Politecnico di Torino

Ingegneria Informatica (Computer Engineering)

Academic Year 2024/2025

Graduation Session October 2025

Functional Safety Analysis and Embedded Safety Mechanisms Implementation for ASIL C Automotive Actuator

Supervisors:

Massimo Violante

Jacopo Sini

Andrea Mongardi

Candidate:

Matteo Gravagnone

Acknowledgements

Many thanks to my family and the few people who were constantly by my side over these years and helped me believe in this growth opportunity

Matteo

Table of Contents

List of Figures

Chapter 1

Background

In the last decades, the adoption of micro-controllers in vehicles has increased greatly in passenger vehicles, as it is related to new functionalities at multiple levels. However, as safety is a key element for road vehicles, it is necessary to understand that these constitute new potential failure points, which can lead to unintended behaviors with risk of injuries and deaths.

Regulations across the world require the manufacturer to produce documentation that these systems are safe and it is in this context that the International Organization for Standardization, also known as ISO, published the standard known as ISO 26262, addressing the functional safety of electrical and/or electronic (E/E) systems in road vehicles.

This thesis covers part of the safety lifecycle for an actual industrial case, for which an analysis on the safety of the system has to be carried out and proper safety measures shall be implemented to reduce risk.

1.1 ISO 26262 standard

With the name ISO 26262[1] we identify the adaptation of the IEC 61508 standard[2] to E/E systems in road vehicles. At the time of writing, two versions were published, respectively, in 2011 and 2018. We will refer to the latest version, unless explicitly stated otherwise.

With safety being a key element in vehicle development and the increased complexity of systems, the standard (or, to be more precise, a series of standards) aims to reduce risks by providing a reference for the safety lifecycle. This series of standards is the state-of-the-art methodology used by manufacturers to demonstrate compliance with safety regulations in many markets.

In the following image, the structure of the series of standards is shown: eleven parts cover multiple phases of product development, and a V model is used as

reference both over multiple parts and for single phases. An additional one adapts the content for motorcycles, which is out of our scope.

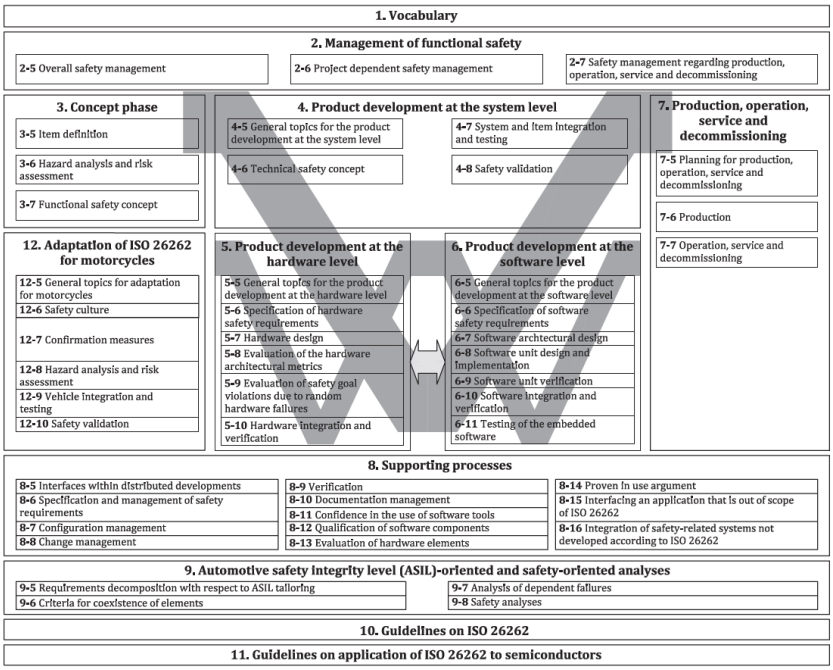

Figure 1.1: ISO 26262 structure

Several phases are declared for each part of the standard. Each one of them has "Prerequisites", consisting in documentation or information produced in previous phases, and activities, which may result in "Work products".

Over the standard, some tables are given for guidance in carrying out the activities. Tables can either be normative or informative and are made of entries or methods, which can either be consecutive or alternative.

For any given entry and safety level, one of three possible levels of recommandation is associated: "++" when highly recommended, "+" when recommended and "o" when it is not.

For consecutive entries, all methods recommended (whether highly or not) for a given safety level apply. Explanations are expected when substituting one of these methods with an unlisted one, or when some of them are not adopted for the given activity.

For alternative entries, appropriate combination of methods shall be adopted, with a bias for the highly recommended ones.

In this work, we will focus on elements discussed mainly in Parts 3 to 6 and 11, in addition to any terms introduced in the Part 1, named "Vocabulary".

1.2 Concept phase

Before diving into the hardware and any development, the Concept phase requires us to start by defining the *item*, which is the system (or combination of) to which ISO 26262 is applied. As anticipated, these involve E/E *elements* (system, components, hardware parts or software units) providing some functionality at vehicle level. No indication regarding their scope is given: this applies to systems that target safety, comfort, performance.

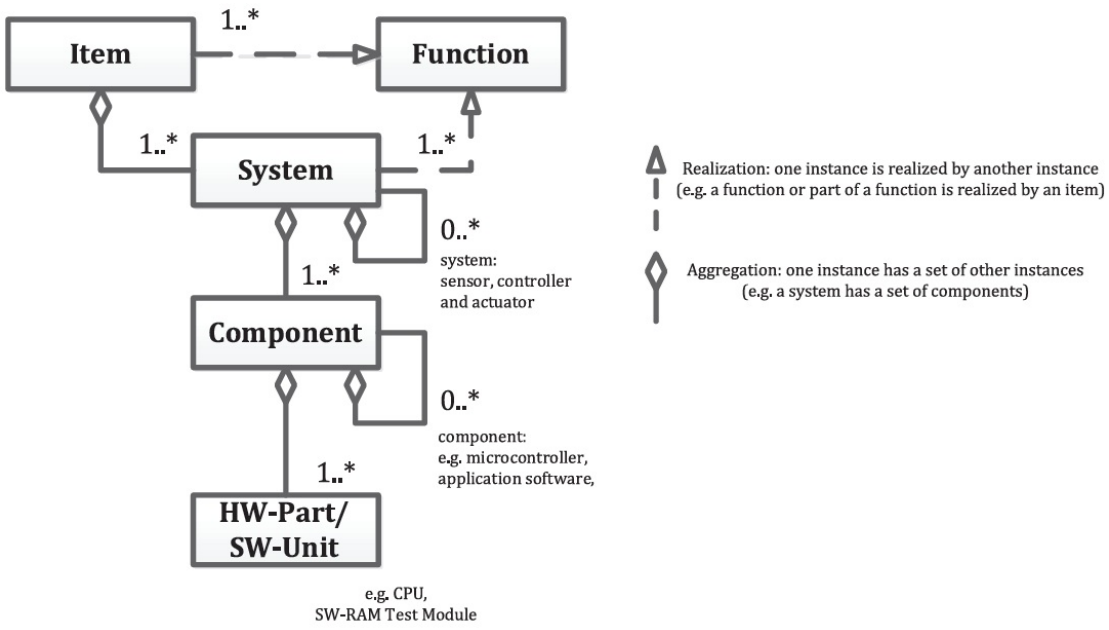

Figure 1.2: Relationship between item elements taken from Part 10 of [1]

While this specific item will not be explicitly defined in this essay, it is possible to say that, as it is composed of at least one system, it has at least a sensor, a micro-controller unit (MCU, also written as μC) and an actuator.

The next step consists in the Hazard analysis and risk assessment (HARA), which identifies hazardous events of items and specifies safety goals and ASILs to prevent or mitigate associated hazards, ultimately reducing risk. *Risk* is the combination of probability that an *harm*, injury or damage to people, occurs and its *severity*, in other words, an estimate of the extent. *Faults* are abnormal conditions which could lead an element or item to *failure*, a termination of the planned behavior.

This is a crucial step, as an incorrect evaluation of the Automotive Safety Integrity level, from now on called ASIL, specifies either insufficient safety requirements, with risk not being mitigated enough, or excessive requirements with an

increased cost needed to contain risk more than necessary.

All the different scenarios possibly occurring during vehicle operation (*operational situations*) shall be identified with the operational modes (i.e., the item in question being actually used in the scenario). With hazards being derived with a variety of approaches, which may involve reports and Failure Mode Effect Analysis, all hazardous events are determined with combinations of operational situations and hazards.

For each event, three important aspects need to be evaluated: *Severity* **S**, *exposure* **E**, which indicates the probability of being in a given scenario which can be hazardous if it coincides with a failure mode, and *controllability* **C**, which measures the ability to avoid a given harm by reaction of people. Severity targets the extent of harm to people and spans from S0, in absence of injuries, to S3 with fatal or life-threatening injuries. The ISO provides a baseline for the evaluation of this parameter by AIS stages, but as the data may vary, based on elements like accident variability or medical research or treatment, different injury scales may be better suited for a given analysis.

Exposure can be classified into five possible levels, from E0 being the lowest (which actually does not require further analysis, as it is associated to very unlikely phenomena) to E4, the highest. The values can be estimated with two approaches, either on duration of the situation or frequency of ending up in the considered situation.

Controllability spans from 0 being generally controllable, to 3 for difficult to control or uncontrollable. Table B.6 of Part 3 provides some examples of classification which could support other procedures.

Severity class	Exposure class	Controllability class		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A[a]
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

[a] See 6.4.3.11.

Figure 1.3: ASIL determination table given in [1]:3

Part 3 of the standard provides the table, also reported in figure 1.3, to compute the ASIL, for each hazardous event, given the values of S,E and C. If the value obtained is QM, no ASIL is attributed as it is considered a Quality Management problem, otherwise the unacceptable risk grows as we go from ASIL A to ASIL D.

For each hazardous event a *safety goal*, which states safety requirement as functional objective, without considering any technical implementation to avoid unreasonable risk, is determined, with an associated ASIL given from the event itself.

In many cases, similar safety goals collapse into a single one, with multiple events eventually being covered by a single goal. In this case, the ASIL assigned to the combined safety goal is the **highest** among the covered ones.

The steps performed to carry out HARA for our application are not part of the scope of this thesis, however it is necessary to know that an **ASIL C** has been assigned.

1.3 Hardware development

Our application case belongs to the widespread case of an MCU being integrated into the rest of the system to provide a given functionality. As one may expect, lots of elements, with the MCU being one, are not developed in the context of an upper-level item, but integrated into it.

According to the ISO standard, these are called *SEooC*, safety-related elements which are not developed in the context of a specific item and could potentially contribute against safety goals.

Part eleven of the standard will help us clarify some concepts about its application to semiconductors, while part 5 gives insight on product development at hardware level, giving us information also when integrating an already existing element, as the MCU, into the overall system.

The MCU chosen for our case is a variant of an Infineon Aurix™ TC36x, having two TriCore CPUs, memories and several peripherals. In addition to the general documentation for this kind of devices, some of which is available freely on their website, additional documentation regarding their use in safety-related applications is given, as this was developed as an SEooC.

The Safety Manual is a key document, as it explains safety mechanisms that are provided or supported in the MCU, along with assumptions of use.

In particular, the manufacturer of this device claims support up to ASIL-D applications and gives information on how to reach given performances for the safety goals of the specific use case.

It is powered by an Infineon TLF35584, a Power Management IC also suited for applications up to ASIL D.

In this thesis we focus mostly on the power supply, the CPU, memories and some other peripherals.

Table D.1 of Part 5 provides typical failure modes for different classes of E/E elements: we will explore these later on, but a preview includes Under and over voltage for power supply, errors in memories and incorrect results by the CPU.

At this point, we have to introduce some new definitions from Part 1 of the standard:

- *Safety mechanism*, a technical solution to detect and mitigate faults or control or avoid failure, maintaining the intended functionality or reaching safe state
- *Failure rate*, denoted as λ the probability density of failure divided by probability of survival for a hardware element
- *Diagnostic coverage*, or DC, is the percentage of failure rate either of an hardware element or of one of its failure mode, detected or controlled by the implemented safety mechanism

Tables D.2 to D.10 give us some safety mechanisms which could be found or implemented for the classes of elements, along with plausible values for achievable diagnostic coverage. This will also be explored further later on.

1.4 Software development

The ISO 26262 series of standards[1] also targets the software development process in part 6. Before the specification of safety requirements for the software, its design and the following development and testing phases, the software development environment and language shall be chosen.

These should be suited for the development of the embedded software in accordance with the guidelines and work products required by the standard.

The programming language adopted in this application case is **C**, a widespread language in embedded environments due several factors, such as efficient code once compiled, existance of compilers and the low-level hardware abstraction. The MCAL (Micro controller Abstraction layer) by Infineon, which is a set of source code files given to customers to abstract the components and peripherals of an MCU, is written in this language.

Unlike languages such as Rust, which enforce strict compiler checks to ensure code correctness, catching issues like concurrency errors at compile time, C offers programmers a high degree of freedom, which, while powerful, also makes it easier to introduce mistakes.

In order to address the criteria for suitable safety-related programming, additional guidelines shall be adopted if these are not addressed adequately by the language itself.

For the C language, a reference coding guideline, also mentioned in the ISO standard, is MISRA C[3].

1.4.1 MISRA C

It is a set of guidelines to reduce probability of mistakes during C programming, as C can lead to undefined behaviors, misuse and little run-time checking.

A guideline can be classified as either *rule*, for which a complete description is provided and the source code is sufficient to verify compliance, or *directive*, for which additional information is required.

For *rules*, a further classification applies: *decidable*, if a program can verify compliance in every case, *undecidable* otherwise, typically for dynamic cases such as those that depend on run-time values.

A guideline can also be classified as either *mandatory* or *required*, for which deviation is allowed but shall be formally justified, or *advisory*, if they should be followed as reasonably as possible but are not necessary.

Directives revolve around broader phases of development, while rules characterize all the aspects of the source code.

Proper adoption of MISRA C into a software development process, which is expected to be integrated into other processes targeting correctness starting from the requirements, requires programmers to make a compliance matrix, along with any deviation justification.

1.5 3-level monitoring concept

With this in mind, it is now possible to introduce the E-GAS Monitoring Concept[4], a monitoring concept standardized by the EGAS Workgroup, made up of several German manufacturers and, in its form, intended for engine control units.

In particular, for our case, the most important part is the *3 Level monitoring concept*, as it plays an important role in how functionalities are spread out.

- Level 1, called *function level*, contains the control functions, implementing the intended functionalities
- Level 2, called *function monitoring level*, monitors the activities of level 1 and the output produced, by comparison with permitted values, and triggers reactions in case of fault (or monitors fault reaction of level 1, in case these cannot be generated by level 2 itself).
- Level 3, called *controller monitoring level*, is independent of the application-specific implementation, as it is responsible for monitoring the correct operation of the MCU.

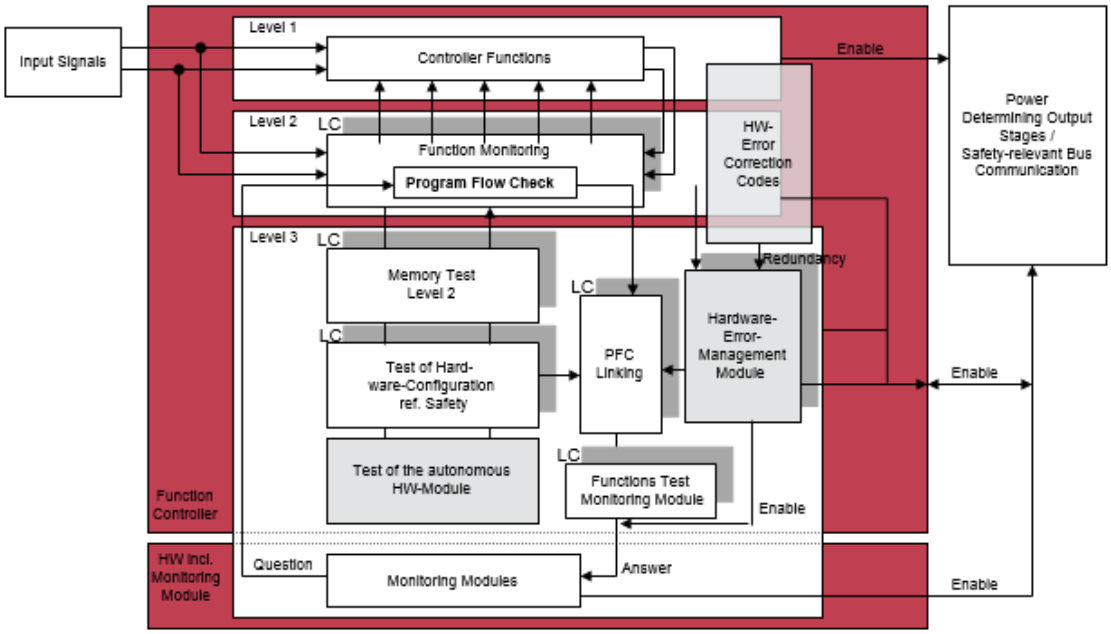

Figure 1.4: 3-level concept with lockstep-core[4]

The focus of the activities carried out for this thesis, and explored in the next chapters is on level 3. Several elements, such as the processing units and memories, are therefore subject to checks which, as the monitoring concept says, can be performed as a combination of internal error detection hardware and software functions.

Figure 1.4 provided in [4] shows how this concept works and, to a wider extent, how safety-related functions play an important role in the overall item, whose functionality is implemented in the first yet smallest level.

In the next chapter we will focus on the combination of software functions and hardware measures which mitigate random hardware failures and lead to satisfactory metrics. As it is not possible to disclose specific documents by the MCU manufacturer, parts 5 and 11 of ISO 26262, along with further available documentation, will be discussed.

1.6 Thesis structure

As an introduction to the problem has been provided, the core of the work can be further discussed: the series of standards can be explored in more detail, posing the foundation for application-specific design choices.

The following structure is adopted for this document in order to cover as

reasonably as possible the activities carried out, part of which unfortunately cannot be disclosed.

- **Chapter 2** will provide the state of the art of hardware development for automotive applications by ISO, providing typical failure modes affecting MCU's elements and common safety measures which increase coverage.
- **Chapter 3** will focus on safety mechanisms chosen and implemented for the specific case. As this document will not be marked as confidential and most of the documentation and software developed is under NDA, this will be approached in a sensible way.
- **Chapter 4** will conclude the document, providing final considerations.

Chapter 2

State of the art

In this chapter, we are going to further analyze the adoption of a microcontroller into the item, with a focus on some of the elements it is composed of. Typical failure modes for these elements will be provided, along with safety measures.

For this purpose, Parts 5 and 11 of the standard will be considered, together with this[5] application note by Infineon, publicly accessible from their website

2.1 Safety mechanisms timing

When adopting safety mechanisms into the item, it is necessary to consider the timing regarding faults and their consequences.

Hence, several time intervals are defined:

- *Fault tolerant time interval (FTTI)* is the minimum interval from a fault happening to the hazardous event, in absence of safety mechanisms.

  It has direct implications on safety goals and safety mechanisms, which have to maintain or reach a safe state within this interval.

- *Diagnostic test time interval (DTTI)* is the interval between executions of diagnostic tests for a given safety mechanism.

- *Fault detection time interval (FDTI)* is the time between a fault occurring and its detection. With lower values of DTTI, this interval becomes shorter, however this implies greater overhead due to frequent diagnostic tests.

- *Fault reaction time interval (FRTI)* is the time-span between the detection of the fault and safe-state being reached.

- *Fault handling time interval (FHTI)* is the sum of FDTI and FRTI. It is a property for a given safety mechanism.

The following figure 2.1, given in the Vocabulary of ISO 26262[1], shows their relation.

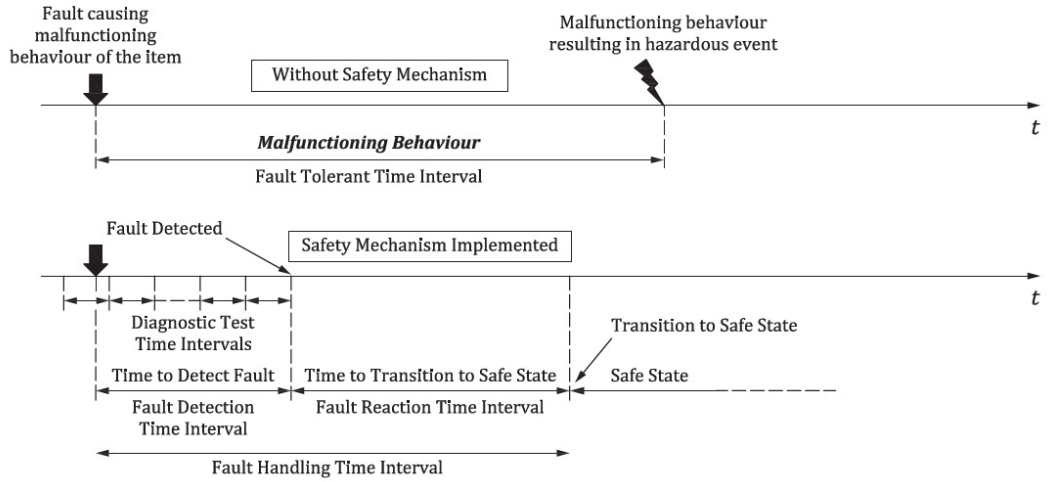

Figure 2.1: Safety time intervals, from ISO26262-1

When designing a safety mechanism and its response time, this crucial property shall always hold true: $FTTI > FHTI = FDTI + FRTI$. Otherwise, there is no point in adopting the measure, as the hazardous event cannot be prevented and a safe state cannot be reached in time.

DTTI shall be chosen appropriately: low values lead to lower FDTI, however more time is spent doing monitoring, which hampers both normal functions and other safety mechanisms.

Moreover, it is possible to have instances in which a single detection of a fault is not valid to raise an alarm, and filtering is needed against disturbances, dirty data and other phenomena, in a similar way to that of debouncing. In this case, lower values of DTTI shall adopted to ensure timely detection and handling of the fault.

The discussion so far implicitly assumes that the safety mechanism activation leads to safe state being also being reached inside the FTTI, which is not always the case.

In this case an *emergency operation* can be adopted. It is an operating mode in which performance is degradated, it shall be initiated before FTTI and safe state shall be reached inside another interval, the emergency operation tolerance time interval (EOTTI).

This mode plays a role in the "Warning and degradation strategy", adopted when it is not possible or convenient to stop providing entirely a functionality. As the name implies, warnings are issued to the driver for proper reaction and performance is limited when compared to the nominal one.

One example could be an electronic accelerator or steering wheel actuator

which, in case of fault, shall not shut down abruptly. Instead, limited performance (restricted acceleration or steering wheel angle) give room to move aside before risk cumulates and becomes too high.

2.2 Hardware metrics

The addition of safety measures contributes, along with design changes, to improved safety properties of the item. Several metrics can be defined, some of them requiring us to classify faults as follows:

- *Single-point fault* is a hardware fault in an element that leads directly to violation of safety-goal and no fault in the element is covered by any safety mechanism
- *Multiple-point fault* is an individual fault that, combined with other faults, could lead to multiple-point failure (and, ultimately, to a violation of a safety-goal)
- *Latent fault*, a multiple-point fault not detected by a safety mechanisms nor perceived by the driver before reaching failure
- *Safe fault* for faults that do not increase greatly the probability of violating a safety goal when they occur
- *Residual fault* is a portion of an hardware fault, which leads to violation, not controlled by a safety mechanism

The following figure 2.2 from Part 5 gives us a visual representation of this classification.

Another important result to be considered during the lifecycle of the product is the evaluation of the hardware architecture metrics, which demonstrate that the design is suitable for the item, as it is able to detect and control hardware failures.

These are the *Single-point fault metric*, from now on called SPFM, and the *latent-fault metric*, from now on LFM. Higher values are associated to a more robust item with respect to either single-point faults or latent faults.

These can be achieved by adopting suitable safety measures or by designing an inherently safe product.

If we adopt the same graphical representation as the standard, high values indicate a lower proportion between the area linked to the considered faults and the overall circle.

We can take note of the target values which are provided in 8.4.5 and 8.4.6 of Part 5.

In particular, for safety goals associated with ASIL C the SPFM has to be $\geq 97\%$, while LFM has to be $\geq 80\%$.

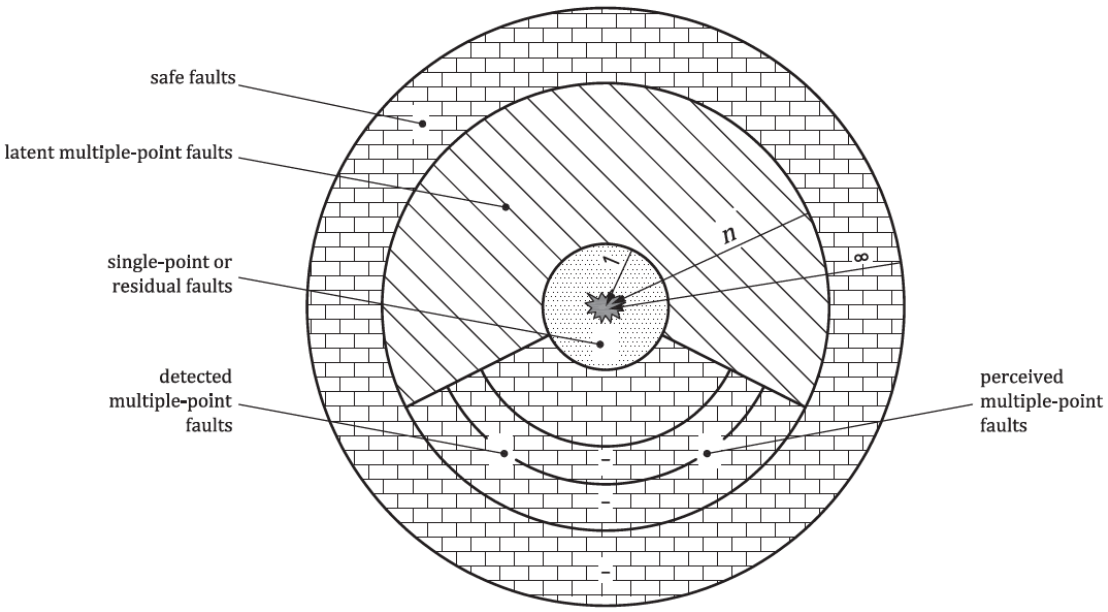

Figure 2.2: Fault classification of hw elements, from ISO26262-5:C.1

2.2.1 Failure rate

We have defined failure rate in chapter 1 as the probability density of failure divided by probability of survival for a given hardware element.

These values, denoted with λ, are necessary to compute SPFM and LFM defined in the ISO 26262 standard, which assumes these to be constant.

In order to provide these values, models for failures are needed, with one of the most popular ones being the *bathtub curve* reported in 2.3, which assumes that failure rate is exponential in the earliest phases of the product life, nearly steady for some time and exponentially increasing when it is worn out.

Simplifications of this model, useful for λ evaluation, include defects during production, and up to the item being used by the customer, being filtered out, and "wear out" failures to be minimal during the expected operational lifespan of the product.

The model can be customized, as screening and stress procedures during production can improve the filtering performance, or it can be based on several fault models for different failure modes, which help in the safety analysis by considering separately the failure modes.

Numerical values can be derived from the models, testing phases and industry books. For semiconductor products, it is common to expect failure rates for the silicon die, for the enclosure and for connections between elements.

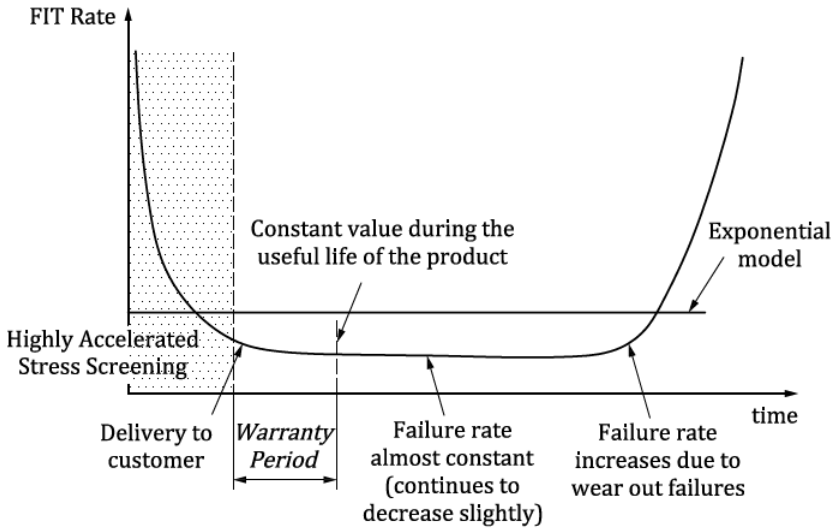

Figure 2.3: Bathtub curve for failure rate evaluation

The manufacturer can then select a suitable λ for their product, based on multiple factors. For example, it can be equal to the "exponential model" value, shown in figure 2.3, calculated for a given confidence level (in the figure, 70%). This approximation could be more appropriate when the product is expected to operate also outside of its nominal lifespan; on the other hand, a less cautious evaluation could assign to λ the value observed during its operational lifespan (when the product is expected to be delivered to the end user, with the negligible decrease over time), making it appropriate only when inside this period.

Failure rate can be expressed with FIT units, which stands for Failure In Time and 1 FIT = 1 failure per 10^9 hours or as failures per hours, h^{-1}.

In addition, it is possible to find in literature the MTTF metric, which stands for Mean Time To Failure, and is the reciprocal of the failure rate expressed as failures per hour. Therefore, one hour is the base unit for it.

It is possible to convert from FIT to failures/hour by dividing by 10^9, the opposite can be done by multiplication.

Once failure rates have been computed or derived, in order to compute the SPFM and LFM, it is necessary to classify them based on the corresponding failure modes. A subscript is added to λ to refer to the specific fault types.

Figure 2.4, provided in [1], shows a suitable flow diagram for classification, starting from the failure modes.

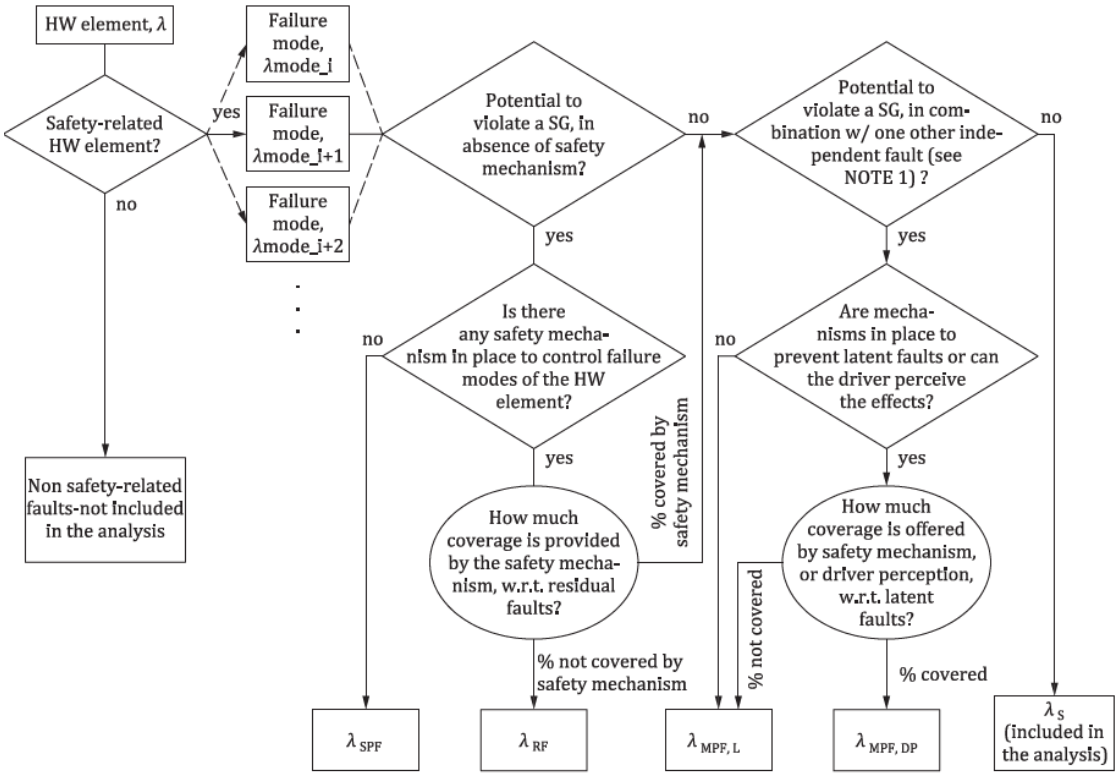

Figure 2.4: Failure mode classification flow diagram, from ISO26262-5:B.2 [1]

SPFM can be computed as follows:

$$1 - \frac{\sum_{SR,HW} (\lambda_{SPF} + \lambda_{RF})}{\sum_{SR,HW} \lambda} = \frac{\sum_{SR,HW} (\lambda_{MPF} + \lambda_S)}{\sum_{SR,HW} \lambda}$$

where the sum symbol denoted the sum of the λ of safety-related hardware elements and $\lambda = \lambda_{SPF} + \lambda_{RF} + \lambda_{MPF} + \lambda_S$.

LFM, on the other hand, is given by the following:

$$1 - \frac{\sum_{SR,HW} (\lambda_{MPF,L})}{\sum_{SR,HW} (\lambda - \lambda_{SPF} - \lambda_{RF})}$$

2.2.2 PHMF

The ISO 26262 part 5, with its Clause 8, requires the evaluation of the aforementioned metrics. However, that is not sufficient for the evaluation of the residual

risk of safety goal violations, due to random hardware failures.

For this purpose, the clause 9 introduces the *Probabilistic Metric for random Hardware Failures*, in short *PHMF*, for a quantitative analysis.

This additional metric covers a gap introduced with the other metrics: even though they were based on failure rates, which is linked to the probability over time, the ratio cancels out the time dimension, leaving only a percentage expressing relative robustness with respect to the faults.

The standard provides once again possible target values, with a constraint of PHMF $\leq 10^{-7}\, h^{-1}$ for ASIL C. This can be also expressed, by FIT units, as an upper limit of 100 FIT.

An approximated value of PHMF can be obtained as follows:

$$PHMF_{EST} = \sum (\lambda_{SPF} + \lambda_{RF}) + \sum (\lambda_{DPF_{det}} * \lambda_{DPF_{latent}} * T_{lifetime})$$

Generally speaking, the first sum is the greatest contribution to PHMF, as single-point faults and residual faults greatly contribute to safety goal violations. However, an additional element can be found, as the probability of dual point failures, based on a detected fault and an undetected latent fault, may not be negligible over the lifetime.

ASIL	Single Point Fault Metric	Latent Fault Metric	Residual PHMF
B	$\geq 90\%$	$\geq 60\%$	< 100 FIT
C	$\geq 97\%$	$\geq 80\%$	< 100 FIT
D	$\geq 99\%$	$\geq 90\%$	$<$ 10 FIT

Table 2.1: Target fault metrics based on ASIL

2.3 Failure modes and safety mechanisms

It is possible to discuss about the elements in a system to which ISO 26262 applies, covering some of the faults that could happen and exploring common mechanisms to detect them and increase the aforementioned hardware metrics.

Our reference will be provided by the standard itself, which targets E/E components either at a generic level or focused on semiconductor ones; however, the *Failure Mode, Effects and Diagnostic Analysis*, hereafter referred to as FMEDA, is a common methodology in the industry to determine failure causes in the device and their effects on the system.

Manufacturers of safety-related components, developed as SEooC, generally provide support for the evaluations of the ISO random-fault-related metrics with a configurable FMEDA template that may be modified for the application. This

gives guidance on how to tailor the design choices according to the desired ASIL. However, it is distributed only under NDA and we can disclose only some elements available to the public.

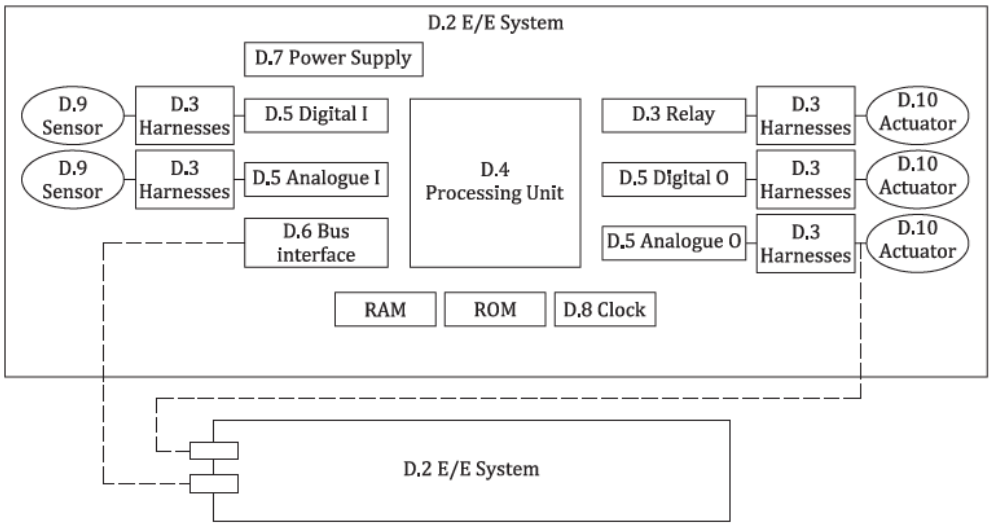

Figure 2.5: Generic hardware of a system, from ISO26262-5:2018:D.1

It is also possible to observe metrics for Hard Errors, which are permanent, and Soft Errors, with a transient before going away, as different elements and safety measures can affect differently these types.

In figure 2.5 an overview of a safety-related system is reported: we can now dive into the several kinds commonly found an discuss at a part level.

2.3.1 Power supply

This element is a semiconductor component which handles both digital and analog signals. With its main purpose being to provide specific voltage levels to the MCU, typical failure modes involve, according to D.1 in iso 26262-5[1], drift and oscillation, undervoltage and overvoltage.

Although the primary functionality is to provide analog voltage signals, safety mechanisms can either alter these signals directly or generate digital outputs to the MCU to signal failures for countermeasures.

Here, we will list some of the approaches commonly found in such devices.

- *Voltage clamp* refers to circuitry that limits voltage in a *node*, with at least an upper or lower bound. Voltage clamps are typically used for transient events.
- A *Current limiter* is used to reduce output current using transistors, ICs, or other components, thereby reducing stress on electronic.

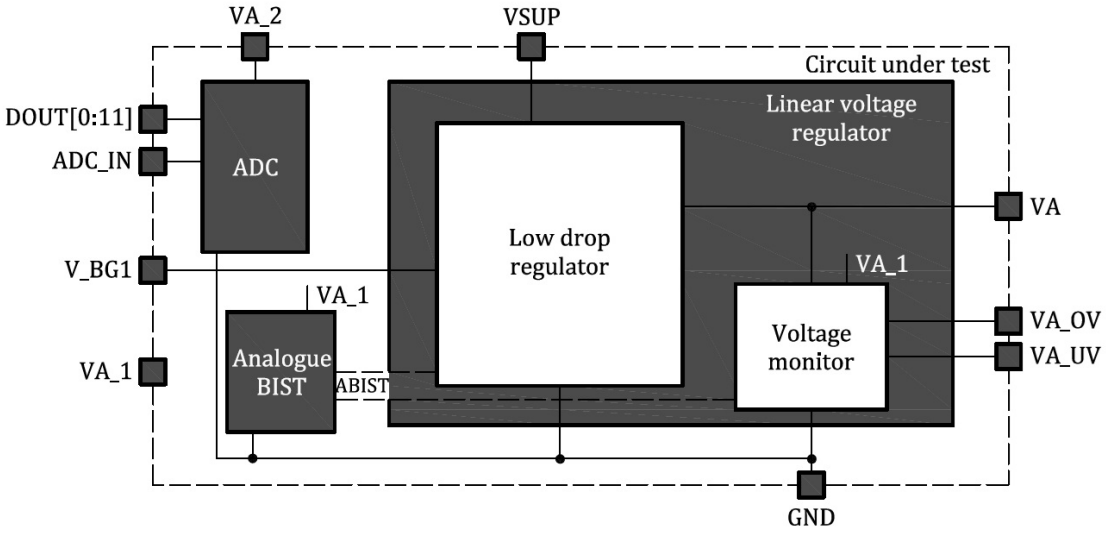

Figure 2.6: Mixed signal hardware element, from ISO26262-11:D.1 [1]

- *Overvoltage and undervoltage monitoring* are used for bounded voltage signals. However, unlike a clamp, this does not limit the amplitude. Instead, the signal is compared to reference signals (which shall be independent to avoid common cause faults) and an output, which can be digital. While voltage clamps actively protect the circuit, these monitoring ensure detection for system-level countermeasures.

- *Resistive pull-up/pull-down* is another way to limit voltage signals. It involves resistive elements that either ground a node or connect it to supply voltage when the driving signal is disconnected.

  Their most common application is with I/O pins, common in MCUs and also found in power supply interfaces with the supplied microcontroller.

  These circuits define default voltage levels for pins, preventing unexpected behavior when a driving signal is absent.

These strategies could suffer from other faults and, based on the desired diagnostic coverage, require additional measures.

One of them consists of *Analog Built In Self-Test*, which improves coverage by verifying the functionality of diagnostic elements under simulated faults. For this element, it is obtained by injecting analog signals. The idea of self-tests is widespread also for other E/E elements.

Power supplies generate heat due to conversion losses, dissipation with resistive elements, conduction losses and control logic. Harsh environmental conditions can aggravate their sensitivity to thermal issues. For this reason, *Thermal monitoring*

is an important measure to signal faults when the measured temperature exceeds a specified threshold.

Countermeasures can be taken to avoid component failure or system malfunctions.

2.3.2 CPU

The processing unit is a key element in any MCU, responsible for executing instructions in a given flow. Typical failure modes for it include, at the very minimum, the following: instruction flow not executed, execution of unintended instruction flow, incorrect flow timing, incorrect results.

Some concepts behind safety measures include self-tests, redundancy either in software or hardware and different types of monitoring.

Let's take a look at some of them, starting from those targeting expecially incorrect results.

- *Hardware redundancy*, with an high achievable diagnostic coverage according to the ISO, consists of a step-by-step comparison of the results produced by different processing units.

  With reference to the *Dual Core Lockstep*, a specific version which is also present in the specific MCU considered, two symmetrical processing units are in the same die, with one being *Master*, i.e. the "nominal" one, and the other being the *Checker*.

  These run the same operations, either in parallel or delayed by a fixed amount of time, in order to avoid certain disturbances, and results are compared: in case of mismatches there are errors and some action is taken, generally with some signal to report the fault or reset.

  With this approach the same code can be used for both units; however, it has disadvantages both in potential performance, as the secondary unit is relegated to a checker role, and failures, as in this form systematic errors are not covered: certain kinds of failure, called [common cause failures] or *CCF* will have the same impact on both cores, leading to wrong behavior without any mismatches.

  Proper implementation of this technique addresses these weaknesses.

- *Software based self-tests* detect failure by software means. A number of patterns is used to test correctness of registers or functional units, with diagnostic coverage generally lower than hw-redundancy, as not all memories or units can be thoroughly tested.

Respectively, IEC 61508[2] reports the need for at least two complementary data patterns for failure detection through patterns, or walking-bit as an example to test data and address registers inside the processing unit.

The application note from Infineon [5] expects the adoption of these tests for non-lockstep CPUs in the device.

- Hardware supported self-tests leads to improved coverage with respect to sw-based tests, as special hardware structures, also with random generated inputs, is used.

  However, these require a more complex design of the processing unit and may interfere with normal operation.

  For this reason, a distinction which can be found also in literature, reported also in [6], is between *online BIST*, which has strict timing constraints, as it shall run inside the diagnostic time interval during operation, and *offline BIST*, relegated to the initialization phase as it is slower.

  Generally, the former improves SPFM, while the latter is used against latent faults.

  The application note[5] by Infineon mentions self-test of the *lockstep comparator*, which is the unit comparing the results from master and checker cores.

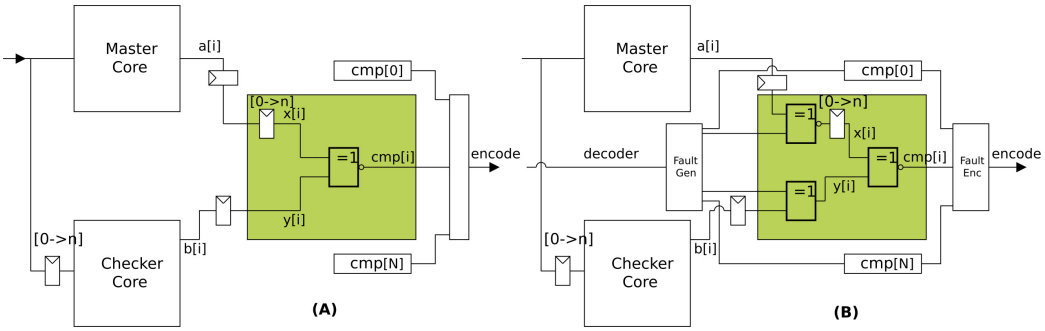

Figure 2.7: Lockstep core comparator, from Infineon AN[5]

The document suggests that it is hardware-based as it uses fault injection, another technique widespread in the industry, however it is *continuously running*, which is atypical.

In order to observe valid behavior, it is also necessary that the instruction flow is correct both temporally and logically. For this purpose, *Programme sequence monitoring* measures shall be adopted.

- *Watchdogs* are key elements in verifying timely correct operations. At their core, these can be seen as timer devices which do not share the clock source with the unit they are monitoring.

  Over time, they are triggered to monitor the CPU and program sequence. As a result, CPUs shall answer.

  ISO 26262 expects different levels of achievable diagnostic coverage, based on *time-window* being present or not.

  A lower coverage is to be expected without the time-window, as only a maximum time interval for triggering the watchdog is specified. This means that, while events like task death could be recognized in absence of response, no error is observed on too fast answers by the processor, which could very well be caused by wrong programme sequence.

  An higher level can be achieved with time-window, as responses by the CPU are valid only when received inside a given period of time, with also lower bounds.

- *Logical monitoring* is another important element in evaluating correctness of the sequence. As the name implies, it checks parts of software. Checkpoints are defined in monitored software entities, covering paths which can be dangerous in case of faults.

In safety-related devices, the sequence is monitored both temporally and logically, with Program Flow Monitoring techniques that analyze sequence results and timing properties through external devices, typically advanced watchdogs.

2.3.3 Memories

With the term memory we refer to any semiconductor element that stores data, which shall be maintained over time and be accessible at any point to continue with the required instruction flow.

The smallest unit is the *bit*, an element which stores a low value, 0, or an high value, 1.

A variety of technologies can be used, as different operating conditions, volatility needs and design apply.

Here are typical fault models for memories:

- Stuck-at Fault (SAF) indicates a memory cell being stuck at a given logic value, due to some physical problem.
- Stuck-Open Fault (SOF) indicates a problem in accessing the memory cell, whose content is not available. This may arise from defects such as open circuits or damaged transistors and any attempt to read the value is unreliable.

- Addressing Faults (AF) and Addressing Delay Faults (ADF) are observed when it is not possible to address and, therefore, reach the memory cell at all or in nominal time.

- Coupling Faults (CF) happen when operation on one cell affects another one.

These are generally intended for RAM (Random Access Memory), but hold true also for FLASH and other technologies.

Safety mechanisms can either apply to all memories or depend properties, like *volatility* or *variability*.

With volatile, we refer to a memory that shall hold data as long as it is powered, while non-volatile ones keep data even without power supply.

Variable memories are subject to modifications in time during operation, while invariable ones, commonly named *read-only*, shall not change its content after the writing procedure.

- *Parity bit* is common technique found not only in memories, but also in telecommunication, which allows to detect corruption for an odd number of bits in a word (which we will consider, for discussion purposes, groups of 8,16,32,64, or 128 "consecutive" bits), but not for even number of faulty bits.

  A bit is added to every word and its value is based on the number of 1's. In case of *even parity bit*, the extra bit is set to 1 when the number of ones is odd. This makes the overall word have an even number of bits equal to 1.

  The opposite holds when *odd parity bit* is adopted, as it makes the word have an odd number of 1's.

  Upon reading the word, parity is checked and, if the number of 1's differs from the expected one, a fault is detected. This approach has limited capabilities, as it does not help in detecting addressing failures, unless the address is included in the parity check and, most noticeably, it only has limited detection properties, as the corrupted bit is unknown, leaving no *correction* possibility.

- The previous technique is a basic form of *Error-detection-code*, where redundant bits are added to the original data bits for detection and correction purposes.

  In particular, Hamming codes are used, with several distances (the lowest one being 2, resulting in the parity bit) linked to the number of additional bits.

  Figure 2.8 shows, at high level of abstraction, the core concept behind ECC. Codes of several lengths are available on the TC3xx microcontroller family based on the specific memories, up to three-bit error detection and two-bit correction for Flash memories[5].

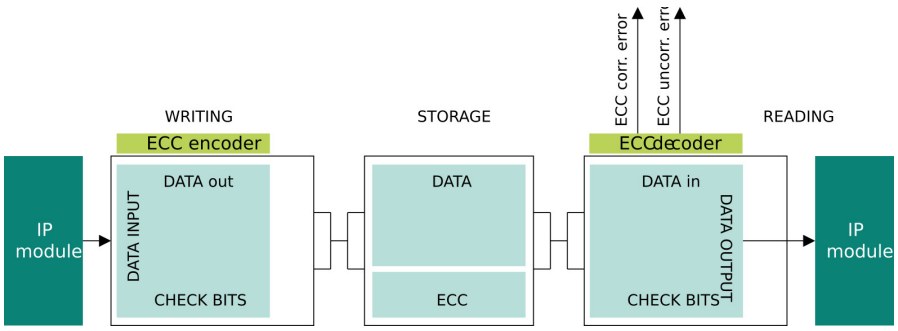

Figure 2.8: ECC concept, from Infineon AN[5]

- *Checksum* is another method for error detection, in which a block of memory is split in segments of equal size that are treated as binary numbers.

  The first two segments are added according to binary sum and result is added to the third segment and so on. Once all segments have been added, the one's complement is applied to the sum, resulting in an additional segment, the checksum, with bits flipped with respect to the sum.

  This value can be stored and used at a later point, when the same activity is performed, and the obtained checksum shall be compared to the stored one. Based on the application, checksum comparison could be substituted with the checksum added to the sum, and the result is complemented. If the complement is made of only 0's, no error has been detected.

- *Cyclic redundancy check*, CRC, is an enhanced checksum alternative that relies on polynomial-based code, further enhancing its detection properties and resulting in high diagnostic coverage.

  In this method the binary addition is replaced by binary division in which information block is the dividend and a suitably chosen generator polynomial is the divisor.

  Specific adoptions of CRC are indicated by the polynominal's degree, which determines the error detection performanc by indicating the max number of bursts. Common CRC are CRC-8, CRC-16 and CRC-32.

  Periodically, CRC is computed for the block of memory and compared to stored value.

- *Memory Signature* is a wider concept consisting in summarizing the content of a memory block in a far smaller number of bits. This can be achieved by polynomial division, with capabilities in terms of coverage that are on level with CRC.

It is mostly used for *invariable* memories either at word level (a signature for each relevant word) or block level.

This variety of methods mostly covers errors at the memory cell level and increases diagnostic coverage; however, it is still important to adopt self-tests, at startup or runtime, that can also target faults at decoder level, leading to proper metrics for the given safety level.

Specific self-test techniques for variable memories, as outline in IEC 61508-7[2], include:

- *Checkerboard*, which consists in itialization with alternating 0 and 1 patterns to detect static bits or addressing faults.
- *Galpat* initializes memory uniformly (all 0s or 1s), then single cells are inverted and the remaining cells are tested to observe any dynamic coupling or static bits.

These can be scheduled for execution and complement the methods general methods above, by covering their dynamic properties.

2.3.4 I/O

Input/Output (I/O) is the interface between a MCU and physical world.

Input allows the MCU to "see" by sensor measurements, used to represent physical parameters. Output allows the MCU to transmit signals to control things outside the system, by means of actuators.

ISO 26262 also targets possible failures caused by undetected faults in these interfaces, as these could affect the functionality provided by the overall system.

The following methods can be applied to achieve required coverage levels and detect failures in I/O units.

- *Test pattern* detects static failures by initiating a test, with predefined pattern, which produces a result that shall be compared to the expected one. Coverage increases as the pattern information, reception and evaluation gets increasingly independent from one another.
- *Code protection* gives detection capabilities that rely on information or time redundancy. Multiple approaches are possible, as additional bits can be used for digital channels and additional signals can validate analog IO.
- *Monitored outputs* can be used to detect several failures when duplication or redundancy of the outputs themselves is not possible. Independent elements compare the values with a tolerance range for values and/or timing.

- *Input comparison or voting* also provides for a comparison with a tolerance range. However, the mechanism becomes stronger when redundancy is used, with independent inputs.

  Based on the adopted number of independent signals, the majority that "shares" a value imposes itself.

2.3.5 Communication bus

The processing unit assumes that any instruction or data required to continues its cycle are provided in its registers. However, as most of the information required is stored or provided by other elements, such as the RAM, non-volatile memories and additional peripherals, a communication channel shall be adopted.

In MCUs multiple communication buses can be provided, as they generally handle different timing and throughput requirements.

Devices like a Direct Memory Access, or DMA, allow to transfer data between peripherals without the intervention of the CPU, which does not have to be busy handling interrupt to serve; however, also when it is adopted, the communication bus still provides the path for signals to go.

Faults for these result in repeated, lost, delayed, incorrect information or also faulty communication path, such as blocked access or incorrect receivers.

Safety mechanisms generally rely on redundancy, patterns and additional monitoring. In the scope of information exchange, the mechanisms seen for memories and I/O are partially useful also in buses.

We can start by taking a look to those that affect the received message.

- *Hardware redundancy* can be tailored by the MCU designer according to desired detection capabilities and coverage. The simplest form consists in one additional line in the bus, equalling an additional bit used for the *parity* check.

  Multiple lines can be added in a bus for stronger mechanisms that use error codes, of which Hamming codes and CRC are a part of.

  A *Complete* hardware redundancy can be reached when two buses are used for a single communication channel: the number of lines available is the double, as values between the channels are directly compared.

  One instance of this is a Dual channel FlexRay. It is still useful to remind that common cause failures, unless somehow targeted in design, can impact the diagnostic coverage which is otherwise high for this method.

- *Transmission redundancy* is used for transient failures detection, since the same information is shared multiple times in a row.

- *Information redundancy* expects data to be sent in blocks with a checksum or CRC.

  The receiver also computes the checksum/CRC and compares it with the obtained one, as we have already stated here.

  Also in this case, the choices for data block size, size of the checksum/CRC and, in the last case, the generator polynomial alter the detection performances.

The following methods can be used for detection of failures affect the message travel.

- *Frame counter* is used to detect frame losses.

  Frames are sets of data that are identified by a message ID. The counter is increased by one for each transmitted frame so the receiver can detect whether a message was lost, with a gap between a frame's ID and the next received frame's ID, or received in the wrong order, as the ID should always increase one by one upon correct reception.

- *Timeout monitoring* also allows detection of frame loss, as no frame has been received in a given period. It can also be tailored to detect loss of messages with a specific identifier.

An additional method reported in Part 5 of the standard is read-back of the message, for which the sender reads back from the bus the message it has just transmitted. It is used by the popular, also in the automotive field, CAN bus.

Let us take a look at sensors and actuators, two types of element which are also subject to failure mode analysis when adopted into a system. We will not go into much detail, as these are outside the scope of the thesis activities.

2.3.6 Sensors

A sensor is an element which includes, either physically or virtually, a transducer and hardware elements that support it and allow the processing of its output.

A transducer is a part that converts energy in one form to another. Therefore, in the E/E context, it is used to convert a physical quantity (lengths for position, speed, acceleration, heat for temperature measurements and so on) into a voltage value. Some mathematical function maps any read voltage value (for a MCU, its binary reading) to values for the original quantity.

It is clear that faults in any part between the transducer itself and the sensor output can lead to wrong overall output and potentially result in hazards.

A common example could be a faulty temperature sensor which reports low temperatures for an element sensitive to heat, which may catch on fire without intervention.

Although specific failure modes vary depending on the transducer technology, typical ones involve:

- *Offset* outside specifications, which increases or reduces the nominal output of a given quantity, which could vary over time or due to heat.

  Ideally, transducer and circuitry would add zero offset, however true sensors always introduce some. Its extent is specified by the manufacturer for given operating conditions.

- *Dynamic range* outside a given range, leading to reachable voltage values lower or higher than anticipated.

- *Sensitivity* or *Gain*, which indicates how sensible the element is to the input quantity, could be too high or low, null, sensitive to temperature or deviate from the mathematical link.

These failure modes can also be derived from the production process of the sensors, as a wide variety of phenomena can stress them without being able to visibly observe a fault.

Safety mechanisms for sensors involve self-tests, calibration procedures, adjustments for gain, and offset minimization.

Redundancy can also be useful, with one example being the *Reference sensor*[2] which monitors periodically the output of the process (main) sensor. Also sensors used for different physical properties could be used for detection, as long as they can be mapped with each other by means of a model.

Therefore, both hardware and software measured can be adopted for these devices.

2.3.7 Actuators

These are the final elements of our systems, which allow the system to control the outside world. However, unlike the previous categories, no generalized failure modes are provided for this class.

As electromechanical elements, we can only provide generic safety measures for these, which may or may not be adopted.

- *Failure detection by online monitoring* is used to detect failures in the system by analyzing its behavior online during operation.

This is not implemented with specific hardware elements, but relies on comparison of some action or entity at system level with the expected one, usually at specific conditions.

Low diagnostic coverage is expected, as this is not a thorough method and localization of the fault is not precise.

- *Test patterns* can be adopted. The properties seen for I/O units also apply here.
- *Monitoring* could also be applied to the actuator, when possible, to monitor its operation. By measuring a physical quantity, affected by the actuator, with the expected one, fault detection is achieved and high coverage can be expected.
- *Measures against physical environment* can be taken during the design or adoption phase, by enclosing the equipment, to prevent it from being affected by physical wear or infiltrations which may cause failures[2]. A watertight enclosure is an example, as it would avoid short circuits caused by water and corrosion due to rust or salt.

Chapter 3

Safety mechanisms implementation

This chapter shall now examine the covered case, providing an overview of the adopted hardware and the implemented safety mechanisms in order to ensure the desired safety level.

An overview of the hardware, and some manuals, are provided by Infineon on their website; however, as we have already stated, much of the safety related documentation is classified, so we will approach the matter in a suitable way, which includes generic mechanisms, without their specific names and characteristics, and the use of pseudo-code to show hints of the software.

3.1 SEooC hardware

The adopted MCU is a variant of an `Infineon Aurix™ TC36x`, which belongs in the greater TC3xx family of embedded microcontrollers with multiple CPUs.

In this MCU family, many peripherals are provided to cover the needs for control applications with real-time requirements, critical bandwidth and signal processing capabilities, supporting also ADAS applications if required.

The TC36x is characterized by two TC1.6.2 CPUs running at up to 300 MHz, 672KB of RAM with ECC method, up to 4 MB of FLASH with ECC and power consumption not exceeding a couple of Watts.

ASIL D can be reached when adopting this device, at it features lots of safety mechanisms, one of which is the adoption of the lockstep cores, with clock delay, for both CPUs.

Another functional block that allows for easier safety scalability is the *Safety Management Unit*, or SMU, which manages several safety critical elements.

The Integrated Circuit which supplies power to the MCU is an `Infineon TLF35584`, also designed for automotive applications up to ASIL D.

It can also act as a safety monitor, with I/O for safety signals and two watchdogs, a window one with timing constraints and a functional one, which can check the correctness of the timely response.

Safety manuals, several safety analysis reports, a customizable FMEDA and other documents, provided only under NDA, give a guidance of reaching the required safety levels, by providing the required metrics to prove compliance with respect to the ASIL.

In addition to the classification, reported in chapter 2, of safety mechanisms being implemented either in hardware or in software, another classification is possible: a mechanism is *internal* if it is provided with the MCU itself (along with software libraries, if any), *external* if the system integrator is expected to implement them. We can refer to the former as simply SMs and the latter as ESMs.

Lists of these are provided in the safety manual for the microcontroller family.

3.2 Safety Mechanisms

3.2.1 Program Flow Monitoring

The program flow monitoring mechanisms is one of those which have been chosen and implemented for the item. It can be a quite powerful measure with respect to soft errors, as it targets several aspects, as mentioned in subsection 2.3.2.

In this implementation, the technique can be used for periodic tasks that are safety-relevant, which we will also call *monitored entity*. Structures in software shall be declared, with useful fields, before compilation, in order to have reference values for each entity. One of these is the monitored entity identifier, which has to be unique.

Due to their periodic nature, constraints on timing are expected: too frequent or delayed activations can result in incorrect behavior and potential hazards.

As a result, a time window is configured, for a single entity, with a minimum inter-arrival time and the maximum inter-arrival time.

Any monitored task is expected to call, through some API, the software module responsible for PFM. Each one of these calls corresponds to a *checkpoint*, points in which the application gives space to the PFM module to either signal some condition or execute some tests.

The first checkpoint shall be invoked as soon as a specific instance of the entity starts executing, as the initial time for the instance shall be noted.

The PFM module stores, in some way, the activation time for the current instance and the previous one: at a certain checkpoint a test on these is performed

and, in case of interval between activations being outside a window, a fault is reported.

Contributing to the detection properties of PFM, logical monitoring is also part of the module.

In the implemented mechanism, for each checkpoint a CRC is generated before compile time and, during normal operation, the monitored entity produces, at runtime, the CRC value for the specific checkpoint.

During the checkpoint call, the PFM module shall compare the generated value with the pre-compile one, and report a fault if these differ, as either a soft fault in the CPU or clock altered the original instruction flow.

This PFM module expects a minimum of two checkpoints for a task: each checkpoint can be useful for logical monitoring, as errors may appear before the task instance ends and one may be interested in checking before or after specific operations, but a minimum of two checkpoints also allows for *deadline monitoring*.

During the discussion about safety mechanisms timing in chapter 2, we have implicitly declared deadline, as some intervals shall be respected.

A *deadline* refers to the maximum time at which some operation can end. For a periodic task, the deadline limits how much time can elapse between one instance's activation and its ending.

In this case, it is a relative deadline (amount of time after each activation), however absolute deadlines can also be used, by defining the time interval against some fixed point in time.

The second checkpoint call can then be used to check whether the deadline was met or not.

```
1 void EntityID0(void){
2     CheckpointAPI(0,FirstCheckpoint);
3     foo();
4     CheckpointAPI(0,SecondCheckpoint);
5     bar();
6     CheckpointAPI(0,ThirdCheckpoint);
7 }
```

Listing 3.1: Periodic task example

Listing 3.1 reports an example of a periodic task, in pseudo C code, with identifier 0 and three checkpoints.

The first and last lines invoke the checkpoint API at the start and end of a given instance, while the middle function call serves for logical monitoring between some initial operation, called `foo()` and the following operations, `bar()`.

Listing 3.2 reports a plausible implementation of the PFM module with the overall behavior, as discussed so far. An array of structures, named `Task` and with length based on the number of monitored entities, contains data for monitoring,

both pre-compile defined properties, as the number of checkpoints it shall call in each instance and the allowed interval times, and runtime fields.

```
1  void CheckpointAPI(int ID, int Checkpoint){
2      if(Checkpoint == Task[ID].FirstCheckpoint){
3          Task[ID].NewActivationTime = getTime();
4          difference = Task[ID].NewActivationTime - Task[ID].
       PreviousActivationTime;
5          if(difference > Task[ID].MaxInterArrival || difference < Task
       [ID].MinInterArrival){
6              ReportFault(ID, TimingFault);
7          }
8      }
9
10     computeCRC = CRC();
11     if(Task[ID].CRC[Checkpoint] != computeCRC){
12         ReportFault(ID, LogicalFault);
13     }
14     Task[ID].currentCheckpoint++;
15
16     if(Checkpoint == Task[ID].LastCheckpoint){
17         Task[ID].endTime = getTime();
18         Task[ID].hasEnded = true;
19         if(Task[ID].endTime - Task[ID].NewActivationTime > Task[ID].
       Deadline){
20             ReportFault(ID, DeadlineFault);
21         }
22     }
23 }
```

Listing 3.2: Checkpoint function example

When a task calls the function, its structure is accessed through its ID. At the first checkpoint, the arrival time is stored in `Task[ID].NewActivationTime`, the activation time of the previous instance is substracted and the result is then compared to the lower and upper allowed bounds; a fault is reported if it is outside specifications. As the code fragment is a simplification, true software shall adopt good practices, for instance by correctly handling boundary cases and initial conditions.

Then, CRC is computed in some way, depending on the task and checkpoint number, and compared to a value stored in the structure.

At final checkpoint, a deadline miss can be detected by comparing the difference between current and activation time with the deadline.

The function `ReportFunction(int ID, FaultType fault)` is external to the PFM module, and its behavior depends on the task which generated the fault and the specific type, as different actions may be necessary.

These monitoring activities rely on a timer, separated from the clock used for

the CPU and assumed to be safe to use, and the tasks themselves being able to invoke the API for the module.

However, it may be the case that a task may fail to do so, which includes a wrong control path or being stuck in some loop.

Guidelines on software development by the ISO and the MISRA C standard mitigate these issues, however *aliveness* monitoring can be implemented for more robustness.

```
1 void EntityID1(void){
2     CheckpointAPI(1,FirstCheckpoint);
3     return; /* Task dies before last point */
4     CheckpointAPI(1,SecondCheckpoint);
5 }
```

Listing 3.3: Misbehaving monitored tasks

Listing 3.3 shows an example task that would require aliveness monitoring, as it behaves differently than what programmers had in mind. The function ends with `return` before calling the last API, with no deadline monitoring possible.

It is clear that, in order to detect this fault type, an external method, with respect to the tasks, has to be adopted.

The use of a watchdog makes this possible by making it a part of the watchdog service routine, which answers the watchdog inside an allowed time window.

```
1  void WatchdogService(void){
2      /* ... */
3      AlivenessMonitor();
4      /* ... */
5  }
6
7  void AlivenessMonitor(){
8      int i;
9      for(i=0; i<MONITORED_TASKS_NUMBER; i++){
10         if(Task[i].hasEnded == false){
11             if(getTime() - Task[i].NewActivationTime >= Task[i].
       maxTime){
12                 ReportFault(i,Aliveness);
13             }
14         }
15     }
16 }
```

Listing 3.4: Aliveness monitoring

As the watchdog routine is independent of PFM tasks, a check on all monitored entities is performed. Information about the task having called the last checkpoint API is stored and, if it did not, two options are possible: either the task has been

interrupted during the watchdog routine, and it shall continue after it, or the task has died prematurely.

If that happened, this is detected during a service routine (not necessarely the first one, depending on the time window properties of the watchdog) by comparison with the maximum allowed time. This last information may be derived from the periodicity of the task: a common assumption is that a task's instance ends before the next one appears.

For example, if a periodic task, of nominal period 100 ms and allowed range of 90 to 110 ms, ends prematurely, there is no point in waiting more than 110 ms: whether it ended or is still stuck running, the intended behavior is not observed.

3.2.2 Tests

We shall now introduce a new type of safety mechanisms which were added into the system, consisting of several tests (mostly self-tests) triggered by the software, at startup or periodically, that are supported by hardware.

Unlike program flow monitoring, which is strongly connected with the periodic tasks, these tests are generally run either once per driving cycle or periodically.

In the former case, they can conveniently be inserted in the startup phase, which also provides results about correct behavior for the following usage; in the latter, the period can be derived from different factors, but is still related to the relevant FHTI constraints.

```
1  int CoreX_main(){
2      InitializeDevice();
3      for(int i=0;i<STARTUP_TESTS_NUMBER;i++){
4          TestResult_t result = ExecuteStartupTest(i);
5          if(result != OK){
6              ReportError(i,result);
7          }
8      }
9
10     InitiatePeriodicTasks(); /* Application  software */
11     InitiatePeriodicTests();
12     while(1);
13 }
```

Listing 3.5: Test management

3.2.3 Logic BIST

Upon startup of the MCU, one of the first activities consists in a set of self-tests to verify the health of its digital logic. These tests rely on specialized on-chip circuitry that generates carefully generated random patterns to examine the system's core

components. The Logic Built-In Self-Test (LBIST) builds on this foundation, offering a robust way to uncover hidden faults in the microcontroller's digital logic without any external testing tools.

LBIST can start automatically during startup or be activated later by software, depending on the system's needs. It works by feeding pseudo-random test patterns through the chip's logic paths, capturing responses, and condensing them into a unique signature using dedicated registers. This signature reveals whether the logic is functioning correctly or carrying faults. In order to maintain the required stability of the MCU during execution, LBIST skips testing the power management and control units, focusing instead on the broader digital logic.

Settings for this mechanism can be tweaked, like the number of test patterns or the speed of execution, to obtain a balance between thoroughness and efficiency.

Once the test completes, the system typically resets to resume normal operation, though some memory areas might need reinitializing afterward.

LBIST allows for high diagnostic coverage, catching potential faults early.

3.2.4 Secondary monitors BIST

For a microcontroller in safety critical applications, unexpected voltage levels can lead to unexpected results, with an impact on the safety of the system and overall item.

A power supply developed as a SEooC may require additional validation to meet the requirements of higher ASIL when integrated into a system, MCU manufacturers may choose to adopt additional measures on the microcontroller itself, using techniques found in power supplies or common to analog signal processing, as discussed in chapter 2.

Monitors, which may work in a purely analog way, or through analog to digital conversion, can be a solid choice to improve fault detection properties on the package.

In this MCU, redundant monitoring for a set of signals improves diagnostic coverage and mitigates common cause failures, as monitors cover different aspects.

For improved detection capabilities, self-tests on the monitors can be adopted.

Among startup tests, the manufacturer also supports tests for the monitors at a secondary level, improving latent fault coverage metrics.

These tests were successfully added to the project and run at initialization phase.

3.2.5 Register self tests

In the previous chapter, we examined common failure modes and mechanisms affecting safety-critical memories. In order to address potential faults, a series of

tests are carried out during startup, before application software is executed along with any cyclic test.

For this case, Built In Self Tests for testable internal memories are performed through the Memory Test Unit (MTU).

Each testable memory is equipped with hardware circuitry to support these tests and to provide interfaces to control them by the MTU. In addition to BIST, MTU and the support hardware also provide support for memory error detection and correction methods, which can be configured via this unit, while also supporting fault detection for addressing elements.

By adopting these features, the system can reach high levels of diagnostic coverage, as discussed in chapter 2.

With this device, BIST setup and execution are managed by configuring specific registers, which also store test status and results.

Chapter 4

Conclusion

The activities carried out for this thesis cover fundamental steps during the development of an item for automotive use.

Although it was not possible to disclose techniques and documentation related to the specific instance, this thesis demonstrates how the entire development process is shaped by risk management requirements.

Efforts to ensure the item's core functionality represent only a fraction of the work, as each component's failure potential must be analyzed and addressed.

For this purpose, ISO 26262 provides an in-depth reference for the entire process, with steps and requirements for each phase, including also production and service after design and development.

This thesis demonstrates that compliance to the ISO 26262 enables deep understanding of the item's parts and quantitative risk metrics to ensure effective mitigation.

Failure modes for each element, based on their type, are also provided in the standard and its generic version, the IEC 61508; however, as most elements are sourced by semiconductor manufacturers, technical documentation from the manufacturers provided are required.

These documents provide deep understanding of the elements, developed as safety elements independent of the item's context, to integrate them into specific devices.

At this point, project decisions and development takes place, with a set of mechanisms implemented in the firmware to contribute towards the required metrics for the assigned safety integrity level.

Subsequent testing of hardware and software safety mechanisms ensures effective risk management without compromising functionality.

Even though compliance with these standards provides improved safety and a legal standpoint against any lawsuit, it also requires additional costs in money and time, making development of modern and reliable systems complex. Addressing

these aspects is not trivial, as absence of thorough analysis may lead to serious danger in the future, with significant costs for recalls or lawsuits.

Adoption of already proven elements, where economically and functionally compatible, provides a big benefit. Suitable project organization, reliance on past reports and experience also help towards efficient development. Future research could also explore cost-effective methods to comply with standards, with model-based software design being an example of streamlined software development phase. These methods would support technological innovation while keeping low risk or even reducing cost to the consumer.

Bibliography

[1] *ISO 26262: Road vehicles – Functional safety*. Geneva, Switzerland: International Organization for Standardization, 2018 (cit. on pp. 1, 3, 4, 6, 11, 14, 15, 17, 18).

[2] International Electrotechnical Commission. *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. Standard. Parts 1-7, Second Edition. International Electrotechnical Commission, 2010. URL: https://www.iec.ch/functionalsafety (cit. on pp. 1, 20, 24, 27, 28).

[3] MISRA Consortium. *MISRA C:2012 Guidelines for the use of the C language in critical systems*. Tech. rep. Motor Industry Software Reliability Association (MISRA), 2013 (cit. on p. 7).

[4] EGAS Workgroup. *Standardized E-Gas Monitoring Concept for Gasoline and Diesel Engine Control Units*. Version 6.0 (cit. on pp. 7, 8).

[5] Infineon. *AN0001 AURIX™ TC3xx functional safety (FuSa) in a nutshell*. Application Note. 2025. URL: https://documentation.infineon.com/aurixtc3xx/docs/owq1745576218449 (cit. on pp. 10, 20, 22, 23).

[6] Alessandra Nardi and Antonino Armato. «Functional safety methodologies for automotive applications». In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017, pp. 970–975. DOI: 10.1109/ICCAD.2017.8203886 (cit. on p. 20).