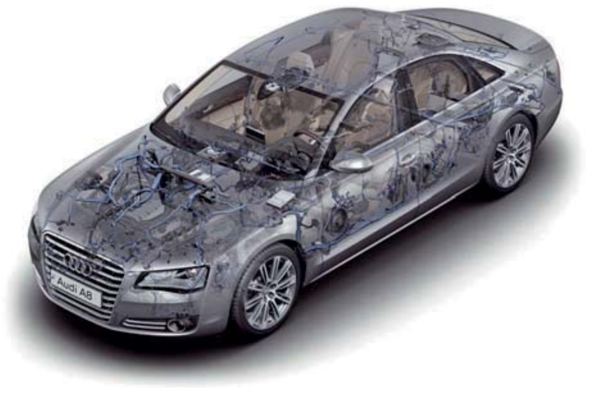
1. Introduction

This chapter will start with a general motivation 1.1 to automotive software engineering as a computer science research field, continuous with the identified problems 1.2 and research questions on software testing within the automotive industry, presents an overview to the thesis contribution 1.3 and concludes with the thesis organization 1.4.

1.1. Motivation

Within the last decade **Software gained importance in cars**. A modern day premium car, *example given* the 2015 Audi A4 [1] may be equipped with up to 90 Electronic Control Unit (ECU), two high resolution displays, two Subscriber Identification Module (SIM) cards, 11 communication networks (Controller Area Network (CAN), FlexRay, Media Oriented Systems Transport (MOST)) and up to six antenna systems (radio, Keyless Entry Start and Exit System (Kessy), WiFi, *etc.*) ensuring wireless communication between the car and various infrastructure. From a computer scientist's perspective a modern day car is a heterogeneous network of embedded computers performing local and distributed tasks. In addition to transport capabilities customers demand up to date entertainment (including music, video or online streaming) and comfort(climate control, massage seats, *etc.*) in a modern day car. Various features, *example given* Advanced Driver Assistance Systems (ADAS), rely on data fusion between multiple sensors and pre calculated values on various ECU. A wide range of sensors starting from simple switches or rotary encoders to advanced Global Positioning System (GPS) Antennas or Radar Sensors will be used to sense the car's environment or interact with the driver. Realizing innovative ADAS like Adaptive Cruise Control (ACC) or Matrix headlamps requires fusioning pre processed measurement data from a camera sensor and a radar sensor as well as a lookup from the road traffic database. This requires four ECU to


(a) Audi A8 Electric wires diagram. Picture extracted from [3]

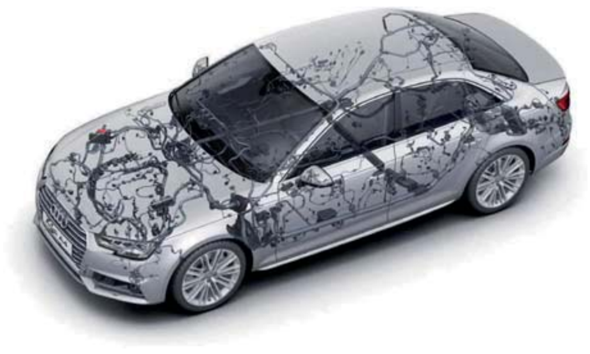

(b) Audi A4 Electric wires diagram. Picture extracted from [1]

Figure 1.1.: Audi A4 and A8 schematic showing the electric wiring harness

work together in realizing one specific driving function. Figure 1.1 shows two modern day premium cars with all their electronic systems and wires, which sum up to 2.5 km in total length, leaving a modern day car to be one of the most technical places within a humans daily live. Following Pretschner *et al.* [2] a 2007 BMW 7 Series contained about 270 software based functions and 67 ECU powered by 65 megabyte of data, the 2010 Model has been expected to contain one gigabyte of software.

The amount of software required to operate a car or an aircraft has increased during past years, even putting the automotive domain in the lead. Robert Charette [4] compares the complexity between an aircraft and a car demonstrating that it requires more Lines Of Code (LOC) to operate a typical car than an aircraft. Dvorak *et al.* [5] states that software realized functionality within a military aircraft raises from below 10% on an 1960is F-4 to 80% on a modern day F-22. Similarly the author states a representative car from General Motors (GM) rises from 100.000 LOC within the 1970is up to 1.000.000 LOC in 2010. Broy [6] states modern day premium cars can contain up to 100.000.000 LOC. In previous works Broy [7] reveals **electronics and software development consumes up to 40% of the whole development budget** nowadays. This is in line with a forecast by Siemens released in 2005, see Aschenbrenner [8]. Similarly a market research from Ehmer [9] states that 20% of the car's total development budget in 2000 will increase to 35% by 2010. They will be distributed among 2% basic software, 8% operating system and 28% application software.

A car recall costs money. Whenever a malfunction within a car is a threat to human life, the law forces an Original Equipment Manufacturer (OEM) to perform repairs. In terms of software this means developing a hotfix and distributing it. Most OEM do not equip their cars with over the air update capabilities this means the car has to be moved into a workshop. Within recent years there have been multiple recalls, compare Figure 1.2. The total number can be extracted by querying the National Highway Traffic Safety Administration (NHTSA) database [10], the amount of software related recalls can been extracted counting each entry containing the terms "software" or "program" within the recall description or required repair action. The number of sold vehicles can be gathered from Wards Auto [11]. The majority of recalls were due to mechanical deficits, but the share of software related recalls is increasing. Within the automotive industry such a recall can cost millions, as an OEM has to pay for contacting the customer, maybe a rental car and the workshop to replace the software. In addition, National Automobile Dealers Association (NADA) published a Whitepaper [12] analysing the impact of recalls on a cars (retail)value. They recognized an increasing number of affected vehicles by recalls within the last decade. Analysing an OEM's average car price compared to competitors they found clear impacts of a recall on achievable market prices, *example given* Toyota dropped by -20% after the 2009 recall on self accelerating models. The authors conclude avoiding a recall will be of economic interest.

Finding bugs later costs more money. A recent Whitepaper from Klocwork [14] stated that finding bugs in early development phases might cost 25$, in a later phase this could climb up to 16.000$. The authors values are based on standard software. Tassey [15] presents multiple analysis concerning the costs when finding bugs in different development stages. The author states 70% of all errors are introduced during the requirements phase but 50% of all bugs will be discovered during the integration testing phase. Further the authors analysed the cost of fixing bugs, see Figure 1.3. In line

[1]Model-year is the first year where a car type is introduced. If a recall is reported in 2008 and 2010, but the car was lunched in 2004, the recall will always bee counted for Model-year 2004. One car can be affected by multiple recalls during its lifetime. The recall data is available via NHTSA [10], the sale statistic via Wards [11].

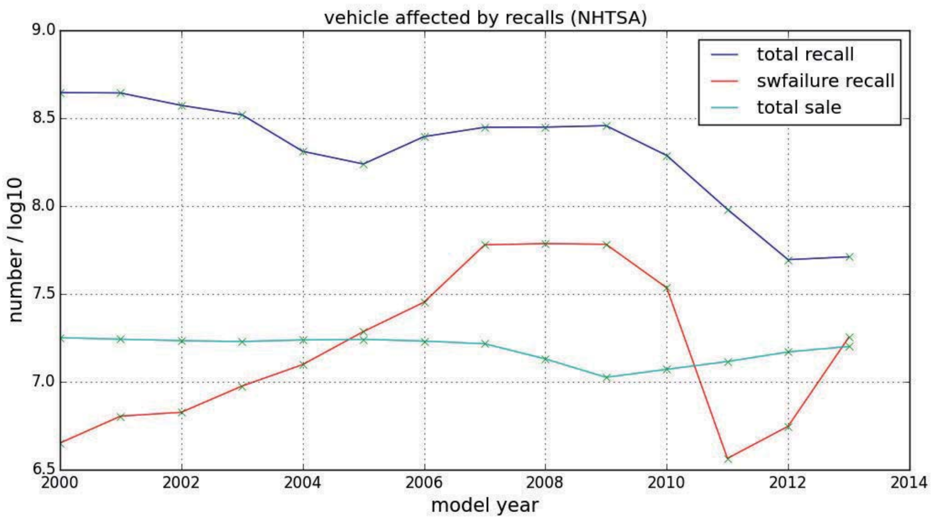


Figure 1.2.: NHTSA recall statistics on Model-year 2000 - 2014, comparing recalls in total with software caused recalls and sales [1]. One can see more recalls than sold vehicles within multiple years. The graphic has been extracted from Altinger *et al.* [13].

with Capers [16] the later a defect is detected, the more expensive it is to fix it. In terms of the automotive industry the numbers will be even higher, as the law forces antecedent tests to be repeated. The majority of tests requires expensive hardware and personnel *example given* to ride prototype cars for a defined mileage.

Within a software's Product Life Cycle (PLC), maintenance can cause the highest costs. ISO/IEC 14764 [17] defines software maintenance as the modification of a product after delivery to a customer. The aim of such a modification is to correct faults, to improve performance or to adopt other quality attributes of the product. As analysed by Kozlov *et al.* [18] 49% to 75% of the total software costs are caused by maintenance. Kozlov *et al.* examined data between the 1970s and 1990s. Recent data published by Confora *et al.* [19] indicate even more than 80% of the total PLC costs are caused by maintenance nowadays. Shull *et al.* [20] states that fixing a software fault during maintenance caused higher costs than finding and fixing it during the early phase of the software's PLC, which is in line with Capers [16]. Even if concrete figures vary, Shull *et al.* [20] analysed that

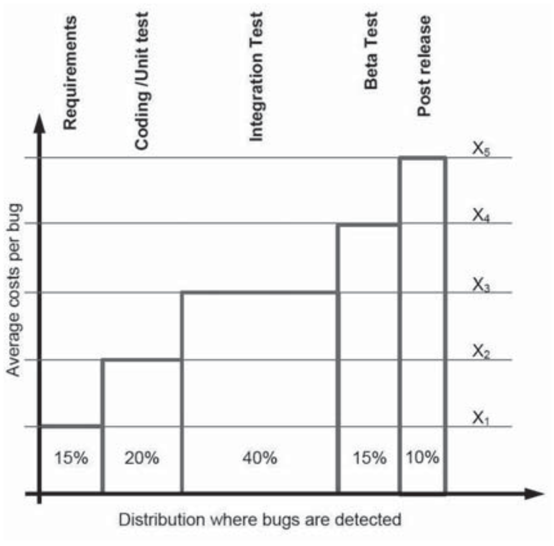


Figure 1.3.: Costs to fix detected bugs, the graphic has been adopted from Tassey [15].

the effort increase by 100:1 for critical defects on large projects and 2:1 for non-severe bugs discovered after release.

Summarizing this chapter's arguments, the share of software increases, fixing bugs out in the field (after release) costs huge amount of money and has a negative impact on the OEM's reputation. Thus finding bugs during early development stages is of economic interest.

1.2. Problem Statement

As stated in the previous chapter, developing software is a complex and expensive task. According to Broy [6] and Aschenbrenner [8] 30-50% from a car's total development costs will be dedicated to software by 2030. Testing has always been a core part of Automotive engineering, as the W-Development process defines a testing stage for every development stage, compare Jin-Hua *et al.* [21]. Indicators where to particular spend TestCase (TC) are welcome to increase efficiency in testing.

The following research questions are identified:

RQ1: What are the common tools automotive software engineers use to specify requirements and write their software?

RQ2: Is it possible to use fault prediction within automotive software projects?

- **RQ 2.1:** Does fault prediction benefit from restrictive development guidelines (Coding-standards and development processes)?
- **RQ 2.2:** What are influential parameters for fault prediction to performing usefully?
- **RQ 2.3:** Do (re)sampling strategies influence the achievable performance?
- **RQ 2.4:** Is it possible to establish Cross-Project Fault Prediction (CPFP) within the restrictive development settings?

RQ3: Which metrics perform best for generated code within automotive software?

- **RQ 3.1:** Which metrics are independent and share no correlation with others?
- **RQ 3.2:** Do metrics represent the occurred bugs?

RQ4: What are good fault predicting methods and what performance can be achieved?

1.3. Thesis Statement

Main parts of this thesis were published on international Workshops and Conferences and are peer reviewed:

- Altinger *et al.* [13] commits to answer **RQ1** by performing a representative survey upon tools and methods.
- Altinger *et al.* [22][2]presents further insights into software methods and development procedures within the automotive industry. This work contributes to answer **RQ1**.
- Altinger *et al.* [23] releases an industry grade dataset containing software metrics on automotive software projects aiming to answer **RQ3** and **RQ3.1** by presenting correlation analysis upon those measurements.
- Altinger *et al.* [24] presents work on Software Fault Prediction (SFP) and CPFP answering **RQ2** by using machine learning classifiers to predict failures. Comparing the achieved performance values with literature **RQ2.1** will be answered. Correlation analysis and information ranking will be used to address **RQ2.2**. Main work will be on **RQ2.4** using state of the art literature methods and comparing their performance. Finally **RQ4** will be answered using a Principle Component Analysis (PCA) on the metric data.
- Altinger *et al.* [25] reports on influences of resampling algorithm to bug prediction performance. This work commits to answer **RQ2.3**.
- Altinger *et al.* [26] presents work on bug analysis to response on **RQ3.2**.

A detailed annotated publication list is given in Section A.1.

[2]This publication is submitted to review and is not published at date of release of this thesis

1.4. Thesis Organization

The thesis will be organized as follows. Starting with Chapter 2 to present the automotive industry as the research area and Chapter 3 containing the related literature with a focus on the field of fault prediction. Chapter 4 reviews a conducted questionnaire survey on tools used to specify, develop and test automotive software. An analysis upon three real world software projects is presented in Chapter 5. The obtained results on fault prediction are contained within Chapter 6. Finally Chapter 7 gives concluding remarks and a preview to further research topics.

2. Field of Study - Automotive Software Development

This chapter gives a short introduction to the automotive industry 2.1 and clarifies some domain specific environment parameters 2.2 along with common testing approaches 2.3. This examination will focus mainly on a computer science perspective.
For a more wider introduction to automotive engineering, including other disciplines such as computer science, the reader is redirected to Winner *et al.* [27], Braess *et al.* [28] and Crolla [29].

2.1. Automotive domain

Compared to the consumer electronic industry the automotive domain has a **rather long PLC**. Volpato and Stoccchetti [30] analysed cars PLC data between 1970 and 2006. They report on small cars to be redeemed by the new model after five years, premium cars after eight years, with a strong trend to shorter cycles. This is in line with Broy *et al.* [31] where they state a PLC is roughly seven to eight years, service and spare parts may last up to 15 years. According to the Kraftfahr Bundesamt (English: German Federal Motor Transport Authority) (KBA) [32] statistically cars in Germany are decommissioned after 8,8 years in use. Considering the average three to four year development phase as reported by Crolla [29], see Figure 2.6, some components development might be 18 to 20 years ago when a car is still on the road. Sabadka [33] predicts a reduction of a cars development time from 40 months to 25 in 2013 and further to 20 months in 2018. Using the VolksWagen (VW) Golf as a case story he analysis a PLC reduction from ten years in the late 1970is to three years in late 2000. In contrast, typical consumer products are replaced every two to three years according to Andrae and Andersen [34], software might be updated within much shorter cycles.

Most software runs on ECU with strict hard real-time constraints, memory and computing power is always limited. A modern day car can be seen as a heterogeneous network of up to 90ECU performing local and distributed computing tasks. Some nodes acquire data via a sensor interface, some pre-process data and some aggregate data, others control actuators. The automotive environment is rather harsh, *example given* the operational temperature is specified between -40° and +120°, shock, Electrostatic Discharge (ESD), vibration, *etc.* Tils [35] presented a rather good overview to all physical requirements to car electronics. These limitations may cause the Central Processing Unit (CPU) to run in throttled computation mode to fulfil operation requirements.

Hartung *et al.* [36] addresses the **variation diversity** within automobiles and visualizes them with examples. Pretschner *et al.* [2] uses 80 components which a customer can order, availability may depend on the country, to calculate 2^{80} variants an OEM can assemble electronics. During production the car is equipped with the ECU, but the actual software configuration is generated and deployed in the production line depending on the configuration the customer ordered. This causes a high number of conditions within the software, to cover all options. Peleska *et al.* [37] released an original software model visualizing the high amount of states and conditions to realize a simple car's turn indicator.

The Ultimate time goal: Start Of Production (SOP), the first day when a new model is built. This day requires all developments to be completed, all software to be tested and all certificates and accreditation documents to be issued. Long planning cycles are invested to solve logistic topics, all components need to pass qualification audits. Crolla [29] gives a brief overview to these milestones, see Figure 2.6. Once the SOP day is defined, customers may no longer order the old model, and logistic does not stock up components from the old model. This means, that from a certain point in time, it is not possible to extend the production of the old model anymore. Assuming a cycle time during assembly of about 70 - 90seconds and an average product price in the five digits regions, a production stop for a day can easily sum up to millions of EUR.

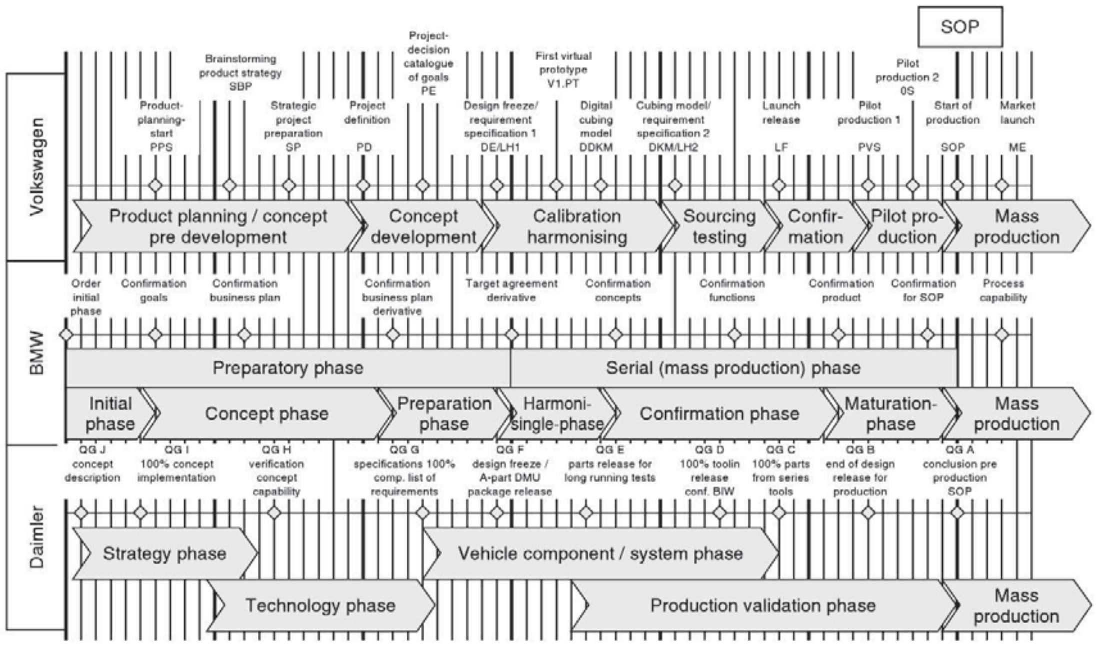

SOP
Volkswagen
Brainstorming product strategy SBP
Project-decision catalogue of goals PE
First virtual prototype V1.PT
Pilot production 2 0S
Product-planning-start PPS
Strategic project preparation SP
Project definition PD
Design freeze/ requirement specification 1 DE/LH1
Digital cubing model DDKM
Cubing model/ requirement specification 2 DKM/LH2
Launch release LF
Pilot production 1 PVS
Start of production SOP
Market launch ME
Product planning / concept pre development
Concept development
Calibration harmonising
Sourcing testing
Confir-mation
Pilot pro-duction
Mass production
BMW
Order initial phase
Confirmation goals
Confirmation business plan
Confirmation business plan derivative
Target agreement derivative
Confirmation concepts
Confirmation functions
Confirmation product
Confirmation for SOP
Process capability
Preparatory phase
Serial (mass production) phase
Initial phase
Concept phase
Preparation phase
Harmoni-single-phase
Confirmation phase
Maturation-phase
Mass production
Daimler
QG J concept description
QG I 100% concept implementation
QG H verification concept capability
QG G specifications 100% comp. list of requirements
QG F design freeze / A-part DMU package release
QG E parts release for long running tests
QG D 100% toolin release conf. BIW
QG C 100% parts from series tools
QG B end of design release for production
QG A conclusion pre production SOP
Strategy phase
Vehicle component / system phase
Technology phase
Production validation phase
Mass production

Figure 2.1.: The automotive milestone plan with SOP as the ultimate goal. Graphic is a licensed copy from Crolla [29] copyright granted by Wiley.

2.2. Development process

Automotive engineering uses the V-Model within all disciplines (power-train, chassis, software, electronics, *etc.*). A recent survey by Bock *et al.* [38] reveals 100% of the interviewee automotive developers are familiar with this development approach. Schäuffele and Zurawka [39] explain this application and its adaptations to automotive in detail. In recent times the W-Development process, see Figure 2.2, is becoming more popular, as stated in a survey by Haberl [40]. This is an explicit testing oriented extension to the well known V-Process. Every specification stage has a corresponding testing stage. Bock *et al.* [38] conducted a survey among fifteen automotive software developers concerning their daily tools and methods usage. He present Matlab, Matlab/Simulink and TargetLink as the most dominating tools out of eight commonly known development products. All respondents are familiar with the V-development process, three quads of them with AUTomotive Open System ARchitecture (AUTOSAR).

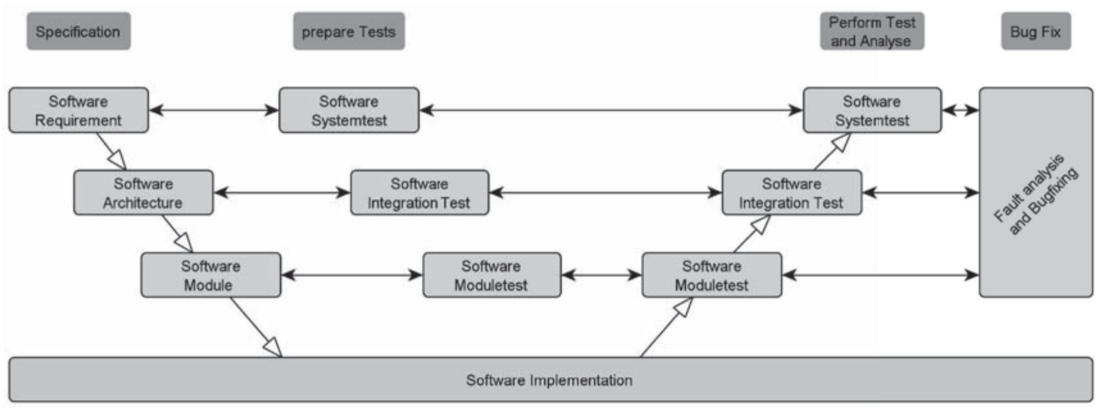


Figure 2.2.: The W-Development process as an testing oriented enhancement from the V-Model, originally presented by Jin-Hua *et al.* [21].

Following Broy *et al.* [31] and Bock *et al.* [38] **the majority of automotive software is developed using model driven or graphical programming approaches**, *example given* Matlab Simulink [41], with automatic code generation, *example given* using TargetLink [42]. Acting according to this process correlating code and model files are available during the development stages.

In accordance with the Motor Industry Software Reliability Association (MISRA) Software development guidelines [43], [44] there are various documents available during all stages:

- *specification:* functionality, timing, memory, processing time, *etc.*
- *software architecture:* modules, target ECU, schedule, *etc.*
- *interface description:* network message layout, method signature, *etc.*
- *static code analysis reports:* MISRA compliance, coverage, *etc.*
- *code review reports:* comments and suggestions
- *test reports:* TC pass rate, code coverage, *etc.*
- . . .

These documents are quite similar to other software engineering disciplines. A concrete workflow for software development is presented in Section 4.2.

A major difference is a rather strict development schedule containing the following milestones:

- *interface freeze*
  Where all network messages and interface (software and hardware) definitions have to be finalized. Beyond this point there is no change of communication messages or data-types.
- *software freeze*
  Where all software modules have to be finished and able to be called upon. There is no need for full functional implementation. Beyond this point no method signature changes are allowed.
- *100% software*
  Where all software modules have to be implemented and be able to pass functional tests. Succeeding this milestone only bug fixes are allowed to be submitted.
- *SOP*
  Where all software has to be finally tested.

Boogerd and Moonen [45] analysed a non automotive software from NXP Semiconductors discovering a similar behaviour. Within the early commits the bug rate increases where the authors conclude this is common behaviour to implement all features in the first place and later on to fix bugs.

As outlined by Schäuffele and Zurawka [39] **not all modules are developed by the same company**. Some do develop modules or components which they deliver as linkable binary. As stated by Pretschner *et al.* [2] an OEM may not even own a full Whitebox specification for third party modules. Another company might be responsible to integrate various modules together with an Operationg System (OS) to be executed at the target ECU. The AUTOSAR standard hosts abstraction layers and defines interface descriptions for all modules to communicate or use services provided by the OS to interact with the ECUs I/O. Figure 2.3 shows a simplified architectural layout. Several vendors offer AUTOSAR OS and basic software components via configurable code generators. The ECU manufacturer needs to implement the hardware drivers, whereas the OEM or system vendor only develops the application modules. As analysed by Dersten *et al.* [46] introducing the AUTOSAR standard to automotive has been beneficial to all software developing parties due to reduced costs on implementation and reuse capabilities. In addition standardized interfaces and Run Time

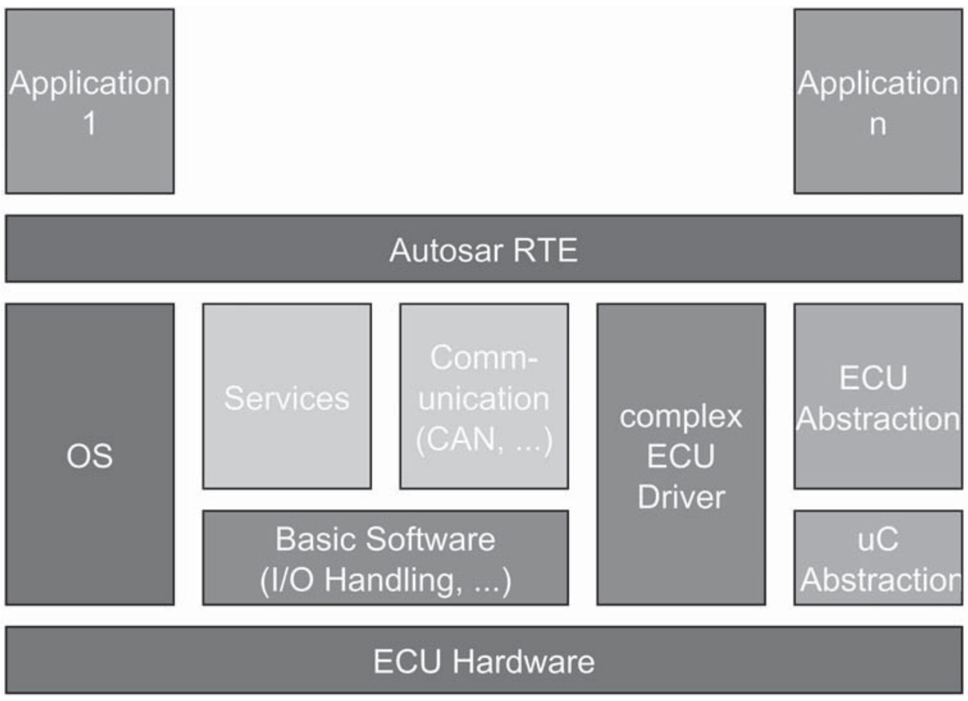


Figure 2.3.: The AUTOSAR architecture schematic as described by [47].

Environment (RTE) enabled developers to simulation and verify their application. The lower layer realizes access to hardware components and provides basic functionality *example given* logging and diagnosis services. Figure 2.3 visualises the architecture. The AUTOSAR RTE provides interfaces similar to an Application Programming Interface (API) and handles the data-flow. The application layers hosts the functionality, it might contain decision logics, controller software, *etc.*
As outlined by Schäuffele and Zurawka [39] the software is organized in modules partially hosting the functionality. A module can consist of a single or multiple software model. An application will assemble all modules and provide the external interfaces.

Developing a car in general requires to follow guidelines and requirements defined by laws (worldwide, national and maybe regional), in addition there are several Norms to be considered. Figure 2.4 lists the most common Norms. ISO/IEC 15504-2: 2003 (Automotive Software Process Improvement and Capability Determination (SPICE)) [48], delivers process documents, ISO 26262, ”functional safety”, [49], guides analysis of hardware and software and is a core part of every architectural decision. Projects are grouped into Automotive Safety Integrity Level (ASIL) QM and A to D, representing the severity of a failure’s consequence, see equation 2.1. *Severity* will range between ”no injuries” and ”life threatening” caused by fatal injuries, *Exposure* ranges from ”incredibly unlikely” to ”high probability” when cases occur under normal operations, *Controllability* ranges between ”controllable” to ”difficult or unable to control”. A system classified as ASIL D will cause harm to human life if it fails in which a highly likelihood of situations occur. A visual example of such a system might be a highly automated car leaving the road due to a software flaw and injuring its passengers in the case of a crash. An ASIL QM functionality will cause no harm in case of failure *example given* a satellite navigation system.

$$Risk = Severity \cdot (Exposure \cdot Likelihood) \tag{2.1a}$$

$$ASIL = Severity \cdot (Exposure \cdot Controllability) \tag{2.1b}$$

2.3. Testing process

Testing a car is a complex task guided by multiple regulations and defined by detailed processes. Following the W-Model, see Figure 2.2, various testing stages have been applied in recent times. Parallel to writing functional requirements dedicated test engineers write test specifications. Every module has to pass multiple tests on test benches prior to its integration assembly in a prototype car.
These are most common the **”in the loop”** tests, *example given* Model in the Loop (MıL), Software in the Loop (SıL), Hardware in the Loop (HıL). These test beds realize module tests in various integration levels. Using model driven approaches *example given* a Simulink Model is the first deliverable to be tested using MıL to pass functional tests for sub functions organized within a single model. These model files are used to derive a generated code which is put on a SıL test, mainly repeating functional tests to ensure correct code generator settings. Putting all modules together and running integration tests is the second part. A SıL will be executed as a simulation running on a computer. Within this stage the *interface freeze* milestone