Chapter 8
Functional Safety of Automotive Software

Per Johannessen

Abstract In the previous chapters we explored generic methods for assessing quality of software architecture and software design. In this chapter we continue with a much-related topic, functional safety of software, in which functional safety assessment is one of the last activities during product development. We describe how the automotive industry works with functional safety. Much of this work is based on the ISO 26262 standard that was published in 2011. This version of the standard is applicable for passenger cars up to 3500 kg. There is also ongoing work on a future version, expected in 2018, applicable to most road vehicles, including buses, motorcycles, and trucks. The scope of the ISO 26262 standard is more than software development and for better understanding we give an overview of these other development phases in this chapter. However, we focus on software development according to ISO 26262. The different phases that are covered are software planning, software safety requirements, software architectural design, software unit design and implementation, software integration and testing, and verification of software.

8.1 Introduction

Functional safety in ISO 26262 is defined as *"absence of unreasonable risk due to hazards caused by malfunctioning behaviour of E/E systems"*. In a simplified way, we could say that there shall not be any harm to persons resulting from faults in electronics or software. At the same time, for an automotive product, this electronic and software is within a vehicle. Hence, when working with functional safety, it is important to consider the vehicle, the traffic situations including other vehicles and road users and the persons involved.

The safety lifecycle of ISO 26262 starts with planning of product development, continues with product development, production, operation and ends with scrapping the vehicle. In ISO 26262, the base for product development is Items. An Item in ISO 26262 is defined as "system or array of systems to implement a function at the vehicle level, to which ISO 26262 is applied". The key term here is "function at the vehicle level", which defines what components are involved. This also implies that a vehicle consists of many Items, to which ISO 26262 is applied.

The work on the ISO 26262 standard started in Germany in the early 2000 and was based on another standard, IEC 61508 ()Functional Safety of Electri-

M. Staron, *Automotive Software Architectures*,
DOI 10.1007/978-3-319-58610-6_8

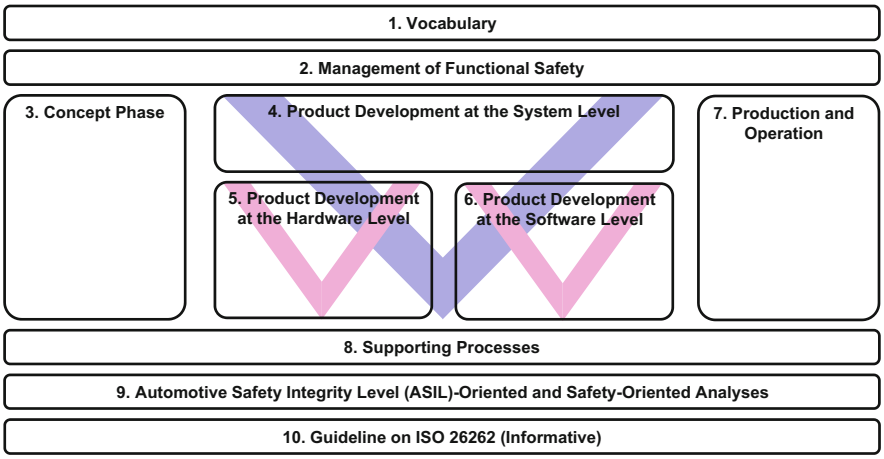

Fig. 8.1 The ten different parts in the ISO 26262 standard, adopted from [ISO11]

cal/Electronic/Programmable Electronic Safety-related Systems). As IEC 61508 [IEC10] originates from the process control industry, there was a need to adapt it to the automotive industry. The work within the ISO standardization organization started in 2005 and resulted in the first edition of ISO 26262 published in 2011.

Even if the automotive industry had been working with functional safety before, this was a significant step to standardize the work across the industry. As with standards in general, the key advantage is simplified cooperation between different organizations. Another benefit with ISO 26262 is that it can be seen as a cookbook on how to develop safe functions on the vehicle level that to some degree are implemented in electronics and software. By following this cookbook, the result is a harmonized safety level across the industry and this level is seen as acceptable.

When looking into ISO 26262, there are ten different parts as shown in Fig. 8.1. In this chapter we focus on part 6, for software development. At the same time, it is important to understand the context in which this software is developed and also the context in which this software is used. Hence, there is a very brief overview of these other parts in ISO 26262 as well.

As we can see in Fig. 8.1, parts 4–6 are based on the V-model of product development which we discussed in Chap. 3 and which is a defacto standard in the automotive industry. It should be noted that even if the V-model is the basis here, the standard is in reality applied in many different ways, including, e.g. distributed development across multiple organizations, iterative development and agile approaches. Independent of the development approach used, the rationale of the safety standardization is argumentation that the requirements in the standards have been appropriately addressed in the product.

In the forthcoming sections we briefly describe parts 2–8 of the standard. Part 1 contains definitions and abbreviations used in the standard. The safety analysis methods described in part 9 are only covered implicitly in this chapter as they are

referenced from the activities in parts 3–6. Also, part 10 is not described here as it is an informative collection of guidelines for how parts 2–6 may be applied.

8.2 Management and Support for Functional Safety

When an organization works with functional safety, there are other processes that should be established. When looking into part 2 of the ISO 26262 standard, there are requirements to have a quality management system in place, e.g. ISO 9001 [ISO15] or ISO/TS 16949 [ISO09]; to have relevant processes, including processes for functional safety, established in the management system, the sufficient competence and experience; and the field monitoring established. Field monitoring from a functional safety perspective is in particular important for detecting potential faults in electronics and software when the vehicle is in use.

During product development, there are also requirements on assigning proper responsibilities for functional safety, to plan activities related to functional safety and to monitor that the planned activities are done accordingly.

In addition, there are requirements to have proper support according to part 8, including:

- Interfaces within distributed developments, which ensure that responsibilities are clear between different organizations that share the development work, e.g. between a vehicle manufacturer and its suppliers. It is often referred to as a Statement of Work.
- Requirements management, which ensures that requirements, in particular safety requirements, are properly managed. This includes identification of requirements, requirement traceability, and status of the requirements.
- Configuration management, which ensures that changes to an Item are controlled throughout its lifecycle. There are other standards for configuration management, e.g. ISO 10007, referenced from ISO 26262.
- Change management, which in ISO 26262 ensures that functional safety is maintained when there are changes to an Item. It is based on an analysis of proposed changes and control of those changes. Change management and configuration management typically go hand in hand with each other.
- Documentation management, which in ISO 26262 ensures that all documents are managed such that they are retrievable and contain certain formalities such as unique identification, author and approver.
- Confidence in the use of software tools, which shall be done when compliance with the ISO 26262 standard relies on correct behavior of software tools used during product development, e.g. code generators and compilers. The first step is tool classification to determine if the tool under consideration is critical and, if critical, tool qualification is done to ensure that the tool can be trusted.

These requirements mean that ISO 26262 poses requirements on the product development databases described in Chap. 3 in terms of the kind of connections and relations that should be maintained.

8.3 Concept and System Development

According to ISO 26262, product development starts with the development of a concept as described in part 3. In this phase, the vehicle level function of an Item is developed. Also, the context of the Item is described, i.e. the vehicle and other technologies such as mechanical and hydraulical components. After the concept phase, there is the system development phase according to part 4 in ISO 26262. In ISO 26262, the system only contains electronic hardware and software components, no mechanical components. The development of these other components is not covered by ISO 26262.

The first step in concept development is to define the Item to which ISO 26262 is applied. This definition of the item contains functional and non-functional requirements, use of the Item including its context, and all relevant interfaces and interactions of the Item. It is an important step as this definition is the basis for continued work.

The following step is the hazard analysis and risk assessment, which includes hazard identification and hazard classification. A hazard in ISO 26262 is a potential source of harm, i.e. a malfunction of the Item that could harm persons. Examples of hazards are no airbag deployment when intended and unintended steering column lock. These hazards are then further analyzed in relevant situations, e.g. driving in a curve with oncoming traffic is a relevant situation for the unintended locking of the steering column. The combination of a hazard and relevant driving situations that could lead to harm is called hazardous event.

During hazard classification, hazardous events are classified with an ASIL. ASIL is an ISO 26262-specific term defined as Automotive Safety Integrity Level. There are four ASILs ranging from ASIL A to ASIL D. ASIL D is assigned to hazardous events that have the highest risk that needs to be managed by ISO 26262, and ASIL A to the lowest risk. If there is no ASIL, it is assigned QM, i.e. Quality Management. The ASIL is derived from three parameters, Controllability, Exposure, and Severity. These parameters estimate the magnitude of the probability of being in a situation where a hazard could result in harm to persons (Exposure), the probability of someone being able to avoid that harm given that situation and that hazard (Controllability), and an estimate of the severity of that harm (Severity). In Table 8.1, a brief explanation of the different ASILs and examples are shown.

In addition to ASIL being a measure of risk, it also puts requirements on safety measures that need to be taken to reduce the risk to an acceptable level. The higher the ASIL, the more the safety measures are needed. Examples of safety measures are analyses, reviews, verifications and validations, safety mechanisms implemented in electronic hardware and software to detect and handle fault, and independent

Table 8.1 Brief description of different ASILs with examples; the examples are dependent on vehicle type

Risk classification	Description of risk	Examples of hazardous event
QM	The combination of probability of accident (*Controllability* and *Exposure*) and severity of harm to persons (*Severity*) given the hazard is considered acceptable. With a QM classification, there are no ISO 26262 requirements on the development	No locking of steering column when leaving the vehicle in a parked position. Not possible to open sunroof
ASIL A	A low combination of probability of accident and severity of harm to persons given the hazard occurring	No airbag deployment in a crash fulfilling airbag deployment criteria
ASIL B	...	Unintended hard acceleration of vehicle during driving
ASIL C	...	Unintended hard braking of vehicle during driving while maintaining vehicle stability
ASIL D	Highest probability of accident and severity of harm to persons given the hazard occurring	Unintended locking of steering column during driving

Table 8.2 A simplified hazard analysis and risk assessment with two separate examples

Function	Hazard	Situation	Hazardous event	ASIL	Safety goal
Steering column lock	Unintended steering column lock	Driving in curve with oncoming traffic	Driver loses control of his vehicle, entering the lane with oncoming traffic	D	Steering column lock shall not be locked during driving
Driver airbags	No deployment of driver airbags	Crash where airbag should deploy	Driver is not protected by airbags in a crash when he should be	A	Driver airbag shall deploy in crash, meeting deployment criteria

safety assessments. If there is a QM, it means that there are no requirements on safety measures specified in ISO 26262. Still, normal automotive development is needed and this includes proper quality management, review, analysis, verification and validation, and much more.

For hazardous events where there is an ASIL assigned, a Safety Goal shall be specified. A Safety Goal is a top-level safety requirement to specify how the hazardous event can be avoided. A simplified hazard analysis and risk assessment is shown in Table 8.2.

The third step is the functional safety concept where each Safety Goal with an ASIL is decomposed into a set of Functional Safety Requirements and allocated to a logical design. It is also important to provide an argumentation of why the

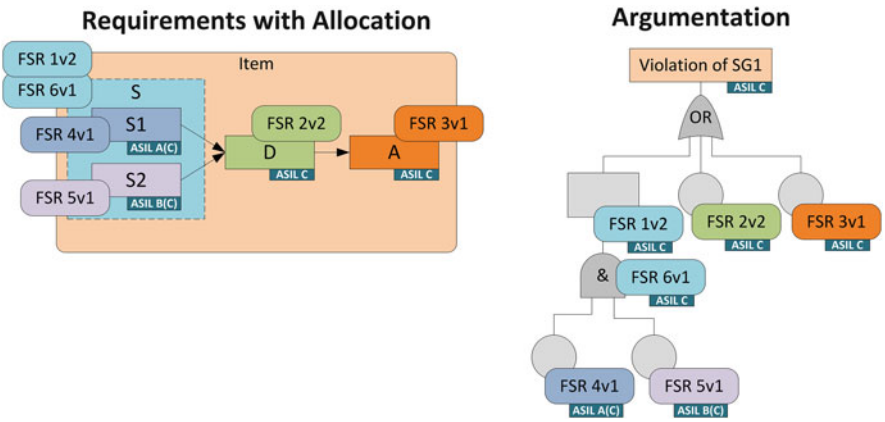

Fig. 8.2 The three parts of a functional safety concept, the Functional Safety Requirements, noted as FSR, their allocation to a logical design, and the argumentation why the Functional Safety Requirements fulfill the Safety Goal, noted as SG

Functional Safety Requirements fulfill the Safety Goal; this argumentation can be supported by a fault tree analysis.

During the Functional Safety Concept stage, and also during later refinements of safety requirements, it is possible to lower the ASILs if there is redundancy with respect to the requirements. There is always a trade-off whether to use redundancy or not. Redundant components could increase cost, at the same time as lower ASILs could save cost. The choice to make needs to be assessed on a case-by-case basis.

An example of a Functional Safety Concept is shown in Fig. 8.2. Here the logical design consists of three parts, the sensor element S, the decision element D and the actuation element A. The sensor element has been refined using redundancy of two sensor elements, S1 and S2. For all of these elements, there are Functional Safety Requirements allocated, denoted bt FSR, with a sequence number and an ASIL. For the argumentation of why these Functional Safety Requirements fulfill the Safety Goal SG1, a fault tree with the violation of the Safety Goal, SG1, as a top event is used.

During system development, as shown in Fig. 8.1 as part 4, the Functional Safety Concept is refined into a Technical Safety Concept. It is very similar to a Functional Safety Concept, but more specific in details. At this point, we work with actual systems and components, including signaling in between. It is common that a Technical Safety Concept includes interfaces, partitioning, and monitoring. The Technical Safety Concept includes Technical Safety Requirements that are allocated and an argumentation of why the Technical Safety Concept fulfills the Functional Safety Concept (Fig. 8.3). An example of one possible level of design for a Technical Safety Concept is shown in Fig. 8.4. Here the design for the decision element has been refined to an ECU that consists of a microcontroller and an ASIC. For these

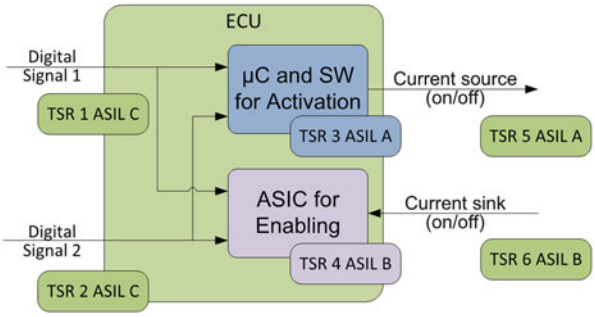

Fig. 8.3 A Technical Safety Concept is one level more detailed than a Functional Safety Concept, here with a microcontroller including software (SW) and an ASIC for ensuring correct activation

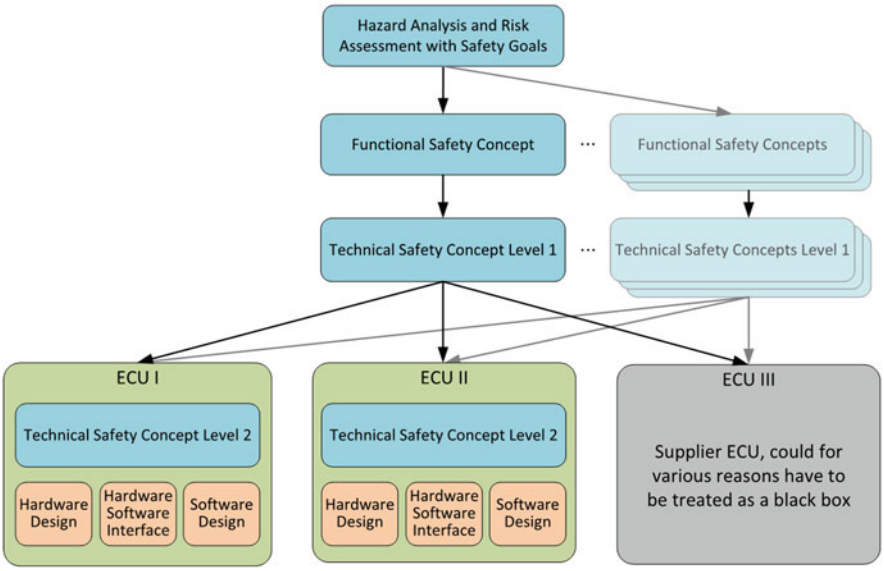

Fig. 8.4 An example of a hierarchy of Technical Safety Concepts

two elements, there are Technical Safety Requirements allocated, denoted by TSR, with a sequence number and an ASIL.

During actual development, it is common that there is a hierarchy of Technical Safety Concepts. In addition, for each Safety Goal with an ASIL there are other Functional Safety Concepts and Technical Safety Concepts. An example of the relationships between safety concepts is shown in Fig. 8.4. In this case, the top ones allocate Technical Safety Requirements to elements that consist of both software and hardware, e.g. an ECU. In the lowest one, the Technical Safety Requirements are allocated to software and hardware. In this lowest level of Technical Safety Concept, there is also a hardware-software interface. The following step in the

design is detailed hardware and software development. In this chapter, we will only consider the software part. The hardware development has a similar structure as the software development.

8.4 Planning of Software Development

The software development starts with a planning phase. In addition to the planning of all software activities, including assigning resources and setting schedules, the methods and tools used need to be selected. At this phase, the modeling or programming languages to be used are also determined. The software activities to be planned are shown in Fig. 8.5 and also described in more detail in this chapter.

Even if ISO 26262 is described in a traditional context with manually written code according to a waterfall model, ISO 26262 supports both automatic code generation and agile way of working.

To support the development and to avoid common mistakes, there is a requirement to have modeling and coding guidelines. These shall address the following aspects:

- Enforcement of low complexity: ISO 26262 does not define what low complexity is and it is up to the user to set an appropriate level of what is sufficiently low. An appropriate compromise with other methods in this part of ISO 26262 may be required. One method that can be used is to measure cyclomatic complexity and have guidance for what to achieve.
- Use of language subsets: When coding, depending on the programming language, there are language constructs that may be ambiguously understood or may easily lead to mistakes. Such language constructs should be avoided, e.g. by using MISRA-C [A+08] when coding in C.

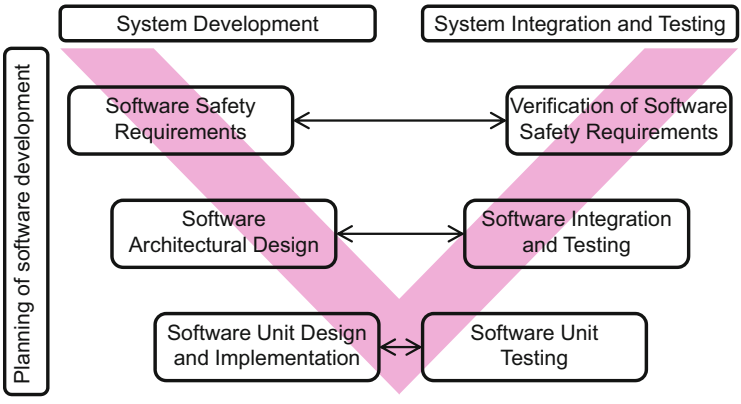

Fig. 8.5 The software development activities according to ISO 26262. Adopted from [ISO1]

- Enforcement of strong typing: Either strong typing is inherent in the programming language used, or there shall be principles added to support this in the coding guidelines. The advantage of strong typing is that the behavior of a piece of software is more understandable during design and review as the behavior has to be explicit. When strong typing is inherent in the programming language, a value has a type and what can be done with that value depends on the type of the value, e.g. it is not possible to add a number to a text string.
- Use of defensive implementation techniques: The purpose of defensive implementation is to make the code robust to continue to operate even in the presence of faults or unforeseen circumstances, e.g. by catching or preventing exceptions.
- Use of established design principles: The purpose is to reuse principles that are known to work well.
- Use of unambiguous graphical representation: When using graphical representation, it should not be open to interpretation.
- Use of style guides: A good style when coding typically makes the code maintainable, organized, readable, and understandable. Hence, the likelihood for faults is lowered when using good style guides. One example of a style guide for C [A+08] is [DV94]
- Use of naming conventions: By using the same naming conventions the code becomes easier to read, e.g. by using *Title Case* for names of functions.

8.5 Software Safety Requirements

Once we have Technical Safety Requirements allocated to software and the software development planned, it is time to specify the software safety requirements. These are derived from the Technical Safety Concept and the system design specification, while also considering the hardware-software interface. At the end of this step, we shall also verify that the software safety requirements, including the hardware-software interface, realize the Technical Safety Concept.

In a safety-critical context, there are several services expected from software that are specified by software safety requirements, including:

- Correct and safe execution of the intended functionality.
- Monitoring that the system maintain a safe state.
- Transitioning the system to a degraded state with reduced or no functionality, and keeping the system in that state.
- Fault detection and handling hardware faults, including setting diagnostic fault codes.
- Self-testing to find faults before they are activated.
- Functionality related to production, service and decommissioning, e.g. calibration and deploying airbags during decommissioning.

8.6 Software Architectural Design

The software safety requirements need to be implemented in a software architecture together with other software requirements that are not safety-related. In the software architecture, the software units shall be identified. As the software units get different software safety requirements allocated to them, it is also important to consider if these requirements, potentially with different ASILs, can coexist in the same software unit. There are certain criteria to be met for coexistence. If these criteria aren't met, the software needs to be developed and tested according to the highest ASIL of all allocated safety requirements. These criteria may include memory protection and guaranteed execution time.

The software architecture includes both static and dynamic aspects. Static aspects are related to interfaces between the software units and dynamic aspects are related to timing, e.g. execution time and order. An example can be seen in Fig. 8.6. To specify these two aspects, the notation of the software architecture to be used is informal, semi-formal or formal. The higher the ASIL, the more the formality needed.

It is also important that the software architecture consider maintainability and testability. In an automotive context, software needs to be maintainable as its lifetime is considerable. It is also necessary that the software in the software architecture easily be tested, as testing is important in ISO 26262. During the design of the software architecture, it is also possible to consider the use of configurable software. There are both advantages and disadvantages when using it.

To avoid systematic faults in software resulting from high complexity, ISO 26262 specifies a set of principles that shall be used for different parts, including:

- Components shall have a hierarchical structure, shall have high cohesion within them and be restricted in size.
- Interfaces between software units shall be kept simple and small. This can be supported by limiting the coupling between software units by separation of concerns.

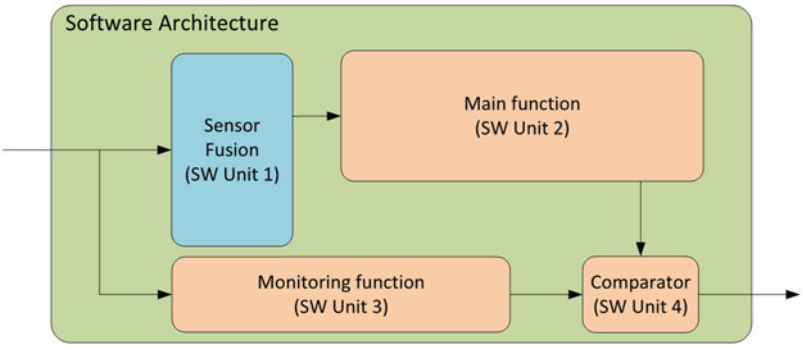

Fig. 8.6 A simple software architecture with four software units

- Scheduling of software units shall be appropriate based on the software, and if interrupts are used, these shall be avoided and be priority-based. The purpose is to ensure timely execution of software units.

At the software architectural level there is a good possibility of detecting errors between different software units. As in general for different ASILs, the higher the ASIL, the more the mechanisms needed. These are mechanisms mentioned in ISO 26262, some overlapping with each other:

- Range checks of data: This is a simple method to ensure that the data read from or written to an interface is within a specified range of values. Any value outside this range is to be treated as faulty, e.g. a temperature below absolute zero.
- Plausibility checks: This is a type of sanity check that can be used on signals between software units. It should, e.g., catch a vehicle speed signal going from standstill to 100 km/h in 1 s for a normal car. Such acceleration is not plausible. A plausibility check could use a reference model or compare information from other sources to detect faulty signals.
- Detection of data errors: There are many different ways of detecting data errors, e.g. error detecting codes such as checksums and redundant data storage.
- External monitoring facility: To detect faults in execution, an external monitoring facility can be quite effective. It can, e.g., be software executed in a different microcontroller or a watchdog.
- Control flow monitoring: By monitoring the execution flow of a software unit, certain faults can be detected, including skipped instructions and software stuck in infinite loops.
- Diverse software design: Using diversity in software design can be efficient. The approach is to design two different software units monitoring each other; if the behaviors differ, there is a fault that should be handled. This method can be questioned, as it is not uncommon that software designers make similar mistakes. To avoid similar mistakes, the more diverse the software functionality is, the lower the likelihood of these types of mistakes.

Once an error has been detected, it should be handled. The mechanisms for error handling at the software architectural level specified in ISO 26262 are:

- Error recovery mechanism: The purpose is to go from a corrupted state back into a state from which normal operation can be continued.
- Graceful degradation: This method takes the system from a normal operation to a safe operation when faults are detected. A common example in automotive software is to warn the driver that something is not working by a warning lamp, e.g. the airbag warning lamp when the airbags are unavailable.
- Independent parallel redundancy: This type of mechanism may be quite costly as it may need redundant hardware to be efficient. The concept is based on the assumption that the likelihood of simultaneous failures is low and one redundant channel should always be operating safely.
- Correcting codes for data: For data errors, there are mechanisms that can correct these. These mechanisms are all based on adding redundant data to give different

levels of protection. The more the redundant data that is used, the more the errors that can be corrected. This is typically used on CDs, DVDs, and RAM, but can be used in this area as well.

Once the software architectural design is done, it needs to be verified against the software requirements. ISO 26262 specifies a set of methods that are to be used:

- Walk-through of the design: This method is a form of peer review where the software architecture designer describes the architecture to a team of reviewers with the purpose to detect any potential problems.
- Inspection of the design: In contrast to a walk-through, an inspection is more formal. It consists of several steps, including planning, off-line inspection, inspection meeting, rework and follow-up of the changes.
- Simulation: If the software architecture can be simulated, it is an effective method, in particular for finding faults in the dynamic parts of the architecture.
- Prototype testing: As with simulation, prototyping can be quite efficient for the dynamic parts. It is however important to analyze any differences between the prototype and the intended target.
- Formal verification: This is a method, rarely used in the automotive industry, to prove or disprove correctness using mathematics. It can be used to ensure expected behavior, exclude unintended behavior, and prove safety requirements.
- Control flow analysis: This type of analysis can be done during a static code analysis. The purpose is to find any safety-critical paths in the execution of the software at an architectural level.
- Data flow analysis: This type of analysis can also be done during a static code analysis. The purpose is to find safety-critical values of variables in the software at an architectural level.

8.7 Software Unit Design and Implementation

Once the software safety requirements are specified and the software architecture down to software unit level is ready, it is time to design and implement the software units. ISO 26262 supports both manually written code and automatically generated code. If the code is generated, the requirements on software units could be omitted, given that the tool used can be trusted, as determined by tool classification, and if needed tool qualification. In this section, the focus will be on manually written code.

As with the specification of the software architecture, ISO 26262 specifies the notation that should be used for the software unit design. ISO 26262 requires an appropriate combination of notations to be used. Natural language is always highly recommended. In addition the standard recommends informal notation, semi-formal notation and formal notation. Formal notation is not really required at this time.

There are many design principles mentioned in ISO 26262 for software unit implementation. Some may not be applicable, depending on the development. Many

could also be covered by the coding guidelines used. However, all are mentioned here for completeness:

- One entry and one exit point: One main reason for this rule is to have understandable code. Multiple exit points complicate the control flow through the code and therefore the code is harder to understand and to maintain.
- No dynamic objects or variables: There are two main challenges with dynamic objects and variables, unpredictable behavior and memory leaks. Both may have a negative effect on safety.
- Initialization of variables: Without initializing variables, anything can be put in them, including unsafe and illegal values. Both of these may have a negative effect on safety.
- No multiple use of variable names: Having different variables using the same name risks, confusing to readers of the code.
- Avoid global variables: Global variables are bad from two aspects; they can be read by anyone and be written to by anyone. Working with safety-related code, it is highly recommended to have control of variables from both aspects. However, there may be cases where global variables are preferred, and ISO 26262 allows for these cases if the use can be justified in relation to the associated risks.
- Limited use of pointers: Two significant risks of using pointers are corruption of variable values and crashes of programs; both should be avoided.
- No implicit type conversions: Even if supported by compilers for some programming languages, this should be avoided as it could result in unintended behavior, including loss of data.
- No hidden data flow or control flow: Hidden flows make the code harder both to understand and to maintain.
- No unconditional jumps: Unconditional jumps make the code harder to analyze and understand with limited added benefit.
- No recursions: Recursion is a powerful method. However, it complicates the code, making it harder to understand and to verify.

At the time of software unit design and implementation, it is required to verify that both the hardware-software interface and the software safety requirements are met. In addition, it shall be ensured that the implementation fulfills the coding guidelines and that the software unit design is compatible with the intended hardware. To achieve this, there are several methods to be used:

- Walk-through.[1]
- Inspection (see footnote 1).
- Semi-formal verification: This family of methods is between informal verification, like reviews, and formal verification, with respect to ease of use and strength in verification results.
- Formal verification (see footnote 1).

[1] See Sect. 8.6.

- Control flow analysis (see footnote 1).
- Data flow analysis (see footnote 1).
- Static code analysis: The basis for this analysis is to debug source code without executing it. There are many tools with increasing capabilities. These often include analysis of syntax and semantics, checking coding guidelines like MISRA-C [A+08], variable estimation, and analysis of control and data flows.
- Semantic code analysis: This is a type of static code analysis considering the semantic aspects of the source code. Examples of what can be detected include variables and functions not properly defined and used in incorrect ways.

8.8 Software Unit Testing

The purpose of testing the implemented and verified software units is to demonstrate that the software units meet their software safety requirements and do not contain any undesired behavior, as shown in Fig. 8.7. There are three steps needed to achieve this purpose; an appropriate combination of test methods shall be used, the test cases shall be determined, and an argumentation of why the test done gives sufficient coverage shall be provided. It is also important that the test environment used for the software unit testing represent the target environment as closely as possible, e.g. model-in-the-loop tests and hardware-in-the-loop tests as described in Chap. 3.

ISO 26262 provides a set of test methods, and an appropriate combination shall be used, depending on the ASIL of the applicable software safety requirements. The test methods for software unit testing are:

- Requirements-based test: This testing method targets ensuring that the software under test meets the applicable requirements.
- Interface test: This testing method target to ensure that all interactions with the software under test work as intended. It should also detect any incorrect

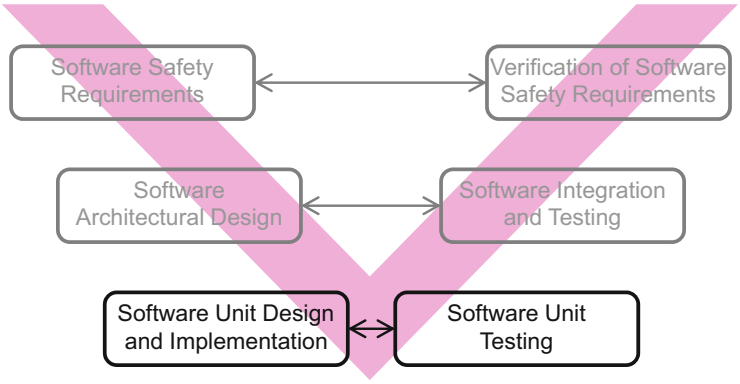

Fig. 8.7 Software unit testing is done at the level of software unit design and implementation

assumptions made on the interfaces under test. These interactions should have been specified by requirements and hence this testing method is overlapping with requirements-based tests.
- Fault injection test: This method is a very efficient test method for safety-related testing. The key part is to test to see if there is something missing in the test target. By injecting different types of faults, together with monitoring and analyzing the behavior, it is possible to find weaknesses that need to be fixed, e.g. by adding new safety mechanisms.
- Resource usage test: The purpose of this test method is to verify that the resources, e.g. communication bandwidth, computational power and memory, are sufficient for safe operation. For this type of testing, the test target is very important.
- Back-to-back comparison test: This method compares the behavior of a model with the behavior of the implemented software when both are stimulated in the same way. Any differences in behavior could be potential faults that need to be addressed.

Similarly, ISO 26262 provides a set of methods for deriving test cases for software unit testing. These methods are:

- Analysis of requirements: This method is the most common approach for deriving test cases. Basically, the requirements are analyzed and a set of appropriate test cases are specified.
- Generation and analysis of equivalence classes: The purpose of this method is to reduce the number of test cases needed to give good test coverage. This is done by identifying equivalence classes of input and output data that test the same condition. Test cases are then specified with the target to give appropriate coverage.
- Analysis of boundary values: This method complements equivalence classes. The test cases are selected to stimulate boundary values of the input data. It is recommended to consider the boundary values themselves, values approaching and crossing the boundaries and out of range values.
- Error guessing: The advantage of this method is that the test cases are generated based on experience and previous lessons learned.

The last step in software unit testing is to analyze if the test cases performed provide sufficient test coverage. If this isn't the case, more tests need to be carried out. The analysis of coverage is according to ISO 26262, done using these three metrics:

- Statement coverage: The goal is to have all statements, e.g. *printf ("Hello World! n")*, in the software executed.
- Branch coverage: The goal is to have all branches from each decision statement in the software executed, e.g. both true and false branches from an if statement.
- Modified Condition/Decision Coverage (MC/DC): The goal of this test coverage is to have four different criteria met. These are: each entry and exit point is executed, each decision executes every possible outcome, each condition in a

decision executes every possible outcome, and each condition in a decision is shown to independently affect the outcome of the decision.

8.9 Software Integration and Testing

Once all software units have been implemented, verified and tested, it is time to integrate the software units and to test the integrated software. At this testing, the target is to test the integrated software against the software architectural design, as shown in Fig. 8.8. This testing is very similar to software unit testing and consists of three steps: selection of test methods, specification of test cases, and analysis of test coverage. Also, the test environment shall be as representative as possible.

The test methods for software integration testing are the same as for software unit testing as described in Sect. 8.8, namely:

- Requirements-based test
- Interface test
- Fault injection test
- Resource usage test
- Back-to-back comparison test

The methods for deriving test cases for software integration testing are the same as for software unit testing as described in Sect. 8.8, namely:

- Analysis of requirements
- Generation and analysis of equivalence classes
- Analysis of boundary values
- Error guessing

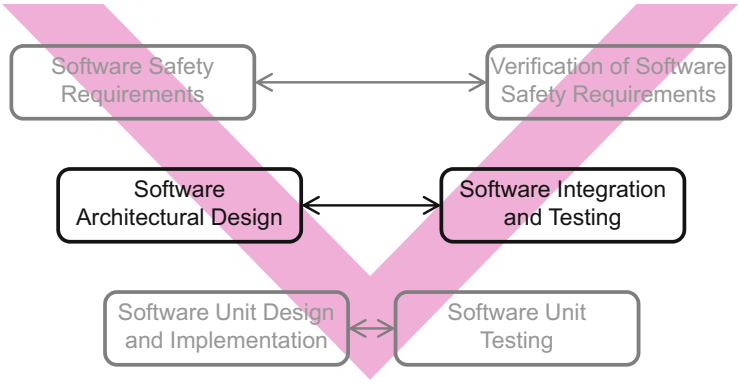

Fig. 8.8 Software unit integration and testing is done at the level of software architecture

The last step in the testing of the integrated software is to analyze test coverage. Again, if the coverage is too low, more tests need to be done. The analysis of coverage according to ISO 26262 is done using the following methods:

- Function coverage: The goal of this method is to execute all functions in the software.
- Call coverage: The goal of this method is to execute all function calls in the software. The key difference of this coverage compared with function coverage is that a function may be called from many different places and ideally all of these calls are executed during testing.

8.10 Verification of Software Safety Requirements

Once the software has been fully integrated, it is time for verification of the software against the software safety requirements, as shown in Fig. 8.9. ISO 26262 specifies possible test environments that can be used. At this point in time, the environment to use is very dependent on the type of development. These test environments may include a combination of:

- Hardware-in-the-loop: Using actual target hardware in combination with a virtual vehicle is a cost-efficient way of testing. As it uses a virtual vehicle, it should be complemented by another environment.
- Electronic control unit network environments: Using actual hardware and software for the external environment is quite common. It is more correct compared to a virtual vehicle; at the same time it may be less efficient in running the tests.
- Vehicles: Using vehicles during this level of testing is in particular useful when there is software that is in operation and has been modified. At the same time, it makes for the most costly test environment.

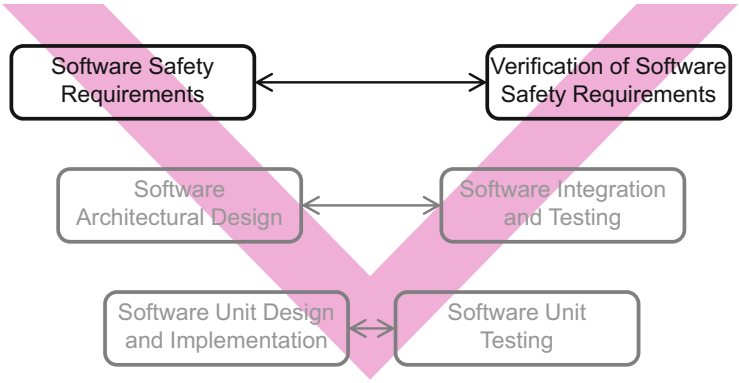

Fig. 8.9 Software unit integration and testing is done at the level of software architecture

8.11 Examples of Software Design

In this section we take some brief examples from the previous sections to show how ISO 26262 could impact software design. In the example in Fig. 8.10, we have an assumed Safety Goal covering faulty behavior classified as ASIL D, and no other Safety Goal. This example has also broken down the ASIL D into two independent ASIL B channels using ASIL decomposition. However, the comparator in the end needs to meet ASIL D requirements as it is a single point of failure.

From the early phases of planning, there will be a requirement on the programming language used, as shown in Fig. 8.10; when using the C language, the MISRA C [A+08] standard is common. An example of a software safety requirement for the comparator in the figure is to transition to a safe state in case of detected errors. In this example a safe state could be no functionality, a so-called fail-silent state. As intentionally shown in Fig. 8.10, working with the software architectural design is quite important. In this example we see the plausibility and range checks on the sensor side as well as external monitoring using diverse software. To make full benefit of this monitoring function, it needs to be allocated to independent hardware. For the testing of the main function, using methods meeting ASIL B requirements on testing is sufficient.

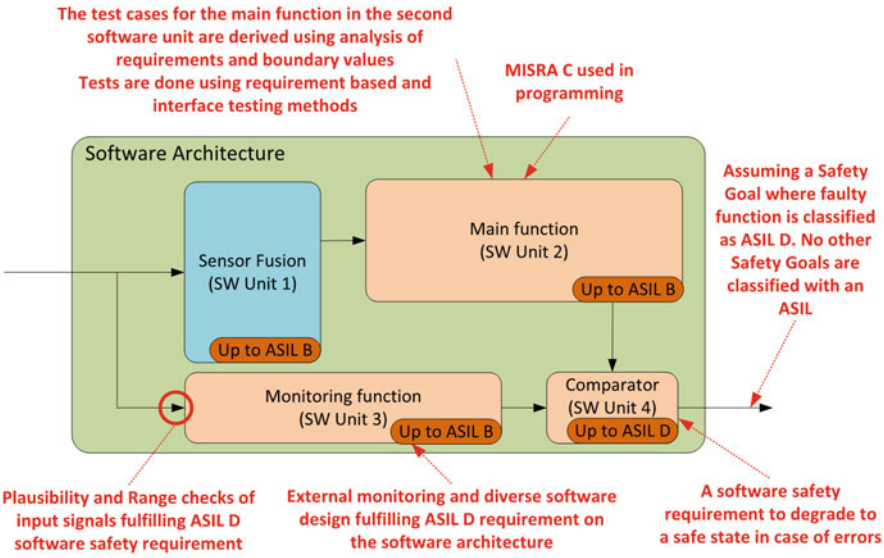

Fig. 8.10 A simple example of a software design for an assumed ASIL D Safety Goal

8.12 Integration, Testing, Validation, Assessment and Release

Once we have fulfilled the Technical Safety Requirements in the design and implementation of software and hardware and also shown by testing that the derived requirements are fulfilled, it is time to integrate hardware and software. In ISO 26262, this is done at three different levels; hardware-software, system and vehicle. At each level, both integration and testing are required. In real development, there can be fewer integration levels and more integration levels, especially when the development has been distributed among vehicle manufacturers and suppliers at many different levels. At each level, there are specific methods to derive test cases and methods to be used during testing. All of these have the purpose of providing evidence that the integrated elements work as specified.

Once we have our Item integrated in a vehicle we can finalize the safety validation. The purpose of safety validation is to provide evidence that the safety goals and the functional safety concept are appropriate and achieved for the Item. By doing so, we have finalized the development and the only remaining activities are to assess and to conclude that the development has resulted in a safe product.

To document the conclusion and the argument that safety has been achieved, a safety case is written. A safety case consists of this argumentation, with pointers to different documents as evidence. Typical evidence includes hazard analysis and risk assessment, safety concepts, safety requirements, review reports, analysis reports, and test reports. It is recommended that the safety case be written in parallel with product development, even if it can't be finalized before the development activities have been finalized.

Once the safety case has been written, it is time for functional safety assessment of Items with higher ASILs. There are many details on how this is to be done, but let us simplify it here. Basically, someone independent is to review the developed system, the documentation that led to the system, in particular the safety case, and the ways of working during the development. If the person doing the assessment is satisfied, it is possible to do the release for production and start producing.

8.13 Production and Operation

Functional safety as a discipline mainly focuses on product development. At the same time, what is developed needs to be produced and is intended to be used in operation by users of the vehicle. Part 7 of ISO 26262 is the smallest part of the whole standard and describes what is required during both production and operation. In addition, planning for both production and operation are activities to be done in parallel to product development.

The requirements for production can be summarized so as to produce what was intended, including maintenance of a stable production process, documentation of what was done during production if traceability is necessary, and carrying out of needed activities such as end-of-line testing and calibration.

For operation, there are clear requirements on information that the driver and service personnel should be aware of in, e.g., instructions in a driver's manual, service instructions and disassembly instructions. One key part during operation is also a field monitoring process. The purpose of this process is to detect potential faults, analyze those faults, and, if needed, initiate proper activities for vehicles in operation.

8.14 Further Reading

In this chapter, we have provided an overview of ISO 26262 and gone into details of the software-specific parts. For more details on both of these parts, the ISO 26262 standard itself [ISO11] is a good alternative when starting to work, especially for the software-specific parts. At the same time, understanding this standard, like many standards, would benefit from basic training to get the bigger picture and the logic behind it. For more details on safety-related software, the work in [HHK10] gives a good start.

To go into details of functional safety in general, there are some books available. One of the classical books that gives a good overview, even if it is a bit old, is [Sto96]. A newer book has been written by Smith [SS10]. This book gives a good overview of functional safety standards in general and details of the IEC 61508 and IEC 61511 standards. Even if these are different from ISO 26262, the book still gives good insights that can be used in an automotive context.

When working with functional safety, it is apparent that much of the work is based on various safety analyses. There is one book that gives a good overview of what is most used in an automotive context, written by Ericson [E+15]. This book is well worth reading.

Also, one of the key parts in ISO 26262 and many other safety standards is the argumentation for safety. This is often documented in a Safety Case. To understand more on Safety Cases, [WKM97] gives a good overview. For the argumentation part, the Goal Structuring Notation is both well recognized and an effective approach. This is well described in [KW04].

8.15 Summary

In this chapter we have described how the automotive industry works with functional safety and in particular focused on software development. As apparent in this section, the ISO 26262 standard is the basis for this in the automotive industry. It is quite a significant standard and it is more or less a prerequisite for being in the industry, both for organizations and for individuals.

It is not a standard that is possible to learn overnight, at the same time it is fairly straightforward for some parts like software engineering. As seen in this section, the software-specific details in ISO 26262 is more or less a set of additional rules that one adheres to following normal software development practices.

The reader should also have seen what is typical of ISO 26262; there is no single answer. This is a standard that describes a simplified way of working with functional safety in the automotive industry. As there are many different types of development, this standard has to be adapted to fit each type of development. Hence, the user of this standard has both a lot of flexibility when applying it and at the same time a lot of responsibility with regard to arguing for the choices made, e.g. for the test methods chosen when testing a software unit. There are also differences in how the standard is interpreted in, e.g., different nations, types of vehicles and levels in the supply chain.

There is currently an ongoing revision of ISO 26262 [ISO16], where the significant changes will be in the scope of the standard, now to include all road vehicles except mopeds. There will also be two new parts, one informative part for semiconductors and one normative part for motorcycles. In addition, from the ISO 26262 community there has been work initiated to cover safety of fault-free vehicle-level functions as well as automotive security. These are not yet ready ISO standards.

References

A+08. Motor Industry Software Reliability Association et al. *MISRA-C: 2004: guidelines for the use of the C language in critical systems*. MIRA, 2008.

DV94. Jerry Doland and Jon Valett. C style guide. *NASA*, 1994.

E+15. Clifton A Ericson et al. *Hazard analysis techniques for system safety*. John Wiley & Sons, 2015.

HHK10. Ibrahim Habli, Richard Hawkins, and Tim Kelly. Software safety: relating software assurance and software integrity. *International Journal of Critical Computer-Based Systems*, 1(4):364–383, 2010.

IEC10. IEC. 61508:2010 – functional safety of electrical/electronic/programmable electronic safety-related systems. *Geneve, Switzerland*, 2010.

ISO09. ISO. Quality management systems – particular requirements for the application of iso 9001:2008 for automotive production and relevant service part organizations. *International Standard ISO/TS*, 16949, 2009.

ISO11. ISO. 26262–road vehicles-functional safety. *International Standard ISO*, 26262, 2011.

ISO15. ISO. 9001: 2015 quality management system–requirements. *Geneve, Switzerland*, 2015.

ISO16. ISO. 26262–road vehicles-functional safety. *International Standard ISO*, 26262, 2016.

KW04. Tim Kelly and Rob Weaver. The goal structuring notation–a safety argument notation. In *Proceedings of the dependable systems and networks 2004 workshop on assurance cases*. Citeseer, 2004.

SS10. David J Smith and Kenneth GL Simpson. *Safety Critical Systems Handbook: A Straightfoward Guide To Functional Safety, IEC 61508 (2010 Edition) And Related Standards, Including Process IEC 61511 And Machinery IEC 62061 And ISO 13849*. Elsevier, 2010.

Sto96. Neil R Storey. *Safety critical computer systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

WKM97. SP Wilson, Tim P Kelly, and John A McDermid. Safety case development: Current practice, future prospects. In *Safety and Reliability of Software Based Systems*, pages 135–156. Springer, 1997.