicitly specified in the design. Hidden control flow describes the change of the sequence in which operations are performed during the execution of a computer program without being explicitly specified in the design.	- [24765] 3.1006, 3.856 - [61508] Part 7, C.5.9 - [61508] Part 7, C.5.10	Hidden data flow: - data exchange via global variables or shared memory which are not explicitly declared as interface in the design - unclear referencing data via pointer and offset Hidden control flow: - uncaught exceptions that transfer the execution to unspecified parts of the program - use of unspecified branch conditions or function pointers	Hidden data or control flow make it difficult to understand, maintain, and verify the software component and may lead to insufficient code coverage in testing. Both data and control flow must be specified in the design in a way comprehensible for a reviewer. Obfuscated implementation patterns should be avoided.

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1i	No unconditional jumps	Unconditional jumps change the program's control flow to a specified location without defining a condition and without an obvious reason.	Schmidt: MISRA C:2012-R8.15 (goto)	- unconditional goto statements - break/continue statements in loops that should be part of the loop condition - setjmp/longjmp	Unconditional jumps reduce the readability of the source code and make it difficult to understand, maintain and verify. Use appropriate logical control flow constructs like if/else, while/do...while, etc. instead.

1j	No recursions	Recursion is a self-referencing control flow, i.e. functions that call themselves multiple times until a specified condition is met. It leads to multiple instances of the same object.	MISRA C:2012-R17.2
			- [61508] part 7, C.2.6.7
				- faculty calculation:
				fak(n) {
				if (n <=0) {
				return 1;
				} else {
				return n*fak(n-1);
				}
				}
					Recursive algorithms are powerful, but difficult to understand and verify. Besides, recursions are error prone to stack overflows and unbounded execution time, which can lead to serious failures.
					Use non-recursive algorithms instead or limit recursion depth by design.

9.7 Table 7, Methods for software unit verification

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Walk-through	see Table 4, method 1a. This method applies to both the unit design and the implemented software unit.			In case of code generation from a model, the walk-through of the implemented software unit is replaced by a walk-through of the model. Note that the code generator needs to be suited for the target ASIL.
1b	Pair-programming	Two developers work jointly and simultaneously on the same software unit design and implementation.			Pair-programming provides the following benefits: - implicit review of developed code from developers with different experience and background - improved design and code quality - improved focus on work item due to joint work - fosters knowledge exchange between involved developers - opportunity to consider a larger number of solutions It is beneficial to assess the personal compatibility of the developer pair.
1c	Inspection	see Table 4, method 1b. This method applies to both the unit design and the implemented software unit.			In case of code generation from a model, the inspection of the implemented software unit is replaced by an inspection of the model. Note that the code generator needs to be suited for the target ASIL.
1d	Semi-formal verification	This method is applicable if a semi-formal specification of the software unit design exists. Semi-formal verification means verifying the semi-formal elements in the software unit design specification as well as verifying the unit implementation against these semi-formal elements, using an automated approach.	[29119] Part 1, Ch. 4.4.2 for "model-based testing" aspect	- tool-based verification of the syntactical correctness of the software unit design - tool-based consistency check between the unit design and the implementation, e.g. a 1:1 mapping for functions, variables, interfaces, etc. - generate test vectors based on the semi-formal elements of the software unit design and use these test vectors for testing the unit implementation against the unit design	Semi-formal verification helps to ensure - completeness and consistency of specification elements - syntactic completeness and correctness of the implementation - completeness of unit-level test cases. However, it should be noted that this method cannot check the correctness of the specification against the requirements or the semantic correctness of the implementation against the unit design.
1e	Formal verification	see Table 4, method 1e. This method is used to verify the correctness of the implemented software unit against the formal unit design.			Formal verification is the strongest method of verifying the implementation and is capable of proving its correctness.
1f	Control flow analysis	see Table 4, method 1f applied to the unit level.			
1g	Data flow analysis	see Table 4, method 1g applied to the unit level.			
1h	Static code analysis	Tool-aided analysis of the source code against the applicable programming and design guidelines (generally recognized, customer-required and company-specific guidelines). This includes checks regarding absence of unreliable or unwanted code constructs, compliance to naming	- [61508] Part 7, B.6.4 - [MISRA] - [CERT C], [CERT C++] - [CWE] - [HIS metrics]	- MISRA - CERT-C - CWE (common weakness enumeration) - various commercial and non-commercial code analysis tools	Static code analysis helps to: - identify possibly incorrect programming patterns without having to execute the program - ensure compliance with coding guidelines (see Table 1) - ensure compliance with design principles (see Table 6)

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
		conventions and measurement of code complexity.			- improve readability and maintainability of the code
1i	Static analyses based on abstract interpretation	Advanced form of static code analysis (method 1h) which in addition takes into account the semantics of the programming language by interpreting the program code and abstracting data values. Elements of such interpretation are control flow analysis, data flow analysis, data value analysis, detection of race conditions and deadlocks, and incorrect use of pointers.		- various commercial code analysis tools	This method helps detecting e.g. - array index out of bounds - division by zero - arithmetic overflows - null pointer access - out-of-range casts These errors are often hard to find in testing as they need special input values to be triggered.
1j	Requirements-based test	With the execution of requirements-based tests, the correct behavior of the unit implementation against the software requirements on unit level is verified. This comprises the software unit design specification as well as software safety requirements allocated to the unit. See Table 8 for information on deriving test cases.		- test cases verifying the behavior of a unit against a state machine specification	This method supports the - compliance of the implementation against the software unit design specification - detection of unintended functionality in conjunction with structural coverage measurements (see Table 9) - completeness of the requirements coverage - verification of safety mechanisms implemented in software units
1k	Interface test	With the execution of interface tests, the correct behavior of a unit interface as defined in the software unit design specification is verified by varying input data. See Table 8 for information on deriving test cases.	- [61508] Part 7, C.5.3	- black box testing of the unit interfaces - fuzz testing (i.e. tests based on random input data)	This method supports detecting - ineffective or missing range checks on input parameters - unexpected error behavior due to invalid input data - invalid output data - incorrect or unintended side effects
1l	Fault injection test	With the execution of fault injection tests, the correct behavior of the unit with respect to injected faults which cannot be triggered via the unit interface is verified. Fault injection includes the manipulation of internal data, code or hardware registers.	- [61508] Part 7, B.6.10	- injection of invalid data - modification of executable code at runtime to trigger error scenarios - corruption of registers which are considered as not reliable Technically faults can be injected via various means, e.g. via a debugger or via modifications done by the test code	This method supports - detecting ineffective or missing safety mechanisms - detecting unexpected error behavior - verifying defensive programming mechanisms that are not reachable under normal conditions. It demonstrates the robustness of the software unit with respect to undefined operating conditions. Fault scenarios can be derived from the safety analysis and via error guessing (see Table 8, method 1d).
1m	Resource usage evaluation	With resource usage evaluation, the maximum resource usage of the implementation is verified against the resource limitations defined in the software unit design specification. Besides, the assignment of hardware resource access according to the hardware-software interface specification can be verified. Note that resource limitations may also be defined on architectural level and not necessarily on unit level.		Measurement or analysis of - worst case execution time - CPU load - memory consumption - network throughput - power consumption. Static check of register access in the implementation.	This method supports detecting e.g. - impermissible high resource usage - memory leaks - non-compliance with the hardware-software interface specification. For this, the unit implementation may have to be integrated in the intended target environment and stimulated at least with the intended use cases.
1n	Back-to-back comparison test between model and code, if applicable	With back-to-back comparison test, the behavior of the unit implementation is compared with the behavior of a unit model that simulates the unit implementation. This method is applicable if a model of the software unit exists.			For this method, either - the implementation generated from the model or - the manual implementation is verified against the model simulation. It supports the verification of the compliance against the design specification.

9.8 Table 8, Methods for deriving test cases for software unit testing

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Analysis of requirements	Systematic analysis of the requirements allocated to the software unit and derivation of a complete set of test cases. This method applies to all requirements of a safety-related		- systematic derivation of test cases out of a state diagram - derivation of interface tests based on the specification	This method supports verifying the compliance of the unit implementation with the unit design specification. Test cases are derived from the software unit design specification and from the software safety requirements

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
		software unit, not only the safety requirements.			allocated to the software unit, not from the source code. At least one test case must exist for each functional requirement. Create traces between the requirements and the test cases to ensure completeness.
1b	Generation and analysis of equivalence classes	Equivalence classes are sets of initial states and input values where all values of one set are expected to cause the same behavior of the test object. The equivalence classes are derived from the software unit design specification.	- [61508] Part 7, C.5.7 - [CTM]	- Classification tree method	This method supports the creation of test cases by - providing a systematic approach to derive test cases by analyzing the states and input values of the test object with respect to the software unit design specification - partitioning the states and input values into a finite number of sets - avoiding test cases which would trigger the same behavior Values outside the allowed value ranges are also considered to form equivalence classes. Equivalence classes should not be derived from the implementation.
1c	Analysis of boundary values	Boundary value analysis focuses on the minimum and maximum edges of the sets formed by equivalence classes. Values around these edges are selected for test cases. A prerequisite for this analysis is that the values are ordered so that boundaries can be identified.	- [61508] Part 7, C.5.4 - [29119] Part 4, Ch. 5.2.3	- detection of off-by-one errors, e.g. usage of "<" instead of "<=" - verification of the boundaries of the value ranges and also the underlying data type by one value below and above in the smallest possible distance	This method supports to systematically verify the correct specification and implementation of the boundaries of the identified equivalence classes, since these boundary values are typically error prone. Together with the equivalence classes (method 1b), it provides a completeness criterion for testing the relevant input values and states of the test object. Boundary values should not be derived from the implementation, yet it should be ensured that the boundary values resulting from the used data types are covered.
1d	Error guessing based on knowledge or experience	Error guessing uses the experience of developers and testers to create test cases that would not be the outcome of one of the above methods.	- [61508] Part 7, C.5.5 - [29119] Part 4, Ch. 5.4.1	- Create tests from the analysis of bug reports from similar projects - Utilization of non-obvious dependencies from e.g. brainstorming	This method brings in the experience from - similar projects - experts in the field - field monitoring Encourage test engineers to apply this method as it fosters creativity and outside-the-box thinking to stress the test object with unconventional scenarios.

9.9 Table 9, Structural coverage metrics at the software unit level

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Statement coverage	Statement coverage is the ratio which statements of a software unit have been executed during test runs relative to all statements in the software unit. This is also known as C0 coverage.	- [26262] Part 1, 3.160 - [61508] Part 7, C.5.8 - [Hayhurst] - [Coverage]	- C statements executed during testing versus the number of all C statements in the unit. Usually shown as percentage.	Statement coverage - measures which statements of the source code have been executed during testing - detects untested code parts when looking at the detailed outcome of the measurement - may detect unintended functionality since untested code might not originate from the specification - may detect gaps in the specification if the yet untested code is considered relevant. In safety-related software all code should have been executed during at least one test. However, a coverage of 100% is not always necessary if the missed code is assessed and a reasonable justification is given.
1b	Branch coverage	Branch coverage is the ratio which branches of the control flow of a software unit have been executed during test runs relative to all branches	- [26262] Part 1, 3.13 - [61508] Part 7, C.5.8 - [Hayhurst] - [Coverage]	- Code branches executed during testing versus the number of all branches in the	Branch coverage - measures which branches of the source code have been executed during testing

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
		in the software unit. This is also known as C1 coverage.		unit. Usually shown as percentage. - All case-branches are considered as separate code branches - An if-statement always counts as two branches, independent of the existence of an else-clause	- detects untested code branches when looking at the detailed outcome of the measurement - may detect unintended functionality since untested code might not originate from the specification - may detect gaps in the specification if the yet untested code is considered relevant - supports detecting code branches that are only executed under rare circumstances. 100% branch coverage implies 100% statement coverage. In safety-related software all branches should have been executed during at least one test. However, a coverage of 100% is not always necessary if the missed code paths are assessed and a reasonable justification is given.
1c	MC/DC (Modified Condition / Decision Coverage)	MC/DC coverage is an extension of branch coverage which considers decisions that are composed of multiple conditions. For example, the decision "A and (B or C)" is composed of the three conditions "A = x<7, B = true, C = y > 0". 100% MC/DC coverage is achieved if each condition is shown to independently affect the decision. This can be done by varying just that condition while holding fixed all other conditions. MC/DC coverage is the ratio of all single condition outcomes that independently affect the decisions in the test object executed during test runs.	- [26262] Part 1, 3.92 - [61508] Part 7, C.5.8 - [Hayhurst] - [Coverage]	The combined condition "if A and (B or C)" leads to e.g. the following test cases: 1. A false, B false, C true 2. A true, B false, C true 3. A true, B false, C false 4. A true, B true, C false For example for condition A, test cases 1 and 2 independently vary the decision outcomes while keeping B and C fixed. 4 test cases are sufficient instead of all 8 possible condition combinations.	MC/DC coverage - measures if an appropriate subset of all possible condition combinations in the source code have been executed during testing - detects untested branching conditions when looking at the detailed outcome of the measurement - may detect gaps in the specification if the yet untested condition is considered relevant - supports detecting code branches that are only executed under rare circumstances - supports limiting the number of test cases when using combined conditions without loss of test quality. 100% MC/DC coverage implies 100% branch coverage. However, a coverage of 100% is not always necessary if the missed condition combinations are assessed and a reasonable justification is given.

9.10 Table 10, Methods for verification of software integration

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Requirements-based test	With the execution of requirements-based tests, the correct behavior of integrated software components or the integrated embedded software as defined in the software architectural design specification is verified. See Table 11 for information on deriving test cases.		- test cases verifying the behavior of the integrated software against a state machine specification - test cases triggering the safety mechanism according to the conditions specified in the architectural design specification	This method supports the - compliance of the implementation against the software architectural design specification - detection of unintended functionality in conjunction with structural coverage measurements (see Table 12) - completeness of the requirements coverage - verification of safety mechanisms implemented in the integrated software components or the integrated embedded software
1b	Interface test	With the execution of interface tests, the correct interaction between integrated software units or components as defined in the software architectural design specification is verified against the applicable interface specifications. See Table 11 for information on deriving test cases.	- [61508] Part 7, C.5.3	- black box testing of integrated software components	This method supports detecting - if integrated components have incompatible interfaces towards each other - unexpected error behavior due to invalid input data - invalid output data and behavior - incorrect or unintended side effects. Tests should be performed on multiple hierarchical levels of the software architectural design. It is not sufficient to limit the tests to the complete integrated embedded software.
1c	Fault injection test	With the execution of fault injection tests, the correct behavior of integrated software units,	- [61508] Part 7, B.6.10 - [26262] Part 11, 4.8	- hardware faults - invalid sensor data - data corruption	This method supports - detecting ineffective or missing safety mechanisms

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
		components, or the embedded software with respect to injected faults and particularly the effectiveness of the safety mechanisms is verified. Fault injection includes the manipulation of internal data and code, but also faults originating outside the software.		See also Table 7, method 1l for further examples.	- detecting unexpected error behavior - verifying freedom from interference. It demonstrates the correct behavior of safety mechanisms with respect to undefined operating conditions and the transition to the safe state. Fault scenarios can be derived from the safety analysis and via error guessing (see Table 11, method 1d).
1d	Resource usage evaluation	With resource usage evaluation, the maximum resource usage of integrated software units, components, or the embedded software is verified against the resource limitations defined in the software architectural design specification. Besides, the assignment of hardware resource access according to the hardware-software interface specification can be verified.		Measurement or analysis of - worst case execution time - typical/average execution time - CPU load - memory consumption - network throughput - power consumption.	This method supports detecting e.g. - impermissible high resource usage - memory leaks - non-compliance with the hardware-software interface specification. For this, the software may have to be integrated in the intended target environment and stimulated at least with the intended use cases.
1e	Back-to-back comparison test between model and code, if applicable	With back-to-back comparison test, the behavior of integrated software units, components, or the embedded software is compared against the behavior of a model that simulates the implementation. This method is applicable if a model of the software exists.		- comparing the output of a system against a second independent path (e.g. output from a model)	This method supports - verifying the compliance of the implementation against the architectural design specification - verifying the correctness of code generated from a model - the automatic generation of expected values for any given test input. This method is very powerful in detecting anomalies, but also very effort intense.
1f	Verification of the control flow and data flow	With verification of the control flow and data flow the integrated software units, components, or the embedded software are verified to adhere to specified control and data flow of the software architecture specification.	- [29119] part 4, Ch. 5.2.8 "State transition testing" - [29119] part 4, Ch. 5.3.7 "Data flow testing"	During test execution, verifying the - control flow by tracing state transitions - data flow by tracing memory read and write accesses	This method supports - verifying the compliance of the implementation against the architectural design specification - verifying the effectiveness of safety measures identified in the software safety analysis.
1g	Static code analysis	See Table 7, method 1h applied to integrated software units, components, or the embedded software.			
1h	Static analyses based on abstract interpretation	See Table 7, method 1i applied to integrated software units, components, or the embedded software.			

9.11 Table 11, Methods for deriving test cases for software integration testing

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Analysis of requirements	See Table 8, method 1a applied to the software architectural level.			
1b	Generation and analysis of equivalence classes	See Table 8, method 1b applied to the software architectural level.			
1c	Analysis of boundary values	See Table 8, method 1c applied to the software architectural level.			
1d	Error guessing based on knowledge or experience	See Table 8, method 1d applied to the software architectural level.			

9.12 Table 12, Structural coverage at the software architectural level

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Function coverage	Function coverage is the ratio which interface functions of a software component have been called during integration test runs relative to all interface functions of the software component.		- 70% function coverage means that e.g. 7 of in total 10 interface functions of the component have been executed during integration testing.	This method supports the test completeness argumentation by verifying that interface functions of a component are executed in integration tests. The results of statement coverage measurements (see Table 9, method 1a) performed on the integration tests can be used to derive this value.
1b	Call coverage	Call coverage is the ratio which external call points of a software component interface function have been executed during integration test runs relative to all external call points of this interface function.		- 70% call coverage for an interface function means e.g. that 7 of in total 10 external call points of this interface function are executed during integration testing.	This method supports the test completeness argumentation - by verifying that all calls between software components in the software architecture are triggered during integration testing - by verifying that interface functions are executed from all existing call points according to the software architecture specification, as even if multiple calls may have identical signature, their context may be different. The results of statement coverage measurements (see Table 9, method 1a) performed on the integration tests can be used to derive this value.

9.13 Table 13, Test environments for conducting the software testing

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Hardware-in-the-loop	This method verifies that the embedded software is functional on the target platform (e.g. the ECU) in a simulated environment.	- [HIL]	- Hardware setup stimulating the ECU with simulated sensor input	This method supports - testing of the ECU without the need of having all other ECUs available - verifying the software's behavior with simulated sensor input
1b	Electronic control unit network environments	This method verifies that the embedded software is functional on the target platform (e.g. the ECU) in an environment consisting of a partially or fully integrated set of ECUs of the vehicle.		- Test communication behavior of the ECU with a rest-bus simulation	This method supports - testing of the embedded software in the ECU without the necessity of integrating the ECU into a vehicle - verifying the interaction of the ECU's software with other ECUs in the network environment
1c	Vehicles	This method verifies that the embedded software is functional on the target platform (e.g. the ECU) in a test vehicle.		- Test drives - Behavior of ECU in the garage	This method supports - verifying the embedded software's functionality in the ECU in the target vehicle - stimulating the software with real-life data

9.14 Table 14, Methods for tests of the embedded software

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Requirements-based test	With the execution of requirements-based tests the correct behavior of the embedded software executed in the target environment with respect to the software requirements is verified. See Table 15 for information on deriving test cases.		- test cases triggering the safety mechanism according to the conditions specified in the safety-related requirements	This method supports the - compliance of the embedded software against the safety-related requirements in the target environment - detection of unintended functionality by considering all software requirements (safety and non-safety related) - completeness of the requirements coverage

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
					- verification of safety mechanisms in the target environment. If possible, test cases from software integration testing can be re-used.
1b	Fault injection test	With the execution of fault injection tests the correct behavior of the embedded software executed in the target environment with respect to injected faults is verified. This includes particularly the effectiveness of the safety mechanisms.	- [61508] part 7, B.6.10	- input values outside of specified range (e.g. temperature, speed, acceleration, voltage, data corruption, ...) - faults in calibration data - extreme environmental conditions as derived from system level analyses (see also part 4, table 3, method 1h and part 4, table 4) See also Table 10, method 1c for further examples.	This method supports - detecting ineffective or missing safety mechanisms - detecting unexpected error behavior - verifying freedom from interference. It demonstrates the correct behavior of safety mechanisms with respect to undefined operating conditions and the transition to an emergency operation and/or the safe state. Fault scenarios can be derived from the safety analysis and via error guessing (see Table 15, method 1d).

9.15 Table 15, Methods for deriving test cases for the test of the embedded software

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Analysis of requirements	See Table 8, method 1a applied to the embedded software executed in the target environment.			
1b	Generation and analysis of equivalence classes	See Table 8, method 1b applied to the embedded software executed in the target environment.			
1c	Analysis of boundary values	See Table 8, method 1c applied to the embedded software executed in the target environment.			
1d	Error guessing based on knowledge or experience	See Table 8, method 1d applied to the embedded software executed in the target environment.			
1e	Analysis of functional dependencies	The analysis of functional dependencies focuses on deriving test cases from (a) specified and identified functional dependencies between elements and (b) from necessary independence and freedom from interference between elements.		- usage of dependent failure analysis to derive test cases - dependencies between safety-related ECUs on system level - dependencies between state machines on different abstraction levels (microcontroller, ECU, system)	This method supports - detecting dependent failures between elements - confirming ASIL decomposition - verifying freedom from interference argumentation - verifying identified functional dependencies. See also Part 4, Table 3, method 1f for deriving test cases from the analysis of functional dependencies on system level.
1f	Analysis of operational use cases	The analysis of operational use cases focuses on deriving test cases by considering the operational states of the embedded software (running, diagnosis, start-up, shutdown, …), as well as software update, bootloader, end-of-line testing, etc., executed in the target environment.		- software update in the field - starting the nominal application only if the integrity of the software is ensured by bootloader - safety-related behavior of the embedded software in different operating modes such as start-up, diagnosis, degraded, power-down (going to sleep), power-up (waking up), calibration, functions for mode synchronization between different ECUs	This method supports - ensuring correct behavior of the software in each phase of the entire life cycle of the embedded software, executed in the target environment - detecting incorrect behavior in exceptional operational use cases - applying a holistic test approach to the complete embedded software. See also Part 4, Table 3, method 1h for deriving test cases from the analysis of environmental conditions and operational use cases on system level.

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

Ref.	Method	What is it?	Reference to further details	Examples	Benefit & Best practice
1a	Plausibility checks on calibration data	The execution of plausibility checks on calibration data aims at verifying if the applied set of calibration data leads to reasonable behavior of the executed software.		- testing known dependencies between different features of a produced vehicle during end-of-line testing or at runtime - verify if calibration data lies within specified ranges - verify if resulting output data based on the calibration data lies within specified ranges	As calibration data can change after release of the embedded software, runtime mechanisms for the detection of unintended changes of calibration data are necessary. This method supports - identifying incompatible calibration data in produced or software-updated vehicles - detecting possibly dangerous values of modified calibration data
1b	Redundant storage and comparison of calibration data	Redundant storage of calibration data aims at detecting random hardware faults in memory areas at runtime and controlling the effects of divergent data values between the content of the redundant memory areas.		- when using calibration data, read both stored values, compare them and either use the data or, if diverging, switch to the safe state	This method supports - mitigating random hardware failures in memory areas - achieving hardware architectural metrics (single-point fault metric, latent-fault metric) and the probability metric for random hardware failures
1c	Calibration data checks using error detecting codes	Using error detecting codes enables detecting faults in calibration data by adding information for data verification.		- Hamming codes (e.g. ECC) - Cyclic Redundancy Codes (CRC)	This method supports - mitigating random hardware failures in memory areas - continuing normal operation even in case of permanent or temporary bit flips - detection of transmission errors when applying the calibration data - protecting data storage through channels of lower safety integrity. For high ASILs, use hardware implementations for error detection and correction (e.g., ECC). Depending on the amount of additional bits one or more faulty bits can be detected or even corrected.

9.17 References

ID	Title	Reference
[24765]	EEE/ISO/IEC 24765-2017 - ISO/IEC/IEEE International Standard - Systems and software engineering -- Vocabulary	https://standards.ieee.org/standard/24765-2017.html
[25023]	ISO/IEC 25023:2016 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality	https://www.iso.org/standard/35747.html
[26262]	ISO 26262:2018. Road vehicles - Functional safety. 12 parts	Vocabulary: https://www.iso.org/standard/68383.html Management: https://www.iso.org/standard/68384.html Concept phase: https://www.iso.org/standard/68385.html System level: https://www.iso.org/standard/68386.html Hardware level: https://www.iso.org/standard/68387.html Software level: https://www.iso.org/standard/68388.html Lifecycle: https://www.iso.org/standard/68389.html Supporting processes: https://www.iso.org/standard/68390.html Safety analyses: https://www.iso.org/standard/68391.html Guidelines: https://www.iso.org/standard/68392.html Semiconductors: https://www.iso.org/standard/69604.html Motorcycles: https://www.iso.org/standard/69605.html
[29119]	ISO/IEC/IEEE 29119:2022 Software and systems engineering — Software testing	https://www.iso.org/standard/81291.html
[61508]	Functional safety of electrical/electronic/programmable electronic safety-related systems. 2010	General requirements: https://webstore.iec.ch/en/publication/5515 E/E/P systems: https://webstore.iec.ch/en/publication/5516 Software: https://webstore.iec.ch/en/publication/5517 Definitions: https://webstore.iec.ch/en/publication/5518 SIL examples: https://webstore.iec.ch/en/publication/5519 Guidelines: https://webstore.iec.ch/en/publication/5520 Overview of techniques: https://webstore.iec.ch/en/publication/5521
[9126]	ISO/IEC TR 9126-3:2003 Software engineering — Product quality — Part 3: Internal metrics	https://www.iso.org/standard/22891.html
[AUTOSAR 207]	AUTOSAR. SW-C and System Modeling Guide. Document ID 207	https://www.autosar.org/fileadmin/standards/R23-11/CP/AUTOSAR_CP_TR_SWCModelingGuide.pdf
[AUTOSAR 839]	Guidelines for the use of the C++14 language in critical and safety-related systems	https://www.autosar.org/fileadmin/standards/R22-11/AP/AUTOSAR_RS_CPP14Guidelines.pdf
[CERT C]	SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition)	https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-c-coding-standard-2016-v01.pdf
[CERT C++]	SEI CERT C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems in C++ (2016 Edition)	https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-standard-2016-v01.pdf
[cohesion]	Wikipedia: Cohesion (computer science)	https://en.wikipedia.org/wiki/Cohesion_(computer_science)
[concurrent]	Wikipedia: Concurrent computing	https://en.wikipedia.org/wiki/Concurrent_computing
[Coverage]	Wikipedia: Code Coverage	https://en.wikipedia.org/wiki/Code_coverage
[CTM]	Classification tree method	https://en.wikipedia.org/wiki/Classification_Tree_Method
[CWE]	Common weakness enumeration	https://cwe.mitre.org/
[def comp]	Wikipedia: Programming complexity	https://en.wikipedia.org/wiki/Programming_complexity
[def prog]	Wikipedia: Defensive programming	https://en.wikipedia.org/wiki/Defensive_programming
[Fagan]	Wikipedia: Fagan inspection	https://en.wikipedia.org/wiki/Fagan_inspection
[Hayhurst]	Hayhurst, Kelly; Veerhusen, Dan; Chilenski, John; Rierson, Leanna (May 2001). "A Practical Tutorial on Modified Condition/Decision Coverage" (PDF). NASA/TM-2001-210876.	https://shemesh.larc.nasa.gov/fm/papers/Hayhurst-2001-tm210876-MCDC.pdf
[hierarchical]	Hierarchical architecture	https://www.tutorialspoint.com/software_architecture_design/hierarchical_architecture.htm
[HIL]	Wikipedia: Hardware-in-the-loop	https://en.wikipedia.org/wiki/Hardware-in-the-loop_simulation
[HIS metrics]	HIS Source Code Metrics 1.3.1 from 1.4.2008	
[MARTE]	Modeling and Analysis of Real-time and Embedded systems (UML extension)	https://www.omg.org/omgmarte/
[MISRA]	MISRA homepage	https://www.misra.org.uk/
[naming]	Wikipedia: Naming convention (programming)	https://en.wikipedia.org/wiki/Naming_convention_(programming)
[style]	Wikipedia: Programming style	https://en.wikipedia.org/wiki/Programming_style
[subset]	MISRA C++ language subset	https://www.embedded.com/using-the-misra-c-language-subset-in-your-application/
[typing]	Wikipedia: Strong and weak typing	https://en.wikipedia.org/wiki/Strong_and_weak_typing
[UML]	Unified modeling language	https://www.omg.org/spec/UML

Table 6: References to the methods of ISO 26262 Part 6

10 Participating Companies

Participating companies in the “UG Software ISO 26262” working group:

- Analog Devices GmbH
- Bertrandt Ingenieurbüro GmbH
- Brose Fahrzeugteile SE & Co.
- Elektrobit Automotive GmbH
- Elmos Semiconductor SE
- Infineon Technologies AG
- innoventis GmbH
- KOSTAL Automobil Elektrik GmbH & Co. KG
- Kugler Maag CIE GmbH
- Mahle International GmbH
- Marelli Automotive Lighting Reutlingen GmbH
- Marquardt Service GmbH
- Melecs EWS GmbH
- Preh GmbH
- OptE GP Consulting
- Optimize E Global Performance
- STMicroelectronics Application GmbH
- TDK Electronics AG
- TE Connectivity Germany GmbH
- vancom GmbH & Co. KG
- Vector Informatik GmbH
- Webasto SE

Contact

Dr.-Ing. Stefan Gutschling • Fachverbandsgeschäftsführer • Fachverband Automotive • Mobility
Tel.: +4969 6302 278 • Mobil: +49162 2664 961 • E-Mail: Stefan.Gutschling@zvei.org

ZVEI e. V. • Electro and Digital Industry Association • Lyoner Straße 9 • 60528 Frankfurt am Main • Germany
Lobbying Register ID.: R002101 • EU Transparency Register ID: 94770746469-09 •

Datum: 19.07.2024