Preface

This thesis was written as result of an internship at VIRTUAL VEHICLE at In\_eldgasse, 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 and concentrates on

the four main research areas Thermo- & Fluid Dynamics, Mechanics & Materials, NVH &

Friction and E/E & Software. For the time of my employment I worked as member of the

Electrics/Electronics (E/E) & Software area, with an emphasis on functional safety. I was

assigned to the European project EMC2, a part of the ARTEMIS programme which focuses

on embedded multi-core systems for mixed criticality applications in dynamic and changeable

real-time environments.

This thesis consists to a great extent of two documents which were produced during

this internship. The \_rst document is a glossary, which aims at de\_ning certain terms

and unifying the opinions from di\_erent research areas. The second document contains an

extensive investigation on how the service oriented architecture paradigm can be applied in a

safety-critical embedded systems like vehicles.

Although the employment was oriented towards the automotive industry, the investigated

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 here

considered with respect to automotive, could also become an important issue for aviation in

near future.

At this point I want to thank my supervisor from FH JOANNEUM, FH-Prof. Dipl.Ing. Dr.

Holger Fluhr, my supervisor provided by the company, Dipl.Ing. Helmut Martin, as well as my

project team members Dipl.Ing.Dr. Andrea Leitner and Mr. Mario Driussi, who supported

me in developing the ideas and concepts featured in this thesis during numerous meetings and

discussions.

Abstract

One of the major issues in the automotive and aeronautics industry is the constantly 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 the event of error. This is conducted by identifying

possible hazards, estimating the potential risks and developing necessary countermeasures,

based on these investigations. These processes require an accurate and thorough comprehension

of the observed E/E system.

The service oriented architecture is a design paradigm which focuses on the concept of

software reuse by implementing functionalities as technology independent and loosely coupled

services. Although this architectural style is already widely applied in web applications, it has

not yet found its way into safety-critical embedded systems.

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 \_rst part

of the thesis mainly deals with terms that are crucial for the subsequent presented

concepts, and investigates them with respect to embedded systems. On this basis it is de\_ned

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 speci\_ed in

the ISO 26262 standard. The ISO 26262 standard is an international safety standard for

safety-critical E/E systems in vehicles with a maximum gross weight of 3.500 kg.

vi

Kurzfassung

Eine der gro\_en Harausforderungen der Automobil- und Luftfahrtindustrie ist die konstant

zunehmende Komplexitat von elektronischen Systemen und die dadurch entstehende Fehlerfortp

anzung aufgrund der weitgehenden Vernetzung dieser. Der Bereich der funktionalen

Sicherheit beschaftigt sich mit der Vermeidung von nicht tolerierbaren Risiken im Fehlerfall.

Dies wird durch die Identi\_zierung von moglichen Gefahren, der Abschatzung von potentiellen

Risiken und, basierend darauf, der Entwicklung von notwendigen Gegenma\_nahmen

gewahrleistet. Diese Prozesse setzen ein detailiertes und umfassendes Verstandnis der betrachteten

elektronischen Systeme voraus.

Die serviceorientierte Architektur ist ein Designprinzip, das sich auf das Konzept der

Wiederverwendung von Software konzentriert. Dies wird durch die Implementierung von Funktionalit

aten als technologieunabhangige und lose gekoppelte Services bewerkstelligt. Obwohl

dieser Architekturtyp bereits weitgehend fur Webanwendungen eingesetzt wird, hat er noch

nicht in sicherheitskritische eingebettete Systeme Einzug gehalten.

Diese Bachelorarbeit untersucht die Verwendung von Serviceorientierten Architekturen in

solchen Systemen und legt dabei einen Schwerpunkt auf funktionale Sicherheit und Fehlertoleranz.

Der erste Teil der Arbeit de\_niert einige Begri\_e, die fur das Verstandnis der vorgestellten

Konzepte ausschlaggebend sind, und untersucht sie im Bezug auf eingebettet Systeme.

Darauf basierend wird de\_niert was Sicherheit as Service im Allgemeinen bedeuten kann, und

wie die schlussendliche Implementierung in ein bestehendes System aussehen konnte, um die

Aussagen des ISO-Standards ISO 26262 zu erfullen. Der ISO 26262 Standard ist ein internationaler

Standard fur sicherheitskritische elektronische Systeme in Fahrzeugen mit einem

maximal zulassigen Gesamtgewicht bis 3,500 kg.

vii

Chapter 1

Introduction

The SoA (Service oriented Architecture) design principle is a promising design philosophy

which features a lot of advantages over the static and vendor dependent architectures, which

are implemented in today's vehicles and aircraft. Hence, it is not surprising that there have

already been various projects dealing with the application of SoAs in real-time embedded

systems. Those include SIRENA, SOCRATES, OASiS, MORE, RUNES and "SOA [1] [2] [3].

However, they do not address the necessary functional safety and fault tolerance requirements,

which are preceding in vehicles. Work Package 1, \Embedded System Architectures", of the

EMC2 project is dedicated to the investigation of these very issues, as this design paradigm

might give way to a new generation of vehicles which are able to interconnect and operate

autonomously.

One of the most important sources for this thesis was the ISO 26262 standard. Emerging

from the IEC 61508 standard, the ISO 26262 standard is an international functional safety

standard for series production passenger cars with a maximum gross weight of 3.500 kg. It is

the state of the art for vehicles and does not set any requirements in terms of performance of

equipment, but is only concerned with possible malfunctions. In total the standard features

ten parts. Part 1, which de\_nes all the related terms and

vocabulary, part 3 which deals with the concept phase, and part 4, which is dedicated to

the development at system level [4] [5] [6] have been of particular relevance.

The standard does not issue any regulations concerning SoA, but only provides certain

requirements which have to be ful\_lled. Nevertheless, there are no prescriptions on how they

should be ful\_lled [7].

Another important source of information was AUTOSAR. The term AUTOSAR is ambiguous

as it can denote either the technical product (AUTomotive Open System ARchitecture),

1

or the related development partnership [8]. The partnership was founded in 2003, with its

members covering more than 80% of the production of cars worldwide [9] [10]. They provide

a standard which aims at establishing an industry norm for automotive software architecture.

This allows di\_erent partners, as well as suppliers and manufacturers, to collaborate without

any obstacles in terms of languages or methodologies. In detail, this is achieved by the

de\_nition of a uni\_ed software architecture and software development methodology, as well

as by standardised application interfaces. By stressing the decoupling of hardware and software,

the standard gives way to software reuse on di\_erent hardware platforms [9] [10], which is

in compliance with the SoA design principles. Thus, many of the terms treated within this

thesis are inuenced by the viewpoint of AUTOSAR.

2

Chapter 2

Methods

This chapter contains a de\_nition of the most important terms related to SoA. Those terms

include system, component, service, architecture, service oriented architecture, dependability

and functional safety. Each of the following sections starts with an investigation and comparison

of viewpoints from di\_erent sources. Subsequently, a de\_nition which is in accordance

with the context of safety-critical embedded systems is presented in a bordered box. This

de\_nition is a combination of valid information from various sources as well as own \_ndings.

Most of the sections include also some additional information to the respective term.

2.1 System

In their GENESYS reference architecture Obermaisser and Kopetz describe a system as \an

entity that is capable of interacting with its environment and is sensitive to the progression of

time" [11, p.7]. The environment is thereby a system itself which produces input for other

systems and acts according to their outputs. Which elements (cf. section 2.1.1) belong to

the system, and which to the environment, is a matter of perspective.

Within the ISO 26262 standard, a system is referred to as a \set of elements that relates

at least a sensor, a controller and an actuator with one another" [4]. This de\_nition

obviously explicitly refers to automotive because in other industry sectors a system does

not necessarily contain actuators.

AUTOSAR, on the other hand, describes a system as \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" [12].

Other typical characteristics are the presence of some kind of internal structure and the

3

hierarchical composition. Those are included in the resulting de\_nition below.

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 speci\_c 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 therefore consists of other vehicles and the surrounding infrastructure.

Nevertheless, the entire tra\_c could also be taken as a system and the environment

would then be a di\_erent one.

2.1.1 System Element

In the ISO 26262 standard system elements are described as \system or part of system including

components hardware, software, hardware parts, and software units" [4]. In general,

system element is a very generic term and does not refer to any entities at a speci\_c layer or

with a speci\_c characteristic. Instead, it can be more or less any entity of a system, since a

system itself is de\_ned as set of elements [4]. This is depicted in \_gure 2.1, which also shows

the naming convention for other terms as it is used throughout this thesis.

**ISO 26262 Introduction and Overview 62**

**Basic Terminology**

Function

Item

System

Component

Part/Unit

Element

m n

1

1

n

n

n

1

1

n

Figure 2.1: Relation of system, component and element according to ISO 26262 [7].

4

2.1.2 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) [11, p.7].

Thus, the SoS is in general the level above a given system and is therefore dependent on the

de\_nition of the related systems. SoSs may be geographically distributed and can become

parts of other, bigger SoSs, when collaborating with other SoSs.

For a system vehicle, a SoS could be for example the tra\_c of a city, with many vehicles

participating.

2.1.3 System Layers

The parts of a system can be divided into di\_erent layers of implementation. This kind of

abstraction enables the comprehension of the overall relations. Unfortunately each \_eld of

research features its own, sometimes even contradicting, way of fractionising systems.

The GENESYS architecture by Obermaisser and Kopetz distinguishes between three di\_erent

layers, denoted chip-level, device-level and system-level [11, p.44]. An example of the

hardware elements at di\_erent levels by means of a system can be seen in \_gure 2.2.

System Level. The system level consists of devices which are themselves logically selfcontained

apparatus. With a vehicle as system this could be for example an ECU, a

sensor, an actuator or the like [11, p.45].

Device Level. The devices at the system level contain a certain internal structures themselves.

In terms of embedded systems those are in most cases chips [11, p.45], like the

the AURIXTM 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 [11, p.46]

Another classi\_cation of layers is given by the Arrowhead Framework, which provides an

hierarchy by means of documentation documents. Those are split into the three levels system-

of-systems, system and service [13]. Their involved documents and relations are pictured in

\_gure 2.3.

5

System

Integration Level

Device

Integration Level

Chip

Integration Level

**SYSTEM -** *VEHICLE*

**DEVICE -** *ECU*

**CHIP -** *AURIX*

IP Core IP Core

IP Core IP Core

Chip Chip

Chip

Device

*Sensor*

Device

*Accelerator*

Device

Figure 2.2: Hierarchy of the implementation layers of the system vehicle with examples.

System-of-systems. This level features the two documents SoS Description (SoSD) and

SoS Design Description (SoSDD). They di\_er in the amount of information they are

revealing. While the \_rst represents only an abstract view, the second also reveals the

implementation of the SoS and its technologies [13].

System. The system level contains with the two documents System Description (SysD) and

System Design Description (SysDD). Similarly to the SoS level the \_rst one features a

kind of black box opacity and the second a white box view [13].

Service. The service level contains the four documents Service Description (SD), Interface

Design Description (IDD), Communication Pro\_le (CP) and Semantic Pro\_le (SP). The

service description is referred to in section 2.3.2.

All these documents exist as templates which should be \_lled out during the development

process of a system. They should feature a XML like style in order to be human and machine

readable at the same time.

The fully developed systems can then be constituted to a SoS by an underlying cloud,

as depicted in \_gure 2.4. All the system have speci\_ed and standardised interfaces and

work together by means of three core services, denoted Information Assurance, Information

6

Figure 2.3: Relation of System, SoS and Service to one another [14].

Infrastructure and System Management.

Information Assurance (IA). This service is responsible for providing secure information

exchange through authorization and authentication [14].

Information Infrastructure (II). The II service enables the listing of the services in the

service repository (cf. section 2.5.2) and their discoverability [14].

System Management (SM). This is the core service for the SoS composition and features

logging and monitoring abilities [14].

When comparing these two approaches by GENESYS and ARROWHEAD, it becomes

obvious that the latter has a quite abstract point of view which is biased towards SoA, while

the approach by GENESYS is much more static and hardware oriented.

2.1.4 Embedded System

Embedded systems (ES) are computational modules integrated to physical devices and equipment.

They have a prede\_ned set of tasks and requirements and are capable of processing

information [15] [16, p.xiii]. Compared to general-purpose computation systems ES usually

dispose of less processing resources and come with narrower operation ranges. But at the

same time they feature a high e\_ciency by optimally managing the available resources [17,

p.283] [16, p.5]. Also, their presence is usually quite unobtrusive, because instead of mouse

7

www.arrowhead.eu

Arrowhead Framework

Common systems and components

ARROWHEAD

FRAMEWORK

COMPLIANT

NETWORK

IA

SM

II

Application

system

Application

system

Application

system

Application

system

Application

system

Application

system

Figure 2.4: Implementation example of the Arrowhead framework [14].

and keyboard the user typical interface consists of input devices like buttons, steering wheels,

or pedals.

ESs are reactive systems, whhich 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 [16,

p.8-9]. During operation, ESs are in a certain state and waiting for input. When provided

with that, they perform computations and generate an output which is handed back to the

environment [16, p.9]. In safety-critical applications, issues like time constraints, dependability

and e\_ciency requirements also have to be considered.

Time constraints. One challenge of ESs is the meeting of so called time constraints, which

basically means the conduction of a computation within a given period of time [16,

p.8-9] [15]. Kopetz [18] states \A time-constraint is called hard if not meeting that

constraint could result in a catastrophe."

Dependability. ESs, 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 can be split up in further aspects,

which are reliability (cf. section 2.6.1), maintainability, availability (cf. section 2.6.2),

safety (cf. chapter 2.7) and security [16, p.4-5].

E\_ciency requirements. E\_ciency is a key concept of ESs and is concerned with provid-

8

ing a maximum computation performance while minimizing the required energy. The

e\_ciency is measured in operations per Joule and has been been increasing almost

exponentially over the last twenty years [16].

2.2 Component

The term component often appears in connection with SoA and frequently leads to confusion

when it is put on a level with service (cf. chapter 2.3). This ambiguity is a result of the

historic development of the SoA as successor of the component based software engineering

(CBSE).

Obermaisser and Kopetz state that a component is a software or hardware unit that

performs a speci\_ed computation within a given period of time [11, p.38] and communicates

with other components by means of dedicated interfaces (cf. section 2.2.1). Like systems,

components are hierarchical and therefore dependent on the point of view. Hence, a quantity

of components may be seen as a single component from a di\_erent point of view. The

ISO 26262 standard is in accordance with this de\_nition and describes a component as \nonsystem

level element that is logically and technically separable and is comprised of more than

one hardware part or one or more software units" [4].

In contrast to that, a component is explicitly referred to as a piece of software by

AUTOSAR:

\Software-Components are architectural elements that provide and/or require interfaces

and are connected to each other through the Virtual Function Bus to

ful\_ll architectural responsibilities" [12].

Hence, a software component (SW-C) is an encapsulation of parts of the automotive

functionality. There is no speci\_c granularity dictated, meaning that an AUTOSAR software

component might be either a \small, reusable piece of functionality (such as a \_lter) or a

larger block encapsulating an entire subsystem" [19].

A component is a logical and technical separable hardware or software unit that

is capable of performing a speci\_c computation.

It o\_ers an abstraction that simpli\_es the understanding of complex systems.

Accordingly, components are hierarchical, meaning that they may be composed

9

to other, larger, components.

As suggested by this de\_nition, the term component may be used for both, software and

hardware. A component can be seen as black box, meaning that the more or less complex

internal structure is invisible or of no concern for the user. Hence, other components stay

una\_ected from modi\_cations of this internal structure, given that the behaviour at the Linking

Interface (cf. section 2.2.1) remains unchanged [11, p.38-39] [20] [21]. As a self-contained

subsystem, a component can be developed and tested independently and later be used as

building block for systems or higher level components. In other words, the components can

be described as the basic building blocks of a system [22].

Nevertheless, not everything is a component. According to Sametinger [21, p.2-3], 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-de\_ned interfaces in order to become

a component.

2.2.1 Component Interfaces

The interfaces are necessary for any interaction with other system elements. Following the

de\_nition 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 \_gure

2.5 [11, p.40-41].

Linking Interface (LIF). The LIF is a message based interface and responsible for o\_ering

the component's services. Its denotion is dependent on the level of integration, e.g.

Inter-IP Core LIF at chip level or Inter-Chip LIF at device level. Nevertheless, the LIF

is used only for communication to other components at the same layer and it is also the

only place where a component may provide its services to other components [11, p.9].

The LIFs are always technology agnostic, which means that they do not expose details of

the component's implementation or Local Interfaces. Accordingly, the implementation

of the component can be modi\_ed, without other components noticing, as long as the

speci\_cation at the LIF remains unchanged [11, p.9, 40-41].

10

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 modi\_ed, the semantics and timing of the data should stay the same in

order to not violate the speci\_cation. 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 di\_erent layers to communicate with each other.

However, components do not necessarily require local interfaces. Such are denoted

Closed Components [11, p.40-41].

Technology Independent Interface (TII). The TII is the instrument for con\_guring and

recon\_guring a component, e.g. assigning a name, con\_guring 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 [11, p.40-41].

Technology Dependent Interface (TDI). The TDI 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 [11, p.40-41].

Figure 2.5: Interfaces of a component, with respect to the GENESYS architecture [11, p.40]

Compared to this rather implementation oriented point of view by GENESYS, the approach

by AUTOSAR is much more abstract. According to their de\_nition, a component may dispose

11

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 speci\_es

a kind of contract or speci\_cation, on which services can be called at this port and the format

of the data emitted at this port. There are four di\_erent types of interfaces, belonging to the

two di\_erent communication patterns Client-Server and Sender-Receiver [20]:

Client-Server Communication. When this kind of communication is performed the client-

component requests a speci\_c service from the server-component and sends necessary

parameters. The server then processes the incoming request and returns a response.

Nevertheless, a single component can be a server and a client at the same time [20]. A

schematic illustration of this type of communication is pictured in \_gure 2.6.

Sender-Receiver Communication. The Sender-Receiver approach is a bit di\_erent. 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 ow. 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. \_gure 2.7) [20].

Figure 2.6: Illustration of the client-server communication [20].

2.3 Service

The perception of the term service is quite wide spread and inuenced by a person's experience

as well as the professional environment, e.g. the related research area. Hence, numerous

di\_erent perceptions emerged, supporting various, even contradicting, views.

A quite generic de\_nition of service is given by Arcitura [23]:

12

Figure 2.7: Illustration of the sender-receiver communication [20].

\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)."

The key information in this quote is that a service may o\_er multiple capabilities.

By Arrowhead a service is referred to as a part of a SoA [24]:

\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."

The de\_nition by Obermaisser and Kopetz features a quite di\_erent point of view and

describes a service by means of its surrounding environment [11, p.8]:

\A service is what a system delivers to its environment according to the speci-

\_cation. Through its service, a system can support the environment, i.e., other

systems that use the service."

What the speci\_cation of a service may look like is referred to in 2.3.2. While it is only

indicated that a service interacts somehow with its environment, AUTOSAR gives a concrete

answer to the question how this interaction is conducted [12]:

\A service is a type of operation that has a published speci\_cation of interface

and behaviour, involving a contract between the provider of the capability and

the potential clients."

13

The terms contract, provider and client, which also appear in the de\_nition below are investigated

in detail in section 2.5.2.

As part of a system, a service is an independent logical unit with at least one

capability and well de\_ned interfaces, which have to be fully described by the

service contract.

Services are the building blocks of a SoA and the containers for the func-

tionalities a service provider o\_ers to its consumers. In order to be applied in a

service oriented architecture, a service must comply with certain key concepts.

2.3.1 Key Concepts of a Service

Erl et al. [25, p.27] state eight speci\_c 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

di\_erent nominations of equal characteristics. In those cases the alternative nominations are

stated as well.

Opacity/ Encapsulation/ Abstraction. According to Erl [26, ch.8.1.], abstraction means

to \hide information about a program not absolutely required for others to e\_ectively

use that program." 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] [24] [25, p.27]. This black box encapsulation disables any

modi\_cation of the service by the user and is often referred to as service interface level

abstraction [28].

Reusability. The idea of software reuse has been present since the early days of software engineering.

In SoAs it is the key concept of a service and a necessary basis, as many of the other

concepts would not even be possible without it [26, ch.9.1.] [25, p.27].

14

This design principle aims at making a service applicable for more than just one speci\_c

use case. Usually, it results in a more generic programming logic, which allows a wider

range of application.

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.

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] [24] [28] [25, p.27].

With respect to SoAs, the composing may take place at runtime [28].

Loose coupling. If two or more artefacts are somehow connected within a technical context,

this is referred to by the term coupling. It indicates that two or more of \something"

exist 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, o\_ering the

service provider great exibility in choosing design and deployment environment for

o\_ering their services [28] [24]. Well de\_ned interfaces also allow a simple exchange of

components by components from di\_erent vendors [1].

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 \_les and illustrate them in a

user friendly way". No matter which web browser is used, the end result will stay almost

una\_ected thanks to the well speci\_ed 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,

15

\_ 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 artefact that stores the service information in SoAs is the service repository (cf.

section 2.5.2).

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] [24] [28] [25, 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 ful\_l their functionality [26, ch.11].

On the other hand, services can also be stateless; and in order to optimise reusability,

services should aim at minimising their state information and their holding time for

messages [28] [25, p.27].

Technology neutrality. Services should be independent from used technology in order to

allow di\_erent 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 Erl [25, p.27], \Services within the same

service inventory are in compliance with the same contract design standards." In other

words, the service contract should be created by means of a given template, which is

applied, at least, system-wide. The term service inventory is referred to in 2.5.2.

16

2.3.2 Structure of a Service

Concerning the structure, Krafzig [29, p.44] describes a service as it can be seen in \_gure 2.8

with the artefacts service contract, interface, implementation, business logic and data.

Service contract. \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, as it is the complete

speci\_cation 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 [29, p.44] [30, p.26] [28].

Physically, the service contract is represented by one or more service description doc-

uments, which should be human- and machine readable at the same time. In terms

of web services the contract is usually represented by WSDL (Web Service Description

Language) documents [25, p.43]. There is no dedicated language standard for

automotive yet, but a XML based language like the WSDL 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 service, 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 speci\_es the access points

and the functionalities of the service to the customers, which are connected via a

network. A service may dispose of multiple interfaces [29, p.44] [28].

Implementation. The implementation is realised by programmes, con\_guration data and

databases, which are necessary to provide the functionality speci\_ed in the contract.

The term business logic in \_gure 2.8 is a bit misleading and should be seen as the

algorithms encapsulated in the implementation [29, p.44].

17

Data. As stated in 2.3.1, 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 [29, p.44].

Figure 2.8: Structure of a service with the relations of the particular artefacts [29, p.45].

2.3.3 Services at Di\_erent Layers of Implementation

The system layers speci\_ed in 2.1.3 serve as basis for the assignment of services to di\_erent

levels of implementation.

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. In accordance with that, the services at

this layer are denoted core services [11, p.44].

Typical examples for services located at this level are message based communication services

for the interaction of system elements, 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 [11, p.7-12]. In other

words, the chip layer provides a platform where recurring problems can be dealt with once

and for all.

18

Device Layer

The device layer contains more advanced hardware parts. With respect to \_gure 2.9 these

might be sensors, actuators, ADCs (Analog-to-digital converters), AURIX chips, the CAN

(Controller Area Network) bus 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 thesis.

Device services 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. Same as with the device services, there is no uniform denomination

for services at this layer throughout literature. Thus, 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 2.9 features an example of which hardware parts may belong to which integration

layer with respect to a vehicle as considered the system.

2.4 Architecture

The term architecture is very generic and belongs to various di\_erent domains, like hardware

architecture, software architecture, system architecture or enterprise architecture. In general,

architecture is concerned with how the components of a system can be arranged and

interrelated in order to assemble an overall system [32] [22].

Within the ISO/IEC/IEEE 42010 standard [32] 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." The fundamental concepts are

thereby the system elements (cf. section 2.1.1), the relations inside the system, the relations

to the environment and the principles of design and evolution [32].

19

Core 1 Core 0 Core 2 Core 3

ADC **AURIX** CAN WD

Sensor **ECU** Actuator

HMI **PT**

**VEHICLE**

Chip

Layer

Device

Layer

System

Layer

Figure 2.9: Examples of hardware parts at di\_erent levels [31]

This de\_nition applies to any kind of architecture, but the emphases on these concepts

vary with respect to the considered domain. The software architecture usually focuses very

much on the system elements, while the enterprise architecture is more concerned with the

principles of design and evolution [32].

This is in compliance with the de\_nition of architecture by AUTOSAR [12]:

\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."

In [15] it is stated that the relationship and principles of design of the components,

functions and the interface established between subsystems can also be de\_ned by architecture.

The de\_nition presented below is compatible with all kinds of architecture design, for they

share the same basic principles.

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 speci\_es also

the connections and interactions of a system to its environment.

At the same time, architecture is also responsible for determining the prin-

ciples of design and evolution.

20

2.4.1 Demarcation from Related Terms

The placement of architecture in relation to other entities like system or environment is

illustrated in \_gure 2.10.

Architecture description. Many sources mix up the de\_nitions of architecture and archi-

tecture description, which is a recurring source of confusion. In contrast to the actual

architecture of a system, the term architecture description denotes the artefacts which

document the respective architecture [32].

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:

\_ Speci\_cation 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 [32].

One and the same architecture can be described by several di\_erent architecture descriptions,

and at the same time, an architecture description can also characterise multiple

architectures [32].

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 [32].

System concern. A speci\_c system concern can be held by one or more stakeholders and

can appear in various di\_erent forms, e.g. expectations, responsibilities, requirements,

assumptions, dependencies and more [32].

Purpose. Purposes are one kind of system concerns, which are issued by the stakeholders

[32].

21

Figure 2.10: Relations of architecture to other entities [32]

2.5 Service Oriented Architecture

The term service oriented architecture has been widely used for marketing and praising new

products. 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 was 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 bene\_ts that

come with the modularisation of software. Those are not only reduction of development

costs, but, with respect to embedded systems, also a saving in hardware components and

complexity [2]. Furthermore, it increases exibility, scalability and fault tolerance [33, p.33]

[30, p.17-18].

Di\_erent authors and companies explain the term from di\_erent viewpoints and with different

focuses.

OASIS. OASIS is a non-pro\_t 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 di\_erent ownership domains. It provides a uniform means to of-

fer, discover, interact with and use capabilities to produce desired e\_ects

consistent with measurable preconditions and expectations" [34].

22

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 ap-

plications that use services available in a network such as the web and it

represents the widest accepted model to design geographical distributed sys-

tems" [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 de\_nition presented below has emerged as result of extensive investigations and reviewing

numerous sources. In contrast to all the other de\_nitions, which are somehow biased due to

their relation to a certain research area, this one is quite generic and may be used for any

kind of SoA. In case of embedded- or safety-critical systems, there are of course additional

characteristics which are crucial.

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 with prede\_ned protocols.

As stated, SoA is no speci\_c implementation but a set of rules to achieve a certain target

state. The \_nal implementation can therefore consist of multiple technologies, products, APIs

(Application Programmers Interfaces) and supporting infrastructures [25, p.29]. This enables

an unproblematic exchange of components by components of other vendors, as long as the

provided services remain una\_ected.

23

Nevertheless, a SoA is not just a simple collection of services, but o\_ers also composition

mechanisms 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].

2.5.1 Historic Development

The term Service oriented Architecture \_rst appeared in 1996 [33, p.7]. It emerged from

the CBSE (cf. \_gure 2.11) and was more or less 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 di\_erences [28].

**Software Product Lines**

Modern software demands higher quality and shorter time-to-market cycles at lower costs,

making reuse more important than ever before. Figure 1 illustrates that the idea of reuse is not

new, but has evolved over time. The latest development attempts to make reuse of software more

systematic.

**Figure 1: Development from ad-hoc to systematic reuse in software development [CN03]**

One approach propagating systematic software reuse are software product lines (SPL). The core

idea is to build multiple products from a single infrastructure in a way that is aligned to stated

business goals. An often used definition from Northrop and Clements [NC02] describes a

software product line as *“a set of software-intensive systems sharing a common, managed set of*

*features that satisfy the specific needs of a particular market segment or mission and that are*

*developed from a common set of core assets in a prescribed way.”*

Basically, software product line development consists of two fundamentals:

• The differentiation of domain and application engineering and

• the separation of commonalities and variability in domain engineering.

Mass customization (the ability to efficiently customize a product for specific customer needs) is

one main aim of software product lines. To realize this, variability has to be made explicit and

managed in an effective manner.

Product line adoption is still a challenge in practice. Existing strategies can be classified into

three groups [Kru02, PBvdL05]:

**The proactive or ’big bang’ approach:** Following this strategy, PLE is adopted from scratch.

In order to be successful, the transition has to be planned consequently: scoping and domain

Figure 2.11: Development of software reuse concepts from the 1960 to present [37].

2.5.2 Structure of a SoA

The four main artefacts of a SoA are service provider, service consumer, service repository

and the service contract [24] [28] [15]. There are also alternative terms for those, which are

mentioned here; but throughout the remain of this thesis the previously stated terms will be

used.

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, which is then handed over to the service repository in order to make the

service discoverable [28].

24

Service Repository/ Service Registry/ Service Directory. The service repository is, more

or less, a database of services which may be physically distributed. It disposes of certain

publishing mechanisms in order to make services discoverable by the service consumer.

Therefore it contains all the information of every version of the service, as well as meta

information like physical location, provider information, technical constraints, security

issues, and, of course, the link to the registered service [29, p.60-61] [28] [38].

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 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 speci\_c services by means of the service

interface description. Subsequently, the repository returns a list with suitable services.

If an appropriate service is identi\_ed the service consumer creates a dynamic binding

with the service provider in order to invoke the service and interact with it [28] [38].

This procedure is depicted in \_gure 2.13.

Service Contract. The service contract is described in detail in section 2.3.2. It is handed

over to the service repository from the service provider.

Service Inventory. Erl et al. [25, p.41] mention this additional term in connection with SoA.

According to their de\_nition, \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."

The term enterprise in this de\_nition appears a bit bizarre and is a result of the research

area related with this source. For the scope of this thesis the boundary is de\_ned as a

vehicle. Hence, the service inventory includes all services which could be provided by

the vehicle itself. The di\_erence to the service repository is that the service does not

need to be available, or even implemented. Instead, it is a static list of developed or

conceived services.

The relations of the various artefacts are depicted in \_gure 2.12.

25

Service

Consumer

Service Inventory

Service

Repository

Service

Subscribe service, Provider

Exchange messages

Lookup service Publish service

Contract

Figure 2.12: Relation of service provider, service consumer and service repository [24] [38]

.

2.6 Dependability

The terms availability and reliability, which can be found at any level of implementation, are

closely tied, and usually appear together. Often, they are coupled with the term maintenance,

which also plays an important role in the design of any system [11, p.116] [39].

2.6.1 Reliability

Reliability denotes the time an element needs to fail while it is operating. In other words, it

is \the probability of the failure-free operation of a system for a speci\_c period of time in a

speci\_c environment" [11, p.116]. Nelson [40] describes it more technically 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."

In terms of services, the reliability can be enhanced by the ability to recover state information

after an interruption and continue the service like it has never been interrupted [11].

The reliability of higher level elements is of course dictated and limited by the reliability of its

sub elements.

26

Service

Provider

Service

Repository

Service

Consumer

create

Service

publish Service to Repository

(Contract)

lookup Services

return a list of suitable Services

subscribe to Service

(dynamic Binding)

exchange Service Data

Figure 2.13: Chronology of binding a service in a SoA [38].

27

2.6.2 Availability

The de\_nition of availability is ambiguous. The ISO 26262 standard de\_nes 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" [4]. Obermaisser and Kopetz, however, describe it as the \probability of a software

service or system being available when needed" [11, p.116], which is quite similar to the

de\_nition stated in [39] and [40].

The di\_erence here is that the prior denotes availability as the availability of an element at

this very moment, while the latter de\_nes it as a probability of an element to be operational

and ready to use.

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

therefore be expressed as A = MTTF=(MTTF +MTTR) [40].

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 [39]. The application of dedicated fault tolerance

mechanisms (cf. section 2.8) can improve the availability [40].

2.7 Functional Safety

The term safety denotes the absence of an unreasonable risk [4]. Systems which interact with

people, and could endanger those in case of a malfunction, have certain safety requirements

in addition to functional requirements. Those safety requirements are issued by dedicated

standards like the ISO 26262 standard. In accordance with that, the ISO 26262 standard

describes functional safety as the \absence of unreasonable risk due to hazards caused by

malfunctioning behaviour of E/E systems" [4]. Risk, in turn, is de\_ned as the product of the

probability of occurrence and the severity of harm [4].

In contrast to security, 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.

28

2.7.1 Disambiguation Safety and Security

Two terms which often appear together and appear very much alike are safety and security.

Nevertheless, they have a completely di\_erent meaning in technical systems.

The term safety denotes the \absence of an unreasonable risk" [12] [4] and is concerned

with correct operation of the equipment in a speci\_c system and environment. Therefore, it

is concerned with things like error detection, fault prevention, failure migration, diagnosis and

similar.

Security is concerned with preventing unauthorized access or unexpected input. AUTOSAR

de\_nes the term as \Protection of data, software entities or resources from accidental or malicious

acts" [12]. With the advance of software and interconnection in vehicles, security

becomes more and more of an issue because all the functionality in a car, even highly critical

ones like an automatic brake system or a collision warning system, will be approachable

through a network. Especially if the vehicle gets connected to any proprietary and publicly

accessible network like the Internet. With the advance of the SoA paradigm for automotive,

it is planed to run remote diagnosis and \_rmware updates wireless. This is an obvious weak

point and an exploit of this can a\_ect the operation of the vehicle and the safety of the driver

[41].

Although safety and security are separate disciplines, the functionalities of a vehicle cannot

be classi\_ed into safety or security relevant functionality, but always include aspects of both

disciplines at the same time [41].

This thesis is mostly concerned with the safety-oriented point of view.

2.7.2 Safety Related Terminology

Fault, error and failure are terms which are often mixed up and used in the same context, since

they seem to be very similar in their meaning. Nevertheless, their disambiguation is crucial

for some of the following sections.

Fault. In ISO 26262 standard a fault is listed as an \abnormal condition that can cause an

element or an item to fail" [4] [12].

Faults come in di\_erent forms. An intermittent fault occurs from time to time and

disappears without any repair measures having 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

29

some electromagnetic interference. Permanent faults naturally remain until the faulty

component is exchanged or repaired [4].

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 a\_ecting the system [42].

Error. An error is the deviation of a computed, observed or measured value or condition

to the expected, theoretically correct value. An error occurs as consequence of an

unexpected operating condition or a fault. Nevertheless, a fault does not necessarily

lead to an error [4] [40] [12]. If the error exceeds a certain threshold it can cause a

failure [12].

There are two di\_erent classi\_cations of errors, as they can be divided either into soft-

and hard errors or data- and control ow errors [43] [42].

Soft Error. Soft errors occur when a temporary extra charge is induced into the electric

circuit. Usually this is due to cosmic radiation and causes an ip of the data state

in a memory element [43] [42].

This kind of error will become even more severe in the future, due to the decreasing

size of electronic circuits as well as their increasing complexity [42].

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 [43].

Data Error. A data error denotes a change of the data stored any memory location or

register [42].

Control Flow Error. A control ow error leads to wrong execution sequence of instructions.

This is the case if the memory storing the address of the next execution

instruction is changed [42].

Failure. Failure denotes that an element is not able to provide its functionality any longer.

This happens usually due to errors in the element or the environment, which are in turn

caused by various faults [4] [40] [12].

The averaged frequency of occurrence of a failure is denoted failure rate. It is counted

in hours of operation until a serious fault, e.g. 10e5 to 10e9 hours of operation [15].

30

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 [44], or the manner by which an occurred fault is observed [45].

2.8 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 [40].

Thus, the fault tolerance mechanisms are not responsible for preventing the occurrence

of faults, but assure that a fault does not directly lead to the violation of the safety goals

which maintain the system in a safe state [4]. Possible faults 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. Fault tolerance mechanisms have become indispensable for

advanced electric systems because due to the billions of transistors present and the various

circumstances which can lead to faults, faultless systems remain an utopian

dream [11].

2.8.1 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.

31

Chapter 3

Results

This chapter deepens and extends some of the terms given in chapter 2 and investigates them

with respect to automotive. The chapter is structured into \_ve main section, entitled SoA in

embedded systems (Embedded SoA), SoA in automotive, safety services, service development

process and use case scenario.

The \_rst section extensively deals with the application of the service oriented design

paradigm in embedded systems and highlights the challenges this approach has to face when it

has to face certain safety requirements. Moreover, the di\_erences to \conventional SoAs" as

web application is investigated in detail. In the second section various examples of services

relevant for functional safety are described in detail, and their intended functionality, as well

as possible implementations in terms of architecture is presented. The \_nal part consists of

a simpli\_ed 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 \_ndings and concepts in this chapter are (not yet) covered in literature, but

are the result of numerous meetings and discussions at VIRTUAL VEHICLE Research

Center during April to July 2015.

3.1 SoA in Embedded Systems (Embedded SoA)

The service oriented design paradigm was originally designed for the application on the Internet,

which o\_ered ideal prerequisites for this kind of architectural style. An underlying network

for interconnection already existed and time constraints are no concern, asdelays are

unlikely to cause a disaster. Thus, it is no surprise that web services are the application area,

32

where SoA has scored the highest market penetration [15] [3].

3.1.1 Drawbacks in Embedded Systems

In contrast to web services, ESs consist of numerous interconnected nodes with diverse

measurement-, steering-, or computation capabilities. Accordingly, they have to face additional

challenges like limited resources, di\_erent complexity of hardware, time constraints and

others [1] [2]:

Limited resources. One obvious major drawback of ESs are the quite limited resources

which are designed for highly specialized purposes and lack computation power as well

as storage capacities [15] [1] [2].

Di\_erent levels of complexity. The complexity of hardware in ESs varies greatly. The implemented

components may include very primitive sensors with few capabilities, but also

very advanced nodes like MPSoCs [1] [2]. In other words, there are high level informa-

tion systems services, as well as low level generic embedded system services. A SoA has

to deal with the connection and integration of them [15]. This task gets aggravated if

SoAs in ESs become interconnected with high level SoAs like web applications.

Event- and data-driven. In contrast to web services, an ES disposes of a network with

(many) sensors. Thus, the ad-hoc request-response message pattern, which is common

for web services, cannot be simple adopted for the event- and data driven ES [2].

Instead, the communication in those is conducted mainly by a \_re 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 [2].

Lifespan of services. Another di\_erence to web services is the lifespan of services. While

web services are used to work only a limited number of hours (or even just minutes),

the services in ESs could have application times of multiple years, or may even last for

the lifespan of the system [3].

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" [2]. This could become

33

an issue, as ESs where the set of implemented components is usually determined at

assembly time.

Time constraints. ESs 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 [15]. Nevertheless, Sommer et al. mention various

bene\_ts which come with the application of the SoA design philosophy in ESs:

\_ Decoupling con\_guration from environment,

\_ Improvement of reusability,

\_ Improvement of maintainability,

\_ Higher level of abstraction,

\_ Enhanced interoperability and

\_ More interactive interfaces between devices and information system [3].

To sum up, the SoA paradigm has to overcome many serious drawbacks if applied in

ESs, but at the same time it o\_ers numerous advantages and possibilities, which are just not

possible with current systems.

3.1.2 Embedded SoA

There is no uniform denomination for SoAs in ESs throughout literature. Thus, it is referred

to by the term Embedded SoA (abbreviated as ESoA) for the scope of this thesis.

The following de\_nition is the result of intense investigation in this area and extends the

de\_nition from section 2.5:

The Embedded SoA works at a lower, very hardware oriented level, with a

prede\_ned and unchanging set of component.

The conventional SoA is located above the embedded SoA and connected

through various interfaces.

34

With respect to the example of a vehicle, the ESoA operates on elements like the power

train, various devices (sensors, actuators, controllers, etc.), while the SoA level contains the

vehicle as overall system. With other vehicles and environmental components it may form a

SoS.

Regarding the ISO 26262 standard, the requirements for ESoA can be related to Part 3:

Concept phase, and the SoA to Part 4: Product development at the system level as

depicted in \_gure 3.1.

Embedded SoA

*Implementation*

SoA

*Abstraction*

System, System of Systems,

Environment, Cloud, …

*ISO 26262 Part 3*

Infotainment, Future AUTOSAR, ...

*ISO 26262 Part 4*

Figure 3.1: Relation of conventional SoA to Embedded SoA.

3.2 SoA in Automotive

In the future of automotive all vehicles will, most likely, be interconnected with each other

and also with the environment. This is a necessary prerequisite for acting autonomously,

e.g. detecting whether the tra\_c lights at a crossing are green. Within this document those

futuristic vehicles are referred to as connected cars.

The opportunities and advantages of the SoA approach are described extensively in 3.1 and

2.5. 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. Although it is unused for the majority of time, it

comes with a lot of extra weight, which is a major violation of the guideline stated in [16, p.7],

\Embedded systems should exploit the available hardware architecture as much as possible."

The SoA approach might not only reduce the weight, but thereby also the complexity and

35

costs of the overall vehicle. In today's automotive systems it is common practice that each

vendor uses his own proprietary network and additional hardware, prohibiting the application

of hardware components from another vendor [2].

However, there are some obstacles which prevent the application of the SoA paradigm in

vehicles right now. On the one hand, those are the strict regulations in connection with safety

critical real-time systems like vehicles [46]- with respect to automotive there are not even

regulations, addressing SoA, yet - on the other hand, there are pure technical constraints.

The AUTOSAR architecture, which is widely applied today, is not constructed for dynamic

recon\_guration or binding of services at runtime, but everything has to be speci\_ed at building

time. Nevertheless, there is already an approach by AUTOSAR, denoted Future AUTOSAR,

dealing with this very issue. Unfortunately, the actual implementation in mass produced

vehicles is quite some time ahead / lies in the distant future. Ist seiner Zeit voraus oder liegt in ferner Zukunft?

To sum up, it might take some time before the design paradigm will be used for the lower,

hardware-oriented layers of the ESoA. Instead, it would be a promising approach for higher

layers like the SoS. The vehicle as system could o\_er certain services to its environment and

the other way round. A good example, which has been mentioned before, are tra\_c lights.

If the the tra\_c lights in Germany use another service as the ones in Austria, a SoA could

enable the dynamical binding of the new service, when a vehicle approaches the border.

3.2.1 Location of the Service Repository

The description of the term service repository is covered in 2.5.2. Nevertheless, questions

arose concerning its location and actual implementation with respect to automotive. This

section covers the \_ndings concerning this matter.

For SoAs in automotive it is very likely that there will emerge various, physically and

logically distributed repositories. One repository inside of every vehicle, containing all the

services provided by the vehicle itself and used by the vehicle itself. This repository obviously

has to be located inside the vehicle, for it has to be also available when the vehicle is operating

in urban regions or when it is just not able to connect to its environment. These repositories

may be referred to as local service repositories. Examples for services in this repository are

services which are closely connected to hardware components like sensor measurements or

communication services, but also safety services for fault- and error detection, which need to

be available whenever the vehicle is operated.

The counterpart to the local service repository is denoted external service repository. This

36

could also be physically distributed on various server clusters and should hold mostly services

which are needed for interacting with the environment. To connected cars it could provide

all the services needed for managing the tra\_c. In order to stick to the previous example, a

service at this repository could be dedicated to managing the tra\_c lights of a particular city

or zone. If it is bound by a vehicle operating in this zone, it should then be 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.

3.2.2 Service Contract

The service contract (cf. section 2.3.2) is the complete and extensive description of the service

and with respect to automotive it should also contain documentation from AUTOSAR, the

ISO 26262 standard and documentation regarding functional safety. 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 \_lled more and more as the development advances. A template which

should be used for the development of services in the automotive industry is provided by the

Arrowhead project.

3.3 Safety Services

For most safety-critical ESs security is also an important issue. Accordingly,most of the

provided services must be fault tolerant and secure at the same time.

Functional safety is directly related to availability (cf. section 2.6.2) and thus the overall

safety can be increased by increasing the availability [47]. The availability, in turn, can be

increased by services for failure detection, error detection and error masking, which also enable

recovery from errors.

In the following sections, the interference and connection of safety and security services

is analysed, before the functional principles of a few selected safety services is presented in

detail.

37

3.3.1 Relation of Safety and Security Services

In vehicles it is not really possible to distinguish safety functions from \normal" functions,

because a malfunction in any functionality could lead to a disaster [48]. Accordingly, all

services are also considered to be 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.

In terms of security, there will be dedicated services which take are of authentication,

permission management and comparable functionalities.

Most of those safety- and security services cannot be assigned entirely to one discipline,

but have overlapping responsibilities, as depicted in \_gure 3.2. According to various \_ndings,

this e\_ects especially the services for authentication, memory protection and failure detection.

Safety Services Security Services

Fault

Detection

Authorisation

Data

Encryption

Intrusion

Detection

Requirements

Validation

Vulnerability

Analysis

Error

Detection

Memory

Protection

Figure 3.2: Classi\_cation of various services into safety and security services.

3.3.2 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 ow errors. Control ow 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 speci\_ed threshold time. If this limit

is reached, it changes its state in order to trigger further actions, like restarting a component

or activating another safety service [7]. The advantage of the FDS is the simple design, which

38

reduces the additional complexity of the overall system, as well as the costs. Concerning

the functional principle, there exist di\_erent designs with increasing complexity, which can

provide, for example, a certain time window for the response [7].

24 June 2015

Service 1

Failure detection service

Service 1.1

Service 1.2

Service 1.3

Failure

Detection service

Service

Repository

Service consumer

e.g. other service

a. publish

b. discover

c. bind

heartbeat

update

Figure 3.3: 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 (WDT).

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 are three di\_erent basic designs: the

Standard Watchdog Timer, the Windowed Watchdog Timer and the Sequenced Watchdog

Timer [42].

Standard Watchdog Timer. With this basic setup, the service mirrored by the WDT, periodically

sends a simple heartbeat signal which resets the timer and indicates that the

service is alive and active. If a prede\_ned amount of time elapsed without an incoming

signal, it is assumed that a fault has occurred and the WDT changes its state [42].

In terms of control ow 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 [42] ,too.

The probability of noticing such an error is higher the closer the threshold time is to

39

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 ow 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 [42]. This is illustrated in \_gure 3.4 with the

time on the x-axis and the Timers labeled T1 and T2. The time window for a valid

reset is thereby the result of T2 􀀀 T1.

In case of an control ow error this signal is a bit ahead or past in time. The error

detection coverage increases with narrowing the time window [42].

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.

T2

T1

Time

Window

time

Figure 3.4: Schematic design of a windowed watchdog timer [42].

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 speci\_ed time window, this parameter is evaluated

and compared to a parameter inside the WDT. If those match the sequence variable in

the WDT, the variable gets incremented and the timer reseted, starting the cycle all

over again [42]. The whole process is illustrated in \_gure 3.5.

The disadvantage of this design is the clearly higher complexity, as the Sequenced

Watchdog Timer must be capable of holding and modifying state information as well

as comparing it to received information.

40

The rest of the program Fetch ClearWDT instructions. The ClearWDT Instrucl=tion equidistant locations within RAM.

nstr con

x egisted safe ~win ~ ~ ~ ~ ~ \*- wL

Figure 3. Microprocessor N 0,

cik\_ Figure 2. SequencedWatchdog Timer

Addrs N 5. EFFECT OF FAULTS ON IMAGE DISPLAYbleot Dat\_\_naddr(14.) dot(O)This method uses a simple 8 bit Stored Program (Vonwevou Neumann architecture) microprocessor attached to it a VGA Wsote\_nal

display circuit. This system iS implemented on an FPGA.

An 8 color BMP format file is processed as follows: 1) The file header is removed Ok5 ckck5

2) The matrix is transposed into a vector 3) The 4 bit vector is then copied to the FPGA RAM

at the desired location. Figure 4. VGA The 3 least significant bits represent the RGB of each pixel. The VGA circuit reads The image used is 256 x 96 pixels. The microprocessor then RGB values for each pixel. executes the following: Hz.

Figure 3.5: Schematic illustration of the operational principle of a Sequenced Watchdog Timer [42].

41

3.3.3 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 (e.g. 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 [49]. In event of an

error, another service can be triggered for restarting the erroneous service or performing other

countermeasures.

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.

3.3.4 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 speci\_c purpose

and an unintended behaviour is rather unlikely. In such cases, the overhead in runtime, when

using memory protection just does no pay o\_ [50].

Modern, large scale system, on the other hand, dispose of numerous third-party components

for the interaction with the user and the environment. At the same time, the overhead

in computing is no longer crucial, due to the fast increase in computing power [50]. 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 other aspects stressing the need for memory protection in ES:

\_ It can serve as fault- and error detection and containment mechanism, preventing a

failure of one service to propagate and infect the whole system [50].

\_ It protects the system from unintended behaviour of the particular services [51].

\_ It is an important aid for the development process and helps at debugging by identifying

42

\illegal" behaviour of erroneous services, resulting in a reduced development time [50]

[51].

As a service, it could be implemented like the Information Assurance core service from the

Arrowhead framework (cf. section 2.1.3), with dedicated services for authentication, granting

privileges and managing authorization information.

3.3.5 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 speci\_cation

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 and ASIL D, or

combining two of the ASIL B services.

A service like this is, surprisingly, mentioned nowhere throughout literature, but according

to various \_ndings this is a crucial mechanism in order to do not violate functional safety

requirements when composing services.

3.4 Service Development Process

In order to be in compliance with given design rules and standards, the development of services

has to follow strict protocols.

The development stages considered in this section are an adoption of the development

stages, provided by Erl et al. in [25, p.116]. In detail, the \_rst four stages are the object of

investigation. The stages are renamed to \_t for the development of a single service instead

of an overall SoA and are denominated:

\_ Service investigation/planning,

\_ Service inventory analysis,

\_ Service oriented analysis and

\_ Service oriented design [25, p.116].

43

3.4.1 Service Investigation/Planning

This 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 at all?

3.4.2 Service Inventory Analysis

During the service inventory analysis the service inventory (cf. section 2.5.2) is searched

for services which are required in order to build up the desired service. This simpli\_es 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.

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 speci\_c

language. According to [25, 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 and it is not unlikely that the templates

for the service contract are used and extended from the very beginning of the development

process.

3.4.3 Service Oriented Analysis

During the service oriented analysis phase the service candidate is reviewed and checked in

terms of naming and normalisation.

Service naming. The naming of the service candidate must be in accordance with other,

existing, services [25, p.206].

44

In terms of automotive, the naming is governed by certain standards, like AUTOSAR,

which proposes naming conventions in their SW-C and System Modelling Guide

[52].

A uni\_ed naming convention is quite helpful when dealing with standardised interfaces

and provides a possibility to include relevant meta data, like the related system or

intended functionality, into the name [53].

Service normalisation. Services within the same service inventory shall not have overlapping

boundaries. In other words, redundant logic should generally be avoided, as it would

violate the concept of a SoA. Accordingly, the services are forced to use existing services

if those o\_er the required functionality [25, p.207].

Service Candidate Review. The \_nal 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 [25, 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 still have a thorough knowledge of the SoA principles. Those people might think

of a quite di\_erent approach for the very same problem and 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 [25, p.210].

3.4.4 Service Oriented Design

Erl et al. [25, p.86] 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."

The aim of the service oriented design phase is to physically establish the service contract

by \_lling out the corresponding templates. It is initiated when su\_cient analysis has been

conducted and ends with a \_nished service contract as output.

45

3.5 Use Case Scenario

As \_nal part of this investigation, a short 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 3.4. 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 the development process of a service may look like.

3.5.1 Service Investigation/Planning

Scope of the service. The service should be capable of detecting an error within a given

ECR and act accordingly. Therefore it needs the signal of all the sensors it should

mirror, as well as clock signal for creating sampling times.

Required capabilities. \_ Detection of 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.

3.5.2 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,

\_ A management service for 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 the redundant service inside the ECR, whose functionality

shall be checked for errors. Of course, they should feature di\_erent implementations on

independent hardware components, so that an error could not emerge in multiple services at

the same time.

46

3.5.3 Service Oriented Analysis

Concerning the naming, the example service should be in accordance with the AUTOSAR

naming conventions [52]. Hence, 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.

3.5.4 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.

3.5.5 Possible Implementation

At the end of the \_rst four phases, which are concerned with the conceptual development, the

service contract is issued. Subsequently the actual implementation of the service is followed 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 will be implemented, e.g. which

programming language or approach to use. The outcome is a certain architecture, which can

be based on any arbitrary technology.

A possible architecture resulting from the service of this use case is a triple modular

redundancy approach (cf. section 3.3.3), as depicted in \_gure 3.6.

47

24 June 2015

Service 1

Error detection service

Service 1.1

Service 1.2

Service 1.3

Service

Repository

Service consumer

e.g. other service

a. publish

b. discover

c. bind

Error detection

service

comparison

restart

update

Figure 3.6: Possible architectural implementation of the example service.

48

Chapter 4

Discussion

Most of the concepts and principles features in chapter 3 are quite theoretical and based

on perceptions which have derived from meetings and discussions with experts from this area.

Nevertheless, the plausibility and validity cannot be guaranteed. On the one hand this is due

to the state of the art of this research area, with few meaningful publications available. On

the other hand, it is due to the non-disclosure regulations of companies which operate in the

functional safety and security area.

However, the document from which most of these concepts originate was created with the

aim to demonstrate possible implementations or concepts to the EMC2 project group, since

the SoA concept is quite new and unfamiliar to most of the project members. Accordingly,

the here presented information states the company's way of looking at things with no claim

to correctness or completeness. The SoA paradigm for ES like vehicles or aircraft is still quite

at the beginning of its development, which is the reason why it is impossible to make any

accurate predictions. Hence, the actual purpose, for which use case and other concepts from

chapter 3 were developed, namely the creation of a uniform knowledge base, are ful\_lled.

49

Chapter 5

Conclusion

In order to optimise the amount and maintainability of code, the concept of software reuse

has been present since the very early days of software development itself. The SoA approach,

which is treated within this thesis, is the state of the art concept in terms of software reuse.

The underlying idea with this principle is that functionalities are implemented by means of so

called services which hide their internal logic and allow connections only through well-de\_ned

interfaces.

Although already widely applied in other domains, the SoA concept has not yet been

applied for ES like vehicles or aircraft. On the one hand, because of pure technical constraints

due to the implemented hardware which is not optimised for the application of the SoA

design paradigm. On the other hand, this is due to the safety-critical aspect of such systems.

The ISO 26262 standard does not issue any requirements concerning services or the like.

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 bene\_t observable. At the same time it

is hard to communicate the advantages of a vehicle with a SoA inside to the end user. The

bene\_ts 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.

However, this architectural paradigm o\_ers a wide range of advantages and bene\_ts and

is perhaps a necessary prerequisite for next generation's vehicles and aircraft. Hence it is no

surprise that a lot of research has been conducted recently with this respect. The Work Package 1

50

(Embedded Systems Architectures) of the EMC2 project is currently in the \_rst year of

a total of three. This indicates that an actual implementation of a SoA in a mass produced

transportation system may still lie quite some time ahead. Nevertheless, the emerging of this

kind of technology seems just a matter of time.

51