LAPPEENRANTA-LAHTI UNIVERSITY OF TECHNOLOGY LUT
School of Energy Systems
Degree Programme in Electrical Engineering

Simo Janhunen

DEVELOPMENT OF ISO 26262 COMPLIANT SOFTWARE INTEGRATION TESTING FOR ELECTRIC POWER CONVERTERS

Examiners: Prof. Pasi Peltoniemi
D.Sc. (Tech.) Tuomo Lindh

Abstract

Lappeenranta-Lahti University of Technology LUT
LUT School of Energy Systems
Degree Programme in Electrical Engineering

Simo Janhunen

Development of ISO 26262 Compliant Software Integration Testing for Electric Power Converters

Master's Thesis
2021
121 pages, 42 figures, 33 tables and 9 appendices
Examiners: Prof. Pasi Peltoniemi and D.Sc. (Tech.) Tuomo Lindh

Keywords: functional safety, ISO 26262, electric converter, integration testing, safety lifecycle

Electrification, digitalization, and the ever-developing infotainment have created new challenges for the road vehicle manufacturers to tackle in terms of safety. This progression has led to the development of safety guidelines for road vehicles known as the ISO 26262. Danfoss Editron is one of the key players in the electrification movement. They aspire to compete in the fast-growing all-electric and hybrid on-highway vehicle market, where the ISO 26262 is required.

This study investigates the ISO 26262 and aims to provide a straightforward take on the safety lifecycle phases and activities, as well as a more in-depth look to the product development on the software level. A review on software quality is given, where the common themes between ISO 26262 and traditional software quality are discussed. According to the documented software product development activities, an implementation of ISO 26262 compliant integration testing system is developed. The development of said system is done in order to support the future of ISO 26262 compliant electric power converters at Danfoss Editron.

Based on the conducted example test cases, the developed integration testing system worked as intended and fulfilled the requirements set to integration testing by the ISO 26262. The resulting integration testing system could perform cross-platform testing through both continuous integration and local pipelines.

Tiivistelmä

Lappeenrannan-Lahden teknillinen yliopisto LUT
LUT School of Energy Systems
Sähkötekniikan koulutusohjelma

Simo Janhunen

Development of ISO 26262 Compliant Software Integration Testing for Electric Power Converters

Diplomityö
2021
121 sivua, 42 kuvaa, 33 taulukkoa ja 9 liitettä
Tarkastajat: Prof. Pasi Peltoniemi ja TkT Tuomo Lindh

Hakusanat: toiminnallinen turvallisuus, ISO 26262, sähkömuunnin, integraatiotestaus, turvallisuuden elinkaari

Sähköistyminen, digitalisoituminen ja jatkuvasti kehittyvät infotainment-järjestelmät ovat luoneet uusia turvallisuuteen liittyviä haasteita tieajoneuvovalmistajille. Kyseinen kehitys on johtanut turvallisuusohjeiden kehittämiseen tieajoneuvoille, jotka tunnetaan ISO 26262 -standardina. Danfoss Editron on yksi avaintoimijoista sähköistymisen saralla. He pyrkivät kilpailemaan nopeasti kasvavilla täyssähkö- ja hybridiajoneuvomarkkinoilla, missä ISO 26262 -yhteensopivuus vaaditaan.

Tämä työ tutkii ISO 26262:ta ja pyrkii tarjoamaan suoraviivaisen katsauksen sen turvallisuuselinkaaren vaiheisiin ja aktiviteetteihin, sekä tarkemman katsauksen tuotekehitykseen ohjelmistotasolla. Lisäksi luodaan katsaus ohjelmiston laatuun, jossa käsitellään ISO 26262:n ja perinteisen ohjelmiston laatutarkastelun yhteneväisyyksiä. Dokumentoitujen ohjelmistokehitysaktiviteettien mukaisesti toteutetaan ISO 26262 -yhteensopiva integraatiotestausjärjestelmä. Kyseinen järjestelmä kehitetään tukemaan ISO 26262 -yhteensopivien konvertterien kehitystä Danfoss Editronilla.

Esimerkkinä suoritettujen testitapausten perusteella, kehitetty integraatiotestausjärjestelmä toimi kuten tarkoitettu ja täytti kaikki ISO 26262:n asettamat vaatimukset ohjelmistojen integraatiotestaukselle. Kehitetty integraatiotestausjärjestelmä kykeni suorittamaan testausta usealla eri alustalla, sekä CI-järjestelmän että paikallisten koontiprosessien kautta.

Forewords

I would like to express my gratitude to everyone who directly or indirectly contributed to the creation of this thesis, as well as to the people who supported me through this process.

I want to thank Risto Tiainen and Danfoss Editron Oy for providing the opportunity to work on such an interesting tpic and giving me the freedom to explore and construct this thesis as I saw most fitting. In addition, I would like to thank Tuomo Lindh and Pasi Peltoniemi for the collaboration over the years and the fruitful discussions that led to the finalization of this thesis. A special thanks to Risto for his unwavering support and willingness to provide guidance when needed the most.

Last but not least, my friends, family, and most notably my significant other Essi deserve my utmost gratitude for supporting me throughout this process.

Simo Janhunen

Lappeenranta, 18.06.2021

Table of Contents

Nomenclature

List of Abbreviations

AIS	Abbreviated Injury Scale
ASIL	Automotive Safety Integrity Level
BDD	Behaviour-Driven Development
BEV	Battery Electric Vehicle
BMS	Battery Management System
CCF	Common Cause Failure
CF	Cascading Failure
CI	Continuous Integration
CMF	Common Mode Failure
CPT	Cross-Platform Testing
CPU	Central Processing Unit
DC/DC	Direct Current to Direct Current
DC	Direct Current
DFA	Dependent Failure Analysis
DFI	Dependent Failure Initiator
DPF	Dual-Point Fault
DTTI	Diagnostic Test Time Interval
DUT	Device Under Test
E/E/PE	Electrical and/or Electronic and/or Programmable Electronic
E/E	Electrical and/or Electronic
ECC	Error Correction Code
ECT	Equivalence Class Testing
ECU	Engine Control Unit
EOTI	Emergency Operation Time Interval
EOTTI	Emergency Operation Tolerance Time Interval
EPA	Environmental Protection Agency
FDTI	Fault Detection Time Interval
FFI	Freedom From Interference

FHTI	Fault Handling Time Interval
FIT	Fault Injection Testing
FMEA	Failure Mode and Effects Analysis
FMEDA	Failure Modes, Effects, and Diagnostics Analysis
FPGA	Field-Programmable Gate Array
FRTI	Fault Reaction Time Interval
FSC	Functional Safety Concept
FSR	Functional Safety Requirement
FTA	Fault Tree Analysis
FTTI	Fault Tolerant Time Interval
HARA	Hazard Analysis and Risk Assessment
HAZOP	Hazard and Operability Study
HEV	Hybrid Electric Vehicle
HIL	Hardware-In-the-Loop
HSI	Hardware/Software Interface
HV	High Voltage
I/O	Input/Output
ICE	Internal Combustion Engine
KDT	Keyword-Driven Testing
LFM	Latent Fault Metrics
LV	Low Voltage
MBD	Model-Based Development
MC/DC	Modified Condition/Decision Coverage
ML	Machine Learning
MPF	Multiple-Point Fault
PHEV	Plug-In Hybrid Electric Vehicle
PMHF	Probabilistic Metrics for Hardware Failures
PSU	Power Supply Unit
QM	Quality Management
QMS	Quality Management System
RBT	Requirements-Based Testing
RF	Residual Fault

RPA	Robotic Process Automation
SEooC	Safety Element out of Context
SF	Safe Fault
SG	Safety Goal
SOTIF	Safety Of The Intended Functionality
SPF	Single-Point Fault
SPFM	Single-Point Fault Metrics
SQA	Software Quality Assurance
SQM	Software Quality Management
SWSR	Software Safety Requirement
TDD	Test-Driven Development
TSC	Technical Safety Concept
TSR	Technical Safety Requirement

1 Introduction

Functional safety is a system state where the absence of unreasonable risk due to hazards caused by malfunctioning behaviour of electric/electronic/programmable electronic (E/E/PE) systems has been achieved [1]. The term functional safety comes from the standard IEC 61508 *"Functional safety of electrical/electronic/programmable electronic safety-related systems"*. The first edition of the standard was released in 2001, being the first to describe the activities and methods involved in achieving functional safety in E/E/PE safety-related system. Functional safety does not completely cover the safety of an E/E/PE system or a equipment, but is a part of the overall safety. The IEC 61508 discusses functional safety in more general terms and does not specify the type of E/E/PE systems it is implemented to. Due to its shortcomings as a general purpose toolbox, the IEC 61508 has inspired a variety of child standards with specialized fields, most notably, road vehicles, railways, and nuclear power plants [2]. [3]

The functional safety of road vehicles is discussed in the ISO 26262. The ISO 26262 offers guidance on the safety lifecycle of the electrical/electronic (E/E) safety-related systems in road vehicles. The ISO 26262 term *"E/E"* also includes programmable electronics. The safety lifecycle described in the ISO 26262 can be divided into six distinct parts as given in Figure 1. [1]

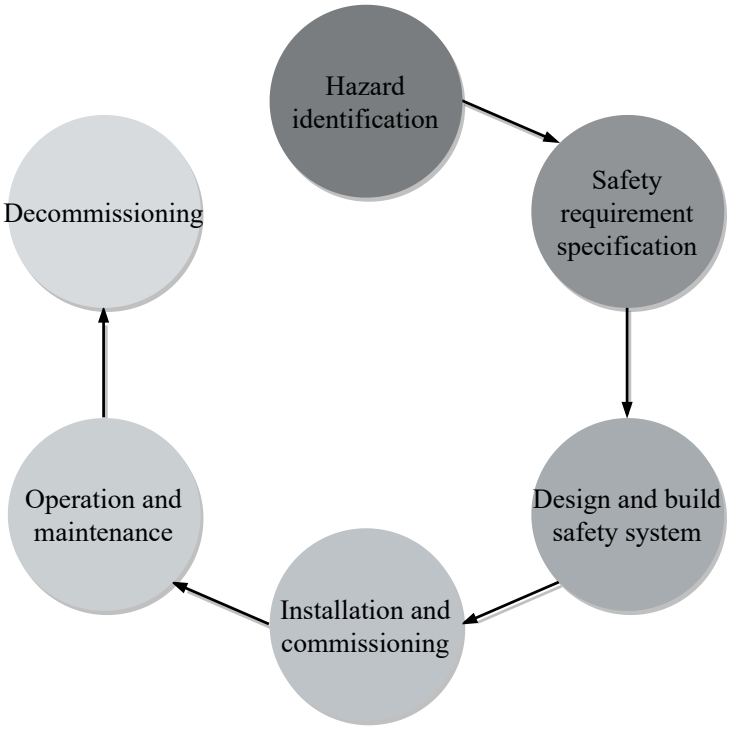


Figure 1. The safety lifecycle of an automotive system.

During the hazard identification phase, hazard are identified and are assigned with automotive safety integrity level (ASIL) that corresponds to the the three aspects: severity, probability of exposure, and controllability. There are four ASILs from the least stringest A to the stringest D. The following phases seek to reduce the level of risk to a reasonable level. [1]

The main motivation behind the ISO 26262 was to create industry-wide guidelines so that the used activities and methods are uniform and the collaboration between different automotive entities can be effectively established, e.g., through the development and use of safety elements out of context (SEooC). The follow-up phases seek to reduce the risk to a level that is reasonable. [2]

The electrification of a conventional car adds to the complexity of its systems, it also adds the possibility of using various topologies and approaches for the manufacturers. In practice, the conventional internal combustion engines (ICE) and fuel tanks are replaced with an electric motor, electric converters, and an energy storage (e.g., battery packs). The most commonly met electric vehicle topologies are presented in Figure 2.

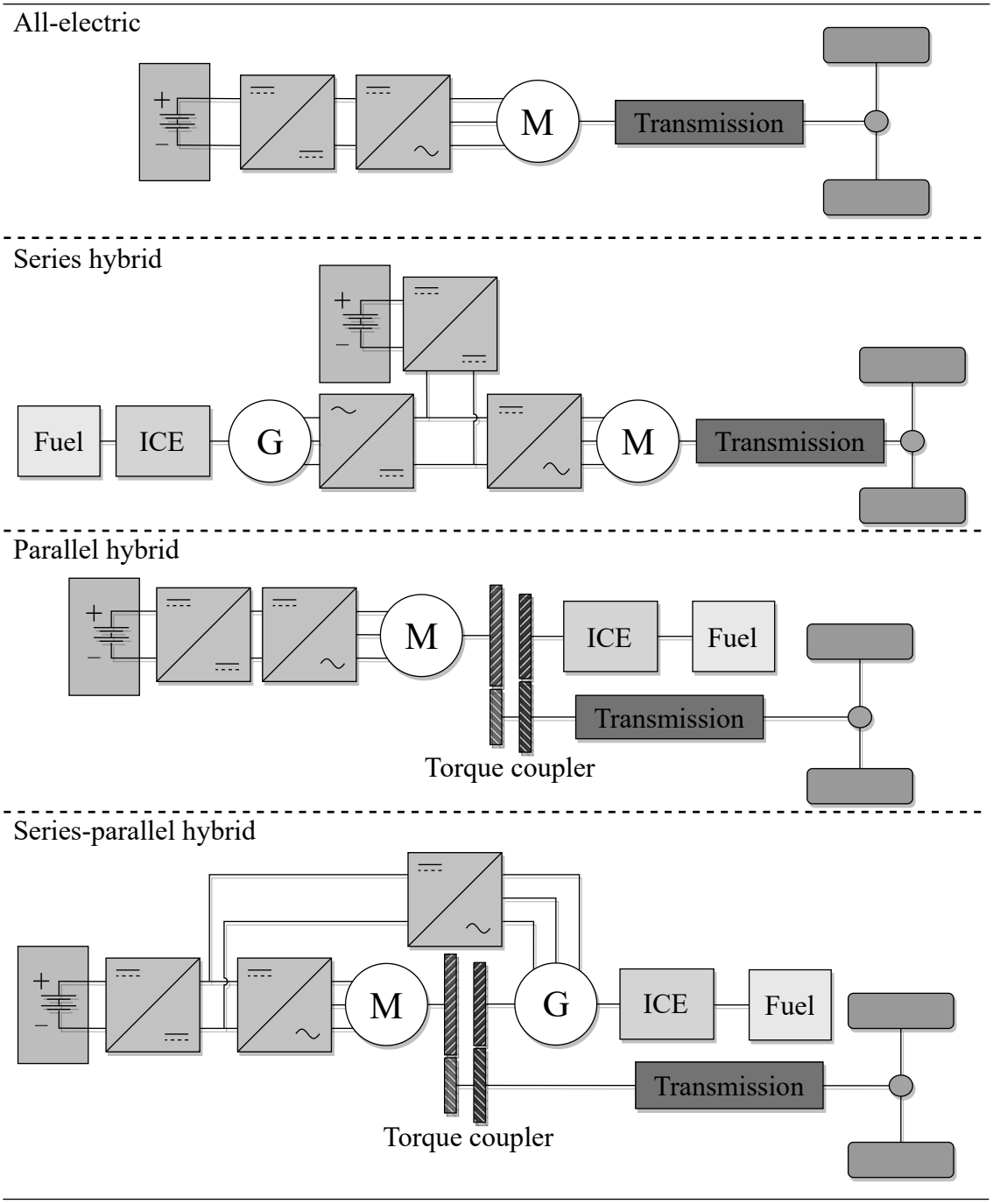


Figure 2. A variety of hybrid and all-electric vehicle topologies. [4], [5]

The topologies presented are simplified versions of the actual used drivetrains. They describe the energy conversion from reservoir (e.g., petrol) to shaft torque and finally, wheel torque. Direct current to direct current (DC/DC) converters are used to balance the charging and discharging of possible battery packs and supercapacitors to achieve sufficient direct current (DC) link voltage. One of the most prominent problems with battery pack run or assisted systems is the balancing of the charge of the battery packs

when regenerative braking is a sought feature. The charge of the battery packs cannot be too high to avoid overheating nor too low to still produce adequate torque. Since the commercial vehicles' power cycle is not deterministic, a control strategy to momentarily disable regenerative braking is needed. [5], [6]

According to U.S. Environmental Protection Agency (EPA), the regenerated braking torque can recover up to 17 % of the power fed to the wheels. The conventional ICE drivetrains have efficiency from 12 % to 30 % and modern all-electric drivetrains have efficiency from 69 % to 73 % [7]. This means that in the ideal conditions the theoretical increase in efficiency can be as much as 508.3 %, and when regenerative braking is included, the increase in efficiency can become as high as 650.0 %. [8]

As a result of this movement towards better efficiency, the number of all-electric and hybrid on-road vehicles has risen steadily for the past 10 years in Western countries and China. According to a report by IEA [9], China was the leader of the vehicle electrification with 3.35 million electric vehicles in 2019. The same trend can be found on the Finnish car market; the number of plug-in hybrid electric vehicles (PHEV), battery electric vehicles (BEV), and hybrid electric vehicles (HEV) has risen increasingly in the past five years [10]. Figure 3 depicts the number of electric cars in Finnish car fleet from 2015 to 2019.

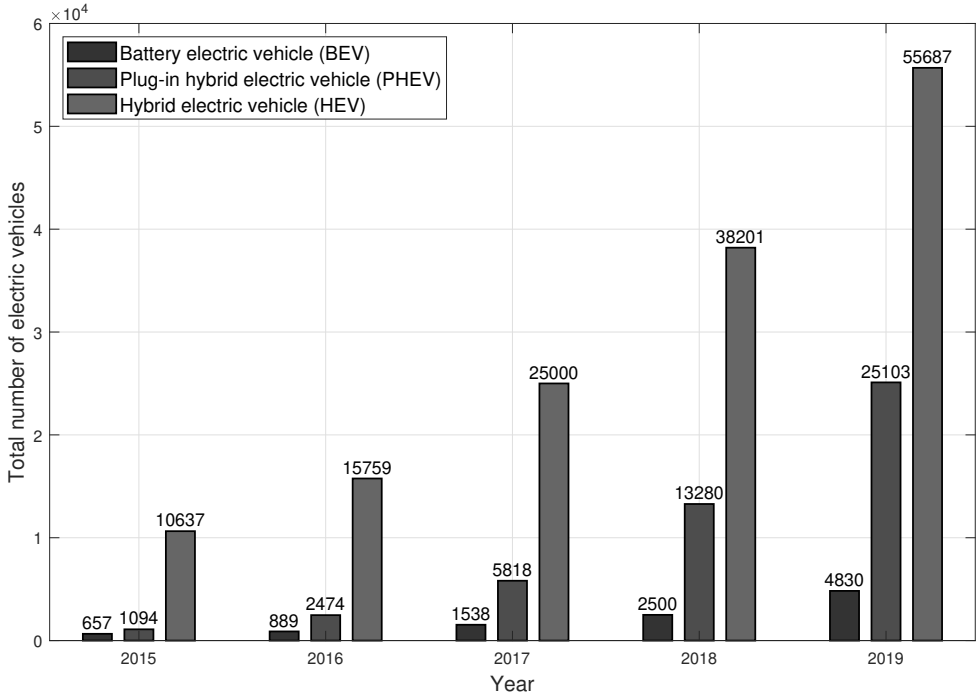


Figure 3. The yearly total number of electric vehicles in the Finnish car fleet. [10]

The share of electric vehicles has been superseding the conventional energy sources like petrol and diesel. The share of electric vehicles in respect to the total number of vehicles is given in Table. 1.

Table 1. The yearly share of electric vehicles in respect to total number of vehicles. [10]

Year	**Total number of vehicles (Proportional change %)**	**Share of electric vehicles**
2015	3,234,860 (+ 1.96 %)	0.38 %
2016	3,322,672 (+ 2.71 %)	0.58 %
2017	3,398,937 (+ 2.30 %)	0.95 %
2018	3,470,507 (+ 2.11 %)	1.56 %
2019	3,549,803 (+ 2.28 %)	2.41 %

Table 1 suggests that the share of electric vehicles is not going to diminish anytime soon, on the contrary. The presented progression is precisely why the functional safety is needed to combat the foreseeable change in the global car market.

1.1 Background and Motivation

Danfoss Editron is a provider of sophisticated hybrid and fully electric powertrain systems for heavy-duty and commercial vehicles and machines, including on-highway, off-highway, and marine applications. The on-highway applications include, e.g., buses, trucks and passenger vans. The off-highway applications include, e.g., harbour equipment, airport equipment, mining machinery, and construction machinery. Lastly, the marine applications include, e.g., workboats and ferries. [11]

Danfoss Editron aspires to be a top contender when it comes to the road vehicle electrification. In order to beat the market, industry standards like the ISO 26262 must be complied to, and the ability to adapt must be greater than that of the competitors. Currently there are no general-purpose safety-related software libraries or components that would comply with the ISO 26262. As the functional-safety-related products have been developed as custom pieces, there are no uniform processes that could be followed. The process of building each and every product from the ground up creates lengthier product development times and requires extra man-hours in each safety lifecycle phase.

The development times and product quality could be improved drastically if general-purpose components, libraries, and processes were introduced company-wide.

The integration of the ISO 26262 processes to the product development cycle would benefit the off-highway applications. The functional safety of off-highway vehicles is discussed in various standards depending on the application, more generally in the IEC 62061. As child-standards of the IEC 61508, the standards share similar traits and subprocesses, consequently the standards could be implemented with less effort. [12]

1.2 Related Works

The fact that the first edition of the ISO 26262 standard was released in 2011 can be seen as a low number of studies that have been conducted on it. Henriksson *et al.* [13] studied how machine learning (ML) can be adapted to the ISO 26262. The applying of ML in ISO 26262 compliant systems has since been moved under ISO/PAS 21448, which discusses intended functionality that requires more complex algorithmic evaluation, and sensory data processing [14].

The applying of ISO 26262 requires a great number of specifications, a solution of some extent was discussed by Lu *et al.* [15] where they studied the applying of failure modes, effects, and diagnostics analysis (FMEDA) on the software development process to reduce the effort to argue hardware's safety.

The identification of underlying failure modes is of utmost importance. To ease said process, Kim [16] proposed a solution to further extend the activities mentioned in the ISO 26262 by applying hazard and operability studies (HAZOP) methodology on the software development. Ward *et al.* [17] studied the applying of ASIL decomposition in great extent, as well as its relation to the parent standard IEC 61508. Lee *et al.* [18] discussed software safety requirement testing using Hardware-in-the-Loop (HIL) as well as the derivation of test case specification using a 2-step methodology.

Apart from Dhaked *et al.* [19], none of the studies focused on the overall process regarding the safety lifecycle depicted in the ISO 26262. Dhaked *et al.* discussed the first lifecycle part, i.e., concept phase, while omitting the rest. The aforementioned studies focused on relatively small issues that are not greatly described in the ISO 26262. Hence,

the safety lifecycle has not been studied but to a small extent.

1.3 Objectives

The objective of this thesis is to record the ISO 26262 safety lifecycle process and the key supporting processes in straightforward summaries, to provide in-depth look to the ISO 26262 software product development, and to implement integration test framework accordingly. The safety lifecycle is recorded to fill the research gap. In addition, a review on software quality is given, this is done in order to support the development of the integration testing system.

The aim of the integration testing is to develop integration testing system that enables cross-platform testing (CPT) and multiple run sources, such as continuous integration (CI) pipelines and both Linux-based and Windows operating systems locally. The integration testing targets ISO 26262 compliant electric power converters that are still in development.

To accomplish the aforementioned objectives, this thesis seeks answers to following questions

- Can the ISO 26262 be implemented on electric power converters whilst successfully reaching highest automotive safety integrity level compliance from the software perspective? If so, what are the requirements?
- How to implement ISO 26262 compliant integration testing for all available runtimes, i.e., enable cross-platform testing?
- Is it possible to run integration testing through continuous integration? If so, how?

1.4 Delimitations

The functional safety and the software quality are substantially large topics and therefore certain delimitations are imposed. The functional safety of motorcycles and semiconductors (Parts 11 and 12) are not discussed. The product development on the hardware level (Part 5) is not discussed directly, but is used to complement topics that involve system or software development, e.g., hardware/software interface.

Since the ISO 26262 can be implemented on wide-range of systems there often is incongruity between the studied system and the requirements. ISO 26262 seeks to solve

these conflicts through requiring rationale when requirements are bypassed or altered. Situations where standard-diverging rationale is required are not taken into consideration, as this thesis aspires to offer generalized overview of the ISO 26262. Also, any normative, i.e., strictly recommendation-giving, parts of the ISO 26262 are omitted.

The software quality is considered from the standpoint of the ISO 26262:6 *"Product development at the software level"*. The subject is being approached from the product developers' perspective. The focus is on how software quality is achieved through different testing levels and techniques. Everything software quality-related surpassing the scope of ISO 26262's quality management and verification methods is omitted.

In the practical integration testing section the robotic process automation (RPA) aspect of Robot Framework is omitted. Python is used as the supporting implementation language.

1.5 Structure

This study is divided in to two distinct parts. The first part is a literature review concerning the functional safety of road vehicles and the software quality. The literature review utilizes the industry standards and is complemented by supporting literature and research. The second part presents the vision for the future Danfoss Editron converters and the integration testing framework. In addition, the vision for the test architecture and test implementation are presented. The second part includes a detailed description of the Robot scripting language, showcases the Robot environment in general, and offers practical test case implementations. The literature review is utilized as extensively as possible.

The first part is discussed in sections 2 and 3, whereas the second part is discussed in section 4. The structure of the sections is as follows

Section 2 discusses the ISO 26262. It summarizes the main points of the safety lifecycle and key supporting processes. The supporting processes are presented after the structure, which is then followed by short introduction to the safety lifecycle and its phases. The phases are further divided in to subphases. The safety lifecycle precedes more detailed depictions of each subphase. Some of the used representation uses *"A-B"*

and *"ISO 26262:A-B.C.D"* notations to indicate connection to part A and clause B of the ISO 26262 standard.

Section 3 discusses software quality. The section does not seek to offer complete depiction of software quality, but gives rough overview of the topic and focuses on the similarities and overlap between ISO 26262 and conventional software quality, i.e., common themes. Topics such as testing on the unit, integration, and system level are considered as well as various testing techniques, e.g., requirements-based testing.

Section 4 presents the vision for the future of electric converters and the integration testing framework. In addition, the section gives short introduction to Robot scripting and showcases some of the core functionality in terms of creation and implementation of new test cases. Finally, the vision for the test architecture and the ISO 26262 compliant practical test case implementations are presented.

Section 5 concludes the proceedings of this study and offers further issues to study.

2 Road Vehicle Functional Safety

This section discusses the functional safety activities described in the ISO 26262. Where applicable, the topics are complemented by their relation to other standards, studies and literature.

In order to achieve functional safety in an item, a number of aspects must be considered. The most critical aspects can be reduced to five main points

- automotive safety lifecycle and activities during the lifecycle phases, i.e., concept phase, product development, production, operation, service, and decommissioning;
- automotive-specific hazard analysis and risk assessment (HARA) to determine ASILs;
- avoidance of unreasonable residual risk through cross-referencing of ASILs and ISO 26262 requirements;
- requirements for management, design, implementation, verification, validation and confirmation measures; and
- requirements for relations between customers and suppliers. [1]

2.1 Structure of the ISO 26262

The ISO 26262 is divided into 12 parts, each having a specific scope. The parts and their clauses are given in Appendix A.

Parts 1, 10, and 11 offer definitions and summaries regarding the whole system, whereas the rest of the parts are divided into clauses that use the following structure

- objectives,
- general,
- inputs to this clause,
- requirements and recommendations, and
- work products.

Objectives summarizes the reasoning and the desired end result behind each clause, general discusses external information that is important to the clause and summarizes the topic, inputs to this clause states what should have been done so far (e.g., specifications), requirements and recommendations discuss how the objectives of this clause are

to be or should be achieved, and work products summarizes the end results based on the requirements and recommendations. [1]

The requirements and recommendations regarding each clause are given in terms of ASILs, method categorization, and notation methods. The method categorization indicates the degree of recommendation for the corresponding ASIL, the methods are categorized as

- "++" highly recommended for the identified ASIL,
- "+" recommended for the identified ASIL, and
- "o" no recommendation for or against the use for the identified ASIL [20].

Notation methods refer to the language used in concept and design phases, it defines how precisely the corresponding documents are to be defined. The notation methods used are

- natural language,
- formal notations,
- informal notations, and
- semi-formal notations.

Natural language indicates that there are no restrictions for syntax nor semantics, formal notations indicate that both syntax and semantics are completely defined, informal notations indicate that neither syntax nor semantics are completely defined and semi-formal notations indicate that syntax is completely defined whereas semantics are incomplete. Natural language and informal notations are used for higher level requirements, e.g., safety goals (SG), where formal notations cannot adequately convey the message. Formal and semi-formal notations are used on low level requirements, e.g., precise software behaviour, where detail and structure provide greater clarity. [1], [21]

2.2 Item Dissolution

Item dissolution defines how an item and its elements are divided into smaller separable elements. The term element applies to the entities *"system"*, *"component"*, *"hardware part"* and *"software unit"*. Hardware parts can further be divided into subparts and elementary subparts while unit is the smallest measure for software dissolution. A system can be divided into any number of subsystems and a component can be divided to sub-

components, but in order to do so, the subsystems have to be hierarchically structured. Figure 4 depicts how item dissolution is regarded in the ISO 26262. [2], [22]

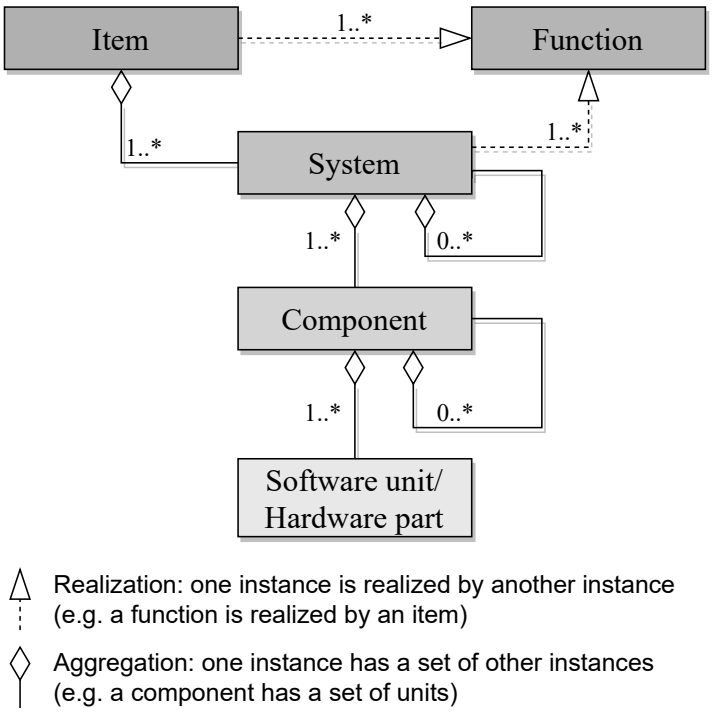


Figure 4. The item dissolution according to ISO 26262. [22]

In modern road vehicles, the item usually refers to the vehicle itself and the system refers to, e.g., an engine control unit (ECU) [2]. An example of the item dissolution is given in Appendix B.

2.3 Faults, Errors and Failures

Fault is an abnormal condition that causes a trickle down effect of faults turning into errors and errors turning into failures. Finally, the trickle down effect causes the element or the item to fail, possibly causing more faults in upper echelons. Errors are considered as a discrepancy between computed, observed, or measured value or condition, and the true value or condition. Failure implies termination of intended behavior due to fault manifestation in an element or an item. Failures are considered as faults on the upper echelon, e.g., failures on system level are faults item level. Failures inherit the type of the fault that caused it, e.g., a single-point fault (SPF) causes a single-point failure. The trickle down effect is depicted in Figure 5. [1], [22]

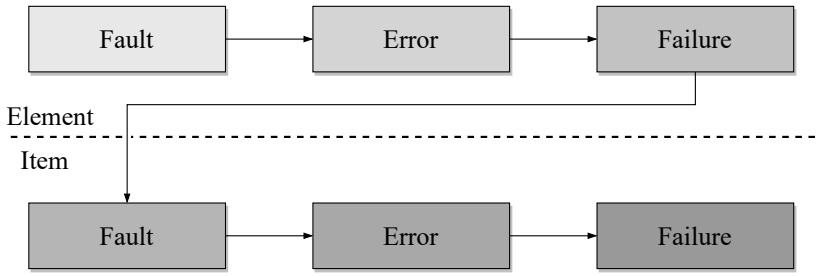


Figure 5. Visualization of the trickle down effect of failures in the lower echelons turning in to faults in the upper echelons.

Faults are divided into three groups: systematic software faults, systematic hardware faults, and random hardware faults. Systematic faults are brought on by design or specification issues and random faults are faults coming from indeterminable sources. Three types of faults are considered: permanent, intermittent and transient faults. Permanent faults stay permanently active after occurring, intermittent faults occur from time to time and then disappear again, and transient faults are one-off occurring faults that subsequently disappear. [1], [22]

2.3.1 Systematic Faults

Systematic faults include both systematic software faults and systematic hardware faults. Systematic faults can be eliminated by measures such as a change in the design, manufacturing process, operational procedures, documentation or other relevant factors. Failures caused by a systematic fault are manifested in a deterministic way, i.e., the cause of the failure is certain. [1]

2.3.2 Random Faults

Random faults consist of only random hardware faults. Random hardware faults are faults coming from indeterminable sources, they are analysed and categorized to determine their impact to a safety goal. The generic random hardware fault analysis is limited to combinations of two independent faults, or dual-point faults (DPF). If the analysis leading to functional or technical safety concept dictates that the point faults of more than two are relevant, then those must be considered as well. In most cases, random hardware fault classes include SPFs, residual faults (RF), detected DPFs, perceived DPFs, latent DPFs and safe faults (SF). [1], [22]

SPF is a random hardware fault in an element that leads directly to the violation of the safety goal, none of the faults in that specific element are covered by safety mechanisms. RF is a portion of a random hardware fault that occurs in hardware element and by itself leads to the violation of a safety goal, the portion of the specific random hardware fault is not controlled by a safety mechanism. [1], [22]

Multiple-point faults (MPF), such as the DPFs require one other independent fault to violate a safety goal. Detected DPFs are detected by safety mechanism that prevents them from becoming latent. Perceived DPFs are perceived by the driver with or without the detection by a safety mechanism within a prescribed time. Latent DPFs are neither perceived by the driver nor detected by the safety mechanism, the system is operable and the driver uninformed about the fault. Together detected DPFs, perceived DPFs and latent DPFs make up the MPFs. [1], [22]

SFs are categorized in two groups; faults that will not contribute to the violation of a safety goal and all point faults greater than two, unless safety concept proves them otherwise relevant [1], [22]. Flow diagram for analysing failure modes for random hardware faults given in Appendix C.

2.3.3 Fault Metrics

The ISO 26262 introduces a number of hardware fault metrics: single-point fault metrics (SPFM), latent fault metrics (LFM), and probabilistic metrics for hardware failures (PMHF). SPFM indicates whether or not the coverage by the safety mechanisms is sufficient to prevent risk from single-point or residual faults in the hardware architecture. Likewise, LFM indicates whether or not the coverage by the safety mechanisms is sufficient to prevent risk from latent faults in the hardware architecture. PMHF considers all of the system's fault patterns and indicates the probability of faults occurring in it.

The ISO 26262 defines quantitative indicators for the fault metrics as a requirement for ASILs exceeding ASIL A. The reference target values for SPFM, LFM and PMHF with corresponding ASILs are given in Table 2. [23]

Table 2. Reference target values for SPFM, LFM and PMHF. [23]

Metrics	ASIL B	ASIL C	ASIL D
Single-point fault metric	≥90 %	≥97 %	≥99 %
Latent fault metric	≥60 %	≥80 %	≥90 %
Probabilistic Metrics for Hardware Failures	$<10^{-7}h^{-1}$	$<10^{-7}h^{-1}$	$<10^{-8}h^{-1}$

These quantitative targets are intended to provide guidance on design and evidence that the design complies with the safety goals. [23]

2.3.4 Failure Types

There are several different kinds of failure types, cascading failures (CF), common cause failures (CCF), and common mode failures (CMF). CF is a failure of an element of an item. CFs are caused by either internal or external root cause. The root cause leads to a failure of the element or elements of the same or different item. CCFs are failures of two or more elements of an item. CCFs are caused by a single specific event or root cause, that leads to the failure of all of these elements. The root cause may be internal or external. CMFs are a subset of CCFs where two or more elements fail in the same manner. Dependent failures include CCFs and CFs. Figure 6 gives visualization on the CF and CCF types. [1], [2]

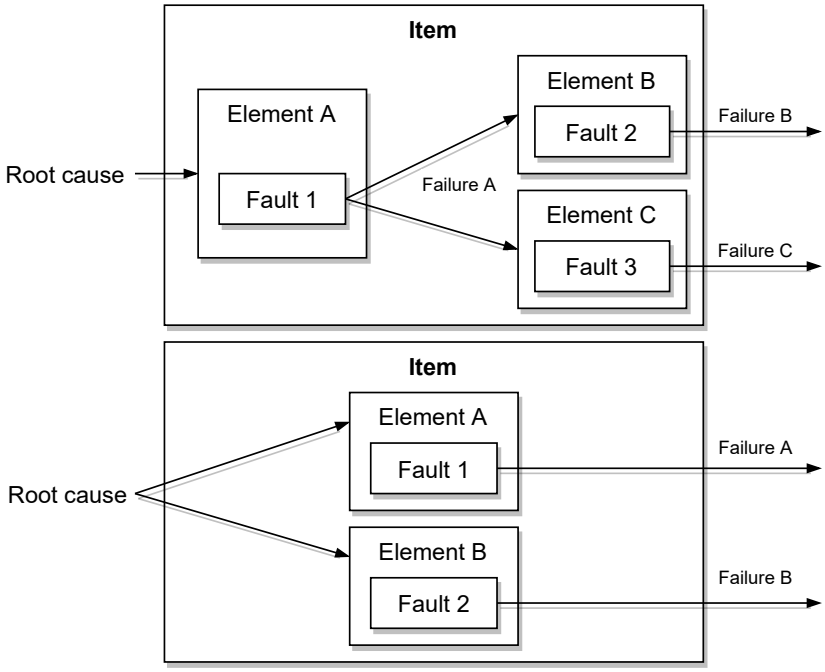


Figure 6. Visualization for CF (top) and CCF (bottom). [1]

Dependent failures are failures that statistically occur more likely when combined rather than independently. Dependent failures occur when single root cause, i.e., dependent failure initiator (DFI), leads to multiple elements to fail through coupling factors. Coupling factor is common characteristic or relationship between the elements that creates dependency in their failures. The coupling factors are identified during dependent failure analysis (DFA). [1], [24]

2.3.5 Dependent Failure Analysis

The DFA analyzes the relationship of two or more elements to prove that they have either sufficient technical independence or sufficient freedom from interference (FFI), and thus the absence of dependent failures. The analysis also defines safety measures to mitigate plausible dependent failures when necessary. These key qualities are used to justify ASIL decomposition and ASIL tailoring-related decision-making. The identification of the potential dependent failures can be based on either deductive or inductive analyses, e.g., fault tree analysis (FTA) or failure mode and effects analysis (FMEA). Deductive analysis refers to analysing known effects and seeking the causes, whereas inductive analysis refers to analysing known causes and seeking the effects. The inductive and deductive analysis complement each other. [24]

Each identified potential dependent failure must be evaluated to determine its plausibility, i.e., see if the technical independence or the FFI between given elements is being violated by reasonably foreseeable cause. The evaluation includes considering different operational situations and operating modes of the item or element that is being analysed. Where applicable, the following topics are considered: random hardware failures, development faults, manufacturing faults, installation faults, service faults, environmental factors, failures of common external resources or information, stress due to specific situations, and ageing and wear. The analysis has to have a level of detail and rigour that is suitable in demonstrating the achievement of the required level of independence or FFI. [24]

2.4 Fault Tolerance

Fault tolerance is the ability to deliver a specified functionality in the presence of one or more specified faults. Fault tolerance has several metrics to communicate the ability to handle, detect and react to fault situations, more namely, fault tolerant time interval (FTTI), fault detection time interval (FDTI), fault reaction time interval (FRTI), fault handling time interval (FHTI) and emergency operation time interval (EOTI), emergency operation tolerance time interval (EOTTI) and diagnostic test time interval (DTTI). [1]

Emergency operation is an operating mode of an item for providing safety after fault is detected up until the transition to a safe state is achieved. Safe state is an operating mode of an item in the case of a failure without unreasonable level of risk. Safe state can be transitioned to without an emergency operation, but in cases where the safe state (e.g., stopping the vehicle) requires preparations, an emergency operation (e.g., run on reduced traction torque) can be specified. [1]

The FTTI is the minimum time-span from the occurrence of a fault in an item to a possible occurrence of a hazardous event, if safety mechanisms are not activated. The FTTI is assigned to the safety goal that is derived from a specific identified hazard, it describes the utmost maximum time-span the item can sustain the malfunctioning behaviour. The FDTI is the time-span from the occurrence of a fault until its detection, it is determined independently of the DTTI. The FRTI is the time-span from the detection of a fault to reaching a safe state or an emergency operation, it depicts the time-span it

takes to realize the designated safe mode, i.e., emergency operation or safe state. The FHTI is the sum of the FDTI and the FRTI, i.e., the time it takes to transition to either the emergency operation or to the safe state after a fault has caused malfunction. The FHTI is a characteristic of a given safety mechanism, it depicts both the time-span the detection takes and the time-span to realize the designated safe mode, i.e., emergency operation or safe state. The EOTI is the specified time-span where the emergency operation is kept before moving on to the safe state. The EOTTI is the specified maximum time-span where the emergency operation can be maintained without an unreasonable level of risk. In order to operate with the absence of unreasonable risk, the EOTTI must be longer than the EOTI. The DTTI is the amount of time in between executions of online diagnostic tests by a safety mechanism, it includes the duration of the test's execution itself. [1], [22]

Figure 7 depicts the safety relevant time intervals when malfunctioning behaviour is not detected.

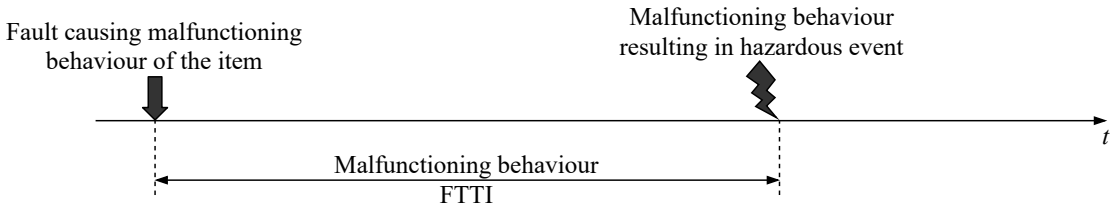


Figure 7. Safety relevant time intervals without safety mechanism. Adapted from [1].

Since there was no activated safety mechanism to avoid malfunctioning behaviour, the hazardous event might occur once the FTTI is exceeded. Fig. 7 does not specify whether the lack of reaction was due to there being no safety mechanism, or the fact that FTTI time-span was shorter than FHTI time-span, i.e., the safety mechanism was not fast enough. [22]

Figure 8 depicts the safety relevant time intervals when a malfunctioning behaviour is detected, but no emergency operation is designated.

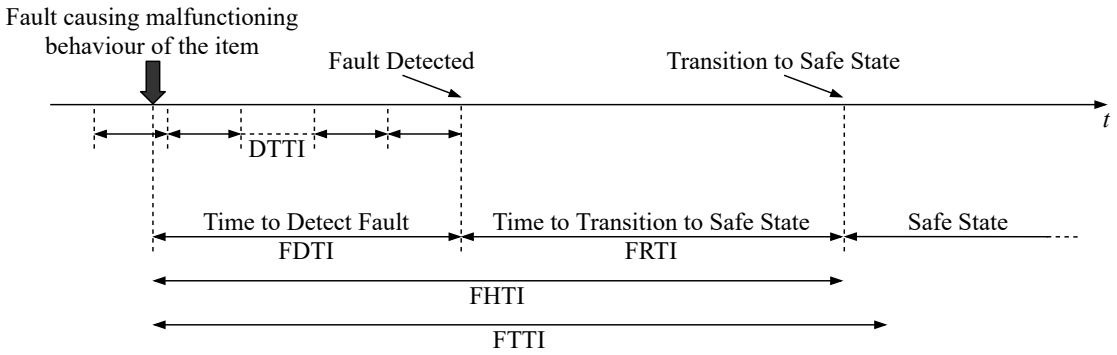


Figure 8. Safety relevant time intervals with safety mechanism but without designated emergency operation. Adapted from [1].

A safety mechanism was implemented that detected the fault. Since there was no determined emergency operation, the item transitions to the safe state. If the FTTI was smaller than the FHTI, the hazardous event might occur. [22]

Figure 9 depicts the safety relevant time intervals when a safety mechanism is activated and an emergency operation is specified.

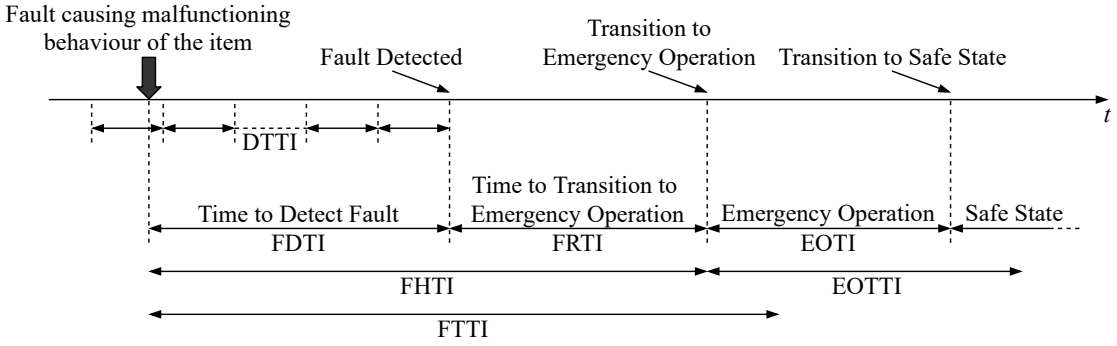


Figure 9. Safety relevant time intervals with safety mechanism and emergency operation. Adapted from [1].

The item has both implemented safety mechanism and a determined emergency operation. The EOTI is smaller than the EOTTI, so the item has enough time to transition to the safe state. On the contrary, if the EOTI was larger than the EOTTI, the cumulated risk would become unacceptable, exceeding the target specified in the safety concept. [22]

2.5 ASIL Decomposition

ASIL decomposition is a method of ASIL tailoring where the apportioning of redundant safety requirements to elements with the objective of reducing the ASIL of the said requirements is done. As a prerequisite, the elements need to have sufficient independence and are to conduct to the same safety goal. The ASIL decomposition can be done only after HARA analysis has been conducted, this is due to there being no ASILs to decompose. [1]

A initial safety requirement is decomposed to redundant safety requirements, these redundant safety requirements are then implemented by sufficiently independent elements. The independence is evaluated using the DFA. Each redundant safety requirements is to comply with the initial safety requirements by itself. ASIL decomposition does not apply to random hardware failures, i.e., safety goal violations due to random hardware failures remain unchanged. [22]

When ASIL D is being decomposed, the following equation is used

$$\text{ASIL D} = \begin{cases} \text{ASIL B (D)} + \text{ASIL B (D)}, \\ \text{ASIL A(D)} + \text{ASIL C (D)}, \\ \text{ASIL QM (D)} + \text{ASIL D (D)}, \end{cases} \tag{1}$$

when ASIL C is being reduced

$$\text{ASIL C} = \begin{cases} \text{ASIL B (C)} + \text{ASIL A (C)}, \\ \text{ASIL QM(C)} + \text{ASIL C (C)}, \end{cases} \tag{2}$$

when ASIL B is being reduced

$$\text{ASIL B} = \text{ASIL A (B)} + \text{ASIL A (B)}. \tag{3}$$

and finally, when ASIL A is being reduced

$$\text{ASIL A} = \text{ASIL A (A)} + \text{ASIL QM (A)}. \tag{4}$$

An example of the ASIL decomposition is given in Appendix D. [17]

2.6 Safety Lifecycle

Safety lifecycle of an ISO 26262 compliant E/E system is divided into three phases

- concept phase,
- product development phase, and
- production, operation, service, and decommissioning phase.

Figure 10 depicts the phases of the safety lifecycle.

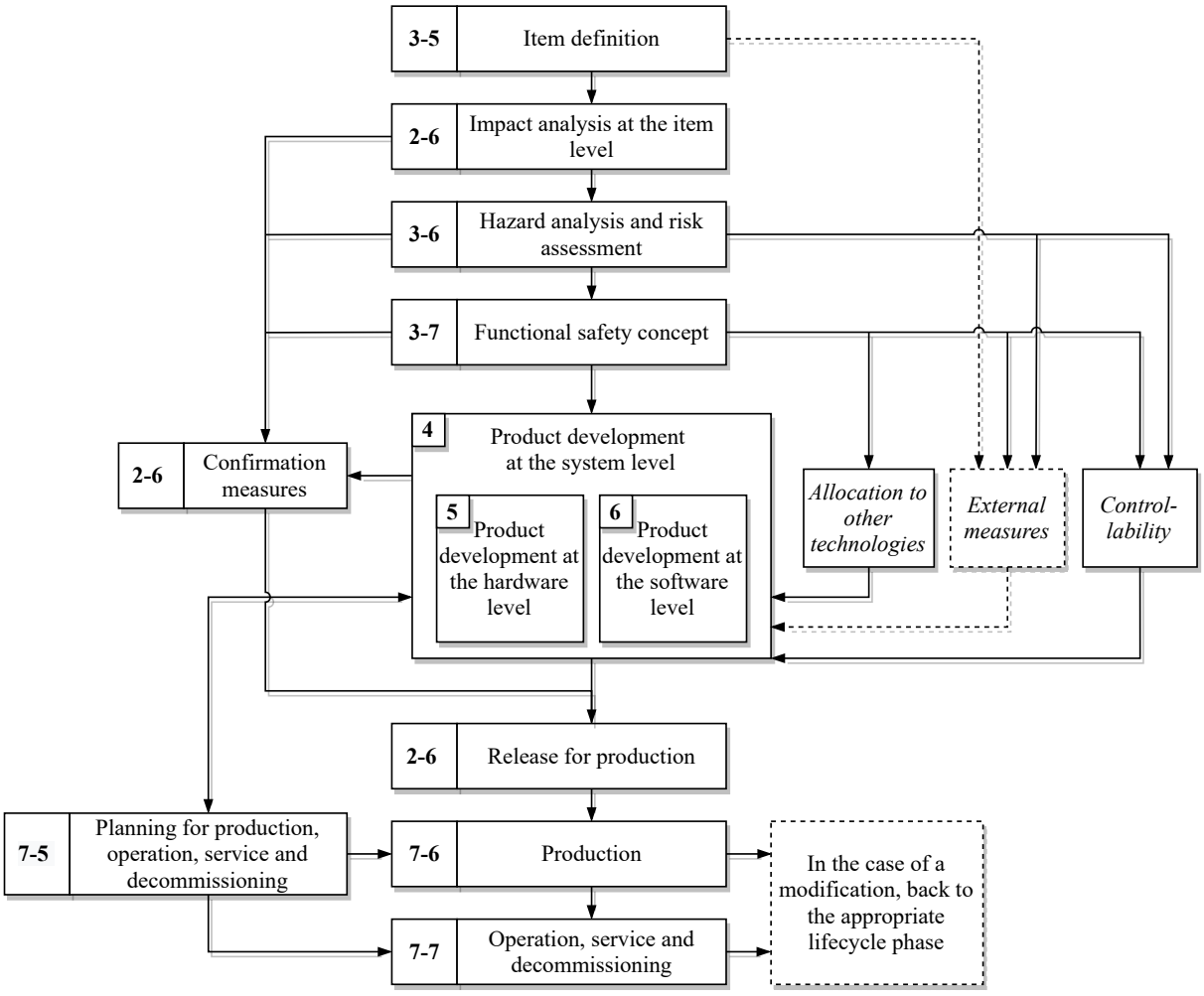


Figure 10. Safety lifecycle in relation to overall safety management. Adapted from [25].

Each phase of the safety lifecycle is subject to verification. Each phase has unique criteria where that specific phase is being evaluated along with its subphases. Safety validation is providing evidence that the safety goals are achieved by the item when integrated into

respective vehicle or vehicles, and the functional safety concept (FSC) and the technical safety concept (TSC) are appropriate for achieving functional safety for the item. The achievement of functional safety is evaluated through controllability, effectiveness of external measures and elements of other technologies, and the assumptions made during HARA that can be only checked in the final product. [20], [26]

Concept phase includes item definition, HARA, FSC, and impact analysis on item level. Impact analysis on element level is only performed when existing element is being reused. The concept phase is subject to project dependent safety management (ISO 26262-2:6). Subsections 2.7–2.9 discuss the concept phase subphases in more detail. [25]

Item definition is where the safety lifecycle is initiated. Item definition develops the description of the item with regard to its functionality, interfaces, known hazards, etc. In addition, it determines the boundaries of the item and its interfaces and makes assumptions concerning other items, elements and external measures. [1], [25]

The HARA identifies and categorizes hazardous events of items and specifies the corresponding safety goals and ASILs related to the prevention or mitigation of the associated hazards in order to avoid unreasonable risk. [1], [25]

FSC is developed based on the identified safety goals whilst considering the preliminary architectural assumptions. The FSC is developed by deriving functional safety requirements (FSR) from the safety goals and by allocating them to the elements of the item. FSC may include other technologies or rely on external measures, or both. [25]

The impact analysis on item level determines whether the item is a new development, a modification of an existing item, or an existing item with a modified environment. If more than one modifications need to be made, the implications of the modifications on functional safety are analysed. [25]

Verification of the concept phase includes ensuring that the concept is correct, complete and consistent with respect to the boundary conditions of the item, and that the defined boundary conditions are also correct, complete and consistent. [20]

Product development phase includes subphases for each level of product development: system, hardware, and software, as well as other key concepts like confirmation measures, external measures, other technologies, and controllability. It also withholds the plan-

ning and decision-making around the release of the safety system. The product development phase is subject to project dependent safety management (ISO 26262-2:6). Subsections 2.10–2.11 discuss the product development phase subphases in more detail. [25]

Product development at the system level follows the V-model concept given in ISO 26262-4 where the left side of the model has technical safety requirements (TSR), system architecture, system design and implementation and on the right side integration, verification and safety validation. The hardware/software interface (HSI) is specified during this phase and later updated during hardware and software product development phases. [22], [25]

Both product development at the hardware level and product development at the software level rely on the system design specification created in ISO 26262-4. The hardware development process follows the V-model concept given in ISO 26262-5 where hardware requirements and hardware design and implementation are on the left side and hardware integration and verification on the right [23]. The software development process follows the V-model concept given in ISO 26262-6 where specification of software safety requirements, software architectural design and software unit design, and implementation are on the left side and software unit verification, software integration and verification, and testing of the embedded software on the right [27].

Confirmation measures are tools used to judge the functional safety achieved by the item or the contribution to the achievement of functional safety. External measures are measures outside of the boundary of the item that reduce or mitigate the potential hazards resulting from malfunctioning behaviour of the item. External measures include both in-vehicle (e.g., run-flat tires) and off-vehicle (e.g., road crash barriers) measures. Controllability is the ability to avoid a specified harm or damage through timely reactions of the persons involved, possibly supported by external measures. The ability of the persons involved is determined during the HARA process. Other technologies are those different from electrical and electronic technologies (e.g., mechanical or hydraulic). An element realized by another technology may be implemented within the item or specified as an external measure. [25]

Verification of the product development phase includes evaluation of the work products, such as the requirement specification, architectural design, models, or software code

to ensure that they comply with previously established requirements for the concept. The evaluation can be performed in various ways: reviews, simulations, or analysis techniques. The verification process is planned, specified, executed and documented in a systematic manner. [20], [21]

Production, operation, service and decommissioning is a phase that addresses the processes, means and instructions to ensure functional safety regarding production, operation, service and decommissioning of the item or element. The planning of this phase starts during the system level product development process and takes place parallel with the system, hardware, and software development. The planning is enabled through exchanging information or requirements, e.g., safety-related special characteristics or requirements that improve the production ability of the product. The production, operation, service and decommissioning phase is subject to safety management regarding production, operation, service and decommissioning given in ISO 26262-2:7. Subsection 2.12 discusses the production, operation, service and decommissioning phase subphase in more detail. [25]

Verification of the production and operation phase ensures that safety-related special characteristics are appropriately met during production, that safety-related information is appropriately provided in the user manuals and in the repair and maintenance instructions, and that safety-related properties of the item are met by the application of control measures within the production process. Special characteristics include specific process parameters, material characteristics, production tolerances and configuration of elements. [21], [28]

2.7 Item Definition

Item definition is the first subphase of the concept phase. The item definition establishes the definition of the item, including its functionality, interfaces, environmental conditions, legal requirements, and known hazards. [20]

The item definition is to define and describe the item, its functionality, dependencies on, and interaction with, the driver, the environment, and other items at the vehicle level; define and describe the item, its functionality, dependencies on, and interaction with, the driver, the environment, and other items at the vehicle level; and to support an adequate understanding of the item so that the activities in subsequent phases can be

performed. [20]

The item definition describes the requirements of the item. The requirements include legal requirements, national and international standards; the functional behaviour at the vehicle level, including the operating modes or states; where applicable, the required quality, performance, and availability of the functionality; constraints regarding the item, e.g., functional dependencies, dependencies on other items, and the operating environment; potential consequences of behavioural deficiency including known failure modes and hazards; and the capabilities of the actuators, or their assumed capabilities. [20]

In addition to the item's requirements, the item definition has to define the boundaries of the item, its interfaces, and the assumptions concerning its interaction with other items and elements. All this while considering the elements of the item itself, the assumptions concerning the item's behavioural effects on the vehicle, the item's functionality as required by other items and elements, the other items' and elements' functionality used in the item, the allocation and distribution of functions among the systems and elements involved, and the operational situations where the functionality of the item is impacted. [20]

2.8 Hazard Analysis and Risk Assessment

Hazard analysis and risk assessment is the process of identifying and classifying hazardous events of items and specifying safety goals and ASIL related to the prevention or mitigation of the associated hazards. Each hazardous event has a determined safety goal and an ASIL is assigned to that specific safety goal. The HARA is done in order to avoid unreasonable residual risk. Residual risk is the risk remaining after the deployment of safety measures. [1], [20]

There are four identifiable ASIL levels, which are assigned to safety-related hazardous events. Events that are non-safety-related fall under quality management (QM). The QM does not indicate that the corresponding hazardous event has no consequences regarding safety, it simply states that the quality management processes used are sufficient to manage the identified risk. It is stated in the ISO 26262-2 that the organization must have a quality management system (QMS) that supports achieving functional safety and complies with a quality management standard, such as IATF 16949 in conjunction with

ISO 9001, or equivalent [25]. The HARA process determines the severity, exposure, and controllability classes for each hazardous event, and assigns the corresponding ASIL. The analysis criteria for severity is given in Table 3. [20]

Table 3. Severity analysis criteria, classes of severity. Adapted from [20].

	Class			
	S0	S1	S2	S3
Description	No injuries	Light and moderate injuries	Severe and life-threatening injuries (survival probable)	Life-threatening injuries (survival uncertain), fatal injuries

The severity of potential harm is estimated on a four-point scale. The risk assessment determines severity for every person that might be harmed due to the risk, i.e., driver, passengers or other persons in potential risk, such as pedestrians or cyclists. The characteristics of severity can be used to determine the corresponding severity class, e.g., using Abbreviated Injury Scale (AIS). The AIS hosts a six-point ordinal scale to determine the relative severity over nine regions of the body. The AIS definitions are given in Appendix E. [29]

The severity class can be based on a combination of injuries leading to a higher level of severity than that of a single injury. Harm caused by accidents that are not caused by malfunctioning behavior of an item are not considered for the classification of the severity. Severity class S0 may be assigned when malfunction clearly limits to material damage. No ASIL assignment is needed when severity class is assigned to S0, i.e., QM is sufficient. The analysis criteria for exposure is given in Table 4. [20]

Table 4. Exposure analysis criteria, probability of exposure regarding operational situations. Adapted from [20].

	Class				
	E0	E1	E2	E3	E4
Description	Incredible	Very low probability	Low probability	Medium probability	High probability

The probability of exposure is assigned on a five-point scale. The determination of the exposure class is done by reviewing operational situations for the target markets. The exposure to a hazard can be estimated in two ways: based on the duration of a situation or based on the frequency in which the situation is encountered. [20]

The duration of a situation is typically determined through the proportion of time spent in the considered situation compared to the total operation time. The exposure classifications are also dependant on other external factors such as geographical location or type of use. There are no scales for determining the probability of exposure set in stone, so they have to be defined case-dependently. No ASIL assignment is needed when exposure class is assigned to E0, i.e., QM is sufficient. The analysis criteria for controllability is given in Table 5. [20]

Table 5. Controllability analysis criteria, classes of controllability. Adapted from [20].

	Class			
	C0	C1	C2	C3
Description	Controllable in general	Simply controllable	Normally controllable	Difficult to control or uncontrollable

The controllability is assigned on a four-point scale. The controllability is evaluated through the estimate of the probability that the driver or an individual in the immediate vicinity is able to sufficiently gain the control of the hazardous event in a way that the specific harm is avoided. The estimate is based on defined rationale for each hazardous event. Some assumptions of the driver have to be made to rule out operator-related errors, these include that the driver is in appropriate driving condition, has driver's training and complies with the currently applicable legal regulations. [20]

The controllability estimates can also be influenced by external factors, such as target market or driving experience. The controllability evaluations may include potential ways to regain the control. No ASIL assignment is needed when controllability class is assigned to C0, i.e., QM is sufficient. ASIL determination matrix is given in Table 6. [20]

Table 6. ASIL determination matrix. Adapted from [20]

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] If several unlikely scenarios are combined to lower probability of exposure than E1, QM may be argued for this particular S3/C3 combination.

An example of HARA process with detailed class assignments is given in Appendix F.

2.9 Functional Safety Concept

Functional safety concept is the specification of the FSRs, their allocation to elements within the system architecture, and their interaction necessary to achieve the safety goals. The objective of the FSC is to specify the functional behaviour of the item in accordance with its safety goals; specify the constraints regarding suitable and timely detection and control of relevant faults in accordance with its safety goals; specify item level strategies and measures to achieve the required fault tolerance or adequately mitigate the effects of relevant faults by the item itself, by the driver or external measures; allocate the FSRs to the system architectural design, or to external measures; and verify the FSC and specify the safety validation criteria. [1], [20]

To comply with the safety goals, the FSC contains safety measures to be implemented in the item's architectural elements and to be specified in the FSRs. Safety measure is

a activity or technical solution to avoid or control systematic failures and to detect or control random hardware failures, or mitigate their harmful effects. Safety mechanism is a technical solution implemented by the E/E functions or elements, or by other technologies. Safety mechanisms are included in safety measures. The objective of a safety mechanism is to detect and mitigate or tolerate faults. In addition, safety mechanisms are used to control or avoid failures in order to maintain intended functionality and to achieve or maintain a safe state. [1], [20]

One safety goal can lead to a singular FSR or multiple FSRs, and a FSR can consist of multiple safety goals, but each safety goal must have at least one FSR [20]. Figure 11 depicts the flow leading to the creation of FSRs.

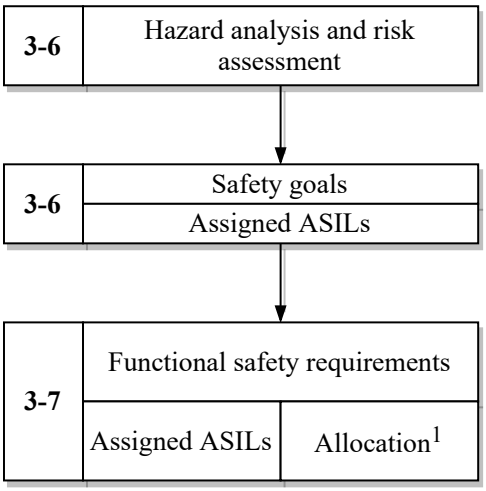


Figure 11. Hierarchy of safety goals and FSRs. [1]May include allocation to system architectural design or external measures. Adapted from [20].

The FSRs are derived from the safety goals while considering preliminary architectural assumptions. Where applicable, each FSR is to be specified by considering the operating modes, FTTI, safe states, EOTI, and functional redundancies (e.g., fault tolerance) [20].

If applicable, the FSRs must specify strategies for fault avoidance; fault detection and control of faults or the resulting malfunctioning behaviour; transition to a safe state, and from a safe state; fault tolerance; driver warnings needed to reduce the risk exposure time to an acceptable duration; driver warnings needed to increase the controllability by the driver; the degradation of the functionality in the presence of a fault and its interaction with either of the previous two points; how timing requirements at the vehicle level

are met; and avoidance or mitigation of a hazardous event due to improper arbitration of multiple control requests generated simultaneously by different functions. Timing requirements could include, i.e., how the FTTI is to be met by defining a FHTI. The process of deriving FSRs from the safety goals may be supported by additional safety analyses, e.g., HAZOP or FTA, in order to develop a complete set of effective FSRs [16]. [20]

When safety goal violation can be prevented by transitioning to, or by maintaining, one or more safe states, then the corresponding safe states are also specified. If a safe state cannot be reached by a transition within an acceptable time interval, and emergency operation must be specified. If there are assumptions made on the drivers, or other persons, necessary actions in order to prevent the violation of a safety goal, then these actions and the means and controls to do so must be specified in the FSC. [20]

The FSRs are allocated to the elements of the system architectural design. During the allocation, the ASIL and the specified strategies are inherited from the associated safety goal. If the FFI between the elements implementing the safety requirements cannot be argued in the system architectural design, then the architectural elements are developed in accordance with the highest ASIL for that specific safety requirements. If the item is compromised of more than one E/E system, then the FSRs for the individual E/E systems and their interfaces are specified, while considering the system architectural design. The corresponding target values for RHFMs can be also specified and allocated to each individual E/E system. Lastly, if ASIL decomposition is applied during the process of allocating the FSRs, then ISO 26262-9:5 must be complied with. [20]

2.10 Technical Safety Concept

Technical safety concept is the specification of the TSRs and their allocation to elements within the system with corresponding system architectural design. The system architectural design provides rationale as to why it is suitable to fulfil safety requirements resulting from activities in the concept phase and design constraints. TSR is a requirement for implementation that is derived from the associated FSRs. TSC is part of the product development phase, whereas FSC is part of the concept phase. [1], [26]

The objective of the TSC is to specify TSRs regarding the functionality, dependencies, constraints and properties of the system elements and interfaces needed for their

implementation; specify TSRs regarding the safety mechanisms to be implemented in the system elements and interfaces; specify requirements regarding the functional safety of the system and its elements during production, operation, service and commissioning; verify that the TSRs are suitable to achieve functional safety at the system level and are consistent with the FSRs; develop a system architectural design and a TSC that satisfy the safety requirements and that are not in conflict with the non-safety-related requirements; analyse the system architectural design in order to prevent faults and to derive the necessary safety-related special characteristics for production and service; and verify that the system architectural design and the TSC are suitable to satisfy the safety requirements according to their respected ASIL. [26]

Figure 12 depicts the flow surrounding the TSC and TSRs, and their assignments to the system architectural design and further to hardware and software requirements.

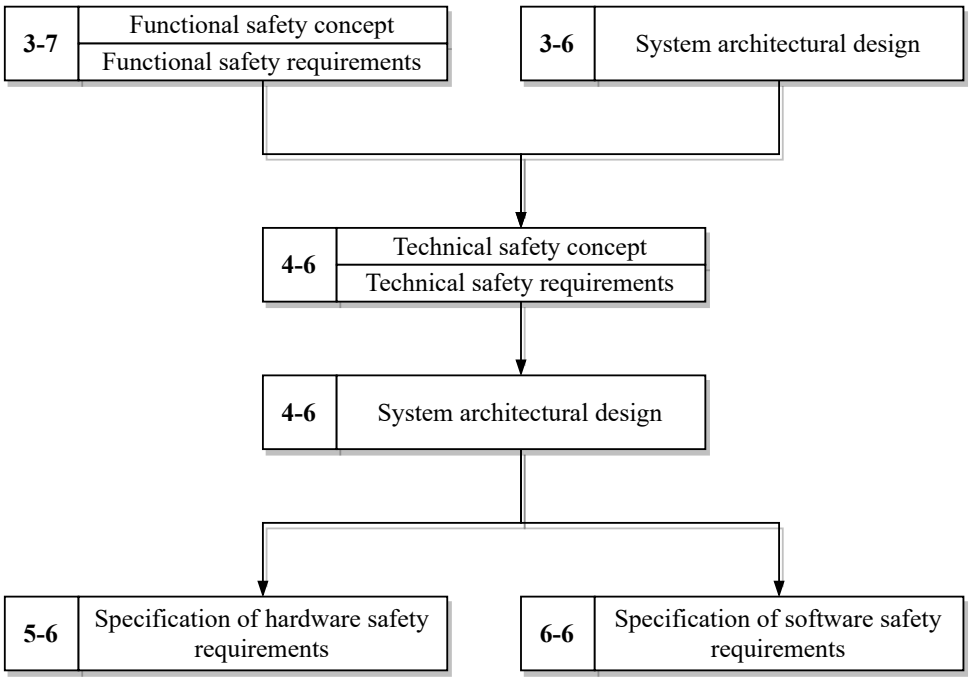


Figure 12. The flow surrounding the TSC, and the allocation of TSRs to the system architectural design and further to software and hardware safety requirements.

TSRs are specified in accordance with the FSC and system architectural design of the item whilst considering the safety-related dependencies and constraints of items, systems and their elements; if applicable, the external interfaces of the system; and the configurability of the system. The TSRs must specify safety mechanisms that detect faults and

prevents or mitigates failures present at the output of the system that violate the FSRs. This includes the safety mechanisms related to the detection, indication and control of faults in the system itself or other external elements that interact with the system; safety mechanisms that contribute to the system achieving or maintaining a safe state of the item; safety mechanisms to define and implement the warning and degradation strategy; and safety mechanisms that prevent faults from being latent. For each safety mechanism that enables an item to achieve or maintain a safe state, has to also specify the transition between states, FHTI with respect to the timing requirements apportioned from the appropriate architectural level, and EOTTI if the safe state of the item cannot be reached within the FTTI. [26]

A safety analysis on the system architectural design must be performed to provide evidence for the suitability of the system design to provide the specified safety-related functions and properties with respect to the ASIL, identify causes of failures and the effects of faults, identify or confirm the safety-related system elements and interfaces, and support the design specification and verify the effectiveness of the safety mechanisms based on identified causes and faults and the effects of failures. The analysis types with corresponding ASIL are given in Table 7. [26]

Table 7. System architectural design analysis methods. Adapted from [26].

Methods	ASIL			
	A	B	C	D
Deductive analysis	o	+	++	++
Inductive analysis	++	++	++	++

The analysis is performed in accordance with ISO 26262:9-8. At this stage of the product development, qualitative analysis is sufficient, quantitative analysis can be performed if deemed necessary. A classic way of going about this is to perform a qualitative FMEA at the system level, design or process level for the inductive analysis and qualitative FTA for the deductive analysis. [24], [26]

The TSRs are allocated to the system architectural design elements with the means of system, hardware, or software as the implementing medium. Each element inherits

the highest ASIL from the TSR that it implements. If TSRs are allocated to custom hardware elements that incorporate programmable behaviour, e.g., a field-programmable gate array (FPGA), an adequate development process combining requirements from the hardware and software development process must be defined and implemented. [26]

A HSI specification that specifies the interaction between hardware and software and is consistent with the TSC is created. The HSI specification includes the component's hardware parts that are controlled by software and hardware resources the support the execution of the software. The characteristics included in the HSI specification are the relevant operating modes of the hardware devices and the relevant configuration parameters, the hardware features that ensure the independence between elements or that support software partitioning, shared and exclusive use of hardware resources, the access mechanism to hardware devices, and the timing constraints derived from the TSC. In addition, the relevant diagnostic capabilities of the hardware, and their use by the software, are specified. The HSI is specified during the system architectural design, and later refined during hardware and software development. The flow surrounding the HSI specification is depicted in Figure 13. [26]

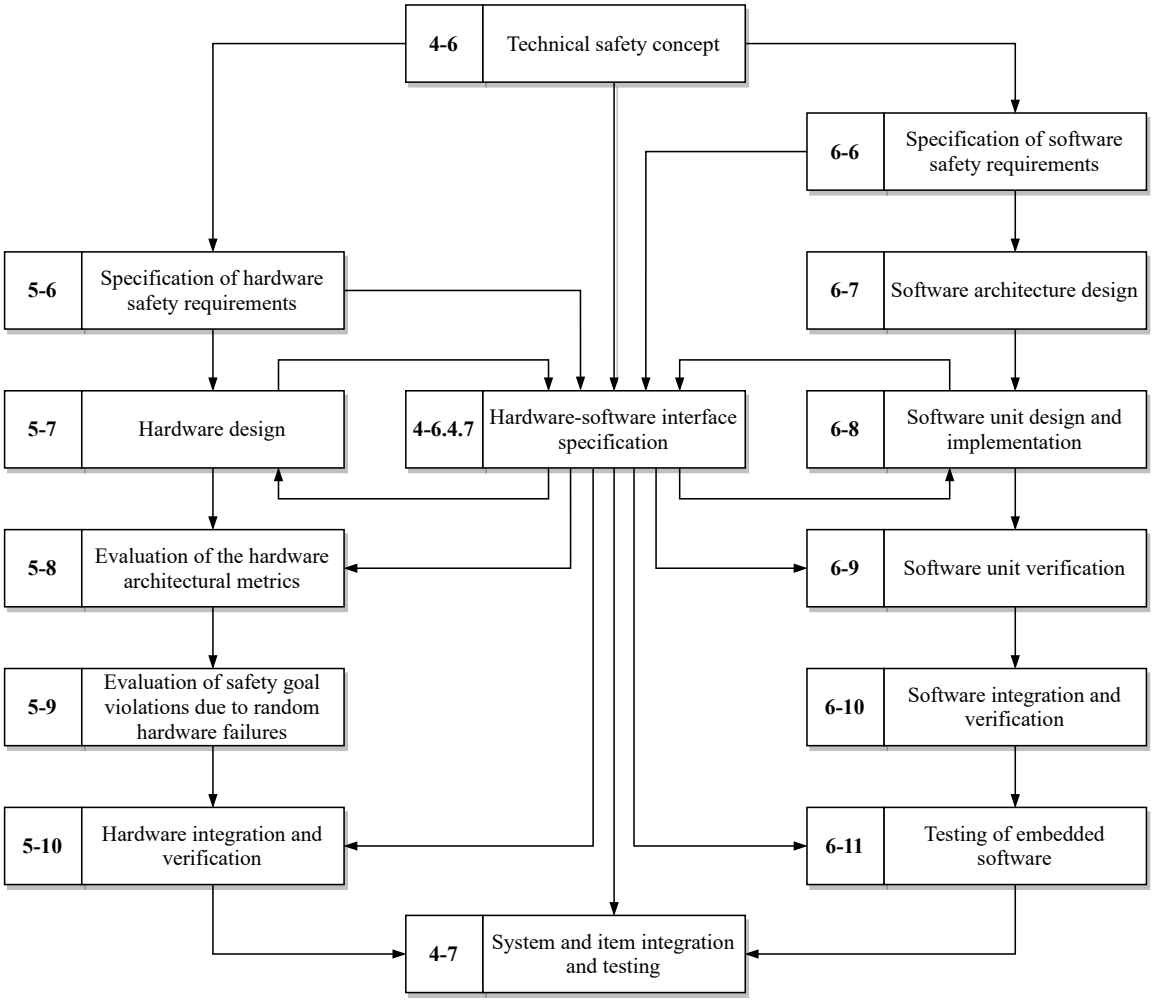


Figure 13. The flow surrounding the HSI specification. Adapted from [26].

Requirements for production, operation, service, and decommissioning are identified and specified during the system architectural design. These include measures required to achieve, maintain or restore the safety-related functions and properties of the item and its elements during production, service or decommissioning; the safety-related special characteristics; the requirements that ensure proper identification of systems or elements; the verification measures for production; the service requirements including diagnostic data and service notes; and measures for decommissioning. [26]

The system architectural design, the TSC, the HSI specification, and the specification of requirements for production, operation, service and decommissioning are verified to provide evidence that the they are suitable and adequate to achieve the required level of functional safety according to the relevant ASIL, consistent between the system architectural design and the TSC, and valid for and compliant with the system architectural

design of prior development steps. The methods of verification are given in Table 8. [26]

Table 8. The methods of verification used. Adapted from [26].

Methods	ASIL			
	A	B	C	D
Inspection	+	++	++	++
Walk-through	++	+	o	o
Simulation	+	+	++	++
System prototyping and vehicle tests	+	+	++	++
System architectural design analyses	See Table 7.			

Inspection and walk-through are used to check that the implementation of the requirements is complete and correct. Simulation, system prototyping, and vehicle tests can be used for fault injection tests to support the argument of completeness and correctness of the system architectural design with respect to faults. [24], [26]

2.11 Product Development at the Software Level

Product development at the software level includes managing both the software development process and the software development environment. The software development process of an item must follow software development processes and use software development environments that are suitable for the development of safety-related embedded software, including methods, guidelines, language, and tools; that support consistency across the subphases of the software development lifecycle and the respective work products; and that are compatible with the system and hardware development phases regarding required interaction and consistency of exchange of information. [27]

The product software development is a parallel process to that of the product hardware development process, and is intertwined through the HSI specification. The safety lifecycle flow from the software development point of view is depicted in Figure 14. [22]

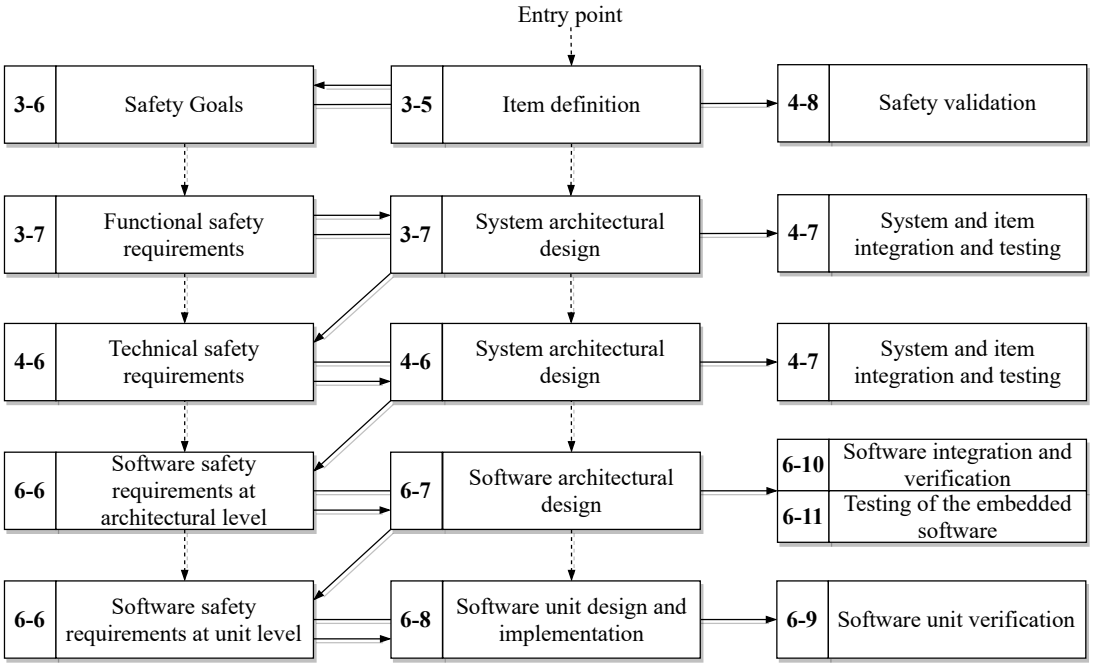


Figure 14. Safety lifecycle flow from concept to working and verified software. Dashed arrows indicate the development flow and others interaction. Adapted from [22], [27].

The process is divided into five horizontal and three vertical levels. The 1st and the 2nd horizontal levels depict the concept phase, while the 3rd horizontal level depicts the product development at system level, finally, the last two horizontal levels (4th and 5th) depict the product development at the software level. There is some overlapping between the horizontal levels in the form of verification and validation. The left-most vertical level is considered as the development phases, the middle one as the design phases and the right-most as the verification phases. Each horizontal level has its test specifications and test cases derived from its requirements, supported by the information on its design. [22], [27]

The software product development is further refined to match the software development V-model. With the V-model incorporated, the reference phase model becomes as given in Figure 15. [27]

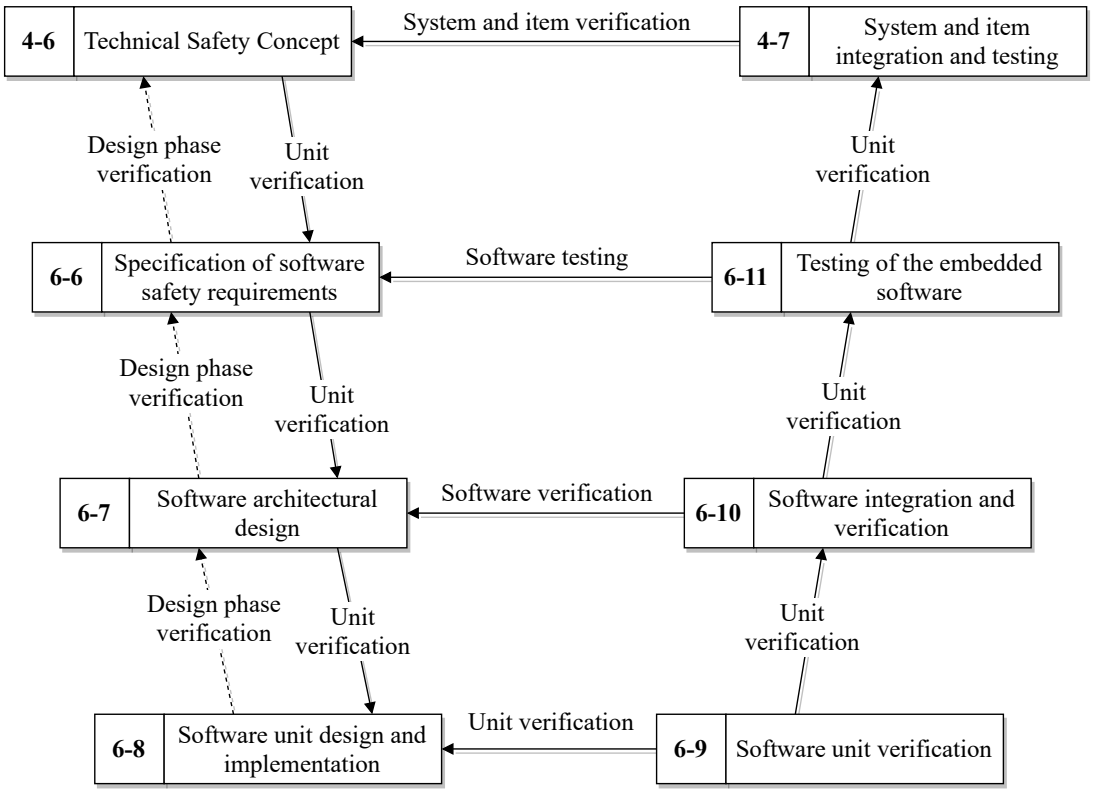


Figure 15. The reference phase model for the development of software according to ISO 26262. Adapted from [27].

The V-model suffers a few drawbacks when compared to other modern software process models. The iterative nature requires a lot of time to create running programs, which could lead to delays in releasing the software [30]. It is also vulnerable to states where the development remains in one state halting the others, e.g., software unit verification state so the other aspects of software development cannot be continued. When the issues presented are regarded, it becomes obvious that the model can handle changing requirements, but can be slow to meet them on the fly.

The ISO 26262 states that the whole extent of the V-model is not inherently mandatory. Other development approaches or methods, e.g., test-driven development (TDD) or CI, from agile software development can also be suitable for the safety-related software development. In order to use the custom-tailored safety activities the ISO26262:2-6.4.5 must be complied with. [27]

The criteria for selecting a design, modelling, or programming language include the ability to unambiguously and comprehensibly define syntax and semantics; suitably specify and manage safety requirements according to ISO 26262-8:6, if modelling is used for

requirements engineering and management; support the achievement of modularity, abstraction, and encapsulation; and support the use of structured constructs. For criterion that is insufficiently addressed by the language itself must be covered by the corresponding guidelines, or by the development environment. The used guidelines could be derived from a pre-existing programming guideline such as the MISRA C [31]. In any case, it has include the topics given in Table 9. [27]

Table 9. Topics to be covered in modelling and coding guidelines. Adapted from [27].

Topics	ASIL			
	A	B	C	D
Enforcement of low complexity	++	++	++	++
Use of language subsets	++	++	++	++
Enforcement of strong typing	++	++	++	++
Use of defensive implementation techniques	+	+	++	++
Use of well-trusted design principles	+	+	++	++
Use of unambiguous graphical representation	+	++	++	++
Use of style guides	+	++	++	++
Use of naming conventions	++	++	++	++
Concurrency aspects	+	+	+	+

Low complexity is often an appropriate compromise between other requirements given in the ISO 26262. The use of language subsets includes the exclusion of ambiguously-defined language constructs which may be interpreted differently by parties involved, exclusion of language constructs which from experience easily lead to mistakes (e.g., identical naming of local and global variables) and the exclusion of language constructs (e.g., if-statements, while-loops and for-loops), which could result in unhandled run-time errors. The enforcement of strong typing promotes the idea the implicit casting should not be done, as the results might be unpredictable. Defensive implementation techniques refer to, e.g., verifying that divisor is not zero in division or detecting errors in switch cases with the use of default keyword. [27]

The entirety of the software development process is subject to the QM. In addition, elements with assigned ASIL are to satisfy the appointed requirements. [27]

2.11.1 Specification of Software Safety Requirements

Specification of software safety requirements is a subphase the discusses the TSRs and their allocation to software safety requirements (SWSR), as well as the creation of a SWSR specification [26]. The SWSRs must be derived by considering the required safety-related functionalities and properties of the software, whose failures could lead to the violation of a TSR allocated to software. SWSRs are not necessarily derived directly from the TSRs, they can also be requirements for software functions and properties that, if not fulfilled, could lead to a violation of the TSRs allocated to software. [27]

This subphase aims to specify or refine the SWSRs that are derived from the FSC and the system architectural design specification, define the safety-related functionalities and properties of the software required for the implementation, refine the requirements of the HSI, and verify that the SWSRs and the HSI requirements are suitable for software development and are consistent with the TSC and the system architectural design specification. [27]

The specification of the SWSRs is derived from the TSRs, the TSC and the system architectural design in accordance with ISO 26262-4:6.4.1 and 6.4.3. It considers the specification and management of safety requirements in accordance with ISO 26262-8:6; the specified system and hardware configurations; the HSI specification; the relevant requirements of the hardware design specification; the timing constraints; the external interfaces; and each operating mode and each transition between the operating modes of the vehicle, the system, or the hardware, having an impact on the software. [27]

If ASIL decomposition is performed to the SWSRs, then the ISO 26262-9:5 must be complied with. If non-safety-related functions are carried out by embedded software, a specification of these functions and their properties in accordance with the applied QM system have to be made available. The HSI specification that was initiated during the product development at the system level is refined to sufficiently allow for the correct control and usage of the hardware by the software, and describe each safety-related dependency between hardware and software. The refined HSI specification is verified jointly

by the persons responsible for the system, hardware and software development. [27]

The SWSRs and refined requirements of the HSI specification must be verified in accordance with ISO 26262-8:6 and 9, to provide evidence for their suitability for the software development, compliance and consistency with the TSRs, compliance with the system design, and consistency with the HSI. [27]

2.11.2 Software Architectural Design

Software architectural design is a subphase that discusses the creation of software architectural design that represents the software architectural elements and their interactions in a hierarchical structure. Both static (e.g., interfaces between software components) and dynamic aspects (e.g., timing behaviour) of the software are described in the design specification. [27]

The subphase aims to develop a software architectural design that satisfies the SWSRs and the other software requirements, verifies that the software architectural design is suitable to satisfy the software safety requirements with the required ASIL, and supports the implementation and verification of the software. SWSRs are considered as safety-related requirements and other software requirements as non-safety-related requirements. The software architectural design provides the means to implement the software requirements and the SWSRs with required ASIL and to manage the complexity of the detailed design and the implementation of the software. [27]

To avoid systematic faults in the software architectural design and in the subsequent development activities, the description of the software architectural design addresses the following characteristics: comprehensibility, consistency, simplicity, verifiability, modularity, abstraction, encapsulation, and maintainability. The addressed characteristics are supported by notations given in Table 10. [27]

Table 10. Notations used in software architectural design. Adapted from [27].

Notations	ASIL			
	A	B	C	D
Natural language	++	++	++	++
Informal notations	++	++	+	+
Semi-formal notations	+	+	++	++
Formal notations	+	+	+	+

Natural language can be used to complement other notation methods where the topics are more readily expressed in natural language or to provide explanation and rationale for decisions captured in the notation. Semi-formal notations can include pseudocode or modelling (with e.g. UML® or Simulink®). If modelling is used then that implies the use of model-based development (MBD). [27]

The development of the software architectural design considers the verifiability of the software architectural design, the suitability for configurable software, the feasibility for the design and implementation of the software units, the testability of the software architecture during software integration testing, and the maintainability of the software architectural design. The verifiability of the software architectural design implies that there must be bi-directional traceability between the software architectural design and the SWSRs. To achieve the presented characteristics (excl. abstraction), a range of principles must be exhibited in the software architectural design. The to-be-applied principles are given in Table 11. [27]

Table 11. Principles applied in order to achieve desired characteristics in the software architectural design. Adapted from [27].

Principle	ASIL			
	A	B	C	D
Appropriate hierarchical structure of the software components	++	++	++	++
Restricted size and complexity of software components	++	++	++	++
Restricted size of interface	+	+	+	++
Strong cohesion within each software component	+	++	++	++
Loose coupling between software components	+	++	++	++
Appropriate scheduling properties	++	++	++	++
Restricted use of interrupts	+	+	+	++
Appropriate spatial isolation of the software components	+	+	+	++
Appropriate management of shared resources	++	++	++	++

Restricted refers to minimize in balance with other design considerations, e.g., minimize the size of interfaces in respect to other valid design considerations. Restricted use of interrupts refers to deterministic use through either minimizing the number of interrupts or using interrupts with clear priorities, or both. Appropriate management of shared resources refers to preventing, detecting, and handling conflicting access to shared resources through safety mechanisms or process measures, or both. Both strong cohesion within each software component and loose coupling between software components can be achieved through separation of concerns, i.e., identifying, encapsulating and manipulating parts of software that are relevant to a particular concept, goal, task, or purpose. Loose coupling between software components also addresses the management of dependencies between software components. [27]

The software architectural design is developed down to the level where software units are identified. SWSRs are hierarchically allocated to the software components down to software units. Each software component is developed in compliance with the highest ASIL of any of the requirements allocated to it. Were pre-existing, non-compliant, and unmodified software architectural elements used, a qualification in accordance with ISO

26262-8:12 must be performed. If embedded software is implemented to software components of different ASILs, or safety-related and non-safety-related software components, then all of the embedded software is to be treated in accordance with the highest ASIL, unless the software components meet the criteria for coexistence in accordance with ISO 26262-9:6. [27]

If software partitioning is used to implement FFI between software components it must ensure that the shared resources are used in such a way that FFI of the software partitions is ensured, the software partitioning is supported by dedicated hardware features or equivalent (only for ASIL D), the element of the software that implements the software partitioning is developed in compliance with the highest ASIL assigned to any requirements of the software partitions, and evidence for the effectiveness of the software partitioning is generated during software integration and verification [27]. The dedicated hardware feature that ensures FFI for software partitioning could be, e.g., having dedicated memory for safety-critical data and another for non-safety-critical data or error correction code (ECC) memory.

A safety-oriented analysis must be carried out at the software architectural level in accordance with ISO 26262-9:8. This is to provide evidence of the suitability of the software to provide specified safety-related functions and properties required by the respective ASIL, identify or confirm the safety-related parts of the software, and support the specification and verify the effectiveness of the safety measures. If the implementation of SWSRs relies on FFI or sufficient independence between software components, then dependent failures and their effects have to be analysed according to ISO 26262-9:7. Depending on the results of the safety-oriented analyses, safety mechanisms for error detection and error handling are applied. An upper estimation of the required resources for the embedded software is made, the estimation includes the execution time, the memory space and the communication resources. [27]

The software architectural design is verified in accordance with ISO 26262-8:9, using the verification methods listed in Table 12. [27]

Table 12. Methods for the verification of the software architectural design. Adapted from [27].

Methods	ASIL			
	A	B	C	D
Walk-through of the design	++	+	o	o
Inspection of the design	+	++	++	++
Simulation of dynamic behaviour of the design	+	+	+	++
Prototype generation	o	o	+	++
Formal verification	o	o	+	+
Control flow analysis	+	+	++	++
Data flow analysis	+	+	++	++
Scheduling analysis	+	+	++	++

Control and data flow analysis can be limited to safety-related components and their interfaces. If MBD is used, then the walk-through and inspection can also be applied to the model. [27]

The objective of the verification is to provide evidence that the software architectural design is suitable to satisfy the software requirements with the required ASIL, the review or investigation of the software architectural design provides evidence for the suitability of the design to satisfy the software requirements with the required ASIL, compatibility with the target environment and adherence to design guidelines. [27]

2.11.3 Software Unit Design and Implementation

Software unit design and implementation is a subphase that discusses the design of the software units and the implementation of them at the source code level. The software unit design is based on the software architectural design. Both SWSRs and non-safety-related requirements are implemented within one development process. [27]

The subphase aims to develop a software unit design in accordance with the software architectural design, the design criteria and the allocated software requirements that supports the implementation and verification of the software unit, and to implement the

software units as specified. [27]

The software unit design and implementation must be suitable to satisfy the software requirements allocated to the software unit with the required ASIL; be consistent with the software architectural design specification; and be consistent with the HSI specification, if the unit has defined behaviour regarding the hardware interface. [27]

To avoid systematic faults and to ensure that the software unit design achieves consistency, comprehensibility, maintainability, and verifiability, the software unit design is described using the same notations and the levels of recommendation as listed prior in Table 10. In the case of MBD being used together with automatic code generation, the notations used to represent the software unit design are applied to the model, which serves as the basis for the code generation. The specification of the software unit design is to describe the functional behaviour and the internal design to the level of detail necessary for their implementation. [27]

The software unit design considers the correct order of execution of subprograms and functions within the software units, based on the software architectural design; consistency of the interfaces between the software units; correctness of data and control flow between and within the software units; simplicity; readability; comprehensibility; robustness; suitability for software modification; and verifiability. To achieve the presented characteristics, a number of design principles are applied. The to-be-applied principles for corresponding ASIL are given in Table 13. [27]

Table 13. Design principles used in the software unit design and implementation. Adapted from [27].

Principle	ASIL			
	A	B	C	D
One entry and one exit point in subprograms and functions	++	++	++	++
No dynamic objects or variables, or else online test during their creation	+	++	++	++
Initialization of variables	++	++	++	++
No multiple use of variable names	++	++	++	++
Avoid global variables or else justify their usage	+	+	++	++
Restricted use of pointers	+	++	++	++
No implicit type conversions	+	++	++	++
No hidden data flow or control flow	+	++	++	++
No unconditional jumps	++	++	++	++
No recursions	+	+	++	++

MISRA C covers multitude of the listed principles [31].

2.11.4 Software Unit Verification

Software unit verification is a subphase that discusses the verification of software unit design and the implemented software units using an appropriate combination of verification methods such as reviews, analyses, and testing. The verification process of a single software unit design considers both SWSRs and non-safety-related software requirements and handles them in one development process. [27]

The subphase aims to provide evidence that the software unit design satisfies the allocated software requirements and is suitable for the implementation, to verify that the defined safety measures resulting from safety-oriented analysis performed in software architectural design subphase (ISO 26262-6:7.4.10 and 7.4.11) are properly implemented, to provide evidence that the implemented software unit complies with the unit design and fulfils the allocated software requirements with the required ASIL, and to provide sufficient evidence that the software unit contains only desired functionalities and properties

regarding functional safety. [27]

The software unit design and the implemented software unit must be verified in accordance with ISO 26262-8:9 by applying an appropriate combination of verification methods. Applying verification methods provides evidence for the compliance with the requirements regarding the unit design and implementation; the compliance of the source code with its design specification; the compliance with the specification of the HSI, if used to implement part of the HSI specification; confidence in the absence of unintended functionality and properties; sufficient resources to support their functionality and properties; and implementation of the safety measures resulting from the safety-oriented analysis performed in software architectural design subphase. The methods for software unit verification are given in Table 14. [27]

Table 14. Methods for software unit verification. Adapted from [27].

Methods	ASIL			
	A	B	C	D
Walk-through	++	+	o	o
Pair-programming	+	+	+	+
Inspection	+	++	++	++
Semi-formal verification	+	+	++	++
Formal verification	o	o	+	+
Control flow analysis	+	+	++	++
Data flow analysis	+	+	++	++
Static code analysis	++	++	++	++
Static analyses based on abstract interpretation	+	+	+	+
Requirements-based test	++	++	++	++
Interface test	++	++	++	++
Fault injection test	+	+	+	++
Resource usage evaluation	+	+	+	++
Back-to-back comparison test between model and code, if applicable	+	+	++	++

For MBD, the walk-through, pair-programming and inspection methods are applied at the model level, if evidence is available that justifies confidence in the used code generator. Static code analysis includes various analysis methods such as searching the source code or the model for patterns matching known faults or compliance with modelling or coding guidelines (e.g., MISRA C). Static analyses based on abstract interpretation are a collection of extended static analysis, they include analysis, such as extending the compiler parse tree by adding semantic information, which can be checked against violation of defined rules or control-flow graph generation and data-flow analysis. The software requirements at the unit level are the basis for the requirements-based testing, these include both the software unit design specification and the SWSRs allocated to the software unit. Interface test can be used to provide evidence for the integrity of used and exchanged data. [27]

Verification methods that include testing require appropriate test cases, the test cases are derived according to the methods given in Table 15.

Table 15. Methods for deriving test cases for software unit testing. Adapted from [27].

Methods	ASIL			
	A	B	C	D
Analysis of requirements	++	++	++	++
Generation and analysis of equivalence classes	+	++	++	++
Analysis of boundary values	+	++	++	++
Error guessing based on knowledge or experience	+	+	+	+

The equivalence classes can be identified based on the division of inputs and outputs, e.g., each unit has their inputs and outputs defined as valid or invalid, so that each equivalence class has representative test values. This type of testing approach is called equivalence class testing (ECT). Boundary values includes approaching, crossing, and out of range values. [27]

The coverage of requirements at the software unit level is determined and the structural coverage is measured. These two coverage metrics seek to evaluate the completeness of verification and provide evidence that the objectives for unit testing are adequately

achieved. If achieved structural coverage is considered insufficient, either additional test cases are specified or a rationale based on other methods is provided. The structural coverage metrics with their corresponding ASIL are given in Table 16. [27]

Table 16. Structural coverage methods at the software unit level. Adapted from [27].

Methods	ASIL			
	A	B	C	D
Statement coverage	++	++	+	+
Branch coverage	+	++	++	++
Modified condition/decision coverage (MC/DC)	+	+	+	++

The test environment for software unit testing must be suitable for achieving the objectives of the unit testing considering the target environment. If the software unit testing is not carried out in the target environment, the differences in the source and object code, as well as the differences between the test environment and the target environment, have to be analysed in order to specify additional tests in the target environment during the subsequent test phases. [27]

2.11.5 Software Integration and Verification

Software integration and verification is a subphase that discusses the verification of the particular integration levels and the interfaces between the software elements according to the software architectural design. The steps of the integration and verification of the software elements are related to the hierarchical architecture of the software. The embedded software can consist of both safety-related software elements and non-safety-related software elements. [27]

The subphase aims to define the integration steps and integrate the software elements until the embedded software is fully integrated, to verify that the defined safety measures resulting from safety analyses at the software architectural level are properly implemented, to provide evidence that the integrated software units and software components fulfil their requirements according to the software architectural design, and to provide sufficient evidence that the integrated software contains neither undesired functionalities nor undesired

properties regarding functional safety. [27]

The approach of the software integration is defined. It describes the steps of integrating of individual software units hierarchically into software components until the embedded software is fully integrated. The integration and test strategy specification must consider the dependencies required to achieve the verification objectives, the functional dependencies that are relevant for software integration, and the dependencies between the software integration and the hardware/software integration. [27]

The software integration is verified according to ISO 26262-8:9 by applying an appropriate combination of verification methods. Applying verification methods provides evidence for the hierarchically integrated software units, the software components and the integrated embedded software achieving compliance with the software architectural design, compliance with the HSI specification, the specified functionality, the specified properties, sufficient resources to support the functionality, and the effectiveness of the safety measures resulting from the safety-oriented analysis in software architectural design subphase (ISO 26262-6:7.4.10 and 7.4.11). The methods for verification of the software integration are given in Table 17. [27]

Table 17. Software integration verification methods. Adapted from [27].

Methods	ASIL			
	A	B	C	D
Requirements-based test	++	++	++	++
Interface test	++	++	++	++
Fault injection test	+	+	++	++
Resource usage evaluation	++	++	++	++
Back-to-back comparison test between model and code, if applicable	+	+	++	++
Verification of the control flow and data flow	+	+	++	++
Static code analysis	++	++	++	++
Static analyses based on abstract interpretation	+	+	+	+

Fault injection tests in the context of software integration testing means to inject faults in particular to test the correctness of the HSI related to the safety mechanisms. The fault injection testing can be used to verify the FFI. Resource usage evaluation is done in order to ensure the fulfilment of requirements influenced by the hardware architectural design with sufficient tolerance, properties such as average processor performance, minimum and maximum execution times, memory usage, and the bandwidth of communication links have to be determined. Some of the resource usage evaluation can only be performed on the target environment, or on an emulator that adequately supports resource usage tests. [27]

Software integration test cases are derived identically to that of software unit testing. The methods for deriving test cases for software unit testing are presented in Table 15. The coverage requirements at the software architectural level are determined to evaluate the completeness of verification and to provide evidence that the test objectives for integration testing are adequately achieved. If the achieved structural coverage is considered insufficient, either additional test cases are specified or a rationale based on other methods is provided. Structural coverage is evaluated using the methods given in Table 18. [27]

Table 18. Structural coverage methods at the software architectural level. Adapted from [27].

Methods	ASIL			
	A	B	C	D
Function coverage	+	+	++	++
Call coverage	+	+	++	++

Function coverage refers to the percentage of executed software subprograms or functions in the software. Call coverage refers to the percentage of executed software subprograms or functions with respect to each implemented call of these subprograms or functions in the software. The inclusion of embedded software as a part of a production release in accordance with ISO 26262-2:6.4.13 is to be verified to contain all of the specified functions and properties and only contain other unspecified functions if these functions do not impair the compliance with the SWSRs. [27]

The test environment for software integration testing must be suitable for achieving the objectives of the integration testing considering the target environment. If the software integration testing is not carried out in the target environment, the differences in the source and object code, as well as the differences between test environment and the target environment, have to be analysed in order to specify additional tests in the target environment during the subsequent test phases. [27]

2.11.6 Embedded Software Testing

The purpose of embedded software testing is to provide evidence that the embedded software fulfils its requirements in the target environment. The evidence is provided by using suitable results of other previously presented verification activities. [27]

The subphase aims to provide evidence that the embedded software fulfils the safety-related requirements when executed in the target environment, and contains only desired functionalities and properties regarding functional safety. [27]

The embedded software testing must be conducted in suitable test environment to verify that it meets the SWSRs in the target environment. The suitable test environments are given in Table 19. [27]

Table 19. Test environments for conducting embedded software testing. Adapted from [27].

Environments	ASIL			
	A	B	C	D
Hardware-in-the-loop	++	++	++	++
ECU network environments	++	++	++	++
Vehicles	+	+	++	++

ECU network environments include test benches that partially or fully integrate the electrical systems of a vehicle, e.g., mule vehicles, among others. [27]

The testing of embedded software must be performed using the methods given in Table 20 to provide evidence that the embedded software fulfils the software requirements as required by their respective ASIL. [27]

Table 20. Methods for testing of embedded software. Adapted from [27].

Methods	ASIL			
	A	B	C	D
Requirements-based test	++	++	++	++
Fault injection test	+	+	+	++

Fault injection test in the context of embedded testing refers to introducing faults by, e.g., corrupting calibration parameters. The test cases for embedded testing are derived using the methods given in Table 21. [27]

Table 21. Methods for deriving embedded test cases. Adapted from [27].

Methods	ASIL			
	A	B	C	D
Analysis of requirements	++	++	++	++
Generation and analysis of equivalence classes	+	++	++	++
Analysis of boundary values	+	+	++	++
Error guessing based on knowledge or experience	+	+	++	++
Analysis of functional dependencies	+	+	++	++
Analysis of operational use cases	+	++	++	++

Operational use cases could be, e.g., software update in the field or safety-related behaviour of the embedded software in different operational modes such as start-up, diagnosis, degraded, power-down, power-up, and calibration. The results of the embedded software testing are evaluated with regard to compliance with the expected results and the coverage of the SWSRs. [27]

2.12 Production, Operation, Service, and Decommissioning

Production, operation, service and decommissioning is the last phase in the safety lifecycle. The phase addresses the processes, means and instructions to ensure functional safety

regarding production, operation, service, and decommissioning. Planning of this phase is started in the product development phase. [28]

The objectives of this phase is to further plan and implement safety-related content of both production and operation, service, and decommissioning. The planning includes specifications, such as, production plan, production control plan, service plan, service instructions, user information, decommissioning instructions, and rescue services instructions. In addition, producibility requirements specification and operation, service, decommissioning requirements specification are created. [28]

Production ensures that functional safety is achieved during the production phase and after release for production by the entity responsible for the production process of the items and elements. The production implements and maintains the processes and control measures specified in the production and production control plans. Reports are created according to the implemented control measures and production process capability. [28]

Operation, service and decommissioning ensures that functional safety is achieved during the operation, service (i.e., maintenance and repair) and decommissioning subphases of the vehicle lifecycle. The operation, service, and decommissioning phase provides the requirements on how the execution and monitoring of operation, service, and decommissioning is done, while taking the safety-related special characteristics of the item into account. The phase introduces field monitoring process that aims to provide and analyse field data to detect the presence of functional safety issues, and trigger actions to address them. The decommissioning subphase is further divided to three parts: pre-disassembling, disassembling, and lastly, post-disassembling. Post-disassembling is not in the scope of the standard. As a result, the implemented field monitoring process is recorded into field observation instructions. [28]

3 Software Quality

Software quality is a term that has various definitions depending on the target environment and audience. From the process point of view, the ISO/IEC 25000 series states that software quality is the capability of the software product to satisfy stated and implied needs when used under specified conditions, whereas, the IEEE 730 states that software quality is the degree to which a software product meets established requirements [32].

The software quality can be divided into three distinct groups: software process quality, software structural quality, and software functional quality. The software quality is made up from quality characteristics, more namely, compatibility, usability, security, functional sustainability, performance efficiency, reliability, maintainability, and portability. Each of the characteristic has a number of subcharacteristics, these are depicted in Figure 16 as well as their relation to the overall software product quality. [33], [34]

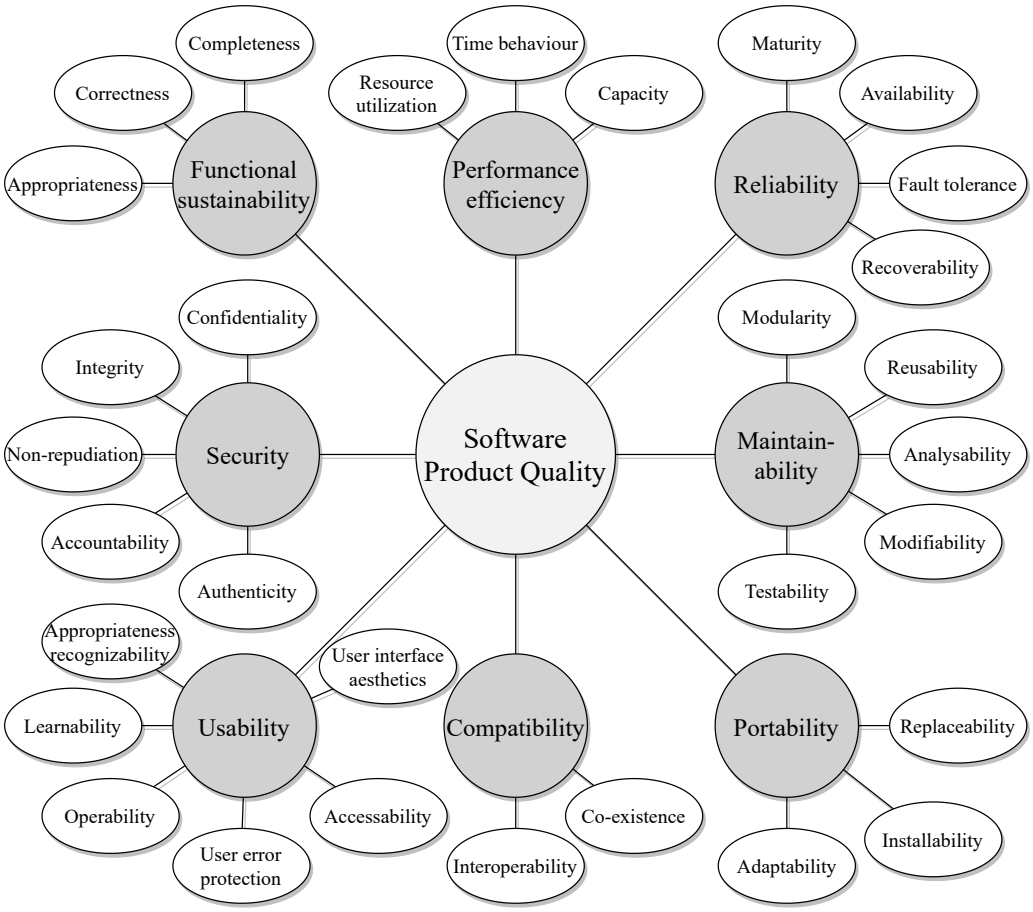


Figure 16. Software product quality as described in the ISO/IEC 25010.

The characteristics and subcharacteristics are used to measure the quality of software in its target environment [34].

The software quality is achieved through different activities during the software quality management (SQM) process.

3.1 Software Process Quality

Software process quality measures how well a process can satisfy stated and implied needs when used in a specified context. The software process quality envelops the entire software development process. Software process quality is controlled through the SQM process. The tools used in the SQM process are specified in a QM. The QMS is often based on quality management standards, such as the ISO 9000 series or its automotive variant, IATF 16949. [32]

The SQM is divided in to three main activities: software quality assurance (SQA), software quality planning, and software quality control. Software quality control is often depicted as a SQA subactivity, but here, it is addressed as a separate activity for the sake of clarity. An exemplary SQM process is depicted in Figure 17. [35]

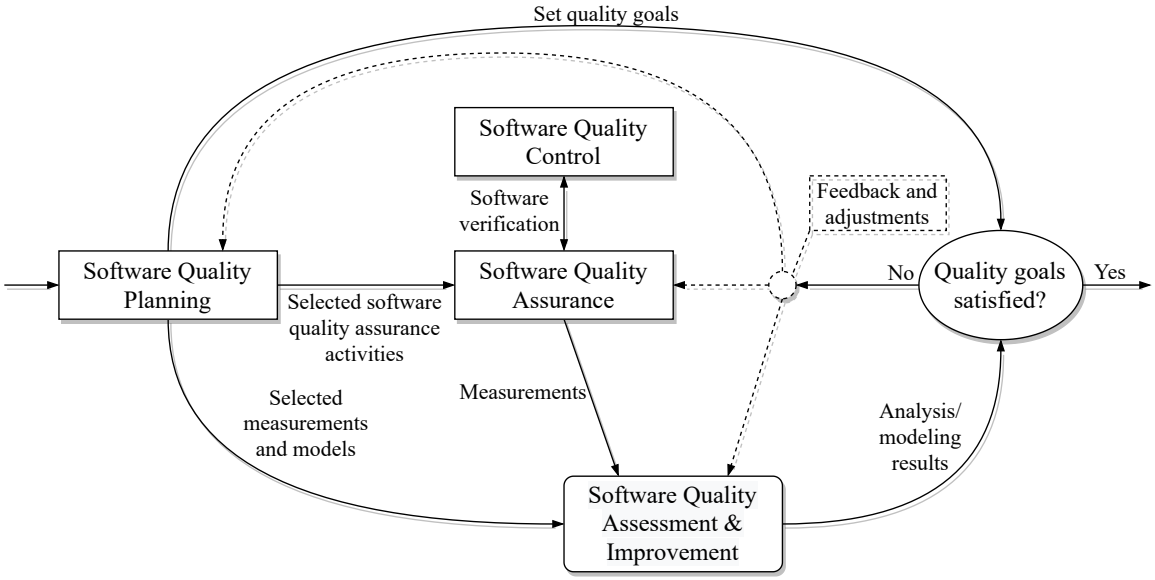


Figure 17. Example of software quality management process. Adapted from [35].

Software quality planning is the first performed of the three. Software quality planning sets

the quality goals and forms a strategy for the overall SQA process, including selected SQA activities and measurements. The quality goals are identified and quantified in respect to the customer's requests. At this point, the quality goals should be considered on a larger scale, which are then partitioned to functional and structural software qualities. The goals are used as an indicator for finished product that meets the customer's requirements for both functionality and quality. [35]

SQA is a practice that seeks to minimize remaining defects, if any, in the released software product. SQA is the essence of the SQM, it acts as an intermediate between planning, control, and assessment, while monitoring the used processes and methods. It is also the place where procedures and standards are recognized and implemented to the process. The results yielded from the software quality assessment and improvement process are then compared to set quality goals, if the results do not meet the goals, a round of feedback and adjustments is performed. [35]

Software quality control performs the verification of the software throughout the development phases. It ensures that processes and standards implemented by the SQA are followed. Software quality control seeks to remove defects from the artifacts of the software development process, whereas SQA seeks to remove defects from the standards' and processes' point of view. Verification is done through dynamic analysis, e.g., testing, and static analysis, e.g., code inspection and code reviews. [35]

Dynamic analysis refers to analysis done during runtime, whereas static analysis is done on static objects, e.g., the source code. Both dynamic analysis and static analysis are tightly thread to functional and structural quality of the software.

3.2 Software Structural Quality

Software structural quality measures how well the software source code is structured. The structural requirements are derived from the functional requirements, which are then allocated to software architectural design. In a sense, the structural quality is that of the software architectural design and its implementation. Structural quality is often identified with knowledge of the internal workings of the software, i.e. white-box. Structural quality can be measured through both dynamic and static analysis, e.g., unit testing against software unit design or static code analysis against predefined coding guidelines, e.g.,

MISRA C. [32], [35]

3.3 Software Functional Quality

Software functional quality measures how well the software conforms to the design based on the functional requirements or specifications. Functional quality is often identified without knowledge of the internal workings of the software, i.e., black-box. Functional quality is mostly measured using dynamic analysis methods. [32], [35]

3.4 Software Testing

Software testing can be performed on both the functional and the structural aspects of software. It can be reduced to three distinct parts: planning and preparation, execution, and analysis. An example of generic testing process is depicted in Figure 18. [35]

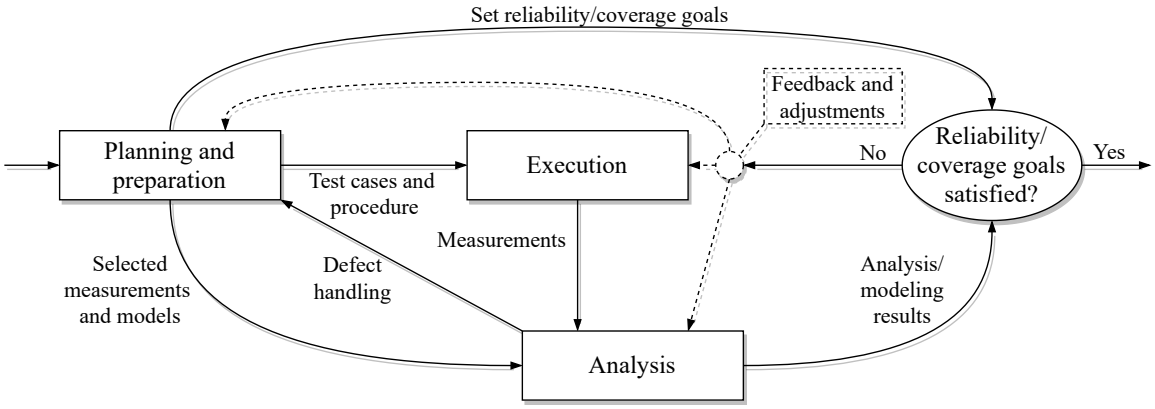


Figure 18. An example of generic testing process. Adapted from [35].

Planning and preparation activity withholds a number of sub-activities regarding pre-execution. Activities such as setting the goals for reliability and coverage goals, information gathering, and creating test cases and procedures, are performed [35]. As in software process quality, the reliability and coverage goals are set to determine the amount of testing that is required to meet a tolerable level of defects left in the final product. Planning and preparation is the cornerstone of systematic testing (cf. ad hoc testing).

Execution receives its instructions from the planning and preparation activity, i.e., the test cases and test procedures to follow. The activity takes measurements from the test artifacts or the device under test (DUT) and serves them to the analysis activity. [35]

Analysis activity receives its instructions on what to check and analyse from the planning and preparation activity. Measurement data is received from the execution activity. The measurement data is being checked for failure in the test run, if failure is met, then either the implementation of the test cases and procedures is improperly done or the actual instruction on their implementation is lacking or unsuitable. The results yielded from the analysis activity are then compared to those set by the planning and preparation activity, if the results are unsatisfactory, a round of feedback and adjustments is performed. Feedback and adjustments could include, e.g., implementation of more test cases to have higher coverage. [35]

The testing process presented in Figure 18 may be implemented on various levels and techniques mentioned in the software verification required by the ISO 26262 [27]. For the purpose of this section, the included software testing levels are unit testing, integration testing, system testing, and acceptance testing. The considered testing techniques, or types, include requirements-based testing, fault injection testing, interface testing, and regression testing. Most of these levels and techniques are considered in the ISO 26262.

The software testing process is started with the most atomic software elements that are available. Figure 19 describes the software testing direction in terms of software testing levels. [30]

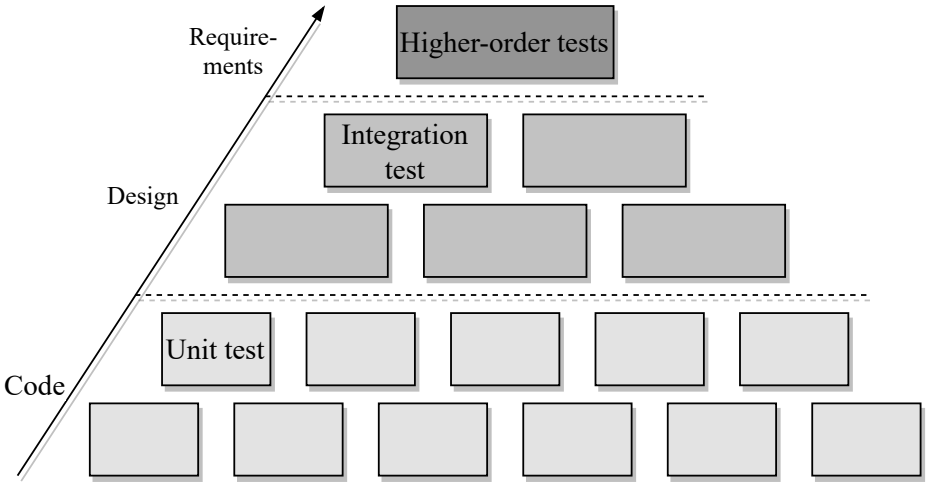


Figure 19. Software testing direction. Adapted from [30].

Higher-order tests are done to ensure the created software works also in combination with other system elements, such as, hardware or people. Higher-order tests usually include

system testing and acceptance testing (also known as validation testing), but often have context-specific additional tests, e.g., performance testing or stress testing. [30]

How the generic testing process relates to presented test levels and test techniques is depicted in Figure 20.

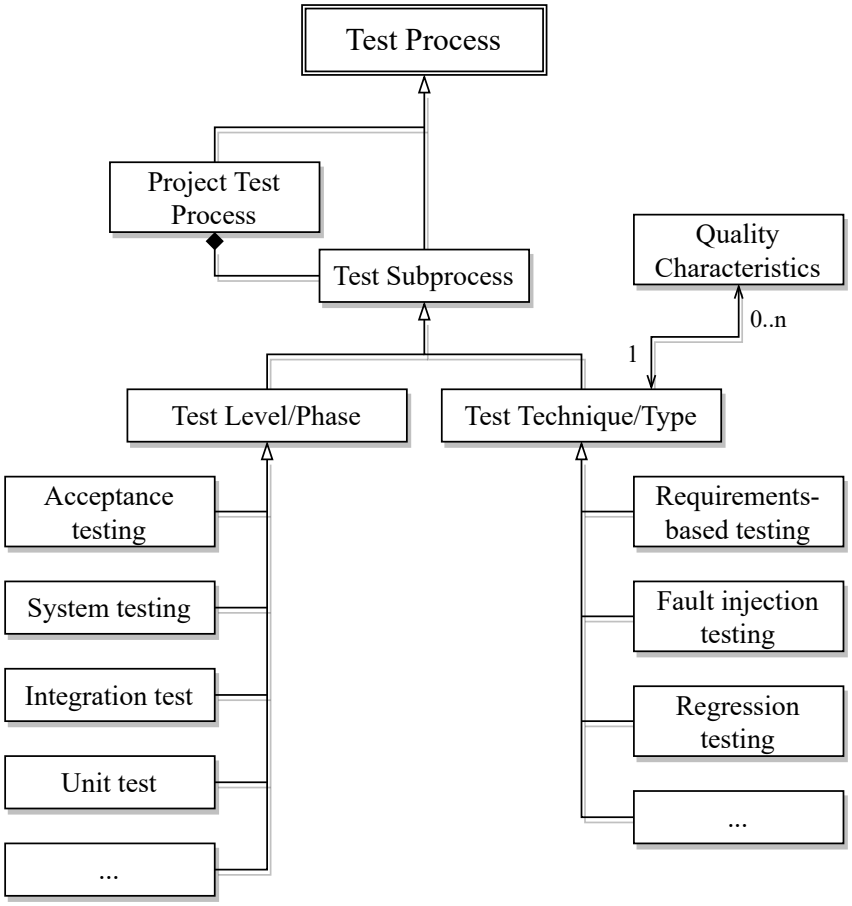


Figure 20. The relationship of the generic testing process, test levels and test techniques. Properties are inherited in upwards direction. Adapted from [36].

The project test process composes of several test subprocesses, e.g., unit testing and integration testing. The quality characteristics depicted are the same ones as presented previously in Figure 16. Test techniques are used to measure certain quality characteristics, e.g., fault injection testing measures reliability characteristics and requirements-based testing measures functional sustainability characteristics. [36]

3.4.1 Unit Testing

Unit testing tests one software unit at a time. Depending on the context, i.e., environment, language, and architectural design, units may have varying compositions. Units are considered as such when they can be isolated to perform standalone testing, i.e., static methods cannot form units on their own, since they can only be called from their limited scope [32].

Unit tests are limited to the scope of the unit that is being tested. Within its boundaries, the internal processing logic and data structures are tested. Internal processing logic includes the unit's interfaces and both error-handling paths and independent paths. If an unit has external dependencies to other units, these have to be simulated using test doubles. [30]

A number of test doubles exist, most notoriously – mocks, stubs and fakes. Mocks are implementations of specific methods that can be preprogrammed to meet the tester's expectations, mocks record the times they are called [37]. Stubs are used to provide predefined answers to function calls, they have the same name and interface as the method they are stubbing, they do not record data. Fakes are minimal working implementations of the actual functionality, they offer limited capabilities to that of the faked method. [38]

3.4.2 Integration Testing

Integration testing tests the functionality of multiple units together, it focuses on the interface and the interaction between the units. The integration of multiple units can have varying approaches, e.g., big-bang integration, top-down integration, and bottom-up integration. The big-bang integration integrates all of the units in to a working system, without any incremental integration. Top-down integration goes through the units incrementally from the unit that has hierarchically most control to the one with the least. The unit where the integration is started is the main program. Bottom-up integration focuses on the smallest atomic units and creates clusters of them. Each cluster is tested and added to the program structure when found working. When the units are fully integrated the integration testing becomes system testing. [30]

3.4.3 System Testing

System testing verifies that the implementation is valid in respect to the system requirements [33]. Execution of fully integrated software is done with test cases verifying that the system requirements are delivered, i.e., the system or program meets its original objectives [39].

The system testing is often considered from the customer's perspective, this means that the software is tested in its target environment. The testing is considered to be done using black-box testing, since the customer's perspective does not include inner workings of the source code. In addition, the testing often adopts characteristics from usage-based testing, which focuses on the available interface, i.e., testing of the user interfaces to establish their validity. [35]

3.4.4 Acceptance Testing

Acceptance testing is usually performed at the end of the testing process to determine the releasability of the product. It is often done by, or in collaboration with, the customer. Acceptance testing is similar to system testing in terms of the used test environment. Due to the similarities in test environment it can be fully or partially performed amidst the system testing process. [35]

Where system testing focuses on fulfilling the given system requirements, acceptance testing evaluates whether the software solution actually works for the user. The acceptance testing process may include other testing, such as, stress testing or performance testing. [36]

3.4.5 Requirements-Based Testing

Requirements-based testing (RBT) is technique that turns requirements in to test cases. RBT can be used in any testing phase where requirements and their allocation is available. RBT is especially advantageous technique to implement on heterogeneous software, i.e., software that is released on multiple platforms, or processors and cores [40]. Figure 21 depicts the conventional approach to RBT from the software testing perspective. [41]

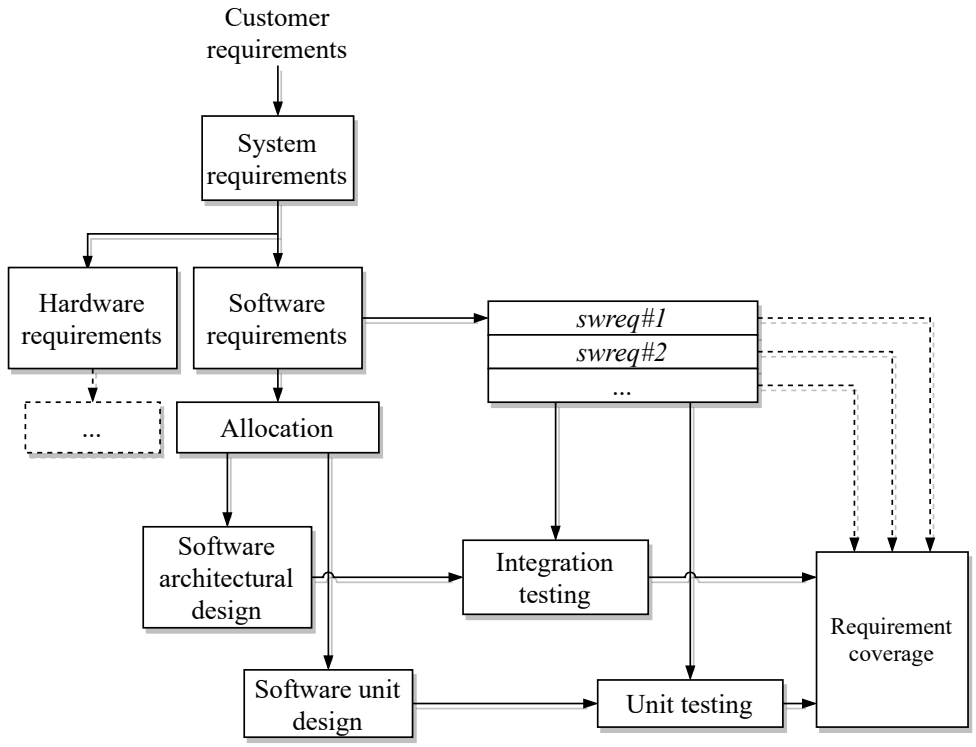


Figure 21. Conventional flow of requirements-based testing from the software testing perspective.

Requirement coverage indicates the percentage of requirements being covered by tests, where often more the better.

RBT can be used to measure software product quality characteristics, such as, the correctness, the completeness and the appropriateness of the system. In some cases where the system requirements specify quality characteristics, e.g., time behaviour, then these can be measured as well.

3.4.6 Fault Injection Testing

Fault injection testing (FIT) is a technique used to analyze the systems behaviour when a fault occurs. The FIT assesses the reliability and dependability of the software. Dependability is a combinatory measure of reliability characteristics and maintainability characteristics. The dependability can also measure other characteristics and omit others, e.g., not all of the reliability subcharacteristics are accounted for or other characteristics, such as security, are analyzed [42]. Figure 22 depicts the basic concept of fault injection testing and system observation. [43]

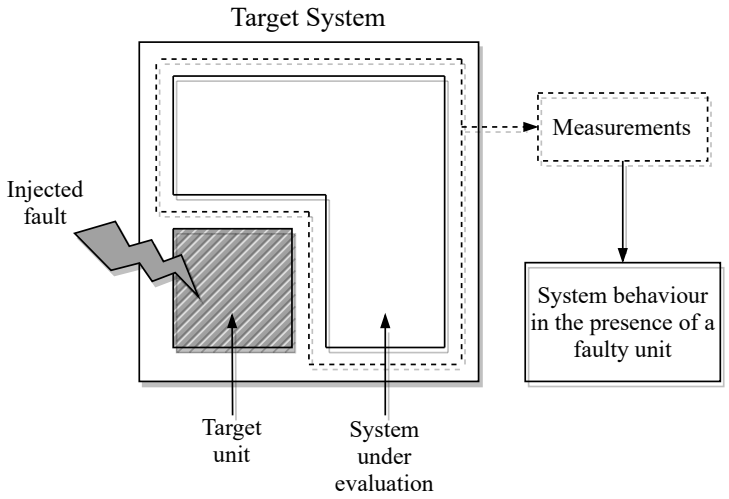


Figure 22. The concept of fault injection and system observation. Adapted from [43].

The rest of the system is measured and analysed to see what kind of behaviour the target system exhibits. The system behaviour can then be cross-referenced to the requested behaviour to see whether additional measures have to be taken to ensure that satisfactory properties, e.g., FFI, are achieved.

FIT can be used to measure software product quality characteristics, such as the fault tolerance and recoverability of the system. Characteristics such as the testability can be partially measured, since the ability to inject faults is a measure of testability in itself, i.e., testability is a biproduct of the FIT.

3.4.7 Interface Testing

Interface testing is a testing technique that is used to evaluate whether systems or units pass data and control correctly to one another. It can be implemented on lower-order testing, such as unit testing and integration testing, as well as on system testing. [32]

Interface testing, or usability testing, can be performed on the user interface level as well as on the unit interface level. User interface level testing measures software product quality characteristics, such as the user interface aesthetics, the operability, and the user error protection of the system, whereas unit interface level testing measures, e.g., interoperability. [44]

3.4.8 Regression Testing

Regression testing is a technique that tests whether the code executes as prior when changes are done or new modules are added to the system integration, i.e., verifies modifications have not caused unintended side-effects and that the system or unit still complies with its original requirements. Regression testing tries to ensure that the changes made do not introduce neither unintended behavior nor defects in the unchanged parts of the system. Regression testing is often executed through automated builds and continuous integration. Continuous integration refers to the integration of source code from several sources to a single software project. [36]

In its most elementary state, regression testing measures the correctness of the software product. The concept of regression testing can be expanded to measure various characteristics. For example, resource utilization subcharacteristic can be regressively tested by confirming central processing unit (CPU) load between builds. [45]

4 Integration Testing in Danfoss Editron Converters

This section focuses on the future Danfoss Editron electric power converters and their development. The test case examples are performed on a work in progress electric power converter that is equipped with the functionality to run electric machines with varying control types, e.g., torque control and speed control. A concept illustration for the future power electronic converters is depicted in Figure 23.

Figure 23. A concept illustration of the future electric power converters.

The converters are isolated in two voltage regions; low voltage (LV) region and high voltage (HV) region. The LV-side is fed with a power supply unit (PSU) that outputs 24 volts. The LV-side has various peripherals, such as the drivers for the external peripherals and an Aurix™ TriCore™ CPU. The HV-side has six semiconductor switches and six freewheel diodes. A technical visualization of the converters is given in Figure 24.

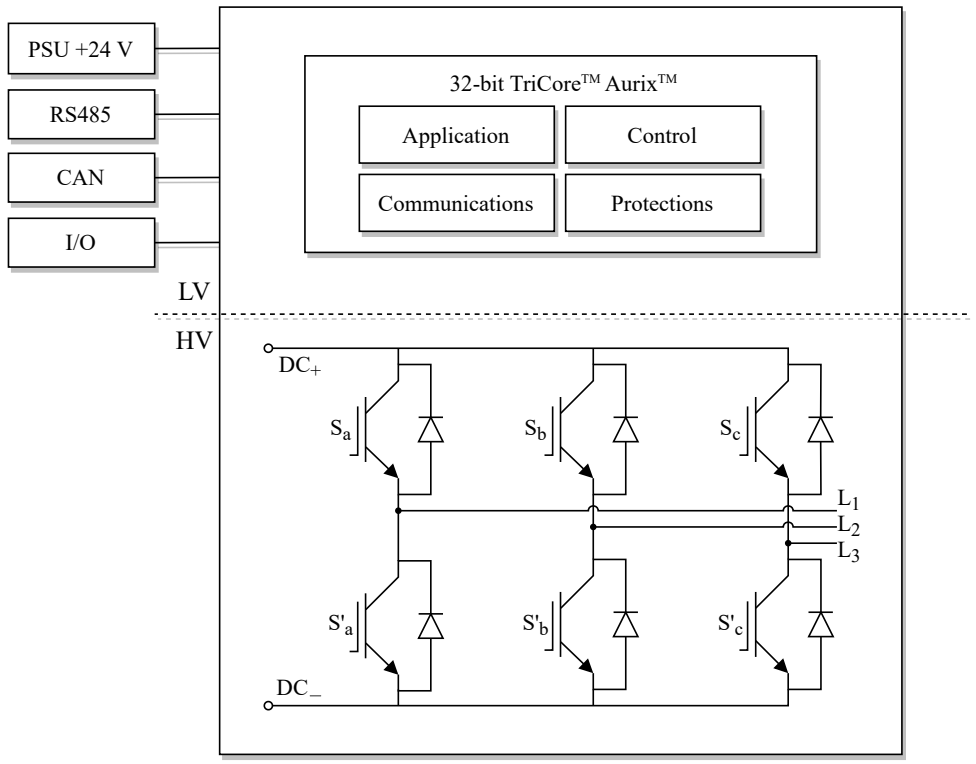


Figure 24. A technical visualization of the future electric power converters.

The converters can communicate through CAN with CANopen and SAE-J1939 protocols, as well as through physical input/output-module (I/O) and PDP serial bus. PDP is an proprietary diagnostic protocol with no relation to the packet data protocol.

The integration testing of said converters is done because the ISO 26262 requires verification of the software integration. Software verification and integration was discussed in section 2.11.5, where most prominently RBT, FIT, and interface testing stood out as the most recommended methods of verification. RBT and FIT are used to accompany the integration testing.

4.1 Robot Framework

Robot Framework is a keyword-driven automation framework for acceptance testing, acceptance test driven development, behaviour-driven development (BDD), test automation, and RPA. Robot Framework can deliver automation solutions from web applications all the way to embedded software applications. Its core framework is implemented using Python, but it also runs on various other Python implementations, such as IronPython (.NET) and Jython (JVM). Being an open source framework, Robot Framework has

amassed rich community around it that contributes to libraries that extend the core functionality of the test framework. [46], [47]

Robot Framework has an easy plain text syntax, that utilizes human-readable keywords, i.e., uses keyword-driven testing (KDT) methodology. The tests are build using Robot scripts and Python scripts. Keywords can be built in either script type, although Python offers more hands-on tools for the developer, whereas Robot scripts are better for assembling keywords from pre-existing ones. The similarity between Robot script and Python language does not go unnoticed, this is further supported by the fact that the Robot Framework is mostly written in Python (63.4 %) [48]. [47]

The Robot Framework interacts with the DUT through libraries and device drivers, which in the case of an electric power converter with CAN connection this means device drivers for the CAN adapter and library to fulfill the used CAN protocol. The development of the framework, e.g., libraries and drivers, and the creation of tests can be allocated to different personnel. The test creation uses the abstracted keywords to create the tests.

Robot Framework was chosen as the integration testing framework due to being internally used in the company. Another point was there had been no encountered problems to date.

4.1.1 Robot Scripting Language

Robot script is written to two types of files: *.robot* files and *.resource* files. Files with the *.robot* extension can be either test suite files or resource files, whereas files with *.resource* extension can only be resource files. Robot script is split to tokens, i.e., keywords and arguments, with either space separated or pipe separated format. In space separated format each token has to have at least 2 spaces between tokens, whereas pipe separated format requires spaces around the pipe as well as on the start and end of each row. [47]

The structure inside Robot test suite and resource files uses following tables

- settings,
- variables,
- test cases,
- keywords, and

- comments.

Settings table includes needed files, such as resources, variable lists, libraries, sets suite level and test level setups and teardowns, and documents the test suite. Resources are files that define keywords that can be used in multiple test suites. Variable lists are collections of variables that help declutter the test suites, supported file extensions include, e.g., *.py*, *.class*, *.robot*, and *.resource*. Variable files can also be set globally with `--variablelist` command line argument. Libraries are collections of keywords and variables, the types of libraries are: external libraries (e.g., SeleniumLibrary, SSHLibrary), standard libraries (e.g., BuiltIn, Collections), and user-defined Python and Java classes [46]. Setup is a pre-run keyword that sets each suite or case up, they are used to reduce boilerplate code and enable systematic testing. Teardown is a post-run keyword that sets the DUT back to default as prior to running the test or suite. The available keywords in the settings table are given in Table 22. [47]

Table 22. Available keywords in settings table. [47]

Keyword	Description
Library	Used to import libraries, this includes user-defined, standard, and external libraries.
Resource	Used to take resource files into use in that specific file. Resource files can be chained together, i.e., resource files included in another files are passed on.
Variables	Used to include variable list files.
Documentation	Used to create documentation for that test suite or resource file.
Metadata	Used similarly to documentation, creates entry to log and report. Can be used to record, e.g., software version and time of running.
Suite Setup	Setup that is executed before running any of the suite's test cases.
Suite Teardown	Teardown that is executed after running all of the suite's test cases.
Test Setup	Setup executed before running each test case.
Test Teardown	Teardown executed after running each test case.
Force Tags	Used to force tags on each of the suite's test cases.
Default Tags	Used to set default tags on each of the suite's test cases. If any of the suite's test cases has its own tags, then these are cleared and those are used instead.

Variables table is used to define test suite level variables and variable lists locally, e.g.

```
${pi}        3.14
@{fibonacci}  1  1  2  3  5
```

defines variable `${pi}` with value `3.14` and list `@{fibonacci}` with values `1`, `1`, `2`, `3`, and `5`. Variables can be imported using the variables keyword in settings table.

Test cases table is used to define new test cases and keywords unique to them. Test cases can be build from available keywords, which can be imported from test libraries or resource files, or created in the keywords table. Test cases table is not used in resource files. Table 23 gives the available keywords in the test cases table.

Table 23. Available keywords in test cases table. [47]

Keyword	Description
[Documentation]	Used to create documentation for that test case.
[Tags]	Used to tag the specific test case. If forced tags are defined, then these are added to those. If default tags are defined, they are overridden with these.
[Setup]	Used to specify setup for the specific test case.
[Teardown]	Used to specify teardown for the specific test case.
[Template]	Used to specify template keyword.
[Timeout]	Used to specify test case timeout.

Keywords table is used to define new keywords. The keyword table syntax is almost identical to that of test cases, excluding some available keywords. Like test cases, keywords can be build from available keywords. Table 24 gives the available keywords in the keywords table.

Table 24. Available keywords in keywords table. [47]

Keyword	Description
[Documentation]	Used to create documentation for that keyword.
[Tags]	Used to tag the specific keyword. If forced tags are defined, then these are added to those. If default tags are defined, they are overridden with these.
[Arguments]	Used to specify the arguments for the keyword.
[Return]	Used to specify return value for the keyword
[Teardown]	Used to specify teardown for the keyword.
[Timeout]	Used to specify keyword timeout.

Comments table is used for internal communication inside the test suites and resource files. Everything under the comments table is omitted by Robot Framework.

Robot Framework has a number of built-in tools:

- Rebot - generates logs and reports based on XML outputs and combines multiple outputs together,
- Libdoc - generates keyword documentation for libraries and resource files,
- Testdoc - generates high level HTML documentation based on Robot Framework test cases, and
- Tidy - cleaning and format changing of Robot Framework data files.

There is also a vast number of external tools that are developed separate of the Robot Framework, e.g., Pabot and Robot Tools. [47]

4.1.2 Robot Showcase

The following showcase seeks to give intermediate level guidance on how testing can be performed with Robot Framework using Python. The showcasing is conducted through an example test suite. The example expects that all of the source files are inside the directory where the tests are ran from. The showcase is produced with Robot Framework version 4.0.2 in Windows environment.

A resource file `example.resource` is created to showcase how resource files can be used in test suites. The source code of `example.resource` is given in Listing 1.

Listing 1. `example.resource` source code.

```
*** Settings ***
Library     OperatingSystem

*** Keywords ***
Check If Exists     [Arguments]     ${path}
    Directory Should Exist          ${path}
```

The `example.resource` uses `OperatingSystem` standard library and its `Directory Should Exist` keyword to evaluate whether directory exists in the argument path. The `Check If Exists` keyword is used as a wrapper for the functionality.

A library `Confirmation` is created with Python, it situates in `confirmation.py`. The source code of the `confirmation.py` is given in Listing 2.

Listing 2. `confirmation.py` source code.

```
from robot.api.deco import keyword, library
from robot.libraries.BuiltIn import BuiltIn
import os

@library
class Confirmation(object):
    def __init__(self):
        pass

    @keyword
    def words_are_palindrome(self, word1, word2):
        """
        Checks whether given words are palindrome of each other.
        """
        are_palindrome = word1 == word2[::-1]
        BuiltIn().log(are_palindrome)
        return are_palindrome
```

`confirmation.py` implements a library that has a keyword. The library is designated with the `@library` decorator and the keyword with the `@keyword` decorator. The keyword can be accessed with the function name. The calling of keywords is case- and space-insensitive, i.e., `words_are_palindrome` can be accessed with both `Words Are Palindrome` and `words_are_palindrome`, and any combination in between. The `BuiltIn().log()` uses exactly same interface as `Log` when implemented as Robot script.

Finally, the test suite file `example.robot` where `example.resource` and `confirmation.py` are used, is created. The source code of `example.robot` is given in Listing 3.

Listing 3. `example.robot` source code.

```
*** Settings ***
Documentation   An example test suite to describe
...             the syntax of Robot scripting.
Resource        example.resource
Library         confirmation.Confirmation

Force Tags      showcase_test

*** Variables ***
${message}      Hello
${path}         C:\\Program Files

*** Test Cases ***
Local and Global Variables Test
    [Documentation]         Demonstrate the use of local
    ...                     and global variables.
    [Tags]                  showcase_test_1
    Log                     ${EXTERNAL_STR}
    Should Be Equal         ${message}          Hello
    Should Not Be Equal     ${EXTERNAL_STR}     ${message}

Python Library Test
    [Documentation]      Demonstrate the use of Python keywords.
    [Tags]               showcase_test_2
    FOR   ${word}   IN   ${message}   World
        ${result}=  Words Are Palindrome  ${EXTERNAL_STR}  ${word}
        IF  ${result} == ${False}
            Fail  Words ${EXTERNAL_STR} and ${word} are not
   palindrome.
        END
    END

Resources and Template Test
    [Documentation]     Demonstrate the use of resource files,
    ...                 and [Template] keyword.
    [Tags]              showcase_test_3
    [Template]          Folder In ${path} Should Exist
        ${path}
        C:\\Users
        C:\\Windows\\System32

*** Keywords ***
Folder In ${path} Should Exist
    Check If Exists     ${path}
```

The resource `example.resource` is taken into use with `Resource` keyword and the library `confirmation.py` with the `Library` keyword. In `confirmation.Confirmation`, `confirmation` refers to the filename and `Confirmation` to the class inside the file. `Force Tags` keyword is used to apply `example_test` tag to each of the test cases. Local variables `${message}` and `${path}` are initialized with demonstrative values (it is common practice to type local

variables in lowercase and global variables in uppercase).

`Local and Global Variables Test` is a test case that showcases few `BuiltIn` library keywords. `Log` keyword adds entry to the generated log file. `Should Be Equal` and `Should Not Be Equal` check whether the two arguments are equal (`arg1 is arg2`) and not equal (`arg1 is not arg2`), respectively.

`Python Library Test` showcases the use of Python library and its keywords, as well as built-in keywords `IN`, `FOR` and `Fail`. `FOR` keyword works similarly to for-loops in Python, e.g.

```
FOR ${loop_variable} IN @{values_to_iterate}
    Log  ${loop_variable}
END
```

In the example's case, a list containing `${message}` and `World` is iterated with `${word}` as the iterator. Each element in the list is compared to a global variable `${EXTERNAL_STR}` using `Words Are Palindrome` keyword which was created in the `Confirmation` library. The result of those comparisons is stored to `${result}` variable, which is then evaluated in an if-expression. `IF` keyword works similarly to the if-expression in Python, e.g.

```
IF ${number} < ${greater_number}
    Pass Execution
END
```

In the example's case, if one of the argument pairs is not a palindrome pair, then the test case fails with the built-in keyword `Fail` and the argument pair with descriptive message is logged.

`Resources and Template Test` showcases the use of resource files, `[Template]` keyword, and keywords with embedded arguments. The `[Template]` keyword enables a data-driven approach where arguments can be given as columns. The keyword `Folder in ${path} Should Exist` uses the `Check If Exists` keyword to check whether that path exist on the host. `Check If Exists` is made available by including `example.resource` in the settings table. Each argument line is evaluated independently, i.e.

```
Folder In ${path} Should Exist
    Check If Exists  ${path}
        Directory Should Exist  ${path}

Folder In C:\\Users Should Exist
    Check If Exists  C:\\Users
        Directory Should Exist  C:\\Users

Folder In C:\\Windows\\System32 Should Exist
    Check If Exists  C:\\Windows\\System32
        Directory Should Exist  C:\\Windows\\System32
```

If any of the directories does not exist, the test case fails.

The execution of the test suite is done with the `robot` command. The command has to be accompanied with arguments to add locations for the Python files and the test suite file. In addition, global variable is created that is required by the `example.robot` test suite. The execution is configured with following command line options

```
robot --pythonpath . --variable EXTERNAL_STR:olleH example.robot
```

The tests are then executed in top-down order since `--randomize` option is not used. The report and log files generated from the test execution are given in Appendix G.

4.2 Test Architecture

There are number ways to approach the implementation of the test architecture. The approach depends strongly on how the test cases are to be implemented; following keyword-driven, data-driven or behavior-driven style [47].

The test case implementation affects the execution of the test suites greatly. Robot Framework offers several solutions on how the test suites and cases can be differentiated to create test runs with limited scopes, e.g., platform-specific, requirement-specific, and feature-specific. The ways of achieving pipelines that run on limited scopes include strategies, such as tagging, naming, clustering, the use of constructor resources, and identical variable naming. Identical variable naming enables fetching the one in scope with `--resource` command line option.

The implemented test cases use both keyword-driven and data-driven styles. Data-driven style is used when testing communication and keyword-driven style when testing functionality. Figure 25 depicts how the test architecture is being approached.

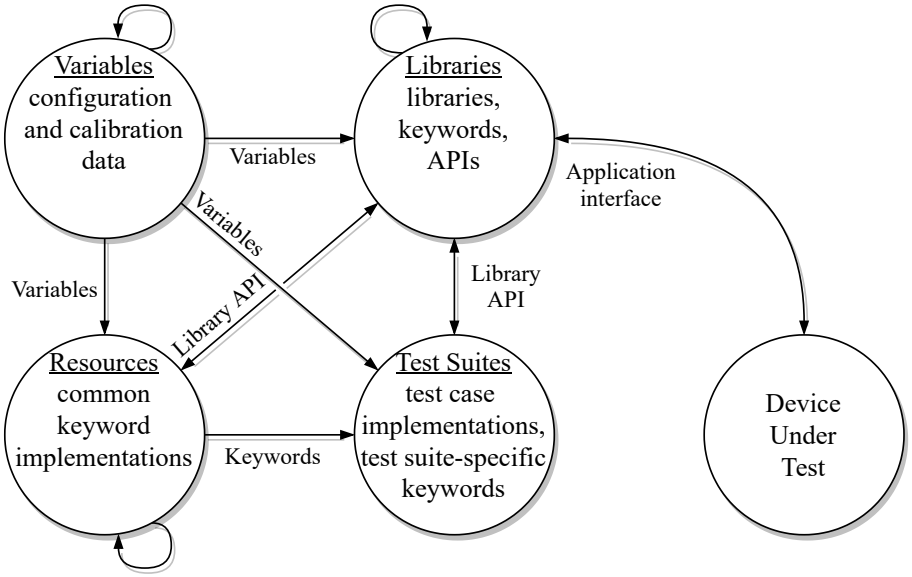


Figure 25. Vision for test architecture.

Variables consists of variable lists, they are included in any of the other architectural element, i.e., libraries, resources, and test suites, or by the variable list itself. Variables are used for the inclusion of necessary DUT resources, this is achieved by including the corresponding constructor resource path with command line option, i.e.

```
robot --variable DUT:dsr/simulator.resource
```

This sets the global variable `${DUT}` to match the tested DUT, i.e., simulator in this case. The appointed `simulator.resource` is then used to include the correct resources, variables, and libraries, it works as a resource cluster.

Resources are used for resource clustering and creating common resources that can be used in several applications or platforms. Resource clustering refers to creating constructor resources that gather needed resources for tidier including. Libraries are used to implement complex keywords and application interfaces for use in the test suites. The application interfaces are implemented to support as much interchangeability as possible, e.g., singular keyword to communicate the same parameter through CANopen and SAE-J1939 protocols. Test suites gather the needed resources, variables, and libraries in order to execute corresponding test cases.

Test repository structure is not as crucial as in other test frameworks, this is due to the modularity and customizability of the Robot Framework [47]. Still, the structure

should possess characteristics such as modularity and partition to create easily navigable, aesthetically pleasing, and non-overlapping structure. The non-overlapping structure refers to dividing the repository in to sections, or subrepositories, that specific teams or people access. Figure 26 depicts the vision for the used repository structure in pseudo-visualization.

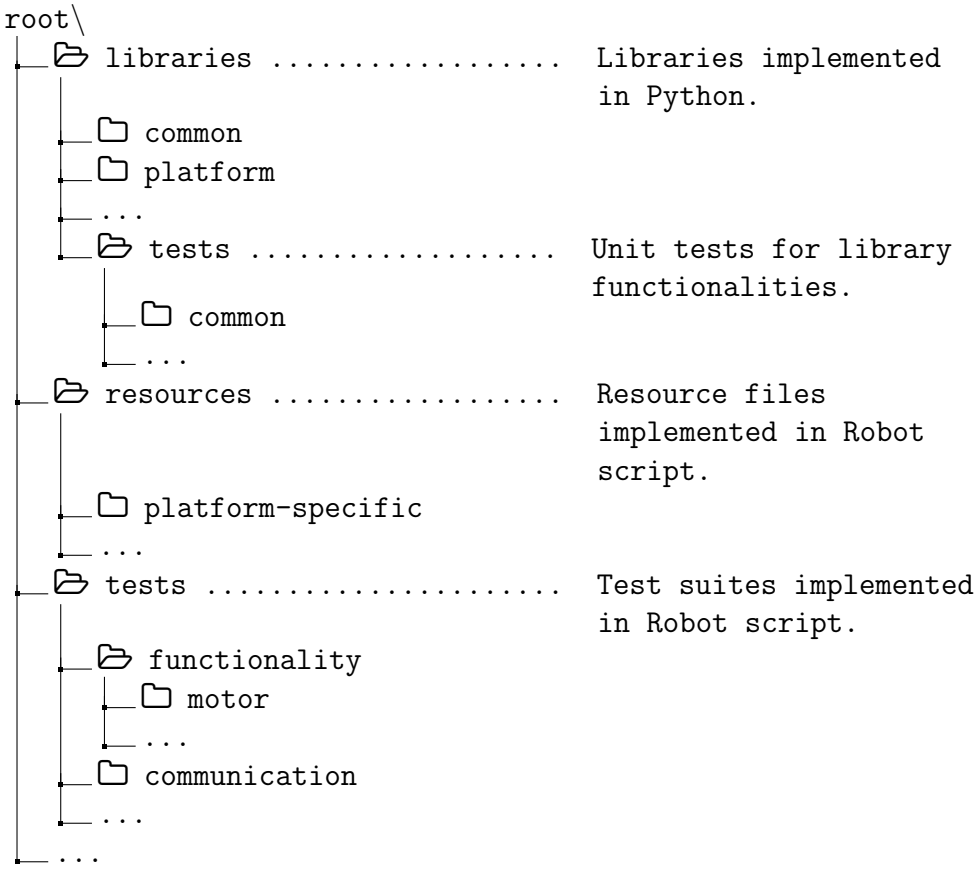


Figure 26. Repository structure as pseudo-visualization.

The running of specific test suites and test cases is done by using the pattern matching Robot Framework offers. For example, following patterns can be used to include test suites

```
robot motor_*.robot          # * matches any string,
                                 including empty string
robot feature_1?0.robot      # ? matches any single character
robot req_[1-2]??            # [a-z] matches any character in range
```

where `motor_*.robot` tries to match every test suite that starts with the string "*motor_*", `feature_1?0.robot` tries to match any feature test suite with identification number in range 100, 110, ..., 190, and `req_[1-2]??` tries to match any requirement in range 100,

101, ..., 299. The `?`-operator matches also letters, whereas the `[a-z]`-operator only matches the range, which is in the example's case, 1...2. [47]

Including and excluding tags uses tag patterns. The same wildcard patterns are accepted as in test cases names. In addition, operators `AND`, `OR`, and `NOT` can be used. E.g., following patterns can be used to select specific test tags

```
robot --include featureANDtargetNOTj1939
robot --include target --exclude j1939
```

where `--include featureANDtargetNOTj1939` tries to match test cases containing both `feature` and `target` tags but not the `j1939` tag and `--include target --exclude j1939` tries to include test cases containing `target` tag and exclude ones with the `j1939` tag. None of the patterns after the first `NOT` must not match, i.e.

```
robot --include targetNOTfeatureNOTj1939
```

where match containing `target` tag, but not containing `feature` tag or `j1939` tag is sought. Operators need to be given in uppercase, whereas tags can be given case-insensitively. [47]

4.3 Test Implementation

Performing integration testing in the electric power converters has various challenges that need to be surpassed, this includes challenges like several software releases, platforms, communication protocols, and the ISO 26262 requirements. Integration testing in both the simulator and the actual converter is enabled by dividing the corresponding libraries and resources in to specific clusters, which are then required from the host who executes the tests. In addition to the platform-specific libraries, the actual converter requires some knowledge about the used peripherals, such as the CAN adapter and the port that connects the diagnostic cable to the converter.

Since the software is deployed on an embedded platform, the integration of the software is restricted, i.e., tests need to be ran in nearly or fully integrated system state. The integration can be verified by running the tests on either the simulator or the converters. In the subsequent sections, the testing is done using the simulator. Figure 25 depicts how the Robot Framework is coupled with the converter and gives insight on how the different firmware layers are organized.

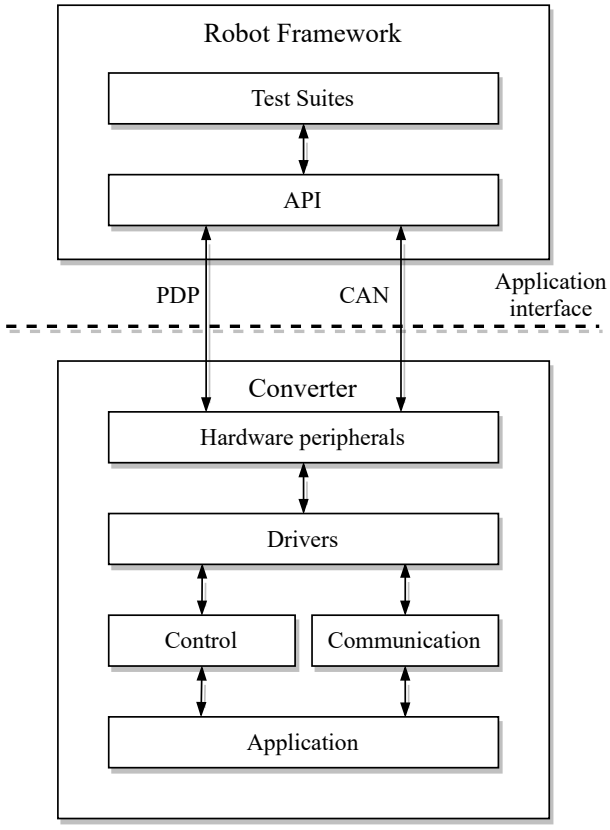


Figure 27. Coupling of the converter and the Robot Framework.

The API block has Python libraries for CAN protocols that use the CAN adapter manufacturer drivers, as well as a PDP library for the diagnostic communication. Hardware peripherals block includes a RS-485 serial port and I/O for connecting the CAN high and low wires. The drivers block converts the CAN and serial data in to readable form, which is then passed to control and communication. Communication block decides what is listened to and what is not, in addition, it sends back status signals from the application. The application block contains the actual functionality of the converter. Control block communicates with both the application to receive its commands and provide statutes and the drivers to pass on the command signals for the semiconductor switches.

One of the main requirements set by the ISO 26262 is the bi-directional traceability. It practically means that the design and the requirements can be traced back to each other. In the case of the integration testing, the traceability is established between the test suites and SWSRs by tagging the test cases with corresponding requirement identification number. The test cases are derived from the requirements and intended functionality.

The requirement coverage can be evaluated by using external tools. This includes ex-

tracting the requirement identification numbers from the application development system of choosing and from the test reports. The identification numbers and test report tags are then matches to determine how many of the requirements have been covered with tests.

The running of the tests using CI is enabled by separating the platform-specific and communication-specific components to be included as variables. This is done by separating the run source independencies to batch scripts, shell scripts and YAML pipelines, and rest of the dependencies to command line arguments. The run source-specific configuration, such as compilers and build directories are auto-generated to match any local environment, and the interchangeable parts, such as the simulator, are defined explicitly in the execution call.

In test suites where test data is available and give insight to the tested functionality, graphical representation is used. The graphical representations, e.g., line graphs, are a way to implement regression testing, this is especially important for the control logic of the converters. The regression testing is still manual, but in the foreseeable future, an attempt on baseline comparison system is a high priority.

4.4 Example of Requirements-Based Testing

RBT is one of the most highly recommended technique of integration verification mentioned in the ISO 26262 [27]. It is considered to be the basis for verifying software, since the ISO 26262 is a requirement-heavy standard.

One of the hazards related to electric vehicles is unintended torque and the excessive acceleration (HZ001) and deacceleration (HZ002) caused by it. When the unintended torque is analysed using the HARA process described in section 2.8, the safety goal (SG001) is identified and ASIL C is assigned to the hazards.

According to the FSC, which was discussed in section 2.9, the safety goal is further processed to specify functional behaviour required to fulfil the safety goal. The derived FSR (FSR001) is further refined to have safety measures that prevents the unintended excessive acceleration, this includes a speed limiter feature. The purpose of the speed limiter is to prevent excessive acceleration after parametrized speed limit is surpassed by reducing the used torque reference and cause an overspeed fault when parametrized

overspeed speed threshold is exceeded. The overspeed fault leads to a safe state when the overspeed condition is met, i.e., surpassing the parametrized overspeed threshold. The parametrized threshold to signal overspeed fault is higher than that of the speed limiter threshold. The FTTI for the overspeed fault to happen is 0.1 s. The FSR is assigned with a safe state where the modulation is stopped altogether.

Section 2.10 discusses the TSC. According to the section, the corresponding FSR is processed and specified as a TSR. The TSR (TSR001) becomes the requirement on how the FSR is to be implemented in order to fulfil the safety requirement and prevent other failures from violating the FSR. In addition, the TSR specifies the functionality of the speed limiter to constantly monitor the speed and when the speed limit is being breached to warn the user and reduce the used torque reference. The overspeed fault is allocated to other elements of the software architecture to ensure independence between the components. The detection time FDTI for the overspeed fault is 0.125 ms and the reaction time FRTI is 0.125 ms as well, which is the time level they run on. This means that the fault handling FHTI is 0.25 ms, which is substantially smaller than the FTTI.

The portion of the TSRs that are to be implemented by software are then specified as SWSRs according to section 2.11.1. The corresponding speed limiter FSR becomes SWSR (SWSR001). The SWSR is then allocated to the software architectural design and further in to unit design according to sections 2.11.2 and 2.11.3. The discussed practical flow is depicted in Figure 28.

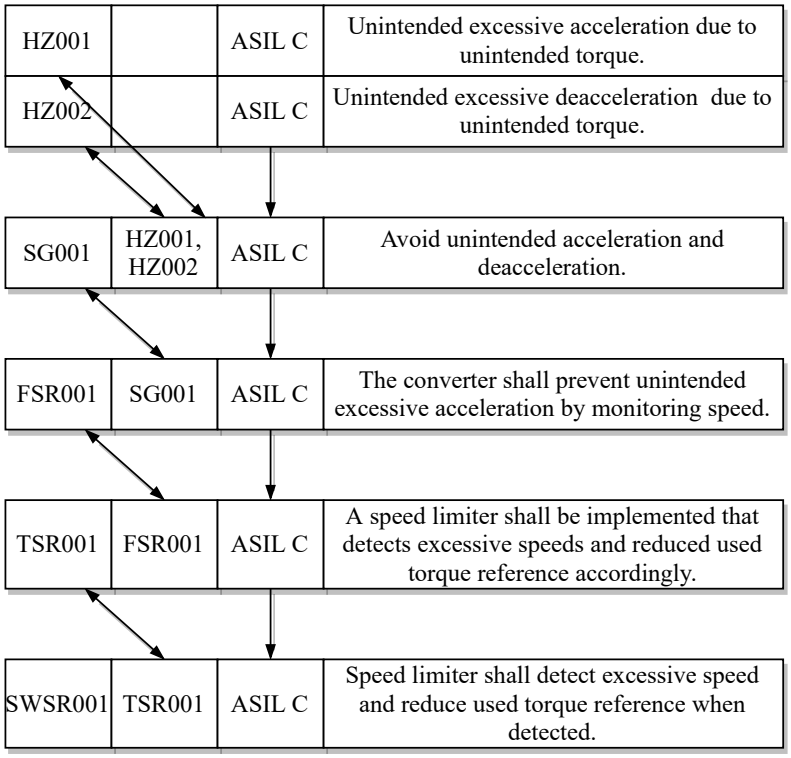


Figure 28. Practical flow from HARA to SWSRs.

Bi-directional traceability between the test cases and the SWSR is created by tagging the test case with the requirement identification number, i.e., `SWSR_001`.

The process of creating suitable test case is approached from the flow depicted in Figure 21. The corresponding SWSR and the software architectural design are taken in to consideration while Table 15 is consulted. The derived test case is given in Table 25, where test identification number, requirement identification number, run type, and description are given.

Table 25. Test case definitions for RBT.

Test ID	Req ID	Type	Description
TEST_001	SWSR_001	manual, CI	Confirm the speed does not exceed speed limits by more than parametrized amount. Pass criteria: speed stays inside the boundaries.
TEST_002	SWSR_001	manual, CI	Confirm that the converter enters safe state when overspeed is met. Pass criteria: converter moves to a safe state.

The implementation of the test case `TEST_001` is given as pseudocode in Listing 4.

Listing 4. Test case `TEST_001` pseudocode.

```
TEST_CASE_001()
{
    // SETUP
    runTestCaseSetup();
    acceptable_overshoot = 100.0;   // rpm
    maximum_speed_limit = 1000.0;   // rpm
    minimum_speed_limit = -1000.0;  // rpm
    torque_reference = 200.0;       // Nm
    actual_speed_data = [];
    used_torque_ref_data = [];

    setMaximumSpeedLimit(maximum_speed_limit);
    setMinimumSpeedLimit(minimum_speed_limit);
    setControllingMode(TORQUE);
    setTorqueReference(torque_reference);

    actual_speed_data = LogSignal(actual_speed);
    used_torque_ref_data = LogSignal(used_torque_ref);

    // EXECUTION
    startRunning();
    waitForSteadyState();
    setTorqueReference(-torque_reference);
    waitForSteadyState();

    // ANALYSIS
    if (max(actual_speed_data) >
            (maximum_speed_limit + acceptable_overshoot)) {
        failTestCase();
    } else if (min(actual_speed_data) <
            (minimum_speed_limit - acceptable_overshoot)) {
        failTestCase();
    } else  {
        passTestCase();
    }
    drawLineGraph(actual_speed_data,
                  used_torque_ref_data);
    runTestCaseTeardown();
}
```

The `SETUP` test phase sets up the testing, i.e., starts simulator and establishes CAN communications, and sets test case specific values. In addition, the `actual_speed` and the `used_torque_ref` values are logged. The `EXECUTION` test phase starts the motor control and waits for the motor to settle in steady state. The converter is then given a negative torque reference with the same magnitude as before, the test waits for the motor to settle in steady state. `ANALYSIS` test phase is started after final steady state is achieved or a timeout occurs before steady state is achieved. The test phase checks whether the motor

had unacceptable overshoot in either direction, if not, the test passes. The logged data is drawn as a line graph and added to the test log. Finally, the test case is torn down. The drawn line graph is represented in Figure 29.

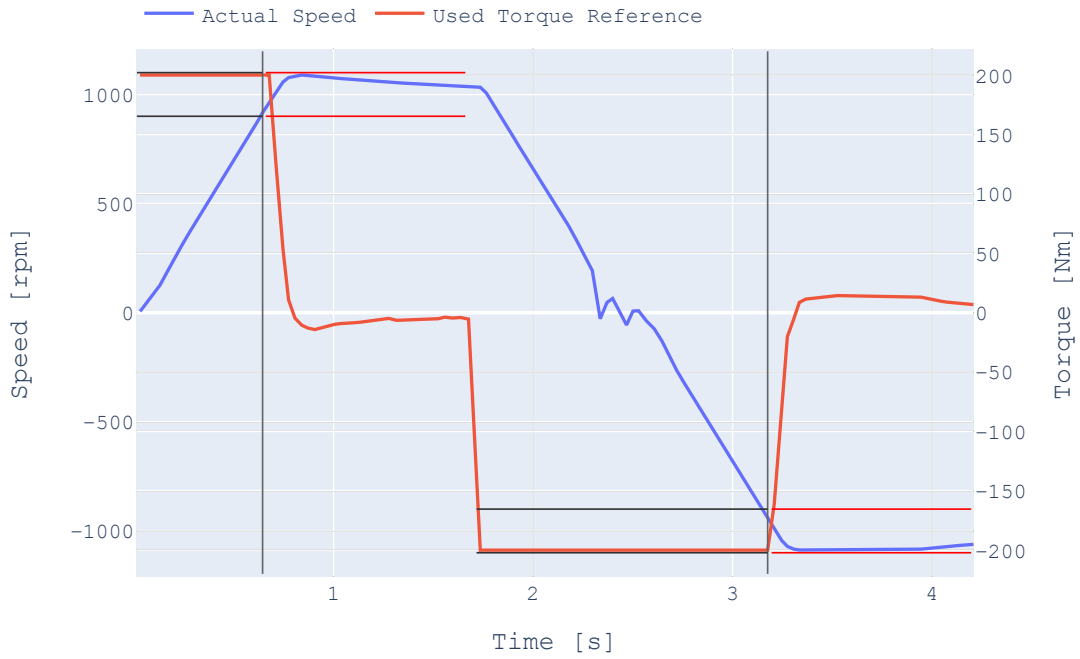


Figure 29. `TEST_001` - actual speed and used torque reference as a function time.

It can be seen that the actual speed does not exceed the acceptable level of overshoot. The speed stays inside the parametrized range, and the used torque reference is reduced when speed limits are surpassed.

The implementation of the test case `TEST_002` is given as pseudocode in Listing 5.

Listing 5. Test case `TEST_002` pseudocode.

```
TEST_CASE_002()
{
    // SETUP
    runTestCaseSetup();
    max_overspeed_threshold = 1050.0;   // rpm
    min_overspeed_threshold = -1050.0;  // rpm
    maximum_speed_limit = 1000.0;       // rpm
    minimum_speed_limit = -1000.0;      // rpm
    torque_reference = 200.0;           // Nm
    actual_speed_data = [];
    used_torque_ref_data = [];

    setMaximumOverspeedThreshold(max_overspeed_threshold);
    setMinimumOverspeedThreshold(min_overspeed_threshold);
    setMaximumSpeedLimit(maximum_speed_limit);
    setMinimumSpeedLimit(minimum_speed_limit);
    setControllingMode(TORQUE);
    setTorqueReference(torque_reference);

    actual_speed_data = LogSignal(actual_speed);
    used_torque_ref_data = LogSignal(used_torque_ref);

    // EXECUTION
    startRunning();
    waitForSpeedToAchieveOverspeed();

    // ANALYSIS
    if (getRunningStatus() == true) {
        failTestCase();
    } else {
        passTestCase();
    }
    drawLineGraph(actual_speed_data,
                  used_torque_ref_data);
    runTestCaseTeardown();
}
```

The `SETUP` phase is done similarly to `TEST_001`, except for the setting of the overspeed thresholds, which are intentionally set close to the speed limits. The `EXECUTION` phase starts the motor control and waits for the speed to hit the maximum overspeed threshold. `ANALYSIS` phase checks whether the motor control is still running, if not, the test passes. The drawn line graph is represented in Figure 30.

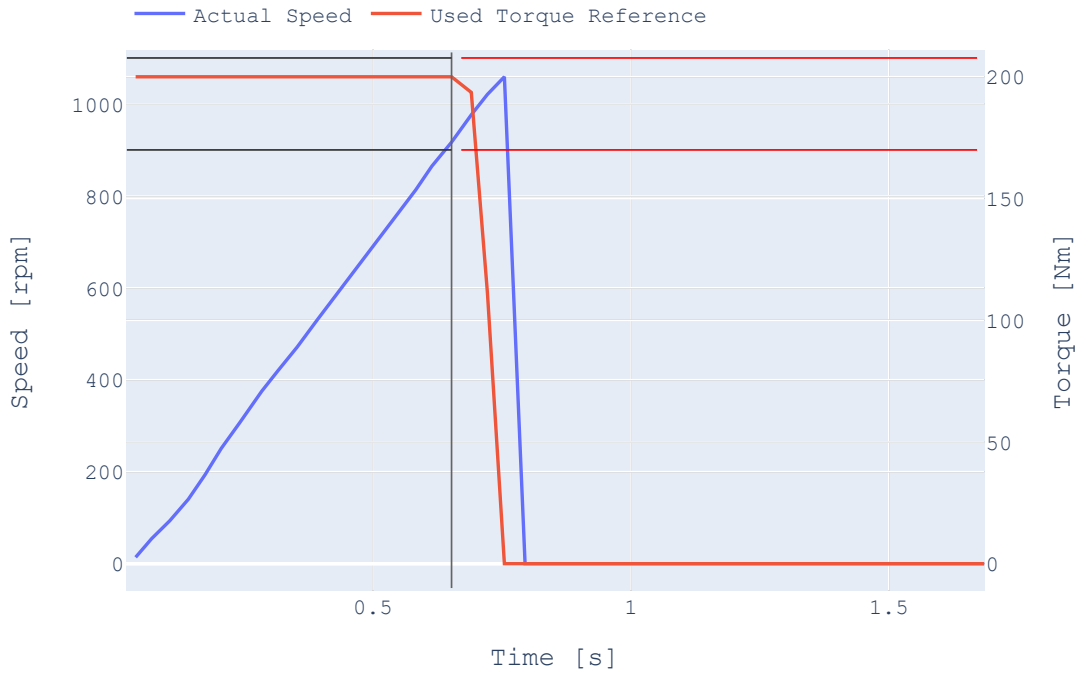


Figure 30. `TEST_002` - actual speed and used torque reference as a function time.

It can be seen that the speed limiter tries to reduce the used torque reference when the speed closes up on the `maximum speed limit`, although it won't prevent overspeed from occurring. When the speed achieves the `max_overspeed_threshold`, the speed falls rapidly to 0 rpm, this is due to overspeed fault being raised and the motor control stopping. It is confirmed that the motor control did in fact stop after the overspeed was achieved. This concludes that the corresponding part of the software architectural design works as intended. The log and report of the test case are given in Appendix H.

4.5 Example of Fault Injection Testing

FIT is one of the most highly recommended technique of integration verification mentioned in the ISO 26262 [27]. The ability to inject faults is also useful for the system testing phase, where the fault reactions can be tested in the target environment.

The basic concept of FIT was depicted in Figure 22. The concept is used to determine the approach taken on creating new test cases. The FIT is not explicitly used for injecting predefined fault codes to the DUT, but as a general disruptive behaviour or stress exhibiting source.

The following example injects a predetermined fault code to the DUT. Introducing

predetermined fault codes is one of the ways to introduce faults in to the system. Tools such as debuggers could be used to inject or corrupt data straight at the memory of the system for more comprehensive testing yielding better test coverage. The example focuses on the same hazards as the section 4.4 did. The overspeed is an abnormal condition where the converter cannot restrain the speed from exceeding parametrized overspeed value. In this case, the overspeed is a result of the speed limiter not working as intended, i.e., not restricting the exerted torque when parametrized speed limit is surpassed. Overspeed fault is tripped when the speed limiter exceeds the parametrized overspeed threshold value and moves to the assigned safe state, i.e., stopping modulation.

The corresponding SWSR and the software architectural design are taken into consideration while Table 15 is consulted. A test case for testing the overspeed fault functionality is derived according to the Table 15. The derived test case is given in Table 26, where test identification number, requirement identification number, run type and description are given.

Table 26. Test case definitions for FIT.

Test ID	Req ID	Type	Description
TEST_003	SWSR_001	manual, CI	Confirm injecting overspeed fault exhibits intended behaviour. Pass criteria: the modulation stops.

The testing of the overspeed fault functionality is done by injecting overspeed fault and confirming whether the converter reacts to it by stopping modulation. The fault injection tests whether the overspeed fault leads to the safe state without there being any overspeed in the first place, it also provides evidence that the software components act independently. The implementation of the test case `TEST_003` is given as pseudocode in Listing 6.

Listing 6. Test case `TEST_003` pseudocode.

```
TEST_CASE_003()
{
    // SETUP
    runTestCaseSetup();
    speed_reference = 500.0;        // rpm
    wait_timer = 1.0;               // seconds
    actual_speed_data = [];
    overspeed_fault_status_data = [];
    running_status_data = [];

    setControllingMode(SPEED);
    setSpeedReference(speed_reference);

    actual_speed_data = LogSignal(actual_speed);
    overspeed_fault_status_data = LogSignal(os_fault_status);
    running_status_data = LogSignal(running_status);

    // EXECUTION
    startRunning();
    waitForTimer(wait_timer);
    injectFault(overspeed_fault);
    waitForTimer(wait_timer);

    // ANALYSIS
    if (getRunningStatus() == true) {
        failTestCase();
    } else {
        passTestCase();
    }
    drawLineGraph(actual_speed_data,
                  overspeed_fault_status_data,
                  running_status_data);
    runTestCaseTeardown();
}
```

The `SETUP` test phase sets up the testing, i.e., starts simulator and establishes CAN communications, and sets test case specific values. The `actual_speed`, `os_fault_status`, and the `running_status` values are logged. The `EXECUTION` test phase starts the motor control and waits for parametrized wait time. The converter is injected with overspeed fault and then the test waits for another parametrized wait time. `ANALYSIS` test phase is started after final wait time is over or a timeout occurs. The test phase checks whether the motor control is still running, if not, the test passes. The logged data is drawn as a line graph and added to the test log. Finally, the test case is torn down. The drawn line graph is represented in Figure 31.

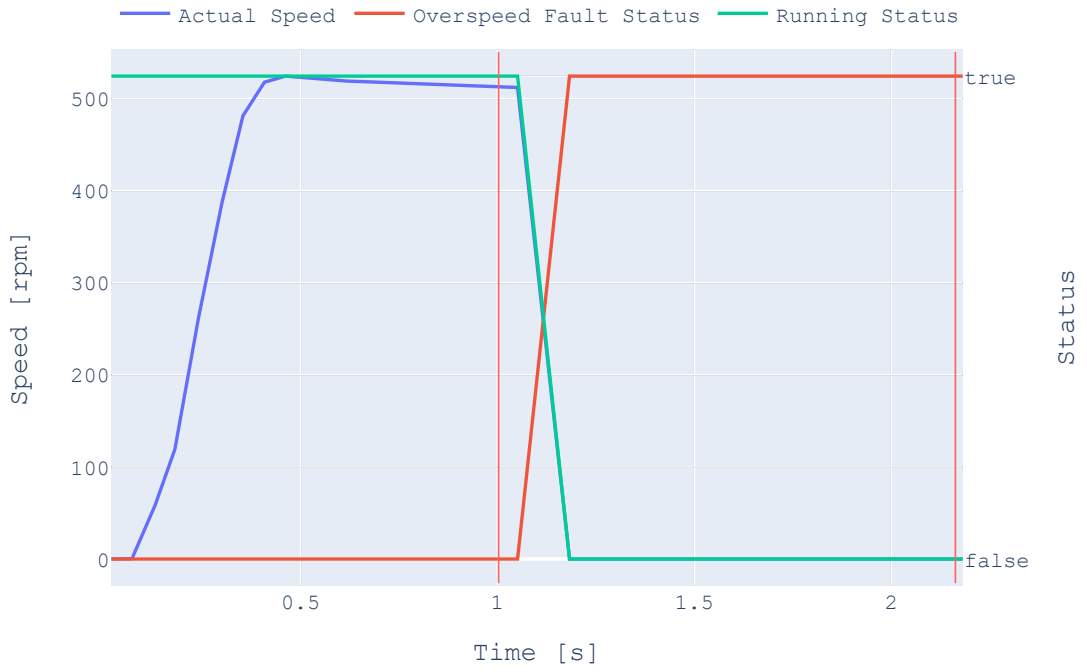


Figure 31. Depiction of the reaction caused by the overspeed fault.

As expected the test case passes and verifies that the corresponding part of software architectural design works as intended. This concludes that the corresponding part of the software architectural design works as intended. The log and report of the test case are given in Appendix I.

5 Conclusions

This research aimed to identify the ISO 26262 requirements that would be needed to achieve ASIL D in electric power converters on the software level. Based on the literature review conducted on available industry standards, research, and other literature, it can be concluded that the highest level of ASIL is achievable in electric power converters. The ISO 26262 compliancy had an extensive amount of requirements, where some of the components, such as the FPGA, must provide rationale for its safety behaviour. The subphases regarding concept phase and product development stood out as the most requirement-heavy ones.

The cross-platform testing was enabled by constructing run pipelines that require the host to define certain run parameters to distinguish what kind of operation is to be expected. The various platform-specific components were implemented as interchangeable to ensure that the same functionality can be utilized platform-independently. To further increase the flexibility of the integration testing pipelines, automated configuration management was introduced. The test running from continuous integration pipelines was enabled in similar fashion as the cross-platform testing. The platform-specific resources were requested from the host using the available command line arguments built in the Robot Framework.

5.1 Discussion

The future of ISO 26262 is still a mystery, will the autonomous driving, new sensory technology, and the increase in algorithmic calculation force the ISO 26262 to account for the change, or will the ISO/PAS 21448 take over from there. The ISO 26262 takes the driver and individuals in the immediate vicinity to be accounted for in the hazard analyses. This might not be the case in the future due to the autonomous driving.

The current version of ISO 26262 is at times subject to interpretation. Especially the analyses subphases, e.g., the HARA, can include various factors like weather, environment, speed, and traffic, through which the analysis is conducted. The freedom of interpretation seems at times too great that some kind of defined metrics or categories would ease the process for both product developers and the customers.

5.2 Further Studies

The ISO 26262 itself is marginally studied in the academia, specifically in terms of knowledge relevant to practical implementation. The ISO 26262 has 11800 results for the keywords *"iso 26262"* in Google Scholar, which is relatively small number [49]. The development of future ISO 26262 editions would benefit from more discussion, aspects, and solutions regarding the ISO 26262. The product development on the hardware level was both omitted in this thesis and has had little study conducted on it. The hardware level product development would gain a lot by having a review paper done to summarize the knowledge so far.

Another marginally studied topic would be the ISO/PAS 21448 and the concept of safety of the intended functionality (SOTIF). The ISO/PAS 21448 has 506 results for the keywords *"iso pas 21448"* in Google Scholar [49]. The ISO/PAS 21448 is only a few years old, but there are already some interesting papers about the ML and the sensory technology regarding automotive.

References

[1] Finnish Standards Association, *SFS-ISO 26262-1:2019: Road Vehicles. Functional Safety. Part 1: Vocabulary*. Helsinki, Finland: Finnish Standards Association, 2019.

[2] H.-L. Ross, *Functional Safety for Road Vehicles - New Challenges and Solutions for E-mobility and Automated Driving*. Lorsch, Germany: Springer, 2016.

[3] Finnish Standards Association, *SFS-EN 61508-1:2011: Functional safety of electrical/electronic/programmable electronic safety-related systems. Part 1: General requirements*. Helsinki, Finland: Finnish Standards Association, 2011.

[4] A. Emadi, K. Rajashekara, S. S. Williamson, and S. M. Lukic, *Topological overview of hybrid electric and fuel cell vehicular power system architectures and configurations*, *IEEE Transactions on Vehicular Technology*, vol. 54, no. 3, pp. 763–770, 2005.

[5] T. Nemeth, A. Bubert, J. N. Becker, R. W. De Doncker, and D. U. Sauer, *A Simulation Platform for Optimization of Electric Vehicles With Modular Drivetrain Topologies*, *IEEE Transactions on Transportation Electrification*, vol. 4, no. 4, pp. 888–900, 2018.

[6] J. Wu, X. Wang, L. Li, C. Qin, and Y. Du, *Hierarchical control strategy with battery aging consideration for hybrid electric vehicle regenerative braking control*, *Energy*, vol. 145, pp. 301–312, 2018.

[7] United States Environmental Protection Agency, *Where the energy goes: Gasoline vehicles*, Online; available at `https://www.fueleconomy.gov/feg/atv.shtml`. [Accessed April 07, 2021], 2021.

[8] United States Environmental Protection Agency, *Where the energy goes: Electric cars*, Online; available at `https://www.fueleconomy.gov/feg/atv-ev.shtml`. [Accessed April 07, 2021], 2021.

[9] IEA, *Global EV Outlook 2020*, Online; available at `https://www.iea.org/reports/global-ev-outlook-2020`. [Accessed March 14, 2021], Jun. 2020.

[10] Eurostat, *Passenger cars, by type of motor energy*, Online; available at: `https://ec.europa.eu/eurostat/web/products-datasets/-/road_eqs_carpda`. [Accessed March 13, 2021], 2021.

[11] Danfoss, *Editron on-highway*, Online; available at `https://www.danfoss.com/en/about-danfoss/our-businesses/power-solutions/danfoss-editron/editron-on-highway/`. [Accessed April 07, 2021], 2021.

[12] Finnish Standards Association, *SFS-EN 62061:2005: Safety of machinery. Functional safety of safety-related electrical, electronic and programmable electronic control systems*. Helsinki, Finland: Finnish Standards Association, 2005.

[13] J. Henriksson, M. Borg, and C. Englund, *Automotive Safety and Machine Learning: Initial Results from a Study on How to Adapt the ISO 26262 Safety Standard*, in *IEEE/ACM 1st International Workshop on Software Engineering for AI in Autonomous Systems (SEFAIAS)*, 2018, pp. 47–49.

[14] Dansk Standards Association, *DS/ISO/PAS 21448:2019: Road vehicles – Safety of the intended functionality*. Copenhagen, Denmark: Dansk Standards Association, 2019.

[15] K.-L. Lu, Y.-Y. Chen, and L.-R. Huang, *FMEDA-Based Fault Injection and Data Analysis in Compliance with ISO-26262*, in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2018, pp. 275–278.

[16] H. H. Kim, *SW FMEA for ISO-26262 Software Development*, in *21st Asia-Pacific Software Engineering Conference*, vol. 2, 2014, pp. 19–22.

[17] D. Ward and S. Crozier, *The uses and abuses of ASIL decomposition in ISO 26262*, in *7th IET International Conference on System Safety, incorporating the Cyber Security Conference*, 2012, pp. 1–6.

[18] C.-S. Lee, Y.-H. Huang, and I.-W. Lan, *Hardware-in-the-Loop Test Case Specification for Verification of Software Safety Requirements in the Context of ISO 26262*, in *International Conference of Electrical and Electronic Technologies for Automotive*, 2018, pp. 1–6.

[19] V. Dhaked, V. Gupta, and J. Harmalkar, *Application of Concept Phase to Design an Electric Powertrain in Compliance with ISO 26262*, in *IEEE 5th International Conference for Convergence in Technology (I2CT)*, 2019, pp. 1–5.

[20] Finnish Standards Association, *SFS-ISO 26262-3:2019: Road Vehicles. Functional Safety. Part 3: Concept phase*. Helsinki, Finland: Finnish Standards Association, 2019.

[21] Finnish Standards Association, *SFS-ISO 26262-8:2019: Road Vehicles. Functional Safety. Part 8: Supporting processes*. Helsinki, Finland: Finnish Standards Association, 2019.

[22] Finnish Standards Association, *SFS-ISO 26262-10:2019: Road Vehicles. Functional Safety. Part 10: Guidelines on ISO 26262*. Helsinki, Finland: Finnish Standards Association, 2019.

[23] Finnish Standards Association, *SFS-ISO 26262-5:2019: Road Vehicles. Functional Safety. Part 5: Product development at the hardware level*. Helsinki, Finland: Finnish Standards Association, 2019.

[24] Finnish Standards Association, *SFS-ISO 26262-9:2019: Road Vehicles. Functional Safety. Part 9: Automotive safety integrity level (ASIL)-oriented and safety-oriented analyses*. Helsinki, Finland: Finnish Standards Association, 2019.

[25] Finnish Standards Association, *SFS-ISO 26262-2:2019: Road Vehicles. Functional Safety. Part 2: Management of functional safety*. Helsinki, Finland: Finnish Standards Association, 2019.

[26] Finnish Standards Association, *SFS-ISO 26262-4:2019: Road Vehicles. Functional Safety. Part 4: Product development at the system level*. Helsinki, Finland: Finnish Standards Association, 2019.

[27] Finnish Standards Association, *SFS-ISO 26262-6:2019: Road Vehicles. Functional Safety. Part 6: Product development at the software level*. Helsinki, Finland: Finnish Standards Association, 2019.

[28] Finnish Standards Association, *SFS-ISO 26262-7:2019: Road Vehicles. Functional Safety. Part 7: Production, operation, service and decommissioning*. Helsinki, Finland: Finnish Standards Association, 2019.

[29] Association for the Advancement of Automotive Medicine, *The Abbreviated Injury Scale 2015 Revision*. Des Plaines, USA: Association for the Advancement of Automotive Medicine, 2016.

[30] R. S. Pressman and B. Maxim, *Software Engineering: A Practitioner's Approach*, 8th ed. New York, USA: McGraw-Hill Education, 2015.

[31] MISRA, *MISRA C:2012 Guidelines for the use of the C language in critical systems*, 3rd ed. Nuneaton, United Kingdom: MISRA, 2019.

[32] Dansk Standards Association, *DS/ISO/IEC/IEEE 24765:2017: Systems and software engineering - Vocabulary*. Copenhagen, Denmark: Dansk Standards Association, 2017.

[33] G. O'Regan, *Concise Guide to Software Testing*. Mallow, Ireland: Springer, 2019.

[34] Dansk Standards Association, *DS/ISO/IEC 25010:2011: Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Copenhagen, Denmark: Dansk Standards Association, 2011.

[35] J. Tian, *Software Quality Engineering*. Hoboken, USA: John Wiley & Sons, 2005.

[36] *ISO/IEC/IEEE International Standard - Software and systems engineering – Software testing – Part 1: Concepts and definitions*, *ISO/IEC/IEEE 29119-1:2013(E)*, pp. 1–64, 2013.

[37] Google, *gMock for Dummies*, Online; available at `https://google.github.io/googletest/gmock_for_dummies.html`. [Accessed May 17, 2021], 2021.

[38] M. Fowler, *Mocks Aren't Stubs*, Online; available at `https://martinfowler.com/articles/mocksArentStubs.html`. [Accessed May 17, 2021], 2007.

[39] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Hoboken, USA: John Wiley & Sons, 2012.

[40] S. Mirarab, A. Ganjali, L. Tahvildari, S. Li, W. Liu, and M. Morrissey, *A requirement-based software testing framework: An industrial practice*, in *IEEE International Conference on Software Maintenance*, 2008, pp. 452–455.

[41] P. Skokovic and M. Rakic-Skokovic, *Requirements-based testing process in practice*, *International Journal of Industrial Engineering and Management*, vol. 1, pp. 155–161, Jan. 2010.

[42] IEC, *Electropedia: The world's online electrotechnical vocabulary*, Online; available at `https://www.electropedia.org/`. [Accessed May 19, 2021], 2021.

[43] J. A. Duraes and H. S. Madeira, *Emulation of Software Faults: A Field Data Study and a Practical Approach*, *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 849–867, 2006.

[44] *ISO/IEC/IEEE International Standard - Software and systems engineering – Software testing – Part 4: Test techniques*, *ISO/IEC/IEEE 29119-4:2015*, pp. 1–149, 2015.

[45] G. M. Kapfhammer, *Empirically Evaluating Regression Testing Techniques: Challenges, Solutions, and a Potential Way Forward*, in *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 99–102.

[46] Robot Framework Foundation, *Robot Framework*, Online; available at `http://robotframework.org`. [Accessed May 21, 2021], 2021.

[47] Robot Framework Foundation, *Robot Framework User Guide*, Online; available at `http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html`. [Accessed May 21, 2021], 2021.

[48] Robot Framework Foundation, *Robot Framework*, Online; available at `https://github.com/robotframework/robotframework`. [Accessed May 25, 2021], 2021.

[49] Google, *Google Scholar*, Online; available at `https://scholar.google.com/`. [Accessed June 15, 2021], 2021.

[50] Tesla, *Emergency Response Guide*, Online; available at `https://www.tesla.com/sites/default/files/downloads/2016_Model_S_Emergency_Response_Guide_en.pdf`. [Accessed April 20, 2021], 2019.

Appendices

A Structure of ISO 26262

The parts and clauses of the ISO 26262 standard are given in Figure 32.

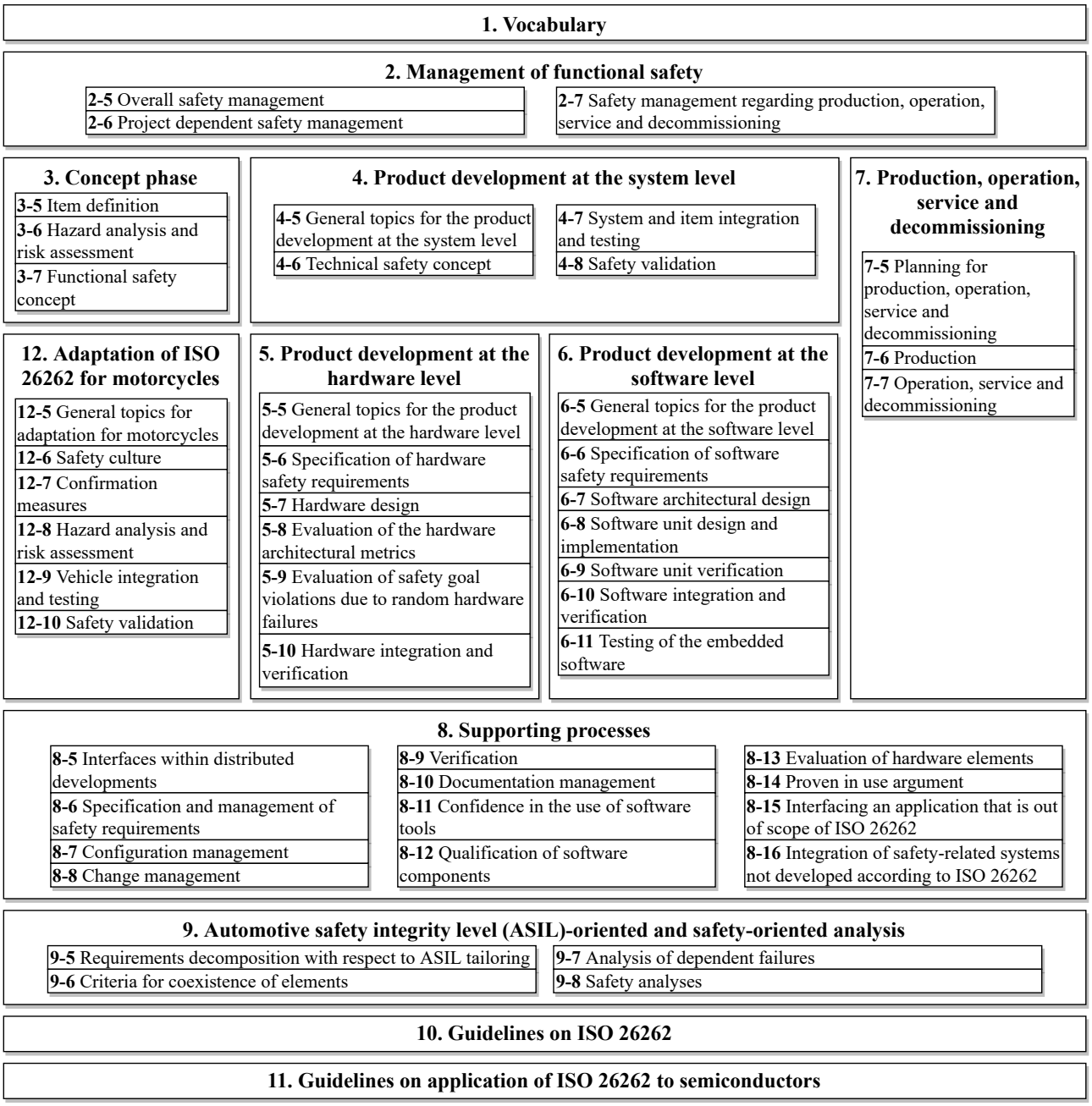


Figure 32. The ISO 26262 standard parts and their respected clauses. [1].

B Example: Item Dissolution

Example of an item dissolution is given to an average all-electric car's electric power conversion system and its thermal protection subsystem, the item is depicted in Figure 33. In this case, the electric power conversion is the item under scope and everything below it in the hierarchy are elements of it.

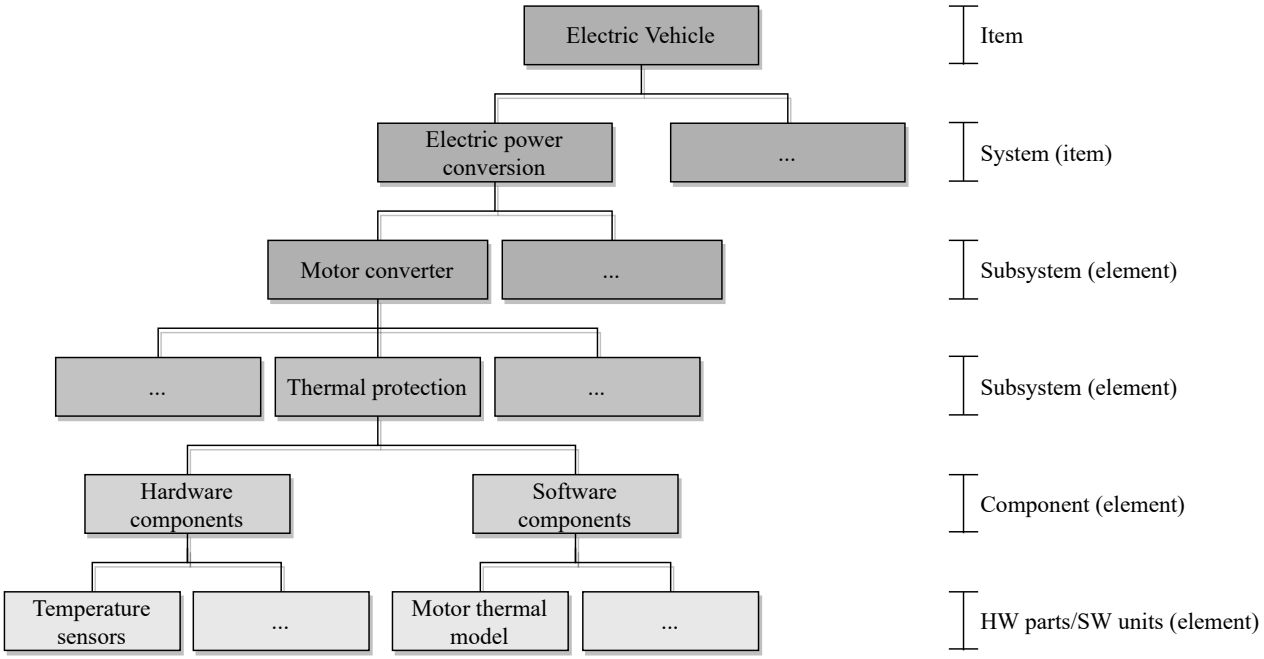


Figure 33. Example of item dissolution for all-electric car's electric power conversion system and its thermal protection subsystem.

C Failure Mode Analyses

Failure mode analyses according to ISO 26262 is given in Figure 34.

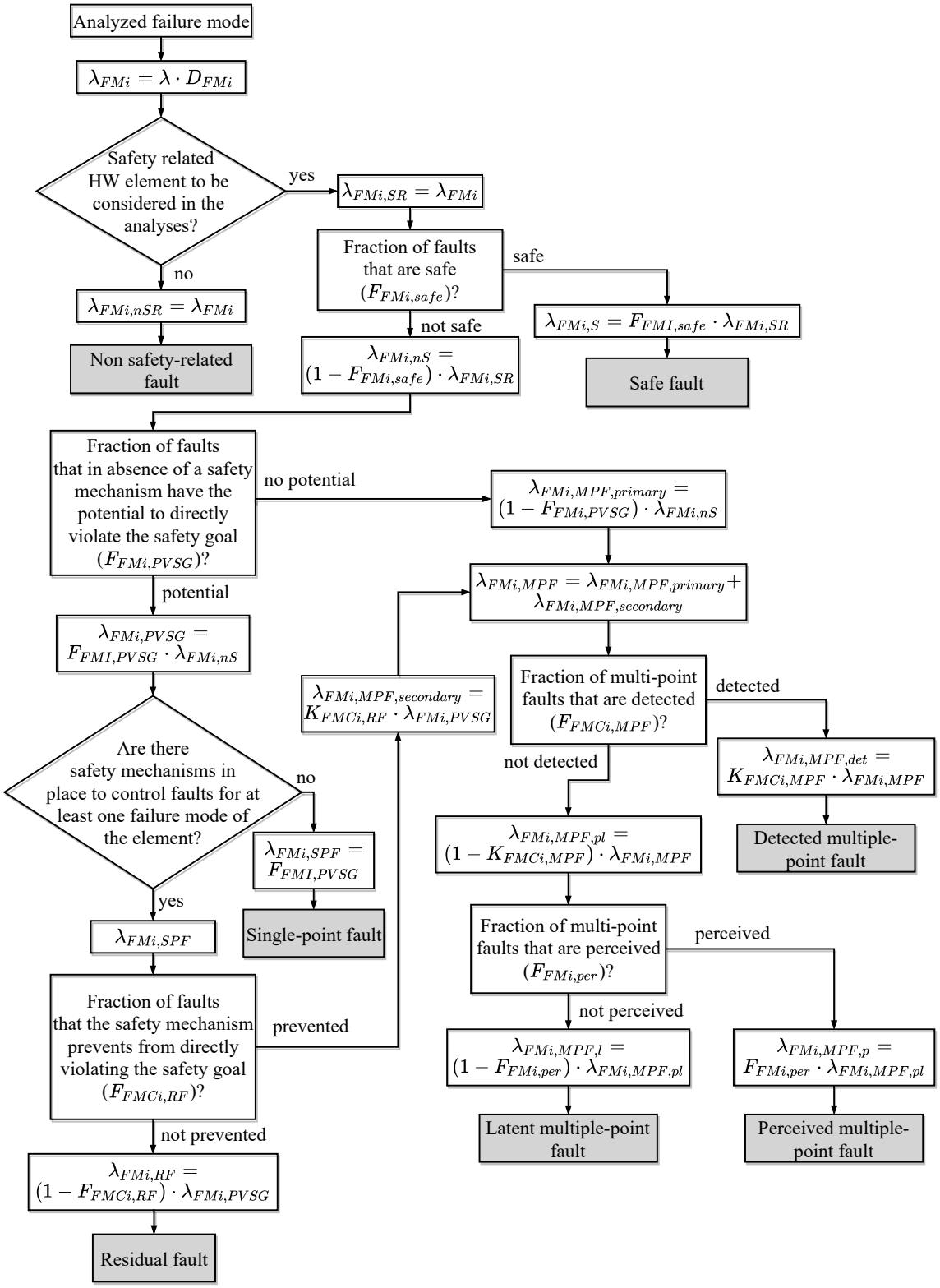


Figure 34. Flow diagram for fault classification and fault class contribution calculation. [22]

Symbols and abbreviations presented in Fig. 34 are described in Table 27.

Table 27. Failure mode analysis symbols and abbreviations. [1], [22]

Variable	Description
λ_{FMi}	The failure rate associated with the i$^{\text{th}}$ failure mode of the hardware element under consideration.
D_{FMi}	The failure mode distribution for the failure mode.
$\lambda_{FMi,nSR}$	Non-safety-related failure rate. Non-safety related failure modes are not considered within the single-point fault metric nor the latent-fault metric.
$\lambda_{FMi,SR}$	Safety-related failure rate. Safety-related failure rates are considered within single-point fault metric and the latent-fault metric.
$\lambda_{FMi,safe}$	Fraction of safe faults of this failure mode. Safe faults do not significantly contribute to the violation of the safety goal.
$\lambda_{FMi,S}$	Failure rate for safe faults.
$\lambda_{FMi,nS}$	Non-safe failure rate. These include single-point faults, residual faults and multiple-point faults (with n=2).
$\lambda_{FMi,PVSG}$	Fraction of non-safe faults that have the potential to directly violate the safety goal without considering any of the safety mechanisms that can exist to prevent this.
$\lambda_{FMi,SPF}$	Single-point faults failure rate.
$\lambda_{FMC,RF}$	Residual fault failure rate.
$\lambda_{FMCi,RF}$	Fraction of prevented $\lambda_{FMi,PVSG}$ by safety mechanisms from violating safety goal. Equivalent to the failure mode coverage with respect to residual faults, i.e., failure mode coverage with respect to residual faults.
$\lambda_{FMi,MPF,primary}$	Primary multiple-point faults failure rate resulting from $\lambda_{FMi,PVSG}$ that have no direct potential to violate safety goal.
$\lambda_{FMi,MPF,secondary}$	Secondary multiple-point faults failure rate resulting from $\lambda_{FMi,PVSG}$ that are controlled by a safety mechanism.
$\lambda_{FMi,MPF}$	The overall multiple-point faults failure rate resulting from the primary and secondary multiple-point faults.
$\lambda_{FMi,MPF,det}$	Failure rate for detected multiple-point faults.
$\lambda_{FMi,per}$	Fraction of the multiple-point faults that are not detected but perceived by the driver.
$\lambda_{FMi,MPF,pl}$	Failure rate for perceived or latent multiple-point faults.
$\lambda_{FMi,MPF,p}$	Failure rate for perceived multiple-point faults.
$\lambda_{FMi,MPF,l}$	Failure rate for latent multiple-point faults.

D Example: ASIL Decomposition

An electric converter has duplicate sensors that measure current from the converter's DC link. The sensors are considered to be working independently, neither of them has an effect on the other. Failure of one sensor does not violate a safety goal. Together the current measurement system is assigned to ASIL D.

The decomposed ASIL is reduced to the same ASIL, this is due to being identical. Therefore, the only available decomposition scheme is found in Equation 1, i.e.,

$$\text{ASIL D} = \text{ASIL B (D)} + \text{ASIL B (D)}.$$

The decomposition is depicted in Figure 35.

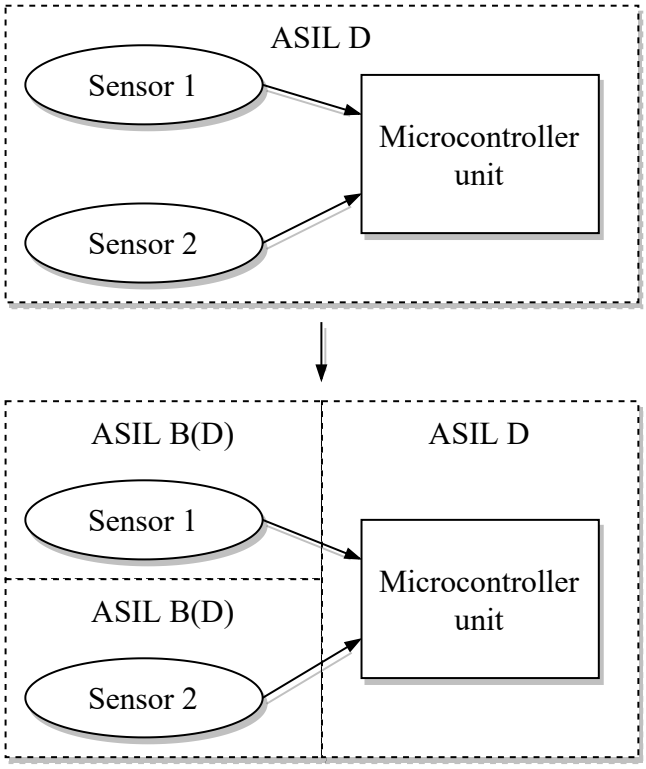


Figure 35. An example of ASIL decomposition.

E Abbreviated Injury Scale

AIS levels and their corresponding descriptions are given in Table 28.

Table 28. AIS levels and descriptions. [29]

AIS level	Description
AIS 0	no injuries
AIS 1	light injuries (e.g., skin-deep wounds or whiplash)
AIS 2	moderate injuries (e.g., deep flesh wounds or uncomplicated long bone fractures)
AIS 3	severe but not life-threatening injuries (e.g., skull fractures without brain injury or more than one fractured rib without paradoxical breathing)
AIS 4	severe injuries, life-threatening, survival probable (e.g., concussion with or without skull fractures with up to 12 hours of unconsciousness or paradoxical breathing)
AIS 5	critical injuries, life-threatening, survival uncertain (e.g., spinal fractures below the fourth cervical vertebra with damage to the spinal cord, intestinal tears, or cardiac tears)
AIS 6	extremely critical or fatal injuries (e.g., fractures of the cervical vertebrae above the third cervical vertebra with damage to the spinal cord or extremely critical open wounds of thoracic or abdominal cavities)

F Example: Hazard Analysis and Risk Assessment

A hazard analysis and risk assessment process for a battery management system (BMS) is conducted. The BMS has a feature that monitors current of the batteries to avoid overcurrents.

The identified hazard is that the overcurrent monitoring malfunctions, i.e., the BMS system does not signal the overcurrent to the protection system nor the operator. The operational situation where the malfunction occurs is on freeway, driving 100 km/h. It is assumed that the battery packs are stored in the chassis of the vehicle and are installed so that the height of the vehicle is minimized, i.e., spread on maximum available area and minimally stacked. It is also assumed that overcurrent will likely cause battery fire.

The identified safety goal is *"The BMS system shall convey its loss of communication"*.

F.1 Severity

The cross-reference between severity classes and AIS scale is presented in the Table 29.

Table 29. The cross-reference between AIS scale and ASIL severity classes. [20]

	Class			
	S0	S1	S2	S3
Description	No injuries	Light and moderate injuries	Severe and life-threatening injuries (survival probable)	Life-threatening injuries (survival uncertain), fatal injuries
AIS scale	AIS 0 and less than 10 % probability of AIS 1-6; or damage that cannot be classified safety-related	More than 10 % probability of AIS 1-6 (and not S2 or S3)	More than 10 % probability of AIS 3-6 (and not S3)	More than 10 % probability of AIS 5-6

Since the batteries are stored under the passengers the potential threat of fire and high heat may cause severe injuries and potentially life-threatening injuries when the age and decision making ability of the passengers is weighted. The determined severity is given in Table 30.

Table 30. The determined severity class.

	Class			
	S0	S1	S2	S3
Description	No injuries	Light and moderate injuries	Severe and life-threatening injuries (survival probable)	Life-threatening injuries (survival uncertain), fatal injuries
Determined class			X	

F.2 Exposure

The layout of the batteries causes the fire and heat to spread evenly throughout the vehicle so the probability of being affected by it is high. The determined exposure is given in Table 31.

Table 31. The determined exposure class.

	Class				
	E0	E1	E2	E3	E4
Description	Incredible	Very low probability	Low probability	Medium probability	High probability
Determined class					X

F.3 Controllability

The control of a battery fire is very hard for the average operator so the controllability class is difficult to control or uncontrollable, e.g. Tesla states in the their emergency response guide for the Model S that approximately 11356 liters of water are required to fully extinguish and cool the batteries, this is not something an average driver can perform [50]. The determined controllability is given in Table 32.

Table 32. The determined controllability class.

	Class			
	C0	C1	C2	C3
Description	Controllable in general	Simply controllable	Normally controllable	Difficult to control or uncontrollable
Determined class				X

F.4 ASIL

When all of the classes have been determined and inserted to the ASIL determination matrix the resulting ASIL level corresponds to ASIL C as seen in Table 33.

Table 33. The determined ASIL level according to previously assigned severity, exposure and controllability classes.

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
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

G Robot Showcase: Test Log and Report

Figures 36 and 37 depict the test log regarding the Robot showcase.

Showcase Log

Generated
20210602 23:00:41 UTC+03:00

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	3	2	1	0	00:00:00	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
showcase_test	3	2	1	0	00:00:00	
showcase_test_1	1	1	0	0	00:00:00	
showcase_test_2	1	0	1	0	00:00:00	
showcase_test_3	1	1	0	0	00:00:00	

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Showcase	3	2	1	0	00:00:00	
Showcase.**Example**	3	2	1	0	00:00:00	

Test Execution Log

SUITE **Showcase** 00:00:00.051
Full Name: Showcase
Source: ...\Showcase
Start / End / Elapsed: 20210602 23:00:41.265 / 20210602 23:00:41.316 / 00:00:00.051
Status: 3 tests total, 2 passed, 1 failed, 0 skipped

SUITE **Example** 00:00:00.025
Full Name: Showcase.Example
Documentation: An example test suite to describe the syntax of Robot scripting.
Source: ...\example.robot
Start / End / Elapsed: 20210602 23:00:41.289 / 20210602 23:00:41.314 / 00:00:00.025
Status: 3 tests total, 2 passed, 1 failed, 0 skipped

TEST **Local and Global Variables Test** 00:00:00.002
Full Name: Showcase.Example.Local and Global Variables Test
Documentation: Demonstrate the use of local and global variables.
Tags: showcase_test, showcase_test_1
Start / End / Elapsed: 20210602 23:00:41.301 / 20210602 23:00:41.303 / 00:00:00.002
Status: PASS

KEYWORD BuiltIn.**Log** ${EXTERNAL_STR} 00:00:00.001
Documentation: Logs the given message with the given level.
Start / End / Elapsed: 20210602 23:00:41.301 / 20210602 23:00:41.302 / 00:00:00.001
23:00:41.302 INFO olleH

KEYWORD BuiltIn.**Should Be Equal** ${message}, Hello 00:00:00.000
Documentation: Fails if the given objects are unequal.
Start / End / Elapsed: 20210602 23:00:41.302 / 20210602 23:00:41.302 / 00:00:00.000

KEYWORD BuiltIn.**Should Not Be Equal** ${EXTERNAL_STR}, ${message} 00:00:00.001
Documentation: Fails if the given objects are equal.
Start / End / Elapsed: 20210602 23:00:41.302 / 20210602 23:00:41.303 / 00:00:00.001

TEST **Python Library Test** 00:00:00.003
Full Name: Showcase.Example.Python Library Test
Documentation: Demonstrate the use of Python keywords.
Tags: showcase_test, showcase_test_2
Start / End / Elapsed: 20210602 23:00:41.304 / 20210602 23:00:41.307 / 00:00:00.003
Status: FAIL
Message: Words olleH and World are not palindrome.

FOR **${word} IN [${message} | World]** 00:00:00.003
Start / End / Elapsed: 20210602 23:00:41.304 / 20210602 23:00:41.307 / 00:00:00.003

VAR **${word} = Hello** 00:00:00.001
Start / End / Elapsed: 20210602 23:00:41.304 / 20210602 23:00:41.305 / 00:00:00.001

KEYWORD ${result} = confirmation.Confirmation.**Words Are Palindrome** ${EXTERNAL_STR}, ${word} 00:00:00.001
Documentation: Checks whether given words are palindrome of each other.
Start / End / Elapsed: 20210602 23:00:41.304 / 20210602 23:00:41.305 / 00:00:00.001
23:00:41.305 INFO True
23:00:41.305 INFO ${result} = True

IF **${result} == ${False}** 00:00:00.000

Figure 36. Showcase test log 1/2.

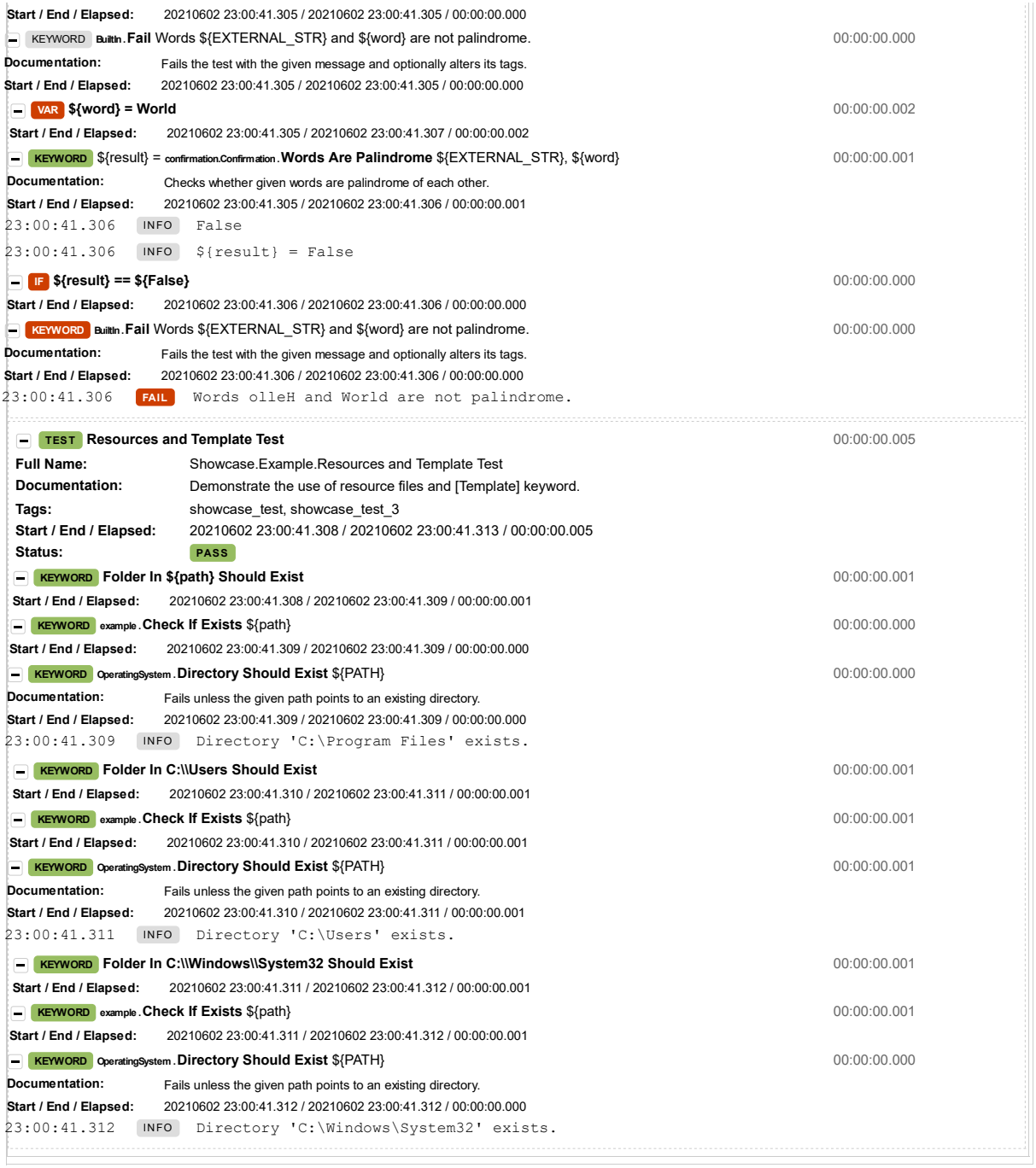


Figure 37. Showcase test log 2/2.

Figure 38 depicts the test report regarding the Robot showcase.

Showcase Report

Generated
20210602 23:00:41 UTC+03:00

Summary Information

Status:	**1 test failed**
Start Time:	20210602 23:00:41.265
End Time:	20210602 23:00:41.316
Elapsed Time:	00:00:00.051
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	3	2	1	0	00:00:00	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
showcase_test	3	2	1	0	00:00:00	
showcase_test_1	1	1	0	0	00:00:00	
showcase_test_2	1	0	1	0	00:00:00	
showcase_test_3	1	1	0	0	00:00:00	

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Showcase	3	2	1	0	00:00:00	
Showcase . **Example**	3	2	1	0	00:00:00	

Test Details

All | Tags | Suites | Search

Suite:

Test:

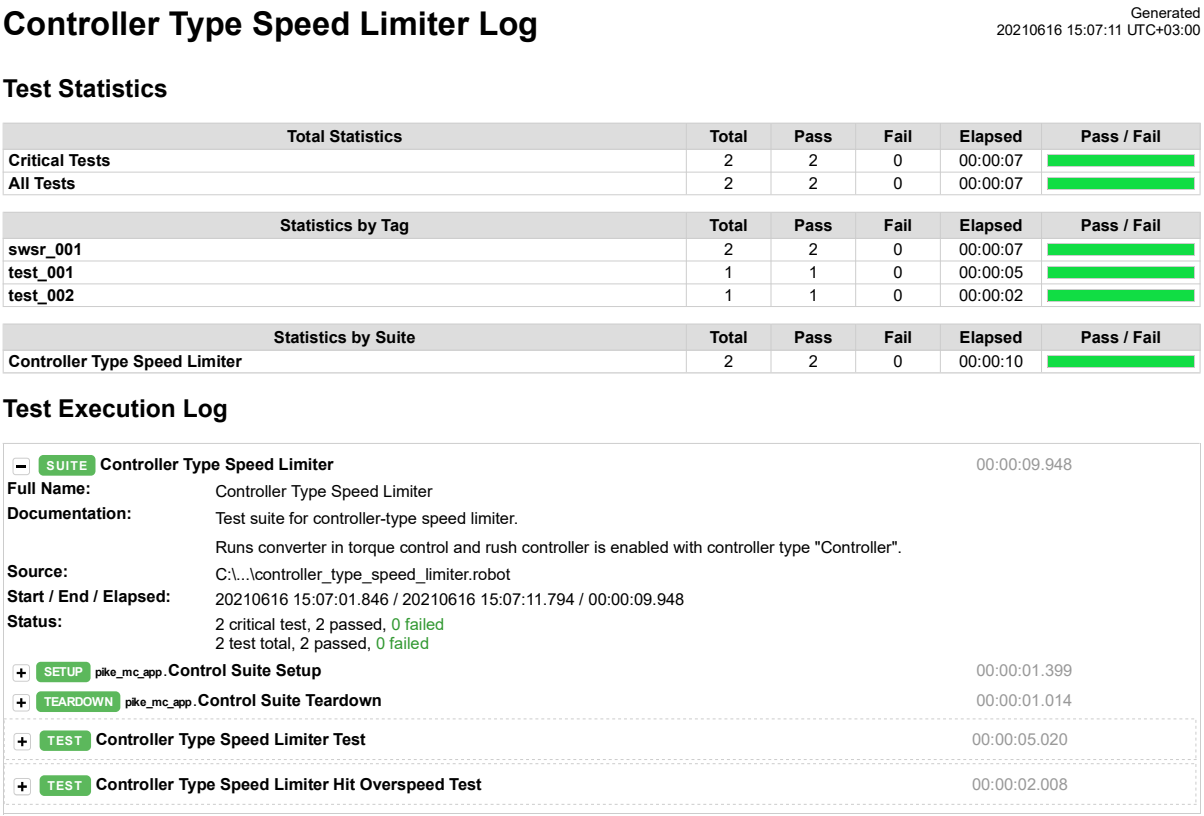
Include:

Exclude:

Figure 38. Showcase test report.

H Requirements-Based Testing: Test Log and Report

Figure 39 depicts the test log regarding the speed limiter using RBT test cases.



Controller Type Speed Limiter Log

Generated
20210616 15:07:11 UTC+03:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	2	2	0	00:00:07	
All Tests	2	2	0	00:00:07	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
swsr_001	2	2	0	00:00:07	
test_001	1	1	0	00:00:05	
test_002	1	1	0	00:00:02	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Controller Type Speed Limiter	2	2	0	00:00:10	

Test Execution Log

SUITE Controller Type Speed Limiter 00:00:09.948

Full Name: Controller Type Speed Limiter
Documentation: Test suite for controller-type speed limiter.
Runs converter in torque control and rush controller is enabled with controller type "Controller".
Source: C:\...\controller_type_speed_limiter.robot
Start / End / Elapsed: 20210616 15:07:01.846 / 20210616 15:07:11.794 / 00:00:09.948
Status: 2 critical test, 2 passed, 0 failed
2 test total, 2 passed, 0 failed

SETUP pike_mc_app.Control Suite Setup 00:00:01.399
TEARDOWN pike_mc_app.Control Suite Teardown 00:00:01.014
TEST Controller Type Speed Limiter Test 00:00:05.020
TEST Controller Type Speed Limiter Hit Overspeed Test 00:00:02.008

Figure 39. Test log for the speed limiter using RBT.

Figure 40 depicts the test report regarding the speed limiter using RBT test case.

Controller Type Speed Limiter Report

Generated
20210616 15:07:11 UTC+03:00

Summary Information

Status:	All tests passed
Documentation:	Test suite for controller-type speed limiter.
	Runs converter in torque control and rush controller is enabled with controller type "Controller".
Start Time:	20210616 15:07:01.846
End Time:	20210616 15:07:11.794
Elapsed Time:	00:00:09.948
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	2	2	0	00:00:07	
All Tests	2	2	0	00:00:07	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
swsr_001	2	2	0	00:00:07	
test_001	1	1	0	00:00:05	
test_002	1	1	0	00:00:02	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Controller Type Speed Limiter	2	2	0	00:00:10	

Test Details

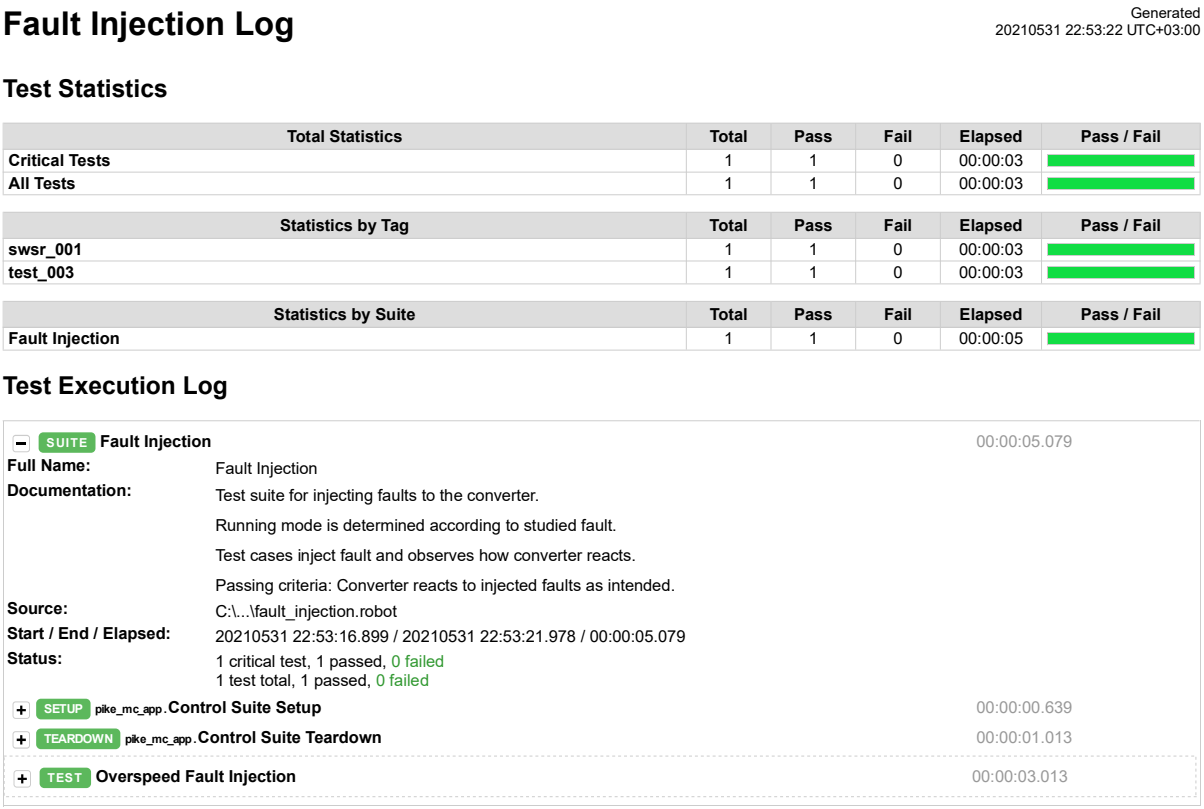
Totals | Tags | Suites | Search

Type:

Figure 40. Test report for the speed limiter using RBT.

I Fault Injection Testing: Test Log and Report

Figure 41 depicts the test log regarding the overspeed fault using FIT test case.



Fault Injection Log

Generated
20210531 22:53:22 UTC+03:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:00:03	
All Tests	1	1	0	00:00:03	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
swsr_001	1	1	0	00:00:03	
test_003	1	1	0	00:00:03	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Fault Injection	1	1	0	00:00:05	

Test Execution Log

SUITE **Fault Injection** 00:00:05.079

Full Name: Fault Injection

Documentation: Test suite for injecting faults to the converter.
Running mode is determined according to studied fault.
Test cases inject fault and observes how converter reacts.
Passing criteria: Converter reacts to injected faults as intended.

Source: C:\...\fault_injection.robot

Start / End / Elapsed: 20210531 22:53:16.899 / 20210531 22:53:21.978 / 00:00:05.079

Status: 1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed

SETUP pike_mc_app.**Control Suite Setup** 00:00:00.639

TEARDOWN pike_mc_app.**Control Suite Teardown** 00:00:01.013

TEST **Overspeed Fault Injection** 00:00:03.013

Figure 41. Test log for the overspeed fault using FIT.

Figure 42 depicts the test report regarding the overspeed fault using FIT test case.

Fault Injection Report

Generated
20210531 22:53:22 UTC+03:00

Summary Information

Status:	All tests passed
Documentation:	Test suite for injecting faults to the converter.
	Running mode is determined according to studied fault.
	Test cases inject fault and observes how converter reacts.
	Passing criteria: Converter reacts to injected faults as intended.
Start Time:	20210531 22:53:16.899
End Time:	20210531 22:53:21.978
Elapsed Time:	00:00:05.079
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:00:03	
All Tests	1	1	0	00:00:03	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
swsr_001	1	1	0	00:00:03	
test_003	1	1	0	00:00:03	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Fault Injection	1	1	0	00:00:05	

Test Details

Totals | **Tags** | **Suites** | **Search**

Type:

Figure 42. Test report for the overspeed fault using FIT.