Die Elektroindustrie

Best Practice Guideline

Software for Safety-Related Automotive Systems

German Electrical and Electronic Manufacturers' Association

Imprint
Best Practice Guideline
Software for Safety-Related Automotive Systems

Publisher:
ZVEI - German Electrical and Electronic
Manufacturers' Association
Automotive – Electronics, Infrastructure & Software
Lyoner Strasse 9
60528 Frankfurt am Main, Germany

Phone: +49 69 6302-276
Fax: +49 69 6302-407
E-mail: zvei-be@zvei.org
www.zvei.org

Responsible: Dr. Stefan Gutschling

Dezember 2020 – Version 2.0

Table of Contents

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 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.
- 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.

or

- 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.

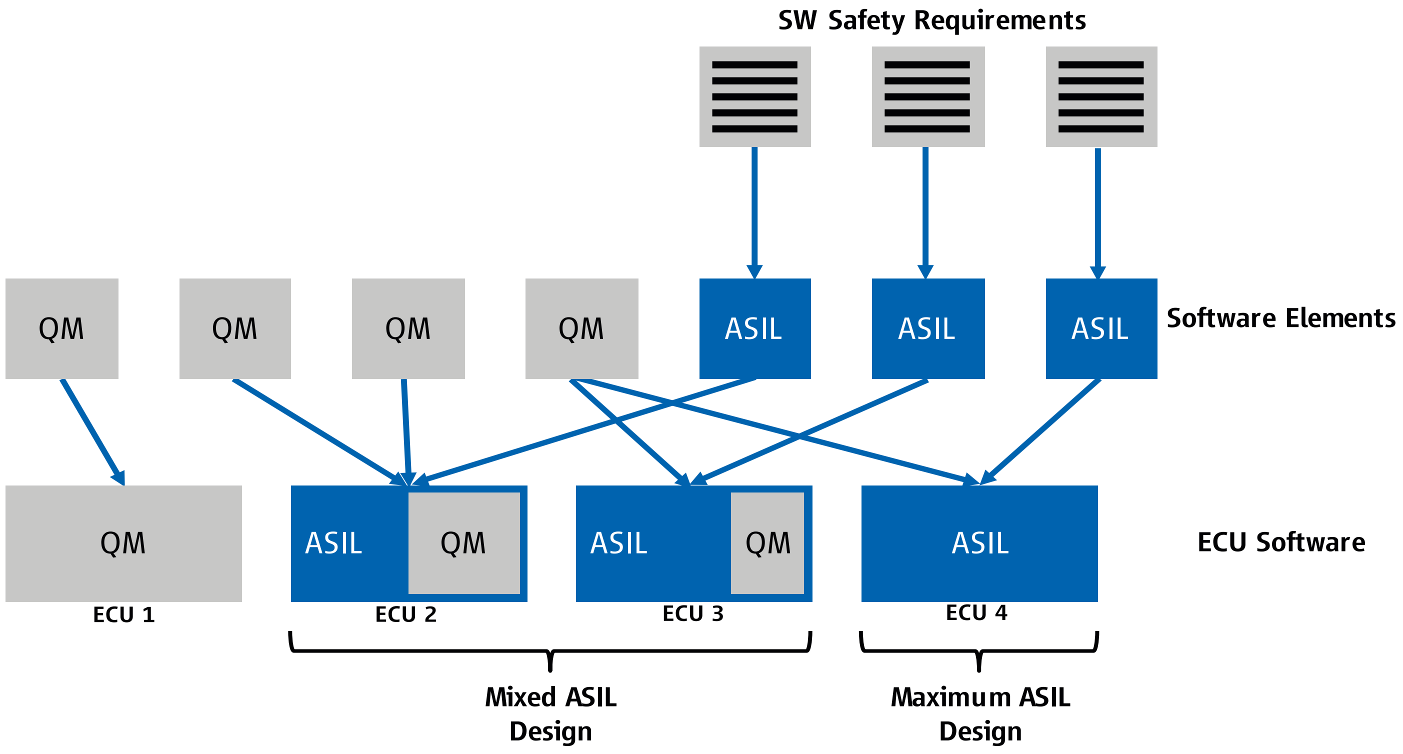


Figure 1: Mapping of software safety requirements to ECUs

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. "black-box" 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 **I**nterrupt **S**ervice **R**outines 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 26262-9:2018 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).

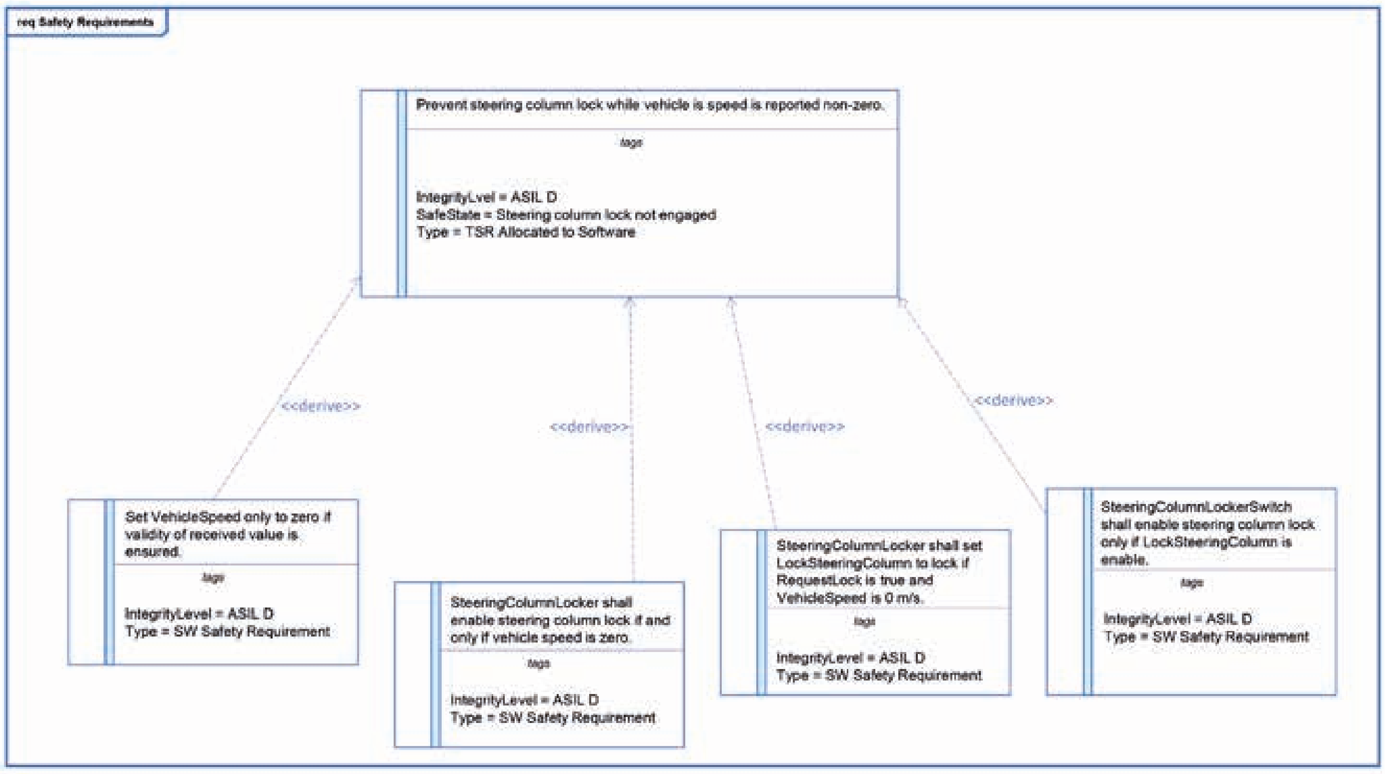

req Safety Requirements
Prevent steering column lock while vehicle is speed is reported non-zero.
tags
IntegrityLvel = ASIL D
SafeState = Steering column lock not engaged
Type = TSR Allocated to Software
<<derive>>
<<derive>>
<<derive>>
<<derive>>
Set VehicleSpeed only to zero if validity of received value is ensured.
tags
IntegrityLevel = ASIL D
Type = SW Safety Requirement
SteeringColumnLocker shall enable steering column lock if and only if vehicle speed is zero.
tags
IntegrityLevel = ASIL D
Type = SW Safety Requirement
SteeringColumnLocker shall set LockSteeringColumn to lock if RequestLock is true and VehicleSpeed is 0 m/s.
tags
IntegrityLevel = ASIL D
Type = SW Safety Requirement
SteeringColumnLockerSwitch shall enable steering column lock only if LockSteeringColumn is enable.
tags
IntegrityLevel = ASIL D
Type = SW Safety Requirement

Figure 2: Safety Requirements

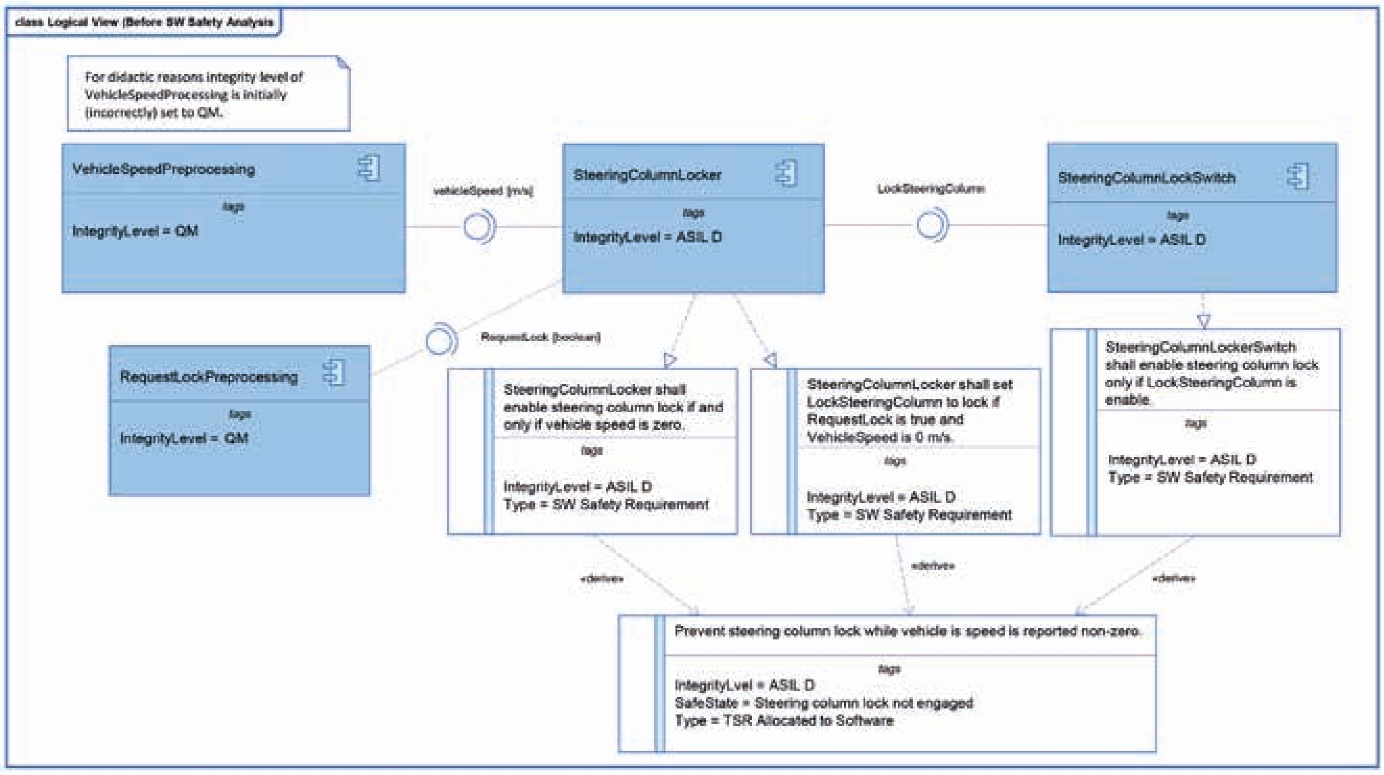

class Logical View (Before SW Safety Analysis
For didactic reasons integrity level of VehicleSpeedProcessing is initially (incorrectly) set to QM.
VehicleSpeedPreprocessing
tags
IntegrityLevel = QM
vehicleSpeed [m/s]
SteeringColumnLocker
tags
IntegrityLevel = ASIL D
LockSteeringColumn
SteeringColumnLockSwitch
tags
IntegrityLevel = ASIL D
RequestLock [boolean]
RequestLockPreprocessing
tags
IntegrityLevel = QM
SteeringColumnLocker shall enable steering column lock if and only if vehicle speed is zero.
tags
IntegrityLevel = ASIL D
Type = SW Safety Requirement
SteeringColumnLocker shall set LockSteeringColumn to lock if RequestLock is true and VehicleSpeed is 0 m/s.
tags
IntegrityLevel = ASIL D
Type = SW Safety Requirement
SteeringColumnLockerSwitch shall enable steering column lock only if LockSteeringColumn is enable.
tags
IntegrityLevel = ASIL D
Type = SW Safety Requirement
«derive»
«derive»
«derive»
Prevent steering column lock while vehicle is speed is reported non-zero.
tags
IntegrityLvel = ASIL D
SafeState = Steering column lock not engaged
Type = TSR Allocated to Software

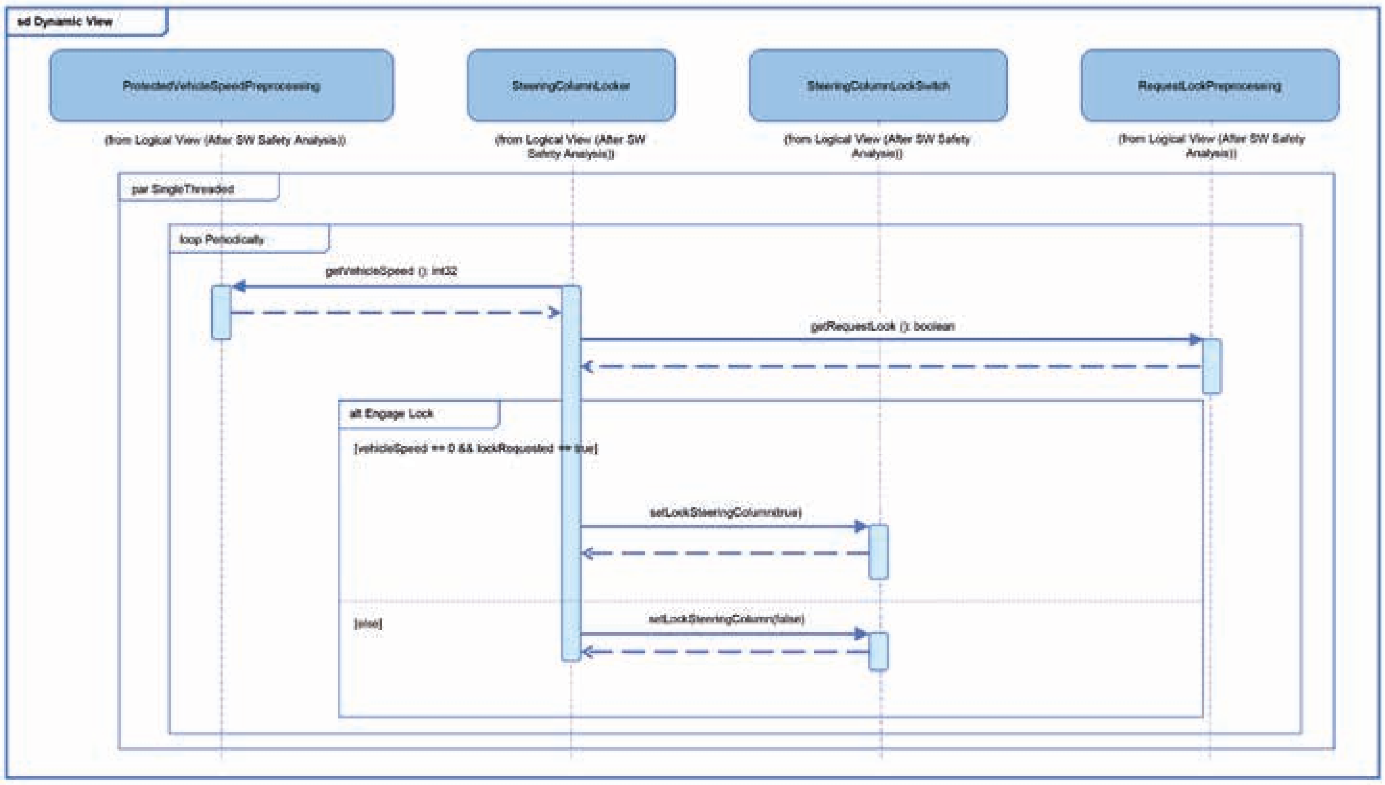
Figure 3: Logical View (Before SW Safety Analysis)

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. If a negative impact is detected, a safety measure is defined. 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.


sd Dynamic View
ProtectedVehicleSpeedPreprocessing
(from Logical View (After SW Safety Analysis))
SteeringColumnLocker
(from Logical View (After SW Safety Analysis))
SteeringColumnLockSwitch
(from Logical View (After SW Safety Analysis))
RequestLockPreprocessing
(from Logical View (After SW Safety Analysis))
par SingleThreaded
loop Periodically
getVehicleSpeed (): int32
getRequestLock (): boolean
alt Engage Lock
[vehicleSpeed == 0 && lockRequested == true]
setLockSteeringColumn(true)
[else]
setLockSteeringColumn(false)
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 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.

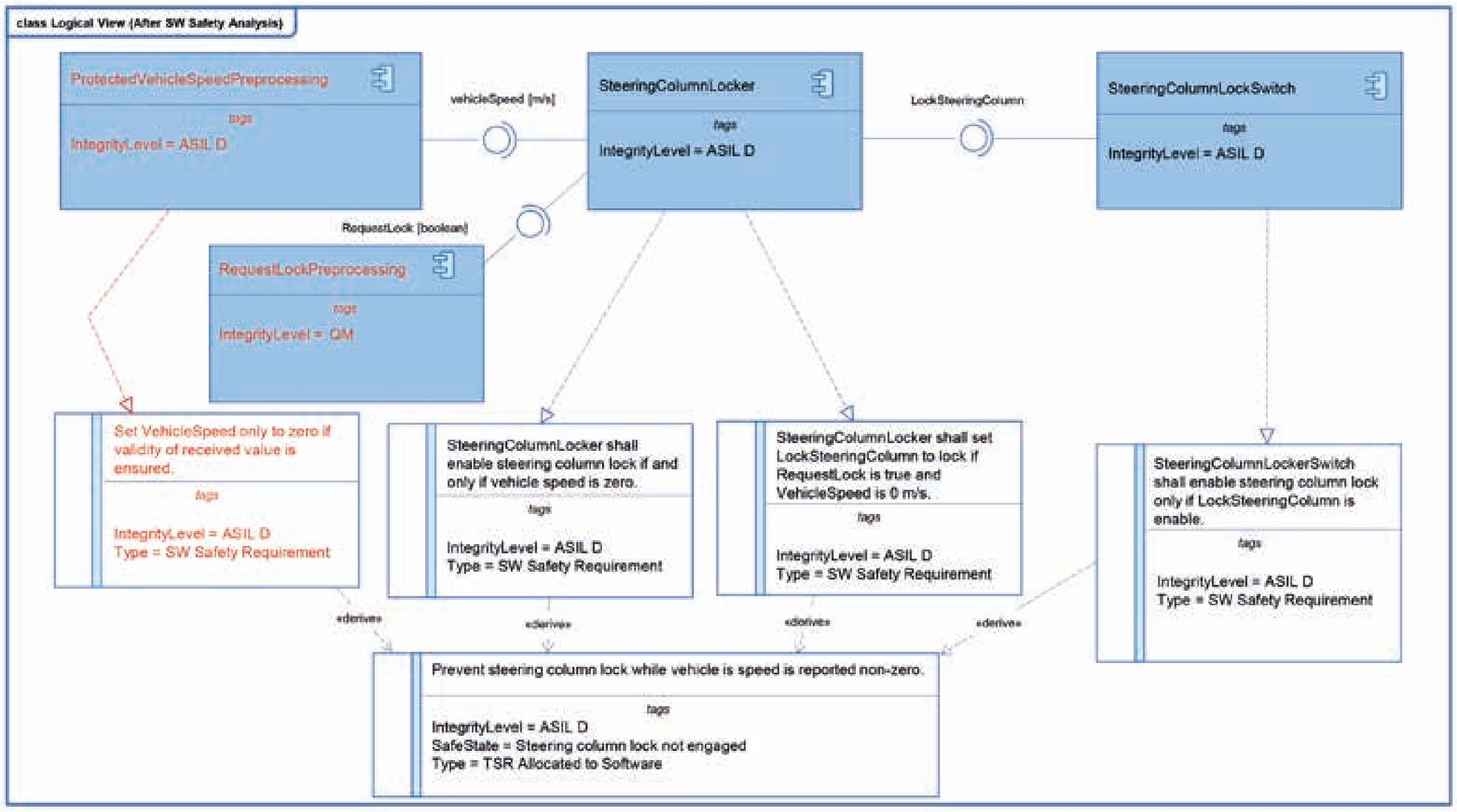

class Logical View (After SW Safety Analysis)
ProtectedVehicleSpeedPreprocessing
tags
IntegrityLevel = ASIL D
vehicleSpeed [m/s]
SteeringColumnLocker
tags
IntegrityLevel = ASIL D
LockSteeringColumn
SteeringColumnLockSwitch
tags
IntegrityLevel = ASIL D
RequestLock [boolean]
RequestLockPreprocessing
tags
IntegrityLevel = QM
Set VehicleSpeed only to zero if validity of received value is ensured.
tags
IntegrityLevel = ASIL D
Type = SW Safety Requirement
SteeringColumnLocker shall enable steering column lock if and only if vehicle speed is zero.
tags
IntegrityLevel = ASIL D
Type = SW Safety Requirement
SteeringColumnLocker shall set LockSteeringColumn to lock if RequestLock is true and VehicleSpeed is 0 m/s.
tags
IntegrityLevel = ASIL D
Type = SW Safety Requirement
SteeringColumnLockerSwitch shall enable steering column lock only if LockSteeringColumn is enable.
tags
IntegrityLevel = ASIL D
Type = SW Safety Requirement
«derive»
«derive»
«derive»
«derive»
Prevent steering column lock while vehicle is speed is reported non-zero.
tags
IntegrityLevel = ASIL D
SafeState = Steering column lock not engaged
Type = TSR Allocated to Software

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 non-safety 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.

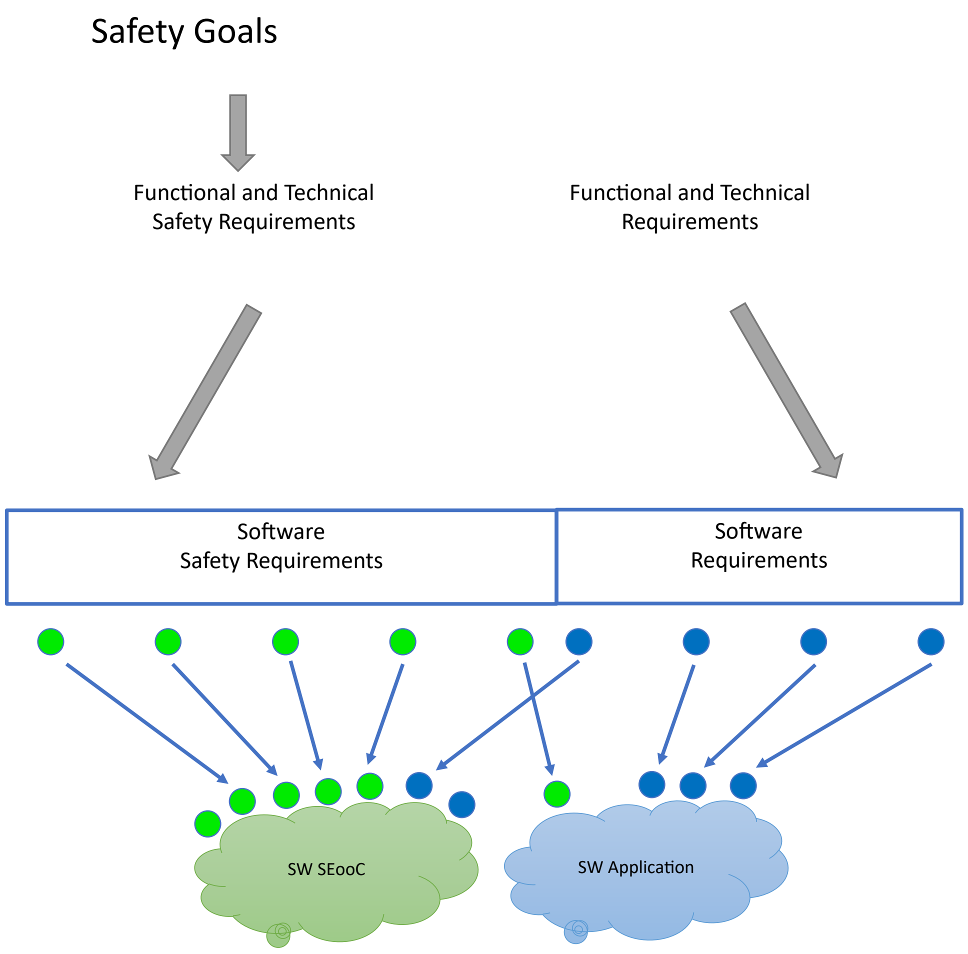

Safety Goals
Functional and Technical Safety Requirements
Functional and Technical Requirements
Software Safety Requirements
Software Requirements
SW SEooC
SW Application

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.

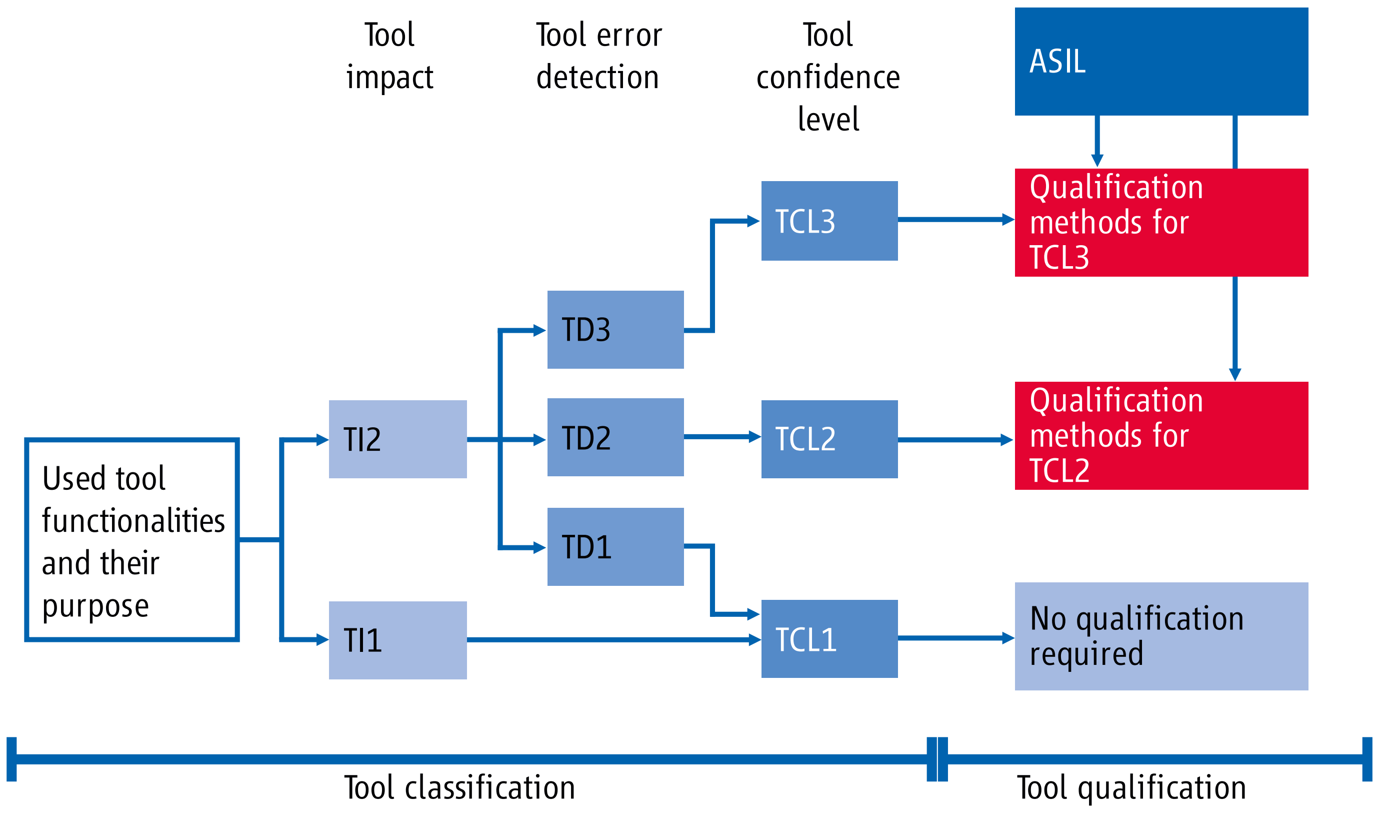

Tool impact
Tool error detection
Tool confidence level
ASIL
Qualification methods for TCL3
TCL3
TD3
Used tool functionalities and their purpose
TI2
TD2
TCL2
Qualification methods for TCL2
TD1
TI1
TCL1
No qualification required
Tool classification
Tool qualification

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

Tool	Use case	Failure mode	TI	Measures to detect or prevent mal-functioning of tool	TD	Rationale	TCL	Qualifi-cation 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 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
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

ZVEI:
Die Elektroindustrie

ZVEI - German Electrical and Electronic
Manufacturers' Association
Lyoner Strasse 9
60528 Frankfurt am Main, Germany

Phone: +49 69 6302-0
Fax: +49 69 6302-317
E-mail: zvei@zvei.org
www.zvei.org