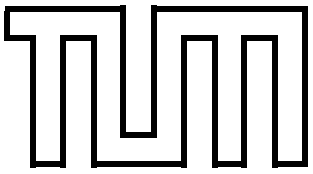


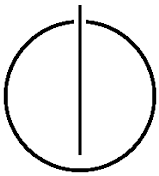
FAKULTÄT FÜR INFORMATIK

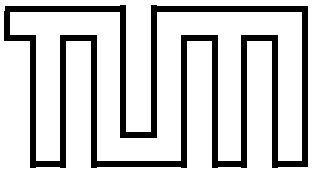
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Development Process for Autonomous Vehicles

Tobias Weigl





FAKULTÄT FÜR INFORMATIK

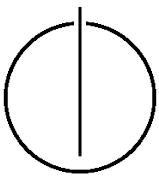
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Entwicklungsprozess für Autonome Fahrzeuge

Development Process for Autonomous Vehicles

By:	Tobias Weigl
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisor:	Jonas Eckhardt (TUM)
Advisor:	Dr. Lukas Bulwahn (BMW Car IT GmbH)
Submission Date:	March 17, 2014



I assure the single handed composition of this master's thesis only supported by declared resources.

March 17, 2014

Acknowledgments

I would like to express my gratitude to my advisor Dr. Lukas Bulwahn for sharing his insights and giving me valuable advice on the topic and on scientific writing.

The same holds for my other advisor Jonas Eckhardt, who also gave me many most helpful remarks on the overall structure of my thesis.

I want to express my sincere gratitude to BMW Car IT GmbH for supporting me, providing me with materials for my work and giving me interesting insights into the development of future automotive technologies.

Abstract

Autonomous vehicles are expected to increase driving comfort and help solve several problems of modern urban car usage. In order to materialize self-driving cars, BMW Car IT GmbH is doing research in algorithmic solutions and is pushing forward prototype construction.

This thesis discusses obstacles in the implementation of safety engineering activities into BMW Car IT GmbH's development process. The presented new activities can help to improve the confidence in the functional safety of the developed prototypes.

GSN safety case modeling is proposed as new safety engineering activity in the agile development process, a management tool for the implementation of a Yocto Linux layer to deploy the developed software is created, and an attempt for a toolchain to develop generic ISO 26262 ASIL D software components is proposed and evaluated.

Zusammenfassung

Von autonomen Fahrzeugen erwartet man einen Komfortgewinn beim Fahren, sowie Lösungen für die vielfältigen Probleme moderner Autonutzung in der Stadt. Um selbstfahrende Fahrzeuge zu verwirklichen, erforscht die BMW Car IT GmbH geeignete Algorithmen und forciert den Bau von Prototypen.

Diese Arbeit diskutiert die Hindernisse bei der Implementierung von Aktivitäten aus der Sicherheitstechnik (Safety Engineering) in den Entwicklungsprozess der BMW Car IT GmbH. Die vorgestellten neue Aktivitäten können dabei helfen, das Vertrauen in die funktionale Sicherheit der Prototypen zu stärken.

Vorgeschlagen wird GSN Modellierung von Sicherheitsstudien (Safety Cases) als neue Aktivität im agilen Entwicklungsprozess der BMW Car IT GmbH; desweiteren wird ein Managementtool zur Implementierung eines Softwarelayers für Yocto Linux entwickelt auf den die entwickelte Software verteilt werden kann; außerdem wird ein Vorschlag einer Werkzeugkette gegeben und evaluiert, um generische ISO 26262 ASIL D Softwarekomponenten zu entwickeln.

Contents

1 Introduction **1**

1.1 Historic Change in Automotive Industry . . . 1
1.2 Motivation for Autonomous Driving . . . 1
1.3 Status Quo and Prospects . . . 3
1.4 Research Context: Autonomous Driving at BMW Car IT GmbH . . . 4
1.5 Contributions, Focus and Methods . . . 5
1.6 Boundaries . . . 6
1.7 Outline . . . 6
1.8 Remarks on ISO 26262 Citation Style . . . 7

2 Background **9**

2.1 Safety Scopes for Autonomous Driving . . . 11
2.1.1 Chosen Safety Scope . . . 14
2.2 Overview of ISO 26262 . . . 14
2.3 Significance of ISO 26262's Release . . . 16
2.4 Related Work . . . 17
2.4.1 Safety-Critical Computer Systems and Functional Safety in General 17
2.4.2 Process Design for Safety-Critical Computer Systems . . . 19
2.4.3 Safety Cases, Functional Safety Planning and Management . . . 25
2.4.4 Status Quo in Prototyping . . . 30
2.4.5 Conclusion on Related Publications . . . 30
2.5 Definitions and Concepts . . . 32

3 The Agile Development Process at BMW Car IT GmbH **39**

3.1 Agile Development for Autonomous Driving . . . 39
3.2 The CAP Reference Process . . . 39
3.2.1 Observations about the CAP . . . 40
3.2.2 The ISO 26262 Safety Lifecycle Mapped on the CAP . . . 40
3.3 Analysis of the Tailored CAP for Autonomous Driving . . . 42
3.3.1 Requirements Engineering . . . 43
3.3.2 Quality Management . . . 43

3.3.3 Build and Deployment Process . . . 44

4 Arguing Functional Safety in Accordance with ISO 26262 45
4.1 In-Depth Analysis of ISO 26262 . . . 45
4.1.1 General Problems . . . 46
4.1.2 Applicability of ISO 26262 . . . 46
4.1.3 Alternative Approaches to Software Development . . . 49
4.1.4 Important Concepts . . . 50
4.1.5 Special Implications on Failure Modes for Autonomous Vehicles . 51
4.2 ISO 26262 Safety Case Management in GSN . . . 52
4.2.1 GSN for Safety Case Development, Reuse, and Maintenance . . . 55
4.3 GSN Applied in the Project . . . 57
4.3.1 Methodology . . . 58
4.3.2 Exemplary High-Level Safety Case . . . 58
4.3.3 Integration in the CAP . . . 58

5 Improvements in Functional Safety 61
5.1 Model-Driven Development . . . 61
5.1.1 Solution Frameworks . . . 64
5.1.2 Comparison . . . 66
5.2 Computing Platform for the Software Deployment . . . 66
5.2.1 Arrival of Automotive Linux . . . 66
5.2.2 A Tool to Supervise the ROS Layer Integration . . . 68
5.3 ISO 26262 Part 6 Checked Against the CAP . . . 68
5.3.1 Safety Lifecycle Topics for the Software Development . . . 69

6 Evaluation 77
6.1 Evaluation of the GSN Safety Case Report . . . 77
6.2 Evaluation of MDD . . . 77
6.3 Yocto Linux Platform Efforts . . . 78
6.4 Evaluation of ISO 26262 Part 6 Checked Against the CAP . . . 79

7 Conclusion 81
7.1 Results . . . 81
7.2 Lessons Learned . . . 82
7.3 Outlook and Further Research . . . 82

8 Glossary 85

9 Appendix 91
9.1 Bitbake Build Report Script Tool - Excerpt . . . 91

Bibliography 93

1 Introduction

Autonomous vehicles will give the driver the chance to choose between manual driving and automatic traveling. The impact of this technology to the automotive landscape will be manifold, yet, there are still challenges to master. The general feasibility of safe self-driving vehicles is, however, largely unquestioned among experts [SW12, p. 10].

1.1 Historic Change in Automotive Industry

The majority of technological advances in automotive industry were evolutionary ones. Cars became cleaner and safer, yet the progress in general was stepwise. Now, the automotive industry finds itself on the verge of significant change that is expected to have a huge impact on the way we perceive mobility itself [SW12, p. 4]: the autonomous car. Competition between traditional manufacturers and non-automotive technology companies in research and prototypes is already serious [Win13] and car manufacturers oversleeping this change are possibly missing one of the most important future markets [Mui13].

> “The revolution, when it comes, will be engendered by the advent of autonomous or “self-driving” vehicles. And the timing may be sooner than you think. (…)
>
> The marketplace will not merely accept self-driving vehicles; it will be the engine pulling the industry forward.
>
> - KPMG and the Center for Automotive Research [SW12, pp. 4-6] ”

1.2 Motivation for Autonomous Driving

The reasons for the market to demand autonomous cars lie in socioeconomic and demographic trends.

Urbanization is such a substantial demographic trend [Rei12] that affects millions of people world wide, which is particularly observable for numerous still growing megacities [KC13]. The resulting chronic traffic congestion, air pollution, slow speed in rush hours, and the struggle for empty parking spots have become overly annoying and lead townspeople and commuters to increasingly demand alternatives and smarter concepts for their personal mobility, culminating in about one third of drivers being interested in buying an autonomous car [Alp12].

Also, for tomorrow's potential car customers currently beeing in their early or late 20's, priorities are shifting. These members of the *millenial generation*—or *generation Y*—have reprioritized their values and the car is declining as a symbol of status and independence [DPA12]; possession of a car or driving one is becoming more and more unimportant [Wei12] and the number of young people that are not in a hurry to obtain a driver's license is growing [Bog11].

Instead, the car is changing from a primary mobility provider to a *primus inter pares* that has to integrate in a mobility environment where going by bike, car sharing and public transportation are gaining momentum.

A substantial aspect of tomorrows urban mobility is the self-driving car and autonomous car fleets that provide solutions for these challenges [New11].

Advantages of Autonomous Driving

Autonomous cars provide opportunities for manufacturers and advantages over manual driving which can improve people's mobility [SW12, pp. 24-31][CG13]:

- Travel time, especially commuting time, can be used productively
- Autonomous parking ends the struggle of finding parking spots
- Intelligent algorithms avoid traffic jams and allow more predictable travel times
- Optimal driving strategies improve energy efficiency
- Self-driving cars can ultimately be crashless cars
- The technology is compatible with existing street infrastructure
- Car-sharing evolves, with cars driving directly to customers
- People without a driver's license are enabled self-determined car usage

At times, people that enjoy driving state that they do not want to be driven autonomously. However, self-driving cars don't have to operate autonomously all the time and they can rather take over only in case the human driver has no interest in driving manually, e.g., because of the traffic situation, or to mitigate hazardous situations the human driver is not aware of or is unable to react to in time.

1.3 Status Quo and Prospects

To realize self-driving vehicles, several distinct sensor types are used to build up a virtual local environment model around the car for autonomous navigation. Some of those sensors are already available in upper-class cars as part of existing advanced driver assistance systems, such as laser or radar sensors in adaptive speed controls [Lau13]. A similar situation holds for required actuators, which are either already part of x-by-wire systems in common cars (acceleration, cruise control) and as such suitable to be put in use also as actuation units for autonomous driving, or—in case of steering and breaking—are potentially upgradable from driver-supporting operation to fully driver-independent operation as demonstrated in many autonomous lab cars. However, evolution of steering and braking to by-wire systems in common cars suffered a little delay recently [WIHS04], yet, at least brake-by-wire systems had already found their way into retail cars, but were canceled again due to technical difficulties, expensiveness, and low customer acceptance [Mei05].

These potential synergistic effects in using already available components motivate the research efforts, since there is hope to design autonomous cars cost efficiently and thus make them available to the average customer comparably soon.

Figure 1.1: Driver Information and Assistance Systems - Influencing the Driving Process

Vehicles with autonomous abilities will arrive in several steps and the fully autonomous car can be seen as the final step in the evolution of safety-critical driver assistance systems, as high-level active safety systems are inherently evolving towards full autonomy [Ben04, p. 3]. Available partially autonomous driver assistance systems today are intelligent parking assistance systems for reverse parallel parking that require the driver to only operate gas and brake pedals, and adaptive speed control that is able to hold a certain distance to a vehicle in front by adjusting the controlled car's speed. These systems have in common that they either leave control of the steering, or of gas and braking to the driver. For this reason, legal obligations are easier to fullfill, since the human driver remains always an active participant in the driving process ultimately and is able to react to and to revise system malbehaviour.

Fully autonomous operating assistance systems that are expected to arrive soon are *autonomous emergency braking assistants* that intervene in critical situations the driver

would not be able to react to [Eur13], *emergency stop assistants* monitoring the driver's health ready to take over in case of health issues to safely stop and park the car [Gov13], and autonomously driving vehicles for slow to medium speed scenarios in easily controllable environments like highway lanes—with the latter systems being referred to as *automated highway systems* [SW12, p. 15] or systems for *highly automated driving* [KAAR12].

Autonomous braking to implement last-chance accident prevention, or to actively aid in the mitigation of the effects caused by an unavoidable accident will be required by legal authorities soon for vehicles that are intended to achieve the highest safety score in safety ratings [Tre12]. Therefore the need to offer such systems becomes urgent soon for manufacturers to stay competetive concerning safety ratings.

While legal regulations today are still an issue in most countries, the situation for fully autonomous cars is increasingly promising, especially in the United States. Several companies have already been able to gain licenses for street use of experimental autonomous vehicles in Nevada [Lav12b] and the Californian governor has signed a bill requesting a legal draft for autonomous driving by 2015 [Lav12a]. Politicians showed themselves convinced of the superior safety of autonomous cars compared to human drivers [Las13] and there are examples for autonomous cars successfully driving on regular streets under supervision of trained drivers [Goo12]. However, there have also been recent cases of autonomous cars considerably failing to operate as intended and instead displayed dangerous behavior [Sch11, min. 68:00-70:00].

1.4 Research Context: Autonomous Driving at BMW Car IT GmbH

BMW Car IT GmbH is doing research on autonomous driving. The current software development process and the final hardware target platform have to be prepared to allow moving the software from simulation environments to autonomously driving lab cars.

To faciliate the construction of real-world prototypes, the development process must be continuously improved regarding safety engineering activities, where the final goal is to build self-driving prototypes that can be used for testing under real-world traffic conditions and to prepare the way for the final autonomous car. During development, such lab cars can also be equipped with additional safety mechanisms, such as emergency shut down buttons that are not meant to be included in the final car.

A comprehensive safety process, however, interferes with the company's agile development process, CAP (Car IT Agile Process), which was established to support research oriented projects through iterative, exploratory development, while it also allows to

manage these projects as ordered and delivered IT projects in a business context to the company's terms. In the CAP, fine-grained requirements are not predetermined at project start, but are *explored* in the course of prototyping, testing, and evaluation of the resulting feedback. At the beginning, there typically stand very general project visions only, which gives the developers more freedom and flexibility to explore solutions and to learn from prototypes compared to plan-driven processes.

Safety-Related Automotive Systems

While other embedded automotive systems, e.g., in the multimedia and entertainment domain, are mostly detached from safety-relevant subsystems, controllers to implement autonomous driving are safety-relevant, because they are directly linked with actuators intended to affect or manipulate the driving process and can therefore cause harm to occupants and the environment. The development process therefore needs to take care of necessary safety reflections in order to minimize potential risks.

A safety process to achieve this consists of potentially expensive (i.e., time-consuming and work-intensive) activities to detect and assess hazards that can lead to hazardous situations and strategies to control or eliminate the risks for such hazardous situations to occur.

Research Question

This raises the research question for this thesis: Considering the current capabilities of the development process, how can confidence in the safe use of developed prototypes for autonomous driving be improved at BMW Car IT GmbH?

1.5 Contributions, Focus and Methods

This thesis explores which safety activities are suitable to accompany the current development process at BMW Car IT GmbH, CAP, so that confidence in the safety of developed self-driving vehicle prototypes is improved. This covers activities to detect and describe safety requirements and their regarding hazards, setting them in relation and deriving appropriate safety concepts.

Since this process of safety aware software development can only be successful in combination with a suitable platform strategy, where the software is eventually deployed, it is also evaluated, how the developed software must finally be built and deployed. In the course of these actions, a practical contribution is made to support the project's progress.

1.6 Boundaries

The boundaries for this thesis have been chosen based on the fact that sophisticated safety activities have not been implemented in the CAP, yet, and the current development focus lies on agile software development.

It is not part of this thesis to evaluate the quality of the agile development process that is enriched with safety engineering activities, but its basic suitability for agile software development is quietly accepted. This is reasonable, since it has been successfully established and is already in use for some time. It is therefore only the goal to enrich it towards product safety. It is also not a goal to provide a comprehensive introduction or a general comparison between agile and plan-driven software development methods, neither in general, nor for the investigated automotive context.

Hardware descriptions play only a subordinate role and are sketched within the platform strategy, only. For all platform considerations we give only an overview of latest technologies and promising approaches, as well as limited practical contributions, while evaluation on application level or concrete platform testing on the hardware or operating system level are not provided. It is also not in focus to explain how to derive test cases, equivalence classes for them or how to develop a testing strategy for the developed software aside from the identified process requirements given in the applied safety standard.

Also not in focus are timing constraints for components to ensure real-time behaviour and their functions respectively, as well as detailed safety activities for guaranteeing safety of hardware components.

There are no measures given to identify or satisfy safety requirements and constraints that consider intentional abuse of the system, and there are no measures given to enable safe production and decommissioning of prototypes or final vehicles.

1.7 Outline

This thesis is divided into the following individual parts:

- *Background* to define the safety scope details, to review normative standards, to discuss related work, and to give relevant definitions.

- *The Agile Development Process at BMW Car IT GmbH* to determine the status quo of the project that will be used to define improvements resulting in the new process.

- *Arguing Functional Safety in Accordance with ISO 26262* to describe suitable methodologies in accordance with the chosen safety scope.

- *Improvements in Functional Safety* to elaborate the new development activities and how they implement chosen methodologies.

We conclude with the summarized insights, provide an outlook and state open research questions.

Since precise wording is important in a safety-related context, an exhaustive Glossary is given, defining important safety-specific terms, in particular those that were found ambiguously defined in literature.

1.8 Remarks on ISO 26262 Citation Style

The analyzed safety standard ISO 26262 is organized in ten different parts. The citation style in this thesis uses the style "PART-CLAUSE" (compare [ISO11a, p. v, all parts]) or "PAGE, PART". The individually used style is unambiguously recognizable.

Examples:

1. "[ISO11a, 1-1.69]" represents clause 1.69 of ISO 26262-1 (i.e., first part)
2. "[ISO11a, p. v, part 2]" represents page v of ISO 26262-2, (i.e., second part)

Note: ISO 26262 part 1-9 and ISO 26262 part 10 are indexed separately as [ISO11a] and [ISO12], respectively. This is due to the guideline character of part 10 and the fact that it is therefore often treated separately. Also, its final edition was released several months after the final release of parts 1-9 and it is treated as a special part of explanatory nature.

2 Background

Autonomous vehicles are complex, safety-critical systems, whose functionality relies heavily on software. In the context of driver assistance systems, they represent the peak of technological advance. To define a suitable development process, we therefore have to deal with these two key characteristics:

- **The safety-critical character:** Malfunctions in an autonomous vehicle can make for a severe threat to occupants and the car's environment, so efforts to prevent them have to be made. To detect hazards causing such malfunctions and to improve safety for the vehicle's autonomous operation, we compare traditional safety concepts, rate their applicability for autonomous driving, and define appropriate safety improvements that can be implemented into the current development process at BMW Car IT GmbH.

- **The need for exploratory development of software-intensive systems:** Systems for autonomous driving still require gathering of experience from prototypes and testing, and experience gain in hardware and software solutions. This means in particular that a set of stable requirements cannot be created for required algorithms and system architecture are still subject to research.

Instead, design changes in the system should be expected in the course of the development and the software development process should lean on agile methodologies.

Safety - A Project Risk

It was observed that safety considerations have to take place early in the development of a product and safety should not be treated as a property that can be integrated in an almost finished product [Kel98, p. 27]. If safety is not considered right from the start, the consequence can be a costly redesign of a mostly finished product, when safety assessments as part of V&V fail [PCGB08, pp. 40]. A safety lifecycle to take care of necessary safety engineering and safety analyses therefore has to be integral part of the ordinary product development process.

Agile Methods in the Automotive Domain

To enable exploratory development, using agile software development has become a noteworthy trend in software engineering lately [BT04, p. xix][Wal11, p. 121]; time-

consuming, excessive documentation obligations and inflexible development processes motivated approaches contrary to traditional plan-driven process models. The "Agile Manifesto" from Kent Beck et al. [BBvB+01] is a famous climax of this trend, where the different ideas have been condensed to a few clear statements, in which frequent releases, changing requirements, refactoring and knowledge-gain during project development are not feared, but embraced and expected. The development in an agile processes happens therefore typically iterative and incremental.

We find this trend of learning-on-the-project and decision making on the basis not only in software engineering, but also in general process and change management, where it is often critized, that developer teams are not encouraged to take responsibility and develop own ideas, but change, requirements and project visions are merely "dictated top-down" by inaccessible authorities [SvS09, pp. 10-16, 37ff].

As a consequence of failed projects under plan-driven development, the unreflective use of heavyweight, non-iterative development models in the traditional waterfall style for industrial projects was increasingly questioned. Established plan-driven methods are often found too heavily regulated and "front loaded", so that a lot of effort has to be put in elicitation and description of potentially unstable requirements. The key differences in the agile development philosophy are therefore frequent preliminary releases that allow to learn from built protoypes and frequent interaction with the customer to provide "rapid values and responsiveness to change" [BT04, p. 26].

The motivation was hence largely triggered by a different attitude towards requirements; while plan-driven models prefer specific, complete and formally baselined requirements [BT04, p. 38], the agile approaches embrace change and changing requirements (even late in the project) and prefer frequent customer collaboration [BBvB+01] and exploratory development. Requirements are communicated and tracked through novel mechanisms such as product backlogs, story cards, and screen mock-ups [JK11, p. 106].

Attracted from these ideas, automotive engineers and the accompanying scientific community started to develop an interest in analyzing the applicability of agile methods to industrial-scale automotive projects. Since these projects underly legal obligations and special duty of care, there are challenges in process design in regard to certification and documentation. This is in particular relevant for projects in the safety domain, where product malfunctions can cause physical harm to users up to fatal accidents.

Outline

We begin with a review of different safety concepts and their scopes, since different normative standards may apply and different tools and methodologies might be necessary

in order to identify and adress relevant hazards.

Next, we discuss well established literature on safety-critical computer systems and scientific publications related to this thesis in order to provide an overview of the topic and to depict the latest state of the art in science and industry; since safety standards are huge and complex documents, it is important to know what to look for and which aspects are critical.

Since the requirement for the process is to be agile, we give an overview of the current state-of-the-art agile development proposals for the safety domain. Primary sources are proceedings to relevant conferences; these are the annual Agile Conference, "Agile", which also, at times, considers safety-critical environments, the International Conference on Computer Safety, Reliability and Security, "SAFECOMP", the Workshop Software Engineering for Secure Systems "SESS", and similar conferences on safety engineering.

Subsequently, we define relevant aspects for process design and the interconnection with model-driven development and the platform strategy based on the literature findings. This prepares the following chapters which elaborate the proposals.

2.1 Safety Scopes for Autonomous Driving

Safety aspects of computer systems can be classified with different categories. For safety-critical embedded systems, two traditionally recited safety scopes are *primary safety* and *functional safety* (compare [Sto96, pp. 4]). Less common is the definition of *safety-in-use* [Krü12].

Figure 2.1: Safety Scopes

There can be different meanings to the terms *safety-related* and *safety-critical* when applied to systems and system parts, where the latter describes systems of high criticality [Sto96, p. 2]. Yet, they are often used as synonyms [Sto96, ibid.]. It is, in general, of no real importance, since they are only informal terms and can as such not be used for the required precise risk quantifications in a safety process.

In case of critical accidents it is important for responsible people to demonstrate safety in a product, because it is a discharge from blame as it demonstrates that potential risks have been minimized to an acceptable level through reasonable measures and hazardous situations have not occured because of gross negligence in product design. This is known as the ALARP principle: all residual risk in a product should be "**A**s **L**ow **A**s **R**easonably **P**ossible", when it is used in its intended operational context.

In the following, we discuss the safety scopes (Figure 2.1) in detail and term the applicable normative standards, if relevant.

Functional Safety

For the automotive domain, functional safety is defined in the ISO 26262 standard [ISO11a] that concretizes the preceeded generic standard [ISO11a, p. v, all parts] IEC 61508 [IEC10], a non-domain-specific functional safety guideline for the development of safety-relevant systems.

ISO 26262 compliance is strongly recommended for automotive manufacturers and suppliers [Fer12], because functional safety in commercial end-user products today is taken for granted [Wal11, p. 1, in German].

The definition of functional safety in ISO 26262 is the "absence of unreasonable risk due to hazards caused by malfunctioning behaviour of E/E systems" [ISO11a, 1-1.51], and *malfunctioning behaviour* being defined as "failure (…) or unintended behaviour of an item (…) with respect to its design intent" [ISO11a, 1-1.73]. We deduce from this that the property of functional safety in automotive E/E systems can be summarized as being *a reliable assertion that the system works as intended.*

Efforts to achieve functional safety are therefore to be concentrated on eliminating unreasonable risk [ISO11a, p. v, all parts], so that a set of individually adequate safety activities can guarantee required reliability in system operation.

Relevance of Functional Safety for Autonomous Driving

Functional safety is a property that is commonly requested in safety-critical embedded systems and cyber-physical systems, and a commonly applied safety scope. Key challenges for achieving functional safety include:

- Handling hardware malfunctions due to wearing effects or environmental stress
- Detecting systematic faults, such as software faults and system design faults
- Minimizing the number of undetected faults by testing and simulation (SIL, HIL)

The distinctiveness for functional safety requirements in autonomous driving lies in the requirements towards the computing platform.

Systems for autonomous driving require a computing platform that provides exceptional computing performance. This is indicated by the expected high throughput of sensor data and complexity of computations. Still, this platform must fulfill safety requirements as traditional ECUs do.

There currently is uncertainty in industry, how such an affordable platform has to look like and how software has to be built and deployed for such a platform to guarantee functional safety. This means, there is also little to no experience which could be re-used based on former, similar computing platforms of such kind.

Safety-In-Use

We define safety-in-use—in contrast to functional safety—as the absence of unreasonable risk due to hazards that can occur although the system does not show malfunctions in regard to the intended behaviour [Krü12] (i.e., it does not necessarily have flaws in the functional safety layer).

Hazardous events in this scope are likely to occur due to high complexity in a system that was not mastered—a typical problem with software-intensive systems—, so they are caused by systematic faults in the intended behaviour that had not been recognized and that remained in the system.

Consequence of this definition is that malfunctions concerning A.I. caused malbehaviour are not in the scope of functional safety, but in the scope of safety-in-use; so is therefore the abstract high-level function “autonomous driving”.

Achieving safety-in-use is a key challenge for autonomous vehicles and it is currently unclear how to reliably argue it (compare [War08, p. 290]).

Primary Safety

Primary safety can address two entirely different safety concepts.

The first definition [Sto96, p. 4] is that of safety measures dealing with physical hazards such as electric shock or burns that are not related to erroneous functional behaviour of the system, but to chemical reactions, radiation, fire, physical design flaws, or the like (compare Figure 2.1).

The second definition [HOO+11, p. 1] includes all safety measures that are implemented to actively avoid the occurence of hazardous situations and accidents. This concept is also termed active safety, in contrast to secondary safety—or passive safety—that only mitigates the effects of an accident during its occurance (compare Figure 2.2). By this definition, functional safety is part of both, primary safety and secondary safety, because

E/E-systems have to function properly before an accident as well as during and after an accident.

The latter definition is less useful in the scope of this thesis, since we concentrate on measures to avoid accidents in the first place and need classifications to refine this class of safety issues. We therefore do not apply the second definition in this thesis.

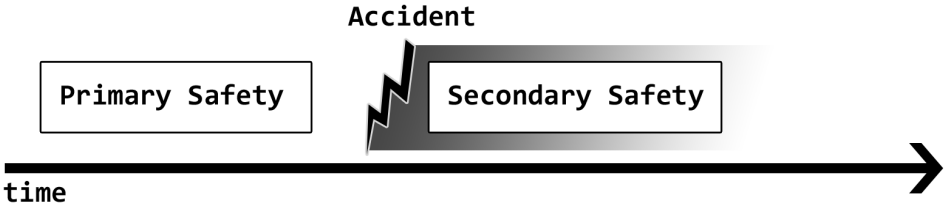


Figure 2.2: Temporal Concept of Primary Safety and Secondary Safety We Don't Apply

2.1.1 Chosen Safety Scope

For this thesis, we concentrate on the scope of **functional safety**, because it is the most relevant safety layer that is not implemented in the current development process, yet. We omit analysis of primary safety, since the implementation of autonomous driving in a car has no special implications on the achievement of primary safety, as it is unlikely that there are any new hazards generated by novel materials or novel electric components that are required for autonomous driving. In addition,—in contrast to the other safety scopes— the concept of primary safety does not apply to software, yet, we are primarily dealing with software development and related safety activities.

Functional safety is arguably a lower layer than safety-in-use; without functional safety there cannot be safety-in-use, because it is impossible to guarantee safe use of a safety-critical system that potentially shows unintended behaviour due to malfunctions in its E/E-systems (Figure 2.3).

Figure 2.3: Safety Scopes as Layers

2.2 Overview of ISO 26262

Before begin the discussion of publications related to this thesis, we briefly give an overview of ISO 26262.

We observe that ISO 26262 describes [ISO11a, all parts, p. vi]:

- an exhaustive **functional safety lifecycle** that is meant to accompany
- a **product development process for E/E systems** [ISO11a, parts 3 - 6],
- **except for business contract negotiation activities**, but including
- short remarks about **production and decommissioning** [ISO11a, part 7],
- optional **supporting processes** and exemplary **guidelines** [ISO11a, parts 8, 10]

The standard defines activities that are to be performed, artefacts that are to be created ("work products"), and requirements for process participants (process roles), based on safety integrity level classifications of the developed items. The focus of ISO 26262 is not completely on activities, since it also thoroughly describes some required artefacts (compare [ISO11a, clauses 2-6.4.3.4ff]), but it has a clear bias to activity-orientation; most work products are briefly described in the beginning, but activity descriptions are given more room.

The product development phase contains a lot of subprocesses, where significant ones are the concept phase where items and hazards are defined on the vehicle level with a derived functional safety concept, the system development level that describes the derived technical safety requirements and the intertwined development process of software and hardware elements, both of which can be run in parallel [ISO11a, all parts, p. vi].

If software is regularly developed in accordance with ISO 26262, a safety management concept, a concept for the achievement of functional safety and a specification of the technical system design has to be defined before the implementation [ISO11a, ibid.].

For the software focussed BMW Car IT company, this has the implication that it cannot run a mere software development process, if development in accordance with ISO 26262 is desired (Figure 2.4).

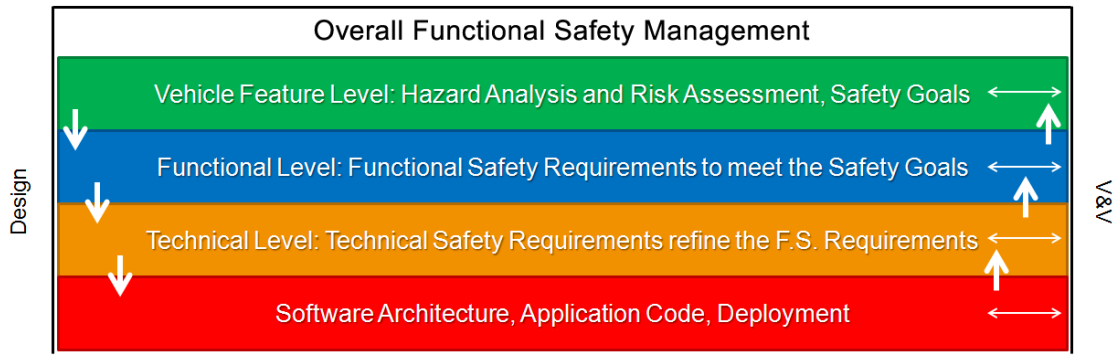


Figure 2.4: Overview of Regularly Developing Software in Accordance with ISO 26262

Tendency Towards Incompatibility with Agile Methods

In contrast to agile methods, ISO 26262 stresses the importance of system design and requirements engineering, in particular as early development activities that are meant to create a precise preliminary description of the system architecture and a detailed requirements specification, which reflects a plan-driven character (compare [ISO11a, part 3] and Figure 2.4).

2.3 Significance of ISO 26262's Release

As the new industrial standard for functional safety in automotive systems, the final edition of ISO 26262 was released in November 2011 [ISO11a, all parts, p. i]. Worked out by a commission of expert authorities, it is now considered *state-of-the-art* concerning functional safety for road vehicles up to 3.5 tons [ISO11a, all parts, p. 1]. A central characteristic of the standard is its risk-based approach [ISO11a, all parts, p. v, b)] to classify system components in five different automotive safety integrity levels (ASILs), the four ASILs A to D and the QM level for quality management of not safety-relevant system components [ISO11a, 3-7.4.4], that determine the recommended safety measures for the identified item.

Though certification of ISO 26262 compliance is not mandatory, it is highly advisable in terms of product liability laws, since it is considered a clear indication of due diligence and exercised responsibility, if the development was performed under state-of-the-art quality standards, such as ISO 26262. Efforts to demonstrate compliance should therefore be performed by OEM manufacturers and all participants of the supply chain (i.e. tier-1 to tier-n suppliers) as of the standard's release date [Fer12], and possibly even as of 2009, when the first draft version was released [HMNS10, p. 7].

When examining ISO 26262, it becomes clear that there is no such thing as ISO 26262 qualification of products, software tools or off-the-shelf components. Products can only be developed in an ISO 26262 certified process and are then potentially qualified to be used in another ISO 26262 aware process, dependent on further requirements for the process integration in accordance with ISO 26262. Summarized, ISO 26262 can be seen as a catalogue of best practices and measures, evaluated for different safety integrity levels that should be applied in a development process [Fer12]. According to Wallmüller, who gives five different concepts of quality notion [Wal11, p. 7ff], ISO 26262 can be classified a *process based attempt* for the largest part in regard to quality management, where safety properties in the developed system are asserted mostly by analytic process activities and descriptions of concrete recommended system properties are rarely found; the product based character [Wal11, ibid.] stays in the background.

2.4 Related Work

To cover general aspects of developing safety-critical systems and to find solutions for the development challenges pointed out in section 2.2, we start with an introductory overview of these systems' characteristics, move on to papers discussing adequate process design and the specific case of developing agile in the safety domain, and close on findings for the state-of-research in autonomously driving prototypes.

2.4.1 Safety-Critical Computer Systems and Functional Safety in General

As a standard reference, Storey's comprehensive book on safety-critical computer systems offers general insights for the understanding of various safety concerns in computer-controlled systems that can possibly harm the user or the environment.

He stresses the importance of safety in all related system components to enable safety for the *equipment under control* [Sto96, pp. 3-7] (compare [ISO12, p. 2]), which includes the four components that form up a (embedded) computer-based control system: Sensors, actuators, hardware and software (Figure 2.5). Hardware or software as unpaired unit cannot satisfy the requirements of safety and assure safe operation, because correct operation of a system is tied to the linkage between hardware and software and achieving system safety requires an integrated approach [Sto96, p.3].

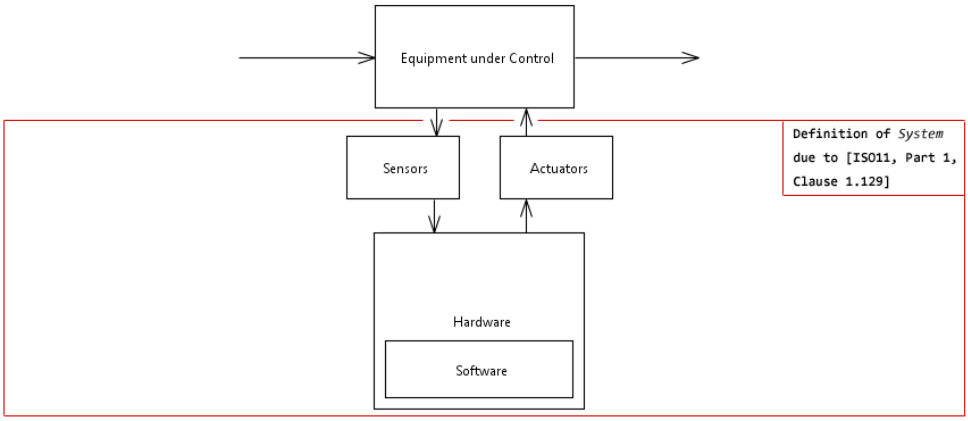


Figure 2.5: Computer-based Control or Protection System due to Storey compared to ISO 26262's definition of the term system

The importance of the term *system* as defined in [ISO11a, 1-1.129] lies in the definition of *item* as defined in [ISO11a, 1-1.69] that is described as "system or array of systems to implement a function at the vehicle level" and the fact, that ISO 26262 is intended to be applied to *items* of the developed product. Storey's definition therefore complies with this definition.

He gives a typical development model that represents a V-Model [Sto96, p. 9] emphasizing the top-down approach for the system design and the bottom up approach for testing. We can observe that it does neither implement iterative nor incremental design, as it does not consider any backward flow or repetition in the development phases, so it is contradictory to agile process design.

We see, however, the trend towards iterative development in V-Model processes, when we compare it to the ISO 26262's V-Model [ISO11a, p. vii, all parts] that was ultimately released 16 years later: It allows far more flexibility in and between development activities and it is actually a V-Model containing multiple V-Models as subprocesses; the software development in ISO 26262 is a V-Model [ISO11a, 6-Figure 2] that can be performed repeatedly within the overall ISO 26262 V-Model development process and that allows repetition of design activities within itself.

An important remark from Horstkötter et al. is that functional safety is ultimately a technical product characteristic [HMNS10, p. 8] (similar in [Wal11, p. 22]). This means, it depends also heavily on the skill and experience of the engineers and not on the process design only, as it is crucial that the activities are carried out competently [Kel98, p. 159]. There are indeed proposals to tackle this by establishing process activity requirements in terms of the engineers' skill levels, which could help to improve on this ("Levels of Software Method Understanding and Use (after Cockburn)" as cited in [BT04, pp. 47ff]).

Remarks on Nomenclature

In the safety domain, it is very important to be precise in wording and ensure unambiguity in communication on all subjects, since ambiguities can lead to flawed—and possibly harmful—designs.

Safety versus Security in Process Design

The distinction between safety and security is particularly important for texts that are not written in English, because in some languages—e.g. German [Ben04, p. 9]— both terms translate to the same word and the individual context has to be considered carefully (which applies to some of the cited papers in this thesis, too).

Although safety and security are completely different concepts, it can be identified that for the development process, they share very similar characteristics; processes have to be enriched with special planning and care taking activities, in particular regarding requirements engineering and testing. These activities are often performed by domain experts (security engineers or safety engineers respectively) and not by regular developers. This means, while keeping in mind the difference, papers for security-critical

system development can also be used to gather ideas for process design of safety-critical systems. Beznosov and Kruchten write:

"The clashes we have identified between the agile methods philosophy and security assurance may not be specific to security engineering only. The certification processes for safety-critical systems (...) lead to similar concerns for the same reasons: an independent inspection process relying on written documentation, and the cost of doing inspection iteratively (each iteration defeating partially what had been assessed previously)" [BK04, p. 50].

Risk versus Hazard

Kletz points out that the term *risk* is used ambiguously [Kle99, p. 5]. For commercial projects, it identifies the aspects that can lead to project failure. This viewpoint is therefore that of a business perspective. It can, however, also adress the chance for physical injuries and damages. The term hazard is unambiguous and describes "a substance, object or situation that can give rise to injury or damage" [Kle99, pp. 5-6].

In ISO 26262, the meaning of the term hazard is very similar, but the term risk was constrained to the "probability of occurrence of harm and the severity of that harm" [ISO11a, p. 15, part 1].

2.4.2 Process Design for Safety-Critical Computer Systems

To understand process design requirements for functional safety, we analyze the context, give the borders and present the most promising ideas to master the identified challenges.

Relationship between Functional Safety and Quality Management

Functional safety is closely linked with quality management [Sto96, p. 3]. It is desirable and mostly inevitable to improve both aspects at once [HMNS10, p. 11]. Process maturity, and hence software quality management, is often explicitly given as part of functional safety (Figure 2.6).

Quality measures within a process can be product- or process-oriented [Wal11, p. 23], which also holds for functional safety assessments [ISO12, p. 7]. A mature process that enables software quality has to provide assessment strategies for both aspects [KB99, pp. 17, 58ff]. For the current CAP, there have been efforts to show compliance with prominent process maturity models like SPICE and CMMI (Figure 2.7). The latest re-

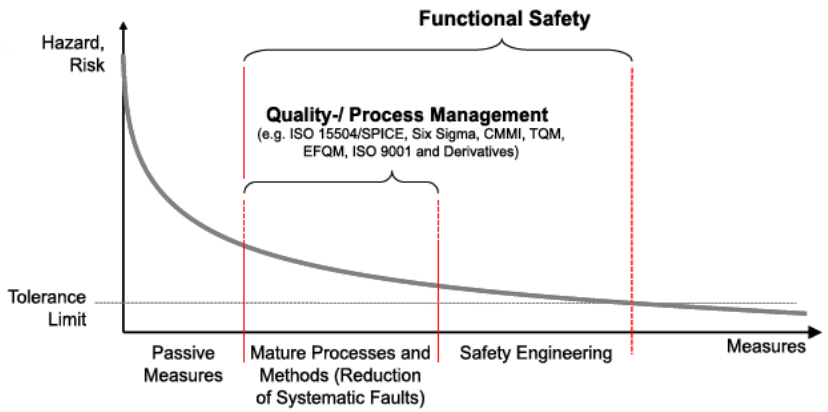


Figure 2.6: Functional Safety and Process Maturity [HMNS10, p. 6, Figure 1 (in German)]

lease, CMMI V1.3, for the first time explicitly mentions agile practices, which reflects the arrival of agile methods in organizations' realities.

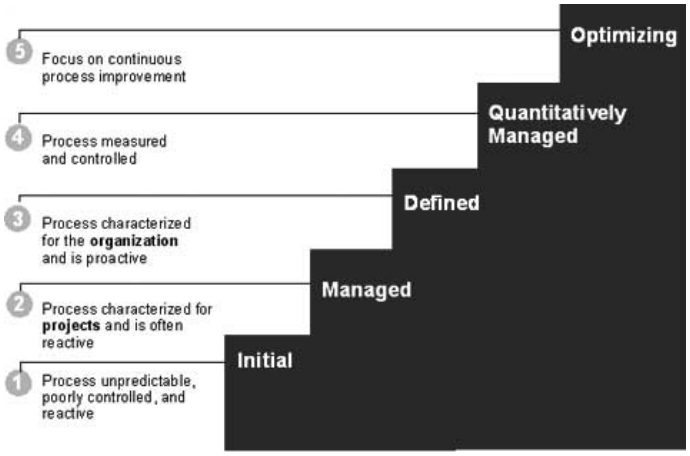


Figure 2.7: The 5 CMMI levels [IBP12]

Indeed CAP was found to be CMMI level 3 compliant [Wik13], it has however not yet been officially certified. This means, it already performs outstandingly for agile industrial processes in terms of process maturity. Although we focus not on CMMI compliance but on ISO 26262 compliance, we have to avoid interfering with CMMI certification improvement efforts.

This is unlikely since we are only enriching the current process and improving its description. This complies with the estimation of Wallmüller, who also states that "CMMI- and SPICE-conformant processes are suitable to be enriched with special ISO 26262 requirements" [Wal11, p. 24, in German]. It is therefore quietly assumed that the CMMI level does not suffer under the introduced changes.

Agile Development in the Safety Domain

As BMW Car IT did for their research project, Paige at al. also found that due to requirements volatility and new emerging technologies, agile methods are "very attractive to software engineers and project managers working in the safety domain", while it also imposes difficulties and challenges [GPM10, p. 36] . Their paper concentrates on using incremental construction of the safety argument to support agile development as they try to find a way to incrementally build safety arguments, so that they can reuse safety arguments from previous releases for producing a safety argument for the current release. This results in *incremental certification*, that is certification of releases without the necessity of building the safety argument from scratch for every release.

This is done with a system safety case that includes a *modular safety argument structure* (Figure 2.8), where the overall safety can be guaranteed based on independent safety modules that are encapsulated from each other based on an *interaction argument*. This enables developers to reuse the safety case of unchanged submodules after an iteration step and thus provides a feasible way to handle the safety engineering activities in an iterative development cycle.

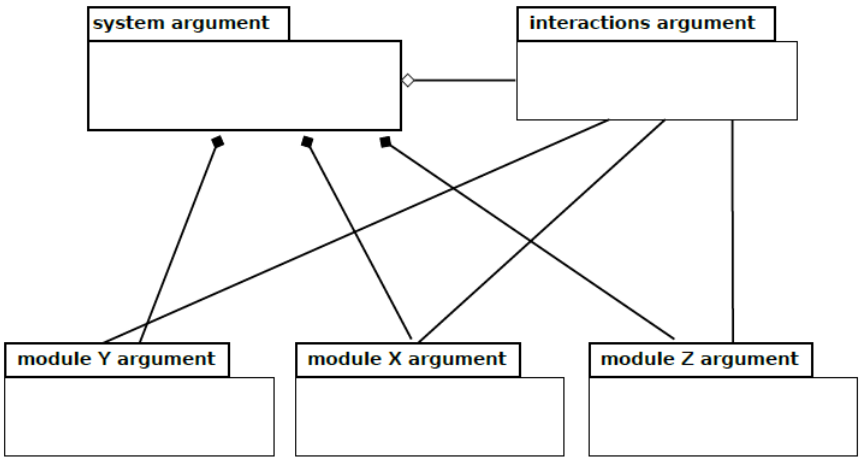


Figure 2.8: Modular Safety Argument Structure [GPM10, p. 39]

This is also indicated in ISO 26262 [ISO12, p. 10]:

"The development of a safety case can be treated as an incremental activity that is integrated with the rest of the development phases of the safety lifecycle."

A critical point in this methodology seems to be the interaction argument, because it is left open, how it is exactly done and how expensive it is to perform. The authors propose the use of safety argument construction patterns [GPM10, p. 40] to tackle this. Also, this procedure only supports incremental design, but not iterative design.

On the project scale, they propose a light-weight up-front design before the iterations phase, that at least produces a coarse-grained system architecture model that at least

identifies the most important components. This would allow for a first hazard analysis that is sufficient to give an argument for the overall system safety and component independence, which is then incrementally and iteratively developed in the course of technical system development.

The presented successful application in an avionic project generally indicates the feasibility of the proposal, albeit the description of the process' agility often remains vague, and the authors finally conclude "that [their] process has the potential to be an Agile [sic] process".

In a first attempt to apply Xtreme Programming in the safety domain (for High-Integrity Systems), Paige et al. [PCMS05] proposed four additional activities in the course of regular XP to take care of safety requirements. They term these activities:

- **Safety Process**, where explicit safety consideration take place
- **Static Analysis** satifying special safety requirements through tool support
- **Process Risk Management** for risk mitigation concerns
- **Design Representations**, meaning easily understandable models

In a subsequent paper, Paige et al. [PCGB08] further developed these ideas and applied them in a case study. They propose a method of both incrementally developing and certifying safety-critical systems through what they call *HIS-XP* (XP for High-Integrity Systems). This method is capable of capturing evidence of compliance to standards: They introduce *safety stories* together with a suitable *safety story card* that captures hazard-related information that has been identified in a safety analysis iteration step, where a domain expert identifies hazards and answers developer questions. This essentially means, they try to keep away the software developers from safety planning activities and leave them open to safety engineers.

The paper's other core idea is *pipelined iterations*, meaning to group activities of different *engineering levels* in a pipeline, where certain groups are inspecting results of previous iterations. This is feasible, because for high-integrity system experts comprising from different domains have to work together on different aspects of the system, and respective activities have to happen in a certain order (Table 2.1).

The table shows that test-driven development (TDD) is the implementation technique, which is not a requirement for ISO 26262 compliance, but also not a confliction. They do not cover white-box tests, such as static verification or code coverage testing, since they find them not to be adressed by agile processes due to their complexity.

They also present the ideas of [GP06], where it is proposed that developers are allowed to perform two different kinds of iterations: a minor one that only increments the functional side of the product and a major one that also considers safety requirements. Both release types are, however, only considered *conditionally safe* until a final acceptance

Story Engineering and Planning	Plan N	Plan N+1	Plan N+2
TDD, Integration	Develop N-1	Develop N	Develop N+1
V&V	Verify N-2	Verify N-1	Verify N
Safety analysis	Assess N-2	Assess N-1	Assess N
Safety case development	Argue N-3	Argue N-2	Argue N-1
Evaluate and adjust	Feedback from N-2	Feedback from N-1	Feedback from N
Iteration	N-1	N	N+1

Table 2.1: HIS-XP with Pipelined Iterations [PCGB08]

test has been completed. This means that a prototype release cannot be declared safe for testing, until some possibly *expensive* finalizing extra activity has been performed after the course of the regular iterations.

The verdict of the paper is somewhat alarming, since the authors found that their incrementally carried-out safety planning frequently invalidated previously made design decisions in their case study [PCGB08, pp. 37-42], resulting in a waste of manpower. They point out that this problem is inherent and can therefore only partially be avoided through tool support. As a result, the authors' final conclusion is that agile methods should complement plan-driven methods where necessary, and they propose the development of a "framework of procedures" to create a hybrid process, which can adapt to the needs of project's current requirements. Regularly updated "high-level architectural plans" should support the understanding of the developed system and motivate process tailoring by risk-based techniques.

A similar, *balanced approach* with aspects of plan-driven processes was presented by Boehm and Turner in which they associate plan-driven methods with discipline and agile methods with the opposite:

> "Discipline creates well-organized memories, history and experience.
>
> Agility is the counterpart of discipline. Where discipline ingrains and strengthens, agility releases and invents.
>
> - Boehm and Turner [BT04, p. 1]"

The foreword of Cockburn [BT04, pp. xvii], however, points out clearer than the authors that this is not uniformly correct, since agile methods could also demand high levels of discipline. Cockburn mentions XP as an example.

They pronounce themselves sceptical about the overall appropriateness of purely agile methods in the safety domain [BT04, pp. 20, 22] or for very large and complex projects [BT04, pp. 28, 150] which they term not a *home ground* for agile methods.

They propose a combined approach of conventional plan-driven and agile methods and present a *risk-based project development strategy* [BT04, pp. 99 ff, chapter 5] that balances *rigor and flexibility*, but does not give concrete definitions; a risk analysis chooses for each iteration of a spiral-model the preferable abstract increment strategy - agile or plan-driven [BT04, p. 101, figure 5-1].

As an important cornerstone for project success they identify the *CRACK stakeholder*, which is collaborative, representative, authorized, commited and knowledgeable [BT04, pp. 44-45]. Decisions worked out with such stakeholders are reliable and support the creation of a dependable requirements set.

They finally give six conclusions [BT04, p. 148] of which (1) and (5) are especially interesting and relevant for our process design:

- (1) Neither agile nor plan-driven methods provide a silver bullet.
- (5) Is is better to build your method up rather than to tailor it down.

We draw from this that our agile process needs to be flexible in its level of *discipline*, which means, it must be able to adjust the share of manpower between documentation and planning activities and exploratory coding activities, within the borders that ISO 26262 allows. Risk-based planning with ability to completely suppress agile development is not incorporated, since it is contrary to our research question.

In a paper that addresses both the safety and the security domain, Beznosov and Kruchten [BK04] give four *pain points*, where they find that security assurance—and in the context of the paper, safety assurance (compare 2.4.1)—practices clash with agile development:

- **Tacit Knowledge** practices and lean documentation.
- The **Iterative Lifecycle** that makes it expensive and adds to the development time by frequently having the demand to integrate third party participants for domain-specific activities that cannot be performed by the software developers.
- Code **Refactoring**, where they find the same as we already reviewed, i.e. that frequent refactoring may be difficult because of the special domain requirements of the security and safety domain regarding architecture decisions.
- The **Testing Philosophy**, where the agile paradigm of early testing lacks of understanding, how thorough the testing strategies are. We find this plausible, because in agile development testing often seems to be very focused on unit testing.

Their solution is to classify development activities into ones that can be automated or semi-automated, and those that can not [BK04, p. 49, Table 1]; those that can, should

be implemented lightweight in execution, so that they can be seemlessly integrated in the iterative workflow without delaying the development cycle. For the others, there is again proposed to start the process at the very beginning with a coarse-grained analysis of what to consider, and another one "closer to the end", when the product is heading towards its release.

Interim Result

Incrementally constructing a safety case is hard to do, iteratively maintaining it even harder. A pipelined process might need high process discipline, because it must always be defined, which team works on which version of what artefacts. For a modular safety case construction that needs no pipelining, it is, however, not clear how to construct the interaction argument over the components' mutual impact. The immediate capturing of safety concerns can be done with adapted agile techniques, such as safety story cards.

After we have now given an overview over general agile process design ideas and concerns for the safety domain, we proceed to inspect current proposals for the concrete safety management activities and how to implement them taking account of the previously found requirements. This means, we need concepts, methods, and tools that are suitable to skillfully construct, manage, and re-use safety cases.

2.4.3 Safety Cases, Functional Safety Planning and Management

ISO 26262 directly cites the ideas of Kelly (Figure 2.9), whereas a safety case that argues the system's safety consists of three principal elements: The safety goals, the safety argument, and the produced evidence, where the argument is the link between the other two.

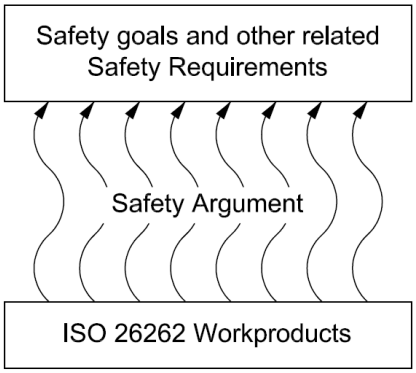


Figure 2.9: Safety Case [ISO12, part 10, p. 9, Figure 6], compare [Kel98, p. 25, Figure 1]

Kelly focuses on the safety argument and addresses the incremental construction and maintenance of a safety case as he states: "Historically, the production of safety cases has

often been viewed as an activity to be completed towards the end of the safety lifecycle. However, it is increasingly being recognised that in order to gain most value out of the safety case, and to present the most convincing argument, safety cases should be developed incrementally in step with system development".

To enable this, he presents "a method and a graphical notation for the presentation of safety arguments" [Kel98, p. 29] he calls the "Goal Structuring Notation" (GSN, [Kel98, pp. 54 ff]), and is able to show that is it qualified for safety argument development and maintenance, and in particular safety case reuse after changes [Kel98, pp. 159 ff]. He also gives an **e**xtension of GSN, EGSN, that enriches GSN for expressiveness in structural abstraction [Kel98, pp. 167 ff].

These proposals and ideas from Kelly, although from 1998, were very influential, not only directly for ISO 26262, but also for recent publications, and eventually found their way into tools, e.g. in 2012 Denney et al. [DPP12] presented AdvoCATE, an Eclipse RCP for extended GSN modelling.

Hillenbrand [Hil12] focussed on the concept phase of functional safety planning in accordance with ISO 26262. As we have seen so far, this was uniformely claimed to be a crucial activity, as weaknesses here can trigger very complex and time-consuming refactorings, when after the coding activities it is found that the implementation does not satisfy later detected safety requirements.

The paper discusses the architecture description language EAST-ADL2 (Figure 2.10), a UML2 profile [ATE10, p. 1] which's metamodell uses the concepts of the AUTOSAR metamodel [ATE10, p. 17]. Hillenbrand [Hil12, pp. 104f] finds that EAST-ADL2 is capable of supporting safety aspects and ISO 26262 requirements through its language-defined elements.

He states that hazard and risk analysis as well as safety goals (in accordance with ISO 26262 part 3 §7) are covered in the vehicle level, specification of functional safety requirements (ISO 26262 part 3 §8) in the analysis level, specification of technical safety requirements (ISO 26262 part 4 §6) and specification of system design (ISO 26262 Teil 4 §7) in the design level [Hil12, p. 105].

For practical appliance he recommends the tool Eclipse Papyrus and the according EAST-ADL Plugin that claims to be EAST-ADL2-compliant [CEA13]. However, in general, every UML tool is able to support EAST-ADL modelling, since it is only a UML profile and can therefore be realised with stereotypes.

EAST-ADL2 has also been found to be highly recommendable for integration in a model-based system engineering context by Chen et al. [CJL+08], which also found, that it "enables fulfilling many requirements on software development as specified by ISO-CD-26262" [CJL+08, p. 73].

A solution to implement simplified but consistent fault analysis [Hil12, p. 120], that is

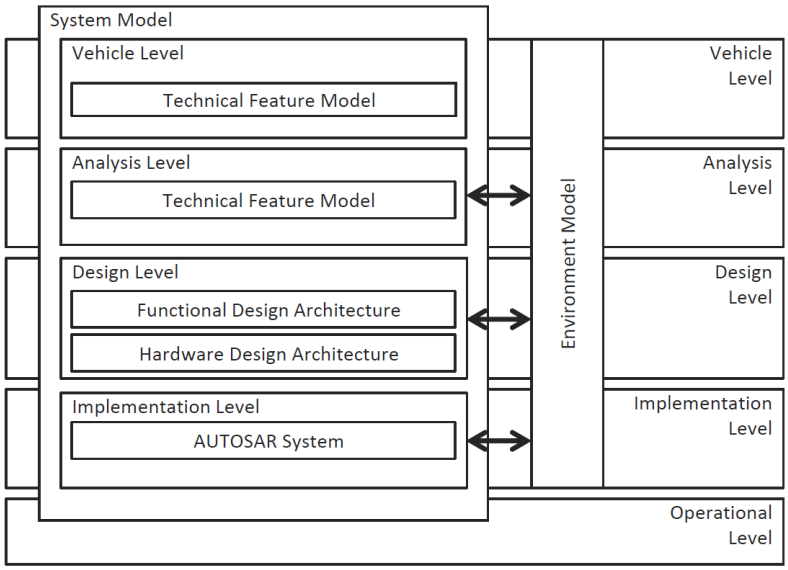


Figure 2.10: EAST-ADL2 layers by ATESST2 as cited in [Hil12, p. 103]

also compatible with EAST-ADL2 models [ATE10, p. 152], are *HiP-HOPS*: "Hierarchically Performed Hazard Origin and Propagation Studies" after Papadopoulos and McDermid [PM99]. HiP-HOPS implement the safety inspections FTA and FMEA that are recommended as general-purpose safety analysis techniques multiple times in ISO 26262 [ISO11a, 3-7.4.2.2, 5-7.4.3][ISO12, Annex B]. The central idea behind HiP-HOPS is to partially automate these usually manually performed safety inspections by generating fault trees out of an enriched system model in a step termed Fault Tree Synthesis [Uni12a, p. 9]. The only manual activity is therefore the creation of a system architecture model, where additional failure data has to be annotated. By combining FTA and FMEA into one integrated method, inconsistencies are avoided [PM99, pp. 139-140]. HiP-HOPS was first implemented in the Safety Argument Manager (SAM) [PM99, p. 139] and is now commercially available as a tool developed by the University of Hall [Uni12b] under Papadopoulos and the dependable systems group. The main requirement to perform HiP-HOPS in the original tool is therefore model-based development in an environment that is able to export models into the tool's open XML format [Uni12c].

Hazard Identification and Risk Assessment

Kletz developed HAZOP and HAZAN as a systematic approach to hazard identification and risk assessment. HAZOP is performed for the identification of hazards and HAZAN for their assessment, so there is a clear separation of concerns. However, both terms are often confused or used incorrectly [Kle99, p. 3].

Relying on an expert team with specific domain knowledge and expertise, the *HAZOP team*, and a set of *guide words*, HAZOP's aim is to reduce the chance that hazards remain

unidentified [Kle99, p. 8]. To detect hazards, the guide words are applied to system operations and the result is evaluated.

When HAZOP was performed, HAZAN is applied to the identified hazards in order to quantify probability and consequences of their occurence if possible, and to decide whether or not action has to be taken on the hazard [Kle99, p. 80].

Another important point Kletz makes is that safety experts to perform safety activities should work together with developers and share their knowledge, help and encourage developers to perform safety activities, but not perform safety activities isolated from regular developers [Kle99, p. 165]. He assumes that most developers are able to understand and learn safety analysis and assessment techniques relatively well under such circumstances [Kle99, p. ibid.].

Although HAZOP and HAZAN was originally designed for plants and the chemical industry, it attracted safety engineers beyond that scope. Pumfrey adapted HAZOP for the software domain as SHARD and performed four major case studies with mixed results [Pum99, pp. 25, 129ff.]. SHARD proposes a set of software domain guidewords [Pum99, pp. 131,132]. In the case studies, the approach was found to be effective [Pum99, p. 146], but time consuming [Pum99, p. 146], since it cannot be automated, and in one case, a system design change required a complete re-evaluation of a largely completed analysis [Pum99, p. ibid.].

To improve on this and to provide an exhaustive guideline for Hazard Analysis, Denger et al. [DTL08] provided the *SafeSpection* framework that allows for execution of a systematic hazard analysis. Within the framework, they propose the methodical finding of software faults through guide words and moderation, like SHARD does, but for three different logical levels of the software architecture, where meta-meta questions, meta questions and concrete questions about the system are discussed.

This approach works as a process framework, because it uses a SHARD-like approach to trigger other activities like FMEA and FTA, so it does not describe any new safety inspections, but consolidates the “whole picture” by linking the hazard identification and analysis activities. In their application results, the authors claim SafeSpection to be feasible and effective [DTL08, p. 56], but the evaluation is limited in extent.

Component Based Software Engineering and Interference Management

As the last proposals indicated, it is necessary to argue on interferences, if we want to reuse evidence that certain system components are still safe after modifications on other components that do not compromise their respective safety case.

As a way to handle component based software engineering in the safety domain, Domis

and Trapp [DT08] propose a Safe Component Model (SCM) that allows to conduct FTA and FMEA for different logical layers of a system. This method is already implemented in the tool ComposeR.

Building on that, the proposals of Graydon and Kelly [GK12] can be applied, where once again guidewords in the style of HAZOP and SHARD are applied to detect possibly conflicting subsystem functions and to identify respective components. They propose the already introduced GSN to model requirements in terms of freedom from interference, which demonstrates the flexibility of GSN.

Handling Automation: Tool and Toolchain Qualification Opportunities

At the moment, there is no unified toolchain available to support all requirements of ISO 26262 [Fer12] and single tools that are qualified might be expensive or hard to apply. However, automation of required process activities is a key challenge for complex agile processes: It keeps away recurrent and bothersome obligations from software developers and avoids deceleration of the iteration cycles.

A big concern on process automation is the reliability of the used tools and tool-chains. *Tool* hereby not only refers to development tool (i.e. a tool that produces or assists in the production of executable program code), but to every tool that is part of the overall development process, e.g., calibration tools or applications to manage requirements [Kri12, p. 233].

The confidence level is based on the probability of the tool or toolchain that "erroneous output can introduce or fail to detect errors in a safety-related item or element being developed" [ISO11a, part 8, p. 20]. This means in particular that when it is shown that effects of a tool are all isolated from safety-relevant items or elements ($TI1$), it has automatically the lowest confidence level ($TCL1$) which means, it needs no qualification ($TI1 \Rightarrow TCL1$). This is likely to be the case for bookkeeping and accounting tools, and but not limited to similar tools that are not directly involved with the concrete development process.

There are two ways to satisfy ISO 26262 requirements for tools that were identified to require a higher confidence level (i.e. TCL2 and TCL3). The one is tool qualification, where an argument for the tool confidence has to be given. This is usually hard to do and expensive, and therefore not always preferred. The other is to modify the related process context, i.e. extending it with a follow-up process activity, where the tool output is reviewed [ISO11a, part 8, p. 24-26] either manually or through comparison with output of alternative tools. The case of output file comparison (e.g. log files) is itself a prominent case for support through tool automation, for which many tools are available. While this

saves tool qualification effort, it on the other hand introduces process complexity.

To get the right mixture between the two approaches, Slotosch et al. [SWP+12, p. 32] introduced a domain model for toolchains ([SWP+12, Figure 3] and implemented it in a toolchain qualification kit. They applied this to an industrial case study and were able to reduce the TCL to 1 for all but one of 37 tools [SWP+12, p. 36], while seemingly holding the process complexity at a tolerable level.

2.4.4 Status Quo in Prototyping

To round out our reflections and to present implemented safety activities, we consider Wardzinski [War08], who gave concrete applications of safety assurance strategies for autonomous vehicles and analyzed participants of the 2007 DARPA challenge [DAR07], which is the most recent one of three, as of now. In this contest, vehicles had to navigate autonomously between abandoned military barracks in a fenced-off environment and obey traffic rules. These conditions were more realistic than in previous challenges that took place in wastelands.

He compares two different approaches: a traditional hazard analysis and safety assurance, where system safety is achieved by implementation of *safety barriers*, and *dynamic risk assessment*, which is novel for safety-critical systems.

The safety-barrier-based approach is intentionally simple, but a meanwhile mature and controllable methodology. It considers, where it is safe to go and where it is not. Different subclasses of barriers can be defined in order to model the environment, e.g. functional barriers, symbolic barriers, and immaterial barriers. The author concludes that this might be enough for "simple" situations and environments, and hints that there were self-driving cars in the DARPA contest that used this approach. He, however, questions, that this approach holds for a more complex environment (e.g. with pedestrians).

For "more complex and less controlled environments", he proposes dynamic risk assessment. Systems capable of this would not think in a binary way and only plan in trajectories where they can go and where it is forbidden, but they would rate the actions that can be performed in a graduate *risk scale*, based on the environment situation. He however concludes that this method is still subject to research and it is unclear how to obtain a valid safety argument for it, as it is not straightforward.

2.4.5 Conclusion on Related Publications

Proposals that were not evaluated thoroughly enough or elaborated too vaguely should not be directly implemented in the productive development process. This holds in particular for those that were not evaluated in larger scale projects, yet.

In contrast, mature state-of-the-art methodologies that can help to improve functional safety are:

- HAZOP for systematical hazard identification of conceptual safety requirements
- SHARD for systematical hazard identification of software safety requirements
- GSN for graphical and reusable construction of a safety case
- EAST-ADL2, a UML profile that implements the domain language of ISO 26262

If the software development for a safety-critical project can make use of component-based software engineering, it should be considered to integrate HiP-HOPS into the development process; this requires to create a failure-probability annotated component model, but enables to perform automated FTA and FMEA.

Assignment of Discussed Methods to ISO 26262 Safety Lifecycle Phases

Each identified method can strengthen certain aspects of product development in accordance with ISO 26262. We can assign them to different parts of the standard as follows:

- HAZOP, or a HAZOP-like approach with modified guidewords, as a systematical approach to identify hazards due to malfunctions on the vehicle level [ISO11a, part 3].

- SHARD as a systematic software hazard identification technique that helps to derive software safety requirements from the technical safety concept [ISO11a, parts 4, 6].

- GSN to argue functional safety, wherever safety goals or - requirements and their solutions are to be documented, and traceability is to be established. This applies for:

- the concept phase, where the functional safety concept is created [ISO11a, part 3]
- the system development phase, where technical safety requirements are defined [ISO11a, part 4]
- for hardware and software safety requirements derived from the technical safety requirements [ISO11a, parts 5, 6]

- EAST-ADL2 can be used to establish traceability across all ISO 26262 design layers, from item definition, to hazard analysis and risk assessment, to the functional safety concept, to the technical safety concept, down to hardware and software architecture. It therefore covers and links the parts from the concept phase to the development of software and hardware [ISO11a, parts 3-6].

Decisions Reached

Based on the literature findings and in coordination with BMW Car IT, we identify potentials and risks of the identified methods.

HAZOP, SHARD and EAST-ADL2 are mature methods, but time-expensive to apply in the process and counter-agile; HAZOP or SHARD cannot be performed efficiently in an iterative process, and EAST-ADL2, as a graphical modelling language intended to describe the entire system in development, is inherently not an agile method. It is, however, possible that selective cut-outs of the system or early up-front planning modelled in EAST-ADL2 can help to avoid too excessive refactoring, although it remains vague to which extent and in which scope this should be performed withouth having long term experience.

This methods are therefore primarily suitable for plan-driven development, and do not fit well in agile development processes.

GSN, however, has the potential to be integrated into an agile development process and can help to begin the integration of a maintained safety case; it is comparably easy to use and understand, and establishes a clear structure of which hazards have already been identified and dealt with, how it was done, which safety measures have been implemented, and to argument why they fit. This satisfies the requirement of ubiquitious clarity in the safety argument as required [ISO12, 5.3]. The possibility to *challenge* existing GSN models allows the use in iterative development and provides a mechanism to generate documentation of the safety case's evolution.

2.5 Definitions and Concepts

Development processes and software development play a central role in this thesis. We give definitions of the important terms.

Definition: Generic Process

A generic process describes the entirety of characteristics and properties of a process class; it is the abstraction of all real-world processes of a process class [Lan10, p. 1].

Generic processes are *tailored* to fit a specific project's requirements (Figure 2.11).

Generic processes are meant to be a template for the documentation of a tailored real-world process [Lan10, p. 2].

The process descriptions in ISO 26262 are generic, since they contain activities, whose existence in the specific development process is dependent upon the developed items' safety integrity requirements.

Important elements to define a process include [KMFK13, p. 138][FR12, p. 185]:

- Roles: Description of process participants (e.g., stakeholders)
- Activities: A set of actions that have to be performed

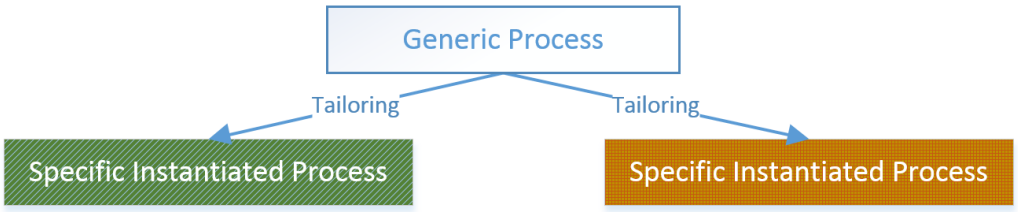


Figure 2.11: Tailoring Generic Processes

- Actions: Atomic activities, “things to do”
- Subprocesses: A set of process activities, that belong together
- Artefacts: Materialized products of process activities
- Gateways: Process-flow control elements
- Events: Trigger for the process flow

Definition: Product Development Process

A product development process consists of all subprocesses that have to be performed prior to manufacturing of the product can start. This includes business contract negotiation, product development process, and definition of the product manufacturing process [Lan10, p. 1].

As stated in 2.2, ISO 26262 describes a safety lifecycle that is meant to accompany the whole product development process, except for contract negotiation activities.

Definition: Software Development Process

A software development process is a development process for creating software products.

Plan-driven software development processes include well-documented requirements engineering, software architectural design, implementation, testing and verification activities (compare [ISO11a, p. 5]).

Agile software development processes tend to disregard activities such as architectural design for organic software evolution that is more frequently tested against prototypes; the coding and learning from experience on the project is stressed as the most important aspects of development [SM01, p. 3].

Definition: Safety Lifecycle

A safety lifecycle is the entirety of all activities to create and maintain the artefacts that document the developed product’s safety [ISO11a, 1-1.72, 1-1.89, 1-1.104].

The ideal safety lifecycle is initiated before the development to determine roles and process tailoring, and integrates then seamlessly with the development process.

Definition: Safety Case

The safety case is the "argument that the safety requirements for an item (...) are complete and satisfied by evidence compiled from work products of the safety activities during development" [ISO11a, 1-1.106], and thus, "the argument as to why the developed system is believed to be safe to deploy in its intended operational context" [WMK02]. It is initiated, maintained and continuously expanded in the safety lifecycle, until a final version is complete for the final product.

For development in accordance with ISO 26262, the creation of a safety case is mandatory "for items that have at least one safety goal with an ASIL (A), B, C or D" [ISO11a, 2-6.4.6.1]. For autonomous driving, the creation of a safety case is therefore mandatory for most items, since the expected ASIL is D on the item level for most items involved.

The safety case should be treated as a living document and be developed seamlessly along the product development, in at least three incremental steps [Kel98, p. 69], [ISO12, 5.3.2]:

1. Preliminary Safety Case: After the initial requirements specification
2. Interim Safety Case: During implementation
3. Operational Safety Case: Prior to completion of the system

The safety case typically references safety concerns, activities, or associated artefacts such as—but not limited to— [Kel98, p. 23]:

- Safety arrangements and organisation (roles in the safety process)
- Safety analyses
- Compliance with the standards and best practice
- Acceptance tests
- Audits, inspections, feedback
- Provision made for safe use

The safety case is only complete, if all involved elements also exist as real artefacts and not as mere references. The safety case therefore is no single artefact, but consists of a multitude of artefacts. This in particular includes results of tests and evidence of process compliance through documentation of performed activities.

Unfortunately, **it is common to use the term safety case synonymously with safety case report** [Kel98, p. 24].

Definition: Safety Case Report

The safety case report (also: safety report) equals the piece of documentation that presents the safety case [Kel98, p. 24]. It only references the artefacts the safety case is compiled from without taking care that they actually exist. The safety case report is part of the safety case.

Definition: Traceability

Traceability ensures that a taken step in building the product can be traced back to the statement in the design that made it necessary; the implementation of the product has to be consistent with the requirements specification, or the wrong product is built [Ste87, p. 181]—in case of ISO 26262 safety requirements, bidirectional traceability is required, which means that high level source requirements have to be traceable to lower level requirements and lower level requirements have to be traceable back to their source [PS11].

To establish traceability from the requirements specification to the implemented features, usually a path of linked elements has to be followed, since traceability is a transitive property.

Definition: Automation

Automation is the use of non-human process participants to complete human tasks (i.e., manual tasks) in a process [UC408, p. 2].

Examples for non-human process participants are software tools or process engines.

Definition: (Software) Build Process

The build process is all activities involved in the conversion of software source code into a stand-alone form that can be run on a computing device [Jan13a].

A software build is hence a version of the software that can be run (i.e., creates artefacts that are executable code) [Jan13a]. It is intended for testing or final release.

The process steps a build process incorporates are dependent upon the chosen programming language and typically involve activities such as compilation or linking (if applicable). More complex build processes also cover testing and automatic documentation generation from source code annotations.

Definition: Deployment

The deployment process incorporates all activities that are necessary for placing software in its operational environment and making it ready for use [Hey08, p. 1]. It is a function that maps software to hardware [Sch08, p. 23]. A typical hardware meta-model in the automotive domain incorporates the elements ECU, bus, bus-controller, sensor and actuator (compare: [Sch08, p. 39]).

Deployment systems usually organize software in modular encapsulated parts (i.e., components). To deploy a software component, it has to be configured and its dependencies on other components have to be satisfied [Dea07, p. 269].

For process design, software deployment "may be considered to be (...) a number of inter-related activities including the release of software at the end of the development cycle[,] the configuration of the software, the installation of software into the execution environment, and the activation of the software" [Dea07, p. 269].

Definition: Platform

We define a platform as the set of all technologies and system elements that are necessary to run application software. This includes hardware that in particular provides the necessary performance to run the software and software layers the application software depends upon [Jan13b].

Definition: Software-Intensive System

Software-intensive systems—such as a system for autonomous driving—are "complex systems where software contributes essential influences to the design, construction, deployment and evolution of the system as a whole" [ISO11b].

Business Process Model and Notation (BPMN)

BPMN is a flow-oriented modelling and notation specification originally developed for business processes. It is, however, useful for any kind of workflow management where message flows, subprocess modelling and interactions between different domains are of interest. BPMN shares similarities with UML activity diagrams and is maintained by OMG. A central principle of BPMN is the directed graph structure of diagrams and the modelling of control token flows. Nodes of the graph are events, activities or gateways [FR12, p. 21]. Events create and consume tokens, gateways clone or merge them, activities hold them. When the last token is ultimately consumed, the process terminates. A process can be instantiated and is alive when a tokens exist, and dead if no tokens exist

anymore. This concept shares many similarities with petri nets and there are proposals to map BPMN to petri nets in order to enable petri net analyzer tools for validation of BPMN processes [DDO07][RPU+07].

The concept of subprocesses and abstraction allows to model different process levels and hence is a good way to separate concerns and to contextually hide irrelevant information.

BPMN provides a semi-formal graphical modelling specification, an XML source code specification, and an operational semantics definition. It supports model checking techniques for automatic validation and the opportunity to execute models in a process engine. The execution of a process means that, in the course of a workflow, activities have to be completed in order to drive the process forward. An activity can be completed by a human or a non-human process participant.

The specification is very detailed about shapes and functionality for concrete tool implementations. It defines important features, such as subprocess modelling [OMG11, p. 386], and gives icon and form descriptions. This creates a homogeneous feeling within a BPMN toolchain.

3 The Agile Development Process at BMW Car IT GmbH

We analyze and give proposals to improve an already established agile development process currently in use at the BMW Car IT GmbH, the CAP. We analyze the presence of quality management activities that already fulfill functional safety requirements and try to map the ISO 26262 safety lifecycle on the process. The result is the basis to add new safety activities in chapter 5.

3.1 Agile Development for Autonomous Driving

Systems for autonomous driving require more software than any other ADAS before and cannot be built upon earlier experience with similar systems, because they also require new hardware platforms. That means, at the current time, the exact requirements are hardly forseeable, as are the technical solutions that should be applied—both in terms of software and hardware—, so development requires a process able to expect changes and allow agile methods.

The other consequence is that the development process is for the largest part an "isolated software development process", since hardware modifications will occur less frequent than software modifications.

3.2 The CAP Reference Process

The reference process considers quality management, but does not implement an explicit safety lifecycle. The CAP is documented through informal descriptions in mostly natural language and a few graphical illustrations. The creation of mandatory artefacts is supported by templates and artefact examples. The process can be run as CAP-S with enhanced quality management to prepare development of systems for **s**eries maturity.

To create more confidence in the analysis, we choose BPMN to capture the process. The clarity of a BPMN model can also help to maintain the process, as coping with large amounts of natural language often becomes tedious and error-prone over time.

BPMN is useful to model processes at different levels of detail, which provides good flexibility; a useful distinction might consider sophisticated process flow for input in a process engine at the lowest level , detailed descriptions of operative process flow for different participants at the medium level, and a strategical overview at the highest level [FR12, p. 120, Figure 3.1].

The CAP in BPMN

For choosing the model's scope and granularity, we orient on the ideas in [FR12, pp. 139ff] and capture the CAP on level 1 to define a strategical overview, where we can determine the right place to insert the new safety activities and provide a basis for the team to discuss the process and the mapping of the ISO 26262 safety lifecycle on it. Important subprocesses can be elaborated in a refined cut-out, in the future.

The CAP consists of three core phases (Figure 3.1) we capture in BPMN lanes (for the strategical overview this is justifiable, whereas for more fine-grained levels, lanes should be used to model roles). The first and last phases are non-iterative and include business and strategic planning activities. Only the second phase implements the iterative development cycle. The first phase not only incorporates business activities, but also activities to create the first coarse-grained design and architecture assumptions, exploration reports, and project risk analysis.

3.2.1 Observations about the CAP

The CAP is a software development process. It requires *discipline* from the participants: It does not abstain from requirements engineering and documentation. Developers are encouraged, however not required, to perform quality management and give rationales througout various stages of the development cycles. This is an advantage, because safety activities can be integrated easier without changing the process too much and achieving better acceptance among established process participants.

Artefacts are to be maintained in the designated storage locations that have been initialized at project start, where artefacts can be stored and versioned minimizing the risk of loss or irrevocable damage.

3.2.2 The ISO 26262 Safety Lifecycle Mapped on the CAP

Without heavy modifications, the CAP cannot perform a complete product development process where ISO 26262 can be mapped on, since CAP's nature implies the focus on software development (Figure 3.2).

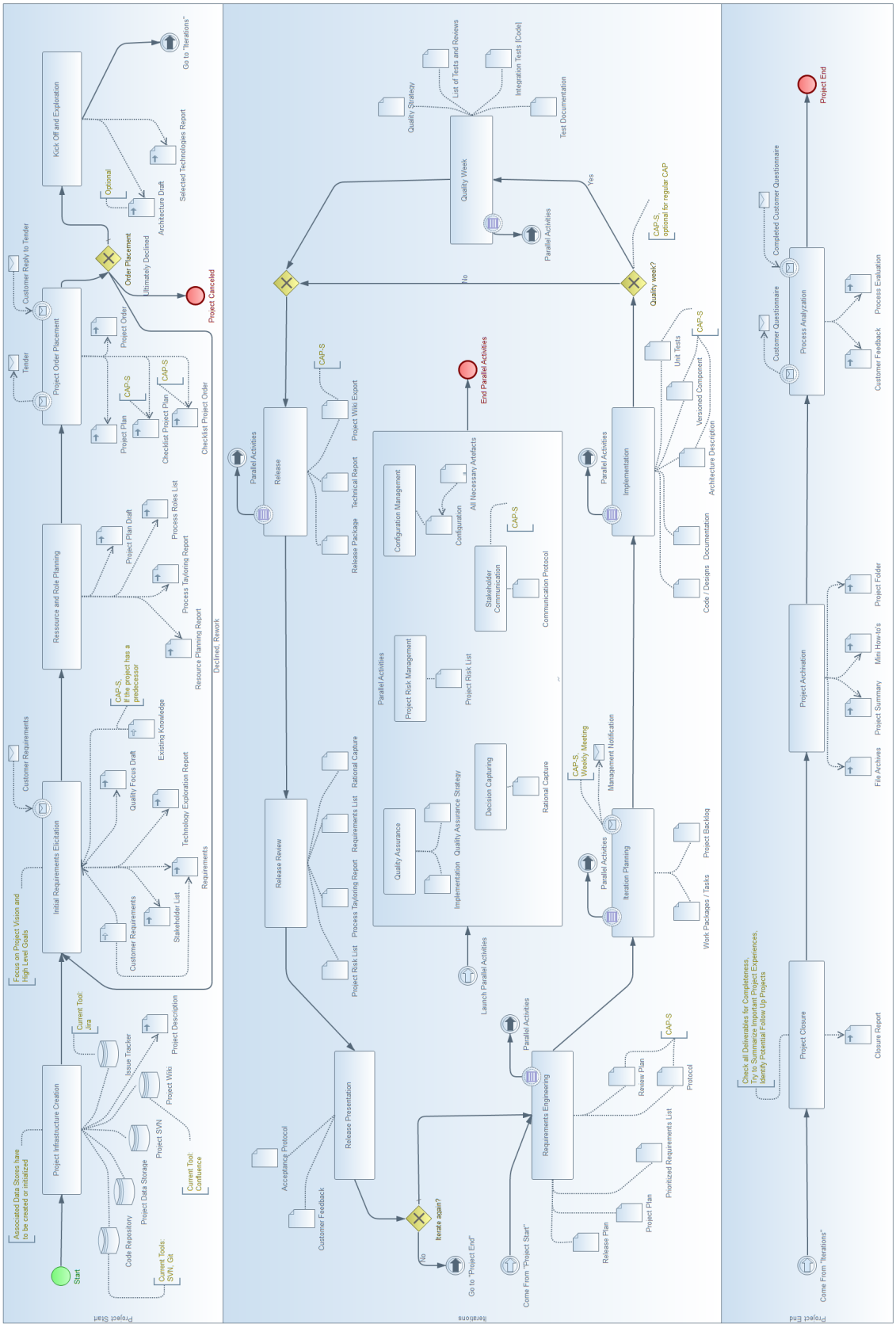


Figure 3.1: CAP Reference Process, Strategical Overview

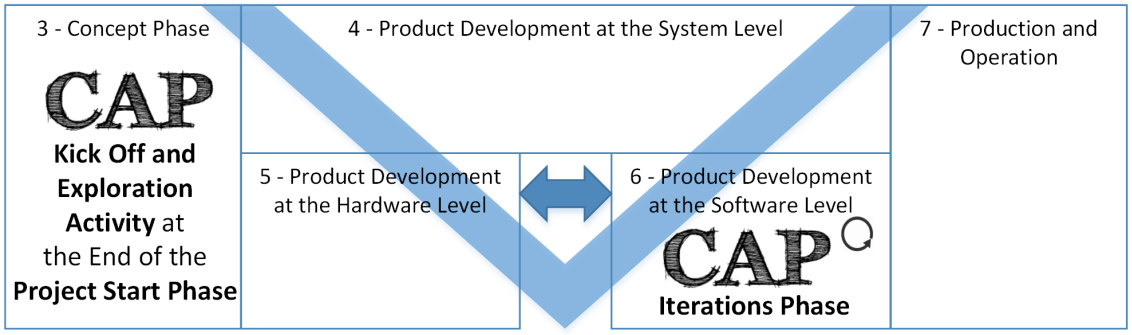


Figure 3.2: ISO 26262 Safety Lifecycle Phases Mapped on the CAP [ISO11a, parts 3 -7]

In consultation with BMW Car IT GmbH it was decided that this nature should not be changed in the course of actions.

The consequence is that ISO 26262 safety lifecycle phases can be mapped on the CAP only—in a lightweight manner based on item assumptions—for the concept phase and the software development phase. It can not perform real system level development [ISO11a, part 4], since this phase is oriented on a combined specification of hardware and software architecture, including the specification of the hardware-software interface (HSI) [ISO11a, 4-5.2], which exceeds its current focus on software requirements and which is not possible on vague item or hardware assumptions.

The intended traceability for the software development from phase 3, to phase 4, to phase 6 is hence interrupted and the progression of the safety lifecycle is hindered [ISO11a, 2-6.4.4]; the artefacts that cannot be created properly are the technical safety requirements (compare [ISO11a, 4-6.4.2]), even if it was desired for the agile process.

3.3 Analysis of the Tailored CAP for Autonomous Driving

To enrich the CAP with ISO 26262 safety lifecycle concerns, we specifically analyzed RE, QM, build and deployment process, in coordination with the autonomous driving system development team. This development process is called AD-CAP in the following.

The insights in the tailored process for the development of the autonomous driving system have been gathered based on weekly meetings with developers and in participation as a source code committer, while the BPMN reference process was created from additional document research and interview of the process designers (Figure 3.3).

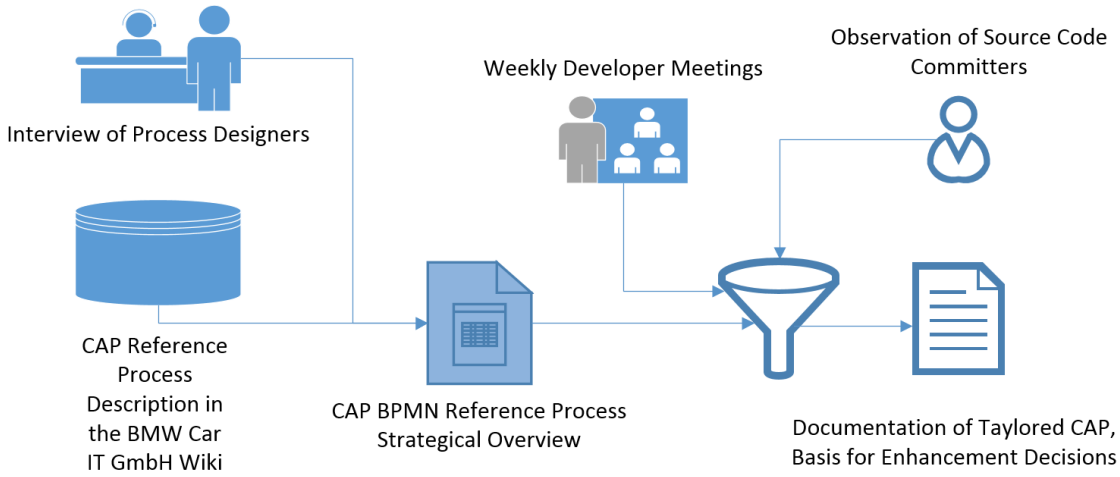


Figure 3.3: Information Collection Strategy

3.3.1 Requirements Engineering

Similar to SCRUM, agility in the AD-CAP is supported by use of a prioritized requirements list (product backlog) that is continuously updated and extended in the course of the development. A set of important, achievable tasks for the next immediate iteration (i.e., sprint) is then put into the sprint backlog with a fixed date. Prioritizing requirements and maintaining the backlog ensures that important features are given higher priority, which adds important value to the product first. This is equal to a timeboxing approach.

3.3.2 Quality Management

The CAP encourages various QM techniques and the documentation in templates. The AD-CAP implements:

1. **Unit Tests:** Unit tests are implemented, but coverage is not exhaustive. Test-Driven Development is not used by developers.
2. **Code Review:** When committing source code, conditions for a mandatory peer-review can be defined, before the commit is let through to the code repository. This is done by the code review system Gerrit on top of the source code management system Git.

Prototyping

Current prototypes of the autonomous driving system are tested using a virtual simulation environment. Knowledge gain for software development therefore does not yet result from real-world prototypes.

3.3.3 Build and Deployment Process

The lowest software layer that has been determined for the AD application is the ROS framework, but not the hardware layer or the operating system layer. The used ROS version is Groovy Galapagos with its build system catkin.

Build process automation is currently supported by a Jenkins continuous integration server that automatically builds the latest committed source code version of the software. Continuous integration encourages developers to maintain a working build process and to detect broken builds immidiately.

Deployment and integration testing plays no role in the process, yet.

Figure 3.4: Build Process Target Platform

4 Arguing Functional Safety in Accordance with ISO 26262

ISO 26262 consists of nine parts that specify a comprehensive safety lifecycle, including non-production activities, such as repair and decommissioning, and a tenth part that is a guideline for correct application, including examples. The tailoring of the safety lifecycle is based on the safety integrity level classification of the developed items. However, to figure out how ISO 26262 can be applied, we need a deeper understanding.

4.1 In-Depth Analysis of ISO 26262

In focus of this thesis are software development activities and their related boundaries. As a quick overview, we give the relevant parts in bold font, and particularly important parts additionally underlined (Table 4.1).

Pt	Phase	Essential Contents
01	**Vocabulary**	Terms and Definitions, Abbreviations
02	Management of Functional Safety	Functional Safety Management, - Planning and Execution
03	**Concept Phase**	Hazard Analysis and Risk Assessment, Functional Safety Concept, ASIL Determination (A-D, QM)
04	Product Development: System Level	HSI Specification, Technical Safety Requirements, Integration Tests, Functional Safety Assessments
05	Product Development: HW Level	HW Design, HW Failure Analysis
06	**Product Development: SW Level**	SW Architecture, SW Unit Design and Implementation, SW Testing
07	Production and Operation	Production, Service, Decommissioning
08	**Supporting Processes**	Configuration and Change Management, COTS Qualification
09	ASIL- and Safety-Oriented Analyses	ASIL Requirements Decomposition
10	**Guidelines on ISO 26262**	Key Concepts, Concrete Examples

Table 4.1: ISO 26262 Compact Description Table (compare [ISO12, p. vi])

4.1.1 General Problems

Definitions in ISO 26262 often leave room for interpretation. We identify two critical points that are both relevant for the safety lifecycle tayloring:

1. The **ASIL** is determined during hazard analysis and risk assessment based on the three parameters "controllability", "severity" and "probability of exposure" of a hazardous event; derived safety requirements inherit the ASIL [ISO11a, 3-7]. However, these three criteria are still subject to personal experience and subjective opinion (compare [Kri12, pp. 230f]), and therefore also the resulting ASIL. A commonly accepted ASIL for hazardous events (and related items) across different manufacturers is rather expected to arise over time as convention [Kri12, p. 232], so ASIL determination should not only be performed independently, but the result should also be compared with competitors.
2. **Tables** in ISO 26262 are meant to give the required "topics" and "methods" for process activities at a glance; they define the safety activities that are to be implemented. However, choosing a valid set of methods in a set of alternatives is vague [ISO11a, all parts, 4.2]: "For alternative entries, an appropriate combination of methods shall be applied [...] independent of whether they are listed in the table or not (...). A rationale shall be given that the selected combination of methods complies with the corresponding requirement" [ISO11a, all parts, 4.2]. Strictly spoken, this means that it is possible to even use unmentioned methods to satisfy a safety requirement, because the importance ultimately depends on the soundness of the rationale, only.

   Observation: For choosing a set of methods from a table, the validity of this set ultimately depends on the rationale alone and is completely independent from the listed methods.

   It is neither possible to give a *necessary* condition for a set of methods, nor a *sufficient* one. While this definition was probably given to allow developers greater freedom in choosing a set of methods for a specific situation, it is also irritating.

Conventions: In coordination with BMW Car IT GmbH we decide that to reach an ASIL, at least all highly recommended methods ("++") should be applied for a table (compare [ISO11a, all parts, 4.2]). To choose an ASIL, a conservative rating is to be applied by selecting the higher ASIL, if an unambiguous determination is not possible.

4.1.2 Applicability of ISO 26262

The development related activities of the ISO 26262 safety lifecycle are **intended to be initiated on the item level** during the concept phase [ISO11a, 1-1.69] (Figure 4.1),

where the first set of top-level safety requirements—the safety goals—are to be identified and which serve as the basis for deriving more fine-grained requirements in the further progressing safety engineering process. An item equals to a "system" or an "array of systems"—which includes by the definition of "system" also whole subsystems [ISO11a, 1-1.129 NOTE 2]. A system is defined as a "set of elements that relates at least a sensor, a controller and an actuator with one another" [ISO11a, 1-1.129], and the term controller being undefined. However, from the context we can understand the term *controller* as a system element that consists of hardware and software meant to operate an actuator based on sensory input data.

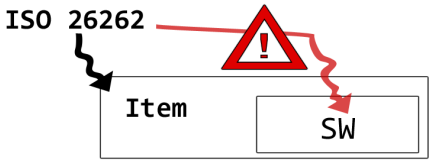


Figure 4.1: Initial Applicability of ISO 26262 Safety Lifecycle

Software components, hence, are no items in accordance with the definition of ISO 26262 [ISO11a, 1-1.32], but only part of an item.

This means, the ISO 26262 safety lifecycle [ISO11a, Part 1-9] is, in general, not directly eligible to be initiated on the level of software development.

Software Safety Requirements

If a software element is part of a safety-relevant vehicle item, it is required to construct an according set of software safety requirements [ISO11a, 6-6]. They are derived from the technical safety concept that is derived from the functional safety concept that is derived from the safety goals (compare Figure 4.3).

Due to the context of autonomous driving—where we need novel hardware components to satisfy novel requirements, e.g. in respect to performance—it is currently not clear, what definitive items the final system will have. This is in particular true for the computing platform that will be running the software and the exact types of sensors the prototypes will need. Therefore, chances to describe stable hardware-software interfaces, as required in [ISO11a, 6-6.1, 6-6.3.1, 6-6.4.2], are very limited at the moment.

Result on ISO 26262 Applicability

The consequence of these inherent applicability constraints is that full ISO 26262 compliance in the software development focussed CAP is not possible without extending it to a full-grown product development process (compare Figure 4.2).

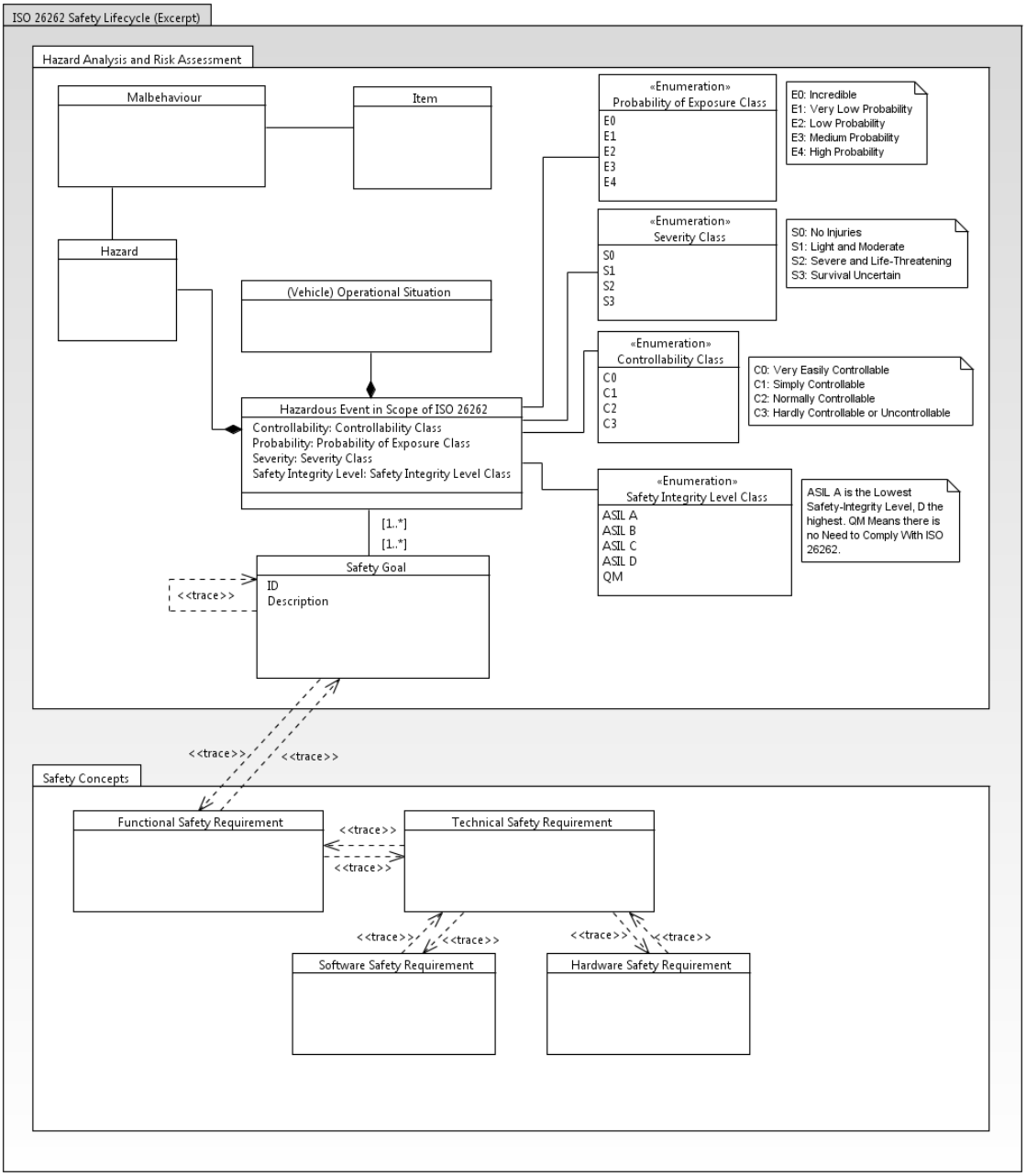


Figure 4.2: Tracing of Safety Requirements [ISO11a, parts 3, 4, 5, 6]

4.1.3 Alternative Approaches to Software Development

ISO 26262 provides some alternative approaches to develop software components independently from hardware that could help to solve this problem.

4.1.3.1 Qualification of SW-COTS

Software can be developed as COTS in accordance with [ISO11a, 8-12] to become qualified for re-use and easier integration.

Benefits: Saves “software unit design and implementation" [ISO11a, 6-8] and “software unit testing" [ISO11a, 6-9] activities during integration.

Drawbacks: Benefits are rather limited. If the software has to be modified during integration, the qualification status is lost and there are no benefits at all [ISO11a, 6-7.4.7].

4.1.3.2 SW-SEooC Development

“Generic elements” can be developed in accordance with [ISO12, 10-9], including software components [ISO12, 10-9.2.4].

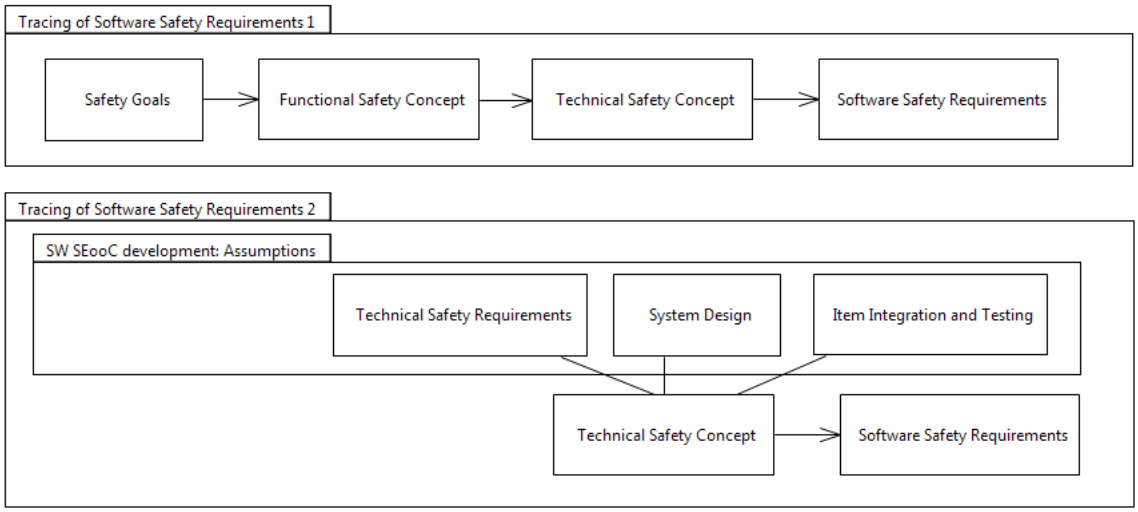


Figure 4.3: Establishing Software Safety Requirements for SW-SEooC’s

Benefits: Suitable for newly developed software, software re-use with and without change [ISO12, 10-9.1, Table 3].

Drawbacks: Limited applicability for autonomous driving (uncertain argumentation for software components that are potentially “generic”, low count of generic software components), high effort and uncertainty in describing technical safety concept based on assumptions.

Verdict on Alternative SW Development Approaches

In coordination with BMW Car IT GmbH it was decided that neither of these encountered concepts should be pursued in the course of actions as they have to be reviewed more carefully first.

4.1.4 Important Concepts

There are a few aspects that are vitally important for the safety case throughout the safety lifecycle, so that many safety activities are somehow related to them.

ASIL

The ASIL (compare 4.1.1) defines the safety integrity level for an item. The ASIL classification determines the tayloring of the ISO 26262 safety lifecycle and defines quantitative constraints for the failure probability of system components.

However, since we are dealing with software components, all faults and failures are systematic [ISO12, 10-4.3, Figure 5] and for systematic failures quantitative safety analyses are not performed [ISO11a, 9-8.2, Note 2], because a quantitative failure rating cannot be applied. Instead, the importance of the ASIL for software development lies in the process tailoring only (i.e., the set of safety activities for the accompanying safety lifecycle).

The ASIL is determined in accordance with [ISO11a, 3-7] by defining probability, severity and controllability classes of hazardous events from which classified safety goals are derived (Figure 4.4, compare Figure 4.2). The ASIL is the key property of all safety goals and requirements that is inherited from all derived safety requirements.

Severity class	Probability 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

Figure 4.4: ASIL Determination [ISO11a, 3-7.4.4.1, Table 4]

Different Views on the Car and Traditional Safety Analyses

When performing the required functional safety analyses, it is important to differentiate different function granularities. Those differences arise from different views on the system that use different degrees of abstraction and address different process participants with different qualifications and capabilities to understand certain aspects of the system. The ISO 26262 vocabulary does not define what a function is, but it is implicitly defined in the type of the input that is necessary for corresponding analysis methods.

The challenge for the process is, to link the related artefacts in order to allow tracing, so that the safety case is always consistent.

Deductive Analyses ("top-down") use the abstract high-level concept of a function as starting value. The prominent methods for this are Fault Tree Analysis (FTA), HAZOP and SHARD.

Inductive Analyses ("bottom-up") require detailed low-level descriptions of functions as starting value such as mathematical definitions, atomic hardware functions, or code. Important safety analysis techniques in this class are: ETA, FME(C/D)A.

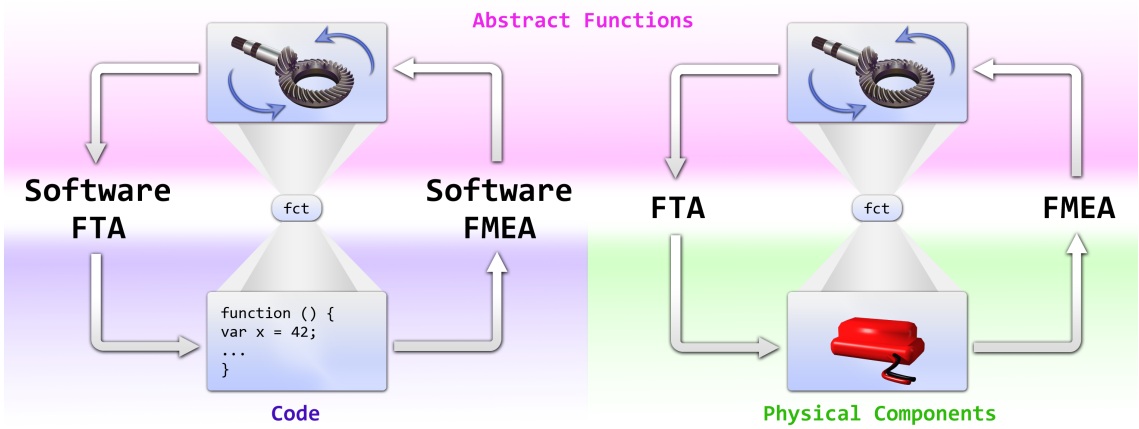


Figure 4.5: Different Function Granularities Provide Input for FTA and FMEA

ISO 26262 requires these analyses to be performed as part of the system design phase [ISO11a, 4-7.4.3.1], after planning of the technical safety concept and the architectural design assumptions.

Artefacts from performed safety analyses appear as "evidence" in the safety case. In addition, the chosen method's appropriateness has to be justified.

4.1.5 Special Implications on Failure Modes for Autonomous Vehicles

A safety case has to describe a strategy to deal with failures and term the mechanisms that allow the system "to achieve or maintain a safe state" in case of failure occurence

[ISO11a, 4-6.4.2.2c]. This argumentation can also include a probability argument that argues safety through unlikeliness of failure occurence [ISO11a, 3-7.4.3.6], or a safe-by-design argument; failures cannot cause hazardous events, if the item can be designed in a way that the system *passively* enters or maintains a safe state in case of failure occurence, i.e., the item does not need to perform any kind of active hazard mitigation for such failures.

Some items can implement simple mechanisms for the system to reach or stay in a safe state on failure. A common approach is to enter *switched-off mode* (compare [ISO11a, p. 14, part 1], [SZ10, p. 103]). Such safety-relevant items that cause no harm in case of failure, but whose functions are no longer available to higher level systems, are called "fail-safe" items.

The contrary design principle is followed by *fail-operational* items, whose provided functions are still available in case of failure. This can be implemented through redundancy [SZ10, p. 103], where the system stays fully operational, or through graceful degradation, where the system continues to operate on limited functionality [ISO11a, 6-Table 5]—a principle also called *fail-reduced* systems" [SZ10, p. 103].

For fully autonomous vehicles in operational mode, the general case is that there exists no safe state that can be entered if a failure occurs. However, there is a chance for certain subsystems to fail safely, namely those that are not directly involved in the driving process. Such a subsystem could be involved in the programming of the travel route—providing some human-machine interface—, because there are arguably no hazardous events for a self-induced shut-down of such a system, neither if the car is driving autonomously, nor if it is not.

4.2 ISO 26262 Safety Case Management in GSN

To improve the CAP in terms of functional safety, we implement the ISO 26262 requirement to manage a lightweight safety case [ISO11a, 2-6.4.6.1] with focus on software safety requirements. We use GSN as indicated in 2.4.5.

The reasons for GSN are:

- The notation is rich enough to cover all important aspects of a safety case
- Instead of managing a textual log of hazards and implemented safety measures, the entirety of safety goals, arguments and solutions in a GSN safety case depicts structure and dependencies clearer to support incremental and iterative design
- It is able to capture required rationales as elaborated in 4.1.1
- It is flexible in the scope and supports entity abstraction for rapid construction of partial aspects of the safety case

- It supports agility for the software development process as well as for the product development process

The safety case is created and maintained as part of the safety management during the concept and product development phases and is a required work product in the safety lifecycle; it is intended to capture all identified safety requirements and argue why they are satisfied [ISO11a, 2-6.4.6, 2-6.5.3].

Hypothesis 1

GSN can be used to support safety case construction in an agile software development process in accordance with ISO 26262.

A safety case aims to "provide a clear, comprehensive and defensible argument, supported by evidence, that an item is free from unreasonable risk when operated in an intended context" [ISO12, 10-5.3.1] (compare [Kel98, p. 3]).

A safety argument therefore has to link these four definitive concerns (compare [Kel98, p. 119]) in order to provide valid safety cases: Safety goals, safety arguments, related evidence, and the context in which they are valid.

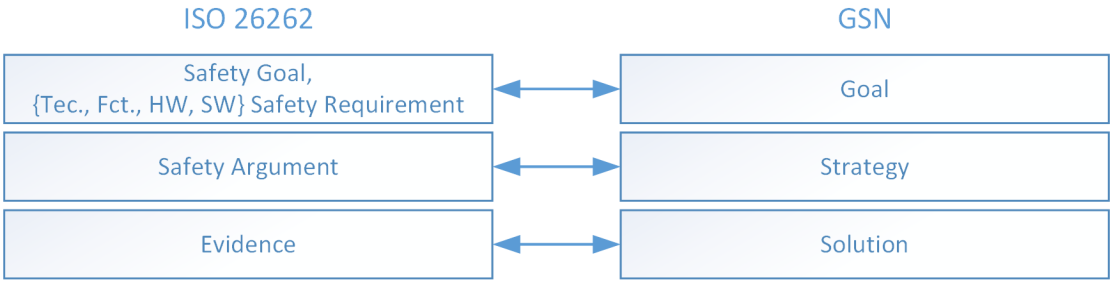


Figure 4.6: Mapping between ISO 26262 Terms and GSN Terms

Safety goals are the starting point in the construction of a safety argument in GSN. They are identified and defined based on identified hazardous events due to item malfunctions in a related operational system context. The operational context describes the state and the outer circumstances of the system when the related hazardous situation occurs.

We define hazards as a subset of malfunctions that lead to hazardous events and which's occurence must be prevented with reasonable effort, because of their potential to harm humans and damage the environment.

Safety Goal

A safety goal is the starting point in the argumentation chain (Figure 4.7). Accurate safety goals are formulated as statements that can be evaluated to true or false; they address a requirement [Kel98, p. 72]. It is therefore sometimes found that safety goals should actually have been termed safety claims [Kel98, p. 204].

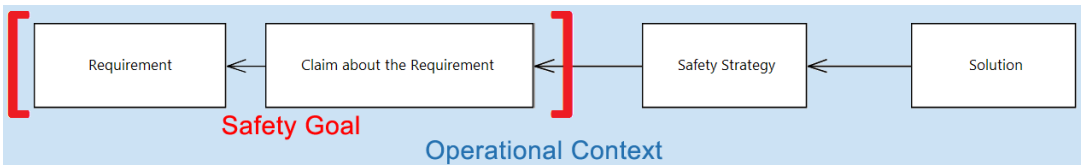


Figure 4.7: Block Diagram: Safety Goal, Safety Strategy, Solution

A safety goal is valid, if:

- it is a claim (i.e., it can be evaluated to true or false), and
- it is atomic

Safety Strategy

It is necessary to justify that implemented safety measures indeed satisfy the safety goals in the safety case. This is the safety strategy. It contains the reasons and explanations, why one can be be confident that the performed functional safety lifecycle and the evidence that it created is qualified to guarantee functional safety. Similar to measures for quality management, there are essentially at least two different types of safety arguments, *product arguments* and a *process arguments*:

The *product argument* relies on features that have been implemented in the product in order to guarantee functional safety [ISO12, 10-5.3]. These can, e.g., be redundancy in parts of the system architecture or additionally implemented safety functions.

The *process argument* argues safety through process quality and safety activities that have been performed [ISO12, ibid.]. Process arguments can typically be planned early as part of early project (safety) management, since there don't have to exist any implementation details or concrete technicals details in order to work them out.

Both types of safety arguments are important to construct an overall sound safety argument.

A safety strategy is valid, if:

- it explains why and how a safety requirement (goal) is met, and
- it is not a claim

Solution

A solution is the result of a test, simulation, analysis, or the like that is created or documented in order to demonstrate that a safety requirement is met. The ISO 26262 term for solution is evidence. An argument without all necessary solutions is unfounded and therefore faulty. Solution artefacts are typically created during the safety lifecycle and compiled in the safety case [ISO11a, 2-6.4.6.2].

It is critical to ensure that created evidence is suitable to support the fulfillment of safety requirements. The bare presence is insufficient, so an argument that reasons why the evidence creates confidence in the system safety is crucial.

A valid solution:

- terms a process activity, required process artefact or product property, and
- does not include a claim

4.2.1 GSN for Safety Case Development, Reuse, and Maintenance

To support agile development, it is important to have a clear structure in the argument that glues together the safety case, because in iterative development, the safety case must change with the system design; the clear structure—in contrast to tabular stored text or unstructured text—makes it feasible to assess the impact of system changes to the safety case [Kel98, p. 28] and to identify invalid elements with need for update and faults in the overall structure. This allows to systematically maintain the safety case [KM99, p.13], which is the desirable capability for agile development.

To achieve this, Kelly used the goal structuring notation (GSN) that is able to establish the tracing between safety goals, safety arguments, evidence, and to link them with context assumptions. With these abilities, it fulfills all requirements (compare Figure 4.2) to model the safety argument and to present and maintain a valid safety case.

A problem with GSN was that it could not express modularization, which lead to very large safety arguments in practise. For this reason, it was later extended to EGSN that implements this ability (compare [Ori11, pp. 17ff]). This leads to some confusion today, because the term *GSN* is often used synonymously with *EGSN*, and the term *EGSN* is not popular.

GSN can be used to argue safety for different system views for horizontal safety argument diversification. Arguing on different views separates concerns and improves clearity in the argument.

Important distinctions are between product and process arguments (compare [Kel98, pp. 92, 94]), functional and non-functional safety properties (compare [Kel98, p. 104]), and qualitative and quantitatve risk mitigation strategies [Kel98, p. 319].

Basic Elements

The basic elements of GSN [Ori11] are designed to structure the safety case argument (Figure 4.8).

To explicitly support construction of preliminary safety cases and incremental development, GSN provides also notions for undeveloped or uninstantiated parts of the safety case through entity abstraction (Figure 4.9) [Ori11, A1.3].

An uninstantiated entity is abstract and needs later concretization [Ori11, A1.3].

- Example: Insertion of numeric values for variables within a GSN element.

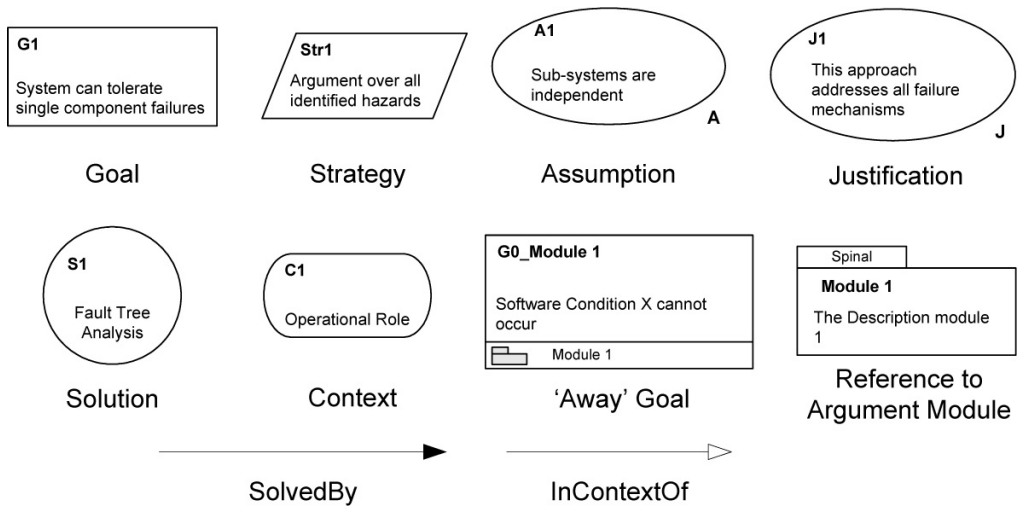


Figure 4.8: The Basic GSN Elements [Wea08]

Figure 4.9: Entity Abstraction in GSN [Ori11, A1.3]

An undeveloped entity is a stub that requires more detailed GSN modelling [Ori11, A1.3].

- Example: A GSN goal that only states that some subsystem must be safe to operate.

These mechanisms for abstract entity modelling are important for interim safety case construction. They allow for selective development of the safety case and communicate its current shortcomings without interrupting the modelling process.

Challenging the Safety Argument for Safety Case Maintenance

If a system is changed, the corresponding safety case is potentially invalidated.

To express this, elements of a GSN model can be *challenged*. This is done by identification of the system change's impact on the safety case.

If items are challenged, the safety argument is considered *damaged* and has to be recovered in the safety case change process [Kel98, p. 122]. The graphical notation for a challenged element is the respective crossed out element [Kel98, pp. 127ff].

If a GSN element is challenged, so are all of its dependent elements. To document the evolution of the safety case, in addition to the deprecated safety case the challenged and the repaired safety case have to be stored as process artefacts, where the repaired safety case has again to be challenged, if the system changes again.

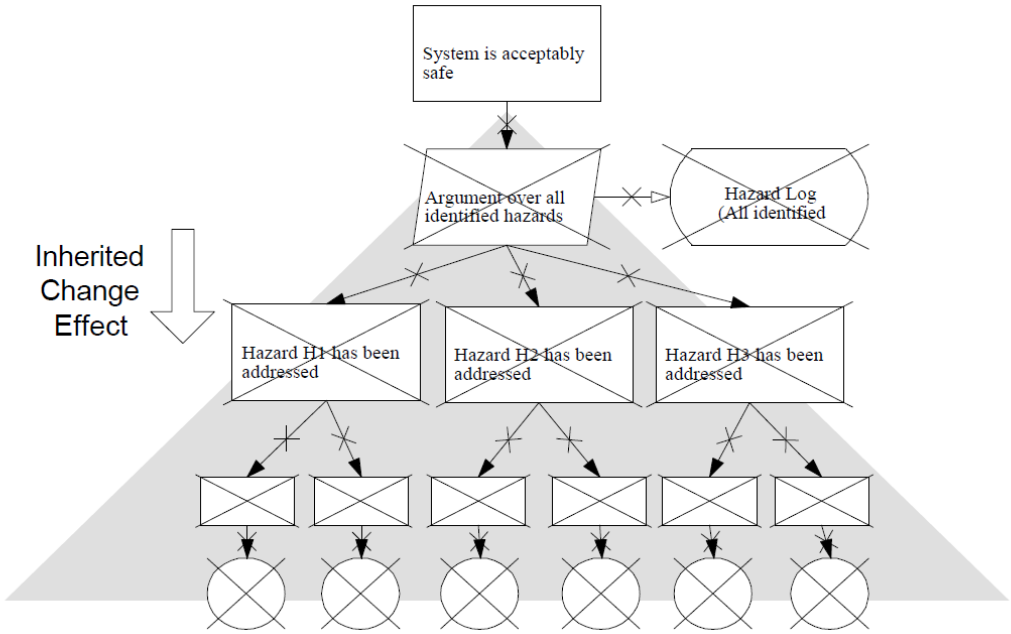


Figure 4.10: Damaged GSN Safety Case with Challenged Elements Crossed Out [Kel98, p. 134]

GSN Patterns

The idea of GSN patterns is to transfer the positive experience with software design patterns to safety case design. GSN patterns allow to identify safety anti-patterns—weak or flawed safety arguments—, capture and discuss good practice in safety argument design, and support reusing established safety arguments for common structures [KW04] (compare: GSN working group weblog for recommended GSN patterns [The13]).

4.3 GSN Applied in the Project

We initiate a first safety case in the autonomous driving project at BMW Car IT GmbH.

We recapitulate that the elaborated safety case will be huge due to manyfold ISO 26262 demands in terms of methods and work products (compare [Pal11, 3.3.3]). In order to modularize and focus it, we concentrate on the currently relevant aspects of the safety lifecycle and its artefacts:

- The safety arguments can be divided into product and process arguments, where the process argument is elaborated.
- The phases of product development (concept phase, system design phase, hardware and software design phase) have their own type of safety requirements and should therefore receive a separate argument module.

4.3.1 Methodology

To construct the GSN safety case, we use the 6-step-methodology presented by Kelly [Kel98, pp. 80ff] (Figure 4.11).

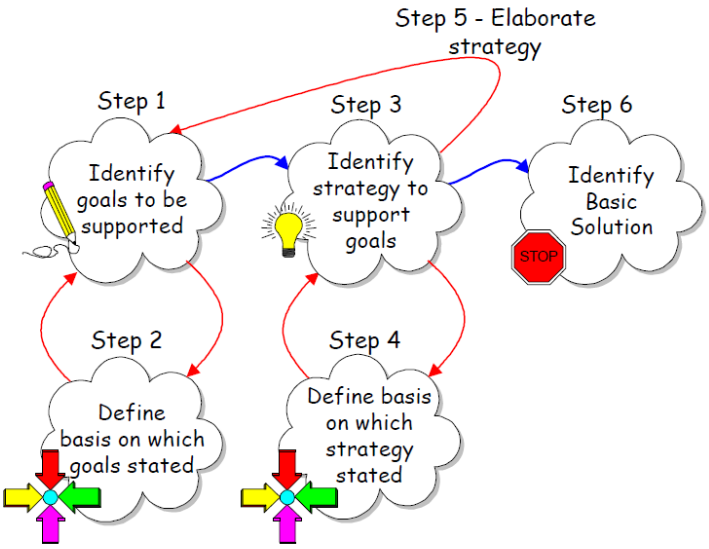


Figure 4.11: Iterative 6-Step Construction Method of a GSN Safety Case [Kel98, Figure 28].

4.3.2 Exemplary High-Level Safety Case

The safety case on the highest level must depict the understanding of safety for the developed system and define the hierarchy of top level safety goals, based on safety scopes and safety standards [Kel98, p. 95]. It gives the fundamental structure and the according justifications, why the developed system will be safe.

The result (Figure 4.12) gives all team members a clear overview of which safety aspects are currently under development, which safety standards are implemented and the distinction between product and process oriented safety arguments.

We mark all safety scopes but the functional safety scope as undeveloped und uninstantiated, since they are out of focus for this thesis. However, we do not drop them in order to express that they are missing for a comprehensive safety case.

4.3.3 Integration in the CAP

GSN modelling can either be integrated as a parallel activity that can be launched at any time the developers see fit, or it can be integrated as a quality activity.

In either case, the artefact to contain the GSN safety case report must be saved in a designated version control file repository (if the chosen modelling program does not provide

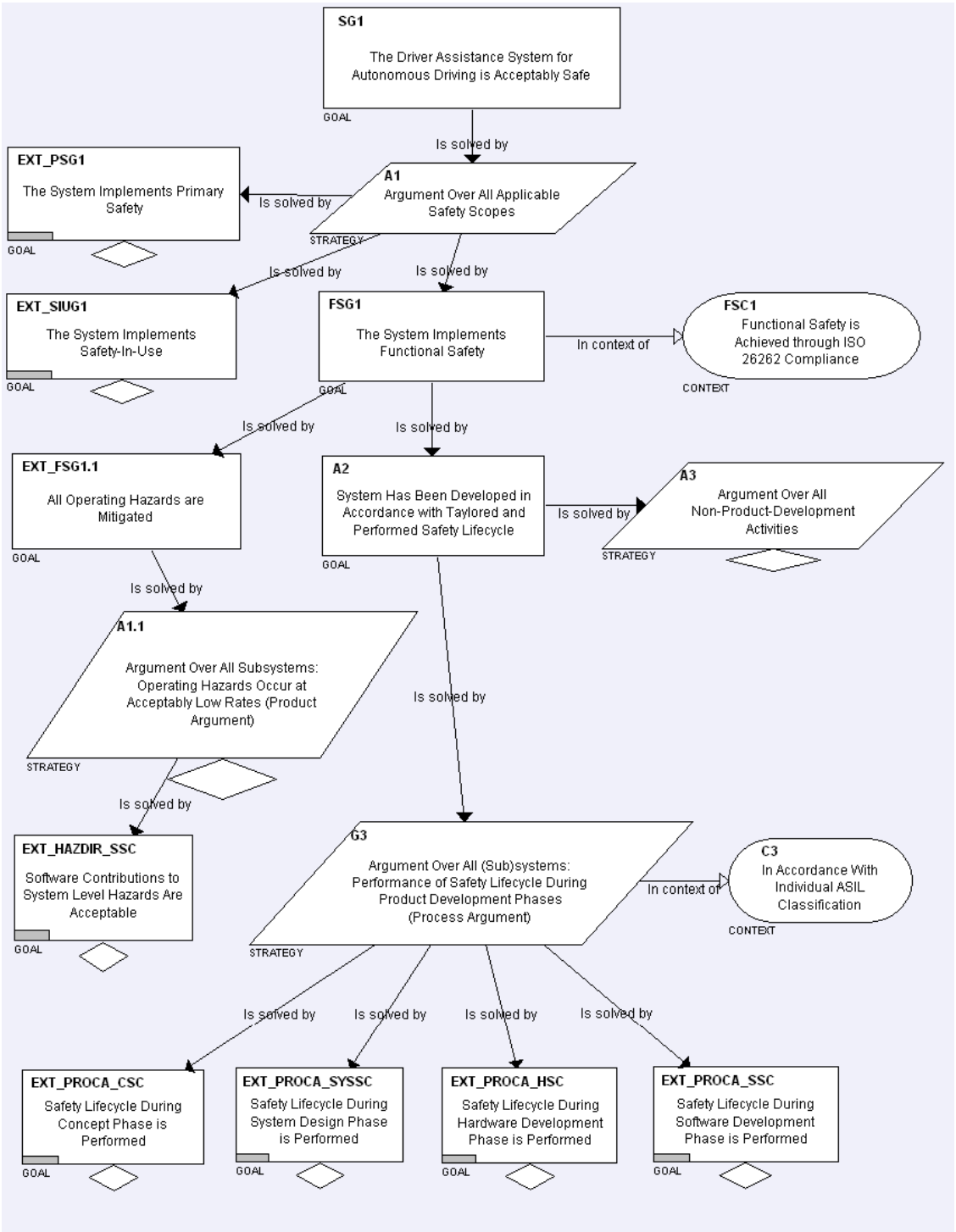


Figure 4.12: High Level Safety Case for the Autonomous Driving Project

such a function), where the history of the artefact is saved. Changes to the safety case report must be done by challenging the concerning GSN element as explained in 4.2.1; the damaged safety case report must be saved and subsequently, a recovered safety case report has to be created and saved.

5 Improvements in Functional Safety

The reference process model in ISO 26262 is a V-Model that contains multiple V-processes as inner subprocesses [ISO11a, pp. vff, all parts]; the outer "V" for the development of the complete product can launch those inner V-processes, after the required requirements have been gathered. The contained subphase for safe software development is such an inner V [ISO11a, 6-Figure 2], which is to be initialized after the system-level development has been completed. The effort is to improve functional safety in the autonomous driving project by mapping the safety lifecycle on the CAP.

Chapter Outline

In this chapter we evaluate the advantages of introducing Model-Driven Development (MDD) to improve functional safety, sketch a platform to deploy the application software on and analyze, if there is a toolchain constructable in order to match the ISO 26262 software development process requirements.

5.1 Model-Driven Development

MDD is not used in the current development process. In coordination with BMW Car IT GmbH, it was decided to only evaluate a selection of promising tools, but not to introduce them in the process.

Hypothesis 2

Model-Driven Development can help to improve the development process according to ISO 26262 requirements.

In the safety domain, the way software is implemented—i.e. which software development paradigms are used—has significant influence on the complexity of manual activities during the development process. Functional-safety-aware development has to ensure traceability between the technical system design requirements and the hardware and software implementation layer, so that it is guaranteed that the implementation on

the code level effectively reflects the system design decisions and completely implements the technical safety concept. This traceability is a challenge, since it means to provide a link between the abstract design and the concrete implementation.

When the implementation is done manually, traceability has to be ensured manually. This is a complex and error-prone task. The linking can only be supported through adequate naming of variables, functions and structures that reflect the naming in the specification, and grouping related code into files or packages, which reflects components and subsystems.

A more integrated solution to this challenge is model-driven development (MDD), where a part of the implementation is automatically generated out of the design layer. For the case of a qualified tool, it is therefore not necessary to prove, that the implementation layer reflects the design. MDD is in principle independent from the target programming language, so it eases code migration.

When using MDD, process activities to check the implementation against the system design can be automated. However, complexity is lifted up from the implementation layer to the system design layer, where a detailed model that implementats the system's functionality has to be created.

MDD Paradigms

The benefit of MDD to the process varies in the way MDD is applied, where different strategies are not mutually exclusive.

Behaviour Models

Behaviour models are state transition diagrams that implement the system behaviour as a sequence of operations in reaction to triggered events. These models are ready to be deployed after code generation and compilation, because they completely specify the program behaviour. However, it is unclear whether implementing a specification through creating behaviour models is slower than coding for software developers, because of the need for graphical modelling, and if there is a gain in productivity over time, since behaviour models for complex systems are easier to oversee than code, so changes are easier to do. The model should be executable and debuggable, so the system can be simulated even before deploying real code, so the system behaviour can be studied early. Also, migration of the project to a different platform is easy, since models are not bound to specific programming languages or runtime environments.

It is, however, not advisable to implement every system as a behaviour model, since not every system can be described by distinguishable states properly.

Model-Driven Architecture

Model-Driven Architecture (MDA) stresses the clear separation of the functional design level and the technical design level. The developed system's portability between different execution platforms is therefore high.

This approach enforces the generated code to be consistent with the system architecture that is explicitly modelled.

Usage Concepts

The front-end concepts for MDD can be distinguished into two categories.

Graphical models

Graphical models allow for a visual system representation.

Textual models

Domain specific languages (DSLs), architecture design languages (ADLs). Speed up the implementation particularly if the development environment provides supporting features such as code auto-completion and code insertion assistants.

V&V Capabilities and Functional Safety Benefits

Properly applied MDD increases code quality, and hence functional safety, for a number of reasons. The important aspect for the AF development process is that code generation out of models establishes traceability between the technical safety concept and the implementation level. The compliance of the implementation to the technical safety constraints can therefore be automatically proven. It also simplifies refactoring, since the refactoring is executed at the level of the model and re-verification of the implementation can be ommitted.

Behaviour models allow semantic checks, model testing and model debugging, since the model is executable and describes the system's behaviour. If a system was modelled as state transition diagrams aware of streams over message channels (compare [BS01]), it can be thoroughly checked not only for correct functional behaviour, but also for timing constraints and concurrency issues.

Model-Driven Architecture allows for explicit modelling of the architecture that is then transformed into code by generators. This potentially allows for verification on the structural level such as presence of safety patterns.

Regarding functional safety, MDA is a benefit and can express a component structure of the software that can be annotated with corresponding failure probabilities of hardware components the software is deployed on. The resulting model can be used for automated failure analysis (e.g., automatical generation of safety engineering evidence, such as HiP-HOPS).

An Ideal Model-Driven Development Environment

A development tool or - environment to support a development process in the safety domain should provide a **strong link between safety engineering, system engineering and implementation**, connecting the different system views through model transformation.

5.1.1 Solution Frameworks

5.1.1.1 AutoFOCUS

AutoFOCUS is a fully integrated development tool for embedded systems that supports the whole development process from requirements specification to deployment [for12]. AutoFOCUS recently moved from a custom Java GUI (AutoFOCUS2) to Eclipse as basis framework.

The latest stable version of AutoFOCUS3 (2.3) currently implements ISO 26262 safety requirements modelling, however, it fails to provide the correct ASILs A-D and instead gives the regular SILs 1-3 from IEC 61508. There exists no exact mapping from SILs to ASILs—except for SIL 1 to ASIL A—, but using the probability of failure definitions in the regarding standards, results in at least an approximate relation for the remaining ASILs B to D (figure 5.1), where, however, the lower bound of the failure probability can sometimes not be guaranteed (e.g., ASIL D to SIL 3).

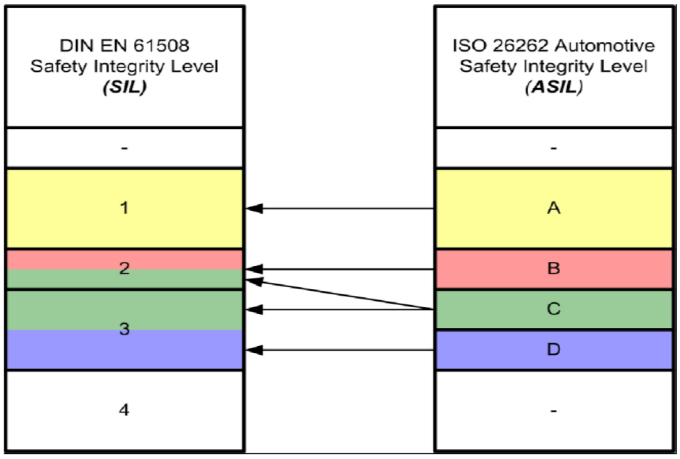


Figure 5.1: ASIL to SIL mapping [Ern10, p. 38]

Requirements can be specified informally, but are hard to trace to the component architecture level.

The nightly build—version 2.4—has improved on this and allows ASIL classification from A to C, missing D. In addition, it allows formal requirements specification, clearer tracing

to the system architecture and automated validation against the system design using the NuSMV model checker for formal requirements specified using temporal logic.

In summary, AutoFOCUS3 in the 2.4 nightly build version is a promising tool to support the required traceability from requirements to the logical (via informal requirements) and technical (via formal requirements) architecture, allows code generation—so traceability from technical architecture to implementation—, and supports automated deployment. This allows for high automation in verification of requirements.

5.1.1.2 Chromosome

CHROMOSOME stands for Cross-domain Modular Operating System or Middleware [for13, p. 4] and is focused on data-centric design that allows automatic computation of communication routes between components based on the data exchange requirements [for13, p. 9].

CHROMOSOME consists of an Eclipse-based model-driven design tool (XMT)[for13, p. 26] for graphical modelling of distributed systems and is able to deploy those models on a runtime system with hardware abstraction layer [for13, p. 4].

As a combination of MDD tool and HW-abstraction layer, it provides a comprehensive approach to potentially tackle the problem of HW dependent software development and also features automatic calculation of network routes for the deployment of the developed software nodes [for13, pp. 32f].

Bride

Bride [BRI13b] is specifically designed to support component based software engineering [BRI13a]. It is focused on the publish-subscribe paradigm that is also the core communication paradigm of ROS. ROS is indeed one of two available target frameworks to deploy the designed components (the other one is RTT).

The functional principle behind BRIDE relays on model-driven architecture, so the architecture—and the architecture alone—is explicitly designed in a graphical model. The concrete functionality is manually implemented in protected areas in generated code. Manually written code within these protected areas is not updated (i.e. deleted) when the code generation is triggered and the implemented method was not modified in the model.

BRIDE does not allow to create state machines. That means, the system behaviour cannot be graphically modeled, but has to be implemented manually.

If an already started development process is switching to BRIDE, it is therefore necessary to model the existing system, but the existing implementation can be copied & pasted, which makes the migration effort feasible.

5.1.1.3 Embedded Coder

Embedded Coder [Mat13] is a Matlab extension to generate code for embedded systems. It can therefore profit from other extensions available for Matlab.

It was specifically designed to comply with safety standards such as ISO 26262, which means, it implements features to manage requirements, ensure bi-directional tracability between requirements, models, and code as required in ISO 26262 and supports automated (SIL-) testing and code generation reports [Mat13].

5.1.2 Comparison

The different approaches to MDD bring different benefits to the development process.

In every case, MDD helps to extend the development beyond the borders of pure software development with manual coding by considering the mapping on hardware (deployment), support of integrated requirements management, or enforcing safety constraints.

5.2 Computing Platform for the Software Deployment

A unique requirement for the safety platform that runs the software for autonomous driving is performance, since the used algorithms are of high complexity and have to process high-bandwidth data streams in real time. Hence, compared to other computing units in a car, they require exceptionally high computational power.

5.2.1 Arrival of Automotive Linux

To solve this problem cost efficiently, recently arrived cheap Single Board Computers (SBCs) like the PandaBoard, BeagleBoard and -Bone, or the Raspberry Pi provide interesting new opportunities, because they can provide the necessary performance for AF. These systems, can in particular run the open source Linux operating system. This trend is known as *Automotive Linux*.

Linux is already pushing forward into the in-vehicle infotainment domain (compare Genivi Alliance [GEN13]), but it is not present in the fundamental systems that control core functions of the car.

BMW Car IT GmbH has projected to use Yocto Linux and ROS on a yet to be determined Linux SBC as a platform for the AF project, and therefore pioneers on the sector of automotive Linux to touch core functionality in a car.

The development process depends on the platform that is chosen, since the platform has impact on the safety lifecycle in terms of testing and deployment automation. It must hence be analyzed, if the platform supports mechanisms to support ISO 26262 compilation, deployment and testing.

Build Process and Deployment for Cross Compilation

In the embedded systems domain, build processes that avoid virtual machines and that want to use higher level programming languages, have to apply cross-compilation. This means that the architecture the compiler is running on (host) is different from the architecture the software is going to be executed on (target). This is enforced by the limited performance and ressources of the target, which make it unable to provide the necessary ressources for the build process.

This brings along a number of problems. Some important software tests, e.g. concerning distributed component communication, that are run in a virtual machine or for a different target architecture have questionable relevance for demonstrating properties of the actual target platform; timing behaviour of a distributed system can be very different on the host and on the target, also, runtime tests are only expressive on a certain target architecture.

This leads to the fact, that ISO 26262 does not accept the testing of the software on the host system only (SIL testing), but require adequate testing also on the target system (PIL and HIL tests).

Linux Environment, Build Process and Deployment

Proposed is a Linux based high performance computing platform that is expected to provide hard real-time support in the future.

OpenEmbedded

OpenEmbedded is a cross-compilation framework for embedded Linux environments able to target many popular architectures for embedded systems [Ope13]. OpenEmbedded consists of two major components:

BitBake is OpenEmbedded's solution to cross-compilation [Ope07].

Its important advantage is that it requires no central make-file that controls the build process, but a set of decentralized build description files that do not have to specify a build order, which is computed automatically. It is therefore an elegant way to automatically resolve the problem of build dependecies for both, run-time and build-time dependencies.

BitBake is also the mechanism to patch the source code, if architecture specific changes are necessary.

The Meta data for BitBake works as input for BitBake and defines package individual dependencies, source code patches and general tasks.

Yocto

Yocto is a layer for OpenEmbedded. It consists of board support packages (BSPs) and a core package. With yocto it is possible to create a custom Linux distribution for a multitude of hardware target architectures.

This helps the AD-development team to solve the problem of portability since the hardware architecture can be changed with little effort and software development efforts are not wasted. Yocto has

5.2.2 A Tool to Supervise the ROS Layer Integration

To simplify the software platform deployment for the AD-project, a required layer for the Robot Operating System ROS has to be implemented to enable the creation of custom Linux distribution using yocto's tools. This is where the final AD applications can run on (compare Figure 3.4).

The developed tool (9.1 Bitbake Build Report Script Tool - Excerpt) is written in bash and uses several Linux tools to provide the additional functionality to document the output of yocto's BitBake build tool and to visualize problems in the build process for a selected set of recipes that have been selected for the build.

5.3 ISO 26262 Part 6 Checked Against the CAP

As has been elaborated in the previous chapters, the full ISO 26262 safety lifecycle cannot be mapped onto the current development process at BMW Car IT GmbH, as the CAP is only a software development process that actually needs to be embedded within a larger product development process that is currently not available and only introduced to the already proposed GSN modelling activity in the enriched process.

However, in coordination with BMW Car IT GmbH we want to find out how well the ISO 26262 safety activities of the software development phase can be mapped onto the CAP based on loose requirements assumptions.

Since the CAP is focussed on iterative development, the question is how well the required activities can be automated, since a high degree of automation is key to keep up

short iteration cycles and not to bother developers with recurring time-expensive process activities. For the AD-software, it is at this time justifiable to expect the requirement for ASIL D for several developed software components, since the final system will have direct influence on the driving process, whichs flaws in terms of functional safety can cause severe accidents.

Hypothesis 3

Ignoring required artefact traceabilities and using assumptions to derive software safety requirements, CAP is able to develop software for ASIL D in accordance with ISO 26262 Part 6 (Software Development) with a high degree of automation using tools that target the C++ programming language.

As a rating scheme we analyze the required topics on: automation ("ATMB"):

$$Activity = (ID, Topic, ATMB, Solution),$$

where we do not want to introduce names for the activities in informal contexts, but identify them by their topic.

In a scheme similar to ISO 26262 table notation [ISO11a, [1-9]-4.2], we define for ATMB:

- **+** well automatable
- **o** partially automatable / supportive tool
- **-** hardly or not automatable

In addition, if there is no meaningful value, we define the value "n.a.":

- **n.a.** a rating or definition is not applicable

As sources for the tool chain have been identified solutions from Parasoft (PS) [KTDA12], Gimpel Software (GS) [Gim13b], Grammatech (GT) [Gra11] and Absint (AB) [Abs13], as well as stand alone tools from ROS or tools that are already available in the process.

5.3.1 Safety Lifecycle Topics for the Software Development

All tools that touch the system under development require the highest tool confidence level 3 (TCL 3) in the participating software tool chain, if they have impact on safety-relevant system components and the tool effects are not supervised or outputs are not reviewed [ISO11a, 8-11.4.5.2], which is actually desirable in order to achieve a high degree of automation in the process.

For choosing a first set of tools it is not too important to select only TCL 3 tools, since tools with lower confidence levels can be qualified later and even their use over time contributes as a qualification property (proven-in-use argument, qualification of a software tool [ISO11a, 8-11.4.6.1]). However, tool qualification will not be treated in the following.

Initiation of Software Development (6-5)

Modelling and coding guidelines [ISO11a, 6-5, Table 1].

ID	Topic	ATMB	Solution
1a	Enforcement of low complexity	+	PS
1b	Use of language subset	+	GS, PS
1c	Enforcement of strong typing	+	GS, PS
1d	Use of defensive implementation techniques	+	PS
1e	Use of established design principles	+	PS
1f	Use of unambiguous graphical representation	+	PS
1g	Use of style guides	+	PS
1h	Use of naming conventions	+	PS

Fulfilling requirements for modelling and coding guidelines is for the most part well automatable, since there exist established and well understood static checking activities.

1a) Code complexity can be analyzed and enforced through graph-theoretic complexity metrics (analyzing logical complexity) and counting methods (analyzing physical size) [Par12, p. 6].

An important graph-theoretic complexity metric is the cyclomatic complexity after McCabe [McC76], who recommended a cyclomatic complexity of ≤ 10 to developers in a field study [McC76, p. 314].

Counting methods include determination of LOC for classes and methods. Upper boundaries should be discussed in the team and captured in a rationale.

1b) The definition of safe language subsets is a traditionally performed method in safety engineering and in the automotive industry. Normative standards are published on a regular basis. An important example is the MISRA-C/C++ standard that includes a subset definition of safe C/C++ language subset [Gim13b].

1c) If a programming language is weakly typed, strong typing should be enforced [ISO11a, 6-5, Table 1, Note c]. This can be checked with linting tools [Gim13c].

1d) Defensive implementation is a vague term. Definitions in literature include techniques such as avoiding self-designed solutions and to prefer library solutions [SB96, p. 174] and to check return values of functions even if it is optional (e.g., error codes that are encoded as integer return value) [Par12, p. 6].

1e) ISO 26262 does not define, what an established design principle is, so it is again necessary to make assumptions.

For object-oriented programming languages, a key idea to re-use established design principles are software pattern [GHJV95], where it was exactly the intention of the authors to

describe recurring software structures that solve recurring problems based on evidence from successful software projects.

It is indeed of some popularity in software engineering to encourage the use of frameworks for generating program skeletons (e.g., “scuffolding” and that was the one prominent feature that made Ruby on Rails popular). The presence of correctly implemented software pattern is, however, not easily verifiable automatically, although there exist approaches for high-level programming languages like Java [BBS05].

Other established design principles could include architecture pattern or the use of established system design modelling languages.

1f) Graphical representations of systems, such as class diagramms for object oriented programming can be generated automatically with many IDEs. Most require a little manual fine-tuning, but the general generation process can be automated.

This requirement should also consider code formatting [Par12, p. 6].

1g) Checking the code against specific coding conventions [Par12, p. 6].

1h) The same as in g) holds for naming conventions [Par12, p. 6].

Specification of Software Safety Requirements (6-6)

This step cannot be performed due to missing technical safety requirements, but it was already introduced, how a GSN safety case with assumptions can prepare and support the development process at this stage.

Software Architectural Design (6-7)

The purpose of this activities is to develop a software architectural design that realizes the software safety requirements and to verify this design [ISO11a, 6-7] in order to create predictability of the system, e.g., upper bounds for stack and heap usage [ISO11a, 6-7.4.17] or execution times [ISO11a, 6-7.4.17], [Abs13] or to prepare the software for partitioning and freedom from interference arguments on the hardware-software level [ISO11a, 6-7.4.11] to improve the desirable system property of loose coupling.

The focus of software architectural design in accordance with ISO 26262 lies in particular in component based design and the allocation of requirements to the planned components, because each software component “shall be developed in compliance with the hightest ASIL of any requirements allocated to it” [ISO11a, 6-7.4.9].

Principles for software architectural design [ISO11a, 6-7, Table 3].

ID	Topic	ATMB	Solution
1a	Hierarchical structure of software components	o	ROS rqt_graph

1b	Restricted size of software components	o	trivial
1d	High cohesion within each software component	o	✘
1e	Restricted coupling between software components	-	✘
1f	Appropriate scheduling properties	n.a.	✘
1g	Restricted use of interrupts	o	GS

If no MDD is used, tools in this process stage can only help to rate the design. Automated satisfaction of these requirements is therefore hard. Constraints that are of trivial nature (in particular size constraints such as 1b) can be statically checked. An exact definition of appropriate sizes is, however, not given. This makes the developers responsible to give a rational of what is appropriate and why.

1a, b, g) These topics can be supported by MDD through code generation and verified by linting tools or ROS tools.

1d, e) Cohesion and coupling are topics where metrics and thus checking tools are hard to find and recommend at the moment.

1f) Automated static checking of scheduling properties can only be done, if there is a temporal logic description of the software requirements (e.g., LTL) that is guaranteed to hold for the code. This again, is hardly possible without MDD. We found that tools like AutoFOCUS (5.1.1.1) can provide such functionality, but it is hardly possible to use LTL-requirements in an agile process, since they are likely to be too time consuming to maintain.

Mcnsms. for error detection at the software architect. level [ISO11a, 6-7, Table 4].

ID	Topic	ATMB	Solution
1a	Range checks of input and output data	-	✘
1b	Plausibility check	-	✘
1d	External monitoring facility	o	✘
1e	Control flow monitoring	-	n.a.
1f	Diverse software design	-	n.a.

These topics cover the implementation of graceful software degradation strategies and software watchdogs in order to provide software solutions for the correction of system malbehaviour in other hardware or software parts.

1a) The software must implement mechanisms to react to invalid values that are out of range. This is a design task that can only be done by an engineer with understanding of the system.

1b) Plausibility checks are performed by comparing actual data to the range, type and dimension of expected data. This means, knowledge about system elements or components is used in order to reject data. Implementing such knowledge is a manual task that can hardly be automated.

1d) To detect nonresponsive applications and trigger resets, software components can watch other software components (software watchdog). This is potentially automatable but no tools for a software watchdog generator has been found.

1e) Prevention of function performance in a wrong sequence or at the wrong time monitoring the program flow at execution time. This can partially be automated using standard services provided by the execution platform, e.g. AUTOSAR. If such a technology can be used is, however, at the moment uncertain, so we apply a conservative rating.

1f) Achieving the same functionality with different technical and methodical solutions is automatable for some approaches through model transformation, i.e., MDD.

Mcnsms. for error handling at the software architect. level [ISO11a, 6-7, Table 5].

ID	Topic	ATMB	Solution
1b	Graceful degradation	n.a.	✘
1c	Independent parallel redundancy	-	✘

1a) This can only be done by an engineer with understanding of the system.

1b) This should be realized as dissimilar software in parallel paths [ISO11a, 6-7, Table 5], which can at best be supported by MDD.

Mthds. for the verification of the software architect. design [ISO11a, 6-7, Table 6].

ID	Topic	ATMB	Solution
1b	Inspection of the design	o	IDE, ROS rqt_graph
1c	Simulation of dynamic parts of the design	o	MDD
1d	Prototype Generation	o	Jenkins ✔
1f	Control flow analysis	+	GS, PS, GT
1g	Data flow analysis	o	PS, GT

1b) A (design) inspection [ISO11a, 1-1.67] cannot be automated, since it's a manual activity per definition [IEE97, p. 13]. It is a (peer) review for anomalies, where usually a checklist of topics is processed under moderation [ISO11a, 1-1.67 Notes 1-4]. It can be supported and supervised by architecture visualization tools, both for statical and dynamical analysis.

1c) The property that classifies a dynamic system is time-dependency. Requires an executable model [ISO11a, 6-7, Table 6].

1d) Continuous builds can help to force developers to keep up working builds of the software. Already implemented in the development process with Jenkins.

1f, g) Can be statically checked to detect errors to trigger semantic program irregularities, e.g., uninitialized variables [Gim13a]. These techiques can also be contained in the compiler, in particular in its optimizer.

Software unit design and implementation (6-8)

Design principles for software unit design and impl. [ISO11a, 6-8, Table 8].

ID	Topic	ATMB	Solution
1a	One entry and one exit point in functions	+	GT
1b	No dynamic objects or variables	+	GT
1c	Initialization of variables	+	GT
1d	No multiple use of variable names	+	GT
1e	Avoidance of global variables	+	GT
1f	Limited use of pointers	+	GT
1g	No implicit type conversions	+	GT
1h	No hidden data flow or control flow	+	GT
1i	No unconditional jumps	+	GT
1j	No recursions	+	GT

1a-j) These principles can be checked statically, e.g. with Grammartech's CodeSonar.

Mthds. for the verific. of software unit design and impl. [ISO11a, 6-8, Table 9].

ID	Topic	ATMB	Solution
1b	Inspection	o	Gerrit ✔
1c	Semi-formal verification	o	✘
1e	Control flow analysis	+	GT
1f	Data flow analysis	+	GT
1g	Static code analysis	+	PS, GT, GS

1a) A code inspection [ISO11a, 1-1.67] cannot be automated, since it's a manual activity per definition [IEE97, p. 13]. It is a (peer) review of the source code for anomalies, where usually a checklist of topics is processed under moderation [ISO11a, 1-1.67 Notes 1-4]. It can be supported and supervised by code review tools. Already implemented in the development process: Gerrit as a moderation and documentation tool.

1c) Semi-formal verification requires the use of a semi-formal notation to model the

desired functions. This can, e.g., be UML class diagramms for classes and their relations to modell objects, and state machines to modell a system in terms of states, events and transitions. If there are no unambiguities in the modell, the verification process can be performed automatically.

1e, f) These analyses are meant to be performed as static source code checking activities at this stage of the development process [ISO11a, 6-8, Table 9 b], [Gra11].

1g) This very general requirement encourages the use of all available static analysis methods.

Software unit testing (6-9), Software integration and testing (6-10)

We exclude testing and test case generation as per definition in the boundaries of this thesis.

6 Evaluation

6.1 Evaluation of the GSN Safety Case Report

We demonstrated the use of GSN to construct an initial high-level safety case for an autonomous driving system.

GSN is a flexible methodology when used to document efforts in implementing functional safety. It is useful to depict and reason about product safety concerns, such as implemented safety functions and custom safety architectures, but it can also be used for process safety arguments, where it can document and justify process tailoring.

A GSN safety case report as proposed has therefore the potential to contain:

- **Assumptions** on requirements for all product development levels of ISO 26262 linking the software development with system development safety goals: ✔
- **Bi-directional traceability** between safety goals and safety requirements: ✔
- System **Context** where a safety concern is valid as required [ISO12, 10-5.3]: ✔
- Software safety requirements as required [ISO11a, 6-6]: ✔
- Chosen safety activities and rationale as required [ISO11a, All Parts, 4-2]: ✔

GSN modelling as introduced creates the required artefact "safety case report" as part of the artefact set "safety case", of which it is the central part and where other safety engineering artefacts are referenced (according to the definitions given in 2.5).

All future artefacts of prospective quality or safety engineering activities that are safety case relevant, can now be referenced and used to document the potentially increased confidence in the system's functional safety.

This approach therefore improves compliance with ISO 26262 process requirements.

6.2 Evaluation of MDD

MDD has the potential to help establish traceability across system design levels and help to tackle the problem of intertwinedness of hardware and software design through an integrated, comprehensive approach.

	AutoFOCUS	Chromosome	Bride	Embedded Coder
Version/Release	AF3 2.4 (nightly build)	v0.4	0.2.0	6.5 / R2013b
Platform	Eclipse	Eclipse	Eclipse	Matlab
Open Source	✔	✔	✔	✘
Stable Releases Used in Production	✔	✘	✘	✔
Integrated Requirements Management	✔	✘	✘	✔
Traceability: Requirements, Functional Safety Concept	✔	✘	✘	✔
Traceability: Functional Safety Concept, Technical Safety Concept	✔	✘	✘	✔
Traceability: Technical Safety Concept, Implementation	✔	✘	✔	✔
Component-Based Design	✔	✔	✔	✔
Behaviour Models (e.g. state machines)	✔	✘	✘	✔
MDA	✔	✔	✔	✔
Modelling Focus	graphical	text	graphical	graphical
Model Checking	✔	✘	✘	✔
ROS Deployment	✘	✘	✔	✘
HiP-HOPS Export	✘	✘	✘	✔

The question that needs further research, however, is, whether an agile development process can make use of MDD without loosing its agility for the developers.

6.3 Yocto Linux Platform Efforts

The developed tool (Appendix 9.1) helped the development progress through status reports about the ROS layer.

In the course of the thesis, the Yocto ROS layer has largely advanced and is hosted open

source at http://github.com/bmwcarit/meta-ros. The tool is meanwhile proven-in-use, and requires no additional evaluation.

6.4 Evaluation of ISO 26262 Part 6 Checked Against the CAP

Quite a few safety activities in ISO 26262 Part 6 can be performed automatically (Figure 6.1). The most problematic activities are found in the design phase of the software architecture, the mechanisms for error detection and handling, where there are activities necessary that can not or hardly automated due to requirement of overall system understanding and human expert knowledge.

The evaluation for each topic or method table was done in the following way:

The set of chosen activities was based on all very recommended activities vor development according to ASIL D. For each table, the worst automatable activity determines the overall automatability rating.

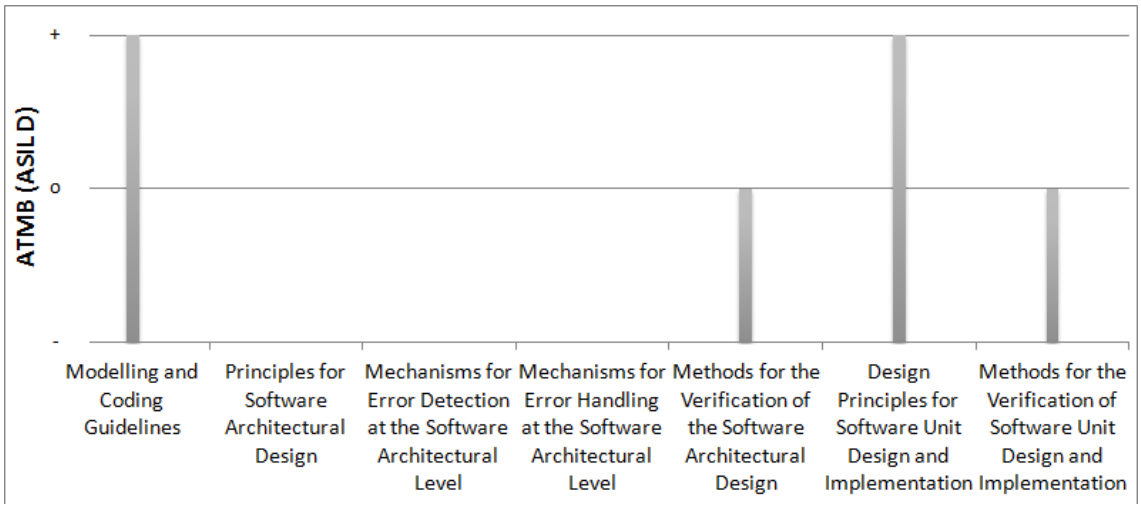


Figure 6.1: ASIL D Automatability of Selected ISO 26262 Method and Topic Tables

The result shows that in particular the principles for software architectural design and the mechanisms for error detection and handling at the software architectural level are hardly automatable or even hard to support by tools.

Hope to improve the situation lies potentially in the use of MDD, which was only sketched in the course of this thesis and not taken into consideration for this particular analysis, where the focus was on manual coding for the software development.

7 Conclusion

We analyzed the functional safety standard ISO 26262 for its compatibility with an agile development process, the BMW Car IT GmbH's CAP. The results showed that the described functional safety lifecycle can not be mapped onto it exhaustively, because the ISO 26262 safety lifecycle is not applicable to a development process with a strong focus on software development only, but requires a development process for system development including product development on the hardware level. By nature, ISO 26262 Part 6 (software development) is hence rather suitable to be mapped onto the CAP, although however, this violates traceability in system design requirements.

However, to prepare the developed software for a system development process that fully implements a functional safety lifecycle, we decided to demonstrate safety case management with focus on software safety requirements and analyzed the general feasibility of applying the software development part of ISO 26262 as standalone part under the aspect of automation.

7.1 Results

- We discussed various safety activities and tool solutions for the initiation of a safety culture at BMW Car IT GmbH and demonstrated the use of GSN to create a maintainable software safety case report and found it promising.
- Evaluating solutions for model-driven development, we found that there are promising approaches available that also support traceability of artefacts throughout several required development phases, e.g., in the tool AutoFOCUS3.
- Only based on the tables in ISO 26262 Part 6, we analyzed if software could be developed under ASIL D with a high degree of automation that does not slow down iterative development and found that it is not completely possible as there are certain design tasks that can or are hardly supported by tools and have to be executed and reviewed by human experts, so tools can only help to document and manage these activities.
- With Yocto Linux we identified a potential platform for the application software deployment and contributed a tool to support the integration of a required software layer.

7.2 Lessons Learned

Learning to apply the ISO 26262 safety lifecycle is hard. The standard contains a maze of methods and terms, which take a lot of time to understand and to learn the proper application. As a developer, one is usually involved only in a limited part of the system development, but safety engineers have to bridge all the gaps between the different involved disciplines and apply the safety lifecycle on all involved activities across all system design levels. By accompanying the entire product development, the safety engineers have to ensure traceability of artefacts among all development phases, where especially the tight intertwinedness of hardware and software development are a challenge for agility in any of those two domains.

Since parts of ISO 26262 are referencing activities and artefacts of other parts, a good understanding of the safety lifecycle can only be gained if the whole standard is read and understood, even if, as in our case, the interest was mainly on software development. Yet, for too ambitious proceeding it is dangerous to get lost in an ocean of cross-referencing clauses.

Finally, even with solid understanding of the standard, it is sometimes not clear, how an artefact should be designed or how an activity should be performed, since there often are no clear, unambiguous descriptions.

7.3 Outlook and Further Research

Implementation of the CAP in a Larger Process Framework

To develop an autonomous driving system, the CAP must be treated as the software development process that is a subprocess of a larger system development process. It remains open, how it can be integrated in such a larger process framework, where an exhaustive product development process, including a hardware development process, is missing.

A case study on how the demonstrated GSN safety case management within the CAP really helps such a superordinate system development process, should be performed. This might include a proposal of which safety argument patterns are suitable to support strong safety arguments for the autonomous driving system.

Agile Model-Driven Development

As indicated, for a full-grown development process in accordance with ISO 26262, MDD is encouraged. It is, however, an open question how agile a development process feels

that makes heavy use of MDD, or how the software development process must balance manual coding and MDD.

Executing BPMN Processes

Modelled BPMN processes don't have to end as "abstract reference processes", but can be input to process engines or workflow management tools. In this case, the CAP's BPMN model has to be refined.

This, however, raises a lot of questions. It is unclear to which extent a process engine should be integrated in the process, since micro management of process participants can easily have negative influence on project success. It is, in general, hard to predict, how process participants accept such mechanisms. This requires field studies and evaluation.

For domains that require reliable software and tool qualification (e.g. the safety domain), it is, in addition, unclear how to qualify a process engine or a workflow management tool, so that is is allowed to participate in the safety process.

Use cases of lightweight process engine integration could be to hint for missing artefacts or forgotten activities.

Safety-in-Use

When the functional safety process becomes mature enough, the safety case and the safety lifecycle must be extended to include safety-in-use considerations. This raises the question for appropriate methods to argue safety-in-use for autonomous driving.

However, GSN as a universal tool for structured assurance cases can be potentially extended to cover also safety-in-use, so the required process enrichment may be well doable.

8 Glossary

Action: Atomic process activity.

Activity: Element of a process that consists of actions.

Activity-Orientation: A process model, where descriptions are focussed on activities, not on artefacts.

Adaptive Speed Control: Also "adaptive cruise control". "[A] system that is capable of automatically adjusting the speed of a vehicle to match the speed of the car or truck in front of it" [Lau13].

ADAS: Advanced driver assistance system.

ALARP: "As low as reasonably possible", often found principle in development of safe systems [Kel98, p. 125].

Artefact-Orientation: A process model, where descriptions are focussed on the artefacts that are to be created and the traceability between them.

ASIL: Automotive Safety Integrity Level [ISO11a, 1-1.6]. Range from A to D, where A is the lowest safety level. See also QM.

Break-by-Wire: See X-by-Wire.

CAP: **C**ar IT **A**gile **P**rocess. The agile development process used at BMW Car IT.

Certification: "[The procedure of testing and evaluating a system] by independent third parties (i.e., a certifying authority) against demanding and standardised criteria, in order to obtain a certificate for release to service" [GPM10].

Component: Subsystems with well-defined interfaces[HDJ+08, p. 1]. Component-based design is a way to potentially achieve separation of concerns in complex systems, which enables mastering complexity [HDJ+08, p. 2].

Component Interference: The property of a component failure affecting—and potentially triggering the failure—of other components that are running on the same host platform as the original component, even if they share no functional dependencies [ZBH+12, p. 333].

Controller: The combination of hardware and software that is used to process sensor data and operate actuators.

Cyber-Physical Systems: "[S]ystems that combine IT infrastructure and functionality with the control of physical processes" [BKM08, V]. Automotive systems are prime

examples of cyber-physical systems[BKM08, ibid.].

DAS: Driver assistance system.

Dependability: Property of a system to be reliable in all aspects. This includes typically safety, maintainability, availability, and security [ALRL04], [DK04, p. 645].

DSL: Domain Specific Language. Helps to handle complexity through domain-specific wording.

Effectiveness: The ability to solve a given, specific task.

Efficiency: The ability to solve a task with low requirements in ressources.

Element: Part of an ISO 26262 "item". An element is considered to be abstract, so software is considered a system element, not a system "item".

Error: Difference between intended and presented behaviour.

ETA: Event Tree Analysis.

EGSN: Extended Goal Structuring Notation. GSN with modular extensions. GSN is used synonymously with EGSN, while the term *EGSN* is rarely used.

Fault: Inherent defect that can lead to failure.

Failure: Unwanted behaviour with a failure mode and a propagation property that can be described in a failure model.

Failure Mode: 1) Inability to perform a function, 2) Unintentional performance of a function, 3) Early performance of a function, 4) Late performance of a function [Hil12, p. 121, in German].

FME(C/D)A: Failure Mode and Effects Analysis. Classifies failure probability, effects and detection rate in their respective mode of operation in a bottom-up approach [Dep80, p. 101-1]. It tries to reveal critical single points of failure and to give an overview of obvious failure modes [RBPR93, pp. 3]. To perform FMEA, a considerable amount of system knowledge is necessary: a functional block diagram of the item under development broken down to the subsystem and component level, function descriptions, the manner of output failures, and a few more [NAS09]. FMEA has been criticized for its assumptions and limitations, e.g., it does not analyze failures based on more than one failing system component, yet it has been found to be rather elaborate.

FMEA can be extended to FMECA for special focus on criticality analysis, or to FMEDA for diagnostic coverage.

Formal artefacts: Artefacts that are based on a formal specification and are most suitable for automatic processing through formal methods.

Freedom from Interference: Constraint that safety-relevant software 1) has protected memory which's integrity is guaranteed (memory protection) and 2) has dependable runtime behaviour in regard to underlying software control flow beeing monitored (program flow monitoring) [WF10].

FTA: Fault Tree Analysis. Top-down method to describe the effect of subsystem faults to the entire system with boolean logic.

GSN: Goal Structuring Notation. Introduced to model the safety argument of a safety case. The term is used in the meaning of EGSN.

HAZAN: Hazard analysis. Synonyms are: quantified risk assessment (QRA) or probabilistic risk assessment (PRA). Description of hazard probabilities [Kle99, pp. 2-6].

HAZOP: Hazard and Operability Study. Identifies hazards and their consequences to system operability, humans and the environment [Kle99, pp. 2-6]. Example: An identified system operation could be "build up pressure in system component X", an applied guide word could be "more". The resulting question could be: "What happens if more pressure is built up in system element X than was planned in the specification?". It is hence necessary to understand the analyzed system to a certain extent in order to perform HAZOP.

HIS: High-Integrity System. A safety-critical or safety-related system [PCGB08, p. 30].

HIL: Hardware in the loop test. The developed system is tested against the final hardware environment. This is the most sophisticated test that comes close to testing of the final product.

HMI: Human Machine Interface. Interface to provide control of machine functions to human beings.

HSI: Hardware-Software-Interface.

Incremental Development: Development of a system as a series of versions, each version having more features as the previous version [Ste87, p. 402].

Iterative Development: Repeatedly improving a system until completion (compare: iteration [Ste87, p. 403]).

Informal artefacts: Natural language or non-formal information. Although frequently present, hard to reuse (compare [Kel98, p. 159]) and almost impossible to automatically process the semantic content [citation needed]. Opposite of formal artefacts. Compare also semi-formal artefacts.

Interference: See "Component Interference".

ISO: International Standards Organization.

Item: "system or array of systems"[ISO11a, 1-1.69]. An item is considered a physical system element, so software is not considered an item.

Lab Car: Laboratory car. Experimental car to test new features.

Linting Tool: A linting tool is a tool that performs static code analysis.

MDD: Model-Driven Development.

Nightly Build: Latest build of a software that—if made accessible to the public by developers—is only meant to be used for testing and evaluation purposes.

OTS: Off-the-shelf. A readily implementable entity not created by the development team that serves a specific purpose in a larger system.

PIL: Processor in the loop test. Testing the application on the final target processor. This test is the step before the final HIL test.

Process-orientation: Prescription or recommendation of appropriate development processes and methods, in contrast to prescription or recommendation of product properties. See also: activity-orientation.

Product: Developed item or array of items that is the ultimate outcome of a development process. This meaning is not given in the vocabulary of [ISO11a, Part 1], however, it can be found implicitly sometimes in ISO 26262, e.g. [ISO12, Part 10, p. 10]: "(...) milestones of the product development".

QM: Quality Management. When no ASIL level applies, the QM level is usually required, meaning the aspect of the system should be developed using appropriate quality management standards.

RE: Requirements Engineering.

Risk: Probability that a hazard occurs combined with the severity of the hazards consequence. Generic formula: $Risk = Probability * Consequence$.

Safety: A property of a system that it will not endanger human life or the environment [Sto96, p. 2].

Safety Activity: Activity that is part of the safety lifecycle.

Safety Case: "A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment" [BB98]. Achieving a valid safety case is the top level goal for the entirity of safety activities and is therefore **the** ultimate goal to proof functional safety.

Safety-Critical: Analogous to safety-related, but with a more critical effect in cause of failure.

Safety Goal: High level safety requirement.

Safety-Related: An item or element is safety related, if it contributes to a safety goal. I.e., it is in any operational mode or operational context related to a hazardous event.

Safety Requirement: Measure, that copes with a hazard when implemented.

SAS: Stand-alone software. Software that is not part of a larger application framework, but only used separately to generate output that is stored or processed by other software.

Semi-formal artefacts: Semi-formal artefacts are in principle eligible for automatic processing through formal methods. They are easier to understand for humans, but carry informational overhead, such as graphical definitions that are superfluous for automatic processing. Compare also formal artefacts and informal artefacts.

SHARD: "Software Hazard Analysis and Resolution in Design" according to [Pum99]. The adaption of HAZOP for the software domain.

SIL: Software in the loop test. The system is tested in a mock software environment that simulates predefined environment conditions in a test run.

Software Partitioning: Achieving freedom from interference through encapsulation of software.

Steer-by-Wire: See X-by-Wire.

System: A "set of elements that relates at least a sensor, a controller and an actuator with one another" [ISO11a, 1-1.129]. There is no definition of controller given in ISO 26262, but it is arguably defined as set consisting of the elements hardware and software.

Task: "An atomic activity that is included within a Process" [OMG11, p. 502]. The complementing non-atomic activity is the subprocess. A Task is used when the work in the Process is not broken down to a finer level of Process Model detail. A task can be performed by humans (human task) or as part of a software system (e.g., process engine task).

TDD: Test-Driven Development.

Tool: Software tool. A piece of software to fulfill a specific purpose.

V&V: Verification and Validation.

Validation: Checking the recorded requirements against the stakeholders' actual requirements [Ste87, p. 409].

Verification: Checking the developed product against the recorded requirements [Ste87, p. 409].

XP: Xtreme Programming. Method of agile software development.

X-by-Wire: System design, where a system to provide an HMI and a system to perform physical tasks are linked only through information exchange [WIHS04, figure 1].

9 Appendix

9.1 Bitbake Build Report Script Tool - Excerpt

Created was a tool to visualize and document the output of Yocto's build engine, bitbake, in order to aid the implementation of a ROS layer for the application software. It depicts the ratio of successfully building recipes (i.e., ROS layer dependencies) in relation to all required recipes for the ROS layer. The reports are traceable to the code repository.

The tool was designed to output .html reports in XHTML 1.0 strict format.

Figure 9.1: Example Meta Reports Generated by the Developed Bitbake Report Tool

```
#!/bin/bash

(...)

bake_all () {
    arg=$1

    array=(‘list_all_available_recipes_with_bakeable_name‘)
    array_length=${#array[*]}

    echo -e "\nChecking_$array_length_recipes."

    # Init the build environment
    cd "$path_to_project_root_relative_to_script"
```

```
    build_directory=build
    source ./oe-init-build-env $build_directory

    # Go to the meta-ros git repository location to gather information
       about commits.
    cd ../meta-ros
    current_git_hash=$(git rev-parse HEAD)

    # The used git command returns an ISO_8601 timestamp. We crop the
       UTC-offset that can be a pos/neg(hh:mm) value, or Z.

    # Satisfies the constraints: Multiple files can be chronologically
       ordered by lexical sorting, Windows character set compatible.
    git_date_filename_compatible=$(git show -s --format="%ci"
       $current_git_hash | sed 's/\ [+ -].*$//' | sed 's/\ Z$//' | sed
        's/-/\./g' | sed 's/:/_/g' | sed 's/\ /-/')

    # Satisfies the constraints: Easy and fast to read, no seconds.
    git_date_pretty_print_temp=$(git show -s --format="%ci"
       $current_git_hash | sed 's/:[0-9][0-9][^:].*$//' | sed 's/-/\./
       g')
    # The date format comes as year-month-day, but we want day-month-
       year.
    git_date_year=$(echo $git_date_pretty_print_temp | sed 's/\..*$//')
    git_date_month=$(echo $git_date_pretty_print_temp | sed 's
       /^[^.]*\.//' | sed 's/\..*$//')
    git_date_day=$(echo $git_date_pretty_print_temp | sed 's
       /^[^.]*\.[^.]*\.//' | sed 's/\ .*$//')
    git_date_pretty_print="$git_date_day"."$git_date_month"."
       $git_date_year"' '$(echo $git_date_pretty_print_temp | sed 's
       /^.*\ //')
    cd ../build

(...)

#main call
standard_action='--bhtmlmeta'
if [ $# -eq 0 ]; then
    eval_args $standard_action
elif [ $# -eq 1 ]; then
    eval_args $1
else
    echo "Error:_Too_many_arguments."
fi
```

Listing 9.1: Bitbake Build Report Script Tool Excerpt of Approx. 650 LOC in Total

Bibliography

[Abs13] AbsInt Angewandte Informatik GmbH, “Relation to Safety Standards”, 2013. http://www.absint.com/qualification/safety.htm.

[Alp12] Alpine Auto, “Do Consumers Want Autonomous Cars?”, 2012. http://www.alpineautotrans.com/?p=326.

[ALRL04] Avizienis, A., Laprie, J. C., Randell, B., and Landwehr, L., “Basic concepts and taxonomy of dependable and secure computing”. *IEEE TDSC*, pages 11–33, 2004.

[ATE10] ATESST2 Consortium, “EAST-ADL Domain Model Specification”, 2010. http://www.atesst.org/home/liblocal/docs/ATESST2_D4.1.1_EAST-ADL2-Specification_2010-06-02.pdf.

[BB98] Bishop, P. G. and Bloomfield, R. E., “A Methodology for Safety Case Development”. In *Industrial Perspectives of Safety-critical Systems: Proceedings of the Sixth Safety-critical Systems Symposium, Birmingham*, London, UK, 1998. Springer. ISBN 3540761896.

[BBS05] Blewitt, A., Bundy, A., and Stark, I., “Automatic verification of design patterns in Java”. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE ’05, pages 224–232, New York, NY, USA, 2005. ACM. ISBN 1-58113-993-4.

[BBvB+01] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., and Marick, B., “Manifesto for agile software development”, 2001. http://agilemanifesto.org.

[Ben04] Benz, S., “Eine Entwicklungsmethodik für sicherheitsrelevante Elektroniksysteme im Automobil”, 2004. http://d-nb.info/989984885/34, Dissertation.

[BK04] Beznosov, K. and Kruchten, P., “Towards agile security assurance”. In *Proceedings of the 2004 workshop on New security paradigms*, NSPW ’04, pages 47–54, New York, NY, USA, 2004. ACM. ISBN 1-59593-076-0.

[BKM08] Broy, M., Krüger, I. H., and Meisinger, M., “Preface”. In *Model-Driven*

Development of Reliable Automotive Services, Second Automotive Software Workshop, ASWSD 2006, Revised Selected Papers. Springer-Verlag, 2008. ISBN 978-3-540-70929-9.

[Bog11] Boghani, P., "Car or Internet? Toss-up for young adults", 2011. http://business.blogs.cnn.com/2011/12/08/car-or-internet-toss-up-for-young-adults.

[BRI13a] BRIDE, "BRICS Component Model", 2013. http://www.best-of-robotics.org/bride/bcm.html.

[BRI13b] BRIDE, "BRICS Integrated Developement Environment", 2013. http://www.best-of-robotics.org/bride.

[BS01] Broy, M. and Stoelen, K., "Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement". Springer, 1. edition, 2001. ISBN 978-0387950730.

[BT04] Boehm, B. W. and Turner, R., "Balancing Agility and Discipline: A Guide for the Perplexed". Addison Wesley Professional, 1. edition, 2004. ISBN 978-0321186126.

[CEA13] CEA, "Papyrus for EAST-ADL", 2013. http://www.papyrusuml.org/scripts/home/publigen/content/templates/show.asp?P=146&L=EN.

[CG13] Cunningham, W. and Goodwin, A., "Six reasons to love, or loathe, autonomous cars", 2013. http://reviews.cnet.com/8301-13746_7-57583372-48/six-reasons-to-love-or-loathe-autonomous-cars.

[CJL+08] Chen, D., Johansson, R., Lönn, H., Papadopoulos, Y., Sandberg, A., Törner, F., and Törngren, M., "Modelling Support for Design of Safety-Critical Automotive Embedded Systems". In *Computer Safety, Reliability, and Security, SAFECOMP Proceedings, Lecture Notes in Computer Science Volume 5219*, pages 72–85. Springer, 2008. ISBN 978-3-540-87697-7.

[DAR07] DARPA, "DARPA Urban Challenge", 2007. http://archive.darpa.mil/grandchallenge/index.asp.

[DDO07] Dijkman, R. M., Dumas, M., and Ouyang, C., "Formal Semantics and Analysis of BPMN Process Models using Petri Nets", 2007. http://eprints.qut.edu.au/7115/1/7115.pdf.

[Dea07] Dearle, A., "Software Deployment, Past, Present and Future", 2007.

[Dep80] Department of Defense, "MIL-STD-1629A", 1980. http://passthrough.fw-notify.net/download/471153/http://sre.org/pubs/Mil-Std-1629A.pdf.

[DK04] Despotou, G. and Kelly, T., "Extending the safety case concept to address dependability", 2004. http://www-users.cs.york.ac.uk/~tpk/issc04b.pdf.

[DPA12] DPA, "Experts say in the future cars will no longer be status symbols", 2012. http://www.chinapost.com.tw/art/lifestyle/2012/09/20/354855/Experts-say.htm.

[DPP12] Denney, E., Pai, G., and Pohl, J., "AdvoCATE: An Assurance Case Automation Toolset". In *Computer Safety, Reliability, and Security, SAFECOMP Proceedings, Lecture Notes in Computer Science Volume 7613*, pages 8–21. Springer, 2012. ISBN 978-3-88579-604-6.

[DT08] Domis, D. and Trapp, M., "Integrating Safety Analyses and Component-Based Design". In *Computer Safety, Reliability, and Security, SAFECOMP Proceedings. Lecture Notes in Computer Science Volume 5219*, pages 58–71. Springer, 2008. ISBN 978-3-540-87697-7.

[DTL08] Denger, C., Trapp, M., and Liggesmeyer, P., "SafeSpection - A Systematic Customization Approach for Software Hazard Identification". In *Computer Safety, Reliability, and Security, SAFECOMP Proceedings, Lecture Notes in Computer Science Volume 5219*, pages 44–57. Springer, 2008. ISBN 978-3-540-87697-7.

[Ern10] Ernst, R., "Formal Performance Analysis and Optimization of Safety-Related Embedded Systems", 2010. http://www.artist-embedded.org/docs/Events/2010/Autrans/talks/PDF/Ernst/ArtistDesign_SummerSchool_Ernst_2010V2.ppt.pdf.

[Eur13] Euro NCAP Advanced, "Autonomous Emergency Braking", 2013. http://www.euroncap.com/rewards/technologies/brake.aspx.

[Fer12] Ferchau Engineering, "Funktionale Sicherheit im Zehnerpack: ISO 26262", 2012. http://www.ferchau.de/news/details/funktionale-sicherheit-im-zehnerpack-iso-26262-969.

[for12] fortiss GmbH, "AutoFOCUS3 - Focus On The System", 2012. http://af3.fortiss.org/.

[for13] fortiss GmbH, "CHROMOSOME in 120 Minutes", 2013. http://download.fortiss.org/public/xme/xme-0.3-tutorial.pdf.

[FR12] Freund, J. and Rücker, B., "Praxishandbuch BPMN 2.0". Carl Hanser Verlag GmbH & Co. KG, 2012. ISBN 978-3-446-42986-4.

[GEN13] GENIVI Alliance, "GENIVI Alliance", 2013. http://www.genivi.org.

[GHJV95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., "Design patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995. ISBN 0-201-63361-2.

[Gim13a] Gimpel Software, "C-lint/FlexeLint for C/C++ Features", 2013. http://www.gimpel.com/html/lintspec.htm.

[Gim13b] Gimpel Software, "PC-Lint", 2013. http://www.gimpel.com/html/pcl.htm.

[Gim13c] Gimpel Software, "PC-lint/FlexeLint Strong Type Checking", 2013. http://www.gimpel.com/html/strong.htm.

[GK12] Graydon, P. J. and Kelly, T. P., "Assessing Software Interference Management When Modifying Safety-Related Software". In *Computer Safety, Reliability, and Security, SAFECOMP Proceedings, Lecture Notes in Computer Science Volume 7613*, pages 132–145. Springer, 2012. ISBN 978-3-88579-604-6.

[Goo12] Google, Official Blog, Urmson, C., "The self-driving car logs more miles on new wheels", 2012. http://googleblog.blogspot.de/2012/08/the-self-driving-car-logs-more-miles-on.html.

[Gov13] Govers, F. X. I., "BMW and Continental team up to develop automated driving co-pilot technology", 2013. http://www.gizmag.com/bmw-continental-autonomous-driving/26671.

[GP06] Galloway, A. and Paige, R. F., "On the use of agile methods for high-integrity real-time systems", 2006. DARP Technical Report DARP-TR-2006-5.

[GPM10] Ge, X., Paige, R. F., and McDermid, J. A., "An Iterative Approach for Development of Safety-Critical Software and Safety Arguments". In *Proc. Agile 2010 Conference*, Orlando, Florida, USA, 2010. IEEE Computer Society. lscits.cs.bris.ac.uk/docs/agile2010.pdf.

[Gra11] GrammaTech, Inc., "Simplifying ISO 26262 Compliance with GrammaTech Static Analysis Tools", 2011. http://www.grammatech.com/images/pdf/iso-26262-compliance-with-codesonar.pdf.

[HDJ+08] Heinecke, H., Damm, W., Josko, B., Metzner, A., Kopetz, H., Sangiovanni-Vincentelli, A., and Di Natale, M., "Software components for reliable automotive systems". In *Proceedings of the conference on Design, automation and test in Europe*, pages 549–554, New York, NY, USA, 2008. ACM. ISBN 978-3-9810801-3-1.

[Hey08] Heydarnoori, A., "Deploying Component-based Applications: Tools and Techniques", 2008. http://gsd.uwaterloo.ca/sites/default/files/2008-heydarnoori-SERA08_0.pdf, Master's Thesis.

[Hil12] Hillenbrand, M., "Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik / Elektronik Architekturen von Fahrzeugen". KIT Scientific Publishing, 2012. ISBN 978-3-86644-803-2.

[HMNS10] Horstkötter, J., Metz, P., Ntima, A., and Seim, W., "Funktionale Sicherheit und ISO 26262 – Entmystifiziert", 2010. http://www.flecsim.de/images/stories/downloads/pdfs/fs-synspace-fusi-26262.pdf.

[HOO+11] Hughes, C., O'Malley, R., O'Cualain, D., Glavin, M., and Jones, E., "Trends towards Automotive Electronic Vision Systems for Mitigation of Accidents in Safety Critical Situations", 2011. http://cdn.intechopen.com/pdfs/13368/InTech-Trends_towards_automotive_electronic_vision_systems_for_mitigation_of_accidents_in_safety_critical_situations.pdf.

[IBP12] IBPI, "CMMI (Capability Maturity Model Integration)", 2012. http://ibpi.org/standard/cmmi.

[IEC10] IEC, "S+ IEC 61508 Commented version - Functional safety of electrical/electronic/programmable electronic safety-related systems", 2010.

[IEE97] IEEE Computer Society, "IEEE Std 1028-1997, Standard for Software Reviews". Institute of Electrical and Electronics Engineers, Inc, 1997. ISBN ISBN 1-55937-987-1. http://www.trempet.uqam.ca/Enseignement/Normes/IEEE_STD1028-1997.pdf.

[ISO11a] ISO, "ISO 26262 International Standard, Road vehicles - Functional Safety, Part 1 - 9", 2011.

[ISO11b] ISO/IEC/IEEE, "Systems and software engineering – Architecture description". IEEE Xplore, 2011. ISBN 978-0-7381-7142-5.

[ISO12] ISO, "ISO 26262 International Standard, Road vehicles - Functional Safety, Part 10 - Guideline on ISO 26262", 2012.

[Jan13a] Janssen, C., "Definition - What does Build mean?", 2013. http://www.techopedia.com/definition/3759/build.

[Jan13b] Janssen, C., "Definition - What does Platform mean?", 2013. http://www.techopedia.com/definition/3411/platform.

[JK11] Jones, L. and Konrad, M., "CMMI V1.3 and Architecture", 2011.

http://cmmiinstitute.com/wp-content/uploads/2013/01/1860_Jones_Konrad.pdf.

[KAAR12] Kämpchen, N., Aeberhard, M., Ardelt, M., and Rauch, S., "Technologies for Highly Automated Driving on Highways", 2012. http://www.atzonline.com/Article/14892/Technologies-for-Highly-Automated-Driving-on-Highways.html.

[KB99] Kenett, R. S. and Baker, E. R., "Software Process Quality - Management and Control". Marcel Dekker Inc., 1999. ISBN 978-0-8247-1733-9.

[KC13] Kotkin, J. and Cox, W., "The World's Fastest-Growing Megacities", 2013. http://www.forbes.com/sites/joelkotkin/2013/04/08/the-worlds-fastest-growing-megacities.

[Kel98] Kelly, T. P., "Arguing Safety – A Systematic Approach to Managing Safety Cases", 1998. ftp://cs.york.ac.uk/reports/99/YCST/05/YCST-99-05.pdf, Dissertation.

[Kle99] Kletz, T. A., "HAZOP and HAZAN: Identifying and Assessing Process Industry Hazards". Taylor & Francis, 4. edition, 1999. ISBN 978-1560328582.

[KM99] Kelly, T. and McDermid, J., "A Systematic Approach to Safety Case Maintenance". In *Computer Safety, Reliability and Security, SAFECOMP Proceedings*, volume 1698 of *Lecture Notes in Computer Science*, pages 13–26. Springer Verlag, 1999. ISBN 978-3-540-66488-8. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.1284&rep=rep1&type=pdf.

[KMFK13] Kuhrmann, M., Méndez Fernández, D., and Knapp, A., "Who Cares About Software Process Modelling? A First Investigation About the Perceived Value of Process Engineering and Process Consumption". In *14th International Conference, PROFES 2013, Paphos, Cyprus, June 12-14, Proceedings, Lecture Notes in Computer Science Volume 7983*, pages 138–152, 2013. ISBN 978-3-642-39258-0.

[Krü12] Krüger, Richard, "Gebrauchssicherheit vs. Funktionale Sicherheit bei BMW", 2012. http://files.hanser-tagungen.de/docs/20120827093732_Kr%C3%BCger.pdf.

[Kri12] Kriso, S., "ISO 26262 - Quo vadis?". In *Automotive - Safety & Security 2012*. Gesellschaft für Informatik e. V., 2012. ISBN 978-3-88579-604-6.

[KTDA12] Kucharski, M., Trujillo, A., Dunlop, C., and Ahdab, B., "ISO 26262 Software Compliance with Parasoft: Achieving Functional Safety in the Automotive

Industry", 2012. http://alm.parasoft.com/how-to-achieve-iso-26262-software-compliance.

[KW04] Kelly, T. and Weaver, R., "The Goal Structuring Notation - A Safety Argument Notation". In *Proceedings of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.66.5597&rep=rep1&type=pdf.

[Lan10] Lange, G., "Der Schlüssel zum erfolgreichen Produkt: Der Produktentstehungsprozess (PEP)", 2010. http://www.cip-delta.eu/downloads/bericht_pep_der-schluessel-zum-erfolgreichen-p.pdf.

[Las13] Lassa, T., published motortrend.com, "The Beginning of the End of Driving - The Autonomous Car Continues to Progress", 2013. http://www.motortrend.com/features/auto_news/2012/1301_the_beginning_of_the_end_of_driving.

[Lau13] Laukkonen, J., "How Does Adaptive Cruise Control Work?", visited: 2013. http://cartech.about.com/od/Safety/a/How-Does-Adaptive-Cruise-Control-Work.htm.

[Lav12a] Lavrinc, D., published on wired.com, "Autonomous Vehicles Now Legal in California", 2012. http://www.wired.com/autopia/2012/09/sb1298-signed-governor.

[Lav12b] Lavrinc, D., published on wired.com, "More Autonomous Vehicles Headed to Nevada Roads", 2012. http://www.wired.com/autopia/2012/12/continental-automous-vehicle.

[Mat13] MathWorks, Inc., "Generate C and C++ code optimized for embedded systems", 2013. http://www.mathworks.de/products/embedded-coder/index.html.

[McC76] McCabe, T. J., "A Complexity Measure". In *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-2, NO.4, D*, 1976. graph-theoreticcomplexitymeasur.

[Mei05] Meiners, J., "Mercedes cancels by-wire brake system; decision a blow to technology's future", 2005. http://www.autoweek.com/article/20051216/free/51216010.

[Mui13] Mui, C., Forbes, "Fasten Your Seatbelts: Google's Driverless Car Is Worth Trillions", 2013. http://www.forbes.com/sites/chunkamui/

2013/01/22/fasten-your-seatbelts-googles-driverless-car-is-worth-trillions.

[NAS09] NASA, "Standard for performing a failure mode and effects analysis (FMEA) and establishing a critical items list (CIL) (DRAFT)", 2009. http://rsdo.gsfc.nasa.gov/documents/Rapid-III-Documents/MAR-Reference/GSFC-FAP-322-208-FMEA-Draft.pdf.

[New11] Newman, P., University of Oxford Press, "Robot cars: more time, fewer prangs, less congestion", 2011. http://www.ox.ac.uk/media/news_releases_for_journalists/111010.html.

[OMG11] OMG, "Business Process Model and Notation (BPMN), Version 2.0", 2011. http://www.omg.org/spec/BPMN/2.0/PDF.

[Ope07] OpenEmbedded Team, "OpenEmbedded Manual", 2007. https://pixhawk.ethz.ch/_media/dev/oebb/oe_manual.pdf.

[Ope13] OpenEmbedded Team, "OpenEmbedded Wiki", 2013. http://www.openembedded.org/wiki/Main_Page.

[Ori11] Origin Consulting, "GSN COMMUNITY STANDARD VERSION 1", 2011. http://www.goalstructuringnotation.info/documents/GSN_Standard.pdf.

[Pal11] Palin, Rob and Ward, David and Habli, Ibrahim and Rivett, Roger, "ISO 26262 Safety Cases: Compliance And Assurance", 2011. http://www-users.cs.york.ac.uk/~ihabli/Papers/2011IETPalinWardHabliRivett.pdf.

[Par12] Parasoft, "ISO 26262 Software Compliance with Parasoft: Achieving Functional Safety in the Automotive Industry", 2012. http://blog.parasoft.com/Portals/69806/docs/iso_26262_software_compliance.pdf.

[PCGB08] Paige, R. F., Charalambous, R., Ge, X., and Brooke, P. J., "Towards Agile Engineering of High-Integrity Systems". In *Computer Safety, Reliability, and Security, SAFECOMP Proceedings. Lecture Notes in Computer Science Volume 5219*, pages 30–43, SAFECOMP Newcastle upon Tyne, UK, 2008. Springer. ISBN 978-3-540-87697-7.

[PCMS05] Paige, R. F., Chivers, H., McDermid, J. A., and Stephenson, Z. R., "High-Integrity Extreme Programming". In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1518–1523, New York, NY, USA, 2005. ACM. ISBN 1-58113-964-0.

[PM99] Papadopoulos, Y. and McDermid, J. A., "Hierarchically Performed Hazard

Origin and Propagation Studies". In *Computer Safety, Reliability, and Security, SAFECOMP Proceedings, Lecture Notes in Computer Science Volume 1698*, pages 139–152. Springer, 1999. ISBN 3-540-66488-2.

[PS11] Pitchford, M. and StClair, B., "Tracing requirements through to verification: How to improve current practices to meet ISO/DIS 26262", 2011. http://johndayautomotivelectronics.com/tracing-requirements-through-to-verification-how-to-improve-current-practices-to-meet-isodis-26262-2/.

[Pum99] Pumfrey, D. J., "The Principled Design of Computer System Safety Analyses", 1999. https://www.cs.york.ac.uk/ftpdir/reports/2000/YCST/05/YCST-2000-05.pdf, Dissertation.

[RBPR93] Reliability Analysis Center; Borgovini, R., Pemberton, S., and Rossi, M., "Failure Mode, Effects and Criticality Analysis (FMECA)", 1993. http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA278508.

[Rei12] Reithofer, N., "Rede von Dr. Norbert Reithofer, Vorsitzender des Vorstands der BMW AG", 2012. https://www.press.bmwgroup.com/pressclub/p/de/pressDetail.html?id=T0128046DE.

[RPU+07] Raedts, I., Petkovic, M., Usenko, Y. S., van der Werf, J. M., Groote, J. F., and Somers, L., "Transformation of BPMN models for Behaviour Analysis", 2007. http://www.win.tue.nl/~jfg/articles/FinalPaper_TransformationForBehaviourAnalysis.pdf.

[SB96] Satir, G. and Brown, D., "C++: The Core Language". O'Reilly & Assoc., 1. edition, 1996. ISBN 978-1565921160.

[Sch08] Schwitzer, W., "Specification and Implementation of a Model-based Deployment-Concept for AutoFOCUS3", 2008. http://www4.in.tum.de/~schwitze/Masters-Thesis-Wolfgang-Schwitzer.pdf.

[Sch11] Schanze, J. (Director), Starring: Weizenbaum, J., et al., "Plug & Pray", Documentary Film, Running Time: 91 minutes, 2011.

[SM01] Succi, G. and Marchesi, M., "Extreme Programming Examined". Addison-Wesley, 2001. ISBN 0-201-71040-4.

[Ste87] Steward, D. V., "Software Engineering". Brooks/Cole, 1987. ISBN 0-534-07506-1.

[Sto96] Storey, N., "Safety Critical Computer Systems". Addison-Wesley, 1. edition, 1996. ISBN 978-0201427875.

[SvS09] Schleuter, W. and von Stosch, J., "Die sieben Irrtümer des Change Managements". Campus Verlag, 1. edition, 2009. ISBN 978-3593391359.

[SW12] Silberg, G. and Wallace, R., "Self-driving cars: The next revolution", 2012. https://www.kpmg.com/US/en/IssuesAndInsights/ArticlesPublications/Documents/self-driving-cars-next-revolution.pdf.

[SWP+12] Slotosch, O., Wildmoser, M., Philipps, J., Jeschull, R., and Zalman, R., "ISO 26262 - Tool Chain Analysis Reduces Tool Qualification Costs". In *Automotive - Safety & Security 2012*. Gesellschaft für Informatik e. V., 2012. ISBN 978-3-88579-604-6. http://www4.in.tum.de/~philipps/pub/automotive2012.pdf.

[SZ10] Schäuffele, J. and Zurawka, T., "Automotive Software Engineering". Springer Vieweg, 4. edition, 2010. ISBN 978-3-8348-0364-1.

[The13] The GSN Working Group Online, "GSN Patterns", 2013. http://www.goalstructuringnotation.info/archives/category/resources/patterns.

[Tre12] Trew, J., published on engadget.com, "New EU legislation requires cars to include autonomous braking system", 2012. http://www.engadget.com/2012/08/05/eu-legislation-requires-cars-to-include-autonymous-braking.

[UC408] UC4, "Workflow vs. Application Automation Tools: Choosing the Right Tool for the Job", 2008. espana.bita-center.com/pdf/UC4_Workflow.pdf.

[Uni12a] University of Hull, "HiP-HOPS - Automated Fault Tree, FMEA and Optimisation Tool User Manual", 2012. Archive containing the report: http://www.hip-hops.eu/hiphopsevaluation.zip.

[Uni12b] University of Hull, "HiP-HOPS goes commercial", 2012. http://www2.hull.ac.uk/science/computer_science/news_and_events/news/hip-hops_goes_commercial.aspx.

[Uni12c] University of Hull, "The Definitive Guide to the HiP-HOPS XML Input File Format", 2012. Archive containing the report: http://www.hip-hops.eu/hiphopsevaluation.zip.

[Wal11] Wallmüller, E., "Software Quality Engineering: Ein Leitfaden für bessere Software-Qualität". Carl Hanser Verlag GmbH & Co. KG, 3. edition, 2011. ISBN 978-3446404052.

[War08] Wardzinski, A., "Safety Assurance Strategies for Autonomous Vehicles". In *Computer Safety, Reliability, and Security, SAFECOMP Proceedings. Lecture*

Notes in Computer Science Volume 5219, pages 277–290, SAFECOMP Newcastle upon Tyne, UK, 2008. Springer. ISBN 978-3-540-87697-7.

[Wea08] Weaver, Rob, "The Goal Structuring Notation GSN", 2008. http://safetyengineering.wordpress.com/2008/04/04/the-goal-structuring-notation-gsn.

[Wei12] Weissmann, J., "Why Don't Young Americans Buy Cars?", 2012. http://www.theatlantic.com/business/archive/2012/03/why-dont-young-americans-buy-cars/255001.

[WF10] Wenzel, T. and Fassl, M. und Kalmbach, J., "Anforderungen der AUTOSAR-Basis-Software für sicherheitsrelevante Steuergeräte", 2010. http://www.elektroniknet.de/automotive/assistenzsysteme/artikel/31185/2.

[WIHS04] Winner, H., Isermann, R., Hanselka, H., and Schürr, A., "Wann kommt By-Wire auch für Bremse und Lenkung?". In *Steuerung und Regelung von Fahrzeugen und Motoren - AUTOREG 2004*, pages 59–71, 2004. ISBN 3-18-091828-4.

[Wik13] Wiki BMW Car IT, "Internal", 2013.

[Win13] Winter, D., WardsAuto, "Google Poses Serious Competitive Threat to Auto Industry", 2013. http://wardsauto.com/blog/google-poses-serious-competitive-threat-auto-industry.

[WMK02] Weaver, R. A., McDermid, J. A., and Kelly, T. P., "Software safety arguments: Towards a systematic categorisation of evidence". In *Proc. 20 th International System Safety Conference, Denver USA, System Safety Society*, 2002.

[ZBH+12] Zimmer, B., Bürklen, S., Höfflinger, J., Trapp, M., and Liggesmeyer, P., "Safety-Focused Deployment Optimization in Open Integrated Architectures". In *Computer Safety, Reliability, and Security, SAFECOMP Proceedings*, pages 328–339, 2012. ISBN 978-3-642-33677-5.