FH | JOANNEUM
Luftfahrt / Aviation

Bachelor's Thesis

Safety as Service

Service oriented Architectures for safety-critical Systems

Submitted by:	Stefan Lengauer
Registration number:	1210587029
Academic Assessor:	FH-Prof.Dipl.Ing.Dr. Holger Flühr
Date of Submission:	xx August 2015

Declaration of Academic Honesty

I hereby affirm in lieu of an oath that the present bachelor's thesis entitled

"Safety as Service - Service oriented Architectures for safety-critical Systems"

has been written by myself without the use of any other resources than those indicated, quoted and referenced.

Graz, xx August 2015

Stefan LENGAUER,

Preface

This thesis was written as result of the internship at VIRTUAL VEHICLE at Inffeldgasse, Graz from April to July 2015, which was conducted as part of the Degree Programme in Aviation at FH Joanneum in Graz, Austria.

The VIRTUAL VEHICLE research center is an international company, specialized in automotive and rails industry. In total, it has more than 200 employees, splitted up in four main research areas. For the time of my employment, I worked as part of the Electrics/Electronics (E/E) & Software Area, ... more specifically .. functional safety.

During my employment is was part of the EMC2 project, a European project, ... my part soa

This thesis consists mainly of two documents, which were produced during my internship. The first one is a glossary, which aims at defining certain terms and unifying the opinions from different research areas. The second document, was an extensive investigation on how the service oriented architecture paradigm can be applied in a safety-critical embedded system. In detail, with a vehicle as the system.

Although the employment was oriented towards the automotive industry, the functional safety and fault tolerance concepts, reappear in a very similar way in other engineering disciplines, like aviation. Furthermore, the service oriented approach, which is considered in terms of automotive, will most likely become also an important issue for aviation in near future.

Furthermore, the location of the company at Inffeldgasse was another big advantage, for it allowed the conduction of various courses at the Technical University of Graz, alongside the employment.

At this point I want to thank my supervisor from FH JOANNEUM, FH-Prof. Dipl.Ing. Dr. Holger Flühr, as well as my supervisor provided by the company, Dipl.Ing. Helmut Martin. ...

and also Dipl.Ing.Dr. Andrea Leitner and Mr. Mario Driussi, which helped develop the ideas and concepts featured in this thesis during numerous meetings and discussions.

Contents

Abstract

One of the major issues in the automotive industry is the constant growing complexity of E/E (Electrics/Electronics) systems and the thereof resulting fault propagation due to the strong interconnection of the systems. The area of *functional safety* is concerned with the prevention of non tolerable risks in event of error. This is conducted by identifying possible hazards, estimating the potential risks and developing necessary countermeasures, based on these investigations. The requirement therefore is an accurate and thorough comprehension of the observed E/E system.

The service oriented architecture is a design paradigm, which features the concept of software reuse by implementing functionalities as technology independent and loosely coupled services. Although the design paradigm is already standard for Web applications, it has not yet been applied in safety-critical embedded systems.

Thus, this Bachelor's thesis investigates the applicability of service oriented architectures for such systems, with a focus on *functional safety* and *fault tolerance* context.

The first part of the thesis is mainly a glossy, which defines certain terms, which are critical for the subsequent presented concepts. On this basis it is defined what *safety as a service* can mean in general, and how the actual implementation in a given safety-critical system may look like, in order to meant with the requirements specified in the ISO 26262 standard. This is the safety standard for safety-critical E/E systems in vehicles with a maximum gross weight of 3500 kg.

Kurzfassung

List of Figures

List of Tables

List of Abbreviations

AADL . . . . . Architecture Analysis and Design Language
ADC . . . . . Analog Digital Converter
ADL . . . . . Architecture Description Language
API . . . . . Application Programmers/Programming Interface
ASIL . . . . . Automotive Safety Integration Level
BB . . . . . Black Box
C . . . . . Controllability
CAN . . . . . Controller Area Network
CBSE . . . . . Component based Software Engineering
CP . . . . . Communication Profile
E . . . . . Exposure
E/E . . . . . Electrics/Electronics
E/E/PE . . . . . Electrical/Electronic/Programmable Electronic
ECR . . . . . Error Containment Region
ECU . . . . . Engine Control Unit
ES . . . . . Embedded System
ESoA . . . . . Embedded Service oriented Architecture
FCR . . . . . Fault Containment Region
GUI . . . . . Graphical User Interface
HMI . . . . . Human-Machine Interaction
HTML . . . . . Hyper Text Markup Language
HW . . . . . Hardware
IA . . . . . Information Assurance
IDD . . . . . Interface Design Description
IEC . . . . . International Electrotechnical Commission
II . . . . . Information Infrastructure

ISO . . . . . International Organization for Standardization
LIF . . . . . Linking Interface
MISRA . . . . . Motor Industry Software Reliability Association
MPSoC . . . . . Multiprocessor Systems on Chip
MTTF . . . . . Mean Time To Failure
MTTR . . . . . Mean Time To Repair
OOP . . . . . Object Oriented Programming
PCI . . . . . Peripheral Component Interconnect
QM . . . . . Quality Management
RAM . . . . . Random Access Memory
REST . . . . . Representational State Transfer
S . . . . . Severity
SD . . . . . Service Description
SIL . . . . . Safety Integrity Level
SM . . . . . System Management
SOAP . . . . . Simple Object Access Protocol
SP . . . . . Semantic Profile
SW . . . . . Software
SW-C . . . . . Software Component
SoA . . . . . Service-oriented Architecture
SoS . . . . . System of Systems
SoSD . . . . . SoS Description
SoSDD . . . . . SoS Design Description
SysD . . . . . System Description
SysDD . . . . . System Design Description
TDI . . . . . Technology Dependent Interface
TII . . . . . Technology Independent Interface
TMR . . . . . Triple Modular Redundancy
UML . . . . . Unified Modelling Language
WB . . . . . White Box
WDT . . . . . Watch Dog Timer
WSDL . . . . . Web Service Description Language
XML . . . . . Extensible Markup Language

Chapter 1

Introduction

- about soa in embedded systems - related projects (epsilon soa, sira)

Due to the very different comprehension of certain terms by different people from different fields of research The first part of my work during the internship was the creation of a glossary describing certain given terms ... in order to unify the understanding of these terms and create a basis for discussions ans researches.

- The glossary was later taken as standard for this tasks ... - and deals with the terms: *service*, *system*, *system of systems*, *architecture*, *service-oriented architecture*, *configuration*, *static reconfiguration*, *dynamic reconfiguration*, *inter-core communication*, *intra-core communication* and *binding*. The most important parts of this glossary will be covered within this chapter. - furthermore a database with the used literature was created in order to provide the necessary references for the glossary and provide a ... where to find further information for the employees.

- disambiguity safety and security - what is safety and fault tolerance

1.1 Related instances

1.1.1 ISO 262662 international standard

1.1.2 MISRA

1.1.3 AUTOSAR

The remains of chapter 1 contains a definition of the most important terms related to SoA (with respect to automotive) with an extensive description of each term. Those terms are

system, *component*, *service*, *architecture*, *service oriented architecture*, *dependability* and *functional safety*. Each of these sections starts with definitions from different sources. The relevant points of those are united in a "self-created" definition, marked by a box. Subsequently follows some additional information related to the specific term.

1.2 System

This chapter explores various definitions of the term system, as well as related hierarchies and terminology. Because of its reappearance in later chapters, the term *embedded system* is also covered.

1.2.1 Definitions

GENESYS architecture . Definition of system according to Obermaisser and Kopetz: *"an entity that is capable of interacting with its environment and is sensitive to the progression of time"* [3, p.7]. The *environment* is a system itself, which produces input for other systems and acts according to their outputs. Which elements (cf. section 1.2.2) belong to the system, and which to the environment, is a matter of perspective.

ISO 26262 . The ISO 26262 standard defines a system as a *"set of elements that relates at least a sensor, a controller and an actuator with one another"* [14]. This definitions is already a bit more specific with a focus on the automotive industry.

AUTOSAR . According to AUTOSAR, a system is *"an integrated composite that consists of one or more of the processes, hardware, software, facilities and people, that provides a capability to satisfy a stated need or objective"* [16].

There is no indication that system have any internal structure, or that services are exchanged inside of them, which is the case according to the other sources. In this definition the is more like a component (cf. *"It may also be referred to as Component or Device"* [15]).

As result of extensive investigation the following definition emerged:

A system is a hierarchical composed, time sensitive element, which interacts with the environment by processing input and providing output in turn.

It is concerned with satisfying a specific need or purpose and disposes of a, more or less complex, internal structure, which may include hardware, software and data.

For the scope of this thesis, the overall system is assumed to be a vehicle, if not stated otherwise. The environment consists therefore of other vehicles or the surrounding infrastructure. Nevertheless, the entire traffic could also be taken as a system and the environment would then be a different one; for example entities like roads, weather conditions and the like.

1.2.2 System Element

> *"element - system or part of system including components hardware, software, hardware parts, and software units"*, [14]

This definition of *element* is taken from the ISO 26262 standard. It is a very generic term and thus it is not advisable to define it for any entities at a specific layer or with a specific characteristic. Instead, according to the quote, it can be more or less any entity of a system, since a system itself is defined as "set of elements" [14].

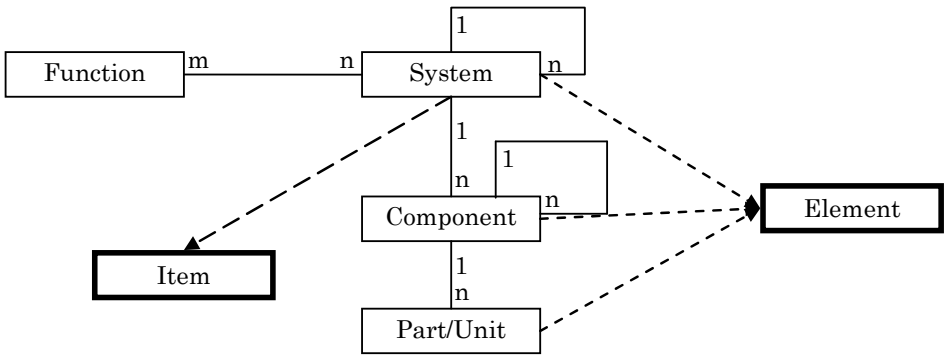


Figure 1.1: Relation of *system*, *component* and *element* according to ISO 26262 [1]. Throughout this thesis the naming convention according to this scheme is applied.

1.2.3 System of Systems

Systems are hierarchical and can be composed or decomposed into sets of interacting constituting systems. Often this is referred to by the term *System of Systems (SoS)* [3, p.7]. Thus, the SoS is in general the level above a given system and is therefore dependent on the

definition of the related systems. With respect to our previous example of a system vehicle, a SoS could be for example the traffic of Graz, with many vehicles participating.

This is in accordance with the definition of system by Arrowhead: *"collaborating, compliant systems become system-of-systems and SoSs can be parts of other, bigger SoSs in turn"* [15].

System of systems features the following five characteristics:

- operational independence of its systems,
- management independence of the systems,
- evolutionary development,
- emergent behaviour, and
- geographic distribution [15].

Concerning this definition, it has to be stated that *geographic distribution* it not necessarily always the case when operating with embedded systems.

1.2.4 System Layers

The parts of a system can be divided into different layers of implementation, for this kind of abstraction makes it easier for human beings to comprehend the overall relations. Unfortunately each field of research features its own way of fractionising systems and most of the approaches are hardly hardly compatible or even contradicting. The following, a segmentation from the GENESYS architecture, revealing a quite hardware and embedded systems oriented point of view, and a more abstract view from the ARROWHEAD project, are investigated in more detail.

System Layers by GENESYS

The GENESYS architecture by Obermaisser and Kopetz distinguish between three different layers, denoted *chip-level*, *device-level* and *system-level* [3, p.44]. An example of the hardware elements at different levels by means of a system vehicle can be seen in figure 1.2.

System Level . The system level consists of *Devices*, which are themselves logically self-contained apparatus. With respect to a system vehicle this could be for example an ECU, a sensor, an actuator or the like [3, p.45].

Device Level . The devices at the system level, contain a certain internal structures themselves. In case of embedded systems these are in most cases *Chips* [3, p.45], like the the AURIX^{TM} chip, which is frequently used in the automotive industry.

Chip Level . According to the implementation layers in the GENESYS architecture, the chip level is the lowest level of implementation. In case of an MPSoC (Multiprocessor System-on-Chip) this level contains the single IP Cores of the chip [3, p.46]

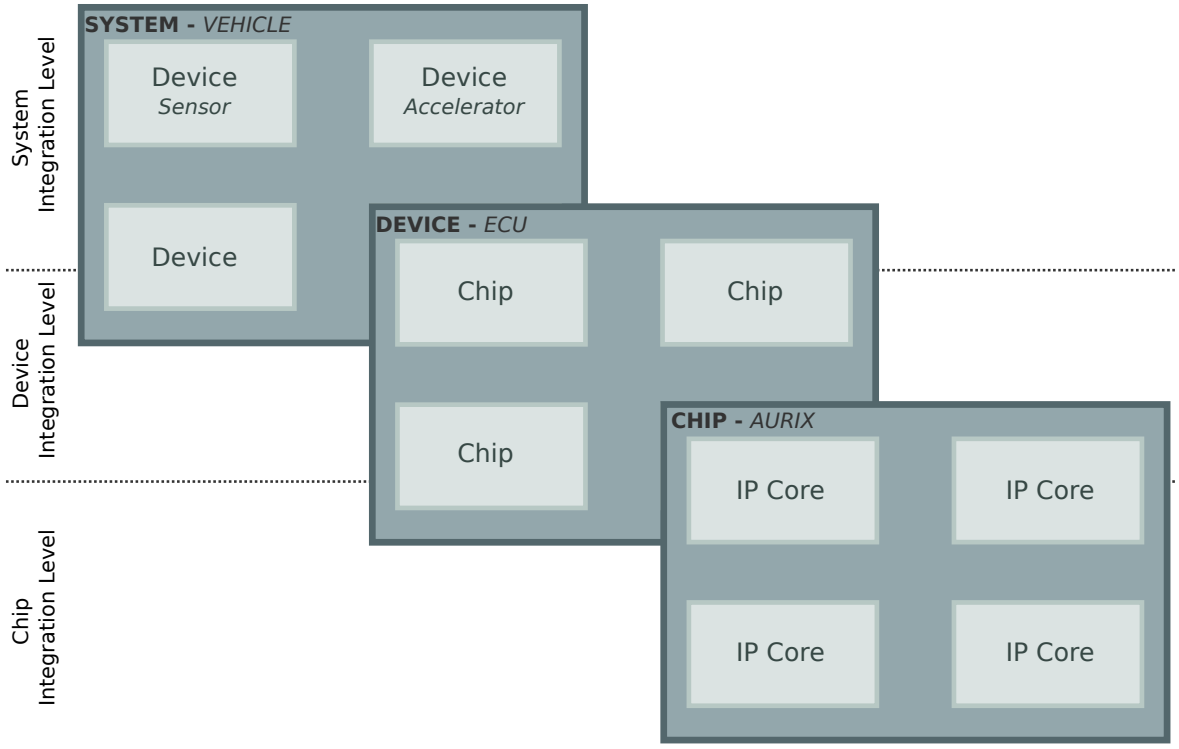


Figure 1.2: The hierarchy of the integration levels of the system "vehicle" with examples

System Layers by ARROWHEAD

The Arrowhead Framework provides an hierarchy by means of documentation documents, which are spitted in the three levels *system-of-systems*, *system* and *service* [15]. Their involved documents and relations are pictured in figure 1.3.

System-of-systems . The levels features the two documents *SoS Description (SoSD)* and *SoS Design Description (SoSDD)*. They differ in the amount of information they are revealing. While the first represents only an abstract view, the second also reveals the implementation of the SoS and its technologies [15].

System . The system level comes with the two documents *System Description (SysD)* and *System Design Description (SysDD)*. Just as at the SoS level the first one features a kind of black box opacity and the second a white box view [15].

Service . The service level contains the four documents *Service Description (SD)*, *Interface Design Description (IDD)*, *Communication Profile (CP)* and *Semantic Profile (SP)*. The service description is referred to in 1.4.3.

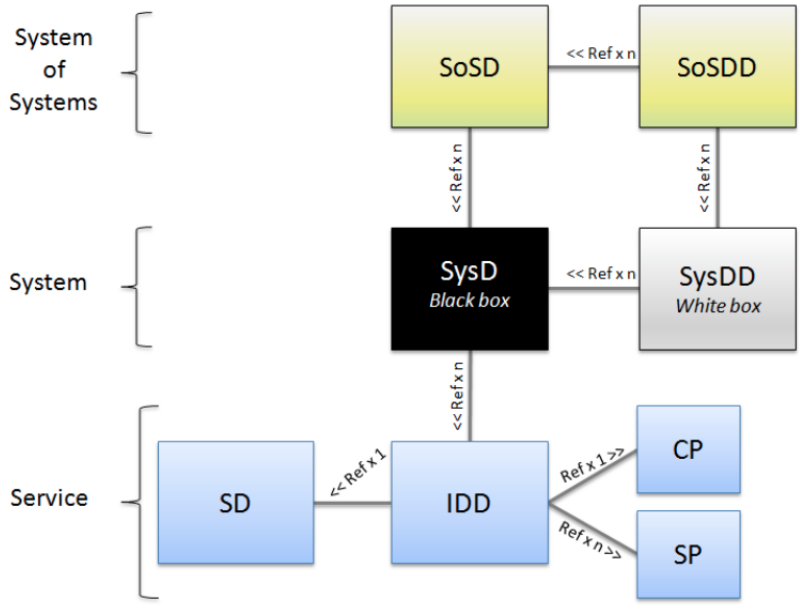


Figure 1.3: Relation of System, SoS and Service to one another [2].

All the documents exist as templates which should be filled out during the development process of a system. They feature a XML like style in order to be human and machine readable at the same time.

Developed systems can then be constituted to SoS by an underlying cloud, as depicted in figure 1.4. All the system have specified and standardised interfaces and work together by means of three *core services*, labeled II, AI and SM.

Information Assurance (IA) . This service is responsible for providing secure information exchange through authorization and authentication [2].

Information Infrastructure (II) . The II service enables the listing of the services in the *service repository* (cf. section 1.6.2) and their discoverability [2].

System Management (SM) . This is the core service for the system of systems composition and features logging and monitoring abilities [2].

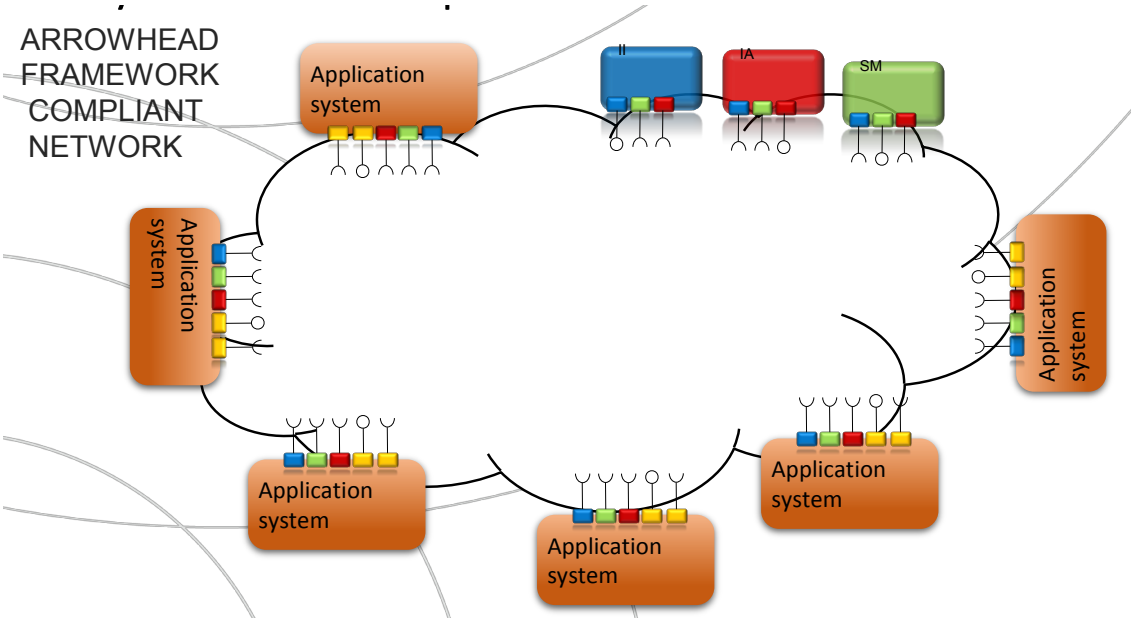


Figure 1.4: Implementation example of an ARROWHEAD framework [2].

1.2.5 Embedded System

Embedded systems are computational modules integrated to physical devices and equipment. The have a predefined set of tasks and requirements and are capable of processing information [17] [18, p.xiii].

Compared to general-purpose computation systems they usually dispose of less processing resources and come with narrower operation ranges, but at the same time they feature a high efficiency by optimally managing the available resources [19, p.283] [18, p.5]. Also, their presence is usually quite unobtrusive, because instead of mice and keyboard the user interface consists of typical input devices like buttons, steering wheel, or pedals.

Embedded systems are *reactive systems*, what means that they perform a continuous interaction with the environment. The connection to the physical environment is realised by means of sensors, responsible for collecting information, and actuators for performing the actual reaction [18, p.8-9].

During operation, embedded systems are in a certain state, waiting for input. When provided with that, they perform computations and generate an output, which is handed back to the environment [18, p.9].

Embedded systems in safety-critical applications have to care about the issues *time constraints*, *dependability* and *efficiency requirements*.

Time constraints . One challenge of Embedded Systems is the meeting of so called *time constraints*, which basically means the conduction of a computation within a specified

time [18, p.8-9] [17]. Kopetz states *"A time-constraint is called hard if not meeting that constraint could result in a catastrophe"* [20].

Dependability . Embedded systems, operating in safety-critical environment, like nuclear power plants, cars, trains or aircraft, must be *dependable*, for they are directly connected to the environment and have immediate impact on it. The Dependability is split up in further aspects - in detail *Reliability* (cf. section 1.7.1), *Maintainability*, *Availability* (cf. section 1.7.2), *Safety* (cf. chapter 1.8) and *Security* [18, p.4-5].

Efficiency requirements . Efficiency is a key concept of embedded systems and is concerned with providing a maximum computation performance while minimizing the required energy. The efficiency is measured in "operations per Joule" and has been been increasing almost exponentially during the last twenty years.

1.3 Component

For the EMC2 project it is crucial to achieve a clear distinction between the term *component* and *service* (cf. chapter 1.4). Due to the historic development of *service oriented architecture* (cf. chapter 1.6), as successor of the *component based software engineering*, those terms have often been put on a level, giving rise to confusion.

Another issue is the term *software component*, which leads to the question, whether components are pure software elements.

In the following some definitions from different relevant resources are presented and discussed, with the aim to eliminate those ambiguities.

1.3.1 Definitions

GENESYS reference architecture . Obermaisser and Kopetz state that a component is a software or hardware unit that performs a specified computation within a given period of time [3, p.38] and communicates with other components by means of specific *interfaces* (cf. section 1.3.2). Like systems, components are hierarchical and therefore dependent on the point of view - from a different viewpoint a quantity of components may be seen as a single component. With respect to this definition all entities in figure 1.2 (*System*, *Sensor*, *AURIX*, etc.) can be denoted as components.

ISO26262 . The ISO 26262 standard supports this concept by its definition of a component as a *"non-system level element that is logically and technically separable and is comprised of more than one hardware part or one or more software units"* [14].

Szyperski . Szyperski features a definition with emphasis on the software oriented point of view: *"Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system"* [21, p.xxi].

AUTOSAR .

> *"Software-Components are architectural elements that provide and/or require interfaces and are connected to each other through the Virtual Function Bus to fulfill architectural responsibilities"* [16].

AUTOSAR defines a *Software Component* as encapsulation for parts of the automotive functionality, but it dictates no specific "granularity", meaning that an AUTOSAR software component might be either a *"a small, reusable piece of functionality (such as a filter) or a larger block encapsulating an entire sybsystem. [22]"*

ARCITURA . Arcitura™ Education Inc. is a global provider of progressive, vendor-neutral training and certification programs and has conducted a lot of research related to SoA. *"A component is a unit of logic that exists as a standalone software program as part of a distributed computing architecture"* [23].

Definition for the EMC2 project

A component is a logical and technical separable hardware or software unit that is capable of performing a specific computation.

It offers an abstraction that simplifies the understanding of complex systems. Therefore they have to be hierarchical, meaning that components can be composed to other, larger, components.

As suggested by this definition the term component should be used for both, software hardware. If not denoted specifically as *software component* or *hardware component*, the term can be referred to both throughout this glossary.

A component can be seen as *black box*, meaning that the more or less complex internal structure is invisible or not of concern for the user. Therefore, other components stay unaffected from modifications of this internal structure, given that the behaviour at the *Linking Interface* (cf. section 1.3.2) remains unaffected [3, p.38-39] [4] [24].

As a self-contained subsystem, it can be developed and tested independently, and latter be used as building block for systems or higher level components. In other words, the components are the basic building blocks of a system [25].

Nevertheless, not everything is a component. According to Sametinger, an algorithm in a book is not a component, but it has to be implemented by means of an arbitrary programming language and equipped with well-defined interfaces in order to become a component [24, p.2-3].

1.3.2 Component Interfaces

The interfaces are necessary for any interaction with other components. In the following two different approaches for describing these interfaces are presented. The first one is from the **GENESYS project** and features a quite implementation oriented point of view. The second definition, by **AUTOSAR**, is more abstract and better adjusted to the service oriented architecture paradigm.

Interfaces by GENESYS

Following the definition from Obermaisser and Kopetz, each component may dispose of up to four interfaces for communication with other entities. The *Linking Interface*, the *Local Interfaces* and the *Technology Independent-* or *Technology Dependent Interface*. They are illustrated in figure 1.5 [3, p.40-41].

Linking Interface (LIF) . The *Linking Interface* is a message based interface and responsible for offering the component's services. The LIF is dependent on the level of integration, e.g. Inter-IP Core LIF at chip level or Inter-Chip LIF at device level. Nevertheless, it is used only for communication to other components at the same layer and also the only place, where a component may provide its services to other components [3, p.9].

The *Linking Interfaces* are always technology agnostic, which means that they do not expose details on their implementation or *Local Interfaces*. Accordingly, the implemen-

tation can be modified, without other components noticing, as long as the specification at the LIF remains unchanged [3, p.9, 40-41].

Local Interface . The *Local Interfaces* establish the connection between a component and its local environment, which could consist of sensors, actuators and the like. If the environment is modified, the semantics and timing of the data should stay the same in order to do not violate the specification. A *Local Interface* could also be mapped to a *LIF* of a component at the next-higher level - this is known *gateway component* and enables different layers to communicate with each other.

Components do not necessarily require local interfaces. Such are denoted *Closed Components* [3, p.40-41].

Technology Independent Interface (TII) . The *TII* is the instrument for configuring and reconfiguring a component, e.g. assigning a name, configuring input and output ports or monitoring the resource management. Starting, restarting and resetting the component is also executed through this interface. The TII communicates with the hardware, the operating system and the middleware, but not with the *application software (service)*, which is reserved for the LIF [3, p.40-41].

Technology Dependent Interface (TDI) . The *Technology Dependent Interface* enables a look inside the component and allows to inspect internal variables and processes. Thus, it is reserved for people who bring a deep understanding of the components internals and is of no relevance for the user of the LIF services [3, p.40-41].

Interfaces by AUTOSAR

AUOSAR follows a slightly more abstract approach concerning the interfaces of a component. According to their definition, a component may dispose of a number of *ports*. A port belongs to one component only and is the interface a component uses to communicate with other components.

Within this context, the term *interface* specifies a kind of contract or specification, on which services can be called at this port, and the format of the data emitted at this port.

There are four different types of interfaces, belonging to the two different communication patterns **Client-Server** and **Sender-Receiver**: [4]

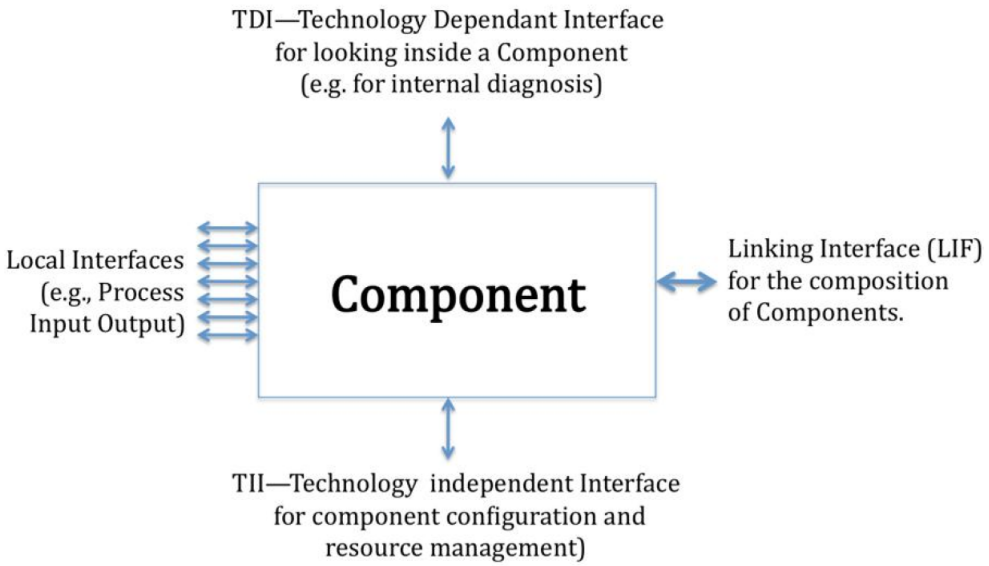


Figure 1.5: The interfaces of a component, with respect to the **GENESYS architecture** [3, p.40]

Client-Server Communication . When this kind of communication is performed the *client-component* requests the a specific service from the *server-component* and sends necessary parameters. The server then processes the incoming request and returns a response. A single component can be a server and a client at the same time [4]. A schematic illustration of this type of communication is pictured in figure 1.6, where SW-C denotes *software component*.

Sender-Receiver Communication . The *Sender-Receiver* approach is a bit different. The task of the sender is to distribute his information to one or more receivers, without ever getting a response in form of data or control flow. In fact he does not even know the number or identity of the receivers. Those have to decide on themselves, how to deal with the received data (cf. figure 1.7) [4].

The different ports and their properties are visualized in figure 1.8.

1.4 Service

The perception of the term *service* is quite wide spread and influenced by a persons experience and environment (e.g. related research area, industries, etc.). Unsurprisingly, this lead to numerous different definitions, supporting various, even contradicting statements.

Accordingly, definitions from several different sources from various fields of research are investigated in the following. The, apparently, most important characteristics, stated in those,

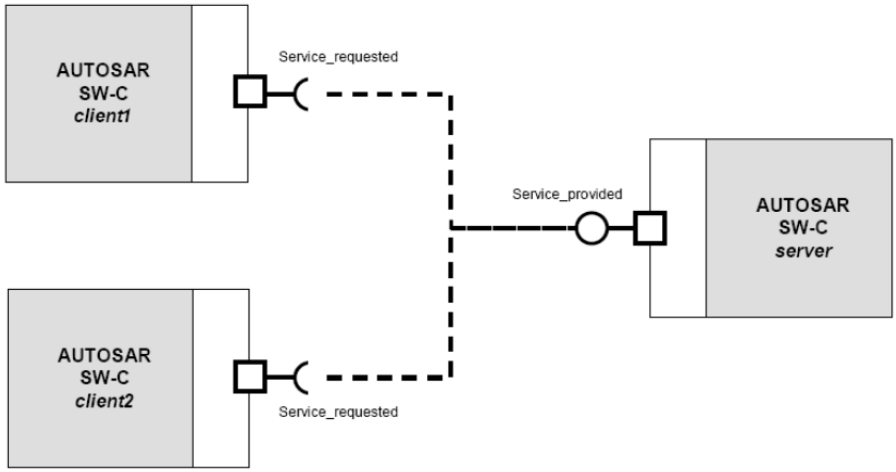


Figure 1.6: Illustration of the client-server communication [4].

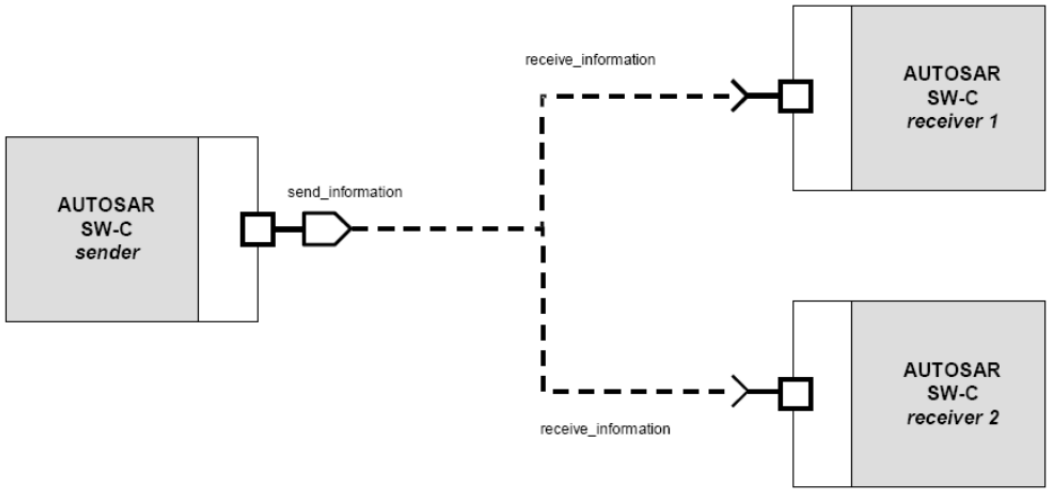


Figure 1.7: Illustration of the sender-receiver communication [4].

have then been concatenated in a rather generic definition, which should be suitable for, more or less, anyone.

1.4.1 Definitions

GENESYS Reference Architecture . The definition by Obermaisser and Kopetz:

> *"A service is what a system delivers to its environment according to the specification. Through its service, a system can support the environment, i.e., other systems that use the service"* [3, p.8].

This definition deals with the terms service from a system point of view. The concepts of *system* and *environment* are referred to in chapter 1.2.

Arcitura .

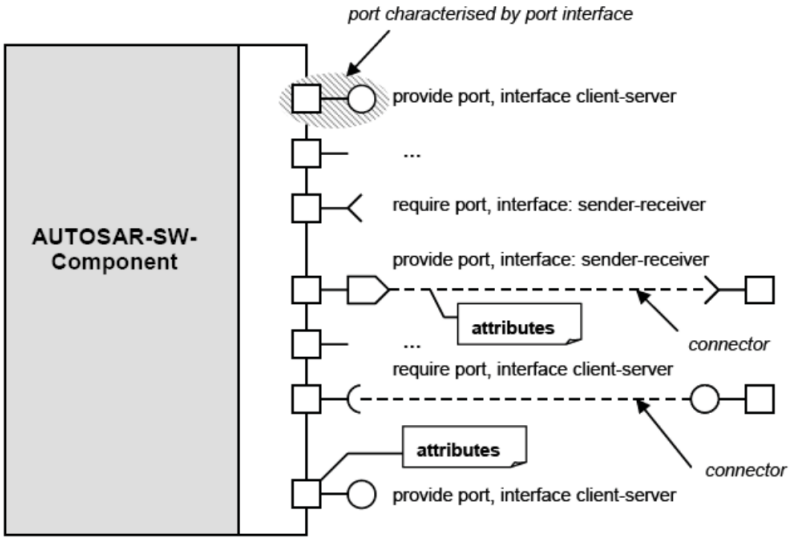


Figure 1.8: Illustration of the different ports at a single component [4].

"Each service is assigned its own distinct context and is comprised of a set of capabilities related to this context. Therefore, a service can be considered a container of capabilities associated with a common purpose (or functional context)" [23].

Erl, Bennett et al. .

"A service is a unit of logic to which service-orientation has been applied to a meaningful extent. It is the application of service-orientation design principles that distinguishes a unit of logic as a service compared to units of logic that may exist solely as objects, components, Web services, REST services, and/or cloud-based services" [11, p.27].

The Representational State Transfer (REST) is a software architecture style, consisting of guidelines and best practices for building scalable web services.

Arrowhead .

"A service is the core building block of SOA, and is basically a software application performing some task, with a formal interface described using a standard description framework." [9]

An important concept, which comes with this quote is, that service have certain interfaces (cf. section 1.4.3).

AUTOSAR . There are two quotes by AUTOSAR. The first one deals with service in general, while the second describes an *AUTOSAR service*, the kind of service that could be applied in an AUTOSAR system.

> *"A service is a type of operation that has a published specification of interface and behaviour, involving a contract between the provider of the capability and the potential clients"* [16].

> *"An AUTOSAR Service is a logical entity of the basic software offering general functionality to be used by various software components. The functionality is accessed via standardized AUTOSAR Interfaces"* [16].

A term, which appears in in connection with services, and frequently leads to misunderstandings is *business function*, with some sources setting it on a level with service. In order to avoid any difficulties in terms of understanding the term *business function* should generally be avoided.

Definition for the EMC2 project

The presented quotes show various different approaches and options, emphasising the various different understandings.

The following definition is a collection of the relevant aspects from various sources and should serve as unified knowledge base for the scope of the EMC2 project.

As part of a system, a service is an independent logical unit with at least one capability and well defined interfaces, which have to be fully described by the service contract.

Services are the building blocks of a service oriented architecture and the containers for the functionalities a service provider offers to its consumers.

In order to be applied in a service oriented architecture, a service must comply with certain key concepts.

The terms *service provider* and *service consumer* are described in detail in chapter 1.6. The required key concepts are the subject of the next section.

1.4.2 Key concepts of a service

Erl, Bennett et al. [11, p.27] state eight specific characteristics any service should posses. Those have been altered and extended with some attributes from other sources, resulting in the following listing with a total of nine characteristics. In some occasions there have been different nominations of equal characteristics. In this cases all alternative labels are stated.

Opacity/ Encapsulation/ Abstraction . According to Erl, *abstraction* means to *"hide information about a program not absolutely required for others to effectively use that program"* [26, ch.8.1.]. Services hide an internal logic, which could be implemented by means of any suitable programming language or operating system. This allows the logic and implementation of the service to evolve over time, while still providing the functionality as it was originally published [26, ch.8.1].

From the service consumer's point of view the service appears as a black box, which does not impart anything of the underlying implementation or how the returned information is generated [27] [28] [9] [11, p.27].

This black box encapsulation disables any modification of the service by the user and is often referred to as *service interface level abstraction* [28].

Reusability . The idea of software *reuse* exists since the early days of software engineering. In SoAs it is the key concept of a service and a necessary basis for many of the other concepts would not even be possible without it [26, ch.9.1.] [11, p.27].

Basically, this design principle aims at making a service applicable for more than just one specific use case. Usually, this means that the programming logic becomes more generic, allowing a wider range of application.

At this point, it should be noted, that the terms *reusability* and *reuse* are not equal. The former is the design principle, while the latter denotes the result which should be achieved by applying the concept of reusability.

Composability . Service are building blocks and thus existing services can be used in order to compose other, possibly more advanced, services through *service orchestration* or *service choreography* (cf. section 1.6.3).

With orchestration, one service acts as a coordinator between all services involved, in contrast to choreography, where all composed services work independently with each other in a completely distributed manner [27] [9] [28] [11, p.27].

In SoAs (cf. section 1.6) the composing may take place at runtime [28].

Loose coupling . If two or more artifacts are somehow connected within a technical context, this is referred to by the term *coupling*. It indicates that two or more of "something" exists and is a measurement of the strength of their relationship, which is given by the amount of dependencies [26].

SoAs should feature a *loose coupling*, which means that dependencies should be reduced as far as possible [26]. This is achieved by using standardised interfaces, what offers the service providers great flexibility in choosing design and deployment environment for offering their services [28] [9]. Well defined interfaces also allows a simple exchange of components by components from different vendors [29].

An example for this concept are web browsers. The service, a web browser provides to the end user, could be described as "interpret the HTML files and illustrate them in a user friendly way". No matter which web browser is used, the end result will stay almost unaffected, thanks to the well specified applied protocols.

Discoverability . The concept of *discoverability* comes with certain requirements:

- services have to constantly communicate the meta information they want to make public and all alternations,
- this information should be centrally stored and maintained in consistent format and
- the meta information must be accessible and searchable by those who want to use this resource [26, ch.12.].

The artifact that stores the service information in SoAs is the *service repository* (cf. section 1.6.2).

The discoverability of a service is not only critical during the runtime of a SoA, but also during the development process, where it provides answer to the question whether a certain functionality already exists or has to be built. Thereby redundancies are reduced or prevented altogether [26, ch.12] [9] [28] [11, p.27].

Self-description . Service provider have to provide their clients with all the relevant information in form of a service description. This includes syntax, semantic and behaviour [28].

Statelessness . A *state* is referred to as the general condition of something. In computational systems a state can be represented by temporary data, describing the state. SoA services are frequently required to hold a certain amount of *state information* through the lifespan of a service composition in order to fulfill their functionality [26, ch.11].

Nevertheless, services can also be stateless and in order to optimize reusability services should aim at minimizing their state information and their holding time for messages [28] [11, p.27].

Technology neutrality . Services should be independent from used technology in order to allow different platforms to use them [28].

This concept is questionable in connection with automotive, because in a car most of the implemented technology is given by certain standards and cannot be changed easily.

Standardised Service Contract . According to [11], *"Service within the same service inventory are in compliance with the same contract design standards"* [11, p.27]. In other words, the service contract should be created by means of a given templates. The term *service inventory* is referred to in 1.6.2.

1.4.3 Structure of a Service

Concerning the structure, Krafzig describes a service as it can be seen in figure 1.9 with the artifacts *service contract*, *interface*, *implementation*, *business logic* and *data* [5, p.44].

Service contract . "SOA Principles of Service Design" by Erl provides the following definition:

> *"A contract for a service (or a service contract) establishes the terms of engagement, providing technical constraints and requirements as well as any semantic information the service owner wishes to make public"* [26, ch.6.1].

The service contract is, more or less, the core part of every service, for it is the complete specification of the service between a provider and a consumer. It provides all the meta information concerning functionality, capabilities, expected *behaviour*, constraints, service owner, access rights, functional and non functional qualities and information about intended performance and scalability of the service [5, p.44] [30, p.26] [28].

Physically, it is represented by one or more *service description documents*, which should be human- and machine readable at the same time. This is the reason, why web services are usually represented by WSDL documents [11, p.43]. There is no dedicated language standard concerning automotive yet, but a XML based language would be an appropriate choice.

Erl [26] distinguishes between technical and non-technical service description documents. If a *consumer* connects to a *provider*, for example a database, the technical contract could contain information like the database protocol and the query syntax or language, while the non-technical contract could contain related meta information like required safety measures or the physical location of the database [26, ch.6.1].

Interface . The *interface* is described in the *service contract* and specifies the access points and the functionality of the service to the customers, which are connected via a network [5, p.44] [28]. A service may dispose of multiple interfaces.

Implementation . The *implementation* is represented by programs, configuration data and databases, which are necessary to provide the functionality specified in the contract. The term business logic in figure 1.9 is a bit misleading. This should be seen as the algorithms encapsulated in the implementation [5, p.44].

Data . As stated in 1.4.2, a service may dispose of some state data for the time of its application. However, this is no necessary requirement for a service and the amount of stored data should be kept as low as possible [5, p.44].

1.4.4 Services at different layers of implementation

For the assignment of services to different system layers serve the layers defined in section 1.2.4 by [3]: *system-level*, *device-level* and *chip-level*. In connection with segmentation, the terms "level" and "layer" are used equally by various sources. To prevent confusion, the term *layer* is applied throughout this glossary.

Figure 1.10 features an example of which hardware parts may belong to which integration layer with respect to the considered system vehicle. In accordance with that, table 1.1 gives an example of which services may be provided by the particular elements.

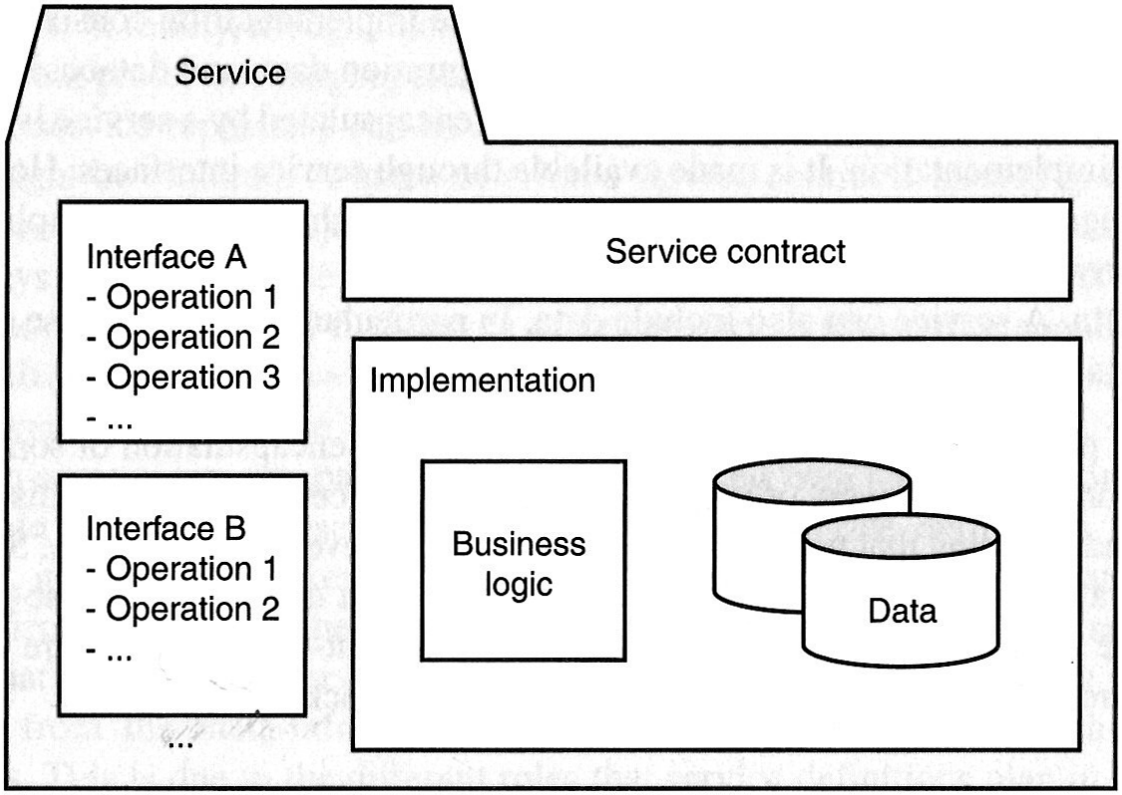


Figure 1.9: Structure of a service with the relations of the particular artifacts [5, p.45].

Chip Layer

The *chip layer* is the lowest layer of implementation and contains very generic services which are used by higher layers in order to create more advanced services. In various sources this layer is referred to as the *core-*, or *platform layer*, which is why the services at this layer are denoted as *core services* [3, p.44].

Typical examples for services located at this level are message based *communication services* for the interaction of *system elements* (cf. section 1.2.2), *global time base services* or mechanisms to compose the overall system out of the independently developed components. Such mechanisms include fault isolation services and clock synchronization services [3, p.7-12].

In other words, it provides a platform, where recurring problems can be dealt with once and for all.

Device Layer

The *device layer* contains more advanced hardware parts. With respect to figure 1.10 these might be sensors, actuators, ADCs (Analog-to-digital converters), AURIX chips, the CAN

bus (Controller area network) or a WDT (Watchdog Timer). Since there is no consistent denomination for the services located at this layer, they are referred to as *device services within this glossary*.

They make use of the underlying *core services* and other services at the device layer in order to provide their intended functionality. An acceleration sensor, for example, could make use of an ADC, which in turn operates with a platform service for the time, for generating periodic sampling points.

System Layer

The highest layer is the *system layer*, containing the most advanced services, which are usually provided to the end user. Like it is the case with the *device services* there exists no uniform denomination throughout literature. Therefore they are denoted *system services*.

System services emerge by binding together services of lower layers. An example for a system service could be a passive safety service for breaking or searing, which bases on an acceleration measurement service and other device services.

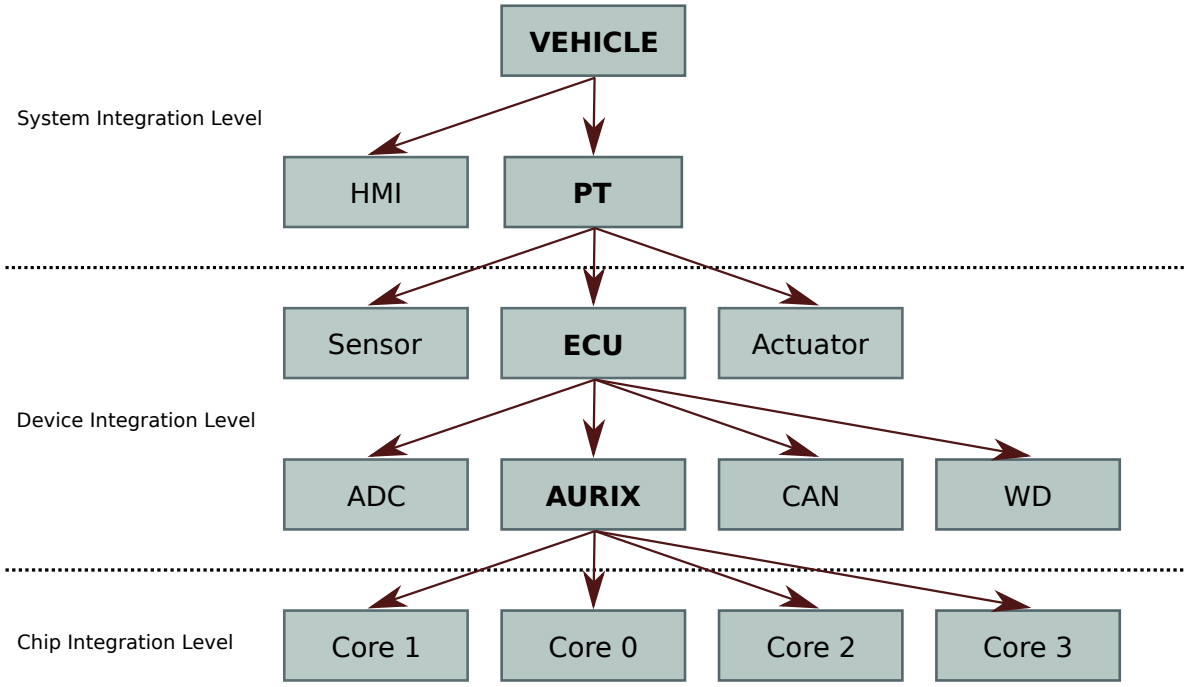


Figure 1.10: Examples of hardware parts at different levels [6]

Table 1.1: Example of which services could operate at different levels

Integration Level	ID	Description
System Level Services	Powertrain	Enable the vehicle to drive.
	Vehicle	Passive safety service.
	HMI	Services for user input and output.
Device Level Services	Sensor	Acceleration measurement service.
	Actuator	Providing a certain degree of acceleration, when triggered.
	ECU[1]	Control of an electronic subsystem.
	ADC[2]	Transform an analog signal to a discrete-time, digital signal.
	CAN[3]	Enable communication between devices.
	WDT[4]	Detect and recover from malfunction.
Chip Level Service	Chip 0-2	Basic time services, component execution services, basic communication services.

[1] According to AUTOSAR an ECU is a *"microcontroller plus peripherals and the according software/configuration. Therefore, each microcontroller requires its own ECU Configuration"*.

[2] An *ADC* (Analog-to-digital converter) is a device for transforming a continuous analog signal into a time sampled discrete digital signal. - `http://en.wikipedia.org/wiki/Analog-to-digital_converter`

[3] The *CAN* (controller area network) bus is a vehicle bus standard that allows devices to communicate with each others by means of a message-based protocol. - `http://en.wikipedia.org/wiki/CAN_bus`

[4] The *Watch Dog Timer* is an electronic timer for detecting computer malfunctions and recovering. It is usually applied for embedded systems, where humans cannot easily configure the internal logic. - `http:en/wikipedia.org/wiki/Watchdog_timer`

1.5 Architecture

The term *architecture* is very generic and related to various different domains, e.g. *hardware architecture*, *software architecture*, *system architecture* or *enterprise architecture*.

In general, independent from any considered domain, architecture is concerned with how the components of a system can be arranged and interrelated in order to assemble an overall system [7] [25].

1.5.1 Definitions

The following definitions of architecture, are domain independent or can be referred to multiple domains.

ISO/IEEE 42010 . The ISO/IEC/IEEE standard defines requirements for the description of system, software and enterprise architectures. Within the standard, the term architecture is referred to by *"fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution"* [7]. What is "fundamental" is described in more detail as:

elements . (cf. section 1.2.2)

relationships . This denotes the relationships inside the system as well as the relationships to its environment

principles of design and evolution .

This definition applies for any kind of architecture, but the emphases on the fundamental concepts (elements, relationships, principles of design and evolution) vary with respect to the considered domain. The *software architecture* usually focuses very much on the elements, while the *enterprise architecture* is more concerned with the principles [7].

Rodrigues, Melo Pires et al. . Rodrigues, Melo Pires et al. feature the following description in their document "Using SOA in Critical-Embedded Systems" from 2011: *"Architecture is a structure that identifies, defines and organizes components. The relationship and principles of design of the components, functions and the interface established between subsystems can also be defined by architecture"* [17].

AUTOSAR .

> *"The fundamental organization of a system embodied in its components, their static and dynamic relationships to each other, and to the environment, and the principles guiding its design and evolution"* [16].

Although EMC2 project is mostly concerned with system- and software architecture, the definition presented below is compatible with all kinds of architecture design, for they share the same basic principles.

It also supports the concept that a system consists of one or more components, as pictured in figure 1.1.

An architecture is a systematic description of the structure of a system by means of its involved components and their relations to each other, and specifies also the connections and interactions of a system to its environment.

At the same time, architecture is also responsible for determining the principles of design and evolution.

1.5.2 Placement of Architecture

The placement of *architecture* in relation to other entities like system or environment (cf. chapter 1.2) is illustrated in figure 1.11.

Architecture description . Many sources mix up the definitions of *architecture* and *architecture description*, what is a recurring source of confusion. In contrast to the actual architecture of a system, the term *architecture description* denotes the artifacts which document this architecture [7]. Their relation is also pictured in figure 1.11.

According to the ISO/IEC/IEEE 42010 standard, the *architecture description* is used to express the architecture of a system of interest by means of the following elements:

- Specification of the purposes of the system,
- Suitability for achieving these purposes,
- Feasibility of construction and applicability,
- Maintainability,

- Evolvability and
- Association of these concerns with the *stakeholders* having these concerns [7].

One and the same architecture can be described by several different architecture descriptions, and at the same time an architecture description can also characterise multiple architectures [7].

Stakeholder . The *stakeholders* are all people, which are somehow related to the system and have any interest in the system. Those include users, operators, owners, developers, maintainers and others [7].

System concern . A specific *system concern* can be held by one more more stakeholders and can appear in various different forms, e.g. expectations, responsibilities, requirements, assumptions, dependencies and more [7].

Purpose . *Purposes* are one kind of *system concerns*, which are issued by the stakeholders [7].

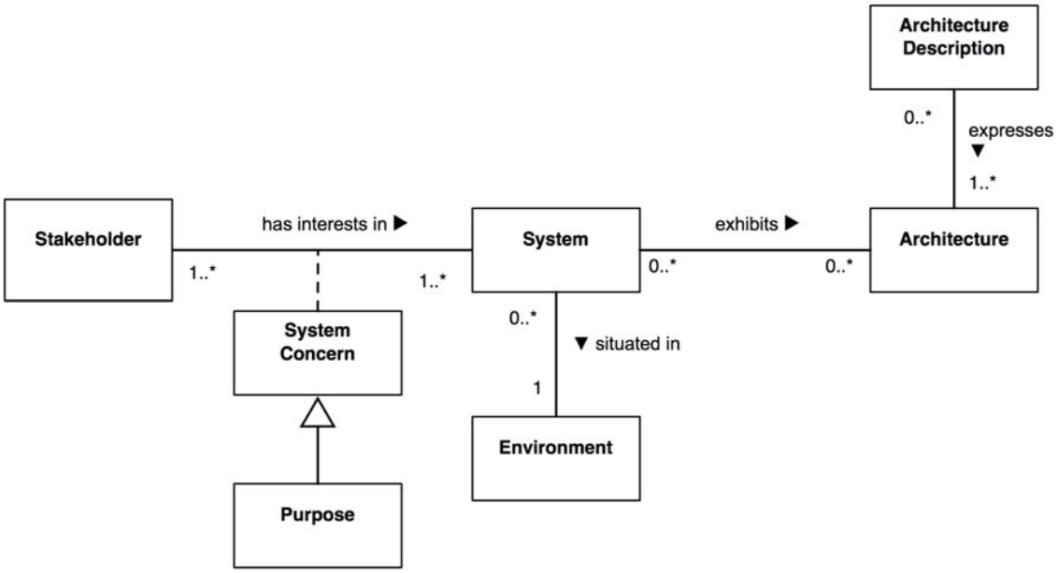


Figure 1.11: Relations of architecture to other entities [7]

1.6 Service oriented architecture

The term *service oriented architecture* has been widely used for marketing and praising new products. Unfortunately, this resulted in various misinterpretations of the term, because some

vendors suggested that a SoA is something that can be bought or installed on an existing system, missing the point that SoA is not a product, but a set of design paradigms that can be applied on architectures.

The driving factor for the application of this design paradigm are certain benefits that come with the modularization of software. Those are not only reduction of development costs, but, with respect to embedded systems, also a saving in hardware components [32]. Furthermore it increases flexibility, scalability and fault tolerance, enabling the system to run, even if parts are erroneous or down [33, p.33] [30, p.17-18].

The following definitions reveal the wide range of different views and understandings.

Oasis . OASIS is a nonprofit consortium that drives the development, convergence and adoption of open standards for the global information society.

> *"A paradigm for organizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations"* [34].

Papazoglou .

> *"Service-oriented architectures (SOA) is an emerging approach that addresses the requirements of loosely coupled, standards-based, and protocol independent distributed computing"* [35].

Donini, Marrone et al. .

> *"SOA is an architectural style for building software applications that use services available in a network such as the web and it represents the widest accepted model to design geographical distributed systems."* - [36]

Arcitura .

> *"Service-oriented architecture is a technology architectural model for service-oriented solutions with distinct characteristics in support of realizing service-orientation and the strategic goals associated with service-oriented computing"* [23].

The EMC2 project aims at implementing the SoA design paradigm into safety critical embedded systems, which comes with various additional challenges. Thus, the definition below was created with regard to embedded systems. However, it is quite generic and fits also for SoAs in web applications.

The service oriented architecture is no actual design, which can be simply implemented or installed, but a collection of design principles for distributed systems.

Originating from the object oriented- and component based engineering, it pushes the concept of software reuse to a new level by using technology independent and loosely coupled services.

Accordingly, a SoA disposes of an arbitrary number of services, which are interconnected by means of an underlying network and predefined protocols.

As stated, SoA is no specific implementation but a set of rules to achieve a certain target state. The final implementation can therefore consist of multiple technologies, products, Application Programmers Interfaces (APIs) and supporting infrastructures [11, p.29]. This enables an unproblematical exchange of components by components of other vendors, as long as the provided services remain unaffected.

A SoA is not just a simple collection of services, but offers also *composition mechanisms* (cf. section 1.6.3) for services, enabling the creation of agile, higher-level services with a more sophisticated functionality, without having to build everything from scratch. [30, p.12].

1.6.1 Historic Development

The service-oriented architecture (the term first appeared in 1996 [33, p.7]) emerged from the CBSE and was, so to say, an improvement, for it allowed the components to be wider distributed and looser coupled. However, since both approaches feature certain advantages, both of them have been developed in parallel ever since, resulting in many similarities, but also many differences [28]. The advance of the software reuse concepts from 1960 to nowadays is illustrated in figure 1.12.

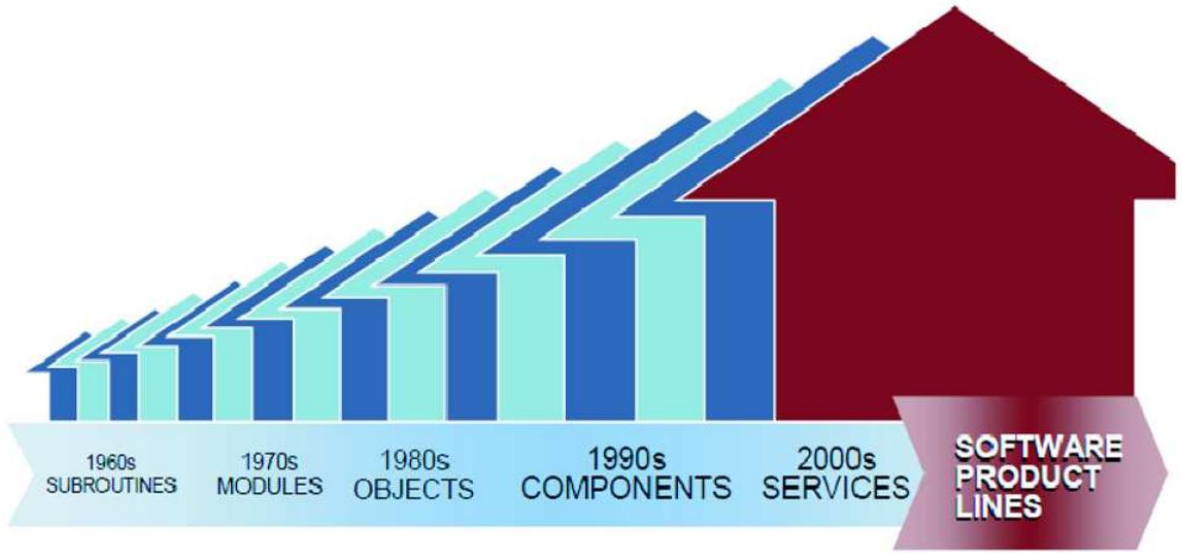


Figure 1.12: Development of software reuse from the 1960 to present [8].

1.6.2 Structure of a SoA

The four main artefacts of a SoA are *service provider*, *service consumer*, *service repository* and the *service contract* [9] [28] [17]. There are also alternative terms for those, which are mentioned here, but throughout this document and the related documents, the previously stated terms will be used.

The relations of the terms is illustrated in figure 1.13.

Service provider/ service owner . The task of the *service provider* is to provide a service and its functionality. Additionally he should formulate the service description in form of a contract (cf. section 1.4.3), which is then handed over to the *service repository* in order to make the service discoverable [28].

Service Repository/ Service Registry . In some sources this entity is referred to as *service registry* or *service directory*.

The *service repository* is, more or less, a database of services, which may be distributed. It disposes of certain publishing mechanisms in order to make services discoverable by the *service consumers*. Therefore it contains all the information from the *service contract* (of every version of the service), as well as additional information like physical location, provider information, technical constraints, security issues and of course the link to the registered service. [5, p.60-61] [28] [10].

Services can be added to, or taken from, the *service repository* dynamically. Thus,

services need to be already running and ready to use in order to be discovered and composed at runtime.

Service Consumer/ Service Client/ Service Requester . The *service consumer* (or *service client*, *service requester*) is the instance that calls the service and can be either an end-user or another service. He sends a request for searching the *service repository* for specific services by the service interface description. Subsequently, the repository returns a list with suitable services. If an appropriate service is identified the *service consumer* creates a dynamic binding with the *service provider* in order to invoke the service and interact with it [28] [10]. This procedure is depicted in figure 1.14.

Service Contract . The *service contract* is described in detail in section 1.4.3. It is handed over to the *service repository* from the *service provider*.

Service Inventory . Erl, Bennett et al. mention this additional term in connection with SoA. According to their definition, *"A service inventory is an independently standardized and governed collection of complementary services within a boundary that represents an enterprise or a meaningful segment of an enterprise"* [11, p.41].

The term enterprise in this definition appears a bit bizarre and is a result of the research area related with this source. For the scope of EMC2 project the boundary is defined as the vehicle. Thus, the service inventory includes all services which could be provided by the vehicle. The difference to the service repository is that the service do not need to be available, or even implemented, but it is a static list of possible services.

1.6.3 Service Composition

Service composition is the key concept of any SoA and denotes the generation of (more) capable services from existing services. In [11] it is stated that *"at least two participating services plus one composition initiator need to be present"* [11, p.40]. In other words, the resulting service of a service composition requires at least two underlying services. Otherwise it would be just a point-to-point exchange [11, p.40].

There are two different methods for composing services, which are denoted *service orchestration* and *service choreography*.

Service Orchestration . With this method of aggregating services, the new service has control over the other services [30].

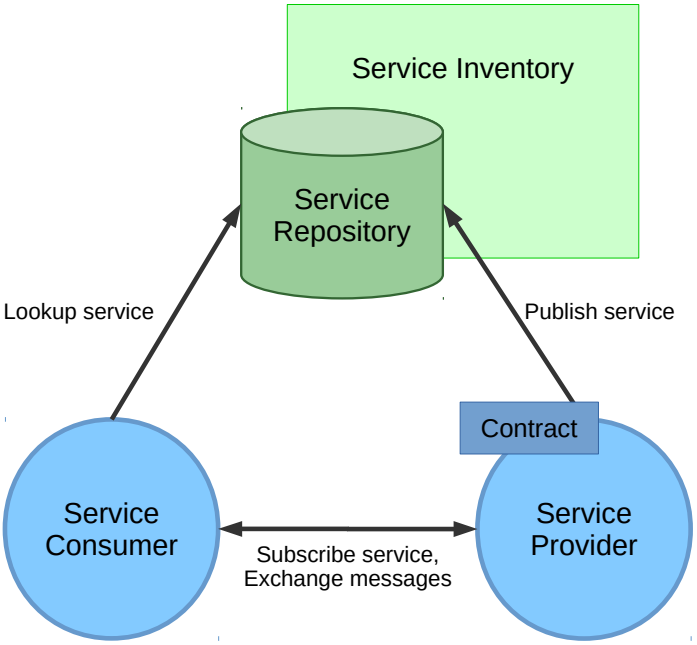


Figure 1.13: Relation of *service provider*, *service consumer* and *service repository* [9] [10]

Service Choreography . In contrast to service orchestration, non of the related services has control over the others. Instead, there are certain rules and policies, which regulate the collaboration [30].

1.7 Dependability

The terms *Availability* and *Reliability*, which can be found at any level of implementation, are closely tied and usually come together. Often, they are also coupled with the term *Maintenance*, which also plays an important role in the design of any system [3, p.116] [37].

1.7.1 Reliability

Reliability denotes the time an entity need to fail while its operating. In other words it is "the probability of the failure-free operation of a system for a specific period of time in a specific environment" [3, p.116]. Nelson describes this also in a very comprehensible way as "Reliability, $R(t)$, is the conditional probability that a system can perform its designed function at time t, given that it was operable at time $t = 0$" [38].

When it comes to services, *Reliability* can also include the ability to recover state information after an interruption and continue the service as it has never been interrupted [3].

Obviously the reliability of higher level entities is based on the reliability of its sub entities.

1.7.2 Availability

The definition of *availability* by various sources slightly differs.

The ISO 26262 standard defines the availability as the *"capability of a product to be in a state to execute the function required under given conditions, at a certain time or a given period, supposing the required external resources are available"* [14]. Obermaisser and Kopetz, however, describe it as the *"probability of a software service or system being available when needed"* [3, p.116], which is very similar to the definition found in [37] and [38].

The difference between the definitions is obviously that the first one denotes *Availability* as the availability of an entity at this very moment, while the other one defines it as a probability of an entity to be operational and ready to use.

For the purpose of this project the term should be referred to as the *probability* of a system of being operational at a given time.

Two important terms which come in connection with availability are *MTTF* (Mean Time To Failure), the expected time until the system fails, and *MTTR* (Mean Time To Repair), the necessary time to restore a failed system to normal operation. The availability A can then be expressed as $A = MTTF/(MTTF + MTTR)$ [38].

Thus, the availability characteristics of a system are represented by its reliability and maintainability. Accordingly, even highly reliable entities can have a poor availability if the repair time of its sub entities takes very long [37]. The availability can be improved by the application of dedicated *fault tolerance* mechanisms (cf. section 1.8.4) [38].

1.8 Functional safety

1.8.1 Definition of safety

Zeilinger, Sevcik et al. (2009) .

> *"The term safety describes the ability of a system not to cause environmentally harming effects, due to loss of mission critical information"* [39, p.1].

ISO 26262 .

> *"absence of unreasonable risk"* [14].

AUTOSAR . AUTOSAR has adopted the definition of the ISO standard: *"Absence of unreasonable risk"* [16].

1.8.2 Definition of functional safety

Zhang, Lia and Qin (2010) . *"Functional Safety processes are necessary to fulfill certain safety requirements (non functional requirements) in addition to functional requirements."* [40].

This definition is not entirely wrong, but its important to note that in automotive *normal functions* cannot be separated from *safety function* and accordingly their respective requirements are also interrelated [1].

ISO 26262 . *Functional safety* means the *"absence of unreasonable risk due to hazards caused by malfunctioning behaviour of E/E systems"* [14]. An unreasonable *risk* is thereby a risk judged to be *"unacceptable according to valid societal moral concepts"* [14].

Risk, in turn, is defined as the product of the *probability of occurrence* and the *severity of harm* [14].

The definition by the ISO standard seems quite promising and should be used as foundation for the scope of this project.

As mentioned in the introduction, the term *functional safety* only addresses the absence of risk due to equipment malfunction. It does not care about possible risks due to malicious events caused by other people, or just inappropriate operation of the user.

1.8.3 Safety related terminology

Fault, *error* and *failure* are terms with are often mixed up and used in the same context, since they seem to be very similar in their meaning. Nevertheless, they stand for different things and its important to be aware of their disambiguation.

Fault . In ISO 26262 standard a *fault* is listed as an *"abnormal condition that can cause an element or an item to fail"* [14] [16].

Faults come in different forms. An *intermittent faults* occurs from time to time and disappears without any repair measures taken place. This is typical for hardware components which are worn out and almost breaking down. *Transient faults*, on the other

hand, appear once, and do not recur afterwards. This is for example the case if there is some electromagnetic interference. *Permanent faults* naturally remain until the faulty component is exchanged or repaired [14].

A term related to *fault*, is *fault coverage*. It denotes "the number of the faults detected as a percentage of the total number of faults affecting the system" [12].

Error . An *error* is the deviation of a computed, observed or measured value or condition and the expected, theoretically correct value. An error occurs as consequence of unexpected operating conditions or a fault. Nevertheless, a fault does not necessarily lead to an error [14] [38] [16].

If the *error* exceeds a certain threshold it can cause a *failure* [16].

There are two different classifications of errors. They can be divided into soft- and hard errors or data- and control flow errors [41] [12].

Soft Error . *Soft errors* occur when a temporary extra charge is induced into the electric circuit, usually due to cosmic radiation. This causes a flip of the data state in a memory element [41] [12].

This kind of error will become even more severe in the future, due to the decreasing size of electronic logic, as well as their increasing complexity [12].

Hard Error . In contrast to *soft errors*, *hard errors* are caused by a constant property change of an electric component. This could happen either because of a change in the input temperature, or input voltage [41].

Data Error . This error means a change of the data stored any memory location or register [12].

Control Flow Error . A *control flow error* leads to wrong execution sequence of instructions. This is the case, when the memory storing the address of the next execution instruction is changed [12].

Failure . *Failure* denotes that an element is not able to provide it's functionality any longer. This happens usually due to errors in the element or the environment, which are in turn caused by various faults [14] [38] [16].

The averaged frequency of occurrence of a failure is denoted by *failure rate*, which is counted in hours of operation until a serious fault - e.g. 10e5 to 10e9 hours of operation

[17].

Another term which frequently comes up in connection with failure, is *failure mode*. This denotes, depending on the consulted source, either the manner in which the component fails [42], or the manner by which a occurred fault is observed [43].

1.8.4 Fault tolerance

The key concept of a *fault tolerant* design is to enable a system to provide its intended functionality in the presence of a given number of faults [38].

The *fault tolerance* mechanisms are not responsible for preventing the occurrence of faults, but assure that a fault does not lead directly to the violation of the safety goal(s), which maintain the system in a safe state [14].

Electric system suffer from numerous different possible faults, which increase with the complexity of the system. They include not only internal faults like design-faults or hardware wear-outs, but also external ones - for example misuse by the user or cosmic radiation.

Due to the billions of transistors present in advanced electric systems and the various circumstances that lead to faults, faultless systems remain an utopian dream [3].

Thus, the *fault tolerance* mechanisms are not actually concerned with the prevention of faults, but assure that a fault does not lead directly to the violation of the safety goal(s), which maintain the system in a safe state [14].

1.9 Design of fault tolerant Systems

The prerequisite for designing a fault tolerant system is a so called *fault hypothesis*, which is an assumption regarding the type and frequency of faults the system is supposed to handle.

These can include

- *Software error* - Especially for complex software it can be assumed that there are a certain amount of bugs and imprecisions.
- *Delay and disruption error* - This is relevant for all networking systems.
- *Transient faults* - This means a possible corruption of Flip-Flops or memory cells with the progression of time.

A way to deal with all these scenarios *ECRs* (Error Containment Regions) are required. An error which occurs within this specific subsystem will not propagate outside the subsystem without being detected.

Another possibility is redundancy, where multiple subsystems are responsible for the same task and are able to detect faults by comparing their results and validating them by means of dedicated voting mechanisms [3].

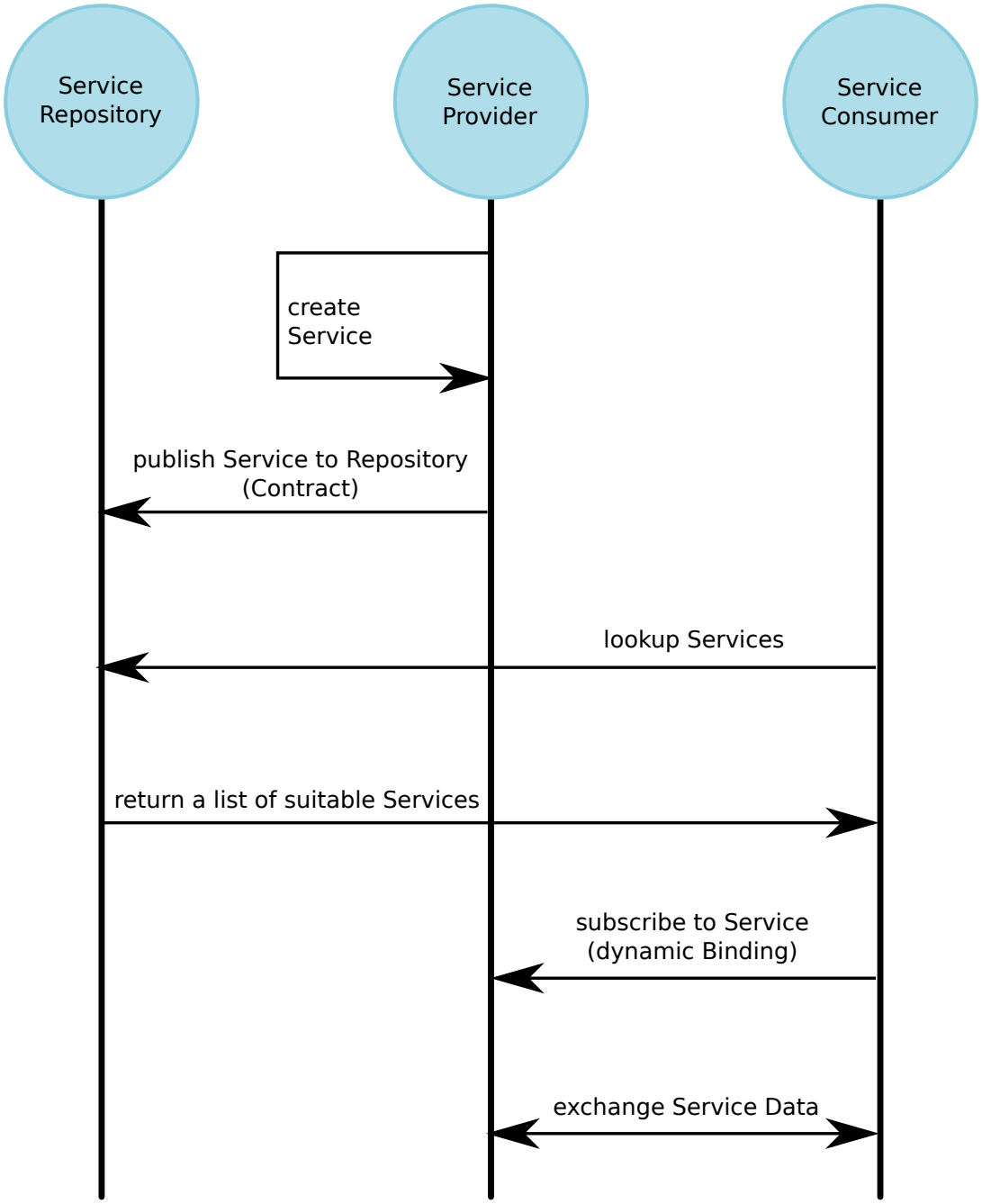


Figure 1.14: Chronology of how a service is used in a SoA [10].

Chapter 2

Methods

This chapter is structured in the four main section, entitled *SoA in embedded systems (Embedded SoA)*, *SoA in automotive*, *safety services* and *service development process*.

The first section deals extensively with the possible application of the service oriented design paradigm in embedded system and deals with the challenges this approach has to face when approached with safety-critical requirements. Moreover, the differences to conventional SoAs, as Web application is investigated in detail. The second section presents various examples of services, which are relevant for *functional safety* in detail and presents their intended functionality, as well as possible implementations in terms of an architecture.

The final part consists of a simplified use case, dealing with the design of an error detection service with respect to the design phases *service investigation/ planning*, *service inventory analysis*, *service oriented analysis* and *service oriented design*.

Most of the findings and concepts in this chapter are (not yet) covered in literature, but is the result of numerous meetings and discussions at VIRTUAL VEHICLE during May to July 2015.

2.1 SoA in Embedded Systems (Embedded SoA)

The service oriented design paradigm was originally designed for the application in the web, which offered ideal prerequisites for this kind of architectural style: An underlying network for interconnection is already present and *time constraints* are no concern, for delays are unlikely to cause a disaster. Thus, it is no surprise that web services are the application area, where SoA has scored the highest market penetration [17] [44].

2.1.1 Drawbacks in embedded systems

In contrast to web services, *embedded systems* consist of numerous interconnected nodes, with diverse measurement-, steering-, or computation capabilities. They have to face additional challenges like *limited resources*, *different complexity of hardware*, *time constraints*, etc. [29] [32].

The most crucial of those obstacles are investigated in more detail in the following.

Limited resources . One obvious major drawback of embedded systems are the quite limited resources, which are designed for highly specialized purposes and lack computation power as well as storage size [17] [29] [32].

Different levels of complexity . The complexity of hardware varies greatly. The applied components may include very primitive sensors, with few capabilities and very advanced nodes like MPSoCs, at the same time [29] [32].

In other words, there are high level *information systems services*, as well as low level generic *embedded system services*. It is the task of SoA to deal with the connection and integration of them [17]. This task gets aggrevated if SoAs in embedded systems are connected to high level SoAs like web applications.

Event- and data-driven . In contrast to web services, an embedded system disposes of a network with (many) sensors. Thus, the ad-hoc *request-response* message pattern, which is common for web services, cannot be simple addopted for the event- and data driven embedded systems [32]

Instead, the communication in those works mainly by the *fire and forget* scheme: A sensor measures some data and publishes it to all connected services, which have to decide on themselves, whether the received data is relevant, and how to process the received information [32].

Lifespan of services . Another difference to web services is the lifespan of services. While web services are used to work only a limited number of hours (or even minutes), the services in embedded systems could have application times of multiple years, or even lasting for the lifespan of the system [44].

Dynamic character . The components of a SoA show dynamic characteristics: *"new nodes may enter the network, existing nodes may fail and network characteristics can change*

over time, especially if wireless communication media are used" [32]. This can become an issue, for conventional embedded systems dispose of a predefined set of components, which are determined when the system is assembled.

Time constraints . Embedded systems are *time-critical*, meaning that computation must be conducted within a given time window in order to allow the correct operation of the system. Especially, in a safety-critical system like a vehicle, which is used to operate at high velocities, a violation of those time constraints could cause serious incidents.

These obstacles are the reason why web services and safety-critical embedded systems are often considered as non-related areas [17]. Nevertheless, the application of the service oriented architecture style in embedded systems offers numerous benefits:

- Decoupling configuration from environment,
- Improvement of reusability,
- Improvement of maintainability,
- Higher level of abstraction,
- Enhanced interoperability and
- More interactive interfaces between devices and information system [44].

To sum up, the SoA paradigm offers a promising approach to overcome many of embedded systems related drawbacks [44] [32]. Accordingly, there have been various projects dedicated to investigation of the applicability of SoA for embedded systems. Those include SIRENA, SOCRATES, OASiS, MORE, RUNES and εSOA.

2.1.2 Embedded SoA

There is no unified denomination for SoAs in embedded systems throughout literature. Thus, it is referred to by the term *Embedded SoA (abbreviated as ESoA)* within this document.

The following definition is the result of our findings:

The Embedded SoA works at a lower, very hardware oriented level, with a predefined and unchanging set of component.

The conventional SoA is located above the embedded SoA and connected through various interfaces.

With respect to the example of a vehicle, the ESoA operates on elements like a power-train, various devices (sensors, actuators, controllers, etc.), while the SoA level contains the vehicle as overall system, but also other vehicles, which can be referred to as the *System of Systems*.

Regarding the ISO 26262 standard, the ESoA can be related to **Part 3: Concept phase**, and the SoA to **Part 4: Product development at the system level**.

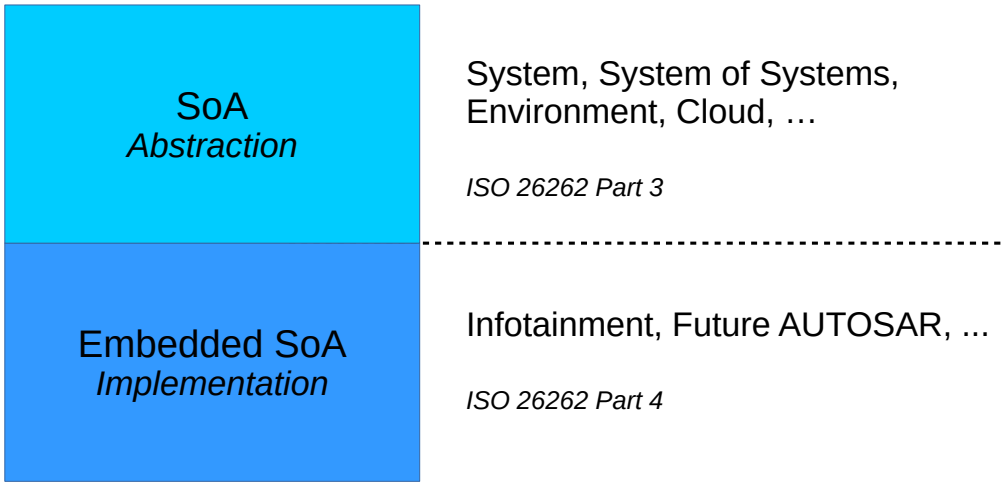


Figure 2.1: Relation of service oriented architecture to service oriented architecture in embedded systems.

2.2 SoA in automotive

This chapter is dedicated to the investigation of the applicability of the SoA concepts for automotive.

In the future of automotive all vehicles will, most likely, be interconnected with each other and also with the environment. This could give them opportunities like automatically detecting whether the traffic lights are green at a crossing and driving autonomous. Within this document those futuristic vehicles are referred to as *connected cars*.

The opportunities and advantages of the SoA approach are described extensively in 2.1. In terms of vehicles, the major advantage would be the possibility to reduce the quantity of computation hardware. At present time, each component disposes of his own dedicated hardware for conducting computations, which is unused for the majority of time. Nevertheless, it comes with a lot of extra weight. As stated in [18, p.7], *"Embedded systems should exploit the available hardware architecture as much as possible.s"*

The SoA approach might reduce not only the weight, but also the complexity and costs of the overall vehicle. So far, it is frequently the case that each vendor uses his own proprietary network and additional hardware, prohibiting the application of hardware components from another vendors [32].

Despite of all these advantages, the SoA paradigm is hardly applicable for vehicles right now. On the one hand, this is due to the strict regulations in connection with safety critical real-time systems like vehicles, where an error or a malfunction can easily harm people [45]. There are not even regulations, addressing SoA in automotive, yet.

On the other hand, there are technical constraints. The AUTOSAR architecture, which is widely applied today, is not constructed for dynamic reconfiguration or binding of services, but everything has to be specified at building time. Still, there is already an approach by AUTOSAR, denoted "Future AUTOSAR", which is dealing with this very issue.

Although, the SoA for vehicles is still a long way ahead, it might be easier applicable at a higher level, like at the *system of systems* level (other vehicles and the overall traffic). For example services for the communication with traffic lights or other participants of the traffic. If the the traffic lights in Germany would use another service as the ones in Austria, a SoA could enable the dynamical binding of the new service, when the border is approached. There are already dedicated projects which are dealing with the communication between vehicles.

2.2.1 Location of the service repository

The terms service repository is already covered in detail in the related glossary. Nevertheless, questions arose concerning its location and actual implementation with respect to automotive. This section covers the findings concerning this matter.

For SoAs in automotive it is likely that there will emerge various distributed repositories. One repository inside of every vehicle, containing all the services provided by the vehicle itself, which is also available if the vehicle is operating in urban regions and is not able to connect to

its environment. For the SoA inside the vehicle, there is no other possible location, because otherwise it would be operable only as long as it is connected to the Internet, or whatever platform, which is used for the interconnection with the environment.

For the scope of this document these repositories will be referred to by the term *local service repository*. Examples for the services they hold are services which origin from hardware components of the vehicle like, acceleration measurement, temperature measurement, communication services and the like. But also safety critical services for fault- and error detection, which need to be available whenever the vehicle is operated, are located within this repository.

The counterpart to the *local service repository* is denoted *external service repository*. This could also be physically distributed and holds mostly services which are needed for interacting with the environment. For "self-driving-" or "connected cars" it could host all the services for managing the traffic. For example: A traffic light in Graz publishes its service, "Show me the traffic light signal", somewhere on a server, where it is detected and bound by a vehicle driving in Graz. It is then able to decide automatically whether it has to stop at an intersection.

Other services which could be located in this repository are *update services*. If an update for an existing service in the local repository is available, the vehicle could detect this automatically and subsequently load and install the service in question.

2.2.2 Service Contract

The service contract is the complete and extensive description of the service and should contain (with regard to vehicles) documents from AUTOSAR, the ISO 26262 standard and documentation regarding *functional safety*. It is one of the goals of **Work Package 1** of the EMC2 project to extend the *service contract* by a functional safety part.

At the beginning of a service development process, the contract exists just as an empty template, which gets filled more and more as the development advances. A template, which could be used for the development of services in the automotive industry is already provided by the ARROWHEAD project.

2.3 Safety services

If safety-critical systems, like vehicles, get connected to the outside world, security becomes an important issue and most of the provided services must be fault tolerant and secure at the

same time.

There are many challenges for safety and security management in SoA systems, like the distributed hardware and software structure, loosely coupled components from different vendors, etc.

Functional safety is directly related to *availability* and thus the overall safety can be increased by increasing the availability [46]. According to [46], a system has to generate the correct output in order to be available. Services for failure detection, error detection and error masking can increase the availability and also enable recovery from errors.

In the following sections, the interference and connection of safety and security services is analysed, before the principles of some of a few selected safety services is investigated in more detail.

2.3.1 Relation of Safety and security services

As described in the glossary, in vehicles it is not possible to distinguish between safety functions and "normal" functions, because vehicles operate at high velocities and a failure of any functionality could lead to a disaster [47]. Therefore, all services are considered safety-critical.

Since future vehicles are planed to be always connected to the outside world, *security* must also be taken into account, because a malicious attack on the system could be equally disastrous. Accordingly, all services must also take a security aspect in consideration. Comparable to functional safety and fault tolerance, security will be taken care of by dedicated services.

However, some services cannot be assigned to any of the disciplines, but have overlapping responsibilities, as depicted in figure 2.2. According to our findings this effects especially the services for *authentication*, *memory protection* and *failure detection*.s

2.3.2 Failure detection service

A Fault Detection Service (FDS) is a service which is capable of detecting faults, and eventually, depending on its implementation, also *control flow errors*. Control flow errors are errors, which lead to a wrong execution sequence of the instructions of a service. Technically, the FDS can be implemented as simple timer circuit with a specified threshold time. If this limit is reached, it changes its state for triggering further actions, like restarting a component or activating another safety service [7]. The advantages of the FDS are the simple design, which

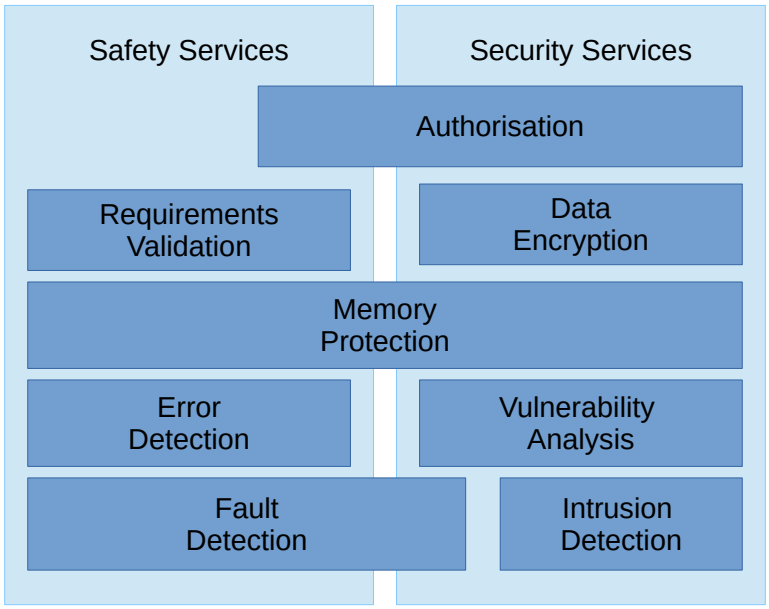


Figure 2.2: Classification of various services into safety and security services.

reduces the additional complexity of the overall system, as well as the costs. Concerning the functional principle, there different designs with increasing complexity, which can provide, for example, a certain time window for the response [7].

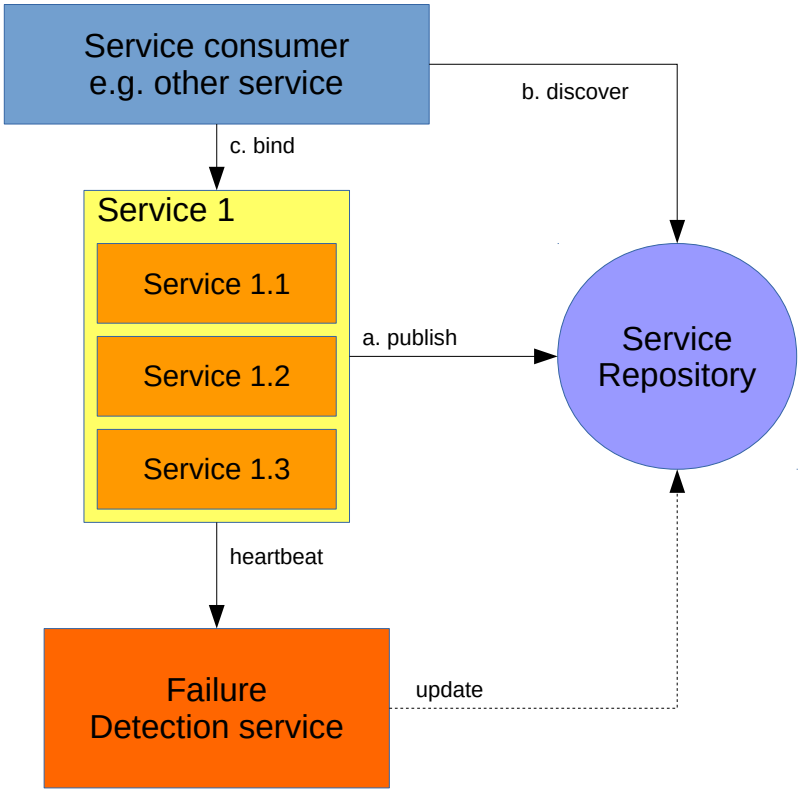


Figure 2.3: Example architecture for an *fault detection service*, like a WDT.

In the following, a fault detection service is described by means of its most prominent

representative, the so called *Watchdog Timer*.

As suggested by the name, the WDT is based on a timer, which gets reset every time, a heartbeat signal from the observed service is received. There three different basic designs: the *Standard Watchdog Timer*, the *Windowed Watchdog Timer* and the *Sequenced Watchdog Timer* [12].

Standard Watchdog Timer . With this basic setup, the service mirrored by the WDT periodically sends a simple heartbeat signal, indicating that the service is alive and active. This signal resets the timer. It a predefined amount of time elapsed, without an incoming signal, it is assumed that a fault has occurred and the WDT changes its state [12].

In terms of *control flow errors* in a service, the heartbeat signal may, or many not, be sent too late. In the latter case, the WDT is capable of detecting the error [12]. The probability of noticing such an error is higher, the closer the threshold time is to the time between the heartbeat signals.

Windowed Watchdog Timer . The *Windowed Watchdog Timer* is an improvement of the standard version, which is capable of detecting most of the *control flow errors*. This is enabled by the application of two timers instead of one. With two timers the WDT is able to specify a time window, during which the heartbeat signal from the observed service must be received. The WDT is triggered if it receives a signal outside the window, or the timer reaches its threshold [12]. This is illustrated in figure 2.4 with the $time$ on the x-axis and the Timers labeled $T1$ and $T2$.

In case of an *control flow error* this signal is a bit ahead or past in time. The error detection coverage increases with narrowing the time window [12].

The advantage of this design is clearly that it allows the detection of more errors, but on the other hand the implementation is slightly more complex.

Sequenced Watchdog Timer . This design is a further improvement of the *Windowed Watchdog Timer* and bases on the same principle. In contrast, to the other designs, the signal sent from the mirrored service carries a sequenced parameter. Only if the signal arrives in time and within the specified time window, this parameter is evaluated and compared to a parameter inside the WDT. If those match the sequence variable in

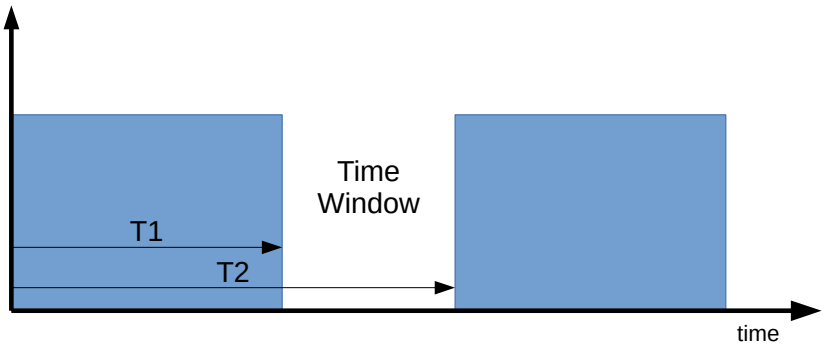


Figure 2.4: Schematic design of a windowed watchdog timer [12]. The time windows is the result of $T2 - T1$.

the WDT is incremented and the timer reseted, starting the cycle over again [12]. The whole proce ss is illustrated in figure 2.5.

The disadvantage of this design is the clearly higher complexity, for the *Sequenced Watchdog Timer* must be capable of holding and modifying state information as well as comparing it to received information.

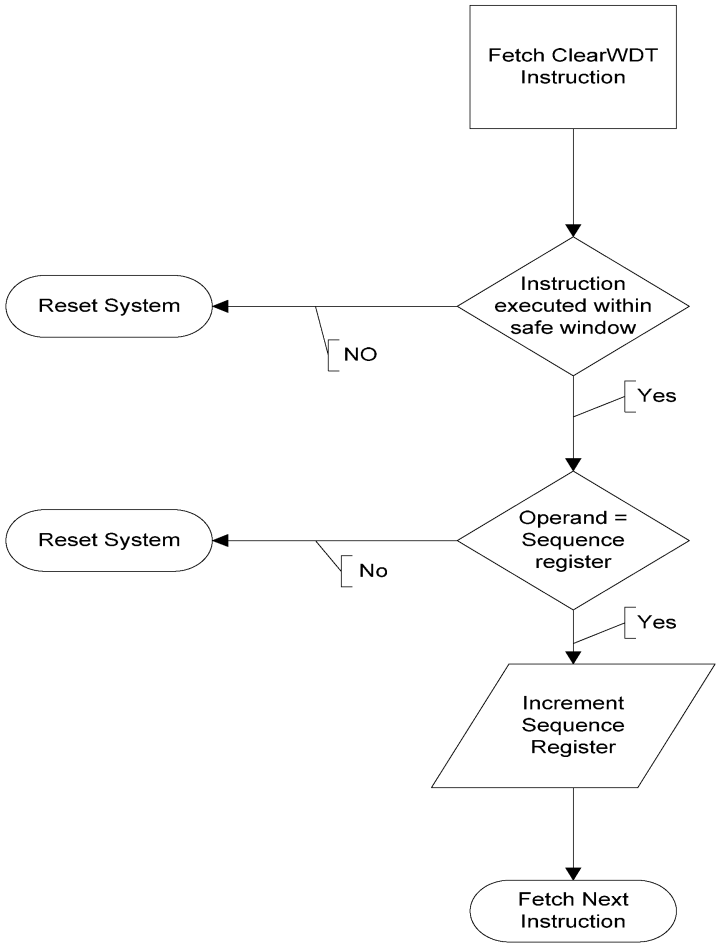


Figure 2.5: Schematic illustration of the working process of a *Sequenced Watchdog Timer* [12].

2.3.3 Error Detection/Masking Service

Most of the errors, especially those which do not alter the timing, remain unnoticed by the WDT. There are several approaches in the design of an *error detection service*, but the most simple and approved one, is based on *triple modular redundancy* (TMR). With this approach an *error detection service* mirrors three individual and independent services, which provide the same functionality. With respect to automotive, this could be, for example, services for acceleration measurement. A comparing logic inside the *error detection service* compares the information received from these services and can identify an error in one of the services by means of a simple voting mechanism [48].

In advance, another service can be triggered for restarting the service in question, or other actions.

The very same implementation is also capable of performing *error masking*, because due to the redundancy and the voting system the service could also hide an error in one of the mirrored elements, without the service consumer noticing.

2.3.4 Memory Protection

Memory protection is related to *safety* as well as *security*. It is a method for preventing processes or users from accessing memory that is not allocated to them.

Former embedded systems, or such that are small in size, do not necessarily require memory protection mechanisms, because all related programs have a very specific purpose and an unintended behaviour is rather unlikely. In such cases the overhead in runtime, when using memory protection just does no pay off [49].

However, modern, large scale system, have numerous third-party components are used to interact with the user and the environment. At the same time, the overhead in computing is no longer crucial, due to the increase of computing power [49]. Especially, if the system is connected to a public network like the Internet, memory protection becomes also a big issue for the prevention of malicious attacks.

There are several aspects that stress the need for memory protection in embedded systems:

- It can serve as fault and error detection and containment mechanism, preventing a failure of one service to propagate and infect the whole system [49].
- It protects the system from unintended behaviour of the particular services [50].

- It is an important aid in the development process and helps at debugging by identifying "illegal" behaviour of erroneous services, resulting in a reduced development time [49] [50].
- It is also crucial from a security point of view, because it prohibits unauthorised access.

As a service, it could be implemented like the *Information Assurance* core service from the ARROWHEAD framework, with dedicated services for authentication, granting privileges and managing authorization information.

2.3.5 Requirements Validation Service

This service should be responsible for checking the compliance of safety margins, when services are composed. For example, if a service is required to be in compliance with ASIL D, it cannot be based on another service, which can only provide ASIL B. According to its specification the requirements validation service could either prevent this orchestration, or even look for possible solutions, e.g. looking for a service with the same functionality, but ASIL D, or combining two of the ASIL B services.

2.3.6 Allocation of safety services to system lifecycle phases

According to ARROWHEAD, the lifecycle of a system can be divided coarsely in the phases engineering, commissioning, operation, maintenance, de-commission [8]. In figure **??**, a mapping of the various safety services to the different phases is proposed. The phases engineering and commission, as well as the phases maintenance and decommission are grouped, because there is no notable difference concerning the related services.

2.4 Service development process (use case scenario)

The preceding chapters analysed the concepts of SoA in embedded systems and safety services from a theoretical point of view. This final chapter investigates the design process of a safety-critical service by means of a simple use case. In detail, an *error detection service*, based on triple modular redundancy, as it is presented in section 2.3.3, serves as example.

The considered development stages for this service are an adoption of the development stages, provided by Erl, Bennett et al. in [11, p.116]. In detail, the first four stages are the

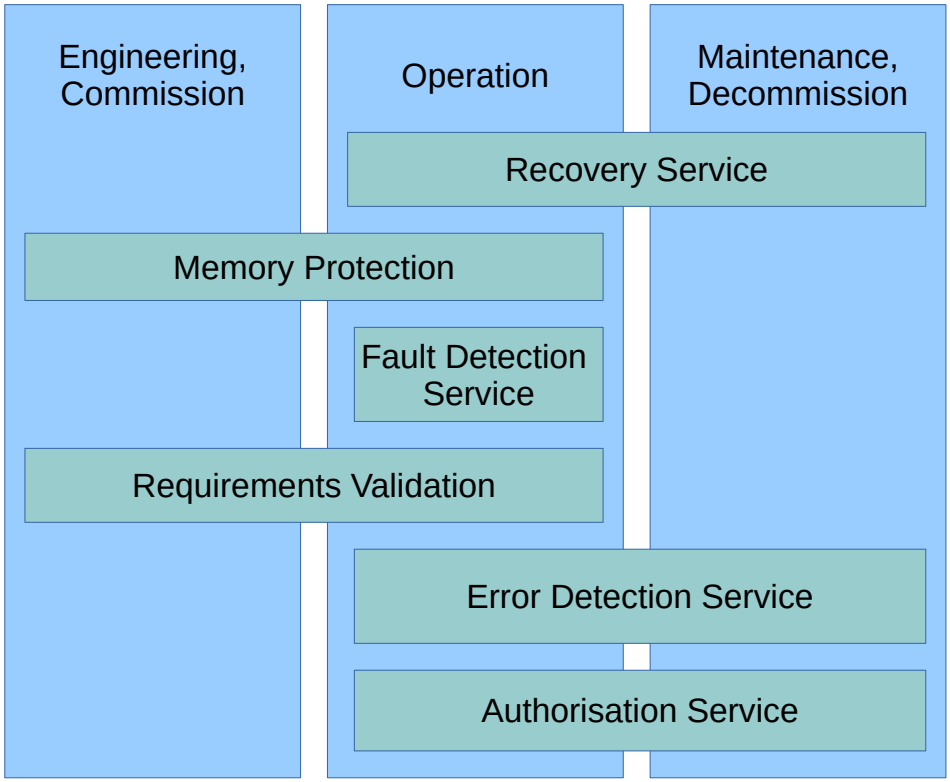


Figure 2.6: Allocation of the safety service to the different system lifecycle phases.

object of investigation. Because the development of a single service is considered, instead of the development of a whole SoA, those stages are renamed:

- Service investigation/planning,
- Service inventory analysis,
- Service oriented analysis and
- Service oriented design [11, p.116].

Each of the following four sections deals with one phase. They start off with a general description on what is conducted during this stage, before the example service is considered in this respect.

2.4.1 Service investigation/planning

This initial phase is concerned with the initial layout of the service. It is the phase where necessary requirements are listed and explored. There are several questions which need to be answered during this stage:

- What is the scope of the service?
- What are required capabilities?
- Which capabilities are not required?
- What are the safety requirements concerning this service?
- Should this functionality be implemented as service?

2.4.2 Service inventory analysis

The service inventory includes all services which could be provided by the vehicle. The difference to the service repository is that the service do not need to be available, or even implemented, but it is a static list of possible services.

During the *service inventory analysis* the inventory is searched for services which are required in order to build up the desired service. This simplifies the development, because much of the required functionality is already provided by other services and at the same time this step is responsible for preventing redundancies.

2.4.3 service oriented analysis

The outcome of the *service inventory analysis* phase is a so called *service candidate*. This denotes a conceptual service model before it is implemented by means of a specific language. According to [11, p.42], the concept of a language independent service candidate is applied, because the service undergoes a lot of changes in these early stages of development. Nevertheless, this may not be practicable in reality. It is not unlikely that the service templates (provided by Arrowhead) are used and extended from the very beginning of the development process.

During the *service oriented analysis* the service candidate is reviewed and checked with respect to *naming* and *normalisation*.

Service naming . The naming of the service candidate must be in accordance with other, existing services [11, p.206].

In terms of automotive, the naming is governed by certain standards, like AUTOSAR, which proposes naming conventions in **SW-C and System Modelling Guide** [51].

A unified naming convention is quite helpful, when dealing with standardised interfaces. It also provides the possibility to include relevant meta data into the name [52].

Service normalisation . Services within the same *service inventory* shall not have overlapping boundaries. In other words, redundant logic should generally be avoided, for it would violate the concept of a service oriented architecture. Accordingly, the services are forced to use existing services if those offer the required functionality [11, p.207].

Service Candidate Review . The final phase of this stage is a review by the related developers. A possible outcome of this review is the approval of changes or extensions to the service candidate [11, p.210].

In order to ensure an unbiased outcome, this review could be conducted by "external" people, which have not been included into the development process up to this step, but have a thorough knowledge of the SoA principles. Those people might think of a quite different approach for the same very same problem. This could raise new questions concerning the proposed design and composition.

A review can also take place if an existing and already implemented service is extended by one or more new capabilities [11, p.210].

2.4.4 service oriented design

Erl, Bennett at al. describe this phase as:

> *"Service-Oriented Design subjects this service candidate to additional considerations that shape it into a technical service contract in alignment with other service contracts being produced for the same service inventory"* - [11, p.86].

The aim of the *service oriented design* is to physically establish the service contract by filling out the corresponding templates. It is initiated when sufficient analysis has been conducted.

Chapter 3

Results

As a result of the investigation in chapter 2, a small use case scenario was conducted. In detail, the development process of an *error detection service* was investigated by means of the development phases featured in section 2.4. Due to the scarce reference material related to this topic, the use case scenario is rather short and quite theoretical. However, it provides a good example on what a service development process may look like.

As a result of the first four phases, which are concerned with the conceptual development, the contract is issued. Subsequently follows the actual implementation of the service by a developer, who was (most likely) not included in the preceding design phases. Therefore, the contract must provide all the necessary information for the implementation process, leaving no ambiguities or open questions.

It is then up to the developer to decide how the service is implemented, e.g. which programming language or approach to use. The outcome is a certain architecture, which can be based on arbitrary technologies.

3.1 Service investigation/ planning

Scope of the service . The service should be capable of detecting an error within a given ECR and act accordingly. Therefore he needs the signal of all the sensors he should mirror, as well as clock signal for creating sampling times.

Required capabilities .

- Detection an error in one of the sensors.

- Restart of an erroneous sensor.
- Purging the service from the service repository if more than two sensors fail and errors can no longer be detected.

Non-required capabilities .

- Detection of a fault.
- Detection of a failure.

3.2 Service inventory analysis

The error detection service will require:

- A clock service for creating sampling times,
- A fault detection service for detecting a fault in one of the mirrored services,
- An management service of the service repository, to update the repository in case of an erroneous service, and additionally perhaps
- A service for restarting another service, which is erroneous.

During operation, it will also need three services providing the measurement, which shall be checked for errors. These could be for example acceleration measurement services. Of course they should feature different implementation on independent hardware components.

3.3 Service oriented analysis

Concerning the naming, the example service should be in accordance with the AUTOSAR naming conventions [51]. Accordingly, a suitable candidate would be **error_detection**.

The *service normalisation* and *service candidate review* are not really doable just theoretically. Therefore, it shall be assumed that the example service has no overlapping functionality and has passed the review.

3.4 Service oriented design

Unfortunately, the service contract templates, which are designed by Arrowhead, were not yet available at the time this thesis was written. Accordingly, there could be no example contract be established.

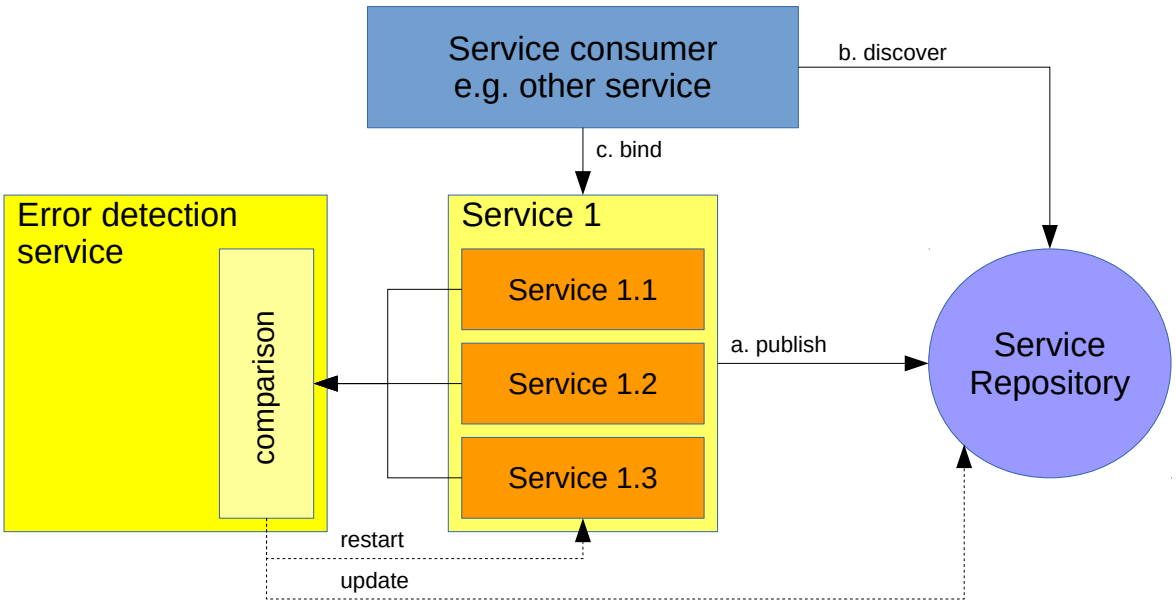
3.5 Possible implementation

As a result of the first four phases, which are concerned with the conceptual development, the contract is issued. Subsequently follows the actual implementation of the service by a developer, who was (most likely) not included in the preceding design phases. Therefore, the contract must provide all the necessary information for the implementation process, leaving no ambiguities or open questions.

It is then up to the developer to decide how the service is implemented, e.g. which programming language or approach to use. The outcome is a certain architecture, which can be based on arbitrary technologies.

A possible architecture resulting from the service of this use case is a triple modular redundancy approach (cf. section 2.3.3), as depicted in figure 3.1.

Figure 3.1: Architecture of the example service (error detection service).



Chapter 4

Discussion

Like chapter 2, this chapter is mostly a result from numerous meetings at VIRTUAL VEHICLE addressing the topic of service oriented architectures in automotive.

For the future of automotive with self-driving interconnected cars, the application of a service oriented architecture paradigm will be the logic choice. However, there are various obstacles which delay, or even prevent, this development. Two of them have been mentioned in section 2.2.

Those are the technical constraints due to the implemented hardware, which is not optimised for the application of the SoA design paradigm. The second was the safety-critical aspect of vehicles. The ISO 26262 standard does not issue any requirements concerning services or such. Generally, functional safety in **safety-critical embedded systems** is an area with various unsolved questions and issues as it is stated in this thesis.

A third drawback, which has not been mentioned so far, is the economic factor. The SoA for automotive requires dedicated hardware, which needs to be developed and adopted accordingly. In turn, this leads to high development and testing costs, before there is any benefit observable, because it is hard to communicate the advantages of a vehicle with a SoA inside to the end user. The benefits become only obvious, when there is a lot of environment and other vehicles which allow connection and communication with. In other words, all these concepts and ideas need a huge amount of resources and promotion to get them started in a useful way.

Chapter 5

Conclusion

The Work Package 1 (**Embedded Systems Architectures**) of the EMC2 project is currently in the first year of a total of three. This signifies that self-driving autonomous vehicles may still be quite a long time ahead, but there are already many abitious projects dealing with this issue.

Therefore, I personally think, the emerging of this "new" technology is just a matter of time.

Bibliography

[1] Iso 26262 for safety-related automotive e/e development: Introduction and overview.

[2] Lund Ericsson. Emc^2 service architecture, 2015.

[3] R. Omermaisser and H. Kopetz. *GENESYS - A Candidate for an ARTEMIS Cross-Domain Reference Architecture for Embedded Systems*. Vienna University of Technology, 2009.

[4] An introduction to autosar.

[5] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: service-oriented architecture best practices*. Prentice Hall PTR, Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, Indianapolis, first edition, 2005.

[6] K. Küpper, P. Teufelberger, R. Ellinger, and E. Korsunsky. From vehicle requirements to modular hybrid software. *ATZ worldwide eMagazine*, pages 46–49, 2011.

[7] Iso/iec/ieee systems and software engineering - architecture description, 2011.

[8] P. Clements and L. Northrop. Software product lines, 2003.

[9] F. Blomstedt, L. Ferreira, and et al. The arrowhead approach for soa application development and documentation, 2014.

[10] Architecture of the car2x systems network.

[11] Thomas Erl, Stephen G Bennett, Benjamin Carlyle, Clive Gee, Robert Laird, Anne Thomas Manes, Robert Moores, and Andre Tost. *SOA Governance*. Pearson Education, 2011.

[12] Ashraf M El-Attar and Gamal Fahmy. An improved watchdog timer to enhance imaging system reliability in the presence of soft errors. In *Signal Processing and Information Technology, 2007 IEEE International Symposium on*, pages 1100–1104. IEEE, 2007.

[13] Systems and software engineering - system life cycle processes, 2015.

[14] Iso 26262 road vehicles - functional safety - part 1: Vocabulary, 2011.

[15] Jerker Delsing. Deliverable d1.2 of work package 1, 2015.

[16] AUTOSAR. Glossary, autosar release 4.2.1, 2014.

[17] Douglas Rodrigues, Rayner15 de Melo Pires, Júlio Cézar Estrella, Emerson Alberto Marconato, Onofre Trindade, and Kalinka Regina Lucas Jaquie Castelo Branco. Using soa in critical-embedded systems. In *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, pages 733–738. IEEE, 2011.

[18] Peter Marwedel. *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer Science & Business Media, 2010.

[19] Cesare Alippi. *Intelligence for Embedded Systems*. Springer, 2014.

[20] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[21] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-oriented Programming*. ACM Press Series. ACM Press, 2002.

[22] F. Kirschke-Biller. Autosar - a worldwide standard, current developments, roll-out and outlook. *15th International VDI Congress Electronic Systems for Vehicles 2011*, 2011.

[23] Serviceorientation.com. `http://www.serviceorientation.com`. Accessed: 2015-06-29.

[24] J. Sametinger. *Software Engineering with Reusable Components*. Springer Berlin Heidelberg, 2013.

[25] Jim Q Ning. Component-based software engineering (cbse). In *Assessment of Software Tools and Technologies, 1997., Proceedings Fifth International Symposium on*, pages 34–43. IEEE, 1997.

[26] Thomas Erl. *Soa: principles of service design*, volume 1. Prentice Hall Upper Saddle River, 2008.

[27] The Open Group. The soa source book. http://www.opengroup.org/soa/source-book/intro/. Visited: April 2015.

[28] H. Breivold and M. Larsson. Component-based and service-oriented software engineering: Key concepts and principles. *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference*, pages 13–20, 2007.

[29] Andreas Scholz, Stephan Sommer, Christian Buckl, Gerd Kainz, Alfons Kemper, Alois Knoll, Jörg Heuer, and Anton Schmitt. Towards and adaptive execution of applications in heterogeneous embedded networks. In *Software Engineering for Sensor Network Applications (SESENA 2010)*. ACM/IEEE, 2010.

[30] N. Josuttis. *SOA in Practice*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2007.

[31] Software Engineering Institute. Defining software architectures. http://www/sei.cmu.edu/architecture/. Visited: May 2015.

[32] Stephan Sommer, Christian Buckl, Gerd Kainz, Andreas Scholz, Irina Gaponova, Alois Knoll, Alfons Kemper, Jörg Heuer, and Anton Schmitt. Service migration scenarios for embedded networks. In *The Fifth International Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE 2010)*. IEEE, 2010.

[33] M. Rosen, B. Lublinsky, and K. Smith. *Applied SOA: SERVICE-ORIENTED ARCHITECTURE AND DESIGN STRATEGIES*. Wiley Publishing, Inc., Indianapolis, 2008.

[34] OASIS. Oasis reference model for service oriented architecture 1.0, committee specification 1. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm, 2006.

[35] Mike P Papazoglou and Willem-Jan Van Den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB journal*, 16(3):389–415, 2007.

[36] Renato Donini, Stefano Marrone, Nicola Mazzocca, Antonio Orazzo, Domenico Papa, and Salvatore Venticinque. Testing complex safety-critical systems in soa context. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, pages 87–93. IEEE, 2008.

[37] E. Lessner and P. Ostroumov. Reliability and availability studies in the ria driver linac, 2005.

[38] V. Nelson. Fault-tolerant computing: fundamental concepts, 1990.

[39] Heimo Zeilinger, Berndt Sevcik, Thomas Turek, and Gerhard Zucker. Communication in change–voice over ip in safety and security critical communication networks. In *IT Revolutions*, pages 186–193. Springer, 2009.

[40] H. Zhang, W. Li, and J. Qin. Model-based functional safety analysis method for automotive embedded system application, 2010.

[41] Amber Israr, Sorin Huss, et al. Reliable system design using decision diagrams in presence of hard and soft errors. In *Applied Sciences and Technology (IBCAST), 2014 11th International Bhurban Conference on*, pages 1–9. IEEE, 2014.

[42] International Electrotechnical Commission, International Electrotechnical Commission, et al. *Analysis Techniques for System Reliability: Procedure for Failure Mode and Effects Analysis (FMEA)*. International Electrotechnical Commission, 2006.

[43] United States. Department of Defense. *Mil-Std-1629a: 1980: Procedures for Performing a Failure Mode, Effects and Criticality Analysis*. Department of Defense, 1980.

[44] Christian Buckl, Stephan Sommer, Andreas Scholz, Alois Knoll, Alfons Kemper, Jörg Heuer, and Anton Schmitt. Services to the field: An approach for resource constrained sensor/actor networks. In *The Fourth Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE 2009) - extended version*. IEEE, 2009.

[45] Dae-Hyun Kum, Joonwoo Son, Seon-bong Lee, and Ivan Wilson. Automated testing for automotive embedded systems. In *SICE-ICASE, 2006. International Joint Conference*, pages 4414–4418. IEEE, 2006.

[46] Thomas Turek, Tayyaba Anees, and Simon-Alexander Zerawa. Towards safety and security critical communication systems based on soa paradigm. In *Industrial Electronics (ISIE), 2011 IEEE International Symposium on*, pages 1248–1253. IEEE, 2011.

[47] Iso 26262 for safety-related automotive e/e development: Concept phase.

[48] Wikipedia. Triple modular redundancy. `http://en.wikipedia.org/wiki/Triple_modular_redundancy`. Visited: May 2015.

[49] Shimpei Yamada, Yukikazu Nakamoto, Takuya Azumi, Hiroshi Oyama, and Hiroaki Takada. Generic memory protection mechanism for embedded system and its application to embedded component systems. In *Computer and Information Technology Workshops, 2008. CIT Workshops 2008. IEEE 8th International Conference on*, pages 557–562. IEEE, 2008.

[50] Shigeru Yamada and Yukikazu Nakamoto. Protection mechanism in privileged memory space for embedded systems, real-time os. In *Distributed Computing Systems Workshops (ICDCSW), 2014 IEEE 34th International Conference on*, pages 161–166. IEEE, 2014.

[51] Sw-c and system modelling guide. `http://www.autosar.org`.

[52] Dipl-Math Ute Rehner et al. Naming issues in autosar. *ATZextra worldwide*, 18(9):57–59, 2013.