UNIVERSITÄT
KOBLENZ · LANDAU
FACHBEREICH INFORMATIK

ist

An adaptive Software and System Architecture for Driver Assistance Systems applied to truck and trailer combinations

Vom Promotionsausschuss des Fachbereichs 4: Informatik an der Universität Koblenz-Landau zur Verleihung des akademischen Grades Doktor der Naturwissenschaften (Dr. rer. nat.) genehmigte

DISSERTATION

vorgelegt von
Dipl.-Ing.(FH) Marco Andreas Wagner
Leonberg - Juni 2015

Datum der Einreichung:	19. November 2014
Datum der wissenschaftlichen Aussprache:	3. Juni 2015
Vorsitzender des Promotionsausschusses:	Prof. Dr. Ralf Lämmel
Vorsitzender der Promotionskommission:	Prof. Dr. Hannes Frey
Erstgutachter:	Prof. Dr. Dieter Zöbel, Fachbereich Informatik, Universität Koblenz-Landau
Zweitgutachter:	Prof. Luís Almeida, PhD, University of Porto
Drittgutachter:	Prof. Dr. Ansgar Meroth, Hochschule Heilbronn

Veröffentlicht als Dissertation der Universität Koblenz-Landau.

Abstract

Traditional Driver Assistance Systems (DAS) like for example Lane Departure Warning Systems or the well known Electronic Stability Program have in common that their system and software architecture is static. This means that neither the number and topology of Electronic Control Units (ECUs) nor the presence and functionality of software modules changes after the vehicles leave the factory. However, some future DAS do face changes at runtime. This is true for example for truck and trailer DAS as their hardware components and software entities are spread over both parts of the combination. These new requirements can't be faced by state-of-the-art approaches of automotive software systems. Instead, a different technique of designing such Distributed Driver Assistance Systems (DDAS) needs to be developed.

The main contribution of this thesis is the development of a novel Software and System Architecture for dynamically changing DAS using the example of driving assistance for truck and trailer. This architecture has to be able to autonomously detect and handle changes within the topology. In order to do so, the system decides which degree of assistance and which types of HMI can be offered every time a trailer is connected or disconnected. Therefore an analysis of the available software and hardware components as well as a determination of possible assistance functionality and a re-configuration of the system take place. Such adaptation can be granted by the principles of Service-oriented Architecture (SOA). In this architectural style all functionality is encapsulated in self-contained units, so called Services. These Services offer the functionality through well-defined interfaces whose behavior is described in contracts. Using these Services, large scale applications can be build and adapted at runtime.

This thesis describes the research conducted in achieving the goals described by introducing Service-oriented Architectures into the automotive domain. SOA deals with the high degree of distribution, the demand for re-usability and the heterogeneity of the needed components. It also applies automatic re-configuration in the event of a system change. Instead of adapting one of the frameworks available to this scenario, the main principles of Service-orientation are picked up and tailored. This leads to the development of the Service-oriented Driver Assistance (SODA) framework, which implements the benefits of Service-orientation while ensuring compatibility and compliance to automotive requirements, best-practices and standards.

Within this thesis several state-of-the-art Service-oriented frameworks are analyzed and compared. Furthermore, the SODA framework as well as all its different aspects regarding the automotive software domain are described in detail. These aspects include a well-defined Reference Model that introduces and relates terms and concepts and defines an architectural blueprint. Furthermore, some of the modules of this blueprint such as the re-configuration module and the Communication Model are presented in full detail. In order to prove the compliance of the framework regarding state-of-the-art automotive software systems, a development process respecting today's best practices in automotive design procedures as well as the integration of SODA into the AUTOSAR standard are discussed. Finally, the SODA framework is used to build a full-scale demonstrator in order to evaluate its performance and efficiency.

Kurzfassung

Klassische Fahrerassistenzsysteme (FAS) wie beispielsweise der Spurassistent oder das weit verbreitete Elektronische Stabilitätsprogramm basieren auf statischen System- und Softwarearchitekturen. Dies bedeutet, dass weder die Anzahl oder Topologie der Steuergeräte noch das Vorhandensein oder die Funktionalität von Softwaremodulen Änderungen zur Laufzeit unterliegen. Es existieren allerdings zukünftige FAS, bei denen solche Veränderungen eintreten können. Hierzu gehören beispielsweise Assistenzsysteme für Fahrzeuge mit Anhänger da deren Steuergeräte und Softwaremodule über beide Teile des Gespanns verteilt sind. Diese neue Herausforderung kann nicht durch Ansätze die zum Stand der Technik gehören bewältigt werden. Stattdessen muss ein neuartiges Verfahren für das Design von solch verteilten Fahrerassistenzsystemen entwickelt werden. Der zentrale wissenschaftliche Beitrag dieser Arbeit liegt in der Entwicklung einer neuartigen Software- und Systemarchitektur für dynamisch veränderliche FAS am Beispiel der Assistenzsysteme für Fahrzeuge mit Anhänger. Diese Architektur muss in der Lage sein, Veränderungen in der Topologie eigenständig zu erkennen und darauf zu reagieren. Hierbei entscheidet das System welcher Grad der Assistenz und welche Nutzerschnittstelle nach dem An- oder Abkoppeln eines Anhängers angeboten werden kann. Hierzu werden neben der verfügbaren Software und Hardware die ausführbaren Assistenzfunktionalitäten analysiert und eine entsprechende Re-Konfiguration durchgeführt. Eine solche Systemanpassung kann vorgenommen werden, indem man auf die Prinzipien der Service-orientierten Architektur zurückgreift. Hierbei wird alle vorhandene Funktionalität in abgeschlossene Einheiten, so genannte Services gegossen. Diese Services stellen ihre Funktionalität über klar definierte Schnittstellen zur Verfügung, deren Verhalten durch so genannte Contracts beschrieben wird. Größere Applikationen werden zur Laufzeit durch den Zusammenschluss von mehreren solcher Services gebildet und adaptiert.

Die Arbeit beschreibt die Forschung die geleistet wurde, um die oben genannten Ziele durch den Einsatz von Service-orientierten Architekturen im automotiven Umfeld zu erreichen. Hierbei wird dem hohen Grad an Verteilung, dem Wunsch nach Wiederverwendbarkeit sowie der Heterogenität der einzelnen Komponenten durch den Einsatz der Prinzipien einer SOA begegnet. Weiterhin führt das Service-orientierte System eine automatische Re-Konfiguration im Falle einer Systemänderung durch. Statt eines der vorhandenen SOA Frameworks an die Verhältnisse im automotiven Umfeld anzupassen werden die einzelnen in SOA enthaltenen Prinzipien auf die Problemstellung angepasst. Hierbei entsteht ein eigenständiges Framework namens "Service-oriented Driver Assistance" (SODA) welches die Vorteile einer SOA mit den Anforderungen, bewährten Methoden und Standards vereint. Im Rahmen dieser Arbeit werden verschiedene SOA Frameworks analysiert und miteinander vergleichen. Außerdem wird das SODA Framework sowie dessen Anpassungen bezüglich automotiver Systeme detailliert beschrieben. Hierzu zählt auch ein Referenzmodell, welches die Begrifflichkeiten und Konzepte einführt und zueinander in Beziehung setzt sowie eine Referenzarchitektur definiert. Einige der Module dieser Referenzarchitektur wie beispielsweise das Re-Konfigurations- und das Kommunikationsmodul werden sehr detailiert in eigenen Kapiteln beschrieben. Um die Kompatibilität des Frameworks sicherzustellen wird die Integration in einen bewährten Entwicklungsprozess sowie in den Architekturstandard AUTOSAR diskutiert. Abschließend wird der Aufbau eines Demonstrators und dessen Evaluation bezüglich der Leistungsfähigkeit und Effizienz des Frameworks beschrieben.

Acknowledgements

The completion of this thesis has been a long and winding journey. It would not have been successful without the help, support or guidance given by numerous people around me.
First of all I would like to thank my doctoral adviser Dieter Zöbel. Without his openness in the first place this "experiment" would have never happened. Furthermore, I would like to thank him for all the support, the countless discussions and the guidance he offered me throughout all the years.
Another person that has to be named here is Ansgar Meroth. I would like to thank him for the trust he put in me. For hiring me right after my diploma thesis back in 2008 and for providing me all the necessary guidance, freedom in our work arrangements and advice in both technical and organizational fields.
I would also like to thank Luís Almeida for reviewing this thesis. I always enjoyed the discussions with him and benefited a lot from his advise and experience.

During my doctorate I was fortunate enough to have two quite different but yet very supportive institutions backing me. I would like to thank the associates of Heilbronn University that supported this thesis during the last six years. Without their help huge parts of the prototyping would have not been possible at all. Especially, I would like to thank Raoul Zöllner for the valuable advice and for providing me all the organizational support. Furthermore I would like to thank some of my closest colleagues in particular Mihai Kocsis, Petre Sora, Georg Wörle, Markus Kempf and Jens Bachstein for the support and numerous discussions that helped to get a clear view of many aspects of this thesis. And finally I don't want to forget the many students that designed and implemented prototypes that finally enabled me to build a full-scale demonstrator. While Heilbronn University was very important regarding industry experience and practical implementation of new ideas, the University of Koblenz-Landau was providing me the scientific environment so important for such a thesis. For that reason, I would like to thank Sabine Hülstrunk, Alberto Ballesteros, Christian Schwarz, Simon Eckert, Benjamin Knopp, Uwe Berg and Christian Weyand for creating this inspiring environment. Furthermore, I would like to thank the Thomas Gessmann-Stiftung for the financial support during writing this thesis.

One thing that became clear to me throughout all these years is that a scientific environment is never enough when it comes to writing a dissertation. Sometimes it even seems that the "life outside the doctorate" is much more important than that. In this sense I would like to thank all my family and friends for the great support throughout the years. Especially, I would like to thank my parents that made this all possible. For all the encouragement and help throughout my whole life.

Last, but certainly not least, I would to thank that one special person in my life, my fiancée Lina. She is probably the one that suffered most from my periods of frustration as well as from all the nights, weekends and days off that I worked on this thesis. I would like to thank her for all the support and the patience that I experienced. Her love carries me through always.

Contents

1 Introduction

'A magic dwells in each beginning'

Hermann Hesse.[1]

1.1 Software and System Architectures in state-of-the-art Driver Assistance Systems

Driver Assistance Systems, often abbreviated DAS, have become an important domain within automotive technology in recent years. These systems assist the operator of a vehicle in various situations. For example, systems like the Adaptive Cruise Control (ACC) control the distance to the car ahead which brings some relieve to the driver especially in long distance traveling. Other DAS like a parking assistance system support the driver in a situation that a lot of people are uncomfortable with.

The rise of DAS is one of many reasons for the increase in electronic devices within a car. In 1990 the share of electronic elements of the added value of an automobile was at about 16%. In 2011 this share was already at about 30% to 40%. Furthermore, experts say that about 80% of the innovations made in premium cars are driven by electronics and computer science (see [149]). The German Association for Electrical, Electronic and Information Technologies (VDE) states that the number of Electronic Control Units (ECU) increased to about 70 devices in a single car ([135]).

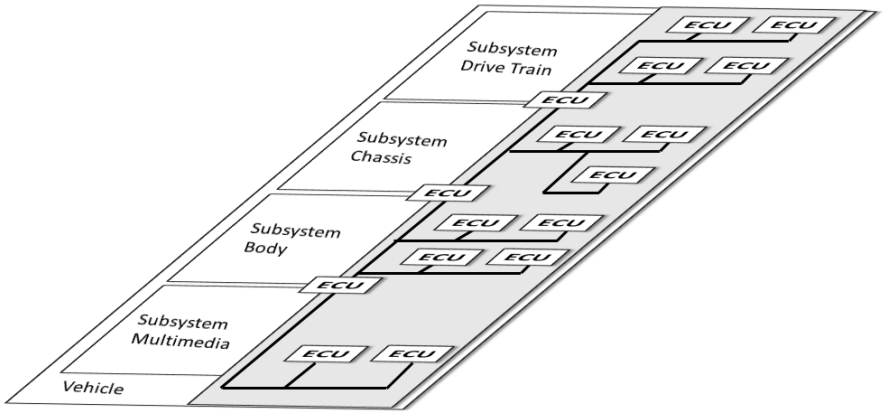


Figure 1.1: The domains of the electronic system of a modern vehicle [117].

With the increasing complexity of the electronic system of a vehicle, developers have started to divide the overall system into several subsystems. Figure 1.1 shows a typical division of the system into four subsystems. The drive train division includes all electronics that are directly interacting with components like engine or gear box. The chassis subsystem influences the driving characteristics by controlling the parameters of for example suspensions. However, this division also includes electronically controlled steering systems and safety relevant assistance systems like the Electronic Stability Program (ESP) or the Anti-lock Braking System (ABS). The body domain

[1] From the poem "Steps", see [68]

includes a lot of comfort functionality. This includes systems like the central locking system, motorized mirror adjusters or the adaptive bend lighting system. Finally, all infotainment and media applications are grouped into the multimedia subsystem. The information exchange between these domains is usually done by using gateway ECUs. These gateways do not only transfer information from one subsystem to another but also between the different communication technologies used.

1.1.1 Today's Software and System Architecture for Driver Assistance Systems

Besides the technical realization of electronic systems as described in section 1.1 today's Driver Assistance Systems can be described using an abstract generic model. This model is presented in Figure 1.2. The model is constituted by five components. The first one is the sensors component. Behind this abstraction several potentially combined sensor-systems determine the state of the vehicle. The quantities measured vary and are tailored to the characteristics needed for the specific DAS. For example, a system helping the driver to keep the vehicle on track (often called Lane Departure Warning System, LDWS) needs to measure two quantities: the course of the road ahead and the current position of the car relatively to this track. In order to do so, most cars use camera systems that observe the road ahead. The pictures created by these cameras are then analyzed by specialized computer vision algorithms that detect the two quantities defined earlier. The generated information is then transferred to the logical core of the assistance system. This part of a DAS is in charge of executing the actual functionality requested by the driver. Coming back to the LDWS example, this piece of software implements an algorithm that tracks the position of the car on the lane and creates a warning signal whenever it is about to leave the track. The warning signal is then transferred to the third part of the generic model, the human machine interface (HMI). This part of an informing assistance system is building the connection between the DAS and the driver. It may interact with the vehicle operator using any of the three major modalities namely the vision, audition or tactition modality. In LDWS implementations the HMI is often based on either the audition or the tactition modality. In the former case acoustic alerts are generated and emitted. In the latter case vibration motors attached to either the driver's seat or the steering wheel inform the driver that he is about to leave the track. Those three components of an informing DAS are influencing two more external components. The first of them is the driver itself. He is stimulated by the HMI of the DAS to conduct proper actions in order to react on the current state of the car. In the LDWS example the acoustic alert may draw the attention of the driver to the fact that he is about to leave the road which allows him to make appropriate course corrections by turning the steering wheel. These actions undertaken by the car operator are directly influencing the vehicle, which is the last component of the generic model. And since the state of the vehicle is observed by the sensor component, this element closes the generic loop of informing DAS.

1.1.2 Automotive network systems

As explained earlier, the numerous ECUs of a car's electronic system are connected using different networking technologies. All of them have been developed in recent years by the automotive industry to fit specific purposes. The first one introduced is the Controller Area Network (CAN). The development of this network is a response to the increase of electronic systems in the 1980s. This increase led to wiring harnesses with growing length, weight and complexity through to the fact that each of these ECUs had to be connected through parallel interfaces. CAN simplified these wiring

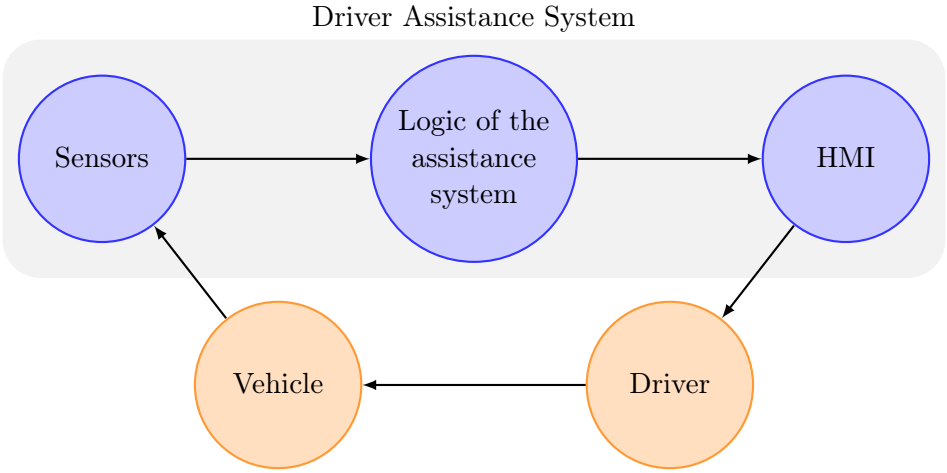


Figure 1.2: The generic loop of informing Driver Assistance Systems (bases on [29])

harnesses by introducing a serial bus system based on simple twisted pair cables. It is a low cost system which provides electric robustness by using a differential signal on the physical layer. It is message-oriented and is restricted to broadcast messaging. The network realizes a prioritized medium access arbitration mechanism based on the identifier of each message. Furthermore, it features error- detection and containment mechanisms. Consequently, although it was originally designed for cars, it soon became widely popular in other domains of distributed embedded control systems, like factory automation, robotics or medical equipment. The CAN specification is restricted to the lower two layers of the ISO/OSI communication reference model. However, several protocols extended this definitions to upper layers like CANopen, the time-triggered network TTCAN or the diagnostics protocol UDS. Despite the fact that the Controller Area Network is almost 30 years old, it is still the most used system in today's cars. It is used in many subsystems like the drive train, chassis and body. To handle the bandwith limitation and to reduce the complexity of network planing, modern vehicles often feature several independent CAN networks that are interconnected using gateways.

While CAN is a very flexible network system it needs specific controllers and transceivers to be mounted on each participating ECU. During the 1990s the automotive industry faced another problem. Besides the fact, that CAN is technically able to be used also for low performance scenarios like the link between motorized mirror adjusters and the switches controlling them, the costs are too high. To face this problem, the Local Interconnect Network (LIN) was developed. It is a master-slave network that uses a simple single-wire connection. Furthermore, it does not need a distinct controller as it can be implemented using the Universal Asynchronous Receiver Transmitter (UART) interface that is integrated in most of the microcomputers on the market. This cheap and simple network system does only provide a bandwith of 20 KBit/s and is restricted to only 16 slave nodes. However, it is the perfect fit for many comfort applications in the body subsystem. Thus it is often used as a sub-network connected to a larger CAN using a gateway approach.

At about the same time when LIN was developed, another automotive network system was brought on its way to the market. While LIN was targeting on small, low-bandwith networks, the Media Oriented Systems Transport (MOST) technology was build to provide enough bandwith to allow audio and video streaming. Targeting the emerging multimedia systems MOST serves a segment where neither CAN nor LIN could be used. In most cases, it is build in a ring topology where it connects units like audio and video players, CD changers or amplifiers. It makes use of a synchronous messaging pattern.

One of the greatest disadvantages in early years was the fact that MOST used plastic optical fibers (POF) to guarantee the bandwith and avoid electromagnetic interference. This circumstance resulted in high prices for this system and in complications in the design of the wiring harnesses through to the rather large bending radii of the POF. This issue was partly eliminated in later versions of MOST which introduced electrical physical layers as well.

Another automotive network communications protocol to be mentioned here is called FlexRay. It supports data rates of up to 10 MBit/s and allows real-time communication. It was initially developed to interconnect time-dependable systems like ESP or ABS. It is highly flexible since the developers can vary the ratio between time-triggered and competition-based messaging as well as the level of fault-tolerance by using the second channel provided either for redundancy or as an independent channel to double the bandwith. Today, it is mostly used as a high-bandwith alternative to CAN.

Currently, another networking technology is on its way to hit the automotive market. Ethernet, that has a long history in office communications, is being adapted to the automotive environment. It is a promising candidate as experts predict cost reduction through to the fact that many techniques and mechanisms can be adopted from other domains of usage. Currently, it is planned to be used as a diagnosis network or to connect video-based assistance systems. In the future it may be used as a central backbone interconnecting the different network systems in the car. For all of these scenarios protocols are being developed or adopted at the moment. For example, it is planned to use Audio Video Bridging (AVB) for connecting video-based systems to one another (see [116]). In order to face the harsh electromagnetic environments in a vehicle, the physical layer used must be much more robust than the one used in office communication. Therefore, BroadR-Reach has been chosen. It uses Quadrature Amplitude Modulation to provide full duplex communications at up to 100MBit/s using single-pair cabling. However, Ethernet is still a future system not yet disseminated widely on the market. Furthermore it is quite questionable if it will completely replace all other automotive network systems in future vehicles.

1.1.3 AUTOSAR

A very important development in the history of automotive software systems was the establishment of the AUTomotive Open System ARchitecture (AUTOSAR). The AUTOSAR consortium which is the organizational backbone of the standard has been founded in 2002 by some major German car manufacturers and suppliers. Over the years, this alliance has attracted many companies of the automotive domain from all over the world leading to a total number of more than 150 members organized in four different groups (see [130]). The latest release of the AUTOSAR specification is version 4.1 and dates from March 2013.

AUTOSAR aims at simplifying and shortening the development of automotive software systems. Therefore, it introduced a layered architecture that supports the exchangeability and re-usability of software components as well as the usage of off-the-shelf modules by specifying dedicated interfaces. It also describes a methodology in order to harmonize the development processes between the manufacturer and its suppliers. The ultimate goal of all AUTOSAR activities is to achieve "[...] a high reliability of the overall system with significant cost and capacity benefits" [130].

Today, the AUTOSAR architecture as well as the associated development procedure are a de-facto standard in the automotive domain. It undergoes continuous development

to integrate upcoming technologies and approaches.

1.1.4 Coordination of development processes

Developing a modern car is a complex and very challenging mission. It is characterized by the high demands set up by a high tech industry and a high cost pressure. Furthermore, cars have to provide a high level of dependability, availability, safety and security (see [117]). These requirements are accompanied by various configurations. For example, the second generation of the Renault Trafic van, built between 2001 and 2014, features about 10^{21} variations ([10]). To face all these requirements the development process is characterized by the divide and conquer principle: the overall vehicle is divided into subsystems and components. This is usually done using the V-model. The divided system is than developed in parallel by different divisions of the car manufacturer as well as its suppliers and sub-suppliers. In the later steps of the development process, the components are integrated into the subsystems which are then integrated into the overall vehicle.

All of these processes are supervised by the car manufacturer (often called OEM). The OEM is responsible for the functioning of the overall system and the seamless integration of all subsystems. In other words, the OEM acts as a central integration instance, being the only participant in the development process that has the full overview of the vehicle.

1.2 Attributes of Distributed Driver Assistance Systems (DDAS)

Most of the Driver Assistance Systems installed onto today's cars can be determined as static in their system and software architecture. This means, that neither the system configuration, by means of the number or topology of the ECUs, nor the appearance or functionality of software modules does change at runtime. As an example, the Lane Departure Warning System described in the former sections of this chapter may consist of a couple of different hardware and software entities. However, all of them are installed and configured at the factory. There is no extra ECU coupled into the system at runtime. Neither is any of the installed hardware units removed while the system runs. The only possible change on the software side is a software update. These updates are typically installed by completely re-flashing an ECU during workshop service. The changes done can be characterized as following:

- The changes to the system are carried out by an especially trained workshop employee.
- The changes to the system are planned, implemented and verified by the manufacturing company.
- The changes to the system are executed in a controlled environment which allows to check and if required to fix the changed system.
- The changes to the system are rather rare.
- The changes to the system are done while the vehicle is parked.

However, some future Driver Assistance Systems may undergo dynamic changes of the system and software architecture at runtime. Examples of such systems are:

- Systems incorporating nomadic devices.

- Systems using specialized equipment that can be added and removed to the vehicle, especially in the domain of commercial vehicles (agricultural, forestry ,etc.).
- Driver Assistance Systems using Car-2-X technology.
- Driver Assistance Systems for truck and trailer combinations.

All of the systems named above have two things in common. First, the functional units are distributed over more than one entity. In the case of Car-2-X systems, this means that the sensors, processing units and HMI components are distributed over several cars or infrastructural units. The second thing these entities have in common is that they are coupled and de-coupled frequently without the support of a specialist supervising these activities. In Car-2-X scenarios this is given through to the movement of the different cars which leads to ever changing ad hoc networks. In order to label this class of systems, Dieter Zöbel, Ansgar Meroth and myself introduced the term Distributed Driver Assistance Systems (DDAS) to refer to them (see for example [140], [144]).

This thesis focuses on the last scenario for DDAS named above: DDAS for truck and trailer systems. While truck and trailer combinations have been on our streets for many many years, the introduction of assistance systems that break the barriers between truck and trailer is still in its infancy. One reason therefore might be, that many of these combinations belong to the domain of commercial vehicles and are driven by professional drivers. These drivers compensate the absence of assistance systems with a high familiarity of the system and years of experience. On the other hand there are many non-commercial usage scenarios like small car trailers or caravans. These scenarios are especially interesting as the drivers of such combinations might use them only once a year or sometimes even less often. Another reason for the absence of DAS distributed over the whole combination is technically in nature. While in the car domain the OEM acts as a central integrator, there is no counterpart within the domain of truck and trailer combinations. Instead, there might be several independent manufacturers engineering different parts of the combination. Besides the OEM of the towing vehicle and the trailer manufacturer, there might be several other companies responsible, for example the superstructure manufacturer for the trailer. The absence of a central integrator that sets mandatory standards and defines the functional attributes of the overall combination is another obstacle in introducing such overlapping assistance systems.

At the same time, DAS for truck and trailer combinations fit well in the definition of DDAS. The different functional blocks might be located on either the truck or the trailer. In future application scenarios this may even be extended by the usage of nomadic devices such as smartphones or handhelds or by integrating web-based services located on a remote server cluster. Furthermore, the system is subject to changes at runtime as it is very likely that different trailers are attached to a towing vehicle over time. Chapter 6 of this thesis will introduce the events of system changes in more detail.

1.3 Service-oriented Computing

The attributes of Distributed Driver Assistance Systems call for new thinking models in the automotive industry. The traditional approach on how a system is built and how the development procedure is organized can't be adapted to this new class of systems that easily. Instead, new approaches should be considered. However, this may be new to the automotive domain, but other domains have already faced such challenges some

time ago. These domains came up with different approaches that solve these issues. One of the most interesting and popular ones is Service-oriented Computing (SOC), often called Service-oriented Architecture (SOA).

The term of Service-oriented Architecture is not clearly defined. One of the most popular definitions has been published by the World Wide Web Consortium (W3C) in 2004: For this institution SOA is "a set of components which can be invoked, and whose interface descriptions can be published and discovered" (see [64]). For this thesis the definition is extended:

"Service-oriented Architecture (SOA) is a set of loosely coupled, distributed components which can be invoked, and whose descriptions of the well-defined interfaces encapsulating its functionality can be published and discovered."

This definition covers the main aspects of Service-oriented Computing:

- A SOA-based application is not a monolithic block, but a set of loosely coupled components, so called Services.
- These Services are distributed over a network or a network of networks and have the ability to invoke each other using their interfaces.
- These interfaces are encapsulating the functionality of each Service and allow a standardized access. It also ensures accessibility by providing public information on how to access the Service and by ensuring discoverability of the Service within the network.

The characteristics of the Service-oriented paradigm, especially the fact that it is based on functionality that is encapsulated in loosely coupled and discoverable Services, is a promising approach to handle distributed applications that change at runtime. This is because the elements of an application are self-contained. The overall application is a combination of the elements currently available. In the case of a system change, this combination can be altered by re-composing the application using the currently available entities.

1.3.1 Areas of application

As explained earlier, Service-oriented Architecture is not a new paradigm. It has been used in several areas of application. One of them, which became rather popular in recent years, is the Android OS. In this Linux-based smart phone operating system the so called contents are an implementation of the Service-oriented principles (see [39]). Besides such popular but rather narrow fields of use SOA has been initially developed and mainly used for two distinct areas of application: Enterprise software and Web Services. The former one is a collective term for software systems that are used in the business domain. Within this area, the growing complexity and degree of distribution requested a new paradigm of software development. Furthermore, such enterprise software faces frequent changes in its structure to react to software modifications, platform updates or hardware replacements. As in many enterprises these systems are the backbone of their business activities, traditional monolithic software blocks were replaced by more flexible, and easy-to-service Service-oriented applications. Many big software companies like Oracle or IBM have developed Service-oriented frameworks that support developers in designing such enterprise systems.

Another very popular field of usage of the Service-oriented paradigm are Web Services. Web Services are Services that run on servers connected to the internet. These Services

can be discovered and invoked by human users as well as software applications. One example for such a Web Service is Google's Elevation Service. This Service provides elevation data for numerous locations on the surface of the earth (refer to [60] for more details). The main principle is, that the user sends his request including the location of interest to the Web Service running on a Google server. The Web Service processes this request and answers it with a response containing the respective elevation above sea level. It is a public, in its basic feature set, free Service. However, many Web Services are either restricted to a specific group of users or do charge the requester a specific amount of money.

The main difference between SOA-based enterprise software and Web Services regards the standardization. While most enterprise solutions are either not complying to any standard or are based on the solution of a software company, the openness of Web Services requests the definition of a common standard. This standard is called Web Services Architecture and has been published by the W3C (see [26]). It is actually a set of standards defining different aspects like messaging and Service Interfaces or Discovery. Furthermore, this specification contains many popular protocols and description languages like the Extensible Markup Language (XML), the Simple Object Access Protocol (SOAP) or the Web Services Description Language (WSDL). Due to its popularity many commercial but also open source frameworks implementing it (or at least parts of it) are available.

However, Web Services are not the silver bullet when building Service-oriented systems. Especially when looking at the embedded domain, Web Services and SOA in general are not having significant market shares. This is mainly due to the large overhead produced by this technology. Besides the fact that many of the implementations of Web Services base on additional middlewares like Java, the messaging and memory footprint are often not compatible to really small embedded devices. In the automotive domain, one of the most challenging domains were embedded systems are used, there is only one SOA-like technology available right now (SOME/IP, please refer to chapter 2 for more details). This thesis will fill this gap by introducing a framework capable of offering the benefits of Service-oriented Computing in an implementation respecting the resources of small embedded devices.

1.4 Objectives and contributions of this work

As described earlier, this thesis will present the research done in the field of Service-oriented Computing for Distributed Driver Assistance Systems on the example of assistance system for truck and trailer combinations. In the following, the main objectives will be presented and the contributions of this thesis will be outlined.

1.4.1 Main objectives

There are three main objectives for this thesis:

- Manage Distributed Driver Assistance Systems on the architectural level.
- Introduce a Service-oriented framework for automotive embedded systems.
- Introduce a suitable development process.

The first objective targets at using the beneficial characteristics of Service-oriented Computing onto DDAS. Hereby, the functionality is to be encapsulated into small,

loosely coupled, self-contained and distributed Services. The heterogeneity of functionality involved in a typical Driver Assistance System and the absence of a central integrator is to be faced by the introduction of well-defined interfaces accompanying the Services. The requirement to be able to react to runtime changes of the system is to be fulfilled by the Service-oriented principles of Discovery and Re-Composition. Furthermore, the demand for the protection of the know how of the implementer of a software functionality is to be faced by the principle of information hiding behind the standardized Service Interfaces.

The second objective targets on introducing Service-oriented Computing into automotive embedded systems. This includes the implementation of Service entities that respect the resource constraints especially regarding the memory footprint and the available computing power. It also requests the implementation of a Communication Model capable of carrying out Service communication using the specific automotive network systems introduced earlier. Additionally, this objective calls for the development and implementation of a re-configuration algorithm capable of reacting to changes of the system at runtime, again while respecting the special conditions of the automotive domain.

As a last objective, a development procedure is to be instantiated that allows to create Service-oriented applications and can be easily integrated into automotive design processes at the same time. In combination, the latter two objectives call for the integration of Service-oriented Computing into widely used automotive standards.

1.4.2 Contributions to fulfill the objectives

All the objectives introduced in the last section are fulfilled by the development and implementation of a Service-oriented framework for automotive applications. This framework is called SODA - Service-oriented driving assistance. It builds the architectural basis by offering a layered software architecture that can be used to implement automotive Services. This architectural blueprint is described in section 4.2.4. SODA also includes a Communication Model that allows to carry out all needed communication between the Services on networks currently used within the automotive domain. Chapter 7 introduces this module of SODA in full detail. The re-configuration algorithm developed for the SODA framework is described in chapter 6. It makes also use of the Quality Model described in section 4.3 of this thesis. Another important contribution of my research within this area is the definition and implementation of a development procedure designed to create applications on the basis of SODA. Chapter 5 of this thesis will show that this development procedure is a powerful tool to guide developers that might have no experience in Service-orientation at all through the design process and can be integrated easily into today's most used development scheme in the automotive domain. The integratability of the SODA framework into automotive standards is proved on the example of the popular AUTOSAR architecture. Chapter 8 of this work explains three different ways of integrating SODA into AUTOSAR. Finally, the SODA framework is evaluated regarding its usability for developing DDAS for truck and trailer combinations. Chapter 9 examines a prototypical implementation of a backing up assistance for a car towing a two-axle trailer.

1.5 Research Methodology: Design Research

The research described within this work has been carried out following the research methodology Design Research. This is an approach that carries out a well-defined

design process aiming on gaining knowledge about the field of study. The approach dates back to a proposal made by March and Smith in 1995 in their publication [89]. Here, they tried to meld research activities like creating and evaluating theories with steps in a design process like modeling or constructing into a combined research framework. The idea has been picked up and developed further by Vaishnavi and Kuechler. In 2004 the authors published a book describing this research methodology in more detail [136]. The descriptions on the Design Research methodology within this section are all extracted from this source. In this book the general cycle of Design Research as it is presented in Figure 1.3 is defined. This cycle consists of five process steps that are executed throughout the procedure. The first one of these steps is called "Awareness of Problem". It is the initiating step that unveils the need for research in a specific area. Often this discovery is caused by conducting market research or by detecting a gap within state-of-the-art technology. Within the Design Research methodology this problem is written down and transformed into a research question or objective.

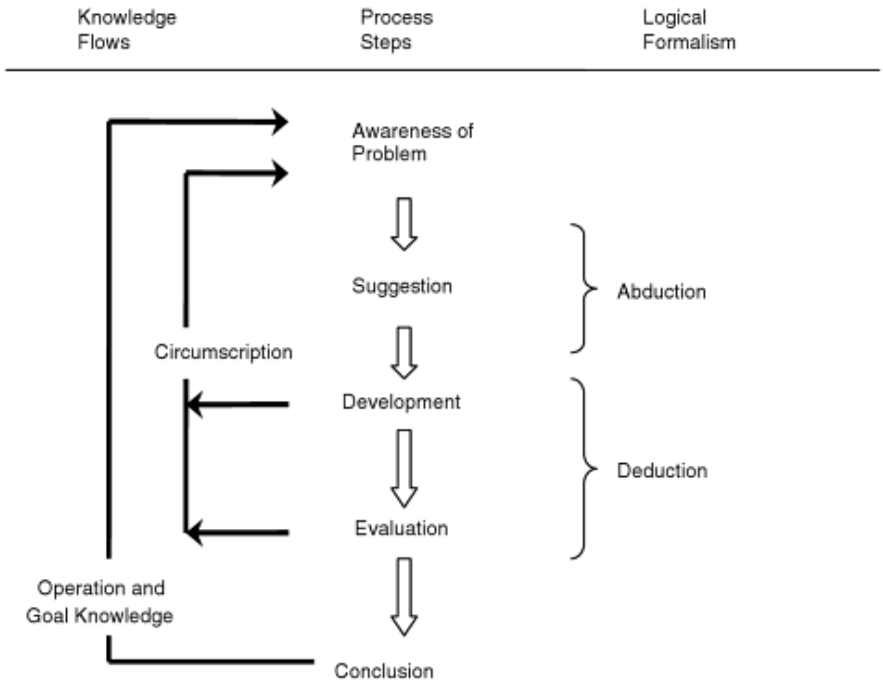


Figure 1.3: The general cycle of the Design Research methodology [136].

Within the second process step called "Suggestion" potential problem solutions are drawn from the existing knowledge. In other words, a technology, principle or theory is suggested that is capable of filling the gap detected in the first step of the cycle. This step can be equated with the logical formalism of abduction which transfers an observation into a hypothesis. The idea created here is refined in the "Development" step were it is implemented at least in a prototypical way. In the case that an idea created in the "Suggestion" phase can't be implemented, the cycle is started again, indicated by the arrow named "Circumscription". Circumscription is a very important process within the methodology as it alters the initial problem definition based on information that has not been available before the Development step has been started. In this sense, it gains scientific knowledge that has not been detected before. In case the Development was successfully completed, the methodology moves on to the "Evaluation" process. It analyses the implementation and tests whether the initial problem has been solved. Again, if this is not the case, the cycle turns back to the initial step altering the problem definition by the findings made through the evaluation. This step again may create new knowledge even in the case the tested implementation has failed. The two steps

of "Development" and "Evaluation" correspond to the logical formalism of deduction as it creates a conclusion by reasoning. In case the Evaluation was at least partially successful the cycle is completed by the "Conclusion" step. This step may terminate the specific design project by writing down the knowledge gained. However, it may also make use of the newly created operation and goal knowledge to jump back to the initial step and alter the problem definition according to the experiences made. For example, the analysis of an implementation of a suggestion made to solve a specific problem may have shown that it does not fulfill the demands of the initial problem scenario but may be successful if the research question would be slightly changed into another direction. In this case the suggested idea is not a solution for the problem but the research conducted has gained knowledge that may help to close related gaps.

The research methodology Design Research has been used throughout this whole thesis to gain scientific knowledge. It fulfills the demand for creating scientific progress while being close to the technological character of this field of research.

2 State-of-the-art in Service-oriented Architectures for embedded systems

'People will not look forward to posterity, who never look backward to their ancestors.'

Edmund Burke[1]

2.1 Requirements on Service-oriented Architectures for DDAS

This chapter discusses several Service-oriented approaches for embedded and automotive systems that have been published in recent years. As described earlier, Service-oriented Architecture is a promising design principle when developing distributed systems. It is capable of handling changes of the software and system architecture of an application at runtime. Through to the clear separation of different modules in the form of Services and the structured description of their relationships by introducing Services Interfaces, it also supports maintainability and software re-use and simplifies the development procedure within big development groups.

However, most of the SOA frameworks proposed in recent years are targeting at domains like enterprise software or Web Services. These domains feature specific characteristics that are quite different from distributed embedded systems as they are common in the automotive domain. These differences regard for example to the computing platforms used to deploy them. In the enterprise software and Web Services domain full scale personal computers and servers are used that provide Gigabytes or sometimes Terabytes of memory, several Gigabytes of RAM and CPUs that often feature several cores which run with speeds of several Gigahertz. Even when smaller units like tablet computers or smartphones are involved the computational power exceeds the performance of automotive ECUs by orders of magnitude. Such automotive computing platforms are often restricted to only a few Kilobytes of ROM and RAM as well as single core CPUs running at a speed within the low Megahertz range in order to reduce costs and electromagnetic influences. These restrictions lead to the fact that well-known technologies like for example Java did never hit this market section. Besides the gap in available resources the networks used in both domains vary, too. While within the internet and in office networking IP-based protocols dominate the market, the automotive sector developed several specialized communication technologies to meet different requirements in different domains of application. For example, small sub-networks containing solely functionality that is not safety relevant, are often based on the simple and cheap Local Interconnect Network (LIN). When ECUs that compute time-relevant and safety critical data are connected, FlexRay is often the network of choice. Within the infotainment domain, where bandwith and packet sizes are relevant parameters, a specialized network named Media Oriented Systems Transport (MOST) is used to connect video players, audio amplifiers and other equipment. The oldest, but still most used automotive bus system, is the Controller Area Network (CAN). It has become quite popular since it first hit the market in 1987, and is now used in many other technological

[1] 'Reflections on the Revolution in France, see [30]

fields like medical equipment, factory automation or elevators, too. In comparison to IP-based networks, the automotive communication technologies with the exception of MOST, lack of several fundamental properties such as plug-and-play characteristics. Furthermore, they differ in basic concepts like the the usage of message-based instead of node-based addressing.

All these circumstances lead to the fact that Service-oriented Computing in embedded systems and especially in automotive embedded systems, has to be revised and tailored to the characteristics of this domain. This fact has been realized by several researchers and practitioners and led to the publication of a number of embedded SOA frameworks in recent years. In order to be able to evaluate these approaches, the specific needs and requirements for the problem scenario of Distributed Driver Assistance Systems are defined:

1. Allow automatic re-configuration of internal components at runtime.
2. Respect the resource limitations of the embedded systems and the specific characteristics of the networks used.
3. Organize the re-configuration in a distributed fashion to avoid a single point of failure.

The first requirement states that the framework should be able to re-configure an application at runtime. This re-configuration should be done automatically. Both needs are aiming at reacting on system changes while the system is running. Furthermore this requirement states that the re-configuration should imply all entities no matter if they are external ones that are brought into the system by the user or a third party or internal ones that are an integral part of the vehicle or its attachments. This is especially important as the discussion in section 2.2 will show that some of the proposed frameworks are following a gateway approach where Service-orientation is only used to manage external components while internal components are developed in the traditional way and accessed through a gateway unit.

The second requirement requests the frameworks to respect the specific characteristics of automotive embedded systems. As discussed earlier, these include resource limitation as well as the usage of specialized broadcast networks. The approaches analyzed in section 2.2 will be evaluated on the basis of the fundamental communication schemes and underlying technologies. For example, approaches that make use of Java-based technologies won't be able to be deployed on some of the small ECUs used within the car. By using such a virtualization method the approach would restrict itself to areas such as the infotainment sector where the computational platforms tend to be more powerful.

The third and last requirement calls for a distributed approach to manage the re-configuration procedure. This is due to the nature of Distributed Driver Assistance Systems. As this specific kind of DAS does not have a single integrator during the development process, nor a clear hierarchy in ad hoc scenarios, introducing a central entity would cause numerous problems. Furthermore, this would be contradictory to the fact that most of the automotive networks are organized decentralized.

The remainder of this chapter is organized as follows. Section 2.2 will discus a number of Service-oriented approaches targeting on embedded systems using the three requirements introduced. The results of this discussion will be summarized and analyzed within section 2.3.

2.2 Discussion of Service-oriented approaches in embedded systems

Using the requirements set up in the former section, 23 different approaches that target on bringing Service-oriented Architecture into embedded systems have been analyzed. Table 2.1 summarizes these proposals. The discussion will show that none of these is capable of meeting all requirements requested by Distributed Driver Assistance Systems.

The first group of approaches lacks of the ability to do runtime re-configuration. One representative of this group has been presented by Ermagan and her colleagues from the University of California and the Technical University of Munich in [47]. It targets to the automotive domain as it focuses on a use case that describes the development of a central locking system. However, the used technologies like for example the Common Object Request Broker Architecture (CORBA) cause a huge overhead that makes it difficult to bring this approach to small automotive ECUs. Another approach that tries to introduce Service-orientation into the automotive domain is presented by Shokry et al. in [122]. It focuses on the re-use benefit of SOA and suggests to implement all vehicle functionality as Services to simplify product line management. However, this proposal does not consider changes at runtime and again makes use of CORBA technologies which create a significant overhead. A third approach in this group not capable of reacting to runtime changes has been presented by Eichhorn et al. in [45]. It targets in the infotainment system within a vehicle and describes how several components could be connected using the Devices Profile for Web Services (DPWS) a subset of the popular Web Services specification. Besides the fact of not being capable of runtime re-configuration, the DPWS approach requires the engineer to use IP-based networks to connect the entities. Since these networks are still in their infancy within the automotive domain, this approach is not very promising. The same dependence from networks that are not common in cars can also be discovered in two other approaches listed in Table 2.1. The first one has been described by Lopez and his colleagues in [86]. It describes the integration of Service-oriented principles into unmanned aerial vehicles (UAV). It makes use of a specially developed communication model while based on the TCP/IP protocol family. The other proposal to be named here is the eSOA project (see [121]). It makes use of technologies from the enterprise software and Web Services domain like the Business Process Execution Language (BPEL) and the Web Services Description Language (WSDL). Besides, it uses ZigBee communication to allow interaction between different Services. Since targeting on the building automation sector where systems are installed and configured by professionals it does not consider automatic configuration. While some of the proposals of this group provide some excellent solutions for particular problems of the problem scenario, none of them is close to being a potential approach for DDAS.

The second group of proposals to be discussed introduces a gateway to connect embedded devices to the SOA world. One of these approaches is called DOMINION and has been described by Gacnik and Haeger in [53]. It follows the idea of integrating external software functionality into a vehicle. The case study used describes a travel assistance system that makes use of Web Services to request train information and other Services to improve the travel experience of the driver. The system architecture features a gateway device that collects data from the CAN bus of the car as well as external information using Web Services. However, there is no attempt to introduce Service-orientation in the in-vehicle domain. In another proposal named CODAR Röckl et al. use the DOMINION framework to built cooperative sensor networks for future driver assistance systems. In using DOMINION, the CODAR approach suffers from the same disadvantage as the internal components are not implemented as Services.

A similar approach has been presented by Foster and his colleagues in [51]. Within the DINO project self-x-properties for Service-oriented applications were examined. One of the case studies focused on a coordinated route planning application that uses Service-oriented communication to exchange traffic information between vehicles. Again, Service-orientation was only used between the gateways of two or more cars, not within the vehicles. Other approaches try to brake the technological barriers between different domains by introducing Service-orientation. Ragavan et al. for example use a Service-oriented framework based on well-known technologies like OSGi, Java and TCP/IP (see [108]). They are aiming at connecting embedded devices from domains like factory automation or automotive by introducing Service gateways. Tsai and colleagues present an approach in [134] which tries to include embedded entities like robots into the Web Services domain. Therefore, technologies like SOAP or WSDL are used to establish a connection between web servers and the embedded device using a gateway in between. Again both proposals analyzed do not implement Service-oriented principles on an embedded device. Another interesting scenario where SOA can help to connect embedded systems to other domains is when nomadic devices are to be integrated into a vehicle. This scenario is the basis of considerations in the proposals of Sonnenberg and Bohn et al. In [123] Sonnenberg describes a technique to couple nomadic devices to vehicles using Web Services. Thereby, technologies like OSGi and DPWS are used on a so called Vehicle Software Platform that connects the external entities to the ECUs of the vehicle. Within the SIRENA project, described by Bohn et al. in [25], a similar approach is described. Using DPWS on a gateway dividing the inside and the outside of a car, smartphones and other equipment are connected and the exchange of data between both worlds is enabled. Finally, a last scenario that inspired people to facilitate Service-oriented frameworks on gateways is the connection of a vehicle to the cloud. One approach targeting on this is presented by Iwai and Aoyama in [72]. Within a project called ACSS the authors developed a gateway that was attached to a vehicle and was able to request Web Services from the cloud. Thereby, common Web Services technologies like for example SOAP were used to receive data that is then forwarded to ECUs within the car. The same idea follow Xu and Yan in [151]. The gateway used in these approaches transfers the information from the cloud in form of xml files into serialized data streams in order to transport them to the ECUs in the vehicles. All the approaches within this second group that make use of gateways between the embedded devices and the Service-oriented domain lack of mechanisms or ideas on how the benefits for SOA could be transferred to those small ECUs. However, since the problem scenario of Distributed Driver Assistance Systems requires this transformation, none of them can be used to build the desired systems.

A third group of proposals from industry and academia fulfills the requirements of allowing online re-configuration of internal applications. Their authors have claimed that they are respecting the low resources provided by embedded systems. However, the following discussion will show that none of them is capable to be used within today's cars either since they make use of frameworks causing a high overhead or are not deployable on standard automotive network systems. One of these proposals has been made by Baresi et al. in [14]. It is targeting on automotive entertainment systems. By using two different Java-based, Service-oriented frameworks components like the radio or a display are implemented as Services to communicate with in-vehicle and external entities. Besides the enormous overhead caused by creating a system based on Java, the approach makes use of standards like SOAP, HTTP and TCP/IP. Since all of these technologies are IP-based they can't be applied to broadcasting automotive networks like CAN or LIN that easily. Some other proposals published in recent years are Java-based as well. One of them is part of the RT-Llama project published by Panahi et al. in [103]. Besides the disadvantage of creating a lot of overhead by using Java, it makes

Approach	Automatic configuration of internal components at runtime	Respects the characteristics of automotive embedded systems	Re-configuration is realized in a distributed manner
DOMINION - Gacnik and Haeger [53]	×	×	×
DINO - Foster et al. [51]	×	×	×
CODAR - Röckl et al. [112]	×	×	×
Xu and Yan [151]	×	×	✓
Ragavan et al. [108]	×	×	×
Ermagan et al. [47]	×	×	n.a.
Shokry et al. [122]	×	✓	n.a.
Baresi et al. [14]	✓	×	×
Static HMI - Eichhorn et al. [45]	×	×	n.a.
Flexible HMI - Eichhorn et al. [44]	✓	×	×
SIRENA - Bohn et al. [25]	×	×	✓
iLAND - Garcia-Valls et al. [57]	✓	×	×
RT-Llama - Panahi et al. [103]	✓	×	×
ACSS - Iwai and Aoyama [72]	×	×	×
DysCAS - Jahnich et al. [73]	✓	(✓)	×
SOME/IP - Völker [138]	✓	×	✓
eSOA - Scholz et al. [121]	×	×	×
Lopez et al. [86]	×	×	n.a.
OASiS - Koutsoukos et al. [78]	✓	×	×
Tsai et al. [134]	×	×	n.a.
SMEPP - Brogi et al. [28]	✓	×	✓
Bridges and Mostashfi [27]	✓	×	×
Sonnenberg [123]	×	×	×

Table 2.1: Service-oriented Computing for embedded systems

use of a central composition entity called "global resource manager" and IP networks to communicate. Another approach very similar to the two of Baresi et al. and Panahi et al. has been published by Koutsoukos et al. in [78]. The publication is part of the OASiS project which tries to create a Service-oriented middleware for wireless sensor networks. Within the project, each of the sensor hardware entities runs TinyOS and a Java virtual machine that hosts the middleware as well as the implementations of the Service logic. While trying to reduce the network load by introducing tailored message objects to be exchanged it still relies on the Internet Protocol and does not consider any other communication stacks. Other proposals do also use well-established technologies as a basis for their frameworks. One example therefore is the approach published by Bridges and Mostashfi in [27]. The authors implement their Service Instances using the Microsoft Sharepoint technology. This technology is Microsoft's platform to create distributed web applications. However, besides the fact that the Sharepoint installation causes an enormous overhead, the technology only runs on the Microsoft Windows Server OS. This leads to the fact that the computing platforms participating have to own at least a 64-bit CPU running at a speed of 1.3 GHz, 2 GB RAM and 160 GB hard disk space as these are the minimal system requirements for the operating system (see [96]). Even for the target domain of sensor networks these requirements are truly restricting the possible deployment scenarios. Other approaches try to deploy Web Service technologies within embedded systems. One example therefore is the SMEPP project funded by the European Union (see [28]). The developed Service-oriented middleware for peer-to-peer systems targets on devices like PDAs, handhelds or smartphones. Besides the fact that using Web Services standards like WSDL makes it necessary to use IP-based networks the overhead created by these technologies is too high for really small embedded ECUs like those that are used for simple sensors within the automotive domain. A solution for this problem could be the usage of the DPWS profile which uses only a subset of the Web Services specification to limit the amount of resources needed. This idea has been followed by Eichhorn et al. in [44] were the static approach of [45] has been enhanced to allow online re-configuration. However, this approach is still bound to IP-based communication. One last approach within this group is the iLAND project. Within this project, runtime changes of distributed embedded systems were examined and a Service-oriented middleware was developed (see [57]). However, the domains focused on were wireless applications for public transport, video surveillance and health care. For this reason none of the automotive networks is integrated into the middleware which makes it hard to use it in today's vehicles.

Finally, the last group of approaches to be discussed in this chapter actually focuses on automotive systems. Within this group, two approaches have been identified. The first one is called DysCAS (see [73]). It is the result of a European project with the same name that targeted at self-configuration properties for automotive systems. While focusing on telematics and infotainment it is able to be used in other scenarios, too. It integrated several network stacks including the Controller Area Network. However, the DysCAS middleware can't be used for the DDAS scenario for several reasons. The first one is that the organization of re-configuration is done by a single central entity. This includes the device integration on the CAN network (see [137]). This central organization approach causes the problems described in section 2.1 like the need for a central integrator or the inclusion of a single point of failure into the system. Furthermore, the scenario of truck and trailer applications is explicitly excluded from the project (see [52]) which causes a gap which can't be accepted within the scope of this thesis. In the last few years another Service-oriented approach has been created that targets especially on the automotive domain. It is called SOME/IP and has been developed and published by BMW (see [138]). Since version 4.1 it is part of the AUTOSAR specification. It organizes itself in a distributed manner and hence does not create a single point of

failure. However, SOME/IP is restricted to use IP-based communication to connect the Services to one another. Therefore, it can be rather seen as a future technology that might become more interesting as soon as the first vehicles are equipped with automotive Ethernet. This fact is particularly important since the automotive industry plans to use Ethernet only for video-based systems and as a central backbone in the near future. Therefore, small devices will not be IP-enabled for quite some time. As the truck and trailer scenario used within this thesis relies on the information of small ECUs and does not want to depend on some future but current technology, SOME/IP is not considered to be used within this thesis.

2.3 Summary

This chapter analyzed and discussed the Service-oriented approaches targeting on embedded systems that have been published in recent years. The analysis has been carried out by using three important requirements for a middleware solution in the scope of Distributed Driver Assistance Systems. The requirements were the possibility of runtime re-configuration of internal components, the limitation of size and computing power needed to be executable on small embedded devices, the possibility to use automotive networks as well as the absence of a central management unit that would create a single point of failure.

Using these demands, 23 different approaches were discussed. The first group of them lacked in offering online re-configuration capabilities. This is a important feature since it is the nature of DDAS to change their constitution at runtime. A second group was more promising since it implements this feature. However, the proposals gathered in this group are making use of a gateway approach. In other words, they suggest a additional entity that connects the embedded devices to external entities. However, the use of Service-orientation is restricted to the gateway and the integration of the external components. This circumstance does not allow to carry out a re-configuration procedure of the embedded devices. Group three fulfilled the demands of allowing dynamic re-configuration of the embedded devices. However, none of the devices discussed here completely matched the requirement to be applicable to automotive embedded systems. Some of them are using heavy-weight technologies like Java, Web Services or Microsoft Sharepoint. Others that are more frugal are only usable on IP-based networks which are not yet standard in the automotive domain. Two last proposals were analyzed in more detail since they are targeting directly to embedded automotive systems. The DysCAS middleware is a promising approach to handle self-configuration in vehicles. However, its centrally organized management system and the explicit exclusion of truck and trailer systems make it impossible to be used for assistance systems for truck and trailer combinations. The last framework to be looked at was SOME/IP, an IP-based, Service-oriented technology that even made its way into the AUTOSAR specification. However, since it excludes small ECUs that do not have a Ethernet connection, it cannot be applied to the problem domain within the scope of this thesis.

As a summary one can say that the 23 approaches, although none of them is directly applicable to truck and trailer assistance systems, are peppered with fantastic ideas, mechanisms and technological adaptations. Some of these have been integrated or at least adopted to the SODA framework. To the best of my knowledge, the SODA framework is the only one that fulfills all the demands requested by truck and trailer DDAS and thereby fills an important gap to achieve the benefits that Service-oriented Architecture can add to future automotive systems.

3 State-of-the-art in Model-driven development of SOA-based Systems[1]

'Perfection in design is achieved not when there is nothing more to add, but rather when there is nothing more to take away.'

Antoine de Saint-Exupéry[2]

3.1 Introduction

Model-driven Software Development (MDSD) has its roots in the software crisis of the late 60s and early 70s of the past century. The increasing size and complexity of the software projects in those days led to the failure of numerous of these. These circumstances stimulated research on a discipline which we now call Software Engineering: goal-oriented and well structured processes that support the development procedure of software systems. One approach within Software Engineering is MDSD. Based on the Computer-Added Software Engineering (CASE) tools of the 1980 it has proven its worth in different fields of computer science (see [87]).

This chapter will present and discuss the state-of-the-art in model-driven development of SOA-based systems. It will discuss the benefits of model-driven approaches in general, for embedded automotive systems and for the target domain of DDAS in the section 3.2. Section 3.3 illustrates best practices currently used in the automotive domain while section 3.4 discusses approaches for the model-based design of Service-oriented systems. In section 3.5 SoaML, a widely used, UML-based modeling language, is introduced. Finally, section 3.6 describes IBM's SOMA approach to develop Service-oriented software systems. The results and conclusions of this chapter, which are summarized in section 3.7, build the basis for the SODAdev development process described in chapter 5.

3.2 Model-driven software development

Model-driven Software Development can be described as an approach that focusses on the design of artifacts and on techniques to raise the level of abstraction of systems (see [120]). The basic idea is to use a modeling language to develop the specification of a software product. The resulting model can then be used for further steps such as formal verification or automatic code generation. The modeling languages used are often divided into two groups. The first group are general- purpose languages. These languages do not aim on a specific domain. Instead, they are supposed to be used in many different domains of application. This feature makes them very interesting for developers since they can be used in different projects belonging to diverse fields of application. It also makes them attractive for the developers of CASE tools since it

[1] This chapter is based on my publications [139] and [142]. Parts of it are extracted from these sources.
[2] 'Terre des Hommes, 1939, see [115]

widens the potential market. On the other hand, the wide focus of general- purpose modeling languages leads to a high level of abstraction which often causes problems when building precise models. Domain- specific languages (DSL) on the other hand allow this precision through being explicitly designed for the target domain. Through being tailored, appropriate notation and abstractions are offered (see [95]). Furthermore, only the properties actually needed are defined in their Metamodel which simplifies the development procedure.

One popular example for an implementation of MDSD is OMG's Model-driven Architecture [125]. It picks up the ideas of MDSD while making use of many of the standards defined by OMG itself.

3.2.1 Benefits of model-driven software development

Besides the structuring effect already mentioned MDSD comes with other benefits, too. These benefits have already been discussed in literature quite often (see e.g. [125], [62], [4]).

One advantage of MDSD is that functionality is developed independently from the platform executing them in the final product. Besides the obvious fact that this supports re-use it also lowers the complexity of the development in each step. This is because the software engineer does not have to take hardware restrictions into account while designing functional aspects and vice versa.

MDSD also makes development more convenient. This is mainly for three reasons. First, it creates some abstraction of the target functionality. This abstraction helps the software engineer by simplifying the potentially complex structures to be developed. Second, the modeling language used acts as a common language between the different stakeholders within a development project. This is important since software systems are usually designed by a heterogeneous team. This team often consists of software engineers as well as specialists from the target domain. These specialists may have no computer science background at all but use their own terminologies instead. Modeling languages can also help to build a bridge between the development team and external stakeholders such as management staff, marketing people or an investor. Lastly, MDSD assists by creating a distinct, human and machine readable description of the system under development.

Another benefit of using MDSD is the improvement in software quality. One reason therefore is the fact that model-driven development takes over some of the routine works from the developers. Human beings tend to treat tasks that have to be carried out over and over again more and more carelessly. This carelessness leads to mistakes within the development process which may eventually cause serious bugs in the final system. Another reason why MDSD helps in creating better software are the automatic verification routines that can be integrated into the development procedure. These routines inspect the artifacts under development for any violations of properties given in the Metamodel of the modeling language. Through pointing out problems at the very moment of their appearance this verification functionality helps to detect failures very early in the design process.

One last advantage of MDSD to be mentioned here is the increase of efficiency within the development phase. This is based mainly on three different factors. First, separation of concerns automatically arises when models are used for development processes. This is because the properties of a model are defined one after another rather than all at

the same time. This procedure automatically separates the different requirements and thereby reduces complexity. Second, the introduction of automated steps speeds up the development process and hereby saves valuable engineering time. It enables fast and error free addition of specification details to the system model. The last fact that increases the efficiency is the high potential for re-use. In MDSD functionality is specified using models. These models feature a clear and encapsulated description that can be easily extracted from the overall model to be re-used in another context.
Summarizing these benefits, one can say that using model-driven software development brings clear benefits. It enables the software engineers to design systems offering high quality and platform independence in a convenient and efficient way.

3.2.2 Benefits of model-driven software development for DDAS

Section 3.2.1 has clearly illustrated the advantages of using MDSD to design software systems in general. This thesis targets on a special subdomain: distributed, embedded systems that are used in the automotive domain for building runtime changeable driver assistance systems. Pohl et al. describe in [106] the current situation in developing embedded systems for the automotive domain using five problem field. I strongly believe that these problem fields can be faced using MDSD:

- **Growing complexity:** The desire for more and more features within cars increased the complexity of automotive systems significantly. The major share of these innovations is build in software. This boosts the size of software projects. MDSD is a great way to face this issue through reducing complexity by abstraction.
- **Physical context:** Embedded systems used in the automotive domain are usually connected to sensors and actuators that either observe or influence their physical environment. This requires a very heterogeneous development team with specialists from numerous technical fields. The usage of a modeling language supports the interdisciplinary corporation between the different team members.
- **High quality requirements:** Embedded systems integrated into a vehicle are generally bound to it throughout its whole lifecycle. Unlike other high-tech domains like consumer electronics e.g. a vehicle is used up to 30 years or more with an average lifecycle of about twelve years (see [66]). Furthermore, bug fixes can be only carried out by product recalls which is always accompanied by high costs and a loss of prestige for the car producer. As mentioned earlier, MDSD can help to improve the quality of the software system and thereby reduce these recalls.
- **High cost pressure:** There are only a few domains were high-tech products are developed under such a high cost pressure as it is the case in the automotive industry. Model-driven development can help to make the design process more efficient and hereby reduce development costs.
- **Extensive demand for variability:** A modern car can be ordered by the customer in many different variants. Furthermore car producers try to define common platforms and derive a whole series of different cars from this platforms. Both facts lead to a high demand of variability and a complex configuration management. MDSD can help to manage this issue through encapsulating functionality into separate models which can be easily combined to create new variants of the product. Furthermore, automated verification helps to detect problems of a configuration at an early stage.

Besides these advantages which are applicable to all automotive software systems MDSD is also beneficial looking at two very specific characteristics of DDAS in truck and trailer systems:

- **No central integrator:** The subdomain of commercial vehicles lacks of a central integrating instance. Often, there are different manufacturers of the truck, the trailer and the bodywork of both parts of the vehicle. Model-driven development approaches can help to improve the design procedure for systems in this subdomain since component specifications of the system can be extracted as single models. This high level of encapsulation simplifies the coordination of the development process when distributed over several different manufacturers.
- **Unified model of DAS:** As mentioned earlier in this work, the components and structure of driver assistance systems can be abstracted quite easily. This fact is very helpful when creating Metamodels for specific modeling languages.

Taking into account the general benefits of MDSD as well as its advantages relating to the specific characteristics of DDAS, I am convinced that creating a model-driven development process for the SODA framework is worth the effort. Such a tailored process model will simplify the development process especially in interdisciplinary teams and under high cost pressure. Furthermore, it is expected that such a procedure will increase the software quality significantly and will thereby contribute to increase the acceptance of using SODA for DDAS.

3.3 Best practices in the development of Automotive Software Systems

With the increasing size of software systems in the automotive domain the need for a structured development process rose as well. One of the most popular approaches used in this domain is the V-model. This model has initially been published by the German Federal Defense Ministry in 1991 (see [41]). Its original idea was to set a standard on how software suppliers should organize their development process when designing software for the authorities. In the following years the approach has been picked up by more and more companies to eventually become popular in many domains in the private sector. Compared to similar development models like for example the waterfall model, introduced by Royce [114] or the spiral model, published by Boehm in [24] the main difference of the V-model is the extension of the integration and testing phase. This addition leads to a better link between the specification and the test proceedings.

One software development process model that can be regarded as exemplary for the automotive sector is the so called core process for system and software development (CPSSD) (see [117]). It is based on the V-model while being tailored to the specific needs of the automotive domain.

The CPSSD development lifecycle is the result of many years of experience of it's authors Jörg Schäuffele and Thomas Zurawka. It reflects the processes actually used in the automotive industry in recent years. As pictured in Figure 3.1, the V-model is split up into two main parts. The first one that builds the left arm of the "V" consists of the specification and implementation phase. The second one that builds the right arm holds the integration and testing phases. The V-model is also divided into two levels. The upper level of the model is called the application level and consists of activities that refer to the overall application. The lower part of the model is called the component level. Here, all activities are related to some components of the application to be developed. The design workflow passes through the activities starting on the top left of the "V". By going down the left arm first the application and then the components of this application are specified in increasing detail. At the very bottom of the left side the software components are implemented. The development lifecycle then moves on to the right side of the "V". The level of detail decreases with every activity that leads the development

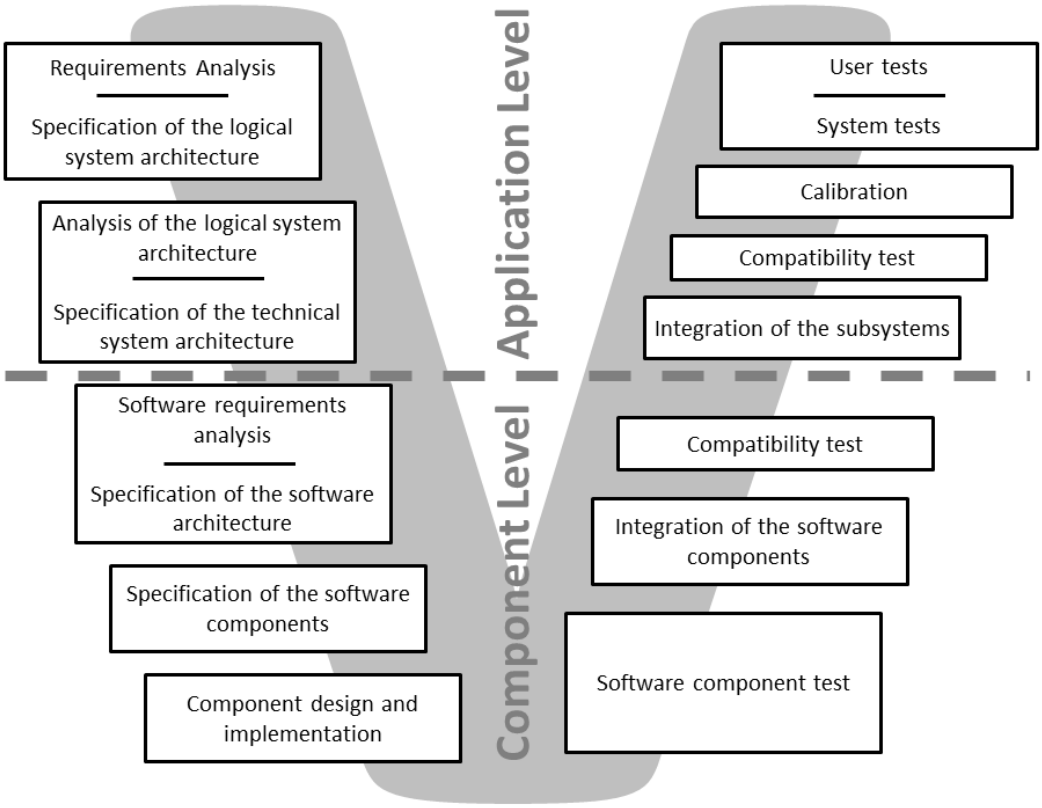


Figure 3.1: The V-model of CPSSD[117]

team up again. After testing the particular components, these are more and more integrated into subsystems. These subsystems are tested again and then integrated into the application. The V-model ends with a test of the overall system. Figure 3.1 presents the V-model in the automotive-specific variant CPSSD.

As this thesis focuses on the specification of self-adaptive automotive systems, the SODAdev development cycle is restricted to the left arm of the "V". According to [117] the first activity of CPSSD is to specify the so called Logical System Architecture (LSA). The LSA is an abstract architecture that does not provide any technical details. It is an intermediate step building a bridge between the requirements of the application and the Technical System Architecture (TSA). In this working product logical components are determined. Furthermore the functionality as well as the interfaces of these logical components are defined. In a next step, the TSA is specified. In contrast to the LSA this description of the application already contains some decisions on how functionalities of the application will be realized. In order to convert the logical system architecture into the technical one, a team of specialists is making technical decisions and proposes suitable solutions. After the TSA is defined, the component level is reached. This means that the specification of the overall application is done and from now on the identified components are specified with increasing detail. It also includes, that the development process splits up into a separate development process for each component to be designed. The authors Schäuffele and Zurawka propose a two-stage process. In the first part of this process, the software architecture of each component is developed. This step defines several software components including their interfaces to each other that build the overall software architecture. These individual software components are then specified in more detail. The last activity of the left arm of the "V" is to implement the specified components.

One important characteristic of the CPSSD development lifecycle is the transition

from system level to component level. At this point not only the overall architecture is defined but the development process is now split up into several smaller processes for the component design. In the automotive industry these processes are often delegated to specialized teams of developers or they are outsourced to suppliers. In some cases the specification of these components is extracted from the process to select already available commercial of the shelf solutions.

The fact that CPSSD is a development procedure describing the way automotive systems have been actually developed in recent years rather than being a purely academic model predestines it to be used within SODA. In this sense the design process within the framework is developed to fit in the course of activities given by CPSSD to prove its practicability in the automotive industry.

3.4 Model-driven process models for Service-oriented Architectures

Through to the popularity of Service-orientation in recent years a huge number of process models to develop such systems has been published. In 2009 Thomas, Leyking and Scheid identified 21 different approaches in [131]. Most of the currently available models are tailored for a special purpose, require a particular tool chain or concentrate on one field of application only. Instead of starting from scratch in designing a development procedure for the SODA framework the available approaches are analyzed. Based on this analysis one of these available approaches is selected to be customized to the unique requirements of the SODA framework. Furthermore, the customized process model is integrated into the CPSSD development procedure. In the following subsections the criteria of the analysis are described and applied to eleven different approaches.

3.4.1 Requirements in the domain of DDAS

The following criteria in order to identify a customizable process model to develop Service-based systems with SODA have been defined:

1. Completeness of the specification phase
2. Independence from a specific field of application
3. Variability in the scenario of development
4. Tool support
5. Acceptance of the modeling language
6. Easy integration into CPSSD

The first criterion states that the modeling approach has to allow a complete system specification which includes the specification of the Services as well as the Service Architectures. This also implies that a detailed technical point of view should be assured rather than focusing on the business domain which is very common using SOA. Finally, concrete methods or techniques on how to carry out the steps within the process model should be proposed.

Due to the lack of specialized approaches for the automotive domain the second criterion states that the field of application should not be restricted. Specialized models, used for Web Services for example, are not very promising since their focus is too narrow. Converting these to suit embedded automotive systems would change too many

of their essential ideas if possible at all.

Another criterion refers to the starting position at the very beginning of the process. In order to be able to use a design procedure this starting point should be variable. This is important because the process model should allow new developments as well as migrating existing systems. Especially the latter point is expected to increase the interest in SODA as it allows to transfer legacy systems into Service-based applications.

The fourth criterion states that tool support should be given. Using a tool that for example allows modeling the system graphically simplifies the development process. In addition, implemented validation functionality decreases the probability of semantic errors as stated earlier in this chapter. Last but not least the existence of a tool that supports the process model simplifies the development of the SODA design procedure as no extra software has to be developed.

Furthermore, the modeling language deployed should be widely-used and hereby accepted. This demand is set up because of the nature of development teams in the automotive industry. As stated earlier, these teams are normally constituted by members with different backgrounds such as software engineers, electrical engineers or mechanical engineers. A widely-used modeling language simplifies the communication within the group and reduces the risk of misunderstandings.

Finally, the last criterion expresses that the development process has to be capable of being integrated into CPSSD. By enforcing this capability the compatibility of the development cycle with best practices of the automotive industry is ensured. This fact may increase the interest of automotive manufacturers to use the SODA framework within their products.

3.4.2 State-of-the-art in model-driven development of SOA-based systems

Using the criteria presented in section 3.4.1, eleven process models are analyzed. These models are representing academic research as well as industrial practices. Table 3.1 gives an overview of these approaches alongside with their characteristics regarding the six criteria.

The first one to be analyzed is a model proposed by Stein and Ivanov in [126]. Both authors are associates of a consulting company. For this reason their approach belongs to the group of industrial practices. The model is based on ten phases starting with a business process model ending with the deployment of the developed system. It focuses on business processes and the modeling languages suggested are very common ones which belong to the domain of Web Services. Through using these widely used languages acceptance and tool support is granted. Furthermore, the composition of the development steps covers all major fields and is quite similar to the specification phase of CPSSD. The latter point ensures an easy integration of the procedure into CPSSD. On the other hand, the development scenario is restricted to new developments. Besides, the development model has a clear focus on business processes which creates a strong barrier when using it for the design of embedded systems.

A similar model, the Enterprise SOA Roadmap method is presented by Hack and Lindemann in [65]. It has a very narrow focus in which a four phase model is presented to introduce and continuously improve SOA-based enterprise software within a company. This model emphasizes the business factors since only one of the four steps to be executed is technical. The authors do not use a common modeling language but a proprietary software product of their company SAP. All these factors restrict the field

Publication	Completeness of the spec. phase	Domain independence	Scenario variability	Tool support	Acceptance of modeling lang.	Easy integration into CPSSD
Stein & Ivanov [126]	✓	×	×	✓	✓	✓
Hack & Lindemann [65]	×	×	✓	✓	×	×
Pingel [105]	×	✓	×	×	✓	✓
Mathas [91]	×	×	✓	×	✓	×
Bell [17]	×	×	✓	×	×	×
Papazoglou & Heuvel [104]	✓	×	✓	×	✓	×
Barry [15]	✓	×	✓	×	✓	×
Nadhan [98]	✓	✓	×	×	×	×
Gebhart et al. [58]	×	×	✓	✓	✓	✓
Elvesæter & Carrez [46]	✓	✓	✓	✓	✓	×
Arsanjani [9]	✓	✓	✓	✓	✓	✓

Table 3.1: Comparison of model-driven development approaches for SOA-based systems

of usage in a way that it would not be possible to use the approach without major modifications.

Other approaches lack of concrete modeling techniques. Pingel [105] for example, introduces a technology independent five phase model extending well-known approaches. Within these phases Services are developed that are build by a four layer architecture each. The steps are shaped in a way that they are capable of being integrated into CPSSD. Through staying on an abstract level the procedure is not focusing on any specific domain. However, besides a hint that UML could be used for modeling, the approach lacks of detailed descriptions on concrete work steps. Furthermore it is not offering any tool support or variability in the development scenario.

Another approach quite similar to the one of Pingel is a proposal of Mathas presented in [91]. It extends the software life-cycle model by adding some SOA-specific tasks and roles. The life-cycle model consists of three phases. In this context only the design phase is of interest. During this phase five development steps are defined while staying very coarse-grained. The cycle style of the model allows to be quite variable in the development scenario. Furthermore, popular languages of the Web Services domain like for example WSDL are used to specify the functionality which ensures acceptance in the community. On the other hand it restricts the usage of the approach to the Web Service domain. Besides, it is difficult to integrate Mathas' process model into CPSSD through to its cycle style. In addition, the approach lacks of the definition of concrete development as well as of tool support.

The Service-oriented Modeling Framework published by Bell is quite generic, too (see [17]). The idea of the author is to design a concrete process model for every case of application derived from his abstract methodology. Bell's approach is variable in the development scenario but clearly targeting on the enterprise software domain. Besides the author proposes a special design notation which violates the criterion of using a widely-used modeling language. In addition there is no tool available to support the development team. The generic and very abstract nature of the model is also hard to combine with the concrete development steps given in CPSSD. Just like the other

procedures discussed so far, Bell's model is rather to be seen as suggestions on how a process model may be set up than being a concrete model itself.

Unlike the previously named ones the model "Service-oriented design and development" published in [104] by Papazoglou and van den Heuvel is quite technical and concrete. The procedure is organized in a cycle style and grouped into five phases plus an additional planing step. This arrangement ensures that new developments as well as the re-design of legacy software can be executed using this model. On the other hand it makes it difficult to use in a V-model arrangement like in CPSSD. The usage of the widespread modeling language "Business Process Modeling Language" (BPML) increases the acceptance in the community but also reduces the application domain to enterprise software systems. Furthermore, the authors do not present any tool supporting the developers.

The approach "Creating Service-oriented Architectures (CSOA)" developed by Barry and presented in [15] is also offering a very concrete procedure. It is organized in five consecutive phases. The main idea of Barry is the introduction of Service-orientation into an enterprise software system by using experiments. This approach is quite different from other development procedures which is a problem when trying to integrate it into CPSSD. The usage of the "Business Process Execution Language for Web Services" (WS-BPEL) to specify the system ensures acceptance while narrowing the focus of the process model. In fact, it would be quite complicated to transfer this approach to other fields of application without major changes. Furthermore, just like the approach of Papazoglou and van den Heuvel no software to be used during the design steps is presented or suggested.

Another approach is presented by Nadhan in [98]. The author describes a seven step procedure to migrate an existing solution into a SOA-based system focusing on technical issues. Targeting only on the migration scenario this model cannot be used for new developments. It is independent from the domain of usage but brings up other issues, too. Although being quite concrete in the different development steps it does not suggest any modeling language to be used. Furthermore, no tool support is given. Lastly the bottom up approach chosen by the author can't be integrated into a V-model that easily.

Some highly interesting approaches are using the Service-oriented modeling language (SoaML), a notation created to model and design SOA-based systems. This is a promising approach because the language itself satisfies the criteria set up in being not restricted to one field of application and being widely used since it is a profile of the popular Unified Modeling Language (UML). One of these process models is presented by the researchers Gebhart et al. in [58]. The authors describe the development of a Service-based monitoring system by identifying and specifying the needed services. Although this is very promising, it does not allow to specify the architecture of the overall system which violates the criterion of enabling the user to carry out a complete system specification. Furthermore it has a very narrow focus on surveillance systems which complicates the usage in any other domain.
Another methodology using SoaML introduced by Elvesæter and Carrez in [46] closely follows the processes defined in the Model-driven architecture (MDA) approach published by the Object Management Group. This process model defines several specification steps within the computational independent model and the platform independent model of MDA. Tool support is granted by the modeling tool "Modelio". Further, the approach allows new developments as well as the transformation of legacy systems. However, the development steps defined within the three main models are organized in a cyclic style. This characteristic would make an integration of the process model into CPSSD quite

complex.
One last approach using SoaML is called "Service-oriented Modeling and Architecture" (SOMA) and is presented by Arsanjani in [9]. This phase-oriented lifecycle model makes use of IBM's "Rational Software Architect" but is in principal executable using any UML modedling software. Besides the fact of using the "Business Process Modeling Notation" (BPMN) to describe a first coarse-grained model of the system under development it is free of any restrictions regarding the area of application. It is able to specify a complete system including a detailed description of the Services used within, using concrete and practical steps. Furthermore, it is variable in the starting point of a development process and easily integrateable due to it's well structured top-down approach. Just like CPSSD it can be also divided into an application and a component level which simplifies the integration even more.
The analysis of available model-driven development processes for Service-based systems under the spotlight of the demands set up by the target domain has shown that SOMA is the only approach to fulfill all requirements. For this reason SOMA is chosen to be adapted to the SODA framework and then integrated into CPSSD.

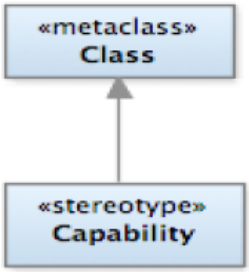
3.5 OMG's Service-oriented Architecture Modeling Language (SoaML)

The Service-oriented Architecture Modeling Language (SoaML) is a specification of the Object Management Group (OMG) dedicated to allow a model-based specification of Service-oriented applications. It is a profile of UML 2 which means that it defines its own stereotypes that represent elements of SOA by deriving them from UML metaclasses. The project to create this profile started in 2006. The first specification published dates from December 2009 and was adopted from OMG in March 2010. The current version SoaML 1.0.1 has been released in May 2012. This last version names a total number of 19 different stereotypes. In this section only seven of them are introduced since only this subset is used in SOMA and the SODA development procedures.

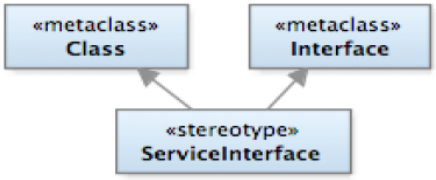
The first stereotype to be looked at is named Capability. A SoaML Capability is used to identify Services needed within an application (see [102]). In this sense they represent some functionality. Another important characteristic of this stereotype is, that it does not provide any hint or definition on how the functionality offered by this prospective Service is realized and implemented. In doing so, SoaML establishes the principle of separation of concerns. From an UML perspective a SoaML Capability extends the metaclass Class as illustrated in Figure 3.2a.

Another important stereotype of SoaML is the ServiceInterface. It is meant to be used as the definition of a Service (see [102]). Referring to the stereotype description given in Figure 3.2b the SoaML ServiceInterface extends two metaclasses. It receives its internal structure from the UML 2 Class and its connectivity from the UML 2 Interface. SoaML allows both methodologies currently popular in Service-oriented Computing: the contract-based and the interface-based approach. In the latter one all information on what the Service offers and how it can be invoked is modeled within the ServiceInterface. In the contract-based methodology this information is partly moved to an additional artifact, namely a contract. SOMA and the derived SODA development process model are using such contracts. Hereby the ServiceInterface is only supplemented by standard UML 2 Interfaces. Using these Interfaces to represent functionalities provided and requested by the Service helps to structure the specification.

Irrespective from the methodology used a ServiceInterface has to be logically connected to the Capability and thereby to the functionality it offers. SoaML therefore

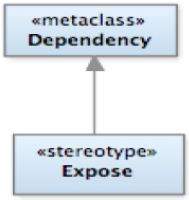


(a) The SoaML stereotype Capability [102]

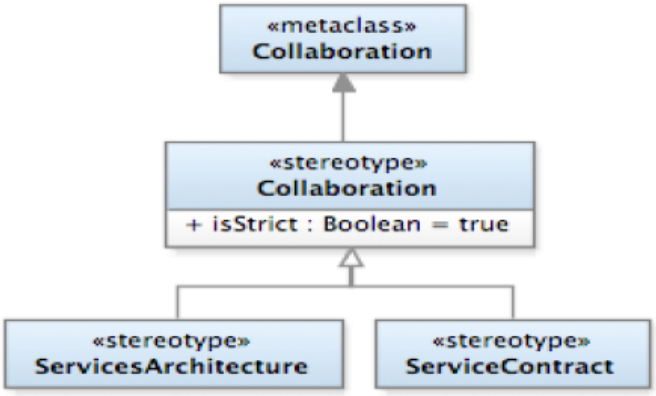


(b) The SoaML stereotype ServiceInterface [102]

Figure 3.2: SoaML stereotypes part 1



(a) The SoaML stereotype Expose [102]



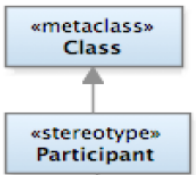
(b) The SoaML stereotypes ServiceArchitecture and ServiceContract [102]

Figure 3.3: SoaML stereotypes part 2

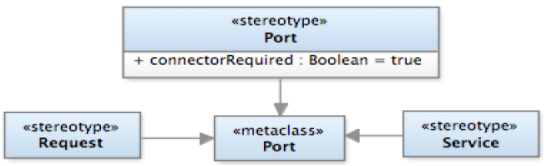
provides a special stereotype called Expose. As pictured in Figure 3.3a it is an extension of the Dependency metaclass. Its only purpose is to indicate that a Capability is represented by a specific ServiceInterface by connecting these two in the ServiceInterface model.

When using the contract-based SOA methodology the SoaML stereotype ServiceContract is indispensable. It is "[...] the formalization of a binding exchange of information [...] between parties [...]" ([102]). In this sense it specifies how a Service is accessed and invoked. This is done through introducing two main specifications: the entities participating in a Service call and the protocol of messages used. The former one is done by introducing roles. Each role represents a participating party. In many cases these roles are a provider and a consumer of a Service. Nevertheless the number of roles could generally be much higher as additional roles, like for example brokers or gateways, may be added. In SoaML roles can be presented by any of the stereotypes ServiceInterface, Interface or Class. The roles determined are picked up in the second part of the ServiceContract specification. In this step any adequate UML diagram capable of defining a message exchange is used to specify the behavior of the Service at its network interface. From an UML perspective a ServiceContract is an extension of a Collaboration which allows to illustrate the interaction and cooperation between two or more entities (see Figure 3.3b).

Another SoaML stereotype used in the SOMA process model is called Participant. It is again an extension of UML's metaclass Class as illustrated in Figure 3.4a. This type is used to model a provider or a consumer of a Service. Thereby a Participant



(a) The SoaML stereotype Participant [102]



(b) The SoaML stereotype Port [102]

Figure 3.4: SoaML stereotypes part 3

represents some involved party within a Service-oriented application (see [102]). SoaML does not describe the characteristics of this stereotype in detail. Instead it leaves it open to the developer whether this type describes for example a person, organization, system, application or component. This is due to the fact that the SoaML profile is meant to be used within a wide spectrum of domains. Any limitation of the character of such a Participant would potentially lead to a reduction of the domains of usage. Irrespective from this coarse-grained definition the semantics of introducing a Participant are quite clear: It is used to establish an assignment of the Service specified to some entity. In other words the functionality within an application is allocated to the different entities available. Therefore Participants, which represent these entities, are equipped with interaction points were Services are offered or consumed. The main type used to realize such an interaction point is the SoaML stereotype Port. As Figure 3.4b shows, a SoaML Port is an extension of the UML stereotype Port. At the same time two specialized types namely Request and Service are introduced. In doing so the direction of Service provision can be modeled. However, it is still up to the developer to introduce this additional information using Request and Service or to leave it open using the more general Port stereotype when specifying an interaction point of a Participant.

One last SoaML stereotype to be mentioned here is the ServiceArchitecture. As shown in Figure 3.3b it is an extension of Collaboration. Unlike the other types presented here this one does not focus on a single Service or entity but on the overall application. It brings together the participating parties and models how they interact with one another in a formal high-level view. Hereby the parties are represented by Participants. Each Participant may offer or request a number of Services modeled by its Ports and the ServiceInterfaces assigned to them. The interaction of the Participants is specified by introducing the ServiceContracts. Within these ServiceContracts each role is assigned to a ServiceInterface of a Participant. By creating such a ServiceArchitecture using the previously defined Participants and ServiceContracts a functional specification of the Service-oriented application is created.

The SoaML stereotypes introduced in this section build the basis for the SOMA methodology as well as the SODA development process derived from it. The remaining sections of this chapter will introduce how these types are interpreted and used within a development procedure.

3.6 IBM's Service-oriented Modeling and Architecture

Service-oriented Modeling and Architecture has been published by IBM employee Arsanjani in [9] in 2008. It is based on the UML profile SoaML introduced earlier in this chapter. To support the developers in creating SoaML models Arsanjani recommends the usage of IBM's modeling solution Rational Software Architect (RSA). However, any other SoaML or UML tool can also be used.

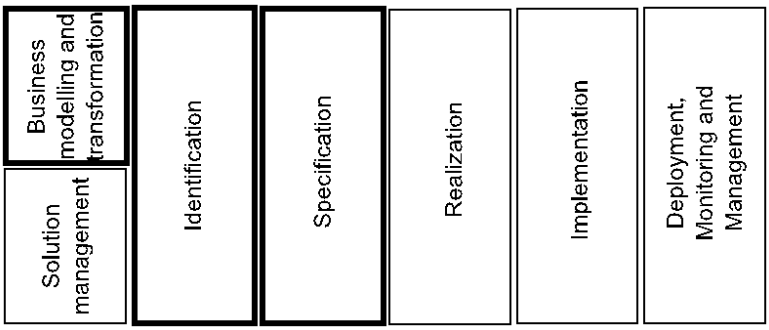


Figure 3.5: The seven phases of IBM's SOMA process model

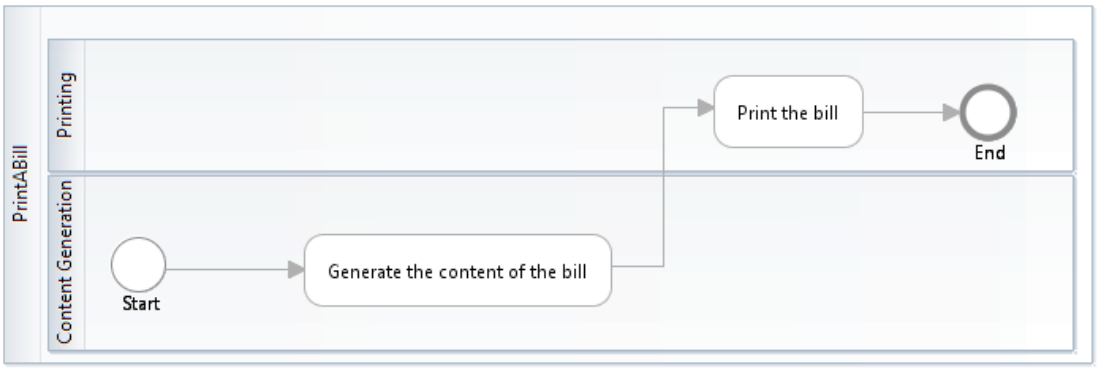


Figure 3.6: The BPMN model of the bill printing example

The process model consists of seven phases. Besides the first two of these, which are carried out simultaneously, all phases are executed consecutively. Figure 3.5 illustrates these development steps. Within the first step called Business modeling and transformation a first coarse-grained and semi-formal description of the system under development is created. The goal of this phase is to get a first idea of the functionalities of the system and how they interact. As IBM mostly used the SOMA approach to develop enterprise software systems it is suggested to use the Business Process Modeling Notation (BPMN). BPMN is a graphical notation to model business procedures and transactions within an organization. In SOMA these procedures are transferred into Services. Figure 3.6 shows a BPMN model of a very simple business process. In this example a bill is prepared and printed. The overall transaction is enveloped by a so called pool. This pool is subdivided into lanes which represent for example different parties within this transaction. In the example given the pool is called "Print a bill". The two lanes are named "Content Generation" and "Printing" respectively. The functionality needed to carry the overall transaction is symbolized using activities. These activities are attached to the parties that execute them by arranging them on the corresponding lanes. A sequence flow connects the activities and thereby defines the order in which they are carried out. The flow is started by a start event and completed by an end event. In the example given in Figure 3.6 two activities are defined. The first one creates the content of the bill while the second one handles the printing.

In the parallel phase Solution Management a project management process is defined and established. Furthermore first technical decisions are made like for example the selection of the platform to host the new system. After finishing both of the first steps the third phase called Identification is started. In this phase potential candidates for Services are identified. In order to do so, SOMA names a number of different approaches. Most of them are focusing on business processes. One very practical technique makes use of the BPMN model created earlier in the development process. Thereby each

Figure 3.7: The the prospective Services of the bill printing example modeled as SoaML Capabilities

lane is extracted and converted into a SoaML Capability. Furthermore, each activity attached to the lane is transferred into a method within the newly created Capability. The sequence flow given in the BPMN description is also transferred into SoaML. This is done by connecting the Capabilities through use dependencies. Figure 3.7 shows such a Capability model for the printing example. In the picture, the lanes and activities of the BPMN model have been converted into SoaML Capabilities and methods respectively. The sequence flow of the BPMN model which led from the content generation activity to the print activity has been transferred into the use dependency connecting the printBill and the generateContent Capabilities. The approach used by SOMA to transfer the lanes which represent acting parties into Capabilities leads to a quite coarse-grained model containing a low number of rich and complex Services.

Having identified the potential future Services and modeled them as SoaML Capabilities the SOMA process model passes on to the Specification phase. As it is very extensive it is split up into four consecutive sub phases namely the specification of the ServiceInterfaces, ServiceContracts, Participants and ServiceArchitectures.

The ServiceInterface is the gateway of the Service to the outside world. They encapsulate the complexity of the Service logic while allowing standardized access to it. In SOMA the ServiceInterfaces are directly derived from the Capabilities. This is done by using SoaML's Expose dependency which indicates that the inherent functionality of the Capability is offered through the associated ServiceInterface. As ServiceInterfaces can be extended by standard UML Interfaces SOMA uses this ability to structure the different functionalities of the Service. Therefore SOMA proposes to create additional Interfaces and transfer the methods executing some functionality into these entities. It also suggests to add Interfaces for those functionalities that are not implemented within the Service but need to be called externally. As a summary SOMA tends to be quite vague at this point which gives the developer a high degree of freedom when modeling the ServiceInterfaces. In this sense, the model pictured in Figure 3.8 represents only one possible way of modeling the ServiceInterface of the exemplary printBill Capability. It uses the Expose dependency to create a logical connection between the Capability and its ServiceInterface. Furthermore two Interfaces are attached with the use dependency. The Interface on the left hand side symbolizes the functionality of the Service while the one on the right hand side describes the need for another Service.

In a next step the ServiceContracts are defined. This is needed since SOMA uses a contract-based specification approach rather than an interface-based one which would be also possible using SoaML. In this contract-based specification details of the protocol to access the Service are described in the ServiceContract rather than being directly attached to the ServiceInterface. The contract specification starts with the definition of the roles within a Service call. The number of parties involved is theoretically unlimited as described in section 3.5. However, in a standard case there are at least a provider which is the Service itself and a consumer which is the calling instance. In broker-based approaches a third role may mediate between these two parties. The stereotypes used for

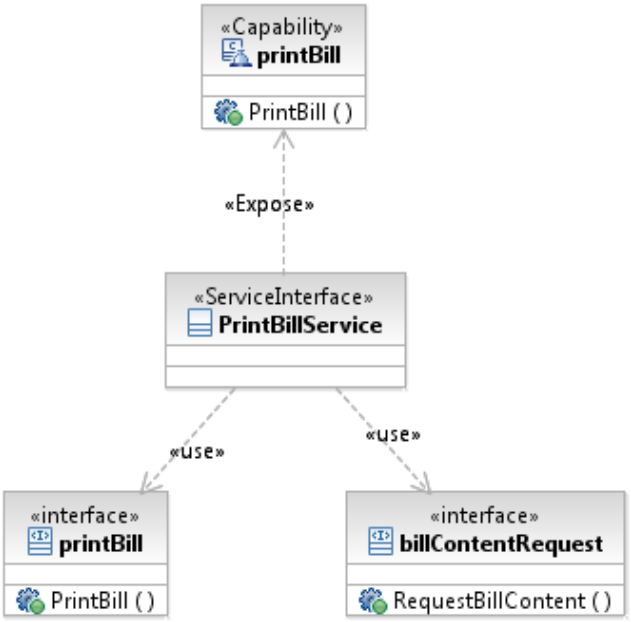


Figure 3.8: A possible model of the ServiceInterface of the Service to print a bill

these roles are not narrowed down by the SOMA methodology. Thereby SOMA allows all possibilities defined by SoaML which names ServiceInterface, Interface and Class. Taking these circumstances into account the decision to introduce only two roles and to use Interfaces to represent them as pictured in the upper part of Figure 3.9 can again be seen as only one option among many others. In a second step the protocol to access the Service is defined into the ServiceContract. Again, SOMA does not provide any details on the type of diagram to be used. Instead it refers to the SoaML specification and hereby allows any adequate UML diagram that is able to specify a network protocol. Hereby the style and the level of detail of the protocol specification remains completely open. The lower part of Figure 3.9 shows a very simple description using an UML sequence chart.

The next step within the specification is used by SOMA to assign the Services to particular hardware units. Therefore one SoaML Participant is introduced for each future hardware entity of the system. These Participants are equipped with interaction points which bind a ServiceInterface. By attaching these interaction points, including their bindings to ServiceInterfaces to a Participant, the functionality offered by the Services is assigned to the hardware entities available. Figure 3.10 illustrates the assignment within the simple example introduced earlier. It shows two SoaML Participants namely "BillPrinter" and "BillContentProducer". Each of them holds one SoaML Port which themselves encapsulate a SeviceInterface each.

In a last specification step the overall architecture of the application is modeled. In SOMA this is done by creating a SoaML ServiceArchitecture. This stereotype brings together the Participants with the ServiceInterfaces attached and defines their relationships through the ServiceContracts. In the model given in Figure 3.11 both Participants specified in the last step are included. They are connected to each other using the GenerateBillContent contract created in the last step of SOMA. Thereby the BillContentProducer Participant takes the role of the provider while the BillPrinter is the consumer in this relationship. In addition, a third party is introduced into the architecture. The UML Actor named ServiceRequester is used to illustrate that the overall application is triggered by an external party. The specification of the ServiceArchitec-

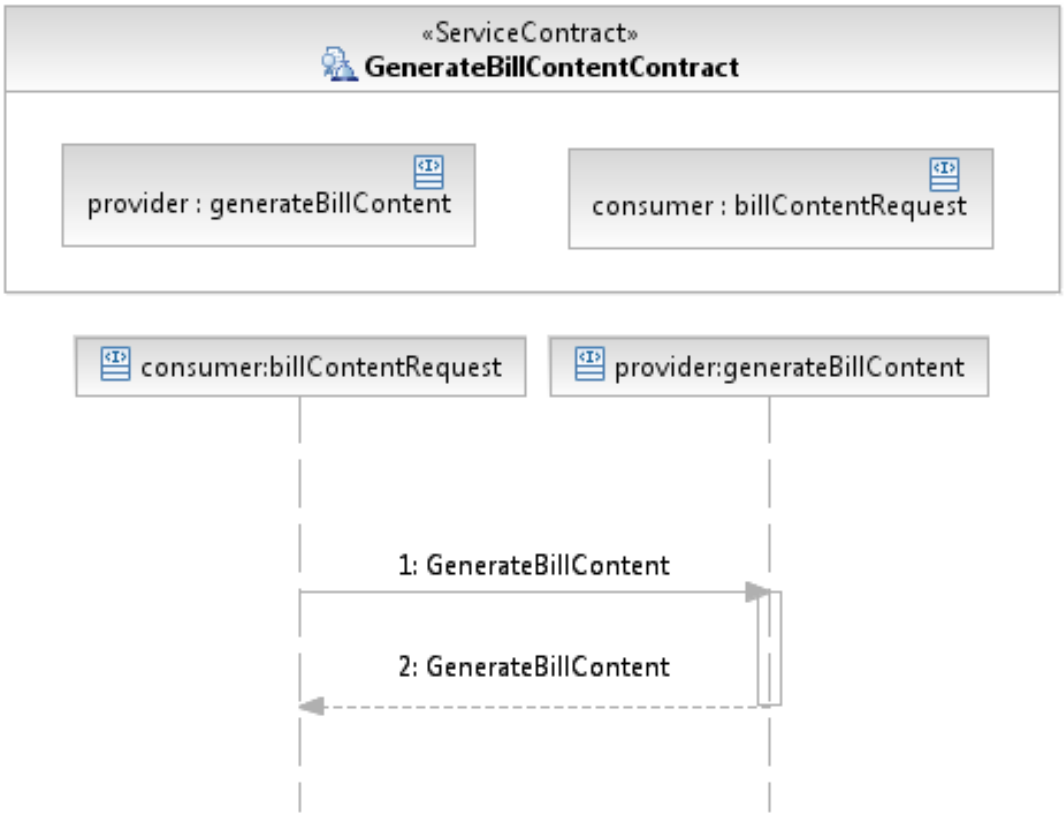


Figure 3.9: A possible ServiceContract of the exemplary Service which creates the content of a bill

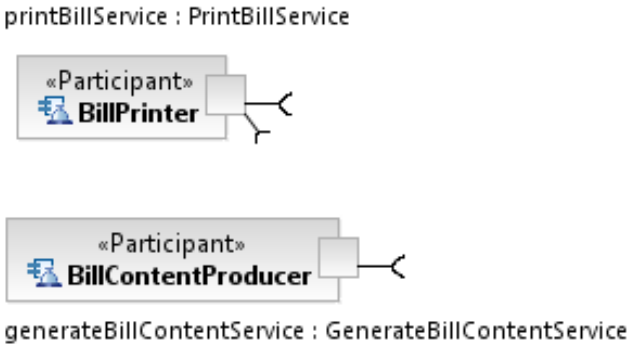


Figure 3.10: The SoaML Participants of the example application

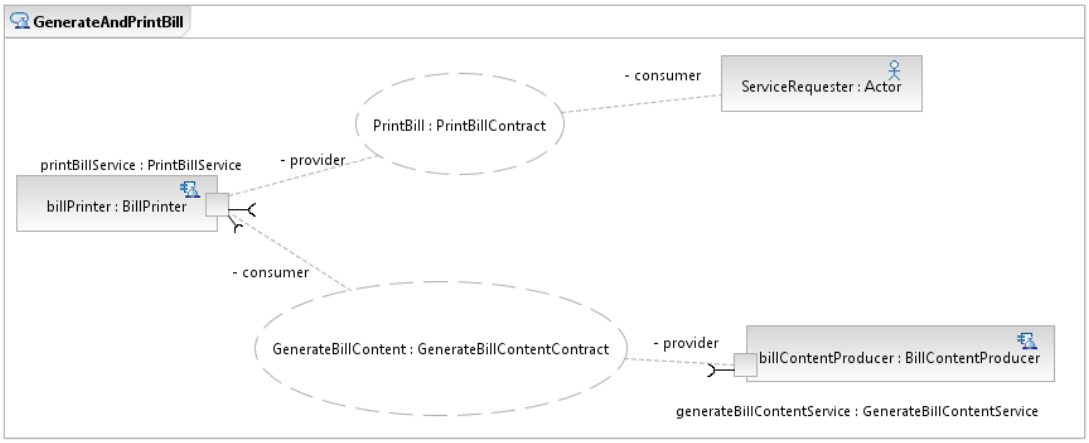


Figure 3.11: The SoaML ServiceArchitecture of the example application

ture finishes SOMA's Specification phase. In doing so it also completes the specification of the functional requirements. The remaining three phases shall be introduced here only very shortly since they are not used in SODA's development procedure. In the Realization phase the focus swaps towards the non-functional requirements of the application. This includes for example the definition of communication layers. As soon as the non-functional specification is done the developers start to write or generate code and thereby move on to the Implementation phase of SOMA. Finally, in the Deployment, Monitoring and Management phase the developed application is launched and observed to ensure it is working correctly.

3.7 Summary

This chapter summarized the state-of-the-art in model-driven development of Service-oriented systems. Such a model-based approach brings clear benefits to the development of DDAS: First, it introduces abstraction which reduces the complexity in each development step. Second, a common language is defined that simplifies the work within multi-disciplinary development teams. The quality requirements of this domain are met by decreasing the amount of error-prone routine work and introducing online verification. By making the development process faster and more efficient the design costs can be reduced especially when creating families of systems with a high variance.

After introducing the widely used V-model and its automotive derivation CPSSD several existing development frameworks for Service-based systems have been analyzed using six domain specific requirements. Thereby IBM's SOMA methodology was identified to build the basis of the SODAdev development procedure. Finally, SOMA as well as its underlying modeling language SoaML have been introduced.

The considerations done in this chapter are picked up in chapter 5 and taken into account when introducing SODA's development process SODAdev.

4 A Reference Model for SOA in the automotive domain

'Any sufficiently advanced technology is indistinguishable from magic.'

Arthur C. Clarke[1]

4.1 Introduction

The term of a reference model does not feature a clear separation within the research community. In this sense, there is a need to specify what a reference model in this context and why it is needed here. According to David Hollingsworth ([69]) a reference model is a model describing "characteristics, terminology and components, enabling the individual specifications to be developed". In this sense a reference model introduces the important ideas and principles of a class of systems. Furthermore, a terminology is defined that simplifies the exchange of information. Besides, it acts as a blueprint for this class of systems that can be applied to a concrete problem to create a specific implementation as it describes the potential components of such a system and how they relate to each other. August-Wilhelm Scheer agrees with this in describing a reference model as a "starting point for the development regarding a concrete problem" (see [118]). The reference model defined in this thesis bases on the definitions of Hollingsworth and Scheer.

The scientific community as well as companies working within the SOA published numerous models for Service-oriented systems in recent years. Some of them, such as the zapthink Service-oriented Architecture Roadmap (see [119]), make use of a phase-oriented approach to introduce Service-oriented concepts. Other models are system models that describe the SOA-components within the overall system architecture of a software solution. Examples of these system models for SOA can be found in the ESOA project published in [128] or in Enterprise SOA Maturity Model published by Durvasula et al. in [42]. A huge number of models regarding Service-oriented Computing matches the category of meta models. Some of them try to structure the terms and concepts of Service-oriented Architecture for a specific domain. One example therefor is the so called Cloud-SOA Meta Model presented by David Sprott (see [124]). Other work targets on a consistent concept definition in order to combine SOA with platform-based computing (see e.g. [18]). But most of the meta models published are serving as the basis for a Service-oriented modeling language. Examples therefore are the meta model for SoaML (see [101]) or the one for the language UML4SOA which is also a Service-oriented modeling language based on UML (see [92]). Another group of models can be described by the term component model. These models introduce several components of a Service-oriented Architecture and specify the relations between them. Examples for this class of models are the OASIS reference architecture (see [48]) and the W3C Web Services Architecture (see [26]). All the types of models named above focus on a single characteristic of a Service-orient system. On the contrary, reference models combine much of these models as they integrate terms and concepts as done in

[1] 'Profiles of the future, see [37]

meta-modeling and architectural elements as described in system and component models.

The number of complete reference models in the SOA world is quite small. Within this work, four of these models have been identified and analyzed. The first one has been developed by Mark Wilkins and his colleagues at the Oracle Corporation (see [148]). The so called Oracle Reference Architecture defines all used terms and principles and suggests a so called logical architecture. This architectural blueprint defines several layers and specifies the relationships between them. A similar model has been presented by Heather Kreger. She is an associate of the IBM Software Group and describes the Web Services Conceptual Architecture in [79]. As it has been done in the Oracle Reference Architecture, this approach defines used terms and principles as well as a reference architecture in form of the IBM Conceptual Web Services Stack. Furthermore, it also demonstrates how this reference model can be integrated into the e-business domain. Another reference model has been described by Farcas et al. in their approach called Rich Services (see [49]). The focus of this model lays on the conversion of legacy systems that are build using a component-based approach into Service-oriented applications. Additionally to the definitions done in the two approaches explained earlier, a detailed development process has been designed that supports the conversion. The last model to be mentioned here is the OASIS Reference Model for Service-oriented Architecture (see [88]). This joint work of many big players within the software industry complements the OASIS reference architecture described earlier by the definition of clear terms and the introduction of many concepts used within SOA-based systems. It also tries to reference itself to given industrial norms like the W3C Web Services Architecture. Furthermore, it includes an information model containing structures and semantics of the data packets exchanged between Services. While all of these reference models describe their specific ideas on what Service-oriented Architecture is and what terms, concepts and components should be used within it, none of them considers the specific needs and goals of the automotive domain. If one would try to adapt one of these models to DDAS this would rather confuse the scientific community. Instead, a unique reference model for the SODA framework has been developed. It will constrain itself to those concepts, terms and components that are actually needed to achieve the goals set. Thereby it will be clearly structured and well tailored to the special needs of Distributed Driver Assistance Systems.

The reference model for SOA in the automotive domain follows the definitions of Hollingsworth and Scheer. Within this chapter, a terminology for the usage of Service-orientation in this domain will be defined. This terminology will help to describe the concepts and components used for the SODA framework. Furthermore, the reference model for the automotive domain will propose an architectural blueprint that can be used as a starting point within a development process for a concrete implementation in the form of a reference architecture.

4.2 The SODA reference model

This section will introduce the SODA reference model. The first part of this model describes the terms and concepts used within the framework. This important basis is introduced in section 4.2.1. In the second part the architectural blueprint of SODA is described. As the SODA framework has been developed for the specific purpose of supporting DDAS, the concepts used are derived from the characteristic needs of such systems. These needs are expressed by a number of goals which are presented in subsection 4.2.2. The goals introduced there are matched with Service-oriented concepts in order to fulfill them. Subsection 4.2.3 illustrates this relationship between the goals and the concepts. In a next step, the concepts introduced are realized by software

components. These components as well as the architectural blueprint for the SODA Services which they build are described in subsection 4.2.4.

4.2.1 Terms and concepts of Service-oriented Computing

The central entity of every Service-oriented system is the Service. Within this work the definition of the term Service is derived from the SOA definition given in section 1.3 of this thesis: a Service is a self-contained entity that encapsulates functionality behind a well-defined interface. This interface ensures that the entity can be invoked in a standardized way and discovered within the network by other participants of a communication system.

Besides this basic definition, Services can be characterized by several properties. First of all one can distinguish between an Abstract Service and a Service Instance (see [150]). An Abstract Service describes a specific functionality as well as the interface to access it. However, it is a conceptual entity which is not actually implemented. In contrast, a Service Instance is a concrete implementation of an Abstract Service. It is executable and available within the system.
Another classification that can be made is the type of a Service. Derived from the generic loop of informing DAS as presented in section 1.1.1 there are three distinct types:

- Source Services
- Sink Services
- Data Processing Services

A Source Service encapsulates a functionality that creates some kind of data. This can be for example a Service reading information from a database or a sensor measuring a physical value. The complement entity is called Sink Service. This kind of Service makes use of some data to realize a specific functionality. In a system containing a human machine interface this functionality can be for example to output a notification to the user. Data Processing Services are encapsulating functionality that makes use of some data to create a specific benefit or enhance the data with additional information. Such a Data Processing Service could for example host a functionality that checks a customer address retrieved from a Source Service for inconsistencies before handing it over to further processing or a Sink Service.
Another distinction that can be made concerning where the functionality is actually located. As mentioned earlier, a Service encapsulates some functionality and in a standard case this functionality is directly located within the implementation of the Service Instance in form of a Service Logic. However, there are other possibilities, too. One of them is called a Service Broker (sometimes referred to as Technology Gateway Services ([50]). This kind of Service does not actually contain a Service Logic but accesses an external functionality whenever it is invoked. Introducing this Service type allows to integrate non-Service-based functionality into a Service-based application. Hereby, the Service Instance acts as a broker that offers and delivers the functionality. This allows for example to integrate legacy components into the system. Another possibility of using external functionality instead of a Service Logic is realized by Service Gateways. Just as the Service Brokers, these entities do not contain an actual Service Logic. The difference between these two types is that the functionality used by a Service Gateway is located in a different network not accessible by other Services of the system. In this sense, a Service Gateway does not only provide access to a non-Service-based functionality but also to entities not attached to the network.

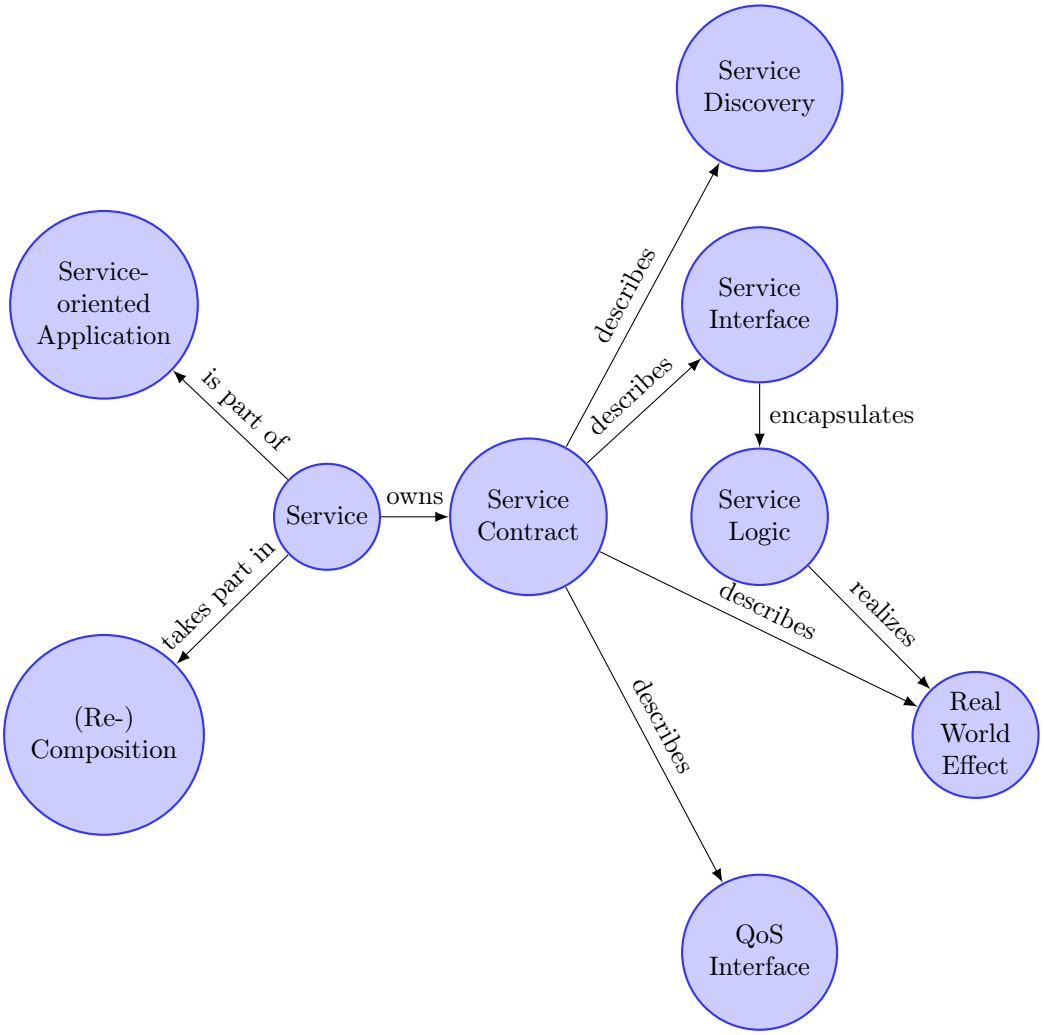


Figure 4.1: Relationship between the terms and definitions of SOA used in this work

As illustrated in Figure 4.1, each Service owns a Service Contract. This Service Contract describes how the functionality contained in the Service can be accessed by another unit. It describes the interfaces as well as the behavior of a Service in different scenarios of interaction. One of these scenarios is when a participant within a system searches for a functionality. In a Service-oriented approach this is carried out by the concept of Service Discovery. The main idea behind this principle is, that each functionality is search-able and detectable by sending out specific messages called Discovery Requests. If a Service Instance receives a message that contains a request to a functionality implemented in its Service Logic, it answers by sending a Discovery Response. This response is received by the original initiator which then can decide whether to use this Service Instance or not. Another Service-oriented concept described within the Service Contract is the Service Interface. It offers access to the actual functionality of a Service and thereby encapsulates its implementation called Service Logic. This Service Logic is responsible for generating a real world effect. This real world effect is also described in the Service Contract in form of the identification of the functionality encapsulated by the Service. A last Service-oriented concept used is the so called Quality of Service (QoS) Interface. The way of interacting with it is also described in the Service Contract. This QoS Interface offers access to the non-functional characteristics of a Service Instance that are decisive for the quality that can be offered.

As illustrated in Figure 4.1, a Service can be part of a Service-oriented application. Such an application can be described as a workflow combining several Services to create the overall functionality. Therefor, the concept of (re-)composition is used which discovers and selects a number of Service Instances to build the actual application. According to Wu et al. in [150], a Service-oriented application can be described by three different graphs:

- Service Function Graph
- Service Selection Graph
- Service Execution Graph

The Service Function Graph contains the abstract workflow of an application by describing what kind of functionality is needed and how the different functional components work together. Therefore, this graph contains a workflow consisting of Abstract Services. In the event of a (re-)composition, the system executes a Discovery for each of these Abstract Services. The result of this procedure is a list of available Service Instances. This list builds the so called Service Selection Graph a graph containing all possible configurations of the system regarding the currently available Service Instances. Within the process of Service selection this graph is analyzed and the optimal combination of Service Instances is determined. This selection procedure creates the Service Execution Graph which only contains those Service Instances that have been selected.

Regarding the (re-)composition procedure itself, Josuttis distinguishes between two different approaches (see [74]): orchestration and choreography. The former method is named after the way a orchestra is lead by its conductor. In this approach the conductor is able to overlook the whole scenario and guide the members of the orchestra. Applied to a Service-oriented composition procedure, this means that there is a central unit that has control over the whole process of Discovery and selection. Its counterpart is the so called choreography. Derived from the idea that trained dancers are able to carry out the movements within a choreography without being guided by a central instance, this approach bases on the autonomy of the individual Service Instances which compose the overall application in a collaborative manner. Hence, no central instance is needed to carry out this approach.

4.2.2 Goals of introducing Service-orientation into DDAS

The step from classic, static DAS towards Distributed Driver Assistance Systems raises several problems as described in section 1.2. The detection of these complications refers to the first process step of the Design Research methodology named Awareness of Problem (see Figure 1.3). In order to overcome these issues the principles of Service-orientation are to be used. This sets up distinct goals that have to be fulfilled by a framework tailored to DDAS. These goals are:

- Automatic re-configuration
- Visibility of the functionality
- Accessibility of the functionality
- Measurability
- Real world effect
- Location transparency

The goal to achieve an automatic online re-configuration is a central issue within a DDAS. It expresses the need of a mechanism that reacts to changes of the software and system architecture of the assistance system. This mechanism has to carry out its task self-contained and without the intervention of the assistance user or a developer. This demand sets up the need of a degree of autonomy, provided by the SODA framework, that realizes online re-configuration automatically.

The second goal defined called visibility indicates the need for each functionality to be detectable by other entities within the system. This is important, since the determination of what kind of sensors, actuators or data processing units are available is a basic precondition in order to be able to carry out automatic re-configuration. The visibility of a functionality within the system allows to detect the currently available entities. This information can then be used by the system to determine what kind of assistance system can be offered to the user.

Another goal to be fulfilled is the accessibility of each functionality. After detecting its presence it must be assured that it can be invoked and used by other units. This can be done for example by defining a strict interaction protocol or by making the behavior of the interface retrievable online.

Whenever a system needs to be re-configured at runtime several decisions have to be made. One of these decisions is the selection of a functionality whenever there is more than one entity available capable of offering the requested output. This sets up the goal, that each functionality should be measurable. In other words, another entity should be able to request information from a functionality that reflects its performance. By providing this information a functionality can be compared to other units which allows to improve the selection process within a re-configuration scenario.

Another goal that has been set up for the introduction of a framework capable of handling a DDAS is the demand for a real world effect. This means, that each entity on the system should actually produce some output that changes the way the overall assistance system behaves. In the case of a sensor this would be the determination of the current value for the dimension under observation. An actuator may physically change the state of the system during the execution of the functionality. A data processing unit may enhance the data collected with additional information or may use it to detect specific

states of the overall system by analyzing a potentially high number of data streams. While the kind of effect may vary in a wide range the framework must ensure that there is an actual effect of the functionality regarding the assistance system. However, since such a middleware is agnostic regarding the functionality entities attached to it, correctness is not within its scope.

The last goal presented here considers the reachability of a functionality. Despite the fact that an actual functionality is physically bound to the hardware unit hosting it, it should be reachable from anywhere within the system in the same way. This is ensured by defining the goal of location transparency. In other words, the way a functionality is invoked shall be independent of the physical location of the caller or the requested component. This demand is crucial when defining a system with a high degree of distribution as common in DDAS.

4.2.3 Concepts of Service-orientation used within SODA

In order to achieve the goals set up in subsection 4.2.2 the Service-oriented design paradigm has been adopted to the DDAS domain. However, the idea was not to simply adopt an existing framework to this new domain. This approach would not have respected the specific characteristics of automotive applications such as the usage of non IP-based network systems or the high cost pressure that calls for a minimized demand for resources. Instead, the SODA framework was created by analyzing the goals of the domain of DDAS. These goals were matched by Service-oriented concepts. While the following chapters within this work present the customization and implementation of these general concepts to this domain, Figure 4.2 illustrates the relationship between between goals, concepts and components within the SODA framework.

The first goal described in subsection 4.2.2 is to automatically re-configure the DDAS at runtime. The Service-oriented concept that is able to handle this issue is composition. Thereby, the system selects a group of Service Instances available at the very moment to build the assistance system. While in some implementations of Service-oriented Architecture, this composition is done at design time by a group of software engineers, in the domain of DDAS this has to be carried out automatically and online. The details of how this concept is applied within SODA is presented in chapter 6. One important requirement to be able to do automatic runtime re-composition is the existence of a mechanism that detects all currently available functionality. In Service-oriented Computing this mechanism is called Service Discovery. It supports the re-composition mechanism in providing all necessary information about what functionality in form of Service Instances is available at the very moment. At the same time, Service Discovery matches the goal of visibility. By being detectable through Service Discovery, all functional entities encapsulated in Service Instances are visible for the other participants within an application.

Another goal defined in subsection 4.2.2 is to make a functionality easily accessible. In Service-oriented Computing this is achieved by introducing two concepts: the Service Interface and the Service Contract. The former one is a well defined interface that encapsulates the Service Logic and offers standardized access to the outside world. The interaction between such a Service Interface is usually based on messages, so called Service calls. The interaction is hereby initiated by some entity that sends a Service request to a Service Instance. This Service request describes what functionality has to be executed. Furthermore, it may contain additional information such as parameters that influence the execution of the Service Logic or data that needs to be processed by the Service Instance. After the execution of the Service Logic the Service Instance answers

the Service request by sending a Service response to the calling instance. This Service response may contain a simple acknowledgment confirming that the Service Logic has been executed. In another possible case this message contains the result of the execution in form of a data section. One additional Service-oriented concept needed to achieve accessibility is the Service Contract. It describes the interaction protocol between the requesting entity and the Service Instance which ensures compatibility between both instances. Chapter 5 explains in detail how these two concepts are integrated into the SODA framework.

The fourth goal within the list calls for measurability. This is achieved by introducing a Quality Model that defines a Quality of Service (QoS) parameter to describe the performance of the entity in a standardized way. It is important since it allows to make decisions during the selection phase of a re-composition procedure which leads to generating the best solution currently available. The principles, structure and propagation mechanisms developed for the QoS implementation within the SODA framework are described in section 4.3. The goal of achieving a real world effect is realized by introducing the concept of the Service Logic. This entity contains the complete functionality of the Service Instance. It encapsulates it from the rest of the Service implementation and thereby achieves the separation of concerns. The Service Logic makes use of standardized interfaces to interact with other parts of the Service Instance which allows to easily re-locate it and simplifies collaborative development.

Finally, the last concept introduced in Figure 4.2 is Service-based addressing. Service-based addressing introduces a labeling system which allows to directly approach a Service Instance without any additional information about the hosting hardware. Such hardware independent addressing realizes the goal of location transparency as it is independent from the physical location of the functionality. The details on how this concept is implemented in the SODA framework are described in chapter 7.

4.2.4 A Reference Architecture for automotive SOA

After correlating Service-oriented concepts to the goals to be achieved by the SODA framework, the reference architecture of a SODA Service Instance can be defined. This is done by deriving software modules from the concepts introduced. Figure 4.2 illustrates the relationship between the goals defined, the concepts achieving them and the components implementing the concepts.

As illustrated in Figure 4.3, most of the implementations of the used concepts are assigned to the so called SOA Middleware layer. Hence, this part of the architecture is handling all Service-based mechanisms. However, there are some exceptions, too. One concept implementation that is not assigned to this layer is the Service Logic implementation. As this component contains the functional software it is assigned to the application layer. Below the SOA Middleware the Communication Model takes place. This entity implements the Service-based addressing. Besides the central addressing scheme module, it also hosts executable code for handling the adaptation of the Service Instance to the network system as well as a transport protocol that implements the segmentation and re-assembly of large packets and the transmission arbitration mechanisms. This Communication Model is explained in detail in chapter 7. Underneath this entity the Hardware Abstraction Layer (HAL) offers interfaces to access the hardware. This includes an interaction point to the communication hardware such for example CAN or LIN. It also contains access to the timing hardware and the input and output pins of the ECU. The HAL is also used by the components arranged besides the SOA Middleware and the Communication Model. One entity that belongs

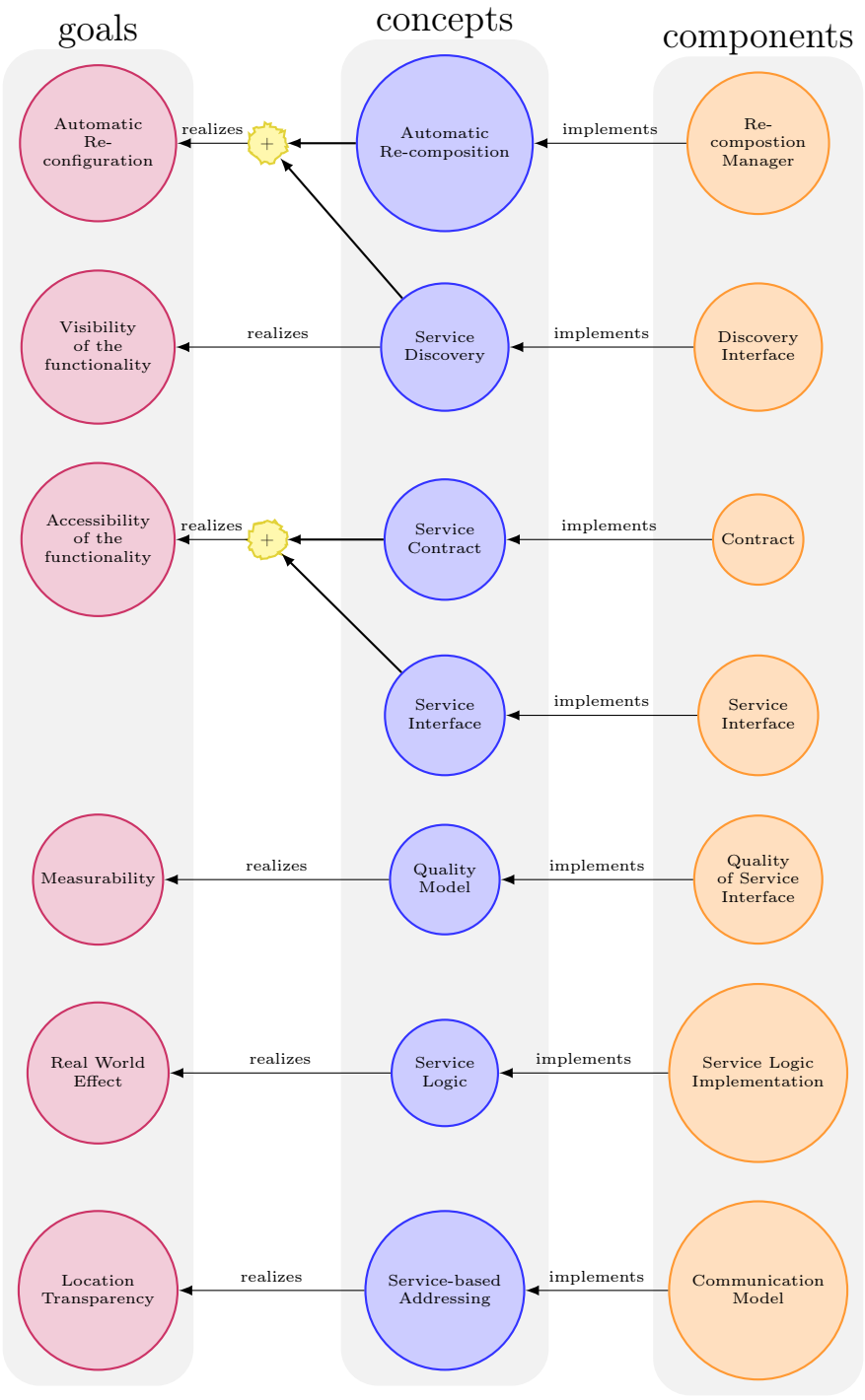


Figure 4.2: Relationship between goals, concepts and components

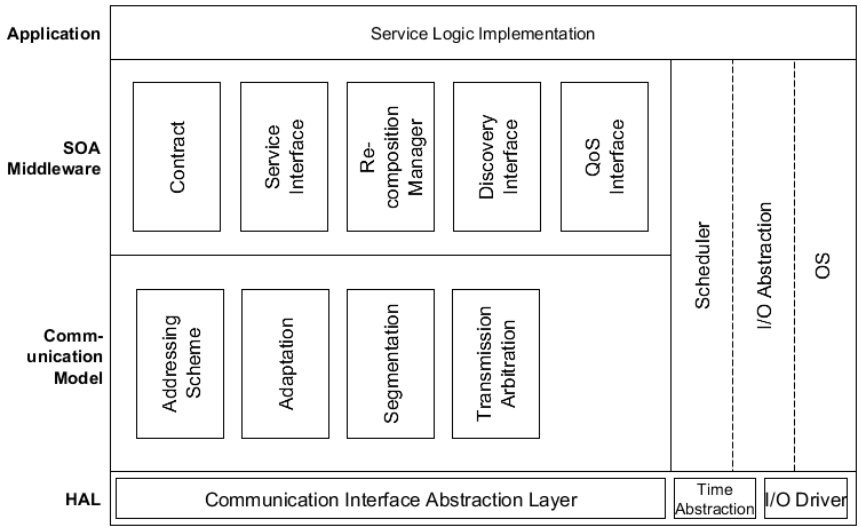


Figure 4.3: Overview of the SODA reference architecture.

to theses components is the Scheduler that controls the execution of the different software modules. A second one offers a handle to the input and output pins directly to the Service Logic implementation which allows to actually trigger hardware units by the software functionality encapsulated by the Service implementation. The third component assigned to this block is the operation system (OS).

As stated before, the SOA Middleware layer holds most of the components that implement the introduced Service-oriented concepts. One of these components is the so called re-composition manager. This software module is attached to each Service Instance and handles the re-configuration events that affect this instance. Hence, it implements the concept of automatic re-composition. As described in more detail in chapter 6, the re-composition manager is executed in every event of re-configuration. It makes use of specifically developed algorithms to compose an overall DDAS that provides the best possible quality. Another component assigned to this layer is the Discovery Interface. It contains the implementation of the concept of Service Discovery and generates as well as answers Discovery request messages. The component called Contract implements the mechanisms behind the Service-oriented concept of a Service Contract. Another component located here is the Service Interface. This entity contains all necessary code to access the functionality contained in the Service Logic implementation. In combination, the Contract and the Service Interface, are realizing the accessibility of the Service Instance from the outside world. Finally, the Quality of Service Interface implements the Quality Model used to satisfy the call for measurability. It provides an interface to retrieve the quality metrics of the Service Instance. Furthermore, it contains mechanisms to compute these metrics and to request it from other entities within the system. Section 4.3 explains this component as well as the mechanism it implements in more detail.

4.3 Quality Model

The Quality Model developed for the SODA framework describes the characteristics as well as the propagation and computing mechanisms used. In Service-oriented Architectures Quality of Service parameters are used to illustrate the performance of a Service Instance. While the functional aspects of two Service Instances of the same Abstract Service are identical, the quality they are able to provide may differ signifi-

cantly. However, as the system seeks to offer the best end-to-end performance possible it needs to be able to identify these quality differences of the Service Instances at runtime. This circumstances have brought up the usage of Quality of Service Interfaces. These interfaces provide mechanisms that allow an external entity to request the QoS parameters of a Service Instance.

4.3.1 The SODA Quality vector

When defining such QoS parameters the developer has to concern about several aspects. One of these is which metrics should be used to characterize the instance. These metrics may include timing issues, availability, accuracy besides many other possibilities. Another important decision is how many different aspects are to be incorporated when making a selection decision. Using several independent QoS parameters to decide on which Service Instance is the best one sets up a NP-hard problem (see e.g. [84],[8]). On the other hand, restricting it to only one characteristic may lead to Service Execution Graphs that suffer from quality issues within those metrics that have not been considered during the selection procedure.

In the SODA framework a combined approach is used. Here, a single quality vector $Q(s)$ is defined. This quality vector holds three components. The first one of these, named $Q_S(s)$, represents the safety and security characteristics of the Service Instance. These may include confidentiality, integrity, compliance to standards, safeguarding against failure and many more. The exact composition of the component is not defined by the SODA framework. Instead, it has to be defined whenever a new Abstract Service is modeled. All Service Instances implemented to match this Abstract Service have to ensure compliance to the composition defined here. This approach of leaving the exact definition open to the developer of the Abstract Service introduces a functionality-based scheme. For each functionality this component may look different while it has to be equal within every Service Instance implemented.

$$\vec{Q(s)} = \begin{pmatrix} Q_S(s) \\ Q_T(s) \\ Q_F(s) \end{pmatrix} \tag{4.1}$$

As illustrated in Equation (4.1) the second component of $Q(s)$ is named $Q_T(s)$. This parameter represents the timing characteristics of the Service Instance. The metrics used to assemble it may include response times, maximum or average age of data, cycle periods or other issues. Likewise to $Q_S(s)$, this component is also functionality-based as its exact composition is defined during the development of a new Abstract Service. This ensures flexibility which is important since for some Abstract Services such as sensors the age of the data requested is important while for other entities, such as data processing Services, it is more important how long the Service Instance needs to compute the desired effect.

The third component of the quality vector, named $Q_F(s)$, describes the functional performance of a Service Instance. This is again a parameter that strongly depends on the type of functionality that is encapsulated in the Service. As an example, for a sensor entity this component may particularly represent the accuracy or repeatability. As a sink Service such as a display may be characterized by completely different metrics such as resolution or refresh rate. In order to allow this high diversity, the $Q_F(s)$ component is also defined at design time of the Abstract Service rather than being fixed within the framework. All three Components of the quality vector have in common that lower

numbers mean higher quality.

4.3.2 Propagation and calculation of the QoS parameter

As stated earlier it is problematic to base a Service selection decision on a multitude of quality metrics. Besides the fact that there is no longer a clear decision to be made, the computing power needed to execute the selection algorithm may be quite exhausting. This leads to the conclusion, that the SODA quality vector, as it has been presented in Equation (4.1), needs to be simplified to a single number before it is handed over to the selection procedure. In the SODA framework this number is called the QoS parameter.

The QoS parameter is calculated by using the Simple Additive Weighting (SAW) procedure. This method multiplies each parameter that is input with a specific weighting factor before adding all results of this multiplication up. The sum computed hereby is then divided by the number of parameters used, resulting in an average value. Equation (4.2) illustrates the principle behind the SAW technique with weighting factors w and values v.

$$V_i = \frac{\sum_0^i w_j * v_{i,j}}{i} \text{ [67]} \tag{4.2}$$

This principle is adopted to the SODA framework. It is used to calculate the single quality parameter named QoS. By using SAW, the three components of the quality vector are combined and averaged to ensure simple calculations when during Service selection. Each of the three components is multiplied to a specific weighting factor. This weighting factor is decisive for the effect of each component on the resulting QoS parameter. Equation (4.3) illustrates the SAW procedure used within SODA.

$$QoS = \frac{w_S * Q_S(s) + w_T * Q_T(s) + w_F * Q_F(s)}{3} \tag{4.3}$$

The definition of the three weighting factors w_S, w_T and w_F are again not defined within the SODA framework. Just as in the case of the composition of the components of the quality vector, this would be quite restrictive. Instead, the definition of these weighting factors is delegated to the developer of the DDAS that is built using the SODA framework. This application-based approach ensures, that the selection process can be adjusted to meet the specific requirements of the driver assistance application developed. In the case of a DDAS directly interfering with safety critical parts of the car, the weighting factor w_S would be significantly high compared to w_T and w_F. In another example the system under development might be purely informing but requires a high responsiveness to meet the drivers needs. In this case w_T would be dominating to ensure composing a DDAS having a high user acceptance.

Since the weighting factors are not fixed within the SODA framework they have to be propagated at runtime. This is done by adding the values selected for w_S, w_T and w_F to the data section of the Discovery request message. Thereby each requesting entity can focus on specific aspects when searching for suitable Service Instances. The transmitted weighting factors are then used by those Service Instances matching the Discovery request to generate their QoS. If one of these Service Instances makes use of other entities itself, it forwards the weighting factors by sending out Discovery request messages containing these numbers on its own. Hereby, the values for w_S, w_T and w_F of an initial request are propagated to all members of the Service Selection Graph.

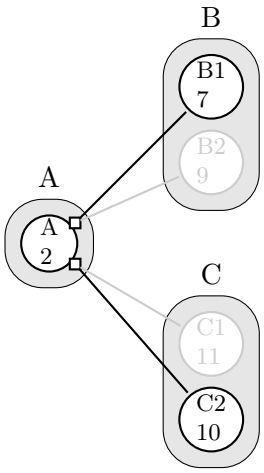


Figure 4.4: Example of a selection decision for a Service Instance having two Requested Interfaces

4.3.3 Calculation of the end-to-end QoS of an application

In order to achieve the best performance of an application, it is necessary to determine not only the QoS values of the Service Instances but also the end-to-end performance of the overall Service Execution Graph. Furthermore, the composition algorithm as it is described in chapter 6 is based on combined QoS parameters of sub-graphs of the application to execute the Service selection.

The end-to-end QoS calculation is based on two simple rules:

Rule 1. *For each requested Abstract Service, the Service Instance offering the lowest end-to-end QoS is selected.*

Rule 2. *The end-to-end QoS of a Service Instance is calculated by adding the highest QoS of the selected Service Instances to its own QoS.*

Rule 1 describes which Service Instance is chosen when several instances match the functional criteria of a requested functionality. It says, that whenever there is a selection to be made, the one Service Instance offering the lowest end-to-end QoS is selected. The graph given in Figure 4.4 illustrates this principle. The Service Instance that has to select other entities is named A. Its own computed QoS value is 2. Service Instance A owns two interfaces that make use of other Services. After executing the Service Selection, it identified two Service Instances for each of these interfaces. The first interface, that requests an Abstract Service called B, discovered the Service Instances B1 and B2. While the QoS value of B1 equals to 7, the one of B2 is 9. For the Abstract Service C the Service Instances C1 and C2 have been detected. These entities provide a QoS value of 11 and 10 respectively. As stated in rule 1, for each interface the Service Instance providing the lower QoS value is selected. Figure 4.4 indicates this decision using the grey out effect for B2 and C1.

The second rule describes the procedure when computing the end-to-end QoS value. It states, that in order to determine the overall QoS of a Service Instance, the entity has to add its own QoS value to the highest QoS within the instances selected for its interfaces. This is due to the fact that for all three components of the quality vector, a single instance of bad quality within the application workflow lowers the end-to-end quality significantly. For example, one entity within the workflow that suffers of long a response time would highly slow down the whole application. The same is true if the accuracy of data is considered. A single instance within the data processing flow adding noise to a data stream would add imprecision no matter how accurate the remaining

entities are. In the field of safety and security that one single weak point could screw up all effort that has been undertaken to safeguard the system. This rule can be again explained using Figure 4.4. After executing the selection as described earlier the Service Instances to be used by instance A are B1 and C2. B1 delivers its functionality with a QoS value of 7. The quality parameter reported for C2 is 10. Following rule 2, Service Instance A adds its own QoS, which is 2, to the one of C2. This calculation results in an end-to-end QoS value of 12. If Service Instance A is requested to publish its overall QoS, it will use a QoS value of 12 to indicate the quality of service it is able to deliver.

4.4 Summary

The SODA reference model as it has been described in this chapter defines the terms, concepts and components used within the framework. It also specifies their relations to one another and derives an architectural blueprint of a Service. In order to develop this model, several other models targeting at Service-oriented Architecture have been analyzed. These models are belonging to one of the categories phase models, system models, meta models, component models or reference models. It has been showed that reference models combine the different aspects included within the other models and combine them into a single structure. However, the reference models for SOA that have been examined within this chapter do not fit the needs of DDAS and the automotive domain. For this reason a unique reference model for the SODA framework has been developed. It is including all the terms, concepts and components used. At the same time it constraints itself to keep the structure tailored and avoid any overhead.

The development of the SODA reference model followed a clear and structured line of action. It started with the definition of the goals of the framework which are expressing the needs of Distributed Driver Assistance Systems. From these goals, Service-oriented concepts to achieve them were derived. In a further step components were defined. These components implement the concepts while keeping a clear structure and ensuring traceability. In a last development step, these components were arranged to build a reference architecture which serves as an architectural blueprint. An actual Service architecture can now be derived from this reference architecture using the process model described in chapter 5.

Finally, a quality model has been defined in this chapter. It respects the constraints of the automotive domain by generating a single QoS parameter. In order to achieve a multidimensional characteristic, this parameter is calculated from a quality vector using the SAW method. By leaving the exact composition open to the person specifying the Abstract Service, it can be adjusted according to the characteristics of the functionality. This approach ensures that the quality vector respects the varying properties of different entities such as sensors, output or processing devices. In order to be able to adjust the selection process to the needs of a specific application, the weighting factors used within the SAW calculation are not fixed by the SODA framework. Instead, they are propagated through the system using the data section of the Discovery request messages. Furthermore, rules are defined that manage the computation of end-to-end QoS parameters for an application or a subset. This fact simplifies the selection process described in chapter 6.

The developed SODA reference model described in this chapter is unique in its constitution and automotive focus. It defines and relates all necessary terms, concepts and components without creating the overhead caused by many other reference models in the literature. It ensures the achievement of the goals defined while respecting the specific needs and characteristics of the automotive domain.

5 Model-driven development of SOA-based DDAS[1]

'As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now that we have gigantic computers, programming has become a gigantic problem.'

Edsger W. Dijkstra[2]

5.1 Introduction

This chapter introduces SODAdev: a model-based development procedure to create SODA-based DDAS in a structured and straight-forward way. It is based on the SOMA methodology that has been introduced and described in chapter 3. Just as SOMA, it makes use of SoaML, a profile of the popular Unified Modeling Language (UML). However, as described in chapter 3, SOMA has several drawbacks that mainly arise from its focus on enterprise software applications. SODAdev refines the procedure in order to enable the specification of Distributed Driver Assistance Systems.

One important point is that SODAdev is restricting itself to the specification of the functional properties of the application under development. This leads to the fact, that only the phases "Business Modeling and Transformation", "Identification" and "Specification" are within its scope. Consequently, this chapter focuses on these three phases and does not discuss the other ones targeting either on administrative tasks or the non-functional requirements.

Furthermore, this chapter discusses how SODAdev can be integrated in an automotive development scenario. Therefore, its three phases are correlated to the different stages of the "core process for system and software development" (CPSSD, see [117]). Since SODAdev does not define any integration or testing stages the focus of this chapter lays on the left arm of CPSSD and leaves the right arm unchanged. Finally, the described SODAdev approach is used and evaluated in a case study that develops an actual DDAS.

5.2 SODAdev: Model-based development of SODA-based DDAS

In this section the SODAdev development process is described in detail. Furthermore, it is integrated into the CPSSD approach and thereby into one of the most used best practices in automotive development.

[1] This chapter is based on my publications [139] and [142]. Parts of it are extracted from these sources.
[2] 'The Humble Programmer, ACM Turing Award Leture 1972, see [43]

5.2.1 Application Level Design

The CPSSD approach of Schäuffele and Zurawka starts on the top-left corner of the V. This first development steps are named Application Level Design. This is because each CPSSD process starts with the definition of the properties of the overall application. In SODAdev the Application Level Design is done by carrying out the phases "Modeling and Transformation" as well as "Identification".

Phase 1: Modeling and Transformation As stated in chapter 3 the SOMA development approach starts with the phase "Business Modeling and Transformation". In this phase of the SOMA model the "Business Process Model and Notation" (BPMN) is used to identify tasks and parties within a business workflow. While this is a great method in the domain of enterprise software or Web Services it is not usable in the automotive domain without completely ignoring the semantics of BPMN. In this sense, it has to be replaced by a description method that allows such a specification. On the other hand the consistence and continuous flow of SODAdev should be maintained. This means that the resulting artifacts of each phase have to be directly usable for the next one. In the specific case of this first phase a replacement for BPMN and especially its "task" stereotypes that are originally used in the specification phase has to be found. In the newly created first phase of SODAdev, which is called "Modeling and Transformation" to emphasis that it is no longer focused on business process modeling, UML2 Activity Diagrams are used to create a first system description. Similar to BPMN models the workflows of applications can be described. On the other hand Activity Diagrams are not restricted to any specific domain. Furthermore, they are part of UML which allows seamless integration into SoaML without any kind of semantic violations. With the help of an Activity Diagram the idea for a DDAS can be modeled as a workflow consisting of a number of activities. These activities can be either executed in a sequence, in parallel or in a mixture of both modes. They are important because each of them represents one functionality of the DDAS to be developed. The overall Activity Diagram on the other hand describes how these activities cooperate to represent the DDAS. Figure 5.1 shows a simplified DDAS modeled as an Activity Diagram. The different functionality is arranged in a workflow to illustrate the mode of operation of the overall application.

It is expected that especially in this first phase of the development process the conducting team is very heterogeneous. This fact calls for a description model that is easy to understand even by team members that have no computer science background. In order to keep it simple the number of nodes used in this development step is restricted to the six entities pictured in Figure 5.1. The main element used here is the action node. It is the fundamental unit of executable functionality [100]. Illustrated as a rectangle with rounded corners it represents an operation were data is generated, processed or displayed. The edges connecting these actions are the second type of element used. They allow to demonstrate directed flows between the nodes within the diagram. The remaining four entities are control nodes that coordinate this flow. The first one is the initial node that symbolizes the beginning of the control or data flow after the system has been invoked. Its counterpart is the final node illustrating the end of the flow. The fork node and the join node allow to split the flow into multiple concurrent flows or synchronize them respectively. This set of nodes should be sufficient to model all needed application configurations in the scope of DDAS.

As already mentioned, the first phase within SODAdev is to transform an idea for a DDAS into a coarse-grained abstraction. In this sense it is the match to the specification of the Logical System Architecture (LSA) in CPSSD as shown in Figure 5.2. The Activity Diagram fulfills all demands of a LSA for being an abstract solution without any

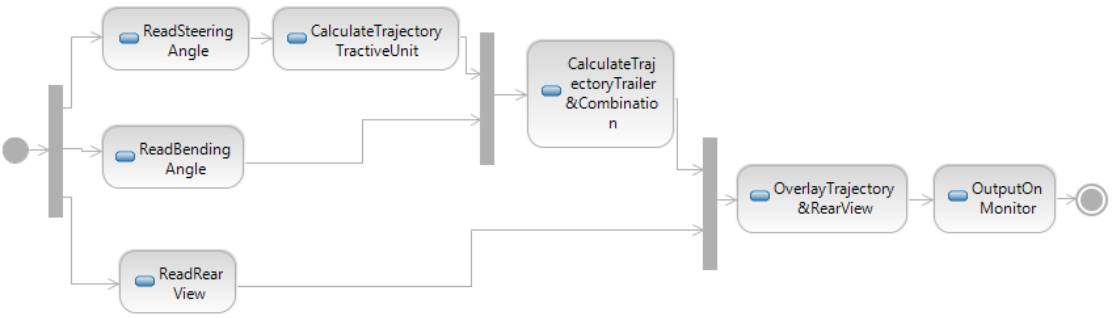


Figure 5.1: An example for a DDAS modelled as an Activity Diagram

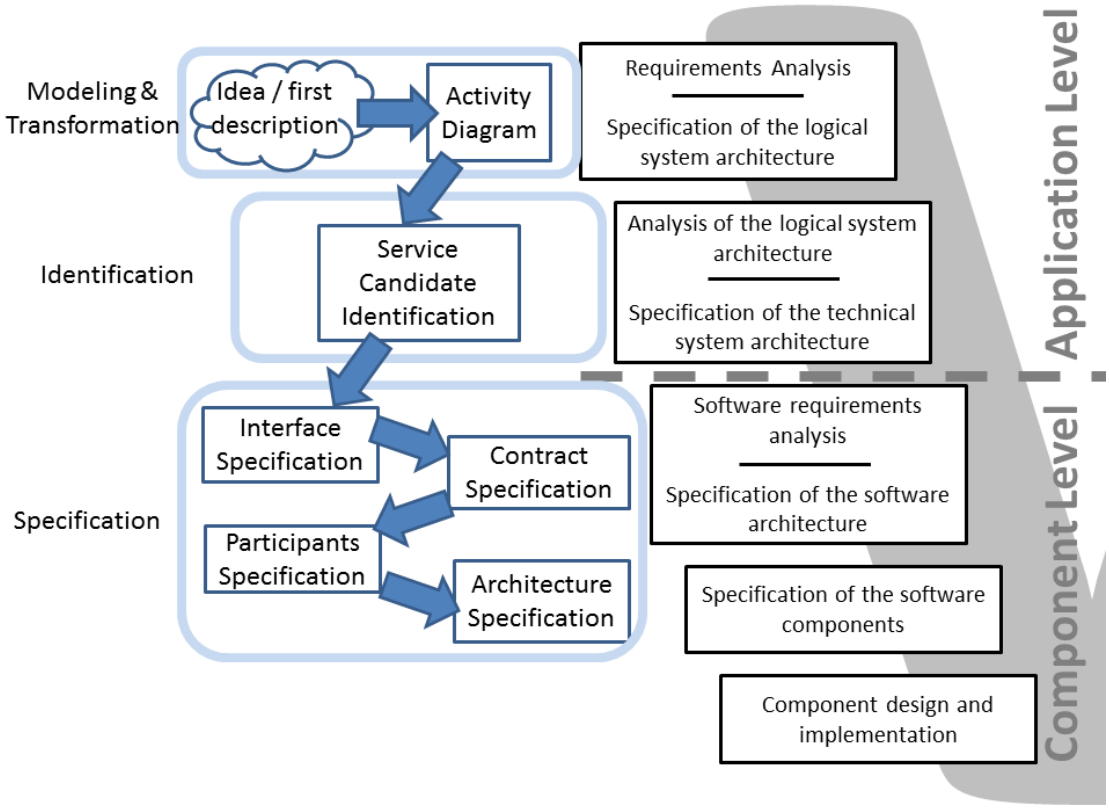


Figure 5.2: Integration of SODAdev into the V-model of CPSSD

technical details describing the components of an application, their functionality and their interfaces. In the automotive industry development teams are often very diverse being constituted for example of software engineers, electrical engineers or mechanical engineers. The usage of widely-known and almost self-explaining Activity Diagrams simplifies the work within such teams. Another important point is, that the initial idea for a DDAS can be described in a variety of ways. It ranges from natural language description to semi-formal or formal representations. Additionally, migration of an existing system into the SODA framework is possible. In this scenario code could be analyzed and converted into an Activity Diagram.

Phase 2: Identification This phase aims at dividing the overall system into small junks of functionality that eventually will become Services. It can be compared to defining the Technical System Architecture (TSA) in CPSSD as concrete decisions on how the system will be realized are made. In this step, the development team determines for example which Services will be needed and what functionality will be carried out in each of these entities. In this sense Identification phase is integrated into CPSSD as

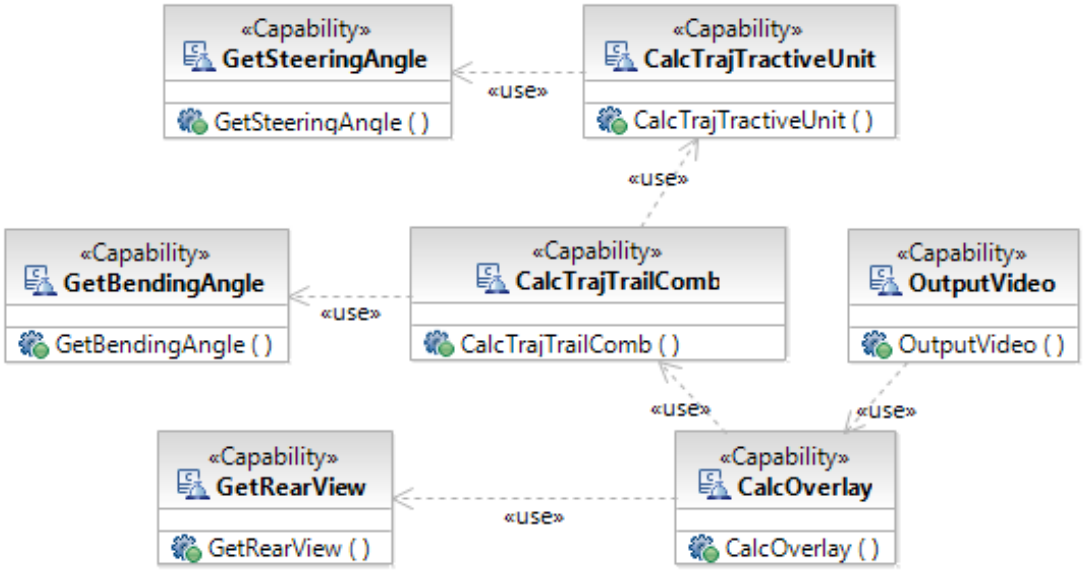


Figure 5.3: The Service Candidates derived from the example Activity Diagram

the definition of the Technical System Architecture as illustrated in Figure 5.2. SoaML defines a specialized stereotype for these Service Candidates called Capabilities. In the original SOMA development process this phase inspects the lanes and tasks of the BPMN model. Each lane, which represents some acting party, is directly transformed into a Capability. Afterwards, every task assigned to the specific lane is added to the corresponding Capability as a so called Operation. A SoaML Operation is what eventually will become a method in the implementation of the Service logic. The result of this procedure is a relatively coarse-grained model with a low number of Services potentially providing a big amount of functionality each. This leads to highly specialized Services tailored to the specific needs of the application under development. However, this specialization makes it difficult to re-use the Service in some other application. Furthermore those extensive Services which are quite demanding regarding computational power and memory requirements limit the possibilities when assigning them to ECUs in a distributed embedded system.

In order to overcome these drawbacks the architecture should be rather fine-grained with a relatively high number of Services. In the SODA framework this is brought to extremes by shrinking down each Service to offer only a single functionality. To achieve this, the Identification phase is modified. In SODAdev the Activity Diagram generated in the previous phase is analyzed and the UML 2 action nodes are extracted. In a second step, each of these action nodes is transformed into a single SoaML Capability. Subsequently, each Capability is enriched with one Operation that will provide the functionality of the Service. Coming back to the example Activity Diagram given earlier, the corresponding TSA is shown in Figure 5.3. This figure shows the seven Capabilities derived from the seven action nodes of the Activity Diagram as developed in phase 1. The Operations attached to these Capabilities reflect the specific functionality offered by each of them. Furthermore, the workflow of the system that has been defined in the Activity Diagram is transformed into usage relationships between the Capabilites. As an example, the Capability "CalcTrajTractiveUnit" is making use of the Capability "GetSteeringAngle". This relationship is directly derived from the fact that within the workflow given in Figure 5.1 the action node "ReadSteeringAngle" processes its data to the action node "CalculateTrajectoryTractiveUnit". The developed model of the Service candidates using the newly created SODAdev Identification phase is now very fine-grained. Each Service that is eventually derived from this descrip-

tion is more likely to be re-used in some other development project than the ones produced by SOMA. Besides, the engineers assigning these Services to ECUs have a high degree of freedom in doing so since the final implementation will be less demanding.

5.2.2 Component Level Design

By setting up the Capabilities model the Application Level Design is finished and the focus swaps towards the specification of the different Services in detail. This corresponds to the changeover between Application Level Design and Component Level Design in CPSSD.

Phase 3: Specification The third and last phase of the SODAdev process model is called Specification. It uses the Capabilities defined during Identification and transfers them through several steps into a full specification of the functional requirements of the system's Services. This transition from application development towards Service development equals to passing from application level into component level in CPSSD's "V". For that reason this phase is arranged as a replacement for the software architecture and software component specification as illustrated in Figure 5.2. Due to its extensiveness the Specification phase is split up into four sub-phases. The first one of these defines the ServiceInterface. It is followed by the Specification of the ServiceContracts. In a third step the so-called Participants are identified and generated. The final sub-phase brings all the Service specifications together to build the overall ServiceArchitecture.

Phase 3.1: ServiceInterface Specification In this first sub-phase a ServiceInterface is derived from every Capability defined in phase 2. In other words, every future Service gets a first representation in form of a ServiceInterface. The original SOMA process model recommends to specify a number of sub-interfaces to each ServiceInterface. These sub-interfaces are of the standard UML type "Interface". However, SOMA does not set up any rules or guidelines beyond this recommendation. The number and function of these entities is left unclear. This fact turns out to be problematic in the DDAS scenario. As the Services to be developed shall be used for runtime adaptation later on, it is essential to obtain a common structure within their SoaML specification.

To achieve this the process of ServiceInterface specification has been refined and stated more precisely. This is done by adding two additional guidelines:

- Transfer Operations into Provided Interfaces using UML Interfaces
- Add a Requested Interface for each external functionality needed using UML Interfaces

First of all, in SODAdev every Operation offered by a Capability is converted into an UML Interface. This approach leads to one Interface per Service which fulfills the demand of creating very fine-grained and easily reusable Services. The Interface derived from an Operation is called a Provided Interface as it reflects the specific functionality provided by the Service. With this step the functionality, that has been initially introduced as an action node in phase 1 and transferred into a Service Candidate in phase 2, is now attached building a Service. In order to use the specification for decentralized reconfiguration this information is not enough. Therefore, the second part of the refinement demands to generate one additional Interface for every external functionality needed by the Service to execute its task. These Interfaces are called Requested Interfaces as the Service itself requests their functionality. They are also

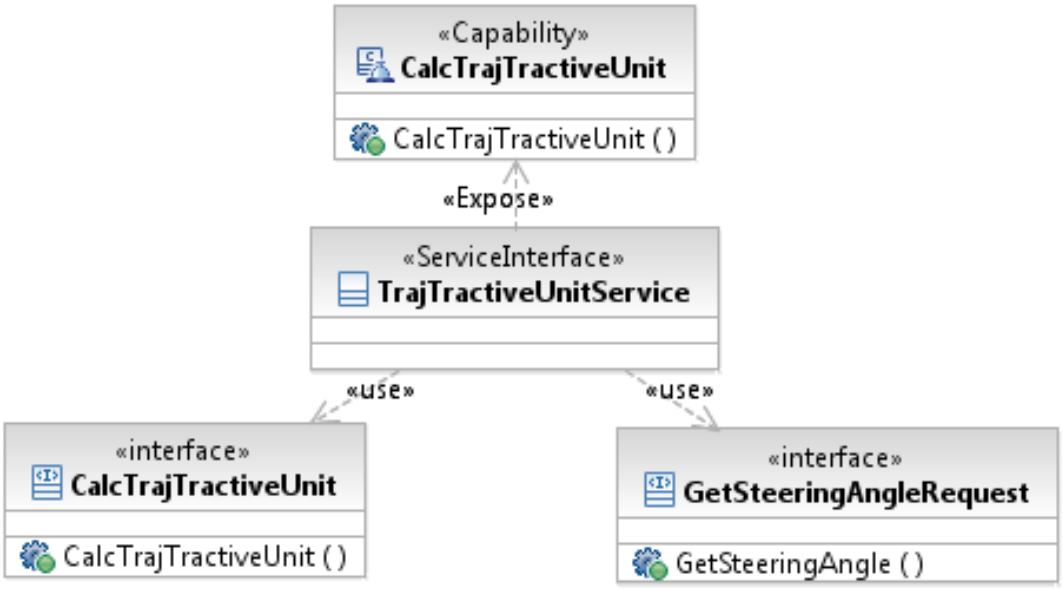


Figure 5.4: The ServiceInterface of one of the exemplary Services modeled using SoaML.

enriched with an Operation. This Operation can be seen as a classic get-method that is implemented to access the desired functionality from the Requested Service in form of a Service call. This extension within the SODAdev procedure is a crucial point regarding the runtime adaptability of the applications to be created. By adding this information each Service is now self-aware of its state. This is because, every Service does now have the knowledge of what other functionality in the form of Services has to be reachable. It can now explore its environment by executing a Service Discovery cycle calling specifically for implementations of the Services needed. If at least one implementation of each requested Service is currently available, the Service can put itself into the operational state. Furthermore, if there is more than one implementation available, it is able to decide which implementation will be used by requesting the quality parameters of each of these Services. In a next step the quality parameters of the selected implementations along with its own characteristics can be used to calculate an overall parameter. This overall parameter is given back to any request directed to this Service. In other words, the inclusion of information about the requested Services enables the SODA system to re-configure itself in a decentralized manner without the usage of a central configuration entity. The details of the Service selection procedure are presented in chapter 6.

An example of the procedures of the ServiceInterface Specification phase is given in Figure 5.4. As illustrated here, a SoaML ServiceInterface is derived from a Capability modeled in the last phase using the Exposed relationship that has been introduced in section 3.5. The offered functionality of the ServiceInterface is expressed by the Provided Interface on the left side of the picture. On the right hand side another Interface is provided. This one is a Requested Interface which determines that a Service called GetSteeringAngle is needed in order to run the Service's functionality. This configuration can be directly derived from the Capability diagram given in Figure 5.3. As this picture shows, the Capability "CalcTrajTractiveUnit", which is the basis for the "TrajTractiveUnitService" offers the functionality "CalcTrajTractiveUnit" which is now expressed by the Provided Interface "CalcTrajTractiveUnit". Furthermore, Figure 5.3 shows, that this Capability makes use of another Capability named "GetSteeringAngle". This circumstance is transferred into the Requested Interface "GetSteeringAngleRequest".

Phase 3.2: ServiceContract Specification As stated in chapter 3 SOMA makes use of the contract-based specification style. In SOA-based systems contracts formalize the exchange of information between the Service and the calling entity. Defining a contract means specifying two things:

- Specification of the roles within a ServiceContract
- Specification of the communication protocols

The first step is to determine the roles within the contract. Roles define which partners interact with one another when the Service is called. The second part of the contract specification is the definition of communication protocols. These protocols constitute the messages and the message sequences to access the Service and all its functionality. Again, SOMA is not very precise in this step. According to the SOMA specification the roles can be of any type that is allowed to be used within the SoaML specification. This opens up three options namely "ServiceInterface", "Interface" or "Class". In the case of diagrams to describe the communication protocols SOMA allows to use any adequate UML diagram. This lack of precision is understandable as SOMA is targeting on a wide audience and different fields of application. On the other hand precision is needed when the models are intended to be used for runtime adaptation. This is why SODAdev adds several constraints to the original SOMA process model:

- Usage of Sequence Charts to model communication scenarios
- Usage of a Remote Procedure Call style
- Usage of the UML Interface type to model roles
- Precise definition of the messages exchanged

As a first constraint UML Sequence Charts are chosen to illustrate the communication cycles. Sequence Charts are easily understandable regardless of being quite flexible. Additionally, they allow to add several extensions to the workflow as for example detailed descriptions of the messages to be exchanged. As a further constraint all communication is done in Remote Procedure Call (RPC) style. Compared to other SOA implementations such as Web Services where XML documents are exchanged, this method guarantees relatively low overheads. This is important because of the restricted transmission rates offered by today's automotive network systems. For the roles within the contracts Interfaces are used. This allows a more detailed description of the communication sequence as the actual Interface involved can be named rather than indicating the ServiceInterface which possibly combines several Interfaces. The last addition to the original SOMA approach is the extended definition of the messages exchanged between the roles of a contract. This information is needed later on to be able to define a tailored SODA middleware as described in more detail in chapter 7. The information needed here is especially the length of each message which can be determined by analyzing it's content. For this reason the UML Sequence Charts used to specify the interactions is extended. The exchange entities used are constrained to be of the Asynchronous Signal Message type defined by the UML specification. These entities allow to link each data exchange with an UML Signal. This Signal on the other hand can be defined in high detail by adding all parts of it alongside with their data type. In the SODAdev development process the Signals that are linked to the Asynchronous Signal Messages are defined within a special section of the SoaML model called "Signals and Datatypes". In the same section the data types used are described. Bringing all three parts together, the Sequence Charts with the Asynchronous Signal Messages, the Signal definitions and the data types, a complete specification of the communication behavior is created. An exemplary Sequence Chart is shown in the lower part of Figure

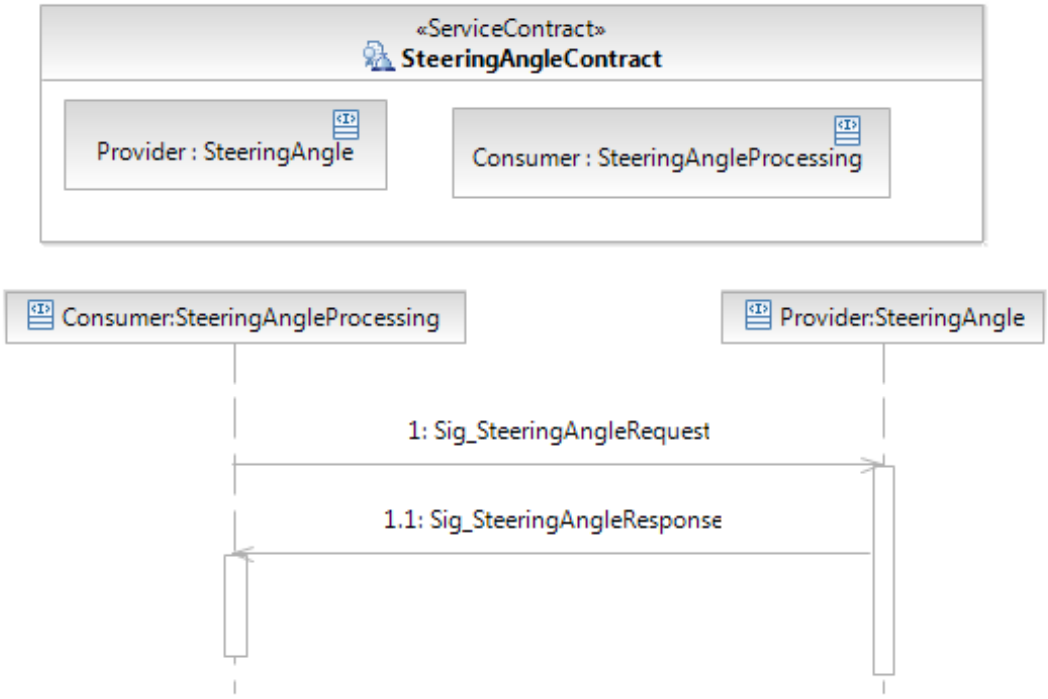


Figure 5.5: An example contract illustrating to roles and the Sequence Chart of the communication

Figure 5.6: The Signals and data types corresponding to the communication scenario illustrated in Figure 5.5

5.5 while Figure 5.6 illustrates the corresponding definitions of the Signals and used data types.

Besides these unavoidable extensions the Sequence Charts are kept short and thereby easily parsable. Therefore, the number of different entities to be used here is kept small. Lifelines are used to describe the interaction of the roles within the contract. The interaction is limited to the already introduced Asynchronous Signal Messages type. Using this small subset of entities it is possible to define event- and time-triggered Service invocations holding all information needed for the subsequent development steps.

Figure 5.5 illustrates a contract defined using the guidelines of SODAdev. In this simple example two roles are defined in form of Interfaces. These two roles are then used in the Sequence Chart to define the message interaction when calling the associated Service. In this example the Sequence Chart is enriched with information on the content of each message by using UML signals.

Phase 3.3: Participants Specification In the next sub-phase the so called Participants are specified. The definition of the stereotype Participant in SoaML is quiet vague. The

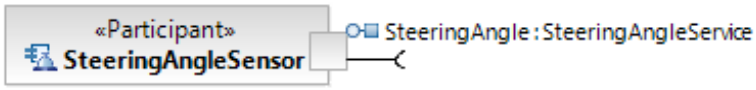


Figure 5.7: An example Participant with it's Service Point

SoaML specification [102] states that it may be a "person, organization, [...] system, application or component". The only qualification is that the entity has to be "a provider and/or consumer of Services". In other words, a Participant is some kind of entity that offers or uses Services. SOMA interprets this stereotype as some kind of computing unit that is able to execute the implementation of such a Service or at least to send a Service request call. In this sense it uses this sub-phase to map the different Services to hardware entities. In a Web Services scenario this may be the mapping of functionality to different servers. In the enterprise software domain such entities may be servers, work stations or even mobile computing units. In the SODA framework this kind of assignment of Services onto some hardware is not necessary as the addressing of the Services is not node-based. Instead a message-based addressing scheme is realized. For this reason the assignment of Services to specific computing units is not part of the specification as it does not have any effect on the functionality of the SODA application. Nevertheless, a model-driven development approach needs to be consistent. As the next step within the process model uses Participants they have to be introduced anyway. At this point, the broadly framed definition of the stereotype is used. In SODAdev Participants are defined to be an instantiated process. This agreement complies with the demand of the SoaML specification as these instantiated processes provide and use Services. At the same time they can still be assigned to any kind of hardware entity at a later stage of the development. This interpretation of the semantics behind the Participant entity does not influence the process flow within SODAdev. From a practical point of view, the modeling step is done exactly in the way that SOMA proposes. From a semantic point of view, this decision is quite far-reaching since it significantly enhances the flexibility of the approach.

Figure 5.7 illustrates such a Participant. The ServiceInterface is assigned to a Service Point pictured as a rectangle on the right hand side of the Participant. In this simple example the Service Point holds a single Provided Interface.

Phase 3.4: Architecture Specification In this last sub-phase the overall architecture of the SODA-based DDAS is defined. It is modeled using the SoaML ServiceArchitecture stereotype. As described in section 3.5 it illustrates the relationships of the Participants involved using their ports and contracts. In the first step of the architecture definition the involved functionality in form of Participants is selected. As the Services specified are very fine-grained and offer only one functionality each Participant holds only one Provided Interface within its port. In a second step the contracts corresponding to the Provided Interfaces of the Participants selected are added to the architecture description. As described earlier, these contracts specify roles that represent participating parties within a Service call. As a last step, the Interfaces within the ports are assigned to the roles of the contracts. With this step the interaction of the selected Participants is defined by the Sequence Charts given in the contract that connects them. Again, the difference between SOMA and SODAdev can be found in the semantics behind the modeling. While the practical operation of setting up a ServiceArchitecture is the same, the meaning of what kind of entities are connected is different since the semantics of the Participant stereotype is interpreted differently.

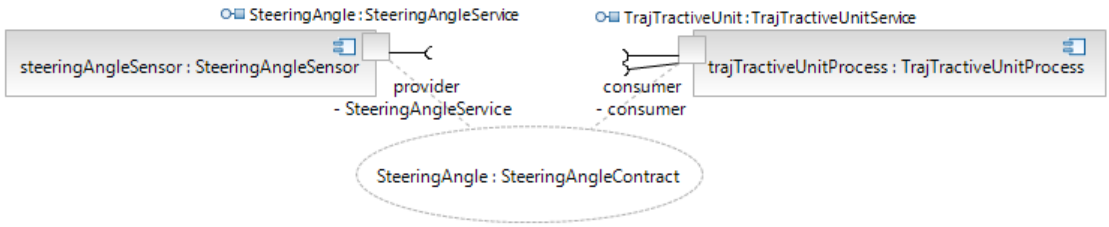


Figure 5.8: An excerpt from a example ServiceArchitecture

Figure 5.8 shows an excerpt of a ServiceArchitecture. In this simple example two Participants are selected. The one on the left hand side offers a Service needed by the one on the right hand side. Within the contract they are assigned to the role of a provider or a consumer respectively. The interaction between the two Participants is defined by the Sequence Charts specified in the contract connecting their ports.
The specification of the ServiceArchitecture completes the specification phase. The last phase of the left arm of CPSSD's "V" is called "Component design and implementation". One part of this is the design of the communication stack which is a central module in every middleware architecture. In SODA, the development of this module is supported by the SOAcom procedure that analyzes the SoaML model in order to define a tailored communication middleware. More precisely, it parses the xml documents containing the ServiceInterface, ServiceContract, Participants and ServiceArchitecture specification. It then determines a list of all Services within the application alongside with some information on the biggest message exchanged as well as the presence of periodic messages. This information is needed to design the communication stack. Please refer to chapter 7 for more details on this consecutive development procedure. Besides the communication module, this phase is to define some implementation details and to finally write code. In order to support the implementation the descriptions of the interfaces and the contracts are analyzed by the same program that collects information on the communication stack. The program goes through the xml documents and extracts the Requested and Provided Interfaces of the Service to be implemented. Furthermore, it parses the Sequence Charts of the contracts to determine the size and format of the data exchanged. Using this information a code skeleton is generated. This skeleton consists of a function body for every interface of the Service enriched with it's parameters. The type and name of the parameters are derived from the messages within the contracts. This automated generation of the code skeleton supports the developer by providing the fundamental structure of the program. However, as the SoaML model does not contain an internal program flow, it is still up to the developer to implement the code within the function bodies.

5.3 Case Study

The exemplary application for this case study with the SODA framework is a Distributed Driver Assistance system for truck and trailer combinations. More precisely the system to be developed using SODAdev is supporting the driver while backing up a two-axle trailer using the visual modality.

Figure 5.9 illustrates the principle structure of such a visual HMI for backing up trailers. A monitor presents a picture of the area behind the trailer to the driver. Furthermore several trajectories are computed and overlayed allowing the driver to predict the future path of the combination. The outer, blue trajectories symbolize the skid marks of the rear axle tires of the trailer assuming the bending angles between truck and trailer won't change. These lines inform the driver on the long-term behavior

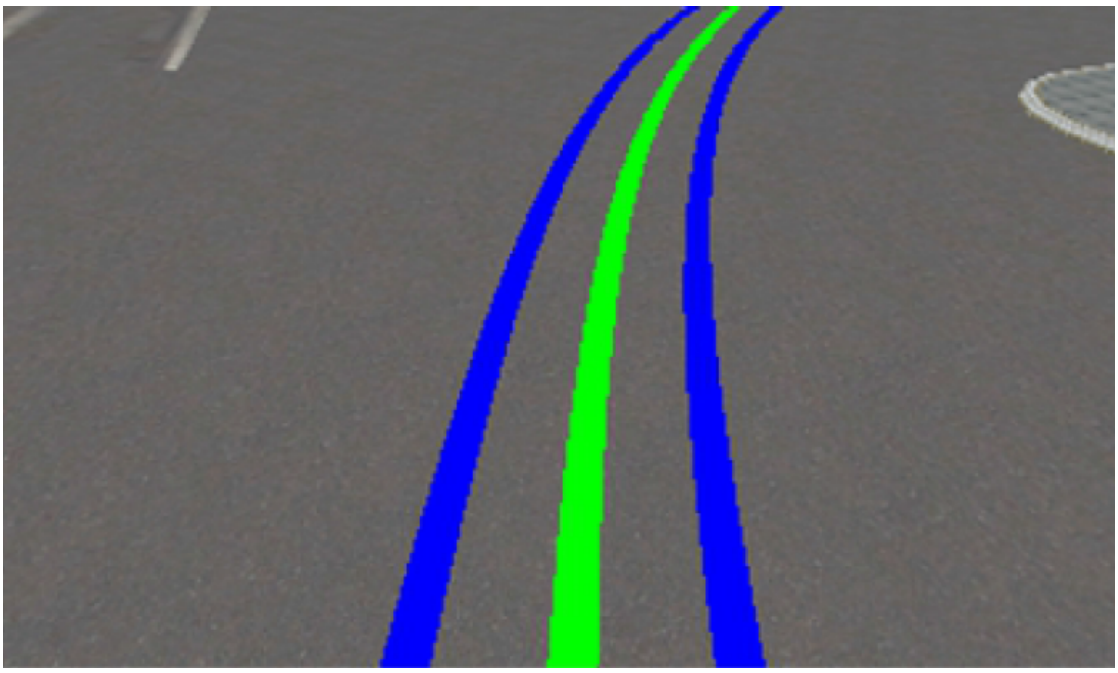


Figure 5.9: The HMI of a Visual Assistance System to back up a trailer [23]

of the vehicle. The inner, green line illustrates the future path of the center point of the trailer depending on the current steering angle. This curve allows the driver to make an assumption on the short-term behavior of the combination. It is directly responding to movements of the steering wheel. These trajectories are calculated using the steering angle as well as the two bending angles. Additionally, a number of dimensions of the truck and trailer are needed like for example the wheelbases. In order to offer the visual HMI as shown in Figure 5.9 a camera mounted to the back of the trailer and a monitor to output the video is needed. This case study will show how such a system is specified and developed using the SODAdev development process.

Phase 1: Modeling and Transformation The first phase is to develop the Logical System Architecture. In the case of SODAdev this artifact is modeled as an Activity Diagram. The basis for this case study is an implementation of the DDAS described earlier on a driving simulator (see [23]). The description of the functional properties as well as the code itself is used to define this first coarse-grained description. Figure 5.10 shows the Activity Diagram developed. The system is built by a combination of 13 functional entities. The ones in the column on the left side are either sensors or entities that offer information about physical dimensions of the truck and trailer combination. All the other Activities carry out some calculations on the basis of the data produced by some other one.

Phase 2: Identification In the second phase the future Services are identified. This is done by analyzing the Activity Diagram and deriving SoaML Capabilities. For the example application used in the case study, this leads to the model illustrated in Figure 5.11. As described earlier each Activity is converted into a Capability in order to develop lightweight Services. As a result each Capability only holds one Operation and thereby only implements a single functionality. The relationship between the Activities of the Logical System Architecture are transferred to the technical one by "use" relations connecting the Capabilities.

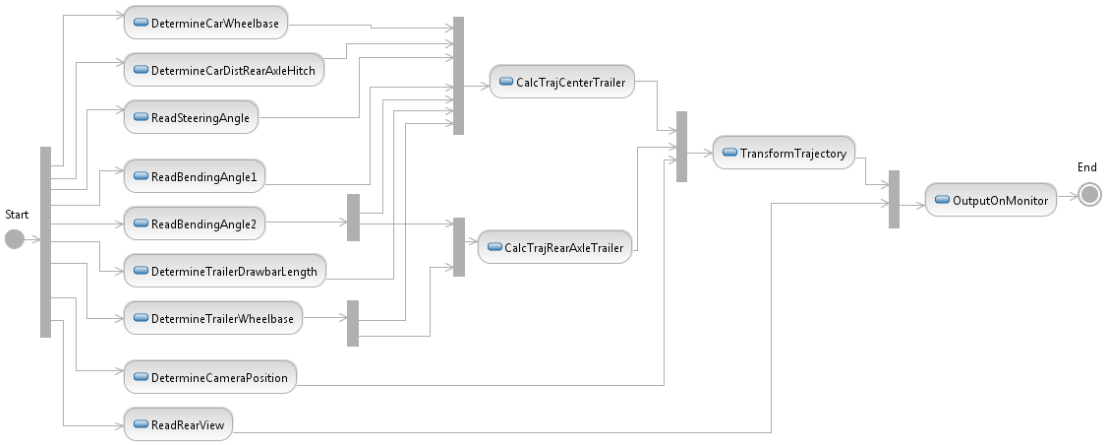


Figure 5.10: The Activity Diagram of the DDAS

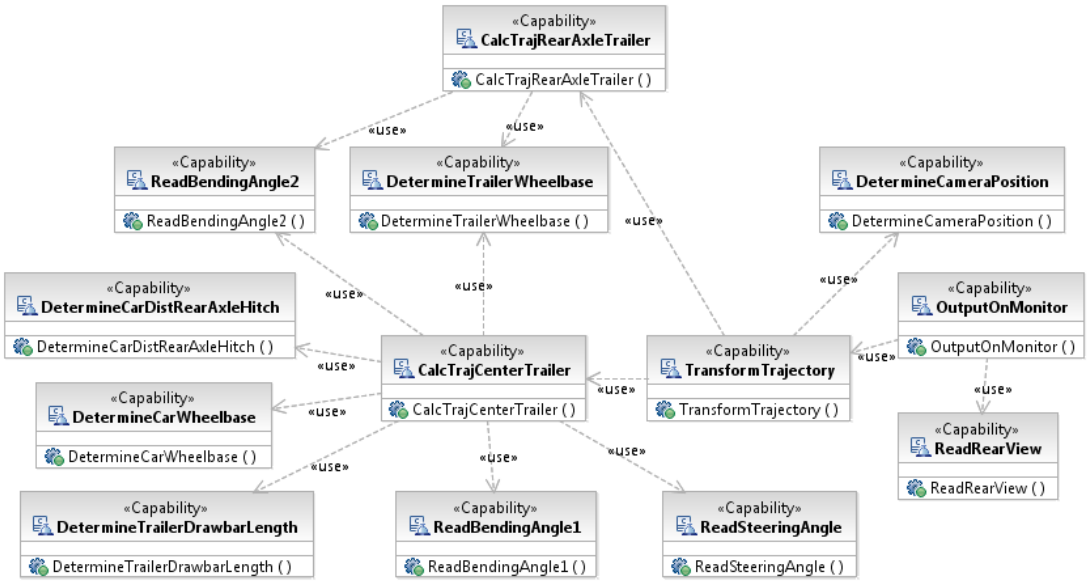


Figure 5.11: Overview of the Capabilities derived for the example application

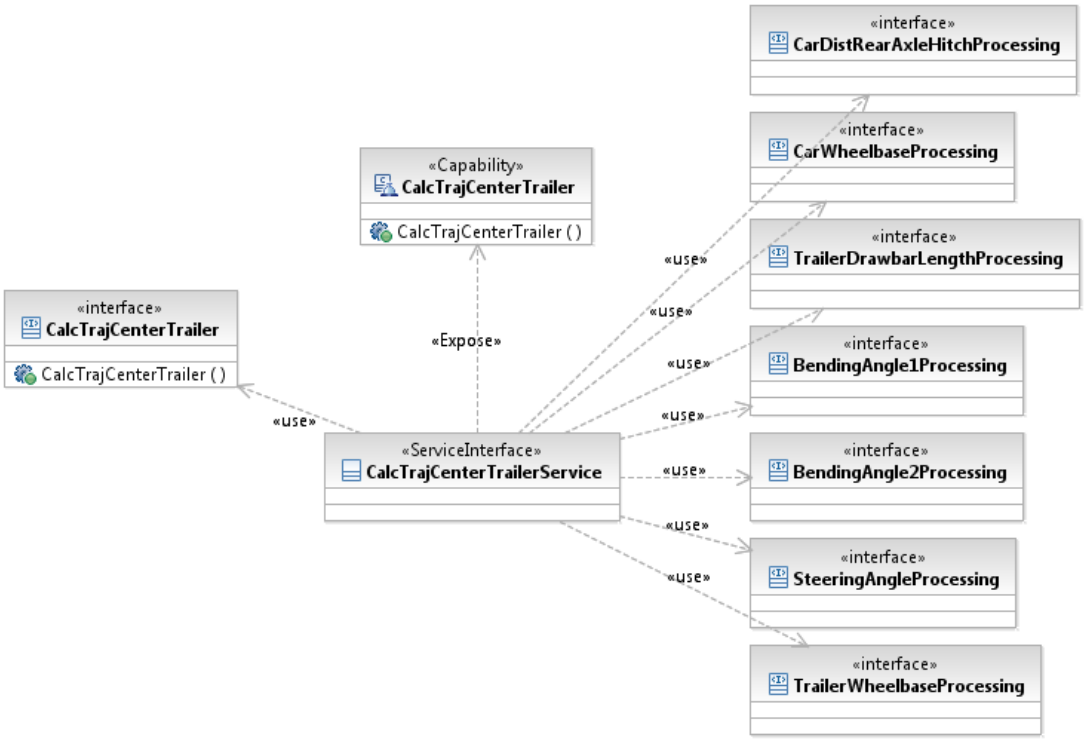


Figure 5.12: The ServiceInterface of the Service to calculate the trajectory of the center of the trailer

Phase 3: Specification Using the Technical System Architecture the Capabilities identified are picked up one by one and specified in-depth. This section focuses on one of the Services to be developed in order to obtain lucidity. The chosen Service is called "CalculateTrajectoryCenterTrailer". It uses a number of sensors and vehicle dimensions to predict the future path of the center point of the attached trailer.

The first step of this phase is to design the ServiceInterface of each Capability. This is done first of all by deriving one single ServiceInterface for each Capability. In a second step a Provided Interface to gain access to the functionality of the Service is created. This Interface holds the Operation actually carrying out the Service logic. The last step of the ServiceInterface specification is to create a Requested Interface for every functionality needed by the future Service to execute its logic.

Figure 5.12 shows the created ServiceInterface of the Service to calculate the trajectory of the trailer center point. In the middle the ServiceInterface is pictured. It is connected to one Provided Interface on the left and seven Requested Interfaces on the right side of the figure.

The next step of the Specification phase is to design the communication scenarios to access the Service by developing contracts. This is done by defining the roles of the communication scenario and the interaction between these roles.

Figure 5.13 pictures such a contract for the example Service. In the upper part of the figure two roles are shown, namely a provider and a consumer. The lower part illustrates the messages to be exchanged in order to access the Service. This contract is kept quite simple for several reasons. First of all the content of the messages exchanged is symbolized by a UML Signal specified in another part of the model. Second, only the scenario of calling the functionality of a Service is described here. All other interaction

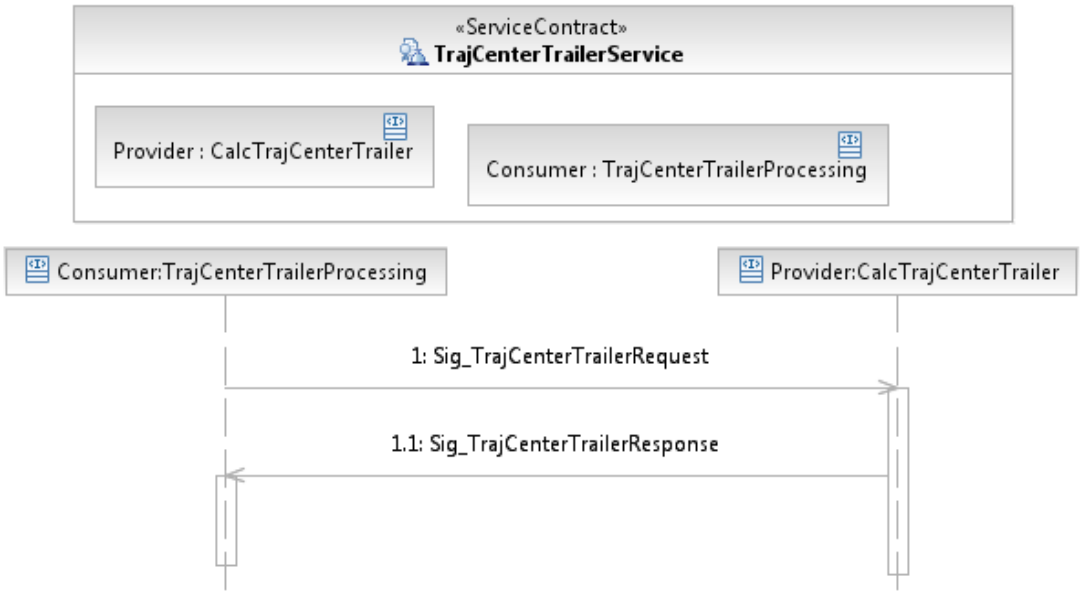


Figure 5.13: The contract of the Service to calculate the trajectory of the center of the trailer

Figure 5.14: The Participant of the example Service

scenarios such as for example Discovery Requests are standardized within the SODA framework. Because of this, they are defined centrally and do not have to be repeated in every single contract.

In the third part of the Specification phase the Service Candidates are converted into Participants. Each Participant is an entity enriched with a Service Point which holds all Interfaces of the Service Candidate. As explained earlier, this step is rather administrative. It does not add any additional information to the specification. But since the next step of the Specification phase uses these entities it is necessary to obtain the consistency of the process. Figure 5.14 illustrates the Participant developed for the Service to calculate one of the trajectories. To the left of the entity the Service Point is attached. The eight Interfaces of this Service are symbolized by the eight ports added to the Service Point.

In the last step of the Specification the overall ServiceArchitecture is defined. In the presented use case all the Services developed are brought together to build the driving assistance application. Therefore, the 13 Participants specified are added to the diagram. Furthermore the contract of each Service is added. In a last step the Participants are connected to each other through these contracts. The procedure of connecting two Participants equals the definition of a role binding within the connecting contract. Figure 5.15 shows the complete ServiceArchitecture of the specified application.

The last step of the left arm of CPSSD's "V" is to implement the specified Services.

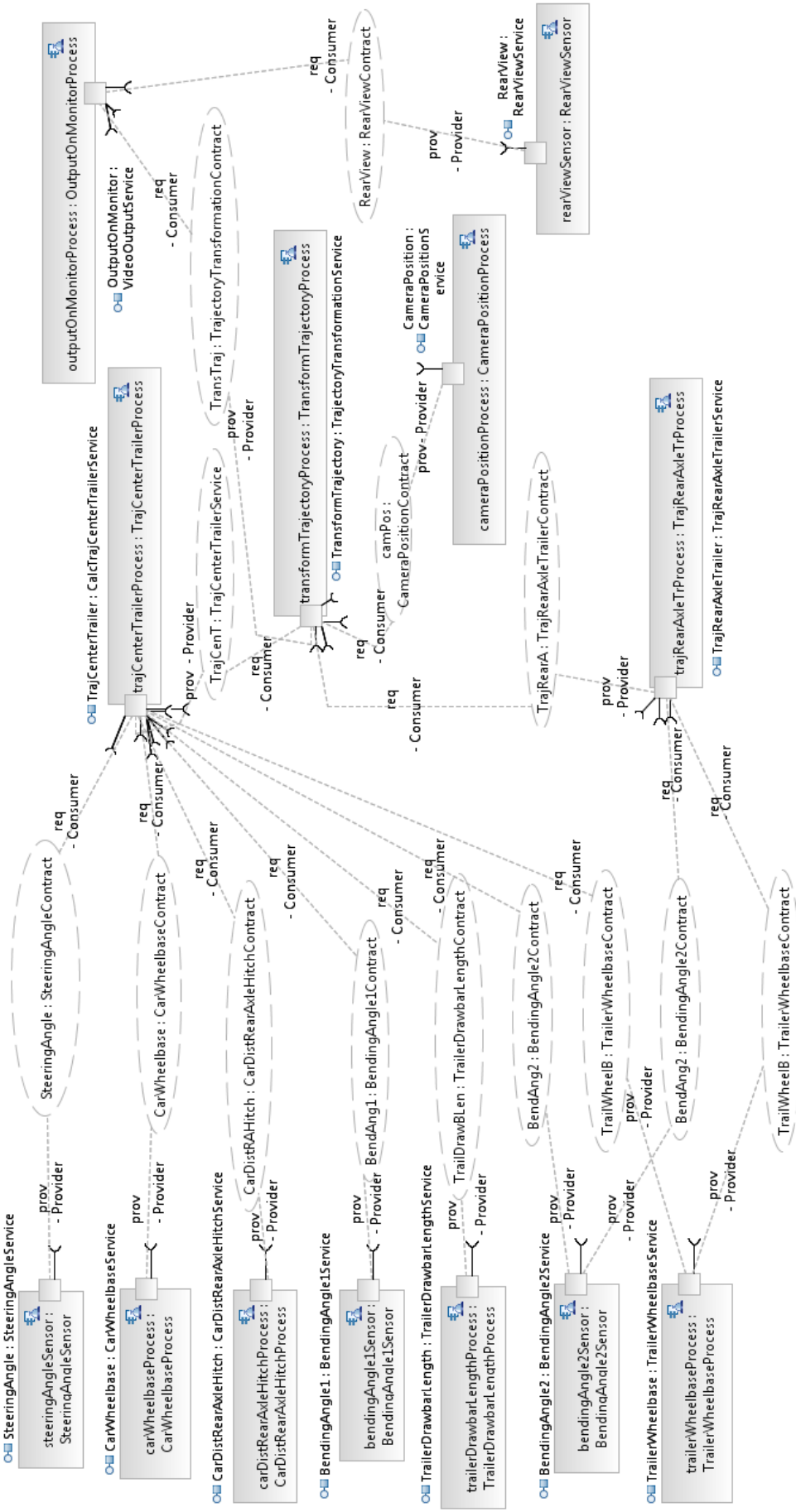


Figure 5.15: The overall ServiceArchitecture of the application

Figure 5.16: Picture of the demonstrator vehicle

This has been done using the SoaML model derived through the SODAdev process. Using the information extracted and the code skeleton generated by the automation tool 13 units have been implemented. These units were either Intel Atom boards running Lubuntu 12.04, Raspberry PI modules running Raspbian OS or small processor boards with an Atmel AT90CAN128 or an Atmel ATmega88 chip respectively. The units were mounted on a demonstrator vehicle consisting of a Mercedes B-Class car and a small two axle trailer as shown in Figure 5.16. Please refer to chapter 9 for more details on the implementation and the evaluation of the demonstrator system.

5.4 Summary

In this chapter SODAdev, a unique, model-driven development process for SOA-based Distributed Driver Assistance Systems, has been introduced. This procedure is based on the Service-Oriented Modeling and Architecture approach introduced by IBM to create Service-based enterprise software. From this approach SODAdev adopts its fundamental modeling language SoaML, a profile of OMG's UML standard.

By changing the starting description from using the BPMN notation to the much broader-focused UML Activity Diagrams, SODAdev enables the usage of such a process model for the DDAS domain. Besides, this modification did not interrupt the flow of the development process but allowed to use this first description within the subsequent phases as well.

In the second phase called Identification SODAdev restricts the size of each future Service to the lowest possible level by allowing only one functionality for each entity. This approach leads to small and easy to assign Services. Furthermore, this method increases the chance to re-use each Service in a different application.

Within the specification phase several refinements of the workflow were done in order to make it applicable to the DDAS domain. For example, the composition of the ServiceInterfaces was structured to get a clear pattern which contains all the information needed for executing Service selection. Moreover, ServiceContracts were defined very precisely in order to use them for communication analysis during the tailoring process of the Communication model. This includes the introduction of an additional section within the SoaML model that contains all UML Signals and data types used within the communication scenarios. Within the Participant specification the semantics of this

stereotype have been re-interpreted. While still compliant to the SoaML specification the interpretation done in SODAdev allows to keep the Services independent from any hardware unit. Finally, this newly interpreted Participants have been used to build the overall ServiceArchitecture of the application.

In order to show that this sequential model-based design approach can be used within an automotive development process, SODAdev has been integrated into the "core process for system and software development" (CPSSD). It has been proved that SODAdev fits well into the structure of CPSSD and its two layers namely Application Level and Component Level Design.

The overall SODAdev procedure has been evaluated by using it to develop a DDAS that assists the driver while backing up a two-axle trailer. This application was completely defined within the SoaML model. This model can now be used for further processing such as automatic code generation, formal validation or to extract specific properties.

6 Adaptation through Re-Composition[1]

'The measure of intelligence is the ability to change.'

Albert Einstein

6.1 Introduction

As stated earlier, the central point of future DDAS is that they need to be able to react to changes within the system at runtime. In this sense they need to adapt themselves whenever software or hardware components are added or removed. In Service-oriented systems adaptation is realized through re-composition of the Services within the application. The process of re-configuration is started by the Service Discovery. This operation determines all Service Instances currently available. In a next step it is decided whether all Abstract Services within the Service Function Graph of an application are matched by at least one Service Instance. If this is not the case the application can't be executed and the re-configuration process stops. Otherwise the composition procedure moves on by selecting one Service Instance for each Abstract Service of the Service Function Graph. This selection is based on the quality parameters of the Service Instances and targets on creating the best composition possible or at least on building a feasible solution. As a result a Service Execution Graph is generated that contains the new configuration of the Service-oriented application.

In this chapter the act of re-composition in SODA is described in detail. In section 6.2 the events that trigger re-configuration are described and analyzed. Section 6.3 discusses two different approaches of controlling this procedure while the sections 6.4 and 6.5 illustrate the concrete steps that are carried out to adapt the system to a new situation. Finally, section 6.6 summarizes the chapter and points out the contributions of this work to the research community.

6.2 Events of re-configuration in DDAS for truck and trailer combinations

In this section the possible events of system changes in truck and trailer combinations are discussed. The discussion is based on [145]. These events are the reason why adaptation is needed. The analysis of their characteristics allows to derive requirements regarding the re-composition procedure. In total six events that are critical to this domain have been identified. These events are:

1. Ignition on
2. Connecting a trailer at runtime
3. Disconnecting a trailer at runtime
4. Change of the type of assistance at runtime

[1] This chapter is based on my publication [143]. Parts of it are extracted from this source.

5. Change of the quality parameters at runtime
6. Failure of a Service Instance at runtime

The first event, called Ignition on takes place whenever the driver starts the vehicle. In this situation a re-composition has to be executed. This is the case because there might have been changes to the system during the time the vehicle was turned off. As the components of the DDAS can't spot such changes in this state the application has to be re-configured to guarantee its performance.

In the second event a trailer is connected to the pulling vehicle. This operation may add several components holding a multitude of Service Instances each. These Service Instances have to be discovered and taken into account when generating the Service Execution Graph for this new situation.

The counterpart of connecting a trailer is disconnecting one at runtime. This third event leads to a situation where there is no trailer connected to the pulling vehicle and hereby to terminating the assistance system. However, as there are no explicit sign off mechanisms defined in SODA it is up to the re-composition algorithm to detect the absence of a trailer.

The next event to look at is the change of the type of assistance at runtime. As Uwe Berg and Dieter Zöbel have shown in [20] there is a multitude of possible human machine interfaces that could be used to assist the driver while backing up. These interfaces might for example use the visual, auditory or tactile modality. From a technical point of view each of these variants is an independent application defined by its own workflow in the form of a Service Function Graph. In this sense, switching between the human machine interfaces executes the deactivation of the current DDAS and the activation of another one. To guarantee the performance of the newly selected system a re-composition takes place before starting it.

The fifth event to be discussed is the change of one or more quality parameters at runtime. One example for such a situation is the variation of response times to a Service call due to unbalanced utilization of the ECU executing it. It may also be possible that external influences affect the performance of a Service Instance. An example is an optical sensor which is often directly influenced by the amount of light currently available. Hence, changes in the amount of available light may directly alter the performance of the sensor both in precision and timing characteristics. Such changes of the performance of individual Service Instances might lead to a situation in which the currently selected composition is no longer the best one available. Only the re-composition of the application could assure optimality in such an event. However, these events might occur very frequently. A re-configuration of the system in each of these events may lead to a DDAS that is constantly re-configuring itself instead of providing the assistance to the driver. To avoid this SODA is not tracking the performance of the Service Instances continuously but periodically. This compromise ensures long-term performance and stability while avoiding frequent re-composition activities.

The last event that might occur is the failure of a Service Instance. In this situation there are three different sub-events:

- Failure of a Service Instance currently not used
- Failure of a Service Instance currently used with alternatives available
- Failure of a Service Instance currently used with no alternatives available

In the first situation the SODA framework does not recognize this circumstance until the next Service Discovery. As it does not affect the performance of the application currently executed this sub-event is not examined more closely. If a Service Instance fails that is currently used, the execution of the present composition is no longer possible. When there are other instances available matching the Abstract Service the re-configuration procedure is carried out to generate an adapted Service Execution Graph. In the case of the absence of any other Service Instance offering the needed functionality the application has to be shut down.

These six events, their characteristics and impacts on the execution of the application have been used to develop the re-composition approach used within the SODA framework.

6.3 Architecture-driven vs. Interface-driven adaptation

The following discussion is based on the ideas first published in [141]. There are basically two different approaches on how the adaptation procedure is supervised. In the first one a central unit that overlooks the whole system controls the process of re-configuration. This central unit has a global knowledge of the application as well as an extensive database of all Service Instances currently available including their functionality and quality characteristics. It is able to select appropriate Service Instances in order to build the best composition currently achievable. In the context of Service-oriented Architectures this approach refers to the term orchestration [74] as the re-composition is controlled by a central unit just like a conductor leads an orchestra.

In the SODA framework this approach is called Architecture-driven adaptation. This is because the specification of the Service Architecture as it is done within the SODAdev development process could act as the Service Function Graph which contains a global view of the application. Such a Service Architecture Specification does not only contain all Abstract Services needed but also how they are connected and what kind of messages they exchange. Due to the fact that the XML model of the Service Architecture is easily parsable it can be directly used to automatically generate data structures that represent it. With the help of this data structures a central re-orchestration of the application could be achieved quite easily.

But orchestration has some major drawbacks, especially in the domain of embedded automotive systems which SODA targets on. The first one is, that introducing a single central unit to control the adaptation procedure creates a single point of failure. If this component stops to work for any reason the system is no longer able to react to changes. Furthermore, this would set up the demand for a computing unit that does not deliver any functionality noticeable by the consumer. Besides, such a central composition unit would require a rather high amount of memory and computing power since it would be in charge for storing a complete Service Repository and would have to carry out the complete Discovery and Selection process. Finally, our target domain of truck and trailer combinations lacks of a system integrator, since truck, trailer and even bodywork are normally produced by different companies. This means that there is no single institution that decides on which supplier has to bring in the orchestration unit. As a result the system would have to face the possibility that either no such device is available or a multitude of orchestration units is present within the system. In the first case the application would not be configurable at all. In the latter case this circumstance would require additional protocols to allow a negotiation between the units in order to select a superior one among them. Besides, the resources of the orchestration units not

selected are wasted.

All these drawbacks call for the second approach which is based on decentralized re-configuration. Such a technique uses small and effective algorithms in each Service instance without the superior control of a central adaptation unit. In Service-oriented Computing this style refers to the term choreography [74]. Just like a group of dances that manage to perform their movements synchronously without the help of a leading person Services are able to compose an application without a central composition unit. The key to this capability is knowledge. Dancers are only able to perform in the right way if they know their individual step sequence. In this sense each Service Instance has to have some knowledge on what actions it should carry out in the event of re-configuration.

In SODA this knowledge is held within the Requested Interfaces of each Service. During the SODAdev development process every Abstract Service is enriched with information on its functionality modeled as a SoaML Provided Interface. With this level of self-awareness it is able to react to Discovery Requests. Furthermore the information on what external functionality is needed in order to provide its own service is held in the Requested Interfaces. In this sense, each Service is able to discover and select the external Service Instances needed by itself without having a full view of the application. In other words, the Service Function Graph splits up into pieces instead of existing in its entirety. In SODA this approach is called Interface-driven adaptation since all knowledge needed can be extracted from the interface descriptions of the involved Abstract Services.
This distributed way of adapting a driver assistance system solves the problems raised by the central approach. As each Service Instance comes with its own knowledge and composition component there is no single point of failure. The failure of one of these components would only cause the drop out of the dedicated Service Instance rather than the whole application. Besides, there is no extra unit needed which does only provide management functionalities not noticeable by the user. Furthermore, the requirements for memory and computing power on each Service Instance are quite low since the database and the selection algorithm only have a local view of their functional environment rather than having to handle the complete DDAS. In terms of the absence of a central integrator in the truck and trailer domain the distribution of the algorithm avoids the problem scenarios of having no or more than one adaptation unit which eliminates the need for a negotiation between these entities.

Taking all these facts into account the SODA framework has been equipped with an Interface-driven, choreography-style, distributed adaptation mechanism. This mechanism will be illustrated in full detail within the sections 6.4 and 6.5.

6.4 Phases of the re-composition procedure

The re-composition process that carries out the adaptation whenever a SODA-based system changes consists of four consecutive phases. These phases are illustrated in Figure 6.1.

The re-composition in SODA starts with the Discovery and Selection phase. During this step the Service Instances build a local Service Repository of potential partners and select one of these partners per Requested Interface. Afterwards each entity calculates its current quality parameter and responds to the initial Discovery Request. This procedure is presented in Figure 6.2. As SODA uses a distributed approach the Discovery Requests to the individual Abstract Services are sent in a sequential manner: The Sink Service starts to send out Discovery Requests for each of its Requested Interfaces. Every Service Instance that fits to one of these also starts to carry out the Discovery

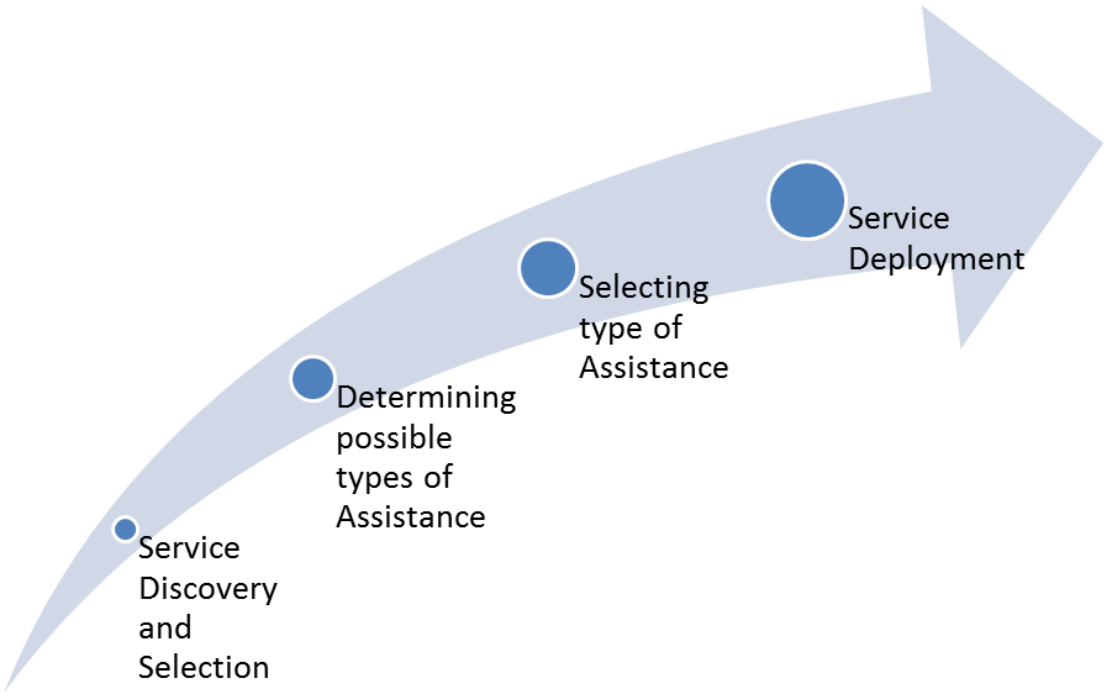


Figure 6.1: The four phases to re-compose a SODA-based DDAS.

and Selection phase and hereby triggers the execution of this phase within the group of Abstract Services requested by one of its own interfaces. This procedure is repeated until all Abstract Services of the Service Function Graph of the application have been invoked.

Having a closer look on the sequence of actions given in Figure 6.2 it can be seen that each Service Instance switches from idle state to Service Repository generation when a Discovery Request has been received. In SODA the Discovery Requests contain the weighting factors of the Quality of Service parameter. These weighting factors are received by the Service Instance and initially used to create the Discovery Request messages for its Requested Interfaces. The local Service Repository is then built by sending out these messages and collecting the responses. As a result every Service Instance generates a local list of potential Instances and their quality characteristics.

At this point a first decision can be made locally in every Service Instance. If there is not at least one candidate available for each Requested Interface the Discovery and Selection phase is stopped. In this case it switches back to the idle state without sending a response to the Discovery Request. The Service Instance hereby becomes invisible to the rest of the system. If there is at least one candidate per Requested Interface available the procedure goes on by switching to the re-choreography state. This step is necessary that early because of the way the QoS parameter is calculated. This single dimensional parameter is composed by the performance of the Service Instance itself as well as the quality of the available partners and the network used to access them. The second part of the calculation which refers to external functionalities can only be carried out when the selection among the candidates has already taken place. As the choreography process is a quite complex topic it is not described here in all details. Instead it is illustrated and discussed in section 6.5.

As soon as the selection process has finished, the QoS parameter can be determined. Therefore, the Quality Vector of the Service Instance is weighted using a Simple Additive Weighting (SAW) procedure. Furthermore it is combined with the QoS parameter of the selected instances matching the Requested Interfaces (for details see section 4.3).

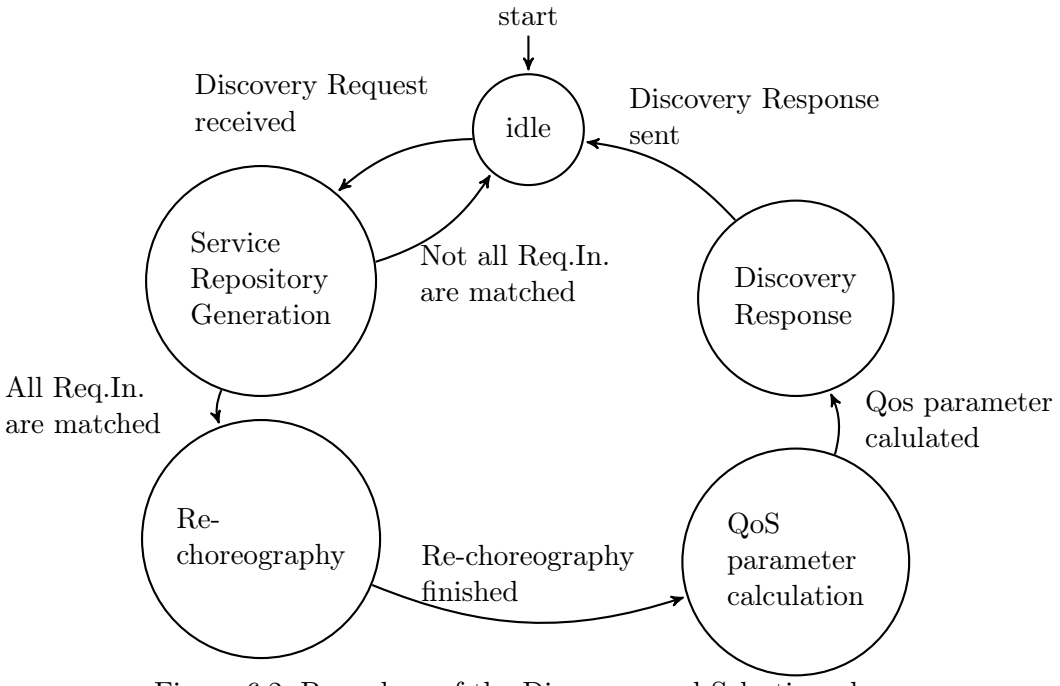


Figure 6.2: Procedure of the Discovery and Selection phase.

This step is crucial as it generates a value not only representing the performance of the Service Instance itself but reflecting the characteristics of the overall Composed Service. This fact allows to compose an application offering the best possible overall performance in a distributed way. As soon as the QoS parameter has been calculated the Discovery Response message is created and sent out on the network. This step completes the Discovery and Selection phase.

After the Discovery and Selection phase has finished the applications are either ready to run or not executable as not all Abstract Services were matched by a concrete Service Instance. The detection of these two possible states is done within the second step. This detection is possible through to the fact that every sink is implemented as a Service as well. Due to this the state of the overall application can be determined using the Discovery Interface of the Sink Service. As every application represents a possible type of assistance this procedure analyzes the availability of assistance systems. In the third step the type of assistance to be executed is selected. This can be done either manually or automatically. In the latter case a computer program makes the decision based on for example pre-configured priorities. A manual decision would be made by the driver himself supported by an appropriate human machine interface.

The execution of this decision is done within the last phase called Service deployment. Again the interfaces of the Sink Service are used to invoke the functionality of the application. This final phase of the procedure lasts until the system changes again and forces the application to re-configure itself.

6.5 Service Selection

In Service-oriented systems the problem of making a selection appears whenever there is more than one Service Instance available within at least one Abstract Service of the Service Function Graph. This selection could be done arbitrary since every possible composition is able to provide the requested output to the user. Then again if there is a number of possible configurations of a DDAS available one would want to make sure that the user's satisfaction is maximized. From a technical point of view this desire

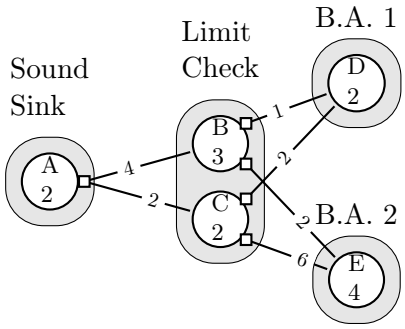


Figure 6.3: The Service Selection Graph of the Bending Angle Warning System

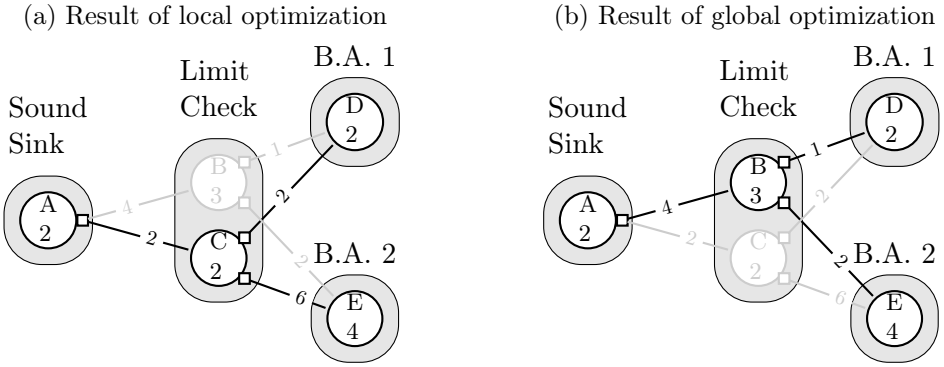


Figure 6.4: The different Service Execution Graphs for local and global optimization

sets up the the problem of finding the composition of Service Instances providing the best end-to-end quality characteristics. This is a well known, NP-hard problem (see e.g. [84],[8]).

One might say, that this problem can be solved easily by simply choosing the best available Service Instance for each Abstract Service. But research, like for example the one published in [8], has shown local selection strategies do not automatically generate the best results regarding the overall quality aspects of the application. One reason for this phenomenon is the influence of the network connections between the Service Instances. This effect can be nicely illustrated using the Bending Angle Warning System (BAWS) example. For the sake of simplicity it is assumed that only the Abstract Service Limit Check is matched by two possible candidates. For all other ones the current system only offers one possible candidate. The resulting Service Selection Graph is illustrated in Figure 6.3.

Using local optimization the system would choose to use Service Instance C to execute the functionality needed since its QoS parameter is lower than the one of its rival instance B. This would result into an overall quality measure of 16. Even if this algorithm would be extended to also take the quality parameter of the local network connections into account the resulting configuration would be the same. However, if the Service Selection Graph is analyzed completely it is to be noted that there is a better composition available offering an overall QoS value of 15. Both Service Execution Graphs are presented in Figure 6.4.

In recent years a multitude of selection algorithms has been introduced to the scientific community. Section 6.5.1 discusses some of these regarding the special demands within this domain of application. Based on the results of this analysis the algorithm developed for the SODA framework is presented in section 6.5.2.

6.5.1 State-of-the-art in Service composition algorithms

The problem of selecting a configuration of Service Instances has been extensively discussed in recent years. The number of published approaches is that high, that even very detailed surveys like for example the one of Strunk [127] are presenting only a subset of these. This discussion of state-of-the-art follows the same idea. Hence, it is not meant to be understood as a complete overview of the approaches published but as a classification of algorithms for Service composition. Furthermore it discusses the presented approaches in the context of the special requirements of Distributed Driver Assistance Systems.

The four requirements to the composition algorithm in the SODA framework are:

1. Re-configuration at runtime
2. Determination of the optimal solution
3. Executing a choreography-based approach
4. Ensure low resource consumption

The first one of these requirements expresses the need for runtime re-configuration. Although this seems to be obvious a number of approaches completely focuses on design-time composition. However, since SODA has to be able to react to system changes at runtime a static composition approach is not satisfying.

The second demand to the composition algorithm for SODA is that it has to be able to find the configuration with the best end-to-end quality value currently executable. This is important since the driver assistance seeks to offer the best performance possible.

Another requirement that is set up by the SODA framework is that the composition algorithm should work in a distributed choreography manner rather than being controlled by a central instance. The reasons therefor are discussed in section 6.3.

Finally, the fact that the algorithm will be executed at runtime on rather small embedded computing units calls for low resource consumption. The discussed approaches will be examined in particular concerning the used memory, processor and network loads.

The algorithms to be investigated are all listed in Table 6.1. The first group within this list accumulates the static composition approaches. Amsden proposes a procedure in [6]. It uses a distributed model-driven approach to generate a Service Execution Graph offering the best QoS available. It is quite close to the ideas of Mayer et al. in [93]. Here, a UML extension is described that enables design-time orchestration using the Business Process Execution Language (BPEL). Both approaches are not examined regarding their resource consumption as this is not a concern when executing the composition at design-time. Although both processes guarantee to generate the best configuration possible they are not taken into account since it is not possible to use them for runtime re-choreography.

A large group of composition algorithms makes use of heuristics to simplify the selection procedure. The general idea of these approaches is to find a configuration that matches a certain level of quality often defined using Service Level Agreements (SLA). The first one to be discussed within this category is the propose of Mohabey et al. in [97]. In this publication a central computing unit acts as an auctioneer that helps finding Service Instances performing at a pre-defined level. However, this approach is based on a central unit and causes a relatively high amount of network load through

Publication	Runtime composition	Optimal solution	Distributed manner	Low resource consumption	Approach
Amsden [6]	✗	✓	✓	-	Design-time composition
Mayer et al. [93]	✗	✓	✗	-	
Mohabey et al. [97]	✓	✗	✗	✗	Heuristics
Li et al. [83]	✓	✗	✗	-	
Yu et al. [153]	✓	✗	✗	✗	
Yuan & Liu [154]	✓	✗	✓	-	
Cardellini et al. [31]	✓	✗	✗	-	
Garcia Valls & Basanta Val [56]	✓	✗	✗	✓	
Liu et al. [85]	✓	✗	✗	✗	Genetic Algorithm
Chang & Wu [34]	✓	✗	✗	✗	
Wu et al. [150]	✓	✗	✗	✗	
Qiqing et al. [107]	✓	✗	✗	✗	
Tao et al. [129]	✓	✗	✗	✗	
Grossmann et al. [63]	✓	✗	✗	✓	Local Optimization
Zeng & Benatallah (local opt.) [157]	✓	✗	✗	✓	
Zeng & Benatallah (global opt.) [157]	✓	✓	✗	✗	Integer Programming
Ardagna & Pernici [7]	✓	✓	✗	✗	
Wan et al. [147]	✓	✗	✗	✗	Divide & Conquer
Huang et al. [70]	✓	✓	✗	✓	Dynamic Programming
Gao et al. [54]	✓	✓	✗	✓	
Aiello et al. [5]	✓	✓	✗	✗	Graph-based
Li et al. [82]	✓	✓	✗	✗	

Table 6.1: Comparison of different selection algorithms

using standards like the Simple Object Access Protocol (SOAP) that exchanges XML or CSV documents instead of messages. Another algorithm is presented by Li et al. in [83]. This procedure replaces Service Instances within a Service Execution Graph until all critical levels of quality are reached. However, it is also controlled by a single instance and the publication lacks of any technical details that would allow to estimate resource consumption. Yu et al. describe a broker-based approach in [153]. Using this method a central entity called broker is responsible for all adjustments at runtime including Service Discovery and Selection. The goal of this broker is to preserve a certain level of end-to-end performance. Just like the propose of Mohabey et al. it suffers from the fact of depending on a single unit that controls the adaptation. Furthermore it also uses heavy-weight protocols like SOAP which decreases the efficiency of the system. Yuan and Liu interpret the Service Selection problem as a modified version of a routing scenario ([154]). The great benefit of their approach is the usage of a distributed algorithm to avoid a single point of failure. However, it limits the number of examined paths by using two different kinds of heuristics. Hereby it can't guarantee to compose the best Service Execution Graph currently available. Furthermore, since no technical details of the implementation are given, the resource consumption can't be evaluated. The approach presented by Cardellini et al. in [31] makes use of SLAs. These contracts set up margins which the selection algorithm tries to match. In the case of the propose of Cardellini et al. this is done using a central broker architecture. Again, as technical details of the implementation are missing the resource consumption can't be ranked. In [56] García Valls and Basanta Val present an algorithm explicitly targeting on embedded real-time systems. In addition to the classic selection criteria it also takes schedulability into account. Through being developed to be used within embedded systems it is very careful in using system resources. Nevertheless it is based on a central entity that overlooks the whole system causing all the problems stated earlier.

In general one can observe that using any kind of heuristic to simplify the composition process results in not being able to guarantee the generation of the best Service Execution Graph currently available. As a result this category of algorithms might be very interesting in scenarios where the number of Abstract Services and/or the number of available Service Instances is very high. In DDAS these numbers are expected to be rather modest. This fact questions the benefits of the simplification of the selection process compared to the potentially major consequences of not executing the composition with the highest performance measure.

The next group of approaches uses genetic algorithms to select Service Instances among each other. Liu et al. for example use an evolutionary algorithm that is inspired by the ideas of natural selection, crossovers and mutations in [85]. The same concepts are applied by Chang and Wu in [34]. Their propose repeats the evolutionary cycle until a certain criteria has been met. Wu et al. have enhanced this method in [150] to speed up the convergence towards the optimal configuration. In order to do so the algorithm starts with an enhanced initial population and uses improved procedures for the selection and mutation processes. Tao et al. pushed the usage of evolutionary algorithms to its extreme in [129]. In this approach the main evolutionary algorithm is accompanied by several so called island models. These island models are evolution systems with a lower population to get a faster generation of extreme configurations. The models run in parallel to the main algorithm to get faster results in case the optimal solution can be found in a rather extreme configuration. The last approach that is examined within the category of genetic algorithms is based on the ant colony optimization algorithm originally published by Dorigo and Di Caro in [40]. The basic idea of the algorithm is to search for a minimum cost path in a graph using artificial ants. These ants mark the path they take using pheromones. Over time the concentration of these pheromones on

the elements of the graph allows to draw conclusions on the optimal path. Qiqing et al. uses this principle in [107] to find a path within the Service Selection Graph of a composition problem.

All these approaches allow to carry out Service Selection in Service-oriented systems at runtime. However, they all face the same problems regarding the requirements set up earlier in this section. None of them allow choreography since they are all controlled by a central instance or need to have a complete overview of the system. Furthermore, none of them is able to guarantee the generation of the optimal Service Execution Graph. But the biggest issue regarding these methods is the high demand for computing power and memory. Just as the algorithms using heuristics they are only interesting for large-scale systems where executing a configuration that meets a certain quality level is sufficient rather than finding the best solution available.

Another group of composition approaches is based on local optimization. Grossmann et al. present an algorithm in [63] that is based on abstract models generated within a model-driven development procedure. The information stored in these models is used to select the Service Instances in the case of runtime re-configuration locally for every single Abstract Service. This simplification decreases the complexity of the selection algorithm which makes the propose interesting for being used in embedded systems. Nevertheless, the approach is controlled by a central instance instead of using a distributed mechanism. Another algorithm based on local selection is presented by Zeng and Benatallah in [157]. The authors make use of a central Service Repository as well as a central Service Composition Manager to locally select a single Service Instance for each Abstract Service of the Service Function Graph. This is done by weighting different quality aspects and then choosing the instance offering the best performance. It is the nature of local selection approaches that they use relatively simple algorithms that are characterized by low resource requirements. Nevertheless, all these approaches do not fulfill the requirement of composing the configuration with the best end-to-end performance.

Other approaches use the principles of linear programming to solve the selection problem. The basic idea of linear programming is that a set of variables, an objective function and a set of constraints is given and an algorithm tries to optimize the objective function by varying the values of the variables while enforcing the constraints. Mapped to the selection procedure a boolean variable for every Service Instance is defined which holds a true if the entity is used and a false otherwise. The overall quality function is used as an objective function and additional constraints are used for example to hinder the algorithms to select more than one Service Instance per Abstract Service. These techniques are used by Zeng and Benatallah in a second algorithm described in [157] and by Ardagna and Pernici in [7]. Both approaches are able to find an optimal configuration at runtime. However, both of them are quite exhaustive and controlled by a central instance. In doing so both do not fit the profile of requirements.

Wan et al. present another technique for handling Service selection in [147]. The basic idea is to split up the overall Service Function Graph into smaller parts and optimize them separately. This divide and conquer strategy simplifies the selection processes within the subgraphs and hereby lowers their complexity. However, since all calculations are done in a single computing unit the overall complexity does not change significantly. Additionally this approach is somehow similar to local optimization as it only guarantees the best selection locally instead of globally.

Another group of algorithms picks up classic graph-based techniques. Li et al. for

example use a Dijkstra algorithm to find the shortest path through the graph and hereby the optimal configuration in [82]. Aiello et al. published a propose in [5] that executes an extended breadth first search with priority queues. All approaches manage to find the best solutions currently available. Nevertheless, they are quite exhausting. Especially the mechanisms used by Aiello et al. may potentially create huge queues and therefore use a high amount of memory. Furthermore they both need a supervising unit that overlooks the whole graph in order to carry out the adaptation procedure.

The last group of approaches to be discussed uses dynamic programming. This method generally solves complex problems by dividing it into smaller subproblems. These subproblems are then solved independently from one another and the subsolutions are combined in a way that the overall optimal solution is reached. Although being very close to the technique of divide and conquer, dynamic programming features two main benefits. Since the combination of the subsolutions is also subject of an optimization procedure end-to-end optimization can be carried out. Furthermore dynamic programming stores and re-uses the subsolutions to calculate them only once per algorithm execution. In the use case of composition algorithms there is another difference to point out. While divide and conquer tries to divide the Service Function Graph, dynamic programming splits up the Service Selection Graph of the application. The technique of dynamic programming has been picked up for example by Huang et al. in [70]. The algorithm presented here divides the Service Selection Graph into smaller chunks that are solved separately. After finishing this procedure the subsolutions serve as an input for a backward search algorithm that computes the best overall composition. The propose of Gao et al. in [54] makes use of a detailed 3-layer model of the Service Selection Graph to divide it into subgraphs, solve those smaller problems and create the overall solution. Both approaches suffer from the fact that they use a central instance that overlooks and controls the whole system. Nevertheless, the principle of dynamic programming has high potential for being used within the SODA framework. It allows runtime re-configuration of the system. It is also able to find the composition offering the best end-to-end quality value. All this is done using a rather modest amount of computing power, memory and network load. Furthermore, as dynamic programming splits up the problem into smaller subproblems, the algorithms to solve them are numerous but rather simple. This fact meets the nature of DDAS as an aggregation of small, distributed, embedded devices. If the problem of being centrally controlled, that the example approaches are faced with, could be solved this technique would fulfill all demands set up by the application domain.

6.5.2 The Service composition algorithm in SODA

As the discussion of the state-of-the-art in Service composition algorithms has shown, the technique of dynamic programming is a promising approach. This is because it allows to build algorithms that find the best composition currently available. Through to its solution strategy that involves dividing the overall problem into smaller subproblems and the reuse of the calculated subsolutions it makes a contribution to create a resource-friendly algorithm. This is important since SODA has to carry out re-composition at runtime using embedded devices. The only open issue in the approaches of Huang et al. and Gao et al. was the usage of a central instance which creates a single point of failure.

For these reasons the composition algorithm designed for SODA picks up the solution strategy of dynamic programming and combines it with a distributed control mechanism. Looking at the structure of algorithms using dynamic programming one can identify three phases:

1. Division of the problem into subproblems
2. Calculation of the subsolution for each subproblem

3. Combination of the subsolutions to achieve the overall solution

During the first phase the overall Service Selection Graph is divided into smaller subgraphs which represent smaller parts of the problem. Hereby, the goal is to find a level of fragmentation that generates subproblems easy to solve. As there is no central unit nor a complete picture of the graph at a single location in SODA the decision on how the division is carried out must be made in a distributed manner.

There are basically two potential approaches to do so. The first one is based on negotiations between the Service Instances. Using this technique an overall view of the graph is generated by merging the local knowledge of the instances. On the basis of this complete Service Selection Graph the problem is divided into subproblems. In a next step these subproblems are assigned to different instances to resolve them. All decisions made are based on negotiations between the Service Instances currently available.

A second approach to carry out the subproblem generation is the explicit specification of cutting lines at design time. This technique does neither require an overview of the system nor any negotiations or calculations at runtime to reach a decision. Due to these benefits, explicit cutting lines are used in the SODA framework. In order to achieve a complexity close to the excellent values of the local composition algorithms the subproblems are limited to one Service Instance and its direct neighbors in form of the candidates to its Requested Interfaces.

The small size of the subproblems simplifies the second step, which carries out the calculation of the subsolutions. It is reduced to selecting the best subsolution candidate offered at the moment. This can be done very easily by going through the possibilities given for a Requested Interface and choosing the one with the best QoS among these. Hereby another characteristic of dynamic programming is used. As soon as a Service Instance calculated the solution for its subproblem it stores it for later use. In the case of a second request for the subsolution the saved result can be used without repeating the calculation.

The third step within a dynamic programming algorithm is to combine the subsolutions in such a way that the best overall solution is found. In the algorithm used in the SODA framework this is automatically done through to the fact that the subprobelms are solved consecutively. Hereby, each subsolution is calculated on the base of the solution of the minor subsolutions calculated before.

The algorithm carrying out this choreography based on dynamic programming is illustrated in Algortihm 1. The presented method *ServiceAssignment* is part of the implementation of every SODA Service. It is called whenever the Service Instance receives a re-composition request. The weighting factors needed to compute the overall QoS are part of this request and handed over to the method through its parameters. In a first step the algorithm checks whether the Service Instance has any Requested Interfaces. If this is not the case, there is nothing to be composed by this Service Instance. The only tasks to carry out are to calculate its own QoS using the weighting factors and to respond to the initial request by sending out this value. These steps are executed in line 2 and 3. If the Service Instance owns at least one Requested Interface not the own QoS but the solution of the subproblem built by selecting a candidate and calculating the overall QoS has to be returned. In the case this subsolution has already been generated, it is load from memory and sent out as shown in line 6 of the algorithm. If this is the first request the lines 9 to 22 are executed. It starts with the calculation of its own QoS. In a next step all Requested Interfaces are inspected. In order to do so a loop counts from 0 to the overall number of Requested Interfaces (#RIF). Within each

Algorithm 1 ServiceAssignment(uint8_t weightingFactors[ ])

```
 1: if Service has no requested interface then
 2:    OwnQoS = CalcOwnQoS(weightingFactors)
 3:    Send a QoS response using OwnQoS
 4:    return true
 5: else if OverallQoS has been already calculated then
 6:    Send a QoS response using OverallQoS
 7:    return true
 8: else
 9:    OwnQoS = CalcOwnQoS(weightingFactors)
10:    for all RequestedInterfaces i such that 0 ≤ i ≤ #RIF do
11:       for all AvailableServices j such that 0 ≤ j ≤ #AS do
12:          QoScurrent = OwnQoS + requestQoSofAvailableService(i, j) +
             requestQoSofConnection(i, j)
13:          if QoScurrent < QoSminimal then
14:             QoSminimal = QoScurrent
15:             Set j as assigned Service
16:          end if
17:       end for
18:       if QoSOverall < QoSminimal then
19:          QoSOverall = QoSminimal
20:       end if
21:    end for
22:    Send a QoS response using OverallQoS
23: end if
24: return true;
```

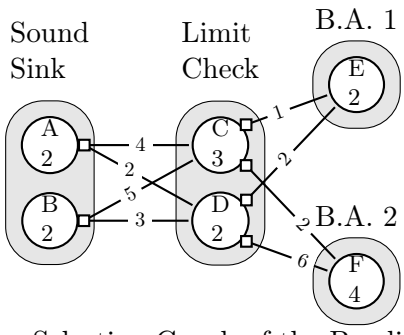


Figure 6.5: A simple Service Selection Graph of the Bending Angle Warning System

cycle the candidates of the respecting Requested Interface are investigated by running a loop from 0 to the overall number of available Service Instances (#AS). In each cycle the QoS of the corresponding candidate, which represents the solution of its subgraph, is requested. The received value is then added to the quality measure of the connection and the Service Instance's own QoS (line 12). In each cycle the solution currently determined is compared to the best solution found so far. In case the new one is better, its QoS is saved and the corresponding candidate is selected (lines 13-16). Finally, the overall QoS of the Service Instance is determined by choosing the highest QoS among the Requested Interfaces and sending it as a response to the initial request.

The global view of the re-configuration of an application can be illustrated using the BAWS example application. One possible Service Selection Graph is presented in Figure 6.5. In this case for both of the Abstract Services *Sound Sink* and *Limit Check* two candidates are available. The procedure of re-composition is illustrated in the sequence chart given in Figure 6.6. It starts with a Discovery Request message including the weighting factors (wf) sent from some unit that wants to execute the driving assistance (Assistance Requester). It is first received by Service A which is an implementation of

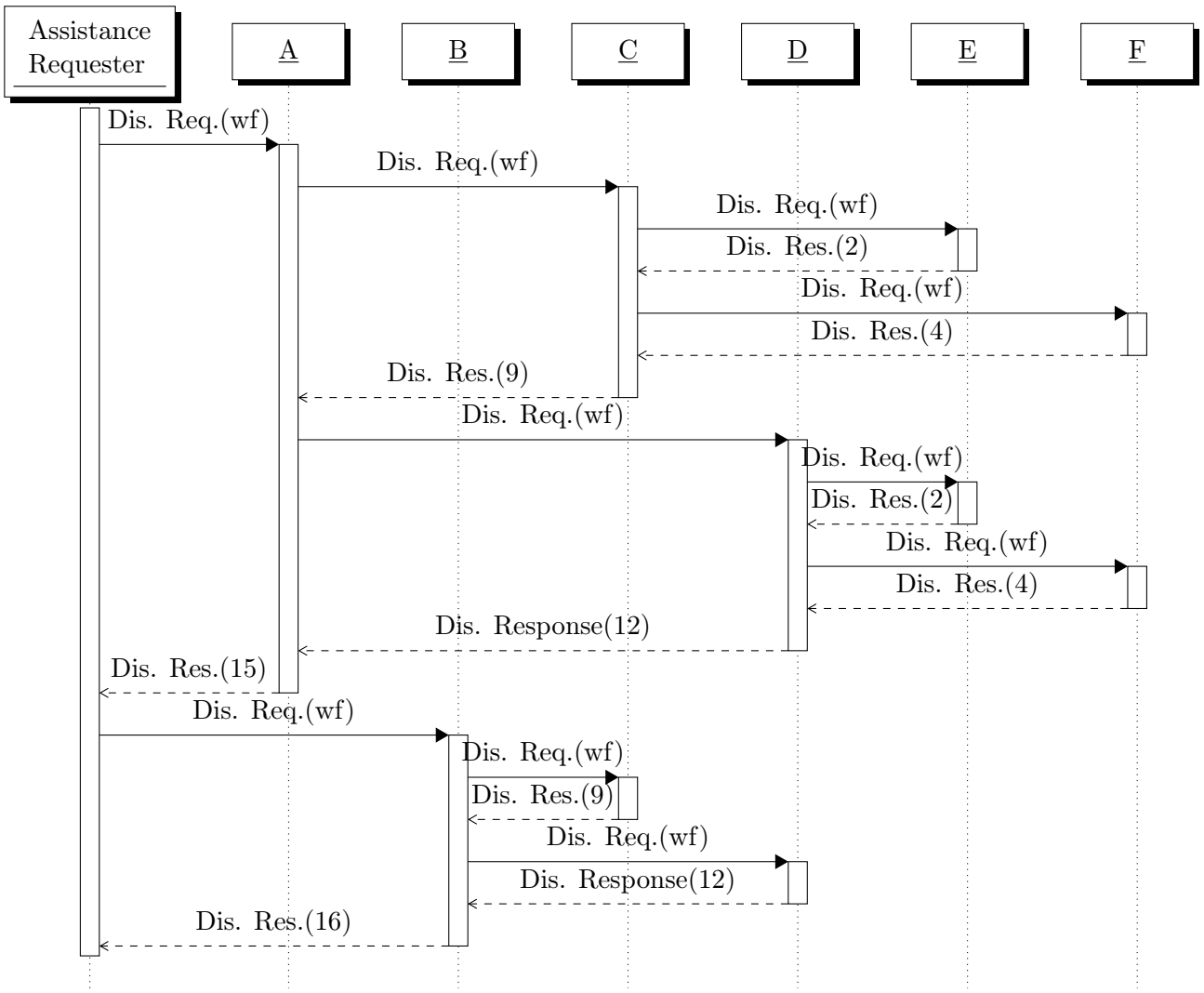


Figure 6.6: The sequence chart of the SODA compositon algorithm for the example given in Figure 6.5

Sound Sink. In order to be able to determine which one of the two candidates C and D, which both offer the *Limit Check* functionality, should be selected A first sends out a Discovery Request to instance C. This instance owns two Requested Interfaces, both matched by one instance each (E and F). To calculate the solution of its subproblem a Discovery Request is sent out by instance C to both Service Instances. Since E and F are implementations of the Abstract Services *Bending Angle 1* and *Bending Angle 2* respectively both do not own any Requested Interfaces. As described in Algorithm 1 both of them do only use the weighting factors received to calculate their own QoS value and directly respond afterwards with the values 2 and 4 respectively. These responses brought together with the quality measures of the connections and its own QoS by instance C. This procedure solves the subproblem assigned to C with the result of 9 which is responded as subsolution to the requesting instance A. Since instance A identified a second candidate D it also requests D's subsolution. Just as instance C did a moment ago, instance D calls E and F to get their QoS. The results of this calls are used to calculate the solution of the subproblem of D which equals to 12 in this example. Since instance A has now all values available it is able to calculate and return its own subsolution. Taking into account A's own QoS, the received subsolutions and the connection qualities the result of this computation is 15. In a next step, the second candidate for the *Sound Sink*, instance B receives the Discovery Request. As it has discovered two candidates, C and D to its Requested Interface it requests both to send

their subsolutions. Both instances make now use of the re-use principle of dynamic programming. As they already solved their subproblems they directly respond to the request using these values. As a result instance B computes an overall solution of 16 and sends that to the initial requester. The Assistance Requester would now choose to use the application offered by Service Instance A with an overall QoS value of 15 over the one offered by B providing a quality measure of 16.

In order to rank the complexity of the re-configuration the overall number of Discovery Requests and thereby of calls of the Service Assignment algorithm is used. The reasons therefore are that it is easy to measure and it is a good indicator for the processor load caused by the adaptation process. Furthermore it is independent from the actual implementation and hardware used and thereby suitable to compare the complexity of the algorithm on an abstract level.

Analyzing the example given in Figure 6.5 the SODA algorithm sends out 10 Discovery Requests. Compared to an exhaustive depth-first-search which is also capable of finding the optimal composition (14 calls) this equals to a cutback of about 30%. Other examples confirm this benefit. Figure 6.7 (a) illustrates another possible Service Selection Graph of the BAWS example. This time the Abstract Services *Sound Sink* and *Limit Check* are matched by three Service Instances each. *Bending Angle 1* and *Bending Angle 2* are represented by two candidates in each case. While the exhausting depth-first-search algorithm triggers 48 executions of the *Service Assignment* method, the one based on dynamic programming cuts them in half by executing the procedure only 24 times. The graph given in Figure 6.7 (b) represents a potential Service Selection Graph of a visual backing up assistance for a truck and a one axle trailer. In the given scenario some of the Abstract Services are matched by only one, some of them by two Service Instances. Composing the application with the exhausting algorithm would trigger 25 executions compared to only 16 with the SODA approach. The last example is presented in Figure 6.7 (c) and represents a potential Service Selection Graph of a visual backing up assistance for a combination of a vehicle and a two axle trailer. The given graph contains twelve Abstract Services and a total number of 15 Service Instances. The reduction of Discovery Requests in this case is about 35% (46 with the exhaustive, 30 with the dynamic programming algorithm).

6.6 Summary

This chapter introduced a novel approach to react to runtime changes in driver assistance systems. In SODA-based systems these changes are addressed by adapting the application through re-composition of the Service Instances.

In order to determine the demands and circumstances of system changes in DDAS six different events have been identified and analyzed.

Using the SoaML models created by the SODAdev process model two possibilities have been identified. The Architecture-driven approach is based on the information included in the Service Architecture. It orchestrates the system using a central unit that overlooks the whole system. Although this is an interesting method, it implies some disadvantages like creating a single point of failure or leading to the need for a high amount of memory and computing power within a single unit. Besides, it is not compatible with the situation in the truck and trailer domain which lacks of a central system integrator. This fact would cause additional overhead through negotiations between multiple re-orchestration instances. In order to overcome these issues an

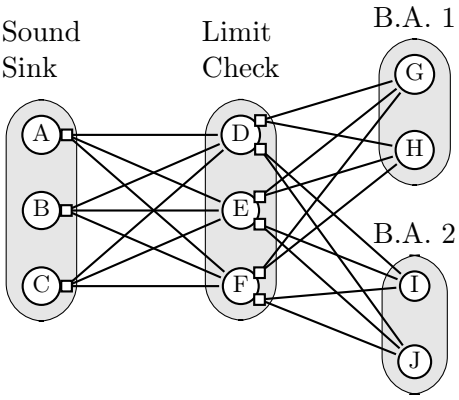
Interface-driven approach has been designed based on choreography.

The overall re-configuration procedure is organized in four consecutive phases. The first one carries out a combined Discovery and Selection algorithm. The following two steps determine the assistance applications currently available and selects one of these. Finally the last phase executes the Service Instances chosen to deliver the assistance to the driver.

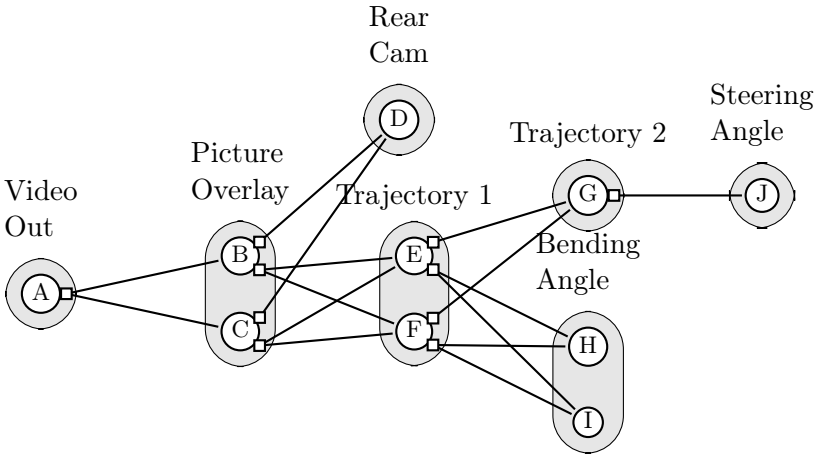
The critical part of this re-configuration procedure is the Discovery and Selection phase. For SODA, a novel algorithm has been designed in order to match the demands set up within this application domain. Therefore, 22 different approaches belonging to eight groups have been analyzed. As most of them are targeting at Web Services, the major application domain of Service-oriented systems, these 22 proposes have been examined using domain-specific requirements. Although none of these proposes completely fulfilled these requirements, the technique of dynamic programming turned out to be quite promising since it splits up the problem into small subproblems that are easily and independently solvable. Furthermore the re-use of already calculated subsolutions saves resources while still being able to generate an optimal overall solution at runtime.

The unique selection algorithm that has been developed within this work uses predefined cutting lines to divide the overall problem into smaller ones. Hereby no overhead is created by deciding on the subproblem size at design-time. The results are very small and easy to solve subproblems that can be calculated locally. Since based on solutions of minor subproblems the overall optimal solution is generated without any additional computations. Analyzing example Service Selection Graphs it has been shown that the SODA algorithm offers superior complexity values compared to a depth-first-search approach also calculating optimal configurations in a distributed manner.

(a) Another example of a potential Service Selection Graph for BAWS



(b) An example for a Service Selection Graph for DAS for one axle trailers



(c) An example for a Service Selection Graph for DAS for two axle trailers

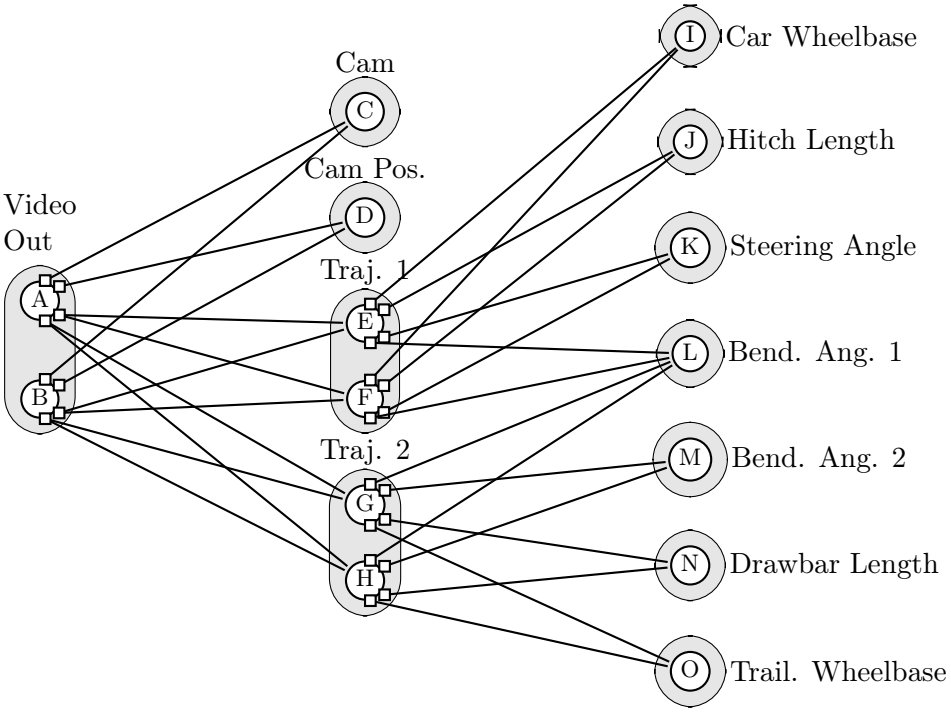


Figure 6.7: Some examples for potential Service Selection Graphs

7 A Communication Model for SOA in the automotive domain[1]

'The single biggest problem in communication is the illusion that it has taken place.'

George Bernard Shaw[2]

7.1 Introduction

In recent years the usage of middleware technologies like for example SOA within the domain of Distributed Embedded Systems (DES) has risen. Middlewares have been identified to be of great use when these DES get very complex or heterogeneous, the time-to-market has to be reduced or the systems have to be runtime adaptive. The SOA paradigm has for example been used in the eSOA project [121] or in the SIRENA project [25] among others. One issue when taking these two approaches into account is the fact that they make use of IP-based protocols. This makes sense since most SOA frameworks are also relying on these technologies. However, in the automotive domain Ethernet and IP-based communication is still in a very early stage. None of the few approaches currently available like for example SOME/IP are run in a production car. Instead, specialized network systems such as the Controller Area Network (CAN) or the Local Interconnect Network (LIN) are used. These systems are not capable of running Service communication directly. This is the case since they have been developed to be used in static scenarios where changes of the system are not taken into account. Furthermore they are used to directly hand over raw data instead of being a base for high-level protocols which is another difference from, for instance, Ethernet. Another important issue is that their addressing schemes are message-oriented and make use of broadcast transmissions rather than using node-based addressing and peer-to-peer communication.

As introduced earlier in this work the SODA framework consists of several layers in order to implement the separation of concerns principle. Figure 7.1 shows the Architecture which is used on the SODA Services within the system. It is divided into four different layers:

1. **Application** Holds the implementation of the actual Service Logic.
2. **SOA Middleware** Implements the SOA specific paradigms and functionalities.
3. **Communication Model** Contains software components to adjust the network used to the SOA Middleware.
4. **Hardware Abstraction Layer** Implements the low level network drivers.

This chapter focuses on the Communication Model layer as well as on a development process which guides the software engineer through the development of this component. This procedure is called SOAcom. The Communication Model has to fulfill the following requirements:

[1] This chapter is based on my publications [13] and [140]. Parts of it are extracted from these sources.
[2] Leadership Skills for Managers, see [32]

1. **Runtime Adaptation** The Communication Model layer must allow connecting or disconnecting Services at runtime and has to handle the integration of new Services into the existing network.
2. **Ensuring advantageous attributes** All of the different automotive network systems are tailored for a special purpose. This means that they are designed for a special class of applications and therefore have unique attributes to ensure the efficiency, safety or timeliness demanded in these applications. These advantageous attributes need to be ensured and preserved wherever possible when adding Service communication mechanisms.
3. **Interoperability** The Communication Model should allow to run mixed networks. Therefore it must be able to connect Services using a network which is used by other components, running traditional communication at the same time.
4. **Stability after initial configuration** In the event of adding a new Service to the communication channel, existing nodes should not have to re-configure their communication stack. Instead, only the newly added entity is to be adapted.

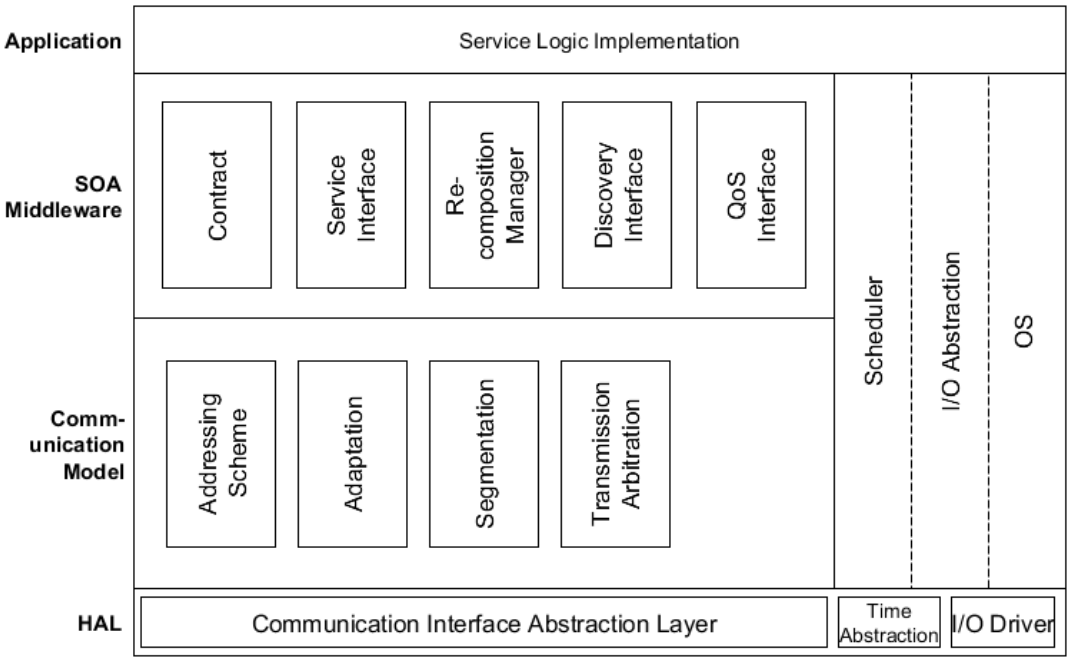


Figure 7.1: Overview of the SODA architecture.

This chapter is organized as follows. Section 7.2 summarizes the related work in this domain to illustrate the state-of-the-art as well as the shortcomings of approaches previously published. Section 7.3 gives an overview of the software components building the SODA Communication Model. Section 7.4 illustrates the associated SOAcom Development Process in detail. This Development Process is used to construct the implementation of the Communication Model of an example application for CAN in section 7.5. Finally, section 7.6 summarizes the chapter.

7.2 Related Work

As stated before, middleware approaches in the domain of Distributed Embedded Systems have become quite popular in recent years. In this section, some of these approaches are evaluated regarding their communication mechanisms. The eSOA project for example used the SOA paradigm to build systems that control smart buildings

Publication	Exchange Format	LAddressing Scheme	Base technology
Scholz et al. [121]	xml	node-based	-
Bohn et al. [25]	xml	node-based	DPWS
Lopez et al. [86]	raw	node-based	-
DySCAS [73] & [52]	raw	node-based	LINX

Table 7.1: Comparison of the some middleware approaches targeting on embedded systems

([121]). However, the approach makes use of several traditional technologies of the Web Services domain such as the Web Service Description Language (WSDL). This fact, for example, requires to send XML files through the network to exchange information between Services. This is a big drawback when a low bandwidth network system like, for example, CAN or LIN is used. Furthermore, it makes use of ZigBee wireless communication technology that works with node-based addressing. Another approach called the SIRENA project, which is explained in [25], has the same drawbacks of not being usable in low-performance networks. This is because it uses the Device Profile for Web Services (DPWS) standard, that makes use of IP-based and hereby node-based communication. Lopez et al. present a Middleware concept in [86] which is targeting on the avionics domain. Instead of exchanging xml files between the Services which causes a significant overhead, the authors developed their own communication protocol directly on the TCP layer. However, since it also uses IP-based communication, it is not directly usable in today's cars. Two other approaches are directly aiming on automotive networks. Jahnich et al. present an approach in [73] which uses a middleware to carry out load balancing in the events of ECU failures or overloads. Their work is part of the DySCAS project which targets on introducing self-configurable systems in the context of embedded vehicle electronic systems. Being designed for automotive infotainment systems low-performance ECUs are not taken into account. However the specific use case within this work, namely the connection of devices distributed over a car and trailer combination, is explicitly excluded from the approach [52]. This is because of communication issues between car and trailer. This issue is fixed by the SODA framework. The four approaches described are summarized in Table 7.1. As shown here, none of them makes use of message-based communication which is the standard in today's automotive network systems.

Another field that is interesting within this context is the use of high-level protocols for automotive networks. Much of the work done in this area has eventually became an industrial standard. Most of these approaches have been targeting on the CAN network. For example, the ISO TP standard defines a transport layer that allows to send frames of a maximum of 4095 bytes via CAN ([71]). The two competing standards CANopen ([35]) and DeviceNet ([36]) also aim on adding higher layers to CAN to offer some extended features. Unfortunately, all of these standards have been developed for being used in only one of the several different automotive networks. In our approach of implementing Service-based technologies, we want to use the whole spectrum of automotive network systems. Other high-level protocols, like for example the XCP protocol ([80]) that enables engineers to calibrate ECUs within a car, offer interoperability throughout a higher number of network systems. Unfortunately they are very restricted in their purpose. Summing up one can say that none of the Middleware approaches or high-level

protocols developed recently has all the features needed to build the Communication Model of the SODA framework.

7.3 Overview of the SODA Communication Model for SOA-based DDAS

As stated before, the Communication Model builds the bridge between the high-level communication located in the SOA Middleware and the low-level automotive network drivers. However, this layer exceeds the functionality of a simple wrapper. This is due to the fact that it has to compensate the shortcomings of the automotive network systems related to the application requirements. Thereby, it maps the particular functionalities of the SOA middlware like for example discovery calls to find another Service in the network to actual messages and hands them over to the Hardware Abstraction Layer. Another very important duty of this layer is to organize the integration of the Service into the underlying network and to offer transport protocol functionality like the segmentation of large Service calls. As shown in figure 7.1 the Communication Model contains four major components:

1. **Addressing Scheme**
   This component maps the Service calls onto message addresses and vice versa. Therefore, it contains all relevant information for this task. One difficulty within this task is given by the fact that in many automotive network systems addresses do not only identify the content of a message but also its priority. A SODA Addressing Scheme component must be aware of this fact. Although the Addressing Scheme is adapted according to the requirements of the application and the network there are some characteristics that are identical within all SODA systems. The first characteristic is, that message-based addressing is used. By using this principle, the address describes the content of the message rather than the sender and/or receiver. A second common technique is the use of a describing number that identifies the functionality of the Service. In the SODA framework this number is called Service Class Address (SCA). The basic idea is to assign a SCA to an Abstract Service and to register it in a global document. By doing so, it is assured that a functionality can be identified within a system using its SCA. Thereby, this functionality can be reused by other applications.

2. **Adaptation**
   The adaptation component carries out the integration of the Service within the network. At the very moment when the Service is added to a network it needs to announce the existence of its functionality and claim addresses or time slots to be able to take part in the communication.

3. **Segmentation**
   As some Service calls or data exchanges might be larger in size as the maximum load of a single message in the underlying network, a segmentation process must be available. This component divides the abstract SOA calls into message sized junks in order to send them over the network. In a second step it takes care of the transmission sequence. Moreover, it assembles the incoming messages when receiving large size SOA calls in order to reconstruct them.

4. **Transmission Arbitration**
   Since all automotive network systems use shared segments, in order to send messages, a channel access control mechanism has to be established. This mechanism can be controlled by a central device, like in some time division approaches or for example the LIN network. It also can be distributed over the nodes in the network. An example for the latter mechanism would be CAN which uses a Carrier

Sense Multiple Access / Collision Resolution (CSMA/CR) technique and the multi-master principle. The Transmission Arbitration component has to be adapted to the underlying network in order to make sure that the network's channel access method keeps the priorities and scheduling defined in the application.

Those components listed above are generic containers that have to deliver a specific functionality. However, they have to be specifically designed and implemented for every assistance project depending on the requirements set up by the application and the characteristics of the actual network used. This approach allows to create a well tailored middleware that fulfills all demands set up by the application in an efficient way. Due to this and the fact that any particular implementation of the Communication Model causes a significant amount of effort, a well structured development process that guides the software engineer through several steps has been defined. This process model is named SOAcom. In the following section the development process is described. Its steps will eventually result in an executable implementation of the SODA Communication Model.

7.4 SOAcom: A development process for SODA Communication Models in automotive SOA-based systems

This section will describe the development process SOAcom. SOAcom is a tailored and well structured approach to analyze the DDAS application carried out by the system as well as the characteristics of the network used. Using the results of both analyses the steps of the development cycle are created. Carrying out these development steps a team of programmers is able to implement a SODA Communication Model tailored to the specific needs of the overall system and thereby efficient in terms of memory and processor usage. In the following subsections an overview of the approach as well as a detailed illustration of the several steps to be carried out are given.

7.4.1 Overview of the SOAcom process model

The SOAcom process model consists of four major steps. These four steps guide the developer in creating a SODA Communication Model tailored to a specific application using a specific automotive network. Figure 7.2 gives an overview over the whole process.

The process starts with phase 1 on the top left corner of Figure 7.2. This step carries out SODAdev which has been described in chapter 5. As described there, SODAdev allows to transform ideas for Distributed Driver Assistance Systems into a full system specification of the functional characteristics in a model-based procedure. As a result, a system description in SoaML is created. This model is used to extract the requirements of the application which influences the communication.

In phase 2, the network protocol used is analyzed. This targets on deriving a summary of relevant characteristics. Phase 2 can be carried out completely independent from the actual application as its only concern is the network itself. The analysis is done on the base of a questionnaire that guides the developer through this phase. As a result of this step, those characteristics of the network which are important for designing a Communication Model are summarized.

Phase 3 merges the results of the phases 1 and 2. The goal of this phase is to identify which steps have to be taken to allow the usage of the desired network protocol within the specific application.

Finally, phase 4 consists of the execution of the tasks identified in phase 3. By carrying out those tasks the Communication Model is implemented step by step until it

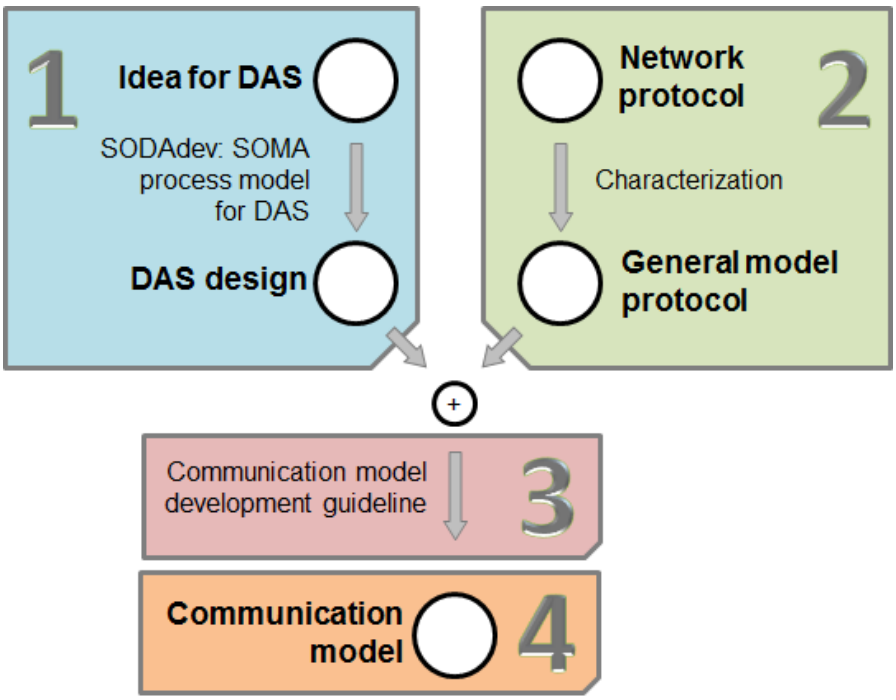


Figure 7.2: Overview of the SOAcom development process.

fulfills all requirements set up by the application and complements the mechanisms of the network system used as necessary.

The remainder of this section will describe the four phases of SOAcom in detail.

7.4.2 Phase 1: Determining the requirements of the Communication Model set up by the application.

In chapter 5 a model-driven development process for SOA-based driver assistance systems has been described. This process allows to generate a detailed description of such a system in SoaML starting from the plain idea of a DAS or a non-SOA legacy system. By working through the phases of the development process described there, the used Services are identified and derived from the functional requirements. These Services are enriched with descriptions about the provided and requested functionality, as well as a contract. As described in chapter 5, a contract in a SoaML model is an artifact that describes how providers and consumers exchange data with one another. Therefore the UML metaclass collaboration is extended to define the roles of the interacting partners as well as their behaviour ([101]).

The resulting SoaML model offers a detailed description of the communication exchanged between the Service and its requester. This description can be used to derive the requirements set up by the application that have to be fulfilled by the Communication Model. Inside the description of the communication in each contract, which is modeled using UML Sequence Charts, messages sent by both partners are attached by UML Signals. A UML Signal is a standard modeling element of the Unified Modeling Language. It is used to describe the data packet exchanged by two entities. This is done by specifying the content of the Signal in detail using attributes. These attributes on the other hand are of a specific data type. By specifying both, the attributes and their data types a detailed description of each message exchanged within the system is given and can be used to derive the requirements regarding the Communication Model.

Figure 7.3a presents an example of such a Signal. In this case the name of the Signal is "Sig_CameraPositionResponse". It owns three attributes namely Height, Distance and Angle. Each of them is of a specific data type. For example, the attribute Height is an unsigned integer with 16 bits of length (abbreviated uint16).

The data types used for the attributes of the Signals have to be defined as well. In SODAdev twelve different types which include, for example, the standard data types of the common programming languages like integer number with 8 bits of length (int8), floating point number with 32 bits of length (float32) or ascii characters have been declared. Figure 7.3b presents the data type uint16 that has been used in the example to specify the size and coding of the Height attribute.

(a) Signal

(b) Data type

Figure 7.3: Example elements

All Signals and data types are defined in a separate package within the SoaML model in order to keep the model structured. The Signals are used in the contract definition within Asynchronous Signal Messages. In contrast to UML Synchronous Messages this type allows to directly add Signals to the message exchange artifact. This allows to create an integrated and detailed description of the communication process at the same time. Figure 7.4 presents an example of such a sequence chart within a contract which uses this kind of messaging. In this example, which is an extract of a SoaML contract created using SODAdev, two entities are participating within a communication scenario. The consuming unit on the left hand side sends an Asynchronous Signal Message containing a predefined Signal to the second one, a Service that provides some information about the position of a camera. This Service answers the call by handing over the information in the format specified in the Signal description given in Figure 7.3a.

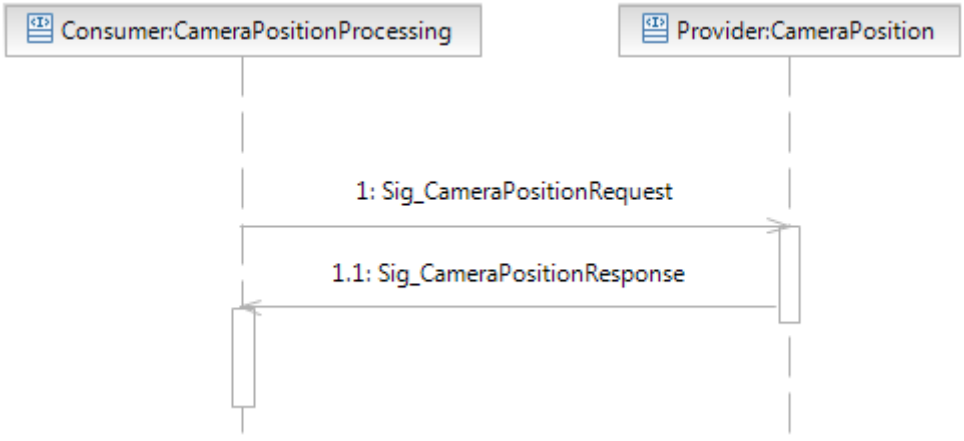


Figure 7.4: Example of a communication sequence using Asynchronous Signal Messages.

With this level of detail in the SoaML model it is possible to derive the requirements of the application regarding the communication aspects. These requirements are:

1. The number of Services in the application.
2. A list of Services alongside with their functional identifier (SCA).

3. The size in Bytes of the biggest message in the application.
4. The presence of periodic messages in the application.

The first requirement, the number of Services, can be easily discovered by counting the number of Participants in the overall Service Architecture, which is part of the SoaML model. In small size applications it can be done manually. For big size applications this job could also be done using a simple program. This is due to the fact that the SoaML model can be represented by an xml file. As such xml files are easily parsable the analysis of the number of existing Services can be determined without big effort.

In the same way the second analysis, which creates a list of Services, can be carried out. Again, the SoaML model can be either parsed manually whenever the application is small or by using an automatic parsing for larger systems. By inspecting the Interfaces package within the SoaML specification all the necessary descriptions of the used Services can be found. The corresponding SCA to describe the functionality of each on the other hand, is meant to be defined in a global Service description file to allow re-use of Services between different applications.

The most efficient way to determine the biggest message in the model is to analyze the package containing the specifications of the used Signals and data types within the SoaML model. This can again be simplified by using an automated script program which analyses the UML model in order to reduce the effort of the developer especially when the specification is large. As a result of this step the largest message as well as its size in Bytes is identified.

The last step is to identify whether the system contains any single periodic messaging scenario. This is important since some automotive network systems do support such communication by default. As there is no central package available containing an overview of all communication cycles the developer needs to sequentially scan all contracts available when determining this issue manually. As an alternative, the xml file describing the SoaML specification can be scanned for the existence of the UML element used for periodic messaging. If there is only one periodic message in any of the contracts, this has to be reported. This last analysis of the application's SoaML model completes the step of deriving the requirements set up by the application.

7.4.3 Phase 2: Characterization of the network protocol

Phase 2, which is designed to develop a high-level description of the automotive network protocol intended to be used, is completely independent from phase 1. Therefore it can be carried out either simultaneously or in sequence. The main goal of this second phase is to hide the complexity of the protocol's implementation and focus strictly on its capabilities relating to the needs of the SODA framework.

The description of network protocols is often done using formal or semi-formal description methods. Two of the most popular ones are formal languages and state diagrams. In the first case the focus normally lays on the exchanged data structures. State diagrams on the other hand try to provide a description that easily allows to understand and follow the flow of the information exchange.
Both descriptions mentioned above are mainly used to give the developer guidance when implementing a communication stack using this protocol. They do not explicitly highlight the capabilities of the protocol. However, for the development of a communication model within the SODA framework this information has to be extracted to be used

within the ongoing process. In order to characterize network protocols, a characterization method based on a questionnaire has been developed. Phase 2 is conducted by analyzing the protocol using this questionnaire.

In the following subsection a list of attributes specified to describe a network protocol is presented. This is complemented by a questionnaire that makes it possible to collect the values of these attributes.

7.4.3.1 Characteristic attributes of a network protocol

The network protocol description paradigm designed for the SODA framework summarizes some of the most important attributes of a network. It is divided in five different groups. The first group, called transmission, describes how the protocol transmits data to the channel. The second group, called physical capabilities, details the topology and physical medium. A third group, named network capabilities, gathers the characteristics concerning the whole network. The fourth group, called dynamic capabilities, manifests how the protocol adapts to environment changes. Finally, the fifth group, called dependability, describes the safety and security mechanisms provided. Figure 7.5 gives an overview over the characteristics used in the protocol description scheme.

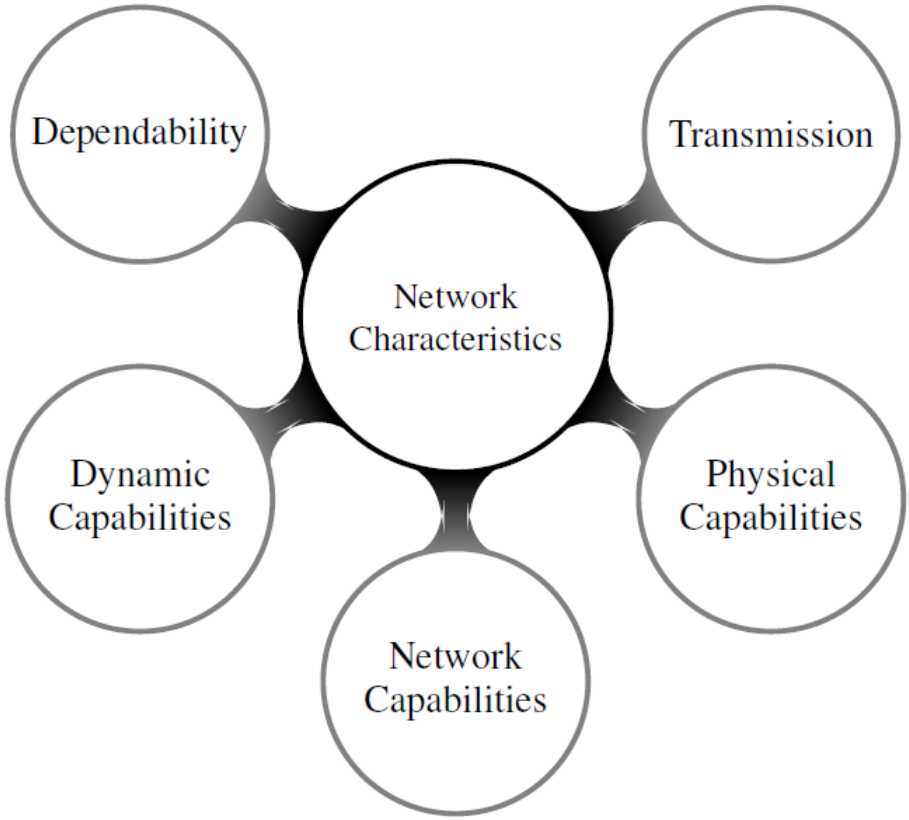


Figure 7.5: Groups of characteristics of a network system.

7.4.3.2 A questionnaire to characterize a network protocol

The discovery of characteristics of the network protocol to be analyzed is done by a questionnaire. Within this questionnaire all relevant values are analyzed and extracted. The questions are dedicated to the groups introduced in section 7.4.3.1. They are build in a manner that the answers to them are one of the following three types:

- **Checkboxes** to allow multiple answers to a question.
- **Radio Buttons** to restrict the possible number of answers to one.

- **Free text answers** to provide the chance to for example input numeric values.

Table 7.2 shows an example of one of these questions. The presented case is the test for the prioritization mechanism of messages within a network. As visualized in the table the answer type of this question is of the Radio Buttons type. This means that only one of the offered options "Node-based", "Message-based", "Scheduler-based" or "No Prioritization" can be selected.

Frame prioritization
o Node-based
o Message-based
o Scheduler-based
o No prioritization

Table 7.2: Extract of the questionnaire to characterize a network: Frame prioritization.

A second example is given in Table 7.3. Here, the possible modes of addressing are determined. In the scope of SODA it is especially interesting whether the network system is able to address nodes or logical components in order to make design decisions for the communication middleware. In this example it is also possible that a network supports both possibilities. Therefore the Checkbox style answer is used.

Possible entities to be addressed
❑ Node
❑ Logical component

Table 7.3: Extract of the questionnaire to characterize a network: Addressing.

The outcome of this second phase is a general model of the protocol that contains all relevant information of the network systems in the scope of SODA. The information gathered here is used together with the requirements of the application which have been determined in phase 1 within the next phase of SOAcom. With the completion of this second phase the analysis part of the process model is done and the focus swaps on creating a well-tailored communication model.

7.4.4 Phase 3: Mapping requirements to attributes

In the third phase the shortcomings of the deployed network protocol compared to the requirements of the application are identified. Furthermore a list of tasks which must be carried out to develop a tailored Communication Model is created. In order to do so, the information gathered in the two previous phases of SOAcom is linked to each other. On the one hand, phase 1 provides the list with the communication requirements of the application which, as explained, is composed of four different values. On the other hand, phase 2 supplies the list containing the values of the network protocol attributes, which describes the capabilities of the automotive network protocol.

These two information streams are processed using a set of flowchart diagrams. This kind of diagrams are normally used by software designers to represent algorithms or processes. In the SOAcom development procedure they are used for identifying the

network protocol shortcomings and in a second step to determine the tasks to be carried out to overcome them.

The following subsections describe these diagrams and how they are used within SOAcom. Furthermore, an example of such a flowchart diagram as well as an extended description of its contained tasks is provided.

7.4.4.1 Flowchart diagrams to guide the developer through the process

Each of the flowchart diagrams represents a structured set of questions and is linked to a list of possible tasks. The questions given match the application's requirements determined in the first phase with the networks' attributes from phase 2. By going through these flowcharts the comparison is carried out in a well structured and comprehensive manner. In the event that one of these flowchart diagrams detects a shortcoming, the selfsame circumstance highlights a pre-defined task to overcome its presence. Therefore, each shortcoming alongside with the task to fix it is recorded. The outcome of this phase is a set of development tasks to be executed by the development team.

The flowchart diagrams of this SOAcom phase are complemented by a text document. This document contains both the explanation of each branch element in the flowcharts and a detailed description of each development task to be carried out.

This third phase of SOAcom contains six different flowchart diagrams, each of which is related to at least one attribute of the network protocol. Besides, the structure of the diagrams reflects directly the configuration of the Communication Model components as presented in Figure 7.1.

The first one of these diagrams corresponds to the Addressing scheme. It is dedicated to analyze all properties and characteristics needed to create a mechanism capable of guaranteeing the delivery of messages to the receivers. In order to fulfill this task it is necessary to take into account the characteristics of the network protocol addressing scheme, the network protocol transmission method and the set of Services defined in the SoaML model describing the application under development.

The second diagram checks the ability of the used network to assign addresses dynamically. This is important as it helps to extend the dynamic capabilities of the network protocol if needed to allow runtime assignment of addresses to the Services and, in some cases, to the nodes. For this, it is necessary to distinguish between node-oriented and message-oriented network addressing modes.

Flowchart number three, which is called mode of communication, assists the developer in adapting the channel arbitration mechanisms of the network in order to allow Service communication using the SODA framework. Therefore it is necessary to check whether the network protocol makes use of a master/slave or a multi-master communication principle.

Similarly to the mode of communication diagram, the fourth one named mode of prioritization assists the developer in checking for an important property when designing runtime adaptive networks. This flowchart detects the network systems ability to manage the prioritization of new Services dynamically.

The Fragmentation diagram which is flowchart number five, helps to ensure that the network protocol is able to transmit messages when their content overcomes the

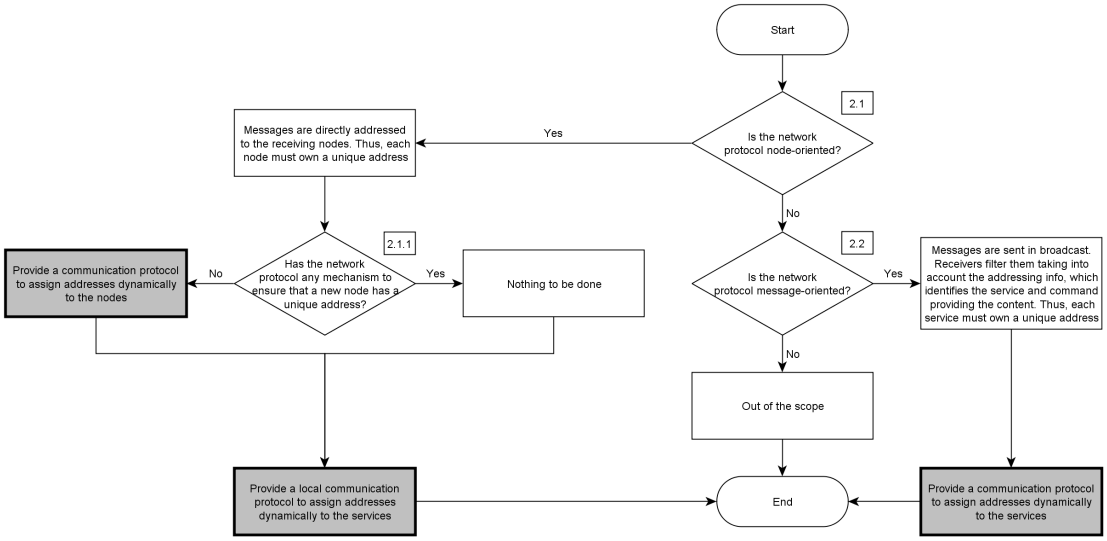


Figure 7.6: An example flowchart diagram.

maximum payload. For this, it is necessary to compare the size of the messages within the application, as well as the maximum message payload of the network protocol. In the case the former value exceeds the latter one the flowchart also determines whether a fragmentation protocol is already provided by the network system.

Finally, the sixth flowchart diagram called trigger condition explores the message scheduling principles to be used. Therefore, the existence of periodic messages in the application on the one hand and the types of scheduling schemes supported by the network protocol on the other hand are examined.

7.4.4.2 Flowchart diagram example

This subsection presents one example of SOAcom's flowchart diagrams. It is called "ability to assign addresses dynamically" and is shown in Figure 7.6.

The semantics of the used flowchart elements are as following: the diamonds used represent conditional statements. In the SOAcom approach they contain the questions to be asked in order to identify the list of tasks to be executed. Within the flowcharts itself these elements change the execution flow.
The rectangles used need to be divided based on the color. The first class of rectangles is white-filled. These elements present important information regarding the flow of the analysis. On the other hand, the gray-filled ones provide a short description of the task to be carried out when passing that element while working through the flowchart.
Finally, the diamonds are equipped with labels. These labels refer to a specific section in the text document which corresponds to them and describes the workflow in more detail.

By identifying shortcomings of the network protocol using the flowchart diagrams the set of tasks to be carried out is determined. As illustrated in Figure 7.6 each of these shortcomings is equipped with a short task description. However, adding the complete definition of the task as well as the possible options to be taken would increase the size of the flowchart enormously. In order to obtain lucidity this additional information is moved to an external document. In Figure 7.7, a summarized version of the document's extract corresponding to this diagram is shown and explained.

The given example helps the developer to achieve an adaptive architecture. Therefore,

2. Ability to assign addresses dynamically

Since we are constructing an adaptive architecture, the LLCM must allow connecting new nodes at runtime and participate in the communication. Moreover, It must also allow to start new services at runtime when connecting a new node, or even later. In order to be able to deliver messages to these new services it is necessary to ensure that they can acquire a unique address dynamically. Additionally, in some cases, it could be also necessary to assign addresses dynamically to the nodes.

2.1. When the communication protocol is node-oriented

Messages are directly addressed to the receiving nodes and, thus, each node must own a unique address.

2.1.1. The LLCM must ensure that new nodes and/or services installed at runtime have unique addresses

In order to allow the communication among services in a node-oriented CP, the network must be able to provide mechanisms to ensure that new nodes have already a unique address, or that they can acquire it dynamically.

In case the CP does not provide any mechanism to do so, the designer of the LLCM must provide a protocol to assign addresses dynamically to the nodes. This protocol should preserve the mode of communication of the CP, that is, the protocol should not force to insert a central node to carry out this task in multi-master networks. One possible option is to look into the existing CP high-level protocols. In case any of them implement a suitable algorithm to assign addresses at runtime, the designer of the LLCM can extract it and then added it into the CP.

Once each physical component of the network is able to own a unique address, services must be provided with unique addresses, so messages can be delivered not only to the nodes but to the services. In this sense, the designer of the LLCM must provide an algorithm to assign unique addresses to the services dynamically. Note that, since part of the node-oriented addressing scheme of the services already specifies the address of the node (which is unique) this algorithm can be executed locally in the physical device.

2.2. When the communication protocol is message-oriented the LLCM must ensure that new services installed at runtime can acquire addresses dynamically

In a message-based CP messages are sent in broadcast and the receivers are the ones responsible for filtering them, so that only the interesting ones are accepted. For this, receivers check the value of the addressing information, which identifies the service and command providing the content of the message.

The designer of the LLCM must provide an algorithm to assign unique addresses to the services dynamically. Note that, contrary to the algorithm used in node-based protocols, this algorithm must communicate with all the services of the network.

One possible option is to use the automatic identifier assignment algorithm presented in "A CAN-based Communication Model for Service-Oriented Driver Assistance Systems". This algorithm is based on the Bully Algorithm used for electing a leader in a distributed system presented in "Elections in a Distributed Computing System". The main advantage of using this algorithm is that all the address assignment process can be carried out without a central logical component.

Figure 7.7: Extract of the document describing the tasks.

the Communication Model must allow connecting new Services at runtime. Moreover, in order to be able to deliver messages to these new Services, it is necessary to ensure that they can acquire a unique address dynamically.

Figure 7.7 shows an extract of the document describing the tasks. In this part of the document the tasks that are executed to allow dynamic addressing are described. The first conditional statement in Figure 7.6 which is labeled 2.1 checks whether the network protocol is node-oriented. In this case, messages are directly addressed to the receiving nodes. If so, each node must own a unique address. In this sense, the Communication Model must ensure that new nodes and/or Services installed at runtime have unique addresses in order to be reached separately.

The requirement of providing unique addressing is expressed in the second conditional statement, labeled 2.1.1. In case the network protocol does not provide such a mechanism, the designer must add a protocol which offers this feature. However, this protocol should preserve the mode of the network. More specifically, the protocol should not force to add a single master node when dealing with a multi-master scheme.

There are different approaches on how to solve such an issue. One of them is to check whether there are extra protocol layers. One example for such a circumstance is the ISO 15765-2. This standard specification is also known as ISO-TP. Originally the CAN specification only defines the layers one and two of the Open Systems Interconnection model (OSI). ISO-TP adds the layers three and four and thereby takes care of for example segmentation. Similarly to this widely used CAN extension other protocols used in the automotive domain also feature such enhancements. Such add-ons are a good starting point when having to add mechanisms that are not part of the original network specification as these gaps might have been problematic in other scenarios of usage as well. In case that any of them already implements a suitable algorithm to assign addresses at runtime, the designer can extract it and then add it into the Communication Model.

Coming back to the initial example, the communication model must ensure that not only new physical components of the network are able to retrieve a unique address. As a matter of fact, new Services, which can be seen as logical components, must be assigned with unique addresses, too. In this sense, the designer of the Communication Model must provide an algorithm to assign unique addresses to the Services dynamically. Note that, if a node-oriented addressing scheme is already executed by the network system, the addressing of the Services can be managed locally inside the physical device.

A completely different path of the flowchart is chosen when the network protocol is message-oriented. This case is visualized at label 2.2 in figure 7.6. Messages in such systems are sent in broadcast which means that they have no explicit receiver. Instead, the addresses of the packets are describing their content. In such networks, it is the receivers that are the ones responsible for filtering them. In the Service communication scheme as it is used within SODA, the address of a message identifies a Service call. Looking at such a Service call in detail, it contains an identification of the functionality and the Service instance. As the addressing is not node-based, the designer does not have to assign addresses to the nodes. On the other hand, an algorithm to assign unique addresses to the Services dynamically must be provided. This means, that each Service must be identifiable by a unique combination of the functional and the instance identifier. One possible option is presented in the example given in section 7.5.3.2.

7.4.5 Phase 4: Implementing the components of the Communication Model

In the last phase of the SOAcom procedure, the Communication Model is constructed by defining the content of each of its components. This is done by executing the tasks collected in the previous phase and hereby extending the capabilities of the network.

One important characteristic of SOAcom is traceability. Trailing a specific piece of code within a Communication Model implementation to the discrepancy between application requirements and network characteristics can be done in two steps. First, using the tasks documentation one can easily backtrack each implemented component of the Communication Model to a task. Second, the flowcharts used allow to trace this task all the way back to the specific characteristics of the network and the application which set up the need of this implementation. Table 7.4 gives an overview of the relationship between the components of the SODA Communication Model and the SOAcom flowchart diagrams. This high level of traceability simplifies changes within an already implemented stack and allows conclusions on the amount of code needed to fix each discrepancy between application needs and network features.

Component	Flowchart diagram
Addressing scheme	Addressing scheme
Adaptation	Ability to assign addresses dynamically Mode of communication Mode of prioritization
Fragmentation	Fragmentation
Transmission arbitration	Trigger condition

Table 7.4: Relation between Communication Model components and flowchart diagrams.

7.5 The usage of SOAcom in a real world example

In this section the SOAcom approach is evaluated using an example development. The running example is a visual assistance application for a car and one-axle trailer combination. The used network is the Controller Area Network which is still the most popular one in today's cars. Within this exemplary development cycle for a specific and well tailored Communication Model for this application-network combination the SOAcom process model is used and thereby evaluated. More specifically the following sections describe how the SOAcom process model is used to analyse the application and the network, to determine its shortcomings and specify the components of a suitable Communication Model.

7.5.1 Example application

As already mentioned, a real world example is used for this evaluation. The application chosen is a visual assistance for a car and trailer combination as shown in figure 7.8. This system helps the driver in the process of backing up a vehicle with a one-axle trailer connected. The main idea is to calculate and visualize the future trajectories of the trailer and the overall vehicle. These trajectories are based on the dimensions of different parts of the combination as well as on the steering angle and on the bending angle between car and trailer. The calculated trajectories are transformed and overlaid onto the picture of a rear view camera mounted on the trailer. Figure 7.9 shows the interface of the system to the driver. For more details on the application or on the semantics of the assistance, please refer to [21].

Figure 7.8: Overview of the prototype.[19]

By using the SODAdev development cycle as described in chapter 5 the application

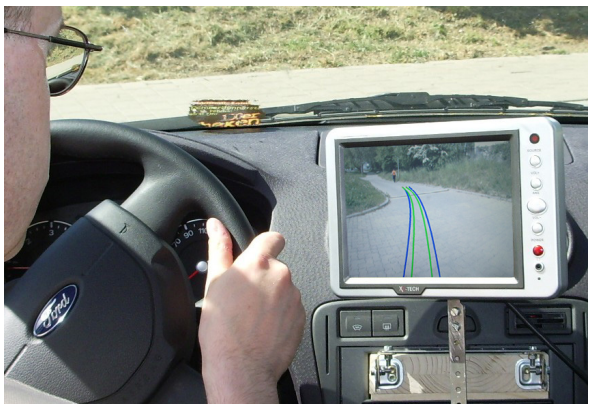


Figure 7.9: Human Machine Interface of the running example.[19]

has been transformed into a SoaML model. This model is used as a base for phase 1 of the SOAcom procedure. This example assistance system is built up by seven different Services. Figure 7.10 illustrates these Services. Some of them are sensing Services, like the ones used to determine the current steering angle or the bending angle. In Figure 7.10 these are shown on the very left of the picture. Some others are Services which process these sensor signals to add additional information to them. Such entities are for example the ones used to calculate the future path of the vehicle and its trailer, named "trajTractiveUnitService" and "trajTrailCombService" in the picture. Finally, the last group of Services outputs some information to the driver. In the example given, this is done by a Service which displays the picture of the rear view camera overlaid by the trajectories.

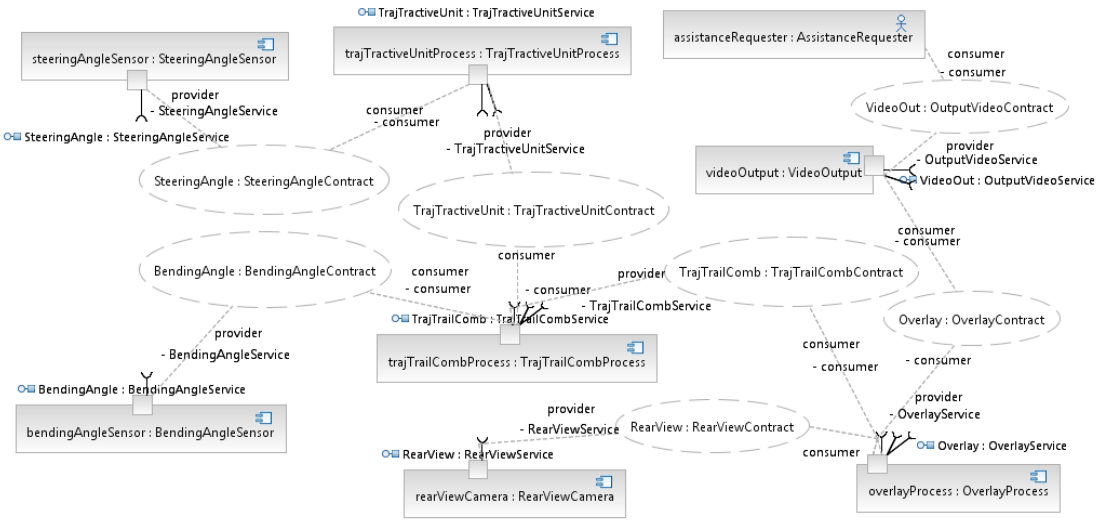


Figure 7.10: Service Architecture of the example application: a visual assistance system for cars with one-axle trailers.

7.5.2 Applying the SOAcom process

In the first phase of the SOAcom procedure, the requirements of the communication requested by the example application are determined. Therefore, its characteristics are analyzed. As stated earlier and illustrated in Figure 7.10, the example application is composed of seven different Services. These seven functional entities are extracted and listed in Table 7.5, alongside with their functional identifier, the so called Service Class

Address.

Name of Service	Service Class Address
SteeringAngleService	290
BendingAngleService	291
TrajTractiveUnitService	300
TrajTrailCombService	301
RearViewService	295
OutputVideoService	310
OverlayService	311

Table 7.5: List of Services in the example application.

In a next step, the message with the biggest size within the application is identified. Therefore, the UML signals which correspond to the CAN messages are inspected. The largest one to be found within the SoaML model of the application is 8 bytes long. It contains the information about the mounting position of the rear view camera and is needed to calculate the transformation of the trajectories into the camera picture. Figure 7.11 shows an illustration of this Signal containing three important values to indicate the position and direction of the camera. Besides that, from the sequence charts within the contracts of the seven Services, one can see that event-triggered messages as well as periodic messages are used in the communication schemes. This means, that the Communication Model to be developed must support both forms of communication.

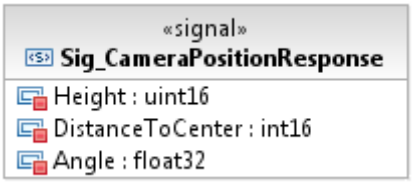


Figure 7.11: The largest UML Signals within the SoaML model of the example application.

In the second phase, the questionnaire is used to characterize the network protocol. In this case the CAN protocol is to be examined. The first value collected is from the transmission group and contains the message-oriented communication style of CAN. Moreover, it does not make use of a central master, but is carrying the bus arbitration on a multi-master base instead. An analysis of the structure of the message shows, that it has a relatively large identifier field, which offers to address more than 536 million messages. A very specific characteristic of CAN is the fact that the identifier of each message is directly defining the priority of the message and therefore is a critical point in the Communication Model.
The second group within the questionnaire is the network capabilities group. One important finding here is that the maximum payload offered by CAN is eight bytes. The third and last group within the questionnaire checks the dynamic capabilities of the network. Here, CAN lacks of mechanisms to add or remove nodes to the network at runtime. This is because the protocol does not feature any mechanisms that allow to assign addresses to newly added nodes or in this case messages.

Continuing with phase 3, the information collected through the two previous phases

is combined by using flowcharts as introduced in section 7.4.4. Hereby, the shortcomings of the CAN protocol in this specific application scenario are determined. Furthermore, the tasks to be undertaken to overcome these shortcomings are defined. The outcome of the procedure supported by the flowchart diagrams is a list of four different tasks:

1. **Set up the SOA-based addressing scheme**
   The Communication Model must allow to deliver SOA-based messages in CAN. Using the questionnaire that examines the network properties, CAN has been identified to be a message-based network protocol. Because of this, the CAN identifier scheme must be constructed in a way that allows to identify the Service call including the command executed. As the CAN addressing field has a relatively high number of addresses, the CAN identifier can be directly used for addressing the Services.
2. **Make the address scheme encoding consistent with the priority scheme used**
   In CAN, the message identifier directly defines the priority of the message. Thus, it has to be set up with care. The priority rules of CAN specify that the higher the number of the identifier, the lower the priority of the message. When setting up the addressing encoding this priority scheme, as well as the priority of the Service must be taken into account.
3. **Provide a communication protocol to assign addresses dynamically to the Services.**
   The Communication Model must allow to add or remove nodes at runtime. This includes the integration of Services and hereby the dynamic assignment of addresses to the Services. Since CAN does not provide such a mechanism it is up to the communication layers of the SODA middleware to provide one. These facts define the task of developing and implementing a dynamic address assignment module.
4. **Schedule periodic messages using event-driven scheduling**
   The Communication Model must ensure that each SOA-based message is scheduled using the most suitable scheduling policy, so time constraints are fulfilled. The CAN protocol is designed to be an event-based bus, which means that there is no in-build mechanism that allows to trigger messages on a periodic base. Unfortunately, as explained previously, the example application contains periodic messages. Because of these circumstances, it is necessary to introduce a periodic scheduling module using the event-driven policy of CAN. This module must ensure that no deadline is violated. Finally, note that is has been proven by Davis et al. in [38] that it is possible to constitute periodic messages using CAN.

In the last phase of the SOAcom process the modules of the Communication layer that have been identified in phase 3 are implemented. In the example used here, these are:

1. **Addressing Scheme** The two first tasks, ”Set up the SOA-based addressing scheme” and ”Make the addressing encoding consistent with the priority scheme used” lead to the modules which form the Addressing Scheme component. It basically manages the addressing of the Services and hereby controls the priority of each message.
2. **Ability to assign addresses dynamically** By executing the task ”Provide a communication protocol to assign addresses dynamically to the Services” the Adaptation component is created. The main goal of this component is to provide mechanisms to manage dynamically the addresses of Services when they are added or removed from the system.

3. **Trigger condition** The task "Schedule periodic messages using event-driven scheduling" leads to the Transmission Arbitration component. This component ensures that any Service is able to sent out periodic messages in an event-triggered network protocol such as CAN.

The fourth component of the Communication Model is not touched in this running example as the biggest message within the application fits into a single CAN message. The concrete design as well as some implementation details are explained in Section 7.5.3 and assessed in Section 7.5.4.

7.5.3 Design and implementation of the Communication Model

This section will explain a possible implementation of a Communication Model for the example application on the CAN network. It starts with introducing existing approaches that have tried to add high-level layers on this fieldbus. Afterwards, it introduces the design of the three modules that have been identified by phase 3 of the SOAcom process.

7.5.3.1 Related work in high-level protocols on the Controller Area Network

One quite obvious solution for such a communication layer that brings Service-orientation to the CAN bus is to introduce IP-based communication to this network. Such an approach has been realized by Reichelt et al. in [109]. The basic idea is to assign a single identifier to every node in the network. Furthermore, the addressing drops out of the identifier section of the CAN message by reserving space for the IP addresses and ports of the sender and the receiver in the data section of the frame. Since the size of an IP packet is normally much higher than the maximum frame size of CAN, the IP packets have to be split up and sent through several CAN messages. Although this approach is quite interesting it has some drawbacks, too. First of all, the static assignment of identifiers to nodes does not allow changes of the CAN network at runtime. It also violates some of the main characteristics of CAN, as it changes the addressing mode from being message-oriented to being node-oriented. As the identifiers in CAN are also determining the priority of the message, the prioritization is no longer message-based but node-based. If now, for example, a node with a high priority identifier sends out a very long IP-based message, the bus may be blocked for a long time whether the message being sent is of high-priority or not. As the addressing scheme of IP is maintained, the communication mode changes from being broadcast into being singlecast. Besides that, the high level of fragmentation as well as the fact that the header and trailer of each packet are sent within in the data section of the CAN package causes a significant overload.

In order to avoid the drawbacks of simply adding an IP-layer to CAN several researchers have published techniques to directly operate middleware technologies on the Controller Area Network. In [81] for example, Lankes, Jabs and Bernmerl present an approach to allow the usage of a CORBA middleware in CAN-based networks. A similar technology has been introduced by Kim et al. in [77]. However, both proposals assume a static assignment of identifiers to nodes within the network configured at design-time. Furthermore the maximum number of nodes is limited to 16 and 32 respectively. Another method is presented by Kaiser, Brudna and Mitidieri in [76]. The main idea is to deploy a real-time enabled middleware in Controller Area Networks. Again, the approach lacks of the opportunity to assign identifiers dynamically to the nodes but uses static ones instead. One well known technology for high-layer protocols on CAN is the DeviceNet standard [99]. In order to guarantee unique identifiers within the network, the identifier field partly consists of a combination of a vendor specific ID and the serial number of the device. This definition also leads to a static identifier assignment which

is not sufficient within SODA-based applications.

Other approaches are capable of handing out identifiers to the nodes connected at runtime. One of these is presented by Cavalieri in [33] to ensure real-time characteristics of a CAN network by dynamically changing identifiers and in doing so changing the priorities of the messages. Another approach is the standard CANopen [11] which includes a mechanism to change the identifiers of messages at runtime within the protocol stack. This mechanism is used and extended by Zhou et al. in [158] to build a self-organized protocol stack for networked control systems. However, all three approaches mentioned are based on a central master device responsible for managing the identifier assignment within the network. Using such a master-based assignment requires maintaining an extensive database of nodes and their identifiers. Besides that a protocol has to be defined to keep this database consistent even in the event of disconnecting a node spontaneously by cutting-off its power source. Since this scenario is typical for embedded systems it has to be taken into account. Finally, the dependency on a central master device generates a single point of failure and thereby reduces the reliability of the network.

A last approach uses the reserved bits specified in the extended identifier specification (CAN 2.0B). In [152] Yellambalase and Choi introduce an automatic assignment mechanism operating without a central master device. The reserved bits of the extended frame are used as flags within a protocol to negotiate identifiers between the nodes of a network. However, the usage of reserved bits as well as the complicated negotiation mechanism requires the design of a specialized CAN controller and in doing so demands a disproportional effort.

None of the approaches mentioned above is fully capable of running SODA-based systems on CAN with adequate performance. In order to achieve this, the modules identified in phase 2 are implemented as described in the next three sections.

7.5.3.2 Addressing Scheme

One of the most significant attributes of Controller Area Networks is the fact that they are message-oriented. This means that other nodes are able to determine the content of the messages by reading the identifier. This principle allows establishing an efficient communication mechanism using broadcast messages. At the same time, the identifier defines the priority of the message and hereby of the data sent inside. In order to preserve these benefits the addressing structure of the Communication Model has to be structured carefully.

For the purpose of keeping the bus message-oriented, the identifier should allow to draw conclusions on the content of the message. This leads to a scheme including the so called Service Class Address (SCA). On the basis of the definition given by Rocco et al. in [111] a Service Class is defined to refer to a specific functionality implemented as a Service. In an automotive environment this may be, for example, functionality offering to retrieve the current steering angle. A Service Class Address is a number identifying a specific Service Class. A Service Instance on the other hand is an instantiated implementation of this Service Class. Using the SCA as an identifier would preserve the message-oriented communication style as the content of the message would be announced. However, this approach would not allow having several Service Instances of the same Service Class on the same network as the identifiers would not be unique anymore. This is why the SCA is combined with an Instance Identifier (IID). The IID is used to distinguish between several Service Instances of the same Service Class. Additionally the identifier is supplemented with the command that has to be executed by the Service addressed. The combined identifier ensures uniqueness and preserves the

expressiveness given by the message-oriented addressing approach.

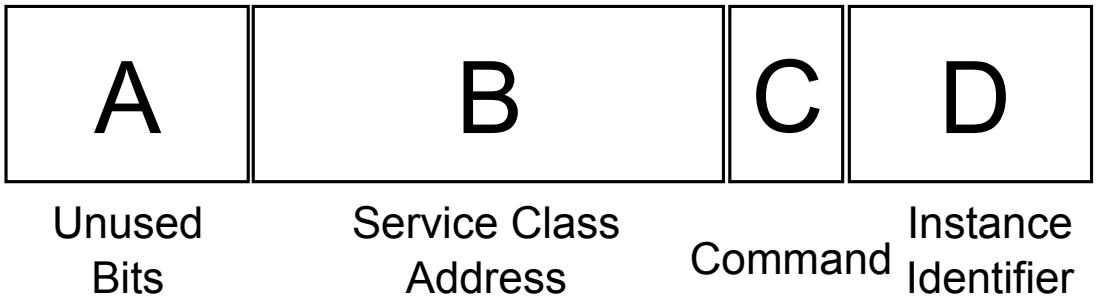


Figure 7.12: The Addressing Scheme developed for SODA Service Communication on CAN.

The overall composition of the identifier is shown in Figure 7.12. The first bits of the identifier, that build section A, are not used within our addressing scheme. In doing so, the demand to allow simultaneous deployment of Services and classical functionalities on the same bus is ensured. This is because only one combination of these bits could be used for the Service-based approach while all other combinations may be used for standard CAN messaging. Furthermore, these bits could also be used to set up virtual channels on the bus offering different priorities. In the automotive domain this could be used for example, to define different channels for safety and comfort applications which, due to the broadcast-style messages, could still exchange data. The next section holds the SCA followed by the command bits included in section C. The least significant bits combined in section D are reserved for the IID. This order allows defining the priority of Service messages and other messages on the bus, as the overall priority is mainly depending on section A. The order of the remaining sections ensures a prioritization primarily given by the functionality which is expressed by the SCA and hereby allows giving important functionalities preferential treatment. The influence of the command and IID fields on the prioritization is rather small. The actual size of the individual sections of the identifier may vary. In this concrete example the extended identifier of CAN, containing 29 bits, has been used. As illustrated in Figure 7.13, section A takes position on the bits 0 to 4 while section B is located between bit 5 and bit 20. The following section C consists of 3 bits and the remaining 5 bits are reserved for section D. This arrangement allows up to 65536 different Service Classes as well as 31 instances of the same class. The number of possible commands is set to 8 which are enough in the light-weight SODA framework.

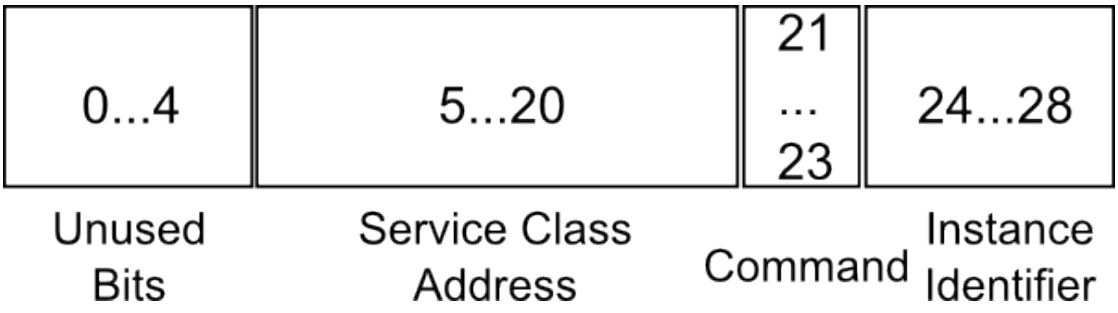


Figure 7.13: The bit lengths used for the Addressing Scheme in the given example.

7.5.3.3 Ability to assign addresses dynamically

The second module of the Communication Model to be implemented in the given example contains a mechanism to dynamically add or remove nodes and thereby Services at runtime. To avoid duplicate identifiers within the network it is essential to automatically assign them in the event of a system change. The approach used in this example is based on the multi-master principle of CAN avoiding the usage of a central network master for administration. Therefore a negotiation mechanism has been developed which enables the nodes to assign identifier among each other. As mentioned before, the identifier consists of three parts. The SCA describes the functionality offered and is hereby predefined for every Service. The command bits are not describing the Service but the action to fulfill and hereby can't be used to ensure uniqueness. This is why the identifier assignment procedure is basically an IID assignment procedure that makes sure that no two instances are using the same identifier.

The algorithm developed is presented in Figure 7.14. It is a recursive, distributed algorithm based on the Bully Algorithm used for electing a leader in a distributed system presented by Garcia-Molina in [55]. After powering up a device the Service running on it is putting itself into idle state until a timeout occurs. The length of this timeout is chosen at random to avoid simultaneous bus access at startup of too many devices. After the timeout the Service reaches the Request state. In this state it waits until the bus is free. Again, it has to be made sure that no two Services with the same SCA are accessing the bus at the same time. Therefore the node waits again for another random period of time. Then, it starts to send out messages to request an identifier. The message is send as a Remote Transmit Request containing no data bytes, carrying an identifier composed of the SCA, the command for a Service Request and the starting Instance Identifier which is 1. All other nodes within the network are now called to check this message. If there is a Service in the network already assigned to this identifier it has to answer the request within a specified period of time. The requesting Service holds the state "Waiting for Answer" until this period of time is elapsed or some other node has answered to this request. If there has been no answer, the algorithm finishes by assigning the request IID as the IID of the Service. If there has been an answer to the request, the IID is increased by 1 and the process of sending out a request message starts again. The algorithm stops after exceeding the maximum Instance Identifier without assigning an identifier to the Service and disables it.

This negotiation approach allows to automatically add nodes by assigning identifiers to the Services running on them. Furthermore the absence of a central network master as postulated earlier is maintained. Finally it assures stability after initial configuration since Services that already have an identifier assigned, do not have to be re-configured when a new node comes into the network.

7.5.3.4 Trigger condition

The last task to be executed in order to fill the gaps of the CAN protocol regarding the example application is to implement the Transmission Arbitration component of the SODA middleware. More specific, this component has to enable the Services to send Service calls periodically.
The implementation of this functionality is carried out by introducing a periodic message handler based on state machines. Within this handler a small state machine for each message which is marked to be periodic is implemented. This state machine has two states namely "idle" and "send periodic message". After initialization the state machine changes to the "idle" state. This state is left as soon as the periodic time defined by

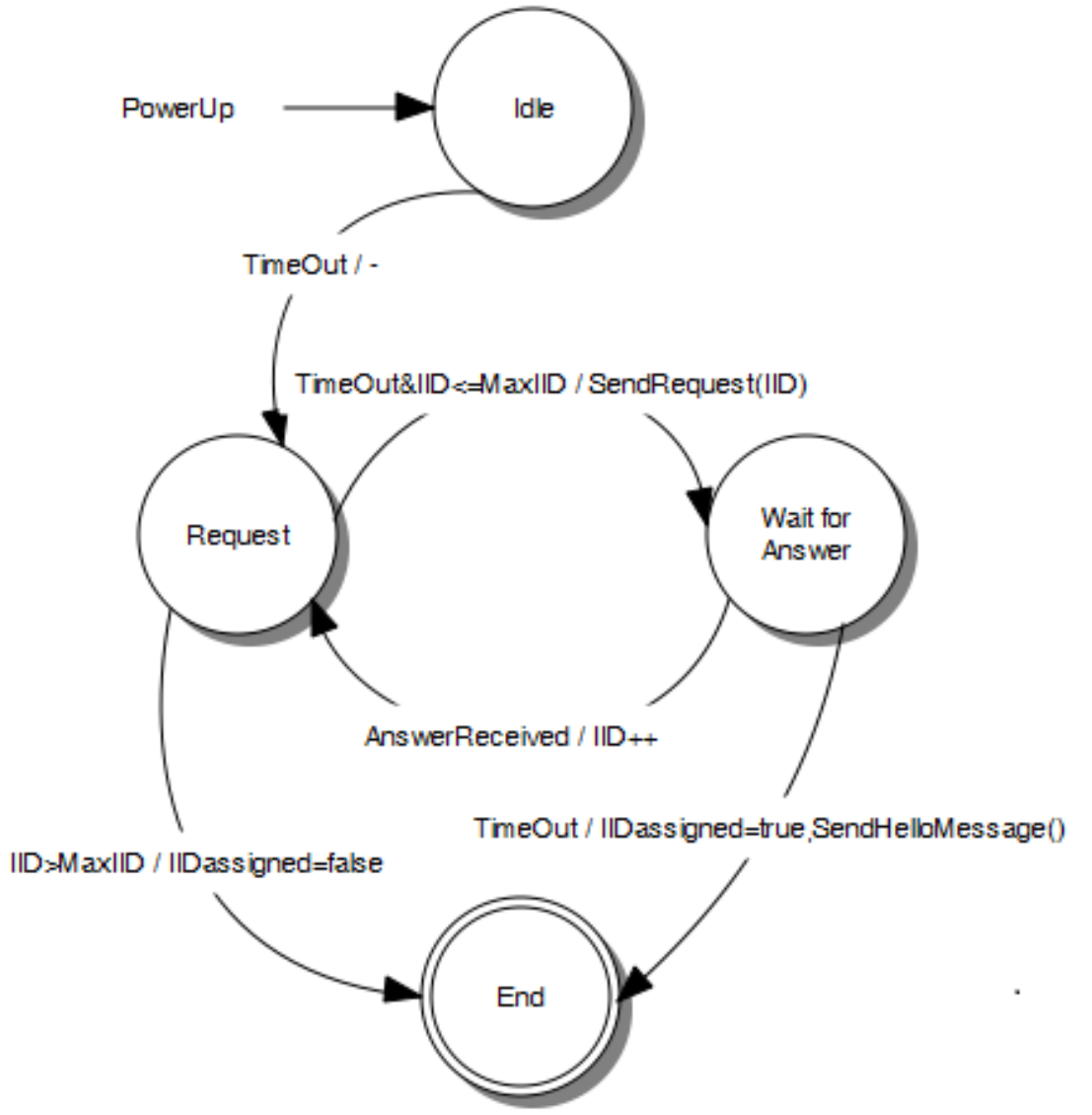


Figure 7.14: Identifier Assignment Algorithm.

the message is expired. The state machine then reaches the "send periodic message" state where the Service call is handed over to the hardware drivers to be sent over the network. Finally, the state machine changes back to the "idle" state where it waits for the next timeout event. Figure 7.15 illustrates this simple state machine for one periodic message. Please note that in order to build an efficient and tailored SODA middleware the Transmission Arbitration module should only be integrated in those Services that have to send at least one message on a periodic scheduling.

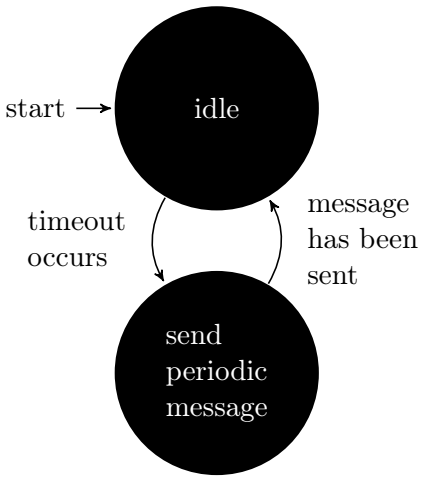


Figure 7.15: State machine for a periodic message.

7.5.4 Assessment of the running example

In order to validate the functionality and the performance of the approach experiments have been run. For these experiments the extended identifier of CAN as well as the section lengths as presented in Figure 7.13 have been used. With these specifications experiments have been prepared to run six different scenarios. For all tests the following devices have been used:

- Embedded boards with an Atmel ATmega88 microcontroller running at 18.432 MHz as well as a Microchip MCP2515 CAN controller and a Philips PCA82C251 transceiver.
- The automotive network testing and simulation environment CANoe 7.6.84 (SP4) from Vector.
- A CANcaseXL bus interface from Vector.

In the following scenarios Services have been implemented in two different flavors. Some of them were executed on the embedded ATmega88 ECUs running the address assignment algorithm in C. Other ones were simulated being implemented in Vector's CAPL programming language running within the CANoe simulation. The former Services had a value for timeout when waiting for an answer to a request of 10ms. For the latter ones the value for timeout was set to 100ms. The assignment negotiation is run by Service Instances all belonging to the same Service Class. The experiments are logged using the trace function of CANoe. The data transmission rate of the network was set to 500 Kbits/s.

Scenario 1 The first scenario contains four Services, running on an embedded board each. All of them were programmed to run the identifier assignment algorithm immediately after power up. The boards were powered up simultaneously and start to obtain identifiers. This scenario simulates turning on the ignition when starting a car.
The experiment succeeded as all four participating nodes assigned themselves with a unique identifier. The time elapsed between the start of the process and the assignment of the last node was always less than 50ms.

Scenario 2 In the second scenario the same four embedded boards were used again. This time only three of the boards were powered up in the beginning. The fourth one was started after the Services running on the other ones already successfully executed the algorithm. This scenario simulates adding a Service at runtime into an already

initialized network.
Again, the experiment went well. The three boards powered up at the beginning negotiated the identifier assignment in little more than 30ms. The fourth module ran successfully through the algorithm in up to 32 ms.

Scenario 3 This scenario combines the four embedded boards used in the other scenarios with three simulated Services to increase the number of nodes involved. As in scenario 1 all nodes are powered up at the same time to simulate the start of a car.
Through to the longer timeout of the simulated Services it took about 430ms to assign identifier to all seven nodes. This time could be decreased significantly using lower timeouts in the nodes ran in CANoe.

Scenario 4 Again, a mixture of Services is used. Scenario 4 consists of four embedded boards as well as 4 simulated nodes. The embedded boards are powered up at the same time as three of the simulated ECUs. The fourth simulated ECU has already assigned an identifier at startup.
As the first IID is already in use the newly added nodes compete starting with the second IID. In comparison to scenario 3 the number of nodes increased which led to a longer assignment time. In this experiment it took about 470ms until the last Service successfully obtained an identifier.

Scenario 5 This experiment represents a scenario were nodes are added on multiple events at runtime. One simulated Service is already running at the start of the test. Three more simulated ones as well as three running on embedded boards are joining the network some time later. Finally, one last embedded ECU comes into the bus and requests an identifier.
The experiment ran well with timing characteristics comparable to the ones in the former scenarios.

Scenario 6 In scenario 6 all formerly used simulated and embedded Services are involved again. In order to validate that identifiers no longer used can be re-assigned again, the following sequence was executed: One simulated Service was pre-assigned with an identifier. In a next step three embedded and three simulated ECUs were powered up. As soon as all identifiers are assigned one of the embedded Services is switched off. In a last step the remaining embedded board is powered up to obtain an identifier.
As a result the lastly connected ECU obtained the identifier originally assigned to the one that already left the network. The experiment proved that identifiers originally assigned to nodes that are no longer part of the network are automatically free to be used by newly added Services.

In addition to the experimental measurements the address assignment algorithm has been evaluated by calculating the worst case times. The scenario created is universal in a sense that all other scenarios can be derived from it. First of all, there might be nodes within the network that use fixed identifiers. As the identifiers of these nodes cannot be used anymore but might be requested by some node running the assignment algorithm each of these nodes causes a delay. This delay consists of the time needed for a Request Frame (t_{RF}), the Data Frame carrying the answer to the request (t_{DF}) as well as the maximum length of the timeout before sending another request (t_{TOR}). This delay occurs for every node with a fixed identifier and therefore is multiplied with the number of these nodes ($\#_{ENODES}$). Equation (7.1) presents the total delay caused by these nodes.

$$t_{ENODES} = \#_{ENODES} * (t_{RF} + t_{DF} + t_{TOR}) \tag{7.1}$$

Besides this delay, the assignment algorithm itself also needs some time to execute. The workflow of the algorithm shows that a node waits for the timeout after the bus is free. In worst case this is the maximum timeout (t_{TOR}). After that, the node sends out a Request Frame which consumes t_{RF} to be proceeded. The whole sequence has to be repeated for every single node which assigns an identifier ($\#_{NNODES}$). The overall amount of time for this sequence is calculated by equation (7.2).

$$t_{NNODES} = \#_{NNODES} * (t_{TOR} + t_{RF}) \tag{7.2}$$

To calculate the overall time elapsed until the assignment algorithm finishes t_{ENODES} and t_{NNODES} have to be summed up. Besides that, the maximum timeout in the beginning of the algorithm (t_{TOS}) as well as the maximum time that a node waits for an answer before it is allowed to use the requested identifier (t_{WA}) have to be added up. The overall time consumed by the assignment procedure in worst case is calculated as stated in equation (7.3).

$$t_{Overall} = t_{ENODES} + t_{NNODES} + t_{TOS} + t_{WA} \tag{7.3}$$

This universal equation might be applied to all possible scenarios. In a normal startup scenario like the one simulated in scenario 1 of the experiments the number of existing nodes ($\#_{ENODES}$) is set to zero. The worst case times for scenario two could be calculated setting the number of existing nodes ($\#_{ENODES}$) to three and the number of new nodes ($\#_{NNODES}$) to one. The worst case times can now be calculated and compared to the times actually measured in our experiments. Scenario 4, for example, would calculate a total amount of 641.32 ms. Since the total measured time averages to 470 ms this is within the expectations. Calculating scenario 3 the ratio is 540.93 ms calculated worst case time to 430 ms measured time during the experiments.

7.6 Summary

In this chapter the Communication Module of the SODA framework has been described. This layer is highly important since it allows the Service-oriented communication mechanisms to be transported using state-of-the-art automotive network systems. This is crucial since all common SOA frameworks make use of IP-based communication which is not a standard in today's cars.

In order to analyze the state-of-the-art and develop an efficient Communication Model, four requirements have been defined: This layer must allow to add or remove Services at runtime. Besides, it has to ensure the advantageous attributes of the used network. Furthermore, interoperability has to be guaranteed. And finally, the last requirement defined is to achieve stability of the Services after initial configuration.

In a next step state-of-the-art SOA frameworks and high-level protocols for automotive networks have been analyzed using the four requirements defined. This step has shown that besides being restricted to IP-based networks most existing SOA frameworks are causing a tremendous overhead by either exchanging whole xml descriptions or making use of large messages to transmit Service calls. Furthermore, none of the examined high-level protocols for automotive networks fulfills the requirements defined.

As a result a unique, SODA specific Communication Model containing four major modules has been developed. The first one of these modules, called Addressing Scheme,

translates the abstract Service calls into actual messages and vice versa. The second one named Adaptation is in charge for allowing to dynamically add or remove nodes to or from the network. Module number three, called Segmentation, divides and reconstructs large Service calls into message-sized junks in order to transmit them through the network. Finally, the last module of the Communication Module takes care of the transmission arbitration.

All the modules named are not static implementations but functional entities that are tailored specifically to the needs of the application/network combination and the Service that owns them itself. This approach ensures to create efficient implementations with a low memory footprint that can be easily integrated even into very small embedded systems. This tailoring is done using the SOAcom process model. It contains four structured phases that guide the development team that implements the Communication Model. The first phase analyzes the SoaML model of the application to derive it's requirements regarding the communication channel. This can be done either manually or automatically using the xml description of the application. In the second phase an easy to use questionnaire guides the developer in determining specific characteristics of the network planned to be deployed. Both results, the requirements of the application as well as the network characteristics, are matched in phase three to identify gaps that have to be filled by the Communication Model's implementation. This third step is carried out by using flow chart diagrams. These diagrams ensure both, the simplicity of usage as well as the possibility to transfer them into code to execute this phase automatically. The result of phase three, a comprehensive specification of the Communication Model, is used in the last phase that finally constructs the modules by writing code.

The overall concept of the Communication Model and the SOAcom process model has been evaluated in an exemplary development. Therefore, a real-world example has been chosen. The application used was a visual driver assistance system for a car with a single-axle trailer. The seven Services within this example were connected using CAN. During the process four gaps between application requirements and network characteristics have been detected and solved using SOAcom. Furthermore, an exemplary Addressing Scheme and Adaptation algorithm have been presented in detail.

The evaluation proved the structured and straight-forward nature of the SOAcom development process. It also showed the high degree of traceability being able to follow each section of the implementation all the way back to the gap that caused all later development steps. The impemented Communication Model proved to be very efficient in terms of adaptation times and memory footprint. This fact is a result of the unique approach to use a well tailored Communication Model rather than inserting pre-implemented code.

8 Integration of the SODA middleware into AUTOSAR[1]

'It's hard to read through a book on the principles of magic without glancing at the cover periodically to make sure it isn't a book on software design.'

Bruce Tognazzini[2]

8.1 Introduction

As described in the former chapters of this thesis the Service-oriented Driver Assistance framework (SODA) has introduced many unique techniques to enable Service-oriented Computing within the domain of automotive embedded systems. While maintaining the main benefits of SOA, SODA has suggested solutions within several different problem areas such as communication, development processes or re-configuration algorithms. One last open point in order to bring down a round figure is to discuss the compatibility of the framework with the most used automotive software standard of the last few years. This standard is named AUTomotive Open System ARchitecture and is widely known under its abbreviation AUTOSAR.

The integration of SODA into the layered software architecture of AUTOSAR sets up 3 different requirements:

- Integration of the Service logic into the AUTOSAR architecture
- Integration of the SODA middleware into the AUTOSAR architecture
- Introduction of runtime adaptivity to the communication modules of the AUTOSAR architecture

The first requirement is to find a proper position for the logical components within a Service which implement the actual functionality and integrate them into the architecture. Hereby, the interfaces of the Service logic component shall be of a type defined in the AUTOSAR specification to ensure compatibility to the surrounding modules rather than introducing legacy interaction points.

A second issue is to insert the two layers of the SODA middleware, namely the SOA middleware and the Communication Model, into the standard's architectural structure. The crucial point here is to stay as close as possible to the original layout to minimize the effects on surrounding modules and to keep the overhead caused by this addition as small as possible.

[1] This chapter is based on my publication [146]. Parts of it are extracted from this source.

[2] 'Principles, Techniques, and Ethics of Stage Magic and Their Application to Human Interface Design, see [132]

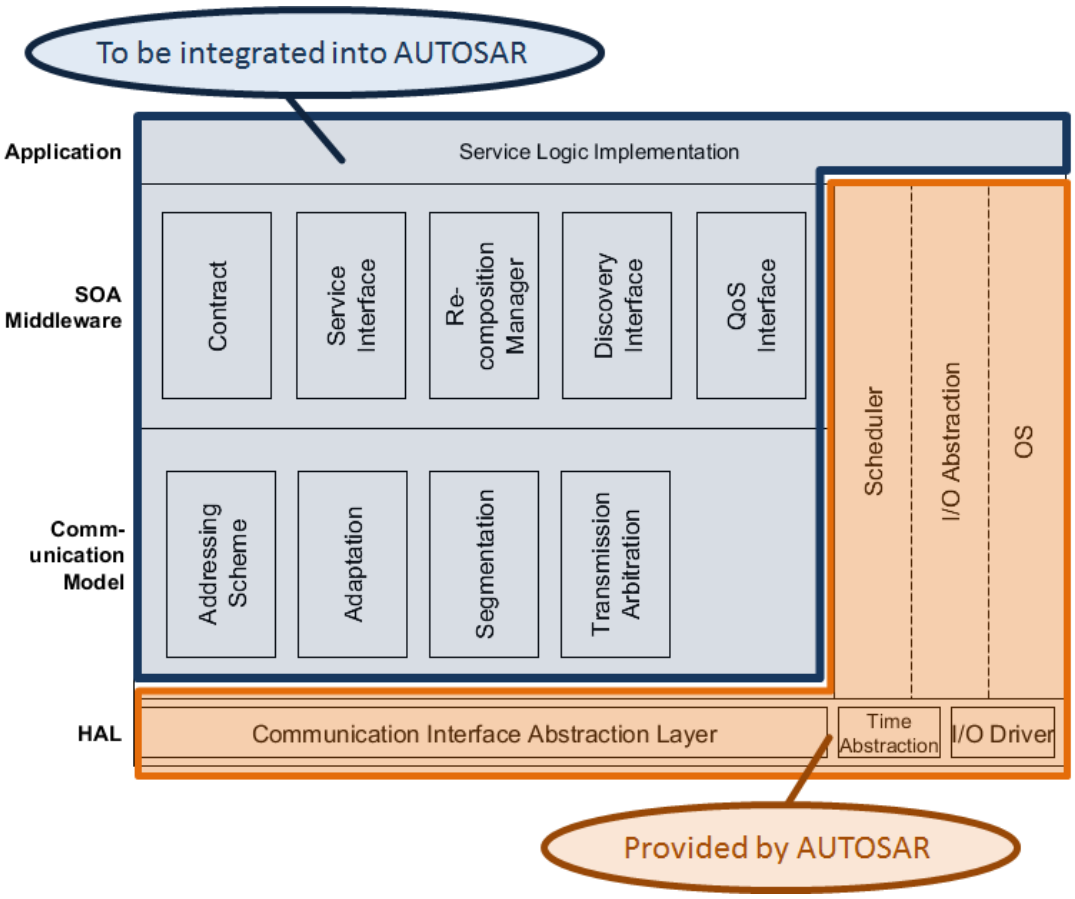


Figure 8.1: The modules of SODA and their relationship to AUTOSAR

Finally, the modules of AUTOSAR handling the communication have to be adapted as well. This is because in the AUTOSAR methodology all modules are static. Regarding communications, this means that all specified messages, routes or handles can't change at runtime. However, since Service-oriented Computing requests a certain adaptability and flexibility AUTOSAR needs to provide this feature. Again, the main goal hereby is to keep the intervention as well as the overhead as low as possible.

As illustrated in Figure 8.1, all parts of a Service's implementation besides the Service logic and the two middleware layers are provided by AUTOSAR. This includes management components such as the scheduler and the OS as well as the abstraction layers to the hardware.

The remainder of this chapter is structured as follows. Section 8.1.1 introduces the main structure and principles of AUTOSAR which is of interest within this context. A discussion and comparison of other approaches to add runtime adaptation to AUTOSAR is presented in section 8.1.2. Section 8.2 presents three different approaches on the integration of the SODA framework into AUTOSAR. Finally, section 8.3 summarizes the contributions presented in this chapter.

8.1.1 The AUTomotive Open System ARchitecture

The AUTomotive Open System ARchitecture is a standardized architecture as well as a development cycle for automotive embedded systems. Figure 8.2 illustrates the overall structure of its architectural blueprint. Note that all illustrations of the AUTOSAR architecture within this chapter do not show a complete view but visualize only those

parts that are of interest in the respective context. As the focus of this chapter lays on the architectural integration of SODA rather than an adaptation of the development processes the AUTOSAR design methodology will not be described here.

The topmost layer is the Application Layer. It holds the AUTOSAR software components (SW-C). These self-contained entities carry the implementations of the functionality offered by the ECU. In the AUTOSAR methodology these SW-Cs can be easily reused and moved to different ECUs during the design process. This is due to the fact that the Runtime Environment (RTE) underneath realizes a so called "Virtual Functional Bus" (VFB) for each particular ECU (see [2]). The main idea of this VFB is that it provides a virtual link to other SW-Cs. A calling SW-C simply uses this virtual link to invoke another SW-C. Through to the high level of abstraction provided by the VFB, this can be done without any knowledge whether the corresponding entity is located on the same ECU or on any other one within the system. The RTE as the local implementation of the VFB on the other hand is aware of these circumstances and routes the request to the target destination. In case the requested SW-C is located on the same ECU, intra-ECU-mechanisms are used like for example direct function calls. In the event of a request to a functionality located on any other ECU within the vehicle, the RTE makes use of static routes calling one of the communication channels available. Within an AUTOSAR development process the code of the RTE of each ECU is generated by a special tool that analyses the distribution of the SW-Cs within the system.

Below the RTE the Basic Software (BSW) stack of AUTOSAR is located. It contains all abstraction layers for the ECU such as for example communication interfaces, input and output pins or the memory of the microcontroller. Using this abstraction the SW-Cs are able to access hardware functionality through the RTE. One main idea behind AUTOSAR is that the software modules building these layers are all well defined in their functionality and interfaces. Through to this fact a company developing an AUTOSAR compatible ECU is able to buy these modules as off-the-shelf products possibly even from a number of different providers. The BSW can be divided into the services layer (blue), the ECU abstraction layer (green) and the microcontroller abstraction layer (red). The services layer is the most abstract one among the layers of the BSW. It contains functionality like the operating system, ECU state and mode management as well as an abstract interface to the memory and the communication modules of the ECU. The ECU abstraction layer underneath adds an additional structure aiming on increasing the hardware independence of the services layer. It offers a programming interface to the hardware drivers that can be accessed by the services layer or directly by the RTE in some cases. The lowest layer of the AUTOSAR BSW is called microcontroller abstraction layer. Its implementation depends on the actual hardware used. It offers the functionality of this hardware to the upper layers of the BSW.

The last software module illustrated in Figure 8.2 is the Complex Drivers component. It spans all the way from the RTE to the hardware of the ECU. Thereby it bypasses all layers introduced before. This allows the introduction of non-AUTOSAR compliant software components into the BSW. Furthermore it can be used to migrate legacy software or functionality with high timing constraints into AUTOSAR. Besides this special case of the Complex Drivers component all interfaces between the different layers as well as the software modules within them are specified in the AUTOSAR standard.

In a typical development scenario the software components to be integrated into a system are providing description files that list their characteristics as well as their hardware requirements. The team of developers collects these files together with the descriptions of the ECUs available in the system. In a next step the SW-Cs are as-

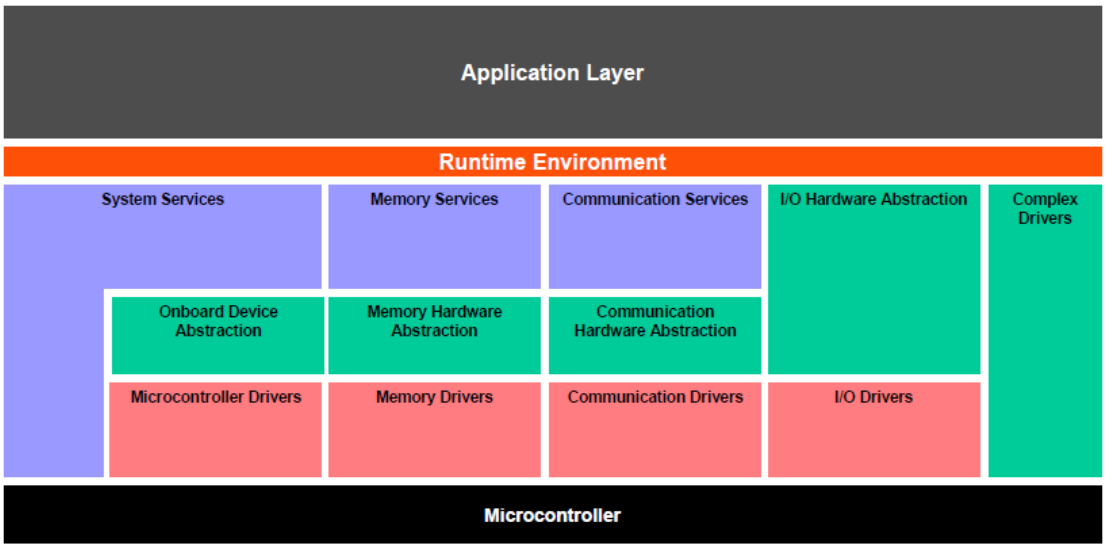


Figure 8.2: The layered software architecture of AUTOSAR [3]

signed to different ECUs in order to create an efficient overall system. This assignment is used by a specialized tool that analyzes the description files of the SW-Cs and the ECUs and then generates the code of the RTE and the modules of the BSW. This code generation includes handles for hardware components as well as static look up tables of the Signals and data packets that are exchanged by the SW-Cs on each ECU.

8.1.2 AUTOSAR in runtime adaptive systems

In recent years there have been a number of publications discussing the issue of creating runtime adaptive ECUs complying to the AUTOSAR standard. These approaches can be divided into two different groups targeting on two different scenarios. One group focuses on the switching between different configurations of SW-Cs at runtime. The second one actually tries to add or remove SW-Cs from the AUTOSAR ECUs while the vehicle is running.

For the first group which uses configuration switching as a basis in order to change the system four different approaches have been identified. The first one, proposed by Zeller et al. introduces an AUTOSAR extension in [156]. This extension adds two additional SW-Cs to the Application Layer of the AUTOSAR-based system: An Adaptation Service and an Adaptation Manager. On the Application Layer, these two components are in charge of activating and de-activating SW-Cs installed on the device. Additionally they interfere with the lower layers as they control a Directory Service added to the BSW. This Directory Service adapts the communication routes going across the AUTOSAR layers for the different configurations. However, this approach has major drawbacks. All possible interactions of the SW-Cs and all potential routes of communication have to be defined at design time just like in a conventional AUTOSAR system. This leads to an overhead within the architecture caused by all the routes and handles compiled into it that might be used very rarely or even never at all. Despite this issue this approach also adds complexity to the development process as all possible configurations have to be determined and modeled during the design process. Finally this approach adds a software component that does not exist in the original AUTOSAR BSW. In doing so potential interference with other components within the stack must be eliminated.

An approach quite similar to the one of Zeller et al. is presented by Berger and Tichy

in [22]. The authors integrated the Operator-Controller-Module (OCM) approach for self-adaptive mechatronic systems to the AUTOSAR architecture. OCM is a architectural model used for example in factory automation to introduce self-x properties. It specifies three different control loops namely a motor loop that actually controls the mechatronic device underneath, a reflective loop that allows to monitor and change the configuration of the entities within the motor loop and finally a cognitive loop that gathers information on the system itself as well as its surroundings to improve the re-configuration mechanisms using a behavioral approach (see [59]). Berger and Tichy combine this architectural style with AUTOSAR in order to allow runtime adaptation. However, this solution is quite limited since it causes the same problems as the approach of Zeller et al. by not adding adaptability to the BSW but integrate all possible configurations at design time instead. Hereby it suffers from the same drawbacks such as overheads in the development process as well as in the final implementation.

The limitations of the previous proposals are also given for the approach of Trumler et al. ([133]). While also based on switching between different configurations of SW-Cs, the idea of the authors contains a negotiation mechanism that is run on different ECUs in a system. This mechanism decides on which configuration should be instantiated in each scenario. This approach draws requirements to the AUTOSAR BSW that are quite similar to the ones of the SODA framework as the SW-Cs to be activated or de-activated are distributed over different devices within the car's network. However, just like in the two former approaches, a static BSW is used which includes all routes and handles possibly needed within any foreseen scenario. This, of course, creates a huge overhead and does restrict the system to those configuration patterns discovered, defined and implemented at design time.

The last approach to be named within this group has been presented by Becker et al. in [16]. Just like the other publications before it is a technique to switch between pre-defined configurations without adding real flexibility to the BSW. However, this approach goes one step further by not only defining a configuration for the communication handles within the BSW but for the overall software architecture of the ECU. A additional component called State Manager organizes the switches between these different configurations at runtime and deploys the pre-defined one for each upcoming situation. Although being integrated in a well defined and AUTOSAR compatible development procedure this approach is even worse in terms of development and implementation efficiency when comparing it to the proposals of Zeller et al., Berger and Tichy and Trumler et al. This is because developing a single configuration equals to the development of a complete ECU in the conventional AUTOSAR procedures.

The second group of approaches has a different perspective. In order to allow upgrades or extensions to AUTOSAR-based ECUs that are already running within a vehicle, these proposals target on dynamically adding and removing SW-Cs at runtime. The first one, presented by Zeeb in [155], intends to enable installing software updates at runtime. This is done by reserving memory capacity for future software variants at design time and re-flashing parts of the memory in the event of an update while maintaining the subroutine addresses of the SW-Cs. This focus on updates sets up two main restrictions. First of all, it is limited to the update of already installed SW-Cs rather than allowing to add additional ones. Second, since those updated components do use the same interfaces to the RTE and the BSW no changes within these layers are necessary. Hence, no flexibility is added to these modules within this approach of Zeeb. This results in the conclusion that, although this proposal adds some kind of runtime adaptivity to AUTOSAR it does not add any dynamics to the modules underneath the Application Layer.

The main goal of another approach presented by Axelson and Kobetski in [12] is to plug in new components at runtime. Therefore the authors suggest to add two SW-Cs to each AUTOSAR ECU: the first one holds a virtual machine to run Java applications. The second one which is called "external communication manager" establishes a direct connection to a external software source. The approach follows the idea that software components developed in Java are downloaded via the external communication manager and installed into the virtual machine at runtime. However, the paper leaves unclear how the Java components downloaded are interacting with the static RTE and BSW underneath since no solution is presented on how to make the communication stack more flexible. This leads to the assumption that those Java-based software modules can either only access a pre-defined set of channels to the hardware and communication devices or are not interacting with the remaining parts of the ECU and the overall system accessible through the networks. This does not only reduce the fields of application but also does not provide any answer to the requirements on a dynamic BSW as needed for Service-oriented system design.

The last approach to be discussed is presented by Martorell et al. in [90]. Just as Axelson and Kobetski the authors extend AUTOSAR to allow the addition of new SW-Cs at runtime. However, the technique used to realize it differs significantly. Martorell et al. suggest to add several empty SW-Cs to the architecture. These can be filled with runnables that add new functionality to the ECU at runtime. However, just like the proposal of Axelson and Kobetski the approach uses a static configuration of the BSW instead of introducing some kind of flexibility. This means that all potential hardware handles and communication routes have to be foreseen and implemented at design time. As discussed earlier in the approaches of the first group, such a technique causes a significant overload in terms of design time as well as in the memory footprint of the final implementation. Furthermore, it can be argued that due to the long life cycles of automobiles a system designer is not able to estimate the hardware handles and communication scenarios of implemented software components during the whole machine life.

The discussion of the existing approaches to integrate runtime adaptability into AUTOSAR is summarized in Table 8.1. As shown in this table, the suggestions vary in their level of integration and the overhead they cause. However, none of them is able to re-configure the AUTOSAR communication stack at runtime which is essential when adding Service-oriented applications to such systems.

8.2 On the integration of the SODA framework into AUTOSAR

As illustrated in section 8.1.2 none of the approaches previously published is able to offer the degree of runtime adaptivity and flexibility needed to run Service-based software applications within an AUTOSAR environment. In particular, those proposals lack of mechanisms and techniques to allow re-configuring the communication stack which is indispensable in Service-oriented Architectures.

In order to overcome these shortcomings three different approaches to integrate the SODA framework with AUTOSAR have been developed. All three techniques provide the necessary flexibility and cause relatively low overhead. Furthermore, they fulfill the two remaining requirements mentioned earlier as they integrate the Service logic as well as the SODA middleware into the AUTOSAR architecture. However, they are different in the way they interact with the different layers and their level of integration into the AUTOSAR BSW. The remainder of this section will describe and discuss these three

Publication	Integration of Service Logic	Level of integration into AUTOSAR	Ability for runtime changes in the communication stack	Overhead
Zeller et al. [156]	✓	medium	×	high
Berger and Tichy [22]	✓	medium	×	high
Trumler et al. [133]	✓	low	×	high
Becker et al. [16]	✓	low	×	high
Zeeb [155]	✓	medium	×	medium
Axelson and Kobetski [12]	✓	unclear	×	high
Martorell et al. [90]	✓	medium	×	high

Table 8.1: Comparison of different approaches to add runtime adaptation to AUTOSAR

approaches.

8.2.1 Integration using the Complex Drivers

The first approach to be discussed makes use of the Complex Drivers module of the AUTOSAR architecture. As mentioned earlier the Complex Drivers are ment to be some kind of bypass to the AUTOSAR BSW. This is because this module directly connects the Application Layer and the RTE to the micro controller. The main idea of introducing such a structure that avoids using any of the BSW layers was to enable developers of highly time-critic applications to get full control over the implementation between their application component and the hardware. However, in the history of AUTOSAR Complex Drivers have often been used to include legacy code that does not comply to the standard into an AUTOSAR ECU as this architectural bypass allows to flexibly add modules that are not subject of the restrictions given in the case of the other BSW components.

Figure 8.3 illustrates how SODA is integrated into AUTOSAR using Complex Drivers. The actual Service logic is encapsulated into AUTOSAR SW-Cs and thereby assigned to the Application Layer. In the example given in Figure 8.3 the Application Layer contains two Service logic implementations. This number is just an example as the actual number of SW-Cs is not limited by AUTOSAR or by this first approach to integrate SODA. The SW-Cs are using standardized AUTOSAR Interfaces to interact with a third SW-C called SODA Component. Such AUTOSAR Interfaces are an important part of the AUTOSAR standard as they specify some kind of interaction point between two entities. Using this interaction point data and services are specified at design time and exchanged during runtime between two SW-Cs by making use of the RTE (see [1]). The SODA component introduced to the Application Layer holds all the modules of the SOA Middleware layer of the SODA framework namely the Contract, Service Interface, Re-configuration manager, Discovery Interface and the QoS Interface. Figure 8.4 illustrates this mapping. Thereby this specialized software component is in charge of the execution of the principles of a Service-oriented Archtiecture such as carrying out Service Discovery or re-configuring the system by conducting Service Composition. This structure makes use of the circumstance that the Service logic SW-Cs are only

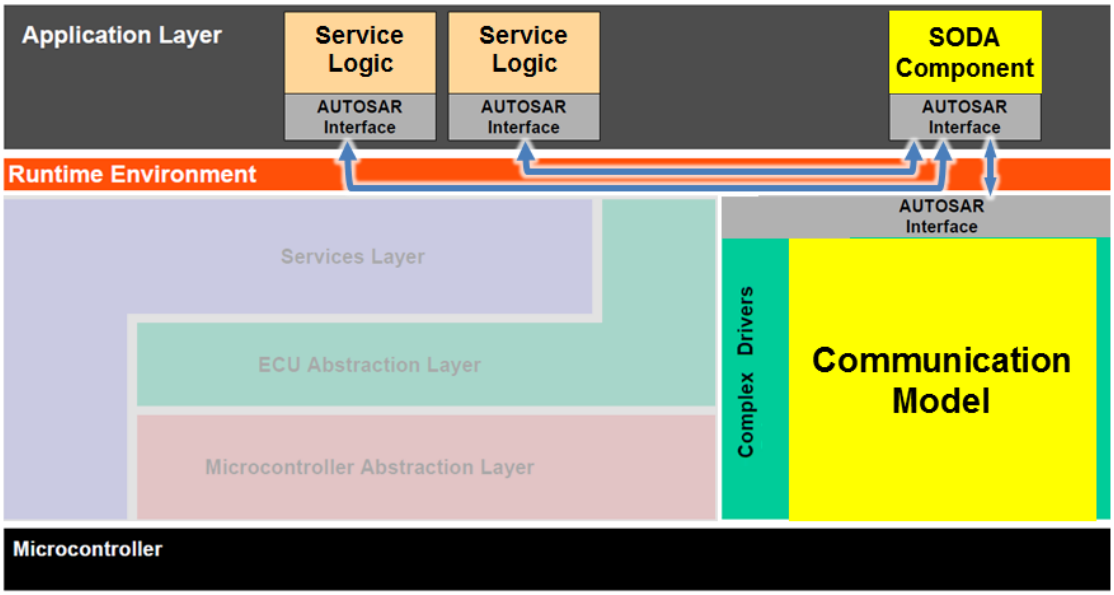


Figure 8.3: Approach 1: bypassing the AUTOSAR BSW by using Complex Drivers (figure bases on [3])

connected to the SODA Component SW-C. Thereby the data and services exchanged between these units are known at design time. This is due to the way SODA systems execute adaptation. In this framework, the adaptation takes place on the architectural level which means that ECUs join and leave the overall system. Inserting or removing software functionality as well as component migration between ECUs is not designed within SODA. Since the number and type of SW-Cs as well as the connections between these entities and the SOA middleware layer beneath are specified during the development phase no changes at runtime are taking place at this point.

The SODA Component SW-C itself interacts through an AUTOSAR Interface with the Complex Drivers module as shown in Figure 8.4. Within this part of the AUTOSAR architecture the second part of the SODA middleware namely the Communication Model is implemented. Being in charge of the abstraction of the communication channels like for example CAN or LIN it hereby offers a direct channel between the SOA middleware and the communication hardware. The connection between the two SODA layers "SOA middleware" and "Communication Model" is predefined at design-time and does not change during runtime. This is due to the fact that those two modules exchange Service calls which only depend on the type and number of available Service logic implementations on the ECU which is static as described earlier. The necessary adaptation within the communication channel is completely carried out within the Communication Model. This technique also allows to create a never-changing interface to the hardware entities at the lower end of the Complex Drivers module. During operation, the calls carried out by the SOA middleware are transmitted to the Complex Drivers and transformed by the Communication Model into actual network messages and vice versa. The adaptation carried out is thereby limited to the re-configuration of the modules assigned to the Complex Drivers stack which directly accesses the hardware without interfering with the static modules of the BSW.

The approach of making use of the Complex Drivers is a quite easy and direct way of adding the SODA framework and thereby adaptation capabilities to AUTOSAR. It uses SW-Cs to encapsulate the Service logic implementations as well as standardized AUTOSAR Interfaces to connect them to the rest of the system. The SODA middleware is integrated into the architecture by assigning the SOA mechanisms into an

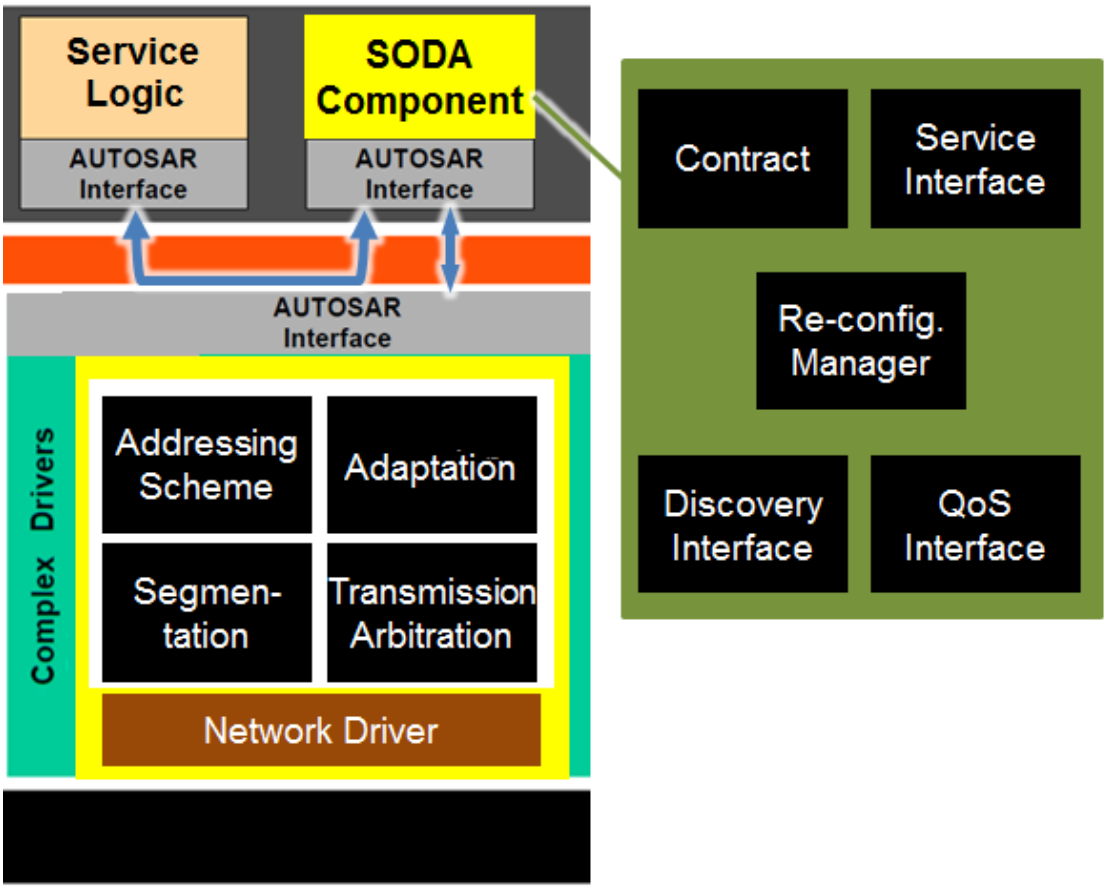


Figure 8.4: Focus on the Complex Driver approach (figure bases on [3])

additional SW-C that acts as a gateway to the Communication Model that is wrapped into the Complex Drivers. The Communication Model is implemented within the Complex Drivers pillar and connects the SOA middleware to the actual communication entities of the hardware. The drawback of this approach is the loose integration into AUTOSAR. By bypassing the whole BSW the ideas and techniques that made the standard popular in recent years as well as a huge part of the AUTOSAR methodology is overruled. Furthermore the Complex Drivers would either have to implement their own communication drivers or would have to get access to the ones of the BSW. In the former case this would bring additional overhead and be problematic since two software drivers would try to access a single hardware entity. In the latter case the AUTOSAR communication drivers would have to be extended to allow access from the Complex Drivers which would lead to non-standard software modules within the BSW.

8.2.2 Integration through replacing the XCP component

The technique to integrate SODA into AUTOSAR using the Complex Drivers sets up several disadvantages as discussed in section 8.2.1. Some of these disadvantages can be avoided by increasing the level of integration. The second approach to be discussed here refrains from creating a bypass but exploits the modules of the BSW instead. Therefore, the XCP protocol as well as its integration into AUTOSAR is examined in detail. XCP is an abbreviation for the "Universal Measurement and Calibration Protocol". It has been designed to read and write to the memory of automotive ECUs through different network interfaces. It is especially used during the development process of a vehicle as it allows to directly overwrite sections of the ECU's memory. Using this technique, parameters, constants as well as code sections can be exchanged quite

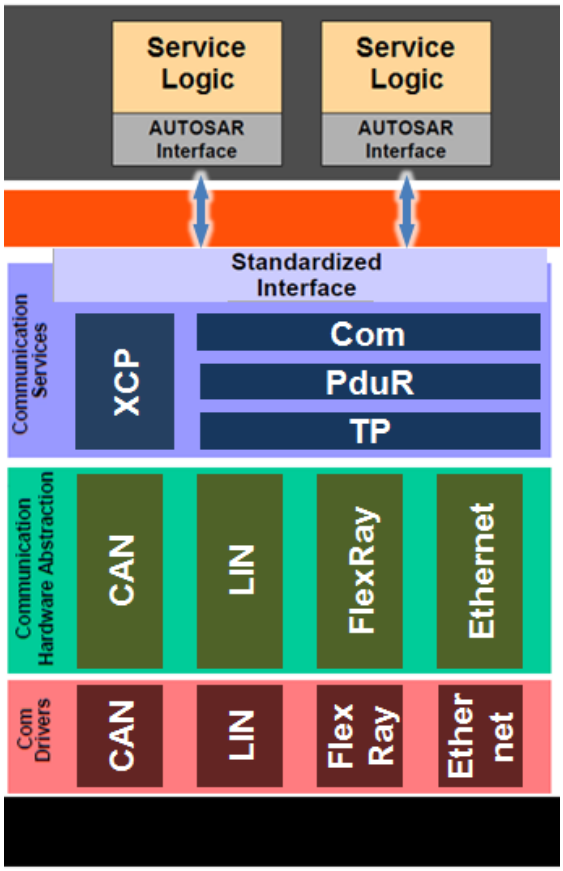


Figure 8.5: Integration of XCP into AUTOSAR (figure bases on [3])

easily in order to be evaluated during test runs. The main benefit of XCP lies in the fact that these changes can be carried out without connecting a specific programming device. Instead, the automotive networks such as CAN, LIN or FlexRay can be utilized for this purpose. Since AUTOSAR release 4.0 the XCP stack is integrated into the architectural blueprint as illustrated in Figure 8.5. In detail, the stack is attached to the Communication Services of the BSW where it bypasses the modules of this layer. The processing of an XCP call is executed in the following way. A message containing an XCP command is sent to the ECU through one of its standard automotive network channels such as for example CAN or FlexRay. This message is proceeded according to the AUTOSAR standard: it is received by the Communication Hardware which is controlled by the specific Communication Driver. This software entity then forwards its content through the Communication Hardware Abstraction layer of the BSW to the Communication Services. Unlike other messages that address SW-Cs the XCP messages are not forwarded to the RTE and the Application Layer but are completely handled within the XCP module. Within the stack the command is interpreted and executed and the memory changes are carried out as desired. The result of this operation is returned to the initial sender by a network message which is proceeded through the lower layers of the AUTOSAR architecture in reverse order.

Unlike the other Communication Services like for example the Transport Protocol (TP) the XCP stack introduces some kind of state-full behaviour to this BSW layer. Rather than just forwarding messages according to pre-defined routes the XCP mechanism has its own execution logic and is able to carry out operations depending on the content of the message and its own state. Additionally, it is able to create and send it's own messages to other entities within the car. A potential way of making use of this new degree of freedom for integrating the SODA middleware into the BSW is to

substitute the XCP stack with a SODA component. As shown in Figure 8.6 the SODA module including both the SOA middleware and the Communication Model replaces the XCP stack. The SW-Cs containing the Service logic implementations are directly interacting with the SODA component through a Standardized Interfaced as specified for all Communication Services within AUTOSAR. According to the AUTOSAR consortium in [1] the only requirement these Standardized Interfaces have to offer is a concrete application programming interface (API) offered to the SW-Cs. This API is given by the interface of the Application Layer and the SOA middleware layer in the SODA framework. By using these Standardized Interfaces AUTOSAR compatibility regarding the interconnection of Service logic and SODA middleware is ensured. On the lower end of the Communication Services layer the interaction to the Communication Hardware Abstraction within the BSW is done through the usage of Protocol Data Units (PDUs). These entities are data structures that contain the properties of a message as well as its content. PDUs are normally determined in their number, format and content at design time. This fact raises a problem since SODA needs to dynamically adapt these PDUs at runtime in order to allow flexible Service communication.

The solution to this problem has been introduced in the latest version of the specification: AUTOSAR release 4.1. Since newly introduced protocols like J1939 or Ethernet also need some runtime adaptability in the two lowest layers a far-reaching change has been done to them. Since AUTOSAR release 4.1 PDUs have an additional property that allows to mark them as PDUs with a changeable address. In this case the address (in CAN this would for example correspond to the Identifier) to be used for sending would be attached to the PDU rather than being archived somewhere in the routing tables of the BSW's communication modules. This addition to the AUTOSAR communication specification can be used by SODA to create messages with dynamic addresses that can be re-defined for every transmission carried out. The technique used here is to introduce a fairly low number of PDUs with changeable addresses. The low number ensures that the created overhead stays quite small. These flexible PDUs are managed by the SODA stack and used to create messages for every possible communication scenario. Using this approach, the Communication Model of SODA is able to realize all communication necessary.

This second approach which makes use of the AUTOSAR XCP module enhances the level of integration significantly compared to the Complex Drivers proposal. It ensures a seamless integration of the SODA middleware into the communication channels of the BSW. It also complies to the AUTOSAR specification in the way the Application Layer is connected to the SODA module. However, one last drawback is left, the need for a SODA specific implementation of the interface between SW-Cs and the Communication Services. In order to overcome this drawback and increase the level of integration even more a third approach is discussed in section 8.2.3.

8.2.3 Integration through transport protocol enhancements

Although the second approach presented increased the level of integration into the AUTOSAR BSW significantly it still requires a SODA specific interface between the SW-Cs containing the Service logic and the SODA stack within the Communication Services. In order to overcome this issue a third approach to integrate Service-orientation into the AUTOSAR architecture has been developed which will be presented and discussed within this section.

This last proposal again makes use of some of the innovations of the AUTOSAR release 4.1. As explained before this new standard introduced the Ethernet and J1939

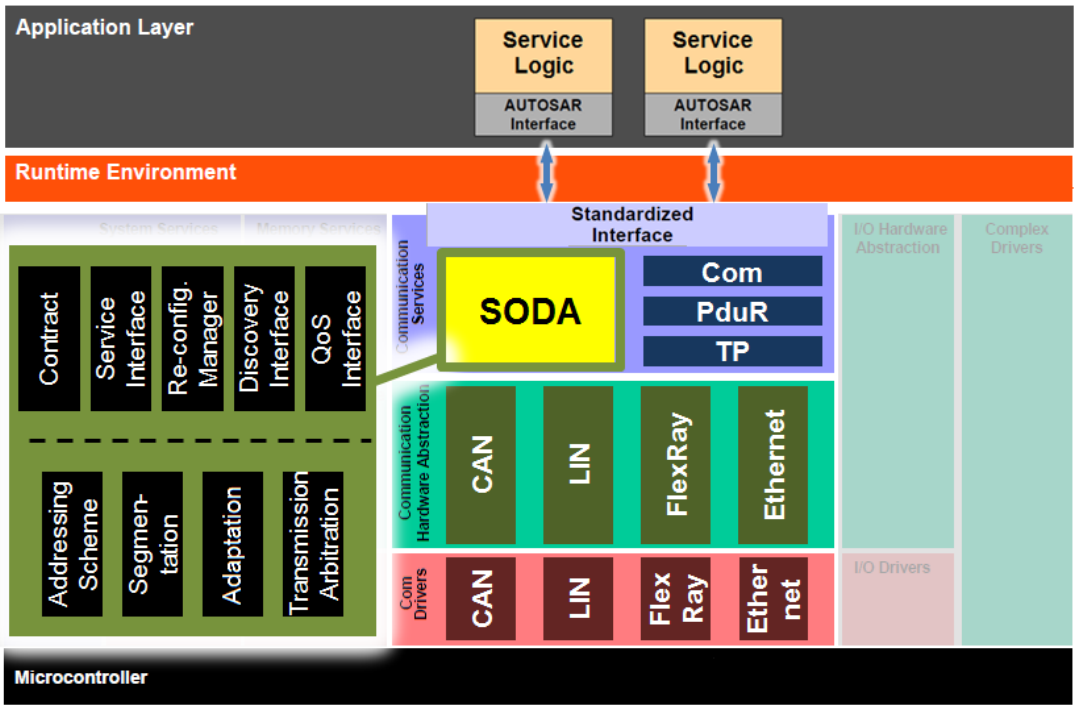


Figure 8.6: Approach 2: medium level of integration (figure bases on [3])

protocol. Although those two networking protocols are quite different in their purpose and technology they have in common that both need some degree of flexibility on the lower levels of the architecture. However, both have to make use of the standard AUTOSAR communication as they are part of the BSW.
In case of the Ethernet protocol this means that its stack has to realize several upper layer protocols such as the User Datagram Protocol (UDP), the Transmission Control Protocol (TCP) or the Address Resolution Protocol (ARP) while complying to the specification of the AUTOSAR communication schemes. However, the IP addresses used are not static but can be assigned to the ECUs within the network using the Dynamic Host Configuration Protocol (DHCP) at runtime which creates the need for some runtime flexibility within the communication pillar of the BSW.
In the case of the J1939 protocol, which is an upper layer protocol on the basis of CAN, dynamic peer-to-peer connections have to be established. In order to realize this the Identifier and the content of CAN messages have to be runtime adaptive rather than being configured at design time.

Figure 8.7 illustrates the integration of J1939 and Ethernet into AUTOSAR. Both use the static approach of AUTOSAR to connect to the SW-Cs via Signals and static PDUs. Hereby a Component on the Application Layer sends a pre-defined Signal to the Com module using the static RTE. The Com module receives the Signal and makes use of a look up table to assign the Signal to one of the pre-defined PDUs. One characteristic of such a PDU is the so called transfer property. It defines the transmission schedule of the entity. This can be either "on change" which means that the PDU is transmitted whenever one of its Signals changes. The other possible option is to simply store new values and send them out periodically. Both types of PDUs are eventually transmitted by being handed over to the PDU Router (PduR). The PduR provides a static routing table. By using this table the module forwards the PDUs to the designated network channel. This is done by handing over the PDU to the Transport Protocol (TP) layer which is normally only in charge of selecting the hardware unit to be used for the transmission. Especially when considering CAN communication it

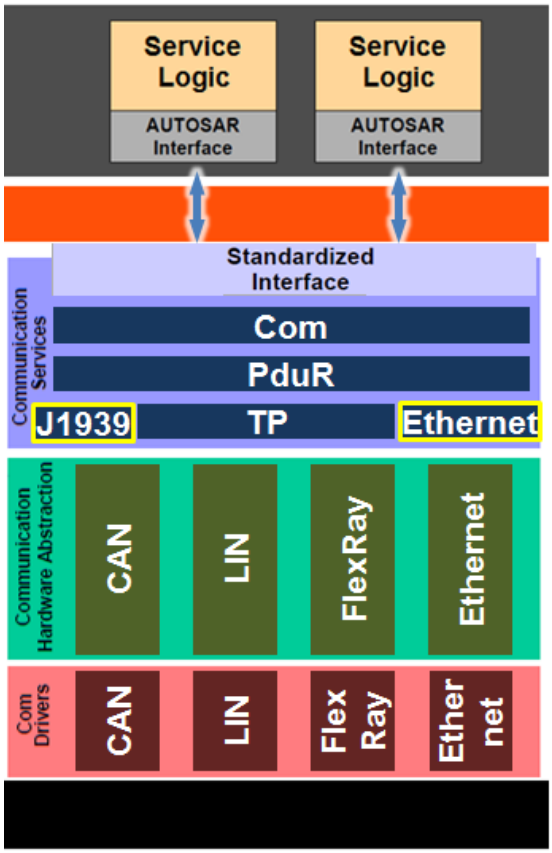


Figure 8.7: Integration of the J1939 and the Ethernet protocol on the Communication Services layer (figure bases on [3])

is quite usual that an ECU possesses more than just one hardware entity. Focusing on the TP layer, the protocols of J1939 and Ethernet are a bit different compared to the other automotive network systems. This is because in these cases the TP layer is not only in charge of forwarding the PDUs to the next layer but executes a much more complex protocol state machine instead. In standard CAN, LIN or FlexRay communication scenarios the next step would be to proceed the PDU to the Communication Hardware Abstraction which transforms it into an actual message which is then forwarded to the Driver layer in order to be eventually sent over to the hardware entity provided by the ECU. In J1939 and Ethernet scenarios this is different since the communication between the SW-Cs and the stacks on the TP layer can be interpreted as commands which are received and realized by the state machines within the TP modules. The communication beneath the TP layer is still controlled by these commands but it is not an immediate through-connection of the Signals initially sent by the SW-Cs.

The principles used by the Ethernet and J1939 modules are applied to the Service-oriented scenario to realize the third approach of integrating SODA into AUTOSAR. It introduces an additional SODA module on the TP layer as illustrated in Figure 8.8. This module can be either replacing J1939 and Ethernet or can be integrated alongside with those two components. The SODA module used in this approach contains all components of the SOA Middleware and the Communication Model layer of the SODA framework. In order to connect to the lower layers and eventually to the communication hardware the PDUs introduced in AUTOSAR 4.1 allowing changeable addresses are used. Just like in the second approach described in section 8.2.2 a low number of PDUs with this feature is instantiated and used to realize the network messages needed in a

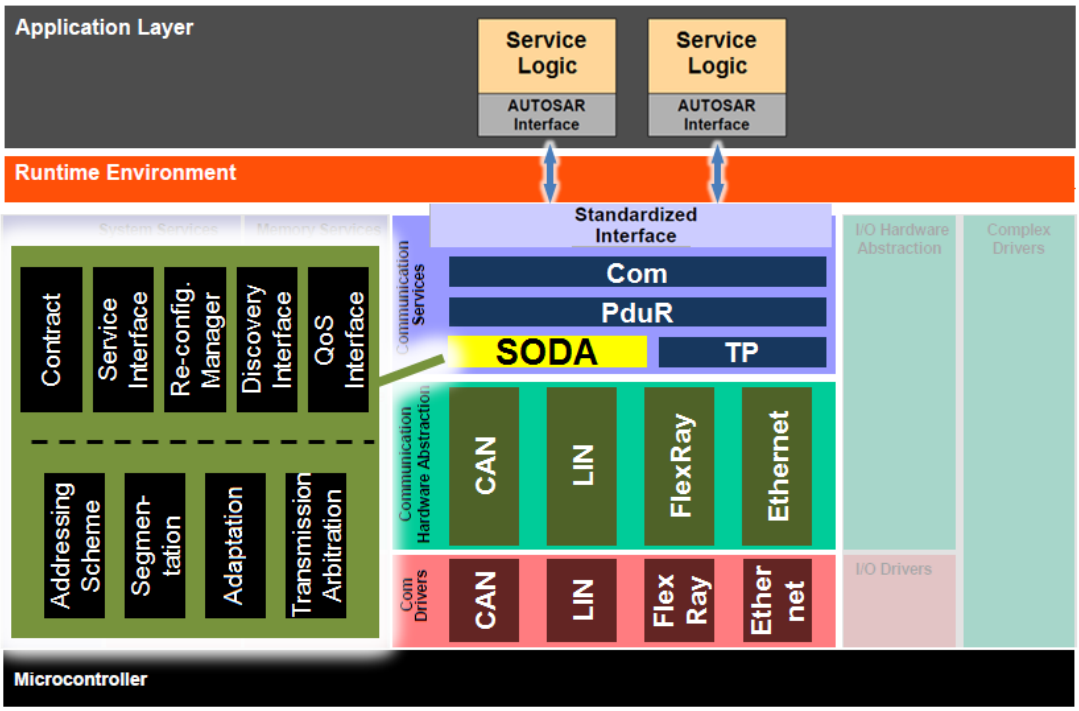


Figure 8.8: Approach 3: high level of integration (figure bases on [3])

very flexible manner.

The interaction of the SW-Cs which contain the Service logic and the SODA middleware consists of a number of commands of procedure calls. Since these interaction handles are known at design time and do not change during runtime static Signals and PDUs can be used. Therefore, each command or procedure call being exchanged between the middlware and the Service logic is represented by a Signal. Each of these Signals is mapped into a PDU which is assigned with a transfer property that ensures immediate transmission in case of a Signal change. By doing so, the Com layer forwards the command directly to the PduR whenever it receives such a Signal. The PduR is statically configured to pass these PDUs to the SODA module in the TP layer. Hereby the command is delivered and can be processed by the SODA framework. This kind of link can also be used in the other direction starting from the SODA module, addressing one of the SW-Cs. Since the number of commands connecting the Service logic and the middleware is very limited and the structure of many of these is quite similar, the number of Signals and PDUs to be created is very low which limits the overhead significantly.

This third integration approach is completely using standard AUTOSAR interfaces to connect the middleware with its surroundings. By including the complete middleware into the Transport Protocol section of the BSW without using any bypasses the level of integration is high.

8.2.4 Comparison of the three approaches

All three approaches discussed in this chapter fulfill the demands set up earlier by providing an integration of both the Service logic and the middleware as well as allowing runtime changes within the BSW. Table 8.2 illustrates the comparison of the three proposals. As this table shows the main difference lies in the level of integration. While the first approach using the Complex Drivers bypasses the whole BSW the two other ones are much more integrated into AUTOSAR's architectural blueprint. The proposal to replace the XCP component still needs to generate a own interface between software

Approach	Integration of Service logic	Level of integration into AUTOSAR	Ability for runtime changes in the communication stack	Overhead
Complex Drivers	✓	low	✓	low
Replacing the XCP component	✓	medium	✓	low
Enhancing the Transport Protocol	✓	high	✓	low

Table 8.2: Comparison of the three approaches suggested to integrate SODA into AUTOSAR

components and the middleware. This last issue is solved by integrating the middleware into the TP layer which allows to make use of standard Signals and PDUs to create this interconnection.

8.3 Summary

In this chapter three different approaches to integrate the Servie-oriented SODA framework and hereby to add runtime adaptation on the architectural level into AUTOSAR have been presented.

The first one makes use of the Complex Drivers component. This stack is part of the AUTOSAR specification and can be used as a bypass to avoid using the static modules of the BSW. The main advantages of this technique are its simplicity and the ability to use it right away within the current AUTOSAR standard. However, since it completely bypasses the BSW it is fully overruling many of the ideas of the AUTOSAR standard such as for example the usage of replaceable off-the-shelf modules. Furthermore this approach has to implement its own communication drivers which raises problems such as adding overhead and the usage of a single hardware communication channel by two different software drivers.

The second approach presented picks up the ideas of the integration of the XCP module into the AUTOSAR software stack. Hereby a communication module is added in parallel to the layers of the Communication Services. In combination with the extensions introduced by AUTOSAR release 4.1 which allow to introduce flexible PDUs to the lower layers of the BSW, this technique makes it possible to integrate the SODA framework rather than just bypassing a huge part as done by the first approach presented. Nevertheless, this approach calls for changes within the AUTOSAR specification. Furthermore it creates an additional Standardized Interface.

An even higher level of integration is given when using the third approach presented. Hereby the TP layer of the BSW and especially its changes in the AUTOSAR release 4.1 are analyzed. With the introduction of Ethernet and J1939 to the specification the previous mode of operation in which every layer of the BSW only adds relatively small changes to the PDUs based on static look up tables changed significantly. This is due to

the characteristic of the Ethernet and J1939 protocol where the data exchange between the SW-Cs and the corresponding communication module is rather an exchange of commands to control the data flow within the lower layers. In the third approach the SODA middleware is integrated into the TP layer and hereby uses all layers beyond and beneath as they were intended.

As a summary one can say that it is possible to integrate runtime adaptation into the AUTOSAR standard using any of the three approaches presented in this chapter. All of the three proposals made are unique compared to other approaches published in recent years as these three techniques offer the capability to re-configure the communication mechanisms of the architecture at runtime rather than designing all possible configurations at design time. In doing so the overhead caused by the mechanisms presented here is significantly lower. This is true not only for the code size and complexity but also for the workload during the development. These unique advantages may pave the way for introducing Service-oriented computing into embedded automotive distributed systems. However, in order to make future AUTOSAR revisions more flexible regarding upcoming technologies it should be opened and generalized. At the moment new technologies are integrated into the standard after they hit the market which causes a huge delay and a high effort. Instead, the architecture and especially the modules of the BSW could be defined more openly and abstract to allow a seamless integration of upcoming technologies without changes on the architectural blueprint itself.

9 Evaluation

'Everything that can be counted does not necessarily count; everything that counts cannot necessarily be counted.'

Albert Einstein

9.1 Example application description

This chapter evaluates the SODA framework. This is done by applying the architecture and development process introduced within this thesis on a driver assistance system for car and trailer combinations. The developed system is implemented on a full scale demonstrator consisting of a Mercedes B-Class car and a small two-axle trailer.

The application used to evaluate SODA is an assistance system that helps the driver to back up a car with a two-axle trailer attached to it. It has been developed within the Real-Time Systems group of the University of Koblenz-Landau and implemented during the Master's thesis of Sascha Berkessel (see [23]). The system makes use of the visual modality by informing the driver through a display within the car's dashboard. This display shows the area behind the trailer. The picture is augmented by additional information through overlaying two different trajectories. Figure 9.1 is illustrating the output of the system to the driver.

The first one of these trajectories, called "A", which is colored in green shows how the center of the rear axle of the trailer would move if the current steering angle is maintained. This allows the driver to estimate the future behavior of the overall vehicle. In addition, trajectory A directly responds on the changes of the steering angle even when the car is stopped. Through to this characteristic the vehicle operator is able to analyze the effect of different steering angles on the future path of the trailer.

The second trajectory augmented is named "B" and colored in blue. The two curves building this trajectory symbolize the future path of the tires of the trailer's rear axle. In contrast to the first trajectory, these skid marks are not corresponding to changes on the steering angle. Instead, they show the trailer's movement assuming that the bending angle between the front and the rear axle of the trailer does not change over time. In this sense, trajectory B shows the path of a stable movement of the combination.

The combination of both trajectories supports the operator of the vehicle in the two main scenarios when backing up such a car and trailer combination. The first one of these scenarios is a stable movement. Here, the driver wants to back up the combination using the path illustrated by trajectory B. In order to do so, the only thing the operator has to do is to keep the single line illustrating trajectory A between the virtual skid marks. This ensures, that the vehicle follows the desired path. In the second scenario, the driver wants to change the stable path of the combination into a specific direction. This can be accomplished by changing the course of trajectory A using the steering wheel into the desired direction. Trajectory B will slowly trail this path until it points to the desired location. As soon as this state is reached, the driver can use the technique

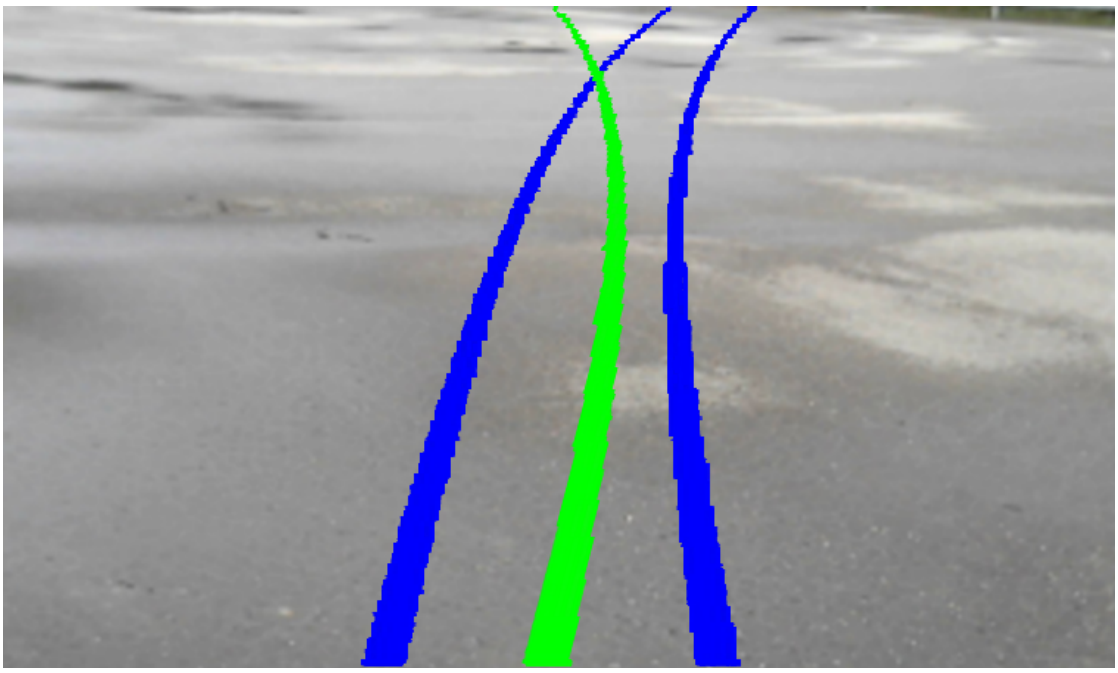


Figure 9.1: The HMI of the visual Assistance System for a trailer

described in the first scenario to move the car-trailer-combination into the required position.

The system as it has been described above requires a number of soft- and hardware entities to offer it's assistance. On the hardware side it needs sensors to determine the current steering angle as well as the two bending angles of the system: the angle between the car and the front axle of the trailer (Bending Angle 1) and the angle between the front and the rear axle of the trailer (Bending Angle 2). Furthermore, a camera recording the area behind the trailer as well as a monitor to output the assistance to the driver are needed. On the software side, the measurements of the sensors have to be processed. Besides, the two trajectories have to be calculated based on the sensor data and some crucial dimensions of the combination. During the further procedure, the computed paths have to be augmented to the picture of the rear camera and presented to the operator. As those hard- and software entities might be distributed over the car-trailer-combination their configuration and topology might change at runtime. For this reason it makes sense to implement the system using the SODA framework and it's Service-based principles.

In chapter 5 the model-based development procedure to create such systems using the SODA framework has been presented. Furthermore, the case study illustrated in section 5.3 showed how this particular DDAS has been specified and developed. The evaluation of the SODA framework is carried out on an implementation of the system developed here. The only difference between the system developed in section 5.3 and the application executed on the demonstrator is the fact that the Services "VideoOutputService" and "TrajectoryTransformationService" are merged into a single Service. A detailed description of this implementation is given in section 9.2. Section 9.3 evaluates this system both on the Service and the system level while section 9.4 summarizes the analysis of the demonstrator.

Figure 9.2: The full scale demonstrator consisting of a Mercedes B-Class car and a two-axle trailer.

9.2 System description of the demonstrator

As stated before, the visual DDAS for cars with a two-axle trailer has been integrated into a full scale demonstrator. This demonstrator consists of a first generation Mercedes B-Class car (T 245) and a small two-axle trailer. This trailer has been originally build for narrow-track tractors. In order to match with the hitch of the car the open-end tow-bar has been removed and replaced by a ball-type tow hitch. Besides that both parts of the combination are in their original condition. Figure 9.2 shows the combination used for this evaluation.

The first addition that has been made to the demonstrator are the sensing entities needed to detect the current status of the combination. This includes bending angle sensors as well as a unit to detect the current steering angle of the car. The two bending angle sensors have been attached to the trailer. The mechanism needed measure these two values has been developed and integrated during a student research project at Heilbronn University (see [61]). Figure 9.3 shows the construction needed to detect Bending Angle 1. It contains a two-piece mechanical actuator that transfers the rotary motion between the car and the front axle of the trailer onto a rotary encoder. This encoder is a industrial off-the-shelf solution with a CANopen interface. Therefore it can be connected directly to the CAN network used for the assistance system. A similar construction has been created to measure the second bending angle between the front and the rear axle of the trailer. Is is attached to the pivot point of the trailer's front axle and makes use of a off-the-shelf rotary encoder of the same type which is also directly connected to the CAN. However, as those rotary encoders do not implement the SODA middleware but a CANopen interface instead the corresponding Services have been implemented in the form of Service Brokers (refer to section 4.2.1 for more details). This means that the Service "Bending Angle 1" does not actually control the hardware measuring the angle. Instead, it requests the angle from the CANopen device measuring it and offers it in the form of a SODA Service. The same principle is used to realize "Bending Angle 2". By using the Service Broker approach within the demonstrator this principle can be tested and evaluated. Both Services are implemented on a very small ECU utilizing a small 8-Bit ATmega88 microprocessor as the main CPU.
The last sensor to be added is the steering angle sensor. This has been done by making use of the car's in-build steering wheel angle sensor device. This small ECU is attached to the steering shaft and detects any rotary movement of this component. The generated values are then transmitted periodically to the Engine CAN of the B-Class periodically. As the other devices within the assistance system are not connected to this network

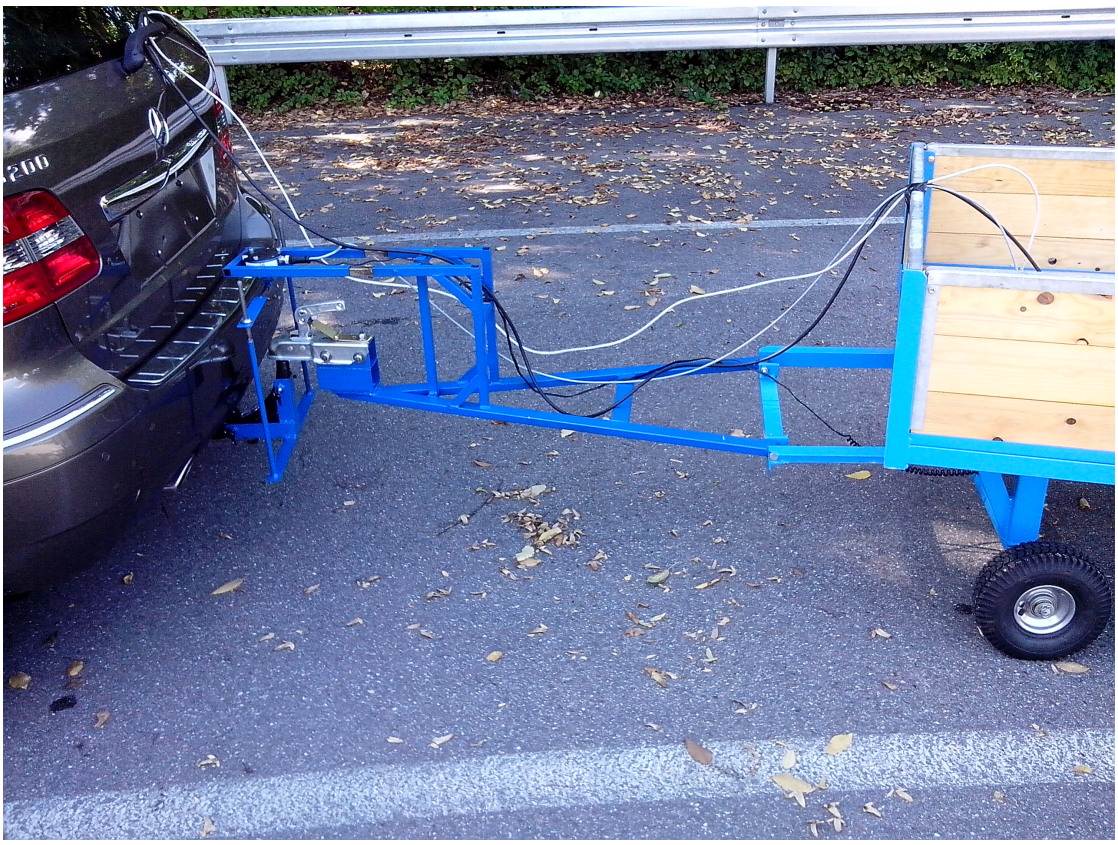


Figure 9.3: The mechanical sensor for the Bending Angle 1.

the Service Gateway approach has been used (refer to section 4.2.1 for more details). Therefore, a Raspberry PI board has been mounted into the car. This specific device owns two independent CAN interfaces. One of these is connected to the engine CAN and listens to the updates of the steering wheel angle sensor. The second one connects to the Assistance CAN and allows other Services to request the current steering angle. The software running on the Raspberry PI is also in charge of converting the steering wheel angle into the actual steering angle.

A second addition to the demonstrator combination are the components needed to allow a visual assistance for the operator of the vehicle. More specifically, a camera that records the area behind the trailer and a monitor within the dashboard have been added. For the rear view camera a simple USB webcam has been mounted to the body of the trailer as illustrated in Figure 9.4. The visual output is done using a 10" color monitor installed within the field of vision of the driver. Both hardware units are connected to computational entities that control them and implement the Services offered. In the case of the camera this entity is a Raspberry PI board connected via USB. The monitor is fed using a HDMI connection by a standard office laptop running Ubuntu OS. Raspberry PI and the Ubuntu laptop are both connected to the CAN network and offer or request Services using this communication technology. However, since the bandwith needed to transfer video streams exceeds the maximum bandwith offered by CAN the pictures can't be transmitted using this network. In the automotive domain this problem is often solved by splitting the communication traffic into control messages and data packets. The former ones are usually sent using a standard automotive network like CAN or LIN. The latter one are exchanged making use of a network system offering more throughput like for example Low Voltage Differential Signaling (LVDS). This ensures easy control of the data streams from the whole network while offering a bandwith that fulfills the requirements set up by video transport applications. In the demonstrator a similar

set-up has been chosen to solve this issue. Besides the CAN connection the Raspberry PI and the Ubuntu Laptop connect to each other using an Ethernet cable and UDP communication. While the Ethernet connection is used to actually transmit the video frames the control over the Services is done using the CAN bus. In other words, the video frames are transferred through the Ethernet connection using a legacy protocol while all Service communication complies to the SODA framework and is routed through CAN.

The demonstrator system, as it is illustrated in Figure 9.5 and 9.1, contains two additional ECUs. The first one bases on a Intel Atom processor. This unit is constructed using the Intel In-Vehicle Infotainment (IVI) reference design. It runs a light-weight variant of the popular Ubuntu operating system called Lubuntu. This entity serves as the platform for the two Services that compute the two trajectories which illustrate the future movement of the car-trailer-combination. The last hardware unit is again based on a small embedded board utilizing a ATmega88 CPU. It is attached to the trailer and offers Services to request the dimensions of this part of the combination.

The overall system architecture is shown in Figure 9.5. This figure presents the six different computing platforms that run the different Services distributed over the car-trailer-combination. The main communication backbone is built by the Assistance CAN. This network has been added to the demonstrator and connects all entities involved. The network colored in blue in Figure 9.5 is running with a bit rate of 500KBit/s which is a very common transfer rate in the automotive domain. As described before, the two computing units exchanging video data are making use of an additional Ethernet network. This switched Ethernet connection provides a maximum bandwith of 100 MBit/s as this is the maximum bandwith supported by the networking chip on the Raspberry PI. The third network pointed out in Figure 9.5 is the Engine CAN. This network, which is colored in green, is an integral part of the Mercedes B-Class. For safety reasons this bus is wiretapped in listen-only mode to ensure that no data on this network is corrupted. The Raspberry PI node (No. 4 in Figure 9.5) realizes the Service Gateway as described earlier by accessing both the Engine CAN and the Assistance CAN to transfer the steering wheel angle information into the assistance system.

Regarding the software aspects of the demonstrator the twelve Services developed in section 5.3 have been assigned to the different ECUs as shown in Table 9.1. The three hardware units attached to the trailer run a combined number of six Services. These are the RearViewService and CameraPositionService which are offering the rear view and the camera attributes respectively. Both of them run on the Raspberry PI module named ECU 2. The bending angles of the trailer are computed and encapsulated by the Services ReadBendingAngle1 and ReadBendingAngle2 which are both executed on the ATmega88-based ECU 5. Finally ECU 6, again a small ATmega88 hardware unit, hosts the remaining two Services. Both of them are enabling the remaining assistance system to request trailer constants. While the first one called TrailerWheelbaseService provides the distance between the two trailer axles the second one named TrailerDrawbarLengthService returns the length of the trailer's drawbar.

The other three ECUs are attached to the car and offer a total of six Services. The Ubuntu Laptop named ECU 1 hosts the VideoOutputService which generates the augmented video picture and displays it on the monitor attached to the dashboard. ECU 3, based on the Intel IVI reference architecture, executes two independent Services. CalcTrajA is in charge of computing Trajectory A which shows the future path of the vehicle involving all current sensor values. Furthermore, CalcTrajB is hosted by this ECU and determines a potential trajectory for a stable movement. Finally, the last ECU within the assistance system is the Raspberry PI board named ECU 4. It hosts a

Figure 9.4: Simple USB webcam to create a picture of the area behind the trailer.

total number of three Services. The first one called SteeringAngleService is the Gateway Service described earlier that retrieves the steering wheel angle from the Engine CAN and computes and offers the steering angle as a Service within the Assistance CAN. The second and the third one, named CarWheelbaseService and CarDistRearAxle-HitchService respectively, provide important mechanical dimensions of the car. While the former one offers the distance between the two axles of the car available to the rest of the system, the latter one describes the space between the rear axle and the hitch.

The workflow of the assistance system, once it is configured, equals to the one described in section 5.3: The two trajectory Services compute the future paths of the combination making use of the sensor values as well as the mechanical dimensions of the combination all provided in form of SODA Services. These two trajectories are then picked up by the VideoOutputService which augments them to the video picture taking into account the pose and characteristics of the camera which are also retrievable in form of a Service call. The resulting video pictures are then output to the monitor to support the driver when backing up the car-trailer-combination.

Additionally to the Services described here, several simulated Services have been placed into the system, too. These simulated Services are providing Service Interfaces of the same kind the real Services do. However, there is no functionality implemented behind these interfaces. The reason for introducing these entities to the demonstrator system is to generate rival Services that compete in the identifier assignment and Service selection processes with those Services that actually provide the promised functionality.

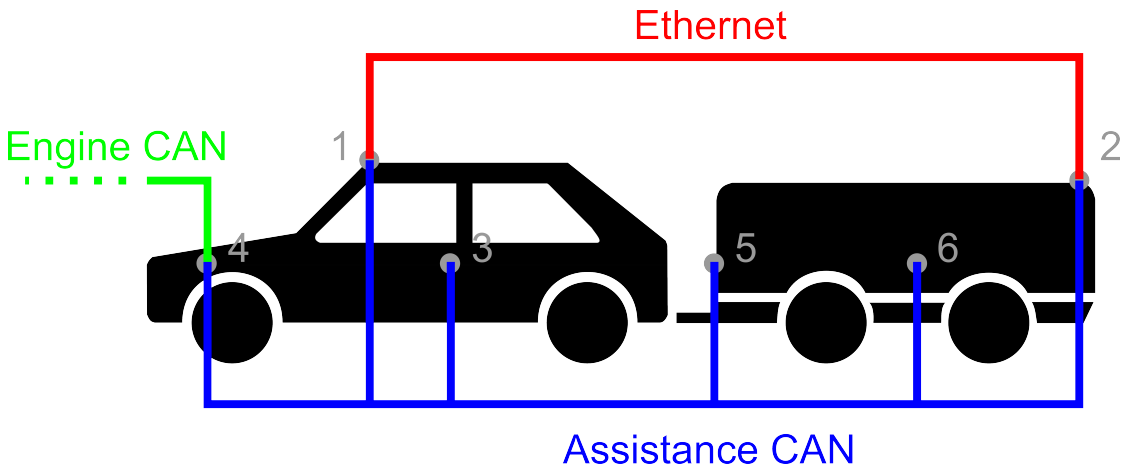


Figure 9.5: The system architecture of the demonstrator

No	Type	OS	CPU	Services Implemented
1	Laptop	Ubuntu (Linux)	Intel Core2Duo 2.53GHz	VideoOutputService
2	Raspberry PI	Raspbian (Linux)	ARM1176JZF-S 700MHz	RearViewService CameraPositionService
3	Intel IVI	Lubuntu (Linux)	Intel Atom E640T 1GHz	CalcTrajA CalcTrajB
4	Raspberry PI	Raspbian (Linux)	ARM1176JZF-S 700MHz	SteeringAngleService CarWheelbaseService CarDistRearAxleHitchService
5	ATmega88	none	ATmega88 18.432MHz	ReadBendingAngle1 ReadBendingAngle2
6	ATmega88	none	ATmega88 18.432MHz	TrailerWheelbaseService TrailerDrawbarLengthService

Table 9.1: Description of the computing units used within the demonstrator

9.3 Evaluation of the demonstrator

In this section the assistance system implemented on the demonstrator is evaluated. This incorporates an evaluation process on the Service level which analyzes the performance of each Service as a single entity. Additionally the overall SODA-based assistance system is examined. In doing so, the application's performance is analyzed in both phases, the re-configuration phase and during provisioning of the assistance to the driver.

9.3.1 Evaluation on the Service level

The evaluation of the individual Service implementations is based on the software metrics presented by Rossi and Tari in [113]. It has been refined and adapted to the characteristics of SODA. The evaluation of the individual Services is done through five key figures.

The first one of these key figures is the so called "Operation Interface Size" (OIS). It describes the aggregated size of the all parameters of a Service call and is measured in byte. In other words, it is the sum of the sizes of all parameters that are transferred within the data sections of the request and the response message of an individual Service. Equation (9.1) illustrates the computation of this quantity.

$$OIS = size_{request} + size_{response} \tag{9.1}$$

For example, if a Service request contains a single byte of data and the corresponding response implies four bytes of data, the OIS of this Service sums up to five bytes. This metric, which can be determined offline using the SoaML specification, is important since it directly influences the bandwith used by the Service in the event of a Service call.

A second metric used is the so called "Service Code Size" (SCS). It reflects the memory footprint of the Service implementation. This includes a detailed separation between the different parts of the executable binary namely the size of the SODA middleware, the communication drivers and the Service logic which contains the functionality. The size of the overall implementation as well as the individual parts are measured in byte. Equation (9.2) shows the composition of the SCS.

$$SCS = size_{middleware} + size_{drivers} + size_{logic} + size_{misc.} \tag{9.2}$$

The importance of the SCS is given through to the fact that SODA Services are meant to be hosted by very small hardware units that offer only very limited application memory of potentially only a few kilobytes.

In order to analyze the involvement of an individual Service into an application the "Collaborator Service Number" (CSN) is determined. This key figure gives the number of connections to other Services within an application and is measured in natural numbers. It is an important indicator of the relationships towards other entities and allows to draw conclusions on the potential consequences of a failure of this Service. Furthermore, a high number of collaborators is often also a sign for very frequent calls of this Service which may lead to a high utilization.

A very important metric when assessing the performance of a Service implementation is its response time to a Service call. This quantity, named "Operation Call Time" (OCT) in [113], is a crucial factor when creating responsive assistance systems that are able to inform the driver about changes within the system without delays noticeable by human beings. In the evaluation process carried out on the demonstrator the OCT metric has been refined to create more details and better insight into the internal workflow of the Service implementation. Therefor, not only the overall network response time has been

measured but also the execution times of the different modules taking part in such a Service call. In detail the analysis done includes the determination of the execution times of the driver module, the SODA middleware and the Service logic. The composition of the OCT metric is presented in Equation (9.3).

$$OCT = t_{middleware} + size_{drivers} + size_{logic} + size_{idle} \tag{9.3}$$

In applying this metric with such a high level of detail the performance of each module can be determined rather than just looking at the Service as a whole. The dimension used for the OCT is milliseconds.

Finally, the last figure to be looked at when analyzing an individual Service is the bandwith used over time. This metric, called "Network Utilization" (NU) in [113], depends on the OIS, the CSN as well as on the rate of Service calls within an application. Its computation is done according to Equation (9.4).

$$\begin{aligned} NU = & \#_{\text{Service calls per second}} * (size_{\text{Request data}} + \#_{\text{Request Messages}} * 67Bit \\ & + size_{\text{Response data}} + \#_{\text{Response Messages}} * 67Bit) \end{aligned} \tag{9.4}$$

The rate of Service calls per second is introduced by the parameter $\#_{\text{Cycles per second}}$. It is multiplied with the amount of bits sent through the communication channel for each of these calls. This amount is a sum of the raw data length of both the request and the response as well as the length of the overhead caused by each extended CAN frame sent. The resulting value is given in bit/s and shows directly the effects of an individual Service on the busload.

The five key figures described above have been applied to the exemplary DDAS implemented on the demonstrator combination. In the following the results of this data collection are presented and interpreted.

The first key figure to be looked at is the OIS. As described earlier it summarizes the sizes of the data sections of both the request and the response message of a Service call. Table 9.2 presents the values of the individual Service implementations for this parameter. As shown in this table the size of the data sections varies significantly. While the size of the Service requests is either zero or one byte, the size of the response can be anywhere between one and 70 bytes. This big differences propagate to the overall OIS. Considering the average values of these three parameters the average size of the data section within the Service requests is quite low with about 0.17 byte. The average values of the Service responses and the overall sum are 12.67 byte and 12.83 byte respectively. This shows that in the average case and every individual case the size of the data section of a CAN message is sufficient to carry the Service requests. This is not true for the Service response as this average value exceeds the maximum of eight byte fitting into a CAN message. However, the high average value is due to the fact that there are two Services with a really huge amount of data namely the CameraPositionService and the CalcTrajA Service. All other Services are far below these maxima and fit well into the CAN data section. In other words, only two of the combined 24 possible calls have to be split up into several CAN messages to be sent over the channel.

The second key figure to be looked at is the Service Code Size. The SCS numbers the memory footprint of a Service. Table 9.3 gives an overview over the results of these measurements. As the SODA Services running on the same hardware platform share the middleware and other resources like the driver layer or OS handles, the SCS is determined for the different ECUs rather than the Services themselves. The only exception here is the VideoOutputService. This Service is the only one to be executed

Service	Size Request [Byte]	Size Response [Byte]	Sum [Byte]
VideoOutputService	1	1	2
RearViewService	1	1	2
CameraPositionService	0	64	64
CalcTrajA	0	70	70
CalcTrajB	0	2	2
SteeringAngleService	0	2	2
CarWheelbaseService	0	2	2
CarDistRearAxleHitchService	0	2	2
ReadBendingAngle1	0	2	2
ReadBendingAngle2	0	2	2
TrailerWheelbaseService	0	2	2
TrailerDrawbarLengthService	0	2	2
Average	**0.17**	**12.67**	**12.83**

Table 9.2: Values of the Operation Interface Size for the individual Services

on ECU 1. The Service has an overall memory footprint of about 22 KByte. The biggest share of the storage needed is allocated by the Service logic that provides the actual functionality of the entity. The 15.9 KByte building this software module are responsible for transforming and augmenting the two trajectories and presenting the computed video to the driver. Another 941 Byte of memory are reserved for the driver layer of the SODA architecture. The software modules summarized under the term miscellaneous contain functionality like OS handles or wrappers to external components. These modules allocate a memory block of a combined size of 322 Bytes. The remaining 4876 Byte are reserved for the SODA middleware containing the SOA layer and the Communication Model.

When comparing this size to the overall amount of memory allocated by the Service implementation, the percentage of the space needed by the middleware is about 22%. As the SODA framework tailors each middleware implementation to the specific needs of the Services that run above it, the size of this module is not constant. When comparing the different ECUs regarding this figure, it can be seen that the VideoOutputService has actually the biggest middleware implementation of all entities listed here. This is due to the fact that it needs almost every single component of the middleware to its full extend. For example, the Segmentation component has to be integrated since it has to handle both Service calls that exceed the maximum length of a CAN message. Furthermore, it not only offers a Service but also uses several other ones which requires a full implementation of the Discovery Interface, QoS Interface and the Re-composition Manager. The differences in memory footprint are most obvious when comparing ECU 1 to ECU 6. Unlike the VideoOutputService implemented on ECU 1, ECU 6 does only host Services that do not have Requested Interfaces. These Services, namely the TrailerWheelbaseService and the TrailerDrawbarLengthService, do not require an implementation of the Re-configuration Manager since their only duty in the event of a Service selection is to respond to Discovery or QoS Requests rather than running extensive algorithms to select external functionality. Furthermore, the Discovery and the QoS Interface do not have to be fully implemented since they also only have to respond to external requests instead of initializing such requests themselves. Finally, none of the two Services has to be able to handle messages that extend the eight Bytes offered by CAN, which eliminates the need for this component as well. All these factors decrease the size of the SODA middleware implementation to as low as 606 Bytes. This low amount of memory needed makes the SODA middle-

ECU	$Size_{SODA}$ [Byte]	$Size_{Logic}$ [Byte]	$Size_{Drivers}$ [Byte]	$Size_{Misc}$ [Byte]	Sum [Byte]	pct_{SODA} [%]
ECU 1	4876	15903	941	322	22042	22.12
ECU 2	4852	17515	960	494	23821	20.37
ECU 3	4577	11440	966	301	17284	26.48
ECU 4	4648	2688	1216	469	9021	51.52
ECU 5	610	746	2400	1332	5088	11.99
ECU 6	606	456	2412	1320	4794	12.64
Average	3361.50	8124.67	1482.50	706.33	13675	24.58

Table 9.3: Memory footprint of the Services on the six ECUs

Service	CSN
VideoOutputService	4
RearViewService	1
CameraPositionService	1
CalcTrajA	8
CalcTrajB	3
SteeringAngleService	1
CarWheelbaseService	1
CarDistRearAxleHitchService	1
ReadBendingAngle1	1
ReadBendingAngle2	2
TrailerWheelbaseService	2
TrailerDrawbarLengthService	1
Average	2.17

Table 9.4: The number of collaborators of the different Services

ware suitable even for tiny microcontrollers like the ATmega88 used in the demonstrator.

When looking at the average numbers of the demonstrator application the middleware allocates about 3.4 KByte of memory. In comparison to the overall size of the different Service implementations this adds up to a percentage of under 25%. This seems to be reasonable when taking into account the benefits of a Service-oriented approach compared to a traditional, static implementation of such a Distributed Driver Assistance System.

The next metric considered is the "Collaborator Service Number" (CSN). As stated before, this parameter counts the connections to other Services. Table 9.4 gives an overview of the Services regarding this key figure. The Service to compute trajectory A is the one having the most connections to other entities. This is due to the fact, that this functionality needs a lot of information from quite a number of external functionality: besides the dimensions of the car and the trailer, it makes use of both bending angles as well as the steering angle to predict the future path of the vehicle. Furthermore, it is called by the VideoOutputService which adds another collaborator to the CSN. Many other Services do only have one collaborator that calls them. All of these are data sources that do not need any further information in order to offer their own functionality. Combining the CSN values of the individual entities the overall application averages at 2.17 collaborations per Service.

Although the exemplary assistance system implemented on the demonstrator is purely informing and does not directly interfere with any actuator like for example the brakes, responsiveness of the Services is highly important. This is due to the fact that studies have shown that high latency within systems using augmented technology techniques reduces the working performance of the people using it (see e.g. [110], [75]). The responsiveness of each individual Service is measured using Rossi's and Tari's "Operation Call Time" (OCT). The results of eleven of the twelve Services are illustrated in Table 9.5. The VideoOutputService is not part of this list, since it is never called by any other entity. Instead, it is the one actually initiating the assistance. For this reason the response time of the functionality can't be determined. Besides, two more Services were not analyzed in full detail. The respective Services are the entities computing the trajectories. The reason for this is technical in nature. Both of the Services are hosted by the Intel IVI embedded board. Unfortunately, it was not possible to access any I/O pin on the board which would have allowed to trigger precise external measurement equipment like for example an oscilloscope. Furthermore, the software-based measurement methods offered by the Linux OS were not accurate enough to generate meaningful results. In the case of these two software components only the overall response time could be determined which was done through the log files of the CAN network.

The remaining Services have been measured by using a digital storage oscilloscope connected to several I/O pins of the hosting hardware. The code of the Services was extended to switch these I/O pins to specified levels at different stages of the workflow. By recording and analyzing the levels of those I/O pins using the oscilloscope, the timing of the Service implementation could be evaluated in detail.

The overall response times vary between 33.15 ms and 0.71 ms as illustrated in Table 9.5. Especially the two Services that compute the trajectories catch the eye of the reader. These long response times can't be clarified completely, since the hardware doesn't allow further investigations. However, there are some correlations that can be made. Comparing the two of them, the only difference has to be in the execution of the Service logic, since both share the same middleware and driver components and run on the same system. The difference of about 5.52 ms can be easily explained by the higher effort to calculate Trajectory A compared to Trajectory B. Furthermore, when looking at the other Services running on Linux OS like for example the RearViewService or the SteeringAngleService, one can say that the average idle times within a Service call are a significant factor regarding the Service response time. At the other end of the scale, the TrailerWheelbaseService and the TrailerDrawbarLength have an OCT of only slightly more than 0.7 ms even though they run on the ECUs with the lowest computational performance. This time, the detailed decomposition of the measurement helps to understand how the ATmega88-powered board manages to provide such a superb response time behavior. The first factor is the very low execution time of the Service logic, due to the simple functionality carried out. Furthermore, as there is no compicated operating system and the SODA Services are the only functionality hosted by this ECU there is no measurable idle time which delays the Service response. The compact CAN driver used in the application only adds about an average of 0.15 ms to the OCT which keeps the response times low although the SODA middleware is a significant factor in this analysis needing an average of 0.24 ms execution time.

Looking at the percentage of the execution time of the SODA middleware compared to the overall response time, Table 9.5 names an average of 19.22%. This number is significant as this means that each time a functionality is called the time before an answer is sent out is almost 20% longer than it would be using a traditional approach. However, this overhead is worth it since the absolute values of the middleware's execution time

Service	t_{SODA} [ms]	t_{Logic} [ms]	$t_{Drivers}$ [ms]	t_{idle} [Byte]	$t_{Response}$ [ms]	pct_{SODA} [%]
RearViewService	0.74	2.97	0.70	2.48	6.89	10.74
CameraPositionService	0.76	1.34	0.71	2.60	5.41	14.05
SteeringAngleService	0.78	1.17	0.72	4.70	7.37	10.58
CalcTrajA	n.a.	n.a.	n.a.	n.a.	33.15	n.a.
CalcTrajB	n.a.	n.a.	n.a.	n.a.	27.63	n.a.
CarWheelbaseService	0.80	0.36	0.77	3.67	5.60	14.29
CarDistRearAxleHitchService	0.83	0.41	0.75	3.26	5.25	15.81
ReadBendingAngle1	1.11	4.09	0.40	0.00	5.60	19.82
ReadBendingAngle2	1.16	3.97	0.39	0.00	5.52	21.01
TrailerWheelbaseService	0.24	0.14	0.35	0.00	0.73	32.88
TrailerDrawbarLengthService	0.24	0.16	0.31	0.00	0.71	33.80
Average	0.74	1.62	0.57	1.86	9.44	19.22

Table 9.5: The response times of the Services

average at about 0.74 ms which is justifiable.

One last key figure to be considered is the "Network Utilization" (NU). It combines the message sizes of the Service requests and responses with the frequency of Service execution. Table 9.6 illustrates these numbers and the overall NU for each Service besides the VideoOutputService. This is again through to the fact that this Service is never actually called but acts self-directed to initiate the assistance functionality. For the remaining entities, Table 9.6 lists how often they are called (Calls/s), the size of the data transferred in the event of a Service request ($size_{\text{Request data}}$) as well as how many CAN messages are needed for a single request ($\#_{\text{Request Messages}}$). Furthermore, the data size of the respective response ($size_{\text{Response data}}$) as well as the number of CAN frames used to transport this data ($\#_{\text{Response Messages}}$) are shown here. Combining these values using the equation given in Equation (9.4), the overall NU value for each Service can be determined. As illustrated in the table, the frequency used to call the Services is either a single call per second for the static dimensions of the car-trailer-combination or 25 calls per second for those quantities that change over time. In doing so, a smooth output to the driver can be accomplished.

The values for the overall NU are varying from as little as 150 Bit/s to as much as about 30 KBit/s. This extreme value is caused by the Service that calculates Trajectory A. This is due to the amount of data needed to represent the trajectory combined with the high frequency of calls. The difference to the other Service calculating a trajectory is due to the fact that Trajectory B always follows a circular path. For this geometry it is sufficient to communicate the respective radius of the curve rather than a point-based description as used for Trajectory A. The overall bandwith occupied by the Services of the exemplary assistance system is about 51 KBit/s second. Since the application is running on a CAN network offering 500 KBit/s the system is using only about 10% of the provided bus capacity. This avoids timing problems through network overloads.

Service	Calls/s	$size_{\text{Request data}}$ [Bit]	$\#_{\text{Request Messages}}$	$size_{\text{Response data}}$ [Bit]	$\#_{\text{Response Messages}}$	NU [Bit/s]
VideoOutputService	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
RearViewService	25	8	1	8	1	3750
CameraPositionService	1	0	1	512	8	1115
CalcTrajA	25	0	1	560	9	30750
CalcTrajB	25	0	1	16	1	3750
SteeringAngleService	25	0	1	16	1	3750
CarWheelbaseService	1	0	1	16	1	150
CarDistRearAxleHitchS.	1	0	1	16	1	150
ReadBendingAngle1	25	0	1	16	1	3750
ReadBendingAngle2	25	0	1	16	1	3750
TrailerWheelbaseService	1	0	1	16	1	150
TrailerDrawbarLengthS.	1	0	1	16	1	150
Average	14.09	0.73	1.00	109.82	2.36	4655.91
					Sum	51215

Table 9.6: The Network Utilization of the Services

9.3.2 Evaluation on the System level

In this section, the focus swaps from analyzing the individual Service implementations towards evaluating the overall system. In a first step the CAN identifier assignment phase is examined. This phase, which starts directly after switching on the Services Instances, assigns a unique CAN identifier using the addressing scheme and algorithm described in section 7.5. The analysis is split up into two different scenarios. In the first scenario, a number of Service implementations is switched on at the same time. The actual number of implementations hereby ranges from only a single instance up to 25 instances. Twelve of them are represented by the actual Services running on the different ECUs of the demonstrator. The remaining 13 ones are simulated entities. They are implemented in the CAN testing environment "Vetor CANoe" using Vector's CAPL programming language to represent the assignment algorithm. All 25 Services are running a unique functionality which leads to the fact that no duplicates are attached to the system. The time out used within the assignment algorithm is set to 2 ms.

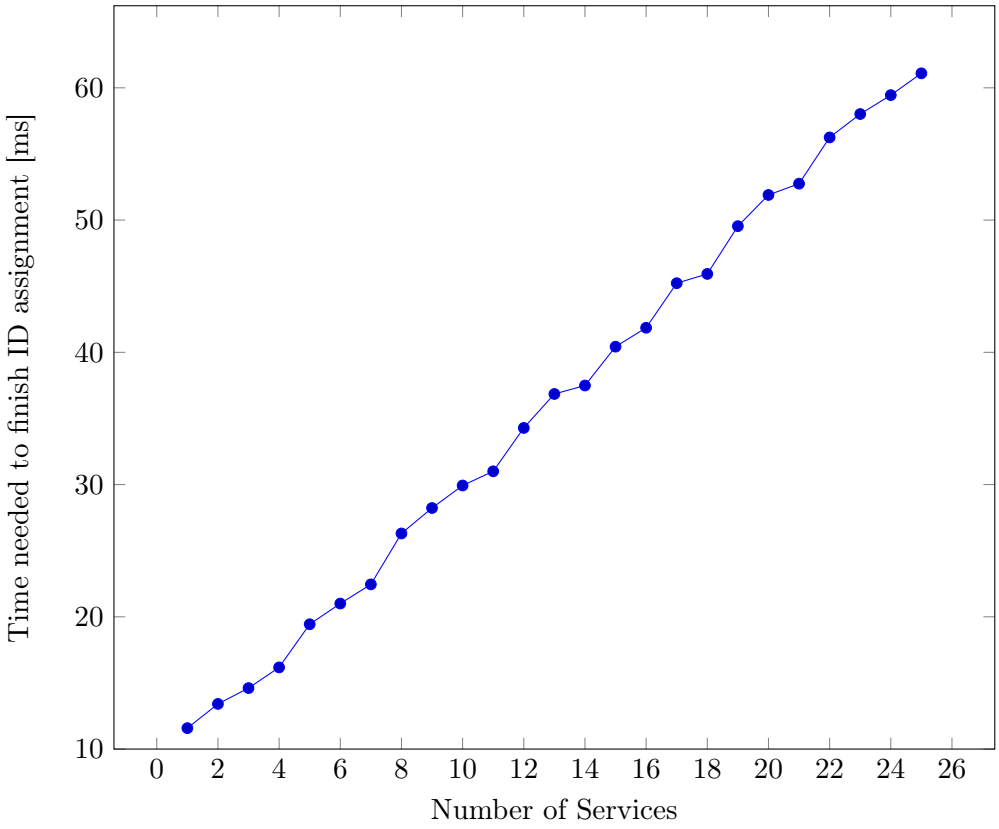


Figure 9.6: The time needed to finish ID assignment in the first scenario

Figure 9.6 shows the results of the evaluation. It illustrates the average time needed to assign an identifier to each entity, based on a series of ten test runs each. The figure reveals the fact that the assignment time rises almost linear with the number of Services present within the configuration. This discovery coincides with the relationship illustrated in Equation (7.3). While a single Service Instance in the network needs about 11.58 ms to generate a unique address, it takes about 61.10 ms to do the same thing with 25 entities. Right in the middle of this scale, a SODA-based DDAS consisting of twelve Service instances as the one implemented on the demonstrator needs slightly more than 34 ms in the average case to assign a unique CAN identifier to each unit.

The second scenario created to evaluate the identifier assignment phase analyzes the effect of duplicate Service implementations. These duplicates slow down the assignment

as they compete with each other and subsequently may block an address. This competition increases the number of requests needed until all entities involved are assigned with a unique CAN identifier. In this scenario all twelve implemented Service instances are started at the same time. Additionally, up to six simulated Service implementations are integrated. Each of these simulated entities matches one of the actual Services in its functionality. All entities, actually implemented as well as simulated ones, are using a time out of 2 ms in their assignment algorithm.

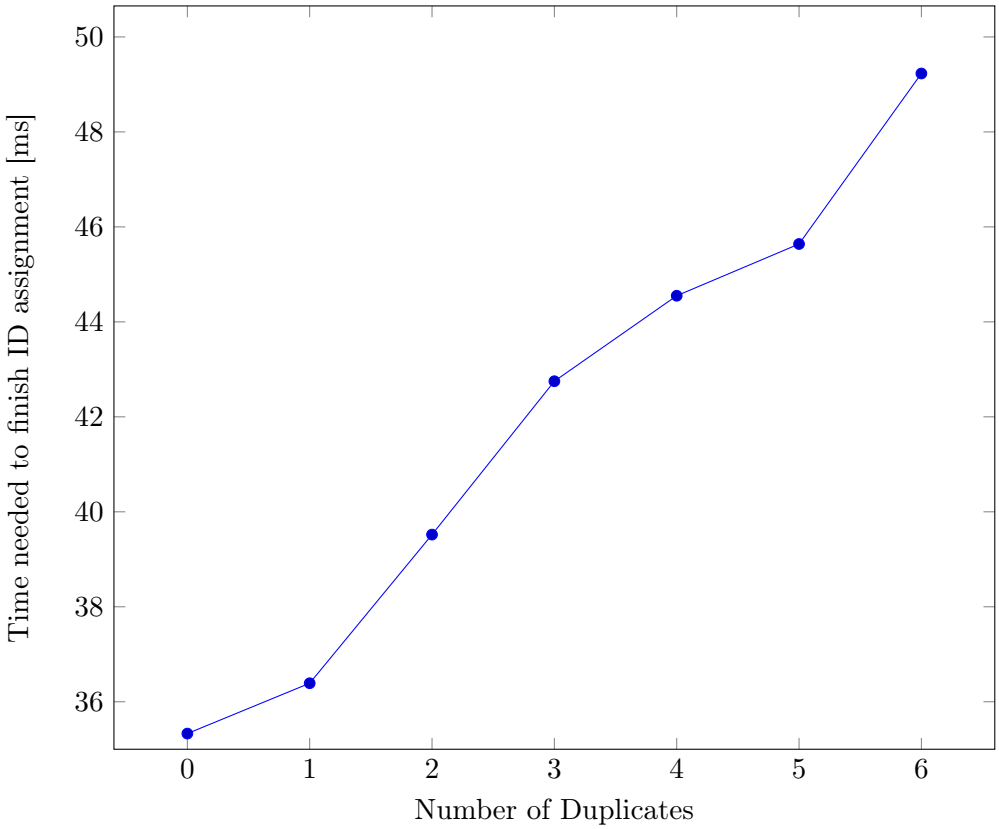


Figure 9.7: The time needed to finish ID assignment in the second scenario

Again, a measurement series of ten samples is carried out to create the average values presented in Figure 9.7. The time needed to assign an identifier to every entity when no duplicate is in the system averages at about 35.3 ms. This coincides approximately with the slightly more than 34 ms measured in the first scenario when evaluating a system counting twelve Services. At the other end of the scale, a system incorporating six additional duplicates, takes about 49.23 ms to finish the assignment phase. In between these two cases, Figure 9.7 demonstrates a linear increase of assignment time.

As a summary one can say that the evaluation of the assignment phase has been very successful. The system managed to assign CAN identifiers to all entities reliably and with good performance. Even in the extreme examples of both scenarios the algorithm managed to fulfill the assignment task quiet rapidly. Additionally, the property of linearity discovered in both scenarios helps to keep the effort and delay caused by the assignment phase manageable and in justifiable ranges.

In a second step, the overall re-configuration phase is evaluated. This includes the identifier assignment phase analyzed before as well as the Service Discovery and Selection procedure. It makes use of the dynamic programming-based algorithm described in section 6.5.2. In this algorithm time outs are introduced. The time outs describe the

Service	Time out [ms]
VideoOutputService	n.a.
RearViewService	10
CameraPositionService	10
CalcTrajA	30
CalcTrajB	30
SteeringAngleService	10
CarWheelbaseService	10
CarDistRearAxleHitchS.	10
ReadBendingAngle1	10
ReadBendingAngle2	10
TrailerWheelbaseService	10
TrailerDrawbarLengthS.	10

Table 9.7: The time outs of the individual Services

amount of time to be waited after a Service Discovery has been sent to the network. If they are too short, a Discovery response might be missed out. If they are too long the runtime perfomance of the algorithms decreases. For the Service Instances implemented on the demonstrator different time outs have been defined. Table 9.7 illustrates these values. As an example, the time out value for the RearViewService is quite short lasting only 10 ms. The low time out arises from the fact, that this Service does not need to discover other entities but can directly respond to the request. This is not true for other Services such as CalcTrajA. As this entity depends on several other Service implementations it needs more time to discover those and finally answer to the request. For this reason, CalcTrajA posses a relatively high time out value of 30ms. Again, the VideoOutputService can't be evaluated within this score, since it is never discovered by any other Service.

These time out values are deployed to the demonstrator's Services. In order to evaluate the system in real-world events, the scenarios developed in section 6.2 are picked up again:

1. Ignition on
2. Connecting a trailer at runtime
3. Disconnecting a trailer at runtime
4. Change of the type of assistance at runtime
5. Change of the quality parameters at runtime
6. Failure of a Service Instance at runtime

While this listing represents a full list of possible events, not all of them make sense regarding the implementation of the demonstrator. For example, scenario 3 can be merged in scenario 6 as the disconnection of a trailer leads to the failure of several Service Instances. Besides that, scenario 4 is not considered as only one type of assistance is implemented on the demonstrator. Lastly, as the demonstrator uses fixed values for the QoS vectors, scenario 5 is also not considered within this evaluation. On the other hand, it is quite interesting to differentiate within the scenario of a Service failure between the failure of a duplicate implementation and a Service Instance that is unique within the system. For these reasons, the scenarios used in this evaluation are as follows:

1. Ignition on
2. Connecting a trailer at runtime
3. Failure of a duplicate Service Instance at runtime
4. Failure of a unique Service Instance at runtime

For all these scenarios the time out of the identifier assignment algorithm is set to 2 ms. The time out values for the Service Discovery are the ones presented in Table 9.7.

The first scenario represents starting the car-trailer-combination by turning on the ignition. The configuration of the system under evaluation is as follows: all twelve Service implementations developed are used. Additionally, four simulated instances are added. Each of this four instances matches one of the actual Service implementations but possesses a slightly worse QoS value. Through to this competition, the Discovery and Selection algorithm as presented in section 6.5.2, is evaluated. The CAN network used to transfer the Service communication is set to a bit rate of 500 KBit/s.

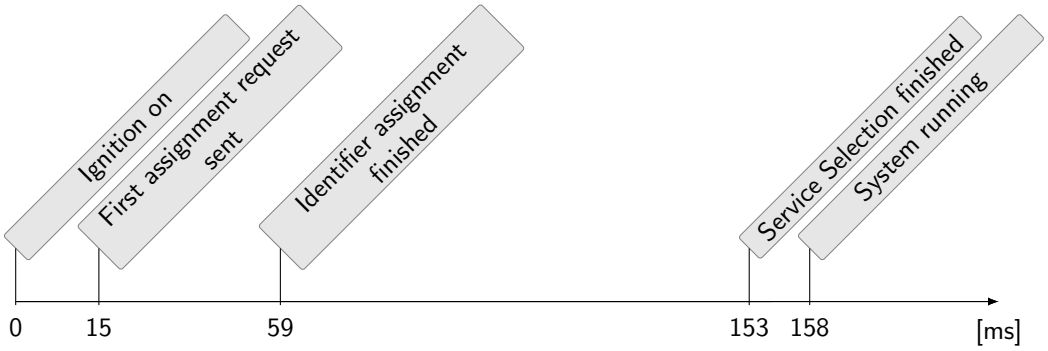


Figure 9.8: Chronology of the events in the ignition on scenario

Figure 9.8 shows the averaged results of a test series of 10 measurements. On the ignition on event the small 8-Bit ECUs are powered up, initialize themselves and start the hosted Service Instances. The Linux-based hardware units are already booted. The ignition on event makes them start loading and executing their Service implementations. As illustrated in Figure 9.8, it takes an average of 15.1 ms until the first assignment request message is sent to the CAN bus. This first message starts the identifier assignment phase which finishes after an average of 59.4 ms. As all Service Instances do now own a unique CAN identifier the system moves on to the Discovery and Selection phase. Within this second phase the Service Instances discover and select their partners using the algorithm described in section 6.5.2. This second phase ends after an average of 153.0 ms with having determined the configuration with the best end-to-end QoS currently available. After an average of 158.2 ms the assistance systems starts working by exchanging the first Service call.

Within the ignition on scenario the two algorithms assigning CAN identifiers and discovering and selecting Service Instances worked reliable. The overall configuration time of an average of 153 ms is reasonable as it is not very likely that the driver wants to use the system during that time period after starting the car. The duration of the identifier assignment matches the experiments done earlier. As illustrated in Figure 9.7, the assignment algorithm needed an average of about 44.5 ms to finish its run when four duplicates were attached. This range has been confirmed in the ignition on scenario as it took about 44.3 ms to solve the same problem.

The second scenario to be looked at when evaluating the re-configuration characteristics of the demonstrator is the event of connecting the trailer at runtime. In this scenario the ignition of the car has been turned on a sufficient period of time before the trailer is attached to ensure the car's Service Instances are completely initialized. The chronology of the of events happening after the trailer is attached is illustrated in the time line given in Figure 9.9.

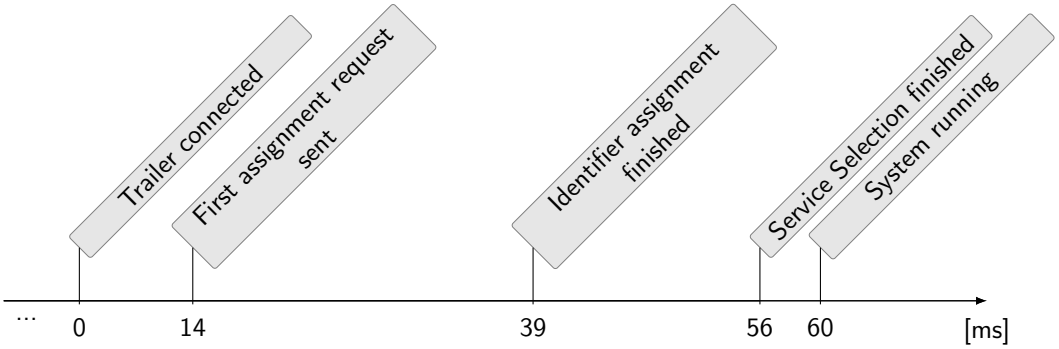


Figure 9.9: Chronology of the events when connecting a trailer at runtime

Again, all numbers given here are average values calculated using a test series of ten sample runs. After about 14.8 ms the three ECUs attached to the trailer are initialized and the first message to request an identifier has been sent. Those three ECUs host a total number of six Service Instances. Three of these Service Instances are not unique within the system but are matched by one duplicate each. These duplicates have been started together with the car. Therefore they are already initialized and got a CAN identifier assigned to their functionality. Through to this fact, the three Service Instances now being added have to compete with these duplicates in both the identifier assignment and the Service Selection phase. These circumstances cause a relatively long assignment phase that finishes after an average of 39.1 ms after the trailer got connected to the car. The second phase, discovering and selecting the configuration that offers the best QoS parameter, is rather fast. After an average of 55.9 ms this configuration is found and established. This relatively short duration of only 16.8 ms arises from the fact that a big share of the problem has been already solved right after the car has been powered up. This leads to the fact that in the average case the newly configured DDAS starts operating 60.2 ms after the trailer has been connected to the car. This means that there is no delay noticeable by the operator of the car-trailer-combination when attaching the trailer to the vehicle.

The third scenario evaluates the behavior of the SODA-based DDAS in the event of the failure of a Service Instance. More precisely, it examines the failure of an entity that is not unique but has a duplicate offering the same functionality. The configuration used for this scenario is as follows: The system has been initialized and the DDAS is active. It uses the twelve Service Instances implemented. The four duplicates are in an inactive state since they have not been selected by the application. The failure scenario is created by switching off one of the Service Instances currently used. In the following the SODA system will try to recover the application by selecting a replacement for the entity not responding any more. This scenario has been ran through with two different Service Instances to be disconnected.

In the first ten test runs the Service CameraPositionService has been disconnected from the system. Figure 9.10 illustrates the process of re-configuration for this case. After the failure of CameraPositionService the system needs an average of 17.2 ms until it detects that circumstance and starts to re-configure. This re-configuration, which

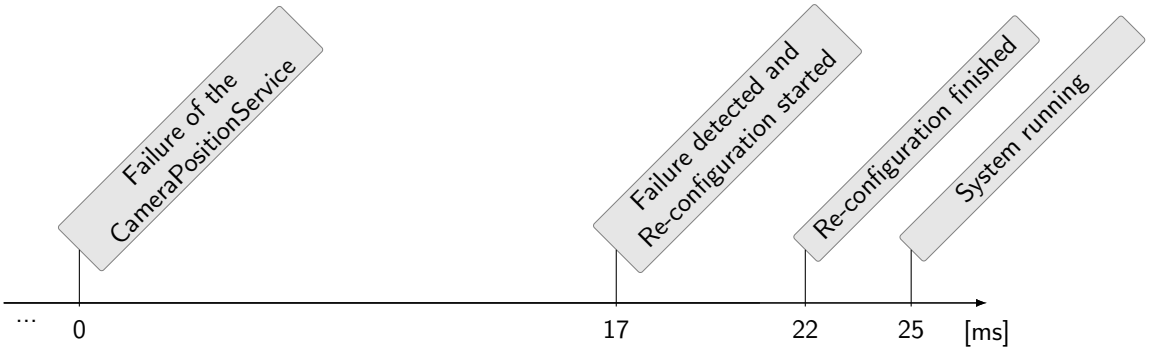


Figure 9.10: Chronology of the events after the failure of the CameraPositionService

includes the Discovery and Selection of the duplicate available for CameraPositionService, last until 21.8 ms after the initial failure event. About 25.0 ms after switching off CameraPositionService the system is back in active mode and executes the DDAS again. This recovery mechanism is fast enough to stay well under the human perception threshold. In other words, the operator of the car-trailer-combination would not even notice that something has happened to the system that required a re-configuration of the Services.

One of the reasons for the good performance in the case of the failure of the CameraPositionService is the fact that this Service Instance has no Requested Interfaces. If it fails, the only thing to do is to find and select a replacement. A more complex scenario is given when a Service Instance fails that actually has Requested Interfaces. In such a case, identifying and activating a duplicate is only the first step. Additionally, this duplicate Service Instance must discover and select the requested functionality specified through the Requested Interfaces as well.

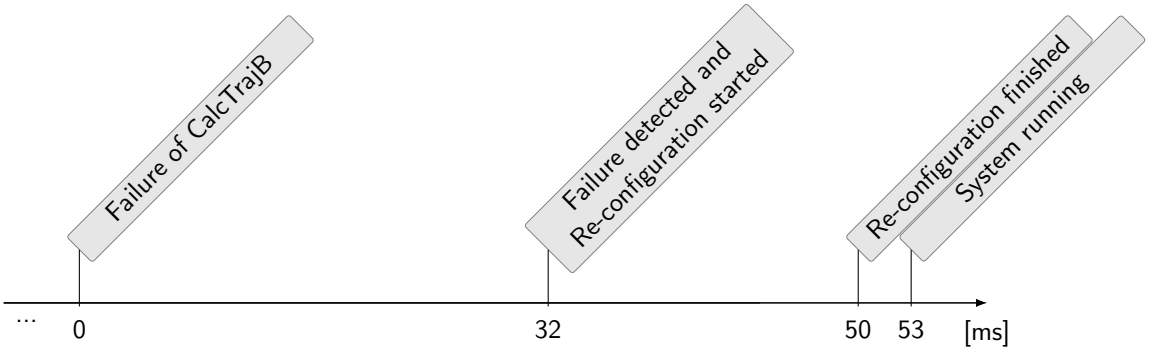


Figure 9.11: Chronology of the events after the failure of CalcTrajB

This more complicated case is evaluated by switching off the Service CalcTrajB. As illustrated in Figure 9.11, the system needs about 32.3 ms until it detects the failure and starts to re-configure the application. Afterwards, it executes the re-configuration which is finished after 49.9 ms. The system is ready and active again after 53.1 ms. The re-configuration time, which is about twice as long as in the case of the failure of the CameraPositionService, is still in a very good range. The driver would not notice anything but maybe a short unsteadiness of the augmented video picture presented.

The last scenario within the analysis of system re-configuration addresses the case when a unique functionality fails. In this event the application has to determine this failure and its inability to provide the assistance system any longer. After this diagnosis has been made, the driver needs to be informed about it to avoid potential misleading

Sensor Value	Average age [ms]
Rear View Picture	49.23
Steering Angle	64.70
Bending Angle 1	63.19
Bending Angle 2	58.52

Table 9.8: The average age of the sensor data

information.

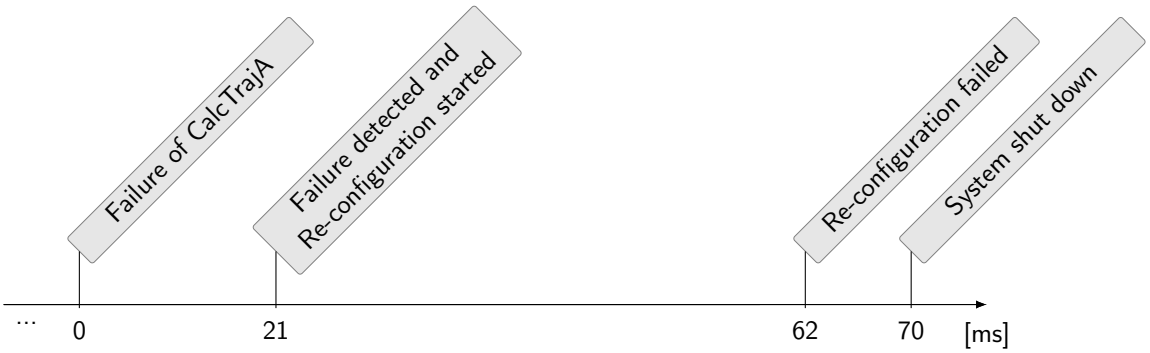


Figure 9.12: Chronology of the events after the failure of CalcTrajA

The time line given in Figure 9.12 shows the results of the system behavior when the Service CalcTrajA is shut down. Again, the numbers are average values computed from a test series containing ten runs. The average time to detect this failure is 21.3 ms. After this discovery the system starts trying to discover alternative entities offering the same functionality. As there is no such duplicate available it recognizes its situation about 62.4 ms after the Service Instance failed. The notice to the operator of the vehicle about this situation is given after approximately 70.2 ms. This relatively low time between the appearance of the failure and the notification of the driver ensures that no out-dated or wrong information is presented by the assistance system. Please note that after the notification the system keeps on sending out Discovery messages in case the failed Service Instance recovers or a new entity offering the missing functionality joins the network.

The third and last step of the system evaluation is the analysis of the execution of the application under normal runtime conditions. Here, the main figure to look at is the timeliness of the overall application. In this sense, a evaluation of the average age of sensor data when being presented to the operator of the vehicle has been carried out. In order to do so, some Service implementations have been slightly altered. This includes those Service Instances that act as sensors. They have been modified to send a specific flag instead of the actual sensor value. This flag can now be traced through the application workflow by observing the Service messages on the CAN network. The VideoOutputService has been changed to send a message onto the CAN bus which contains the flag in the very moment it visualizes the tagged data on the screen. The message flow on the network is observed using Vector CANoe.

Table 9.8 illustrates the age of the sensor values that are requested at frequent intervals. The lowest number is the age of the rear view picture as this value is transmitted directly from the source to the sink. All other sensor data is processed by another instance namely CalcTrajA or CalTrajB. This leads to values of up to 64.70 ms. However,

the system always felt responsive when using it in the field.

9.4 Summary

This chapter described the evaluation of the SODA framework using a real world demonstrator. This demonstrator is built up by a Mercedes B-Class car and a two-axle trailer. This combination was enhanced with additional equipment such as a camera, a monitor and several sensors. The exemplary assistance system created using SODA consisted of 12 Services implemented according to the development described in 5.3. The hardware units hosting them were either Linux-based or small 8-Bit ECUs which were connected through CAN and Ethernet.

The evaluation procedure was split up into two parts: the evaluation on the Service level and on the System level. On the Service level five key figures were examined such as Network Utilization, Operation Interface Size or the number of collaborators of each Abstract Service. Two additional measurements, which were of high interest in order to analyze the effects of adding the SODA middleware were the Operation Call Time and the memory footprint. Regarding the former key figure the time needed to execute the middleware represents about 19.22% of the overall OCT. Looking at the average absolute execution times of the implementations the middleware needed only about 0.74 ms to fulfill all its tasks. Analyzing the memory footprint, the middleware occupies an average of about 22% of the overall implementation size. When evaluating the absolute numbers the maximum size of the SODA middleware is 3.4 KByte. On the other hand, smaller Service Instances with less requirements feature SODA middleware implementations of only 606 Byte of memory.

To summarize the evaluation of the individual Service Instances one can say that the overhead created by the SODA middleware is quite low. The additional amount of memory or execution time is well outperformed by the benefits this framework introduces such as runtime adaptability. In this sense the evaluation of the SODA framework regarding the Service level within the example application was highly successful.

On the System level, the evaluation consisted of three different parts. These were the analysis of the CAN identifier assignment phase, the overall re-configuration of the system as well as the runtime behavior. In order to examine the CAN identifier assignment phase two scenarios have been developed. The first scenario evaluated the execution times of this phase for different numbers of Service Instances. All of these Service Instances were unique which leads to the fact that they did not compete for the same identifiers. The evaluation revealed the linear correlation between the number of Service Instances requesting an identifier and the time needed to complete this assignment phase. For the configuration established on the demonstrator containing twelve Service implementations the assignment phase took an average of about 34 ms.

The second scenario within the assignment phase evaluation examined the effects of duplicate Service Instances within a system. The test scenario introduced duplicates varying from zero to six additional implementations. Again, a linear correlation between the number of entities competing and the time needed to assign a unique identifier to each Service Instance has been discovered. This linear correlation in both identifier assignment scenarios allows the conclusion that the amount of time needed to finish the identifier assignment phase is justifiable even for higher numbers of Service Instances.

In a second analysis on the system level the whole re-configuration phase including the identifier assignment and the Service Discovery and Selection were evaluated. Hereby,

the algorithm described in section 6.5.2 was used to configure the assistance system online. The re-configuration was examined through introducing four different real-world scenarios. The first one of these scenarios was the event of switching on the ignition and thereby all Service Instances at the same time. The demonstrator successfully finished the configuration of the system within an average of 158.2 ms. In the second scenario the trailer was connected to the car at runtime. Here, the SODA middleware needed an average of 60.2 ms to start up and integrate the newly added Service Instances and create the overall application. The remaining two scenarios examined the error handling capabilities of the demonstrator. In scenario three, a Service Instance that is matched by a duplicate within the system was switched off. The SODA middleware was challenged to discover this failure and re-configure the application using the duplicate Service Instance. Again, this task was fulfilled with good performance needing only about 25 ms to recover the assistance application. In the final scenario a Service Instance unique within the system was switched off. The SODA middleware had to detect this issue and warn the driver about the fact that it is no longer possible to offer the requested assistance. This warning was presented an average of 53.1 ms after the Service Instance was shut down.

All these values showed that SODA is capable of re-configure the system reliable and within very short time intervals. Please be reminded that all the numbers presented are strongly depending on the time outs configured within the algorithms. The time outs chosen for the demonstrator were somehow conservative and aimed first and foremost on the reliability of the system. Shortening those time outs would even speed up the algorithms but might lead to an unreliable behavior under specific circumstances. In an actual product development these time outs could be examined and adjusted in order to find a good balance between the reliability and timeliness of the application.

The last part of the analysis of the System level observes the phase when the system is fully initialized and running. Here, a important figure is the age of the data when being presented to the driver. This evaluation revealed, that the average age of the sensor data might be up to 64.70 ms. This relatively high number originates mainly from the long execution times of the Service logic implementation rather than being caused by the SODA framework. Besides, the system always felt responsive.

Summarizing the evaluations done on the demonstrator one can say that SODA proved to be functional and reliable, convinces with a very good performance and well tailored middleware implementations that keep the overhead very low. These impressive results demonstrate that Service-orientation does not only offer the capabilities needed to create runtime adaptive DDAS but also shows that it is possible to apply these principles within the automotive domain.

10 Summary

'There is no real ending. It's just the place where you stop the story.'

Frank Herbert[1]

This thesis described the research that led to an adaptive software and system architecture for DAS for truck and trailer combinations. The architectural style used for the SODA middleware is Service-orientation. Service-oriented Architectures are mostly used on desktop computer systems connected using IP-based networks. Since this methodology became very popular in recent years, many frameworks have been published by industry or academia. Chapter 2 discussed 23 of these approaches showing that none of them fulfills all three central requirements of the given usage scenario like automatic re-configuration at runtime, economical resource usage and distributed management algorithms. Hence, the basic principles of SOA were used to create a tailored, unique framework called SODA.

The foundations of the SODA framework are built by its reference model. This model is constituted by the definition a set of terms and their relationships. This ensures clear definitions of the principles and methods used. Furthermore, it establishes a clear set of Service-oriented concepts used based on the goals set up by the truck and trailer scenario. These concepts are implemented by a set of components. The aggregation of these components builds an architectural blueprint, the so called reference architecture of a SODA Service. Additionally, a Quality of Service parameter has been created that reflects the functional characteristics of each Service Instance. This unique QoS parameter introduced in SODA allows to adapt its characteristics not only to the functionality but also to the application created using this functionality. Besides, it is processed into a one-dimensional parameter which highly simplifies the Selection process. This resource-friendly QoS parameter complements the SODA reference model.

One disadvantage of a reference architecture is the fact that some components might not be used in each Service instantiation as the concepts they implement might not be requested. For example, a Source Service that does not need any other functionality to fulfill its task does not necessarily have to implement all parts of the Discovery or re-configuration algorithms. This is through to the fact, that this kind of Service is purely passive in any kind of re-composition scenario. Hence, from a resource management point of view, it is reasonable to create tailored Service implementations that do only include those components actually needed. In order to support this approach, SODAdev has been created. SODAdev is a model-based development approach that allows to create SODA-based Driver Assistance Systems from various starting points. It's well-structured, phase-oriented procedure guides the development team through the design process and supports them by providing a tool environment. The outcome of this procedure is a SoaML model that can then be used for further processing such as model-checking, code-generation or to create a Communication Model for the Service Instances involved.

[1] 'Willis E. McNelly: Interview with Beverly and Frank Herbert, see [94]

As a matter of fact, the integration of Service communication into today's most common automotive network systems is another important step in introducing SOA into the automotive domain. In order to keep the implementation tailored, another procedure called SOAcom has been created. Here, the requirements regarding the communication stack of the driver assistance application are derived using the SoaML model created by SODAdev. Furthermore, the characteristics of the automotive networks planned to be used are determined by a custom-built questionnaire. Both information sets are then combined using flowchart diagrams to detect the shortcomings of the networks in the specific usage scenario. Using a list of tasks that describe the actions to be undertaken, the design team is provided with a clear specification of the Communication Model under development.

One crucial part of the introduction of Service-orientation into automotive embedded is the re-composition algorithm that creates optimized Service Execution Graphs at runtime using the Service Instances currently available. Despite the fact that there are numerous suggestions on re-composition algorithms in the literature, none of them fulfilled the specific requirements given in truck and trailer assistance systems. These requirements include the ability to re-compose the system at runtime, the creation of the optimal solution, the usage of distributed algorithms and the limitation of resources used. Especially the last requirement is indispensable when trying to make the framework runable on very small embedded devices. The solution to this problem was the design of a novel re-composition algorithm based on dynamic programming. It reduced the complexity of the selection procedure significantly by dividing the overall problem into small subproblems that are solved in a distributed manner. Furthermore, the principle of using hierarchical determination of optimal subsolutions guaranties the achievement of end-to-end optimality without additional computations.

Another important achievement presented in this thesis is the integration of the SODA framework into AUTOSAR. Besides the integration of SODAdev into CPSSD this is another crucial point were the framework proved to be compatible to today's automotive software systems. In chapter 8 three different approaches on the integration of the SODA components into the AUTOSAR architecture were presented. All of them offered the dynamics in the software stacks needed and proved to be rather resource friendly. However, they can be differentiated according to the level of integration offered. While the approach using the Complex Drivers bypasses the complete basic software stack, the one replacing the XCP module uses most of the layers of AUTOSAR. In the third approach that enhances the transport protocol layer, all AUTOSAR layers, interfaces and handles are used the way intended. Although all three approaches are fully functional, they are mend to be suggestions on possible extensions to the AUTOSAR standard.

The final chapter of this thesis described the evaluation of a prototype implementation of a truck and trailer assistance system using the SODA framework. The demonstrator used is a Mercedes B-Class car connected to a two-axle trailer. The twelve Services building the application are distributed over both parts of the vehicle. In order to prove the framework on different hard- and software platforms the devices used ranged from an Ubuntu Laptop, over Intel IVI boards to very small ECUs powered by an 8-Bit Atmel CPU. The system was analyzed both on Service and on application level, during re-configuration and at normal operation. In all cases the application as well as the underlying framework were stable, reliable and performed very well. Especially regarding the memory footprint the system proved to be very resource friendly.

The author Frank Herbert once said: "There is no real ending. It's just the place

where you stop the story." This is also true for this thesis and the research it describes. Although it features many novel approaches and techniques, there is still a number of open issues regarding the usage of Service-oriented Computing in automotive embedded systems. One of theses issues is certainly the need for security. As this approach opens up the car's electronic system, mechanisms have to be defined that ensure that this openness is not used for harmful attacks. Another open point is the definition and implementation of quality of service mechanisms on the network layer. As SODA might be used for safety-related, real-time driver assistance systems, adding guarantees regarding the reliability and timeliness of message transport would be very valuable. Furthermore, this research showed that the usage of SOA is not only capable of managing truck and trailer systems but should be beneficial in other Distributed Driver Assistance Systems, too. Especially applications basing on car-to-x communication may avail oneself by encapsulating functionality into Services and using these entities to build powerful ad-hoc applications. Some other potential field of usage is the development of safety-critical systems. Many of them derive their safety levels by providing redundancy for critical functional blocks. The Service-oriented concepts that enable runtime adaptive DDAS could also be used to manage the state transitions that have to be executed when switching to a redundant component. Hopefully, future research will prove this assumption to be true.

Abbreviations

ABS	Anti-lock Braking System
ACC	Adaptive Cruise Control
API	Application Programming Interface
ARP	Address Resolution Protocol
AUTOSAR	AUTomotive Open System ARchitecture
AVB	Audio Video Bridging
CSMA/CR	Carrier Sense Multiple Access/Collision Resolution
BAWS	Bending Angle Warning System
BPEL	Business Process Execution Language
BPMN	Business Process Modeling Language
BSW	AUTOSAR Basic Software
CAN	Controller Area Network
CASE	Computer-Aided Software Engineering
CORBA	Common Object Request Broker Architecture
CPSSD	Core Process for System and Software Development
CPU	Central Processing Unit
CSN	Collaborator Service Number
CSV	Comma-separated values
DAS	Driver Assistance System
DDAS	Distributed Driver Assistance System
DES	Distributed Embedded Systems
DHCP	Dynamic Host Configuration Protocol
DPWS	Devices Profile for Web Services
DSL	Domain-specific Language
ECU	Electronic Control Unit
ESP	Electronic Stability Program
HAL	Hardware Abstraction Layer
HDMI	High Definition Multimedia Interface
HMI	Human Machine Interface
HTTP	Hypertext Transfer Protocol
IID	Instance Identifier
IP	Internet Protocol
ISO	International Organization for Standardization
ISO-TP	International Organization for Standardization Transport Protocol
IVI	In-Vehicle Infotainment
I/O	Input and Output
LDWS	Lane Departure Warning System
LIN	Local Interconnect Network
LSA	Logical System Architecture
LVDS	Low Voltage Differential Signaling
MDA	Model-driven Architecture
MDSD	Model-driven Software Development
MOST	Media Oriented Systems Transport
NU	Network Utilization
OASIS	Organization for the Advancement of Structured Information Standards
OCT	Operation Call Time
OEM	Original Equipment Manufacturer

OIS	Operation Interface Size
OMG	Object Management Group
OS	Operating System
OSGi	Open Services Gateway initiative
OSI Model	Open Systems Interconnection Model
PDA	Personal Digital Assistant
PDU	AUTOSAR Packet Data Unit
PduR	AUTOSAR PDU Router
POF	Plastic Optical Fibers
QoS	Quality of Service
RAM	Random-Access Memory
ROM	Read-only Memory
RSA	Rational Software Architect
RTE	Runtime Environment
SAW	Simple Additive Weighting
SCA	Service Class Address
SCS	Service Code Size
SLA	Service Level Agreement
SOA	Service-oriented Architecture
SOAcom	Service-oriented Architecture Communication Development Process Model
SoaML	Service-oriented Modeling Language
SOAP	Simple Object Access Protocol
SOC	Service-oriented Computing
SODA	Service-oriented Driver Assistance
SODAdev	Service-oriented Driver Assistance Development Procedure
SOMA	Service-oriented Modeling and Architecture
SOME/IP	Scalable service-Oriented MiddlewarE over IP
SW-C	AUTOSAR software component
TCP	Transmission Control Protocol
TSA	Technical System Architecture
TTCAN	Time Triggered Controller Area Network
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
UDS	Unified Diagnostic Services
UML	Unified Modeling Language
USB	Universal Serial Bus
VDE	German Association for Electrical, Electronic and Information Technologies
VFB	Virtual Function Bus
WSDL	Web Services Description Language
W3C	World Wide Web Consortium
XCP	Universal Measurement and Calibration Protocol
XML	Extensible Markup Language

List of Figures

List of Tables

Bibliography

[1] AUTOSAR Administration. *AUTOSAR: Classification of interfaces*. 2013. URL: http://www.autosar.org/index.php?p=1\&up=2\&uup=3\&uuup=4\&uuuup=0\&uuuuup=0.

[2] AUTOSAR Administration. *AUTOSAR Specification v4.1 Rev2: Specification of RTE*. 2013.

[3] AUTOSAR Administration. *Layered Software Architecture*. 2013. URL: http://www.autosar.org/download/R4.0/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf.

[4] Pekka Aho, Janne Merilinna, and Eila Ovaska. "Model-Driven Open Source Software Development - The Open Models Approach". In: *2009 Fourth International Conference on Software Engineering Advances* (Sept. 2009), pp. 185–190. DOI: 10.1109/ICSEA.2009.37. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5298439.

[5] Marco Aiello et al. "Optimal QoS-Aware Web Service Composition". In: *2009 IEEE Conference on Commerce and Enterprise Computing*. Ieee, July 2009, pp. 491–494. ISBN: 978-0-7695-3755-9. DOI: 10.1109/CEC.2009.63. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5210754.

[6] Jim Amsden. *Modeling SOA : Part 4 . Service composition*. 2007. URL: http://www.ibm.com/developerworks/rational/library/07/1023_amsden/.

[7] D. Ardagna and B. Pernici. "Adaptive Service Composition in Flexible Processes". In: *IEEE Transactions on Software Engineering* 33.6 (June 2007), pp. 369–384. ISSN: 0098-5589. DOI: 10.1109/TSE.2007.1011. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4181707.

[8] Danilo Ardagna and Barbara Pernici. "Global and local qos guarantee in web service selection". In: *Business Process Management Workshops*. 2006, pp. 32–46. DOI: 10.1007/11678564_4. URL: http://link.springer.com/chapter/10.1007/11678564_4.

[9] A. Arsanjani et al. "SOMA: A method for developing service-oriented solutions". In: *IBM Systems Journal* 47.3 (2008), pp. 377–396. ISSN: 0018-8670. DOI: 10.1147/sj.473.0377. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5386496http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5386496.

[10] JM Astesana, L Cosserat, and H Fargier. "Constraint-based Modeling and Exploitation of a Vehicle Range at Renault's: Requirement analysis and complexity study". In: *Workshop on Configuration*. Ed. by Lothat Hotz and Alois Haselböck. Lisbon, 2010, pp. 33–39.

[11] CAL CAN-in Automation. "CAN Application Layer for Industrial Applications". In: *CiA Draft Standard DS-201 to DS-207, Version* (1996). URL: http://scholar.google.com/scholar?hl=en\&btnG=Search\&q=intitle:CAN+Application+layer+for+industrial+applications\#5.

[12] Jakob Axelson and Avenir Kobetski. "On The Conceptual Design of a Dynamic Component Model for Reconfigurabl e AUTOSAR Systems". In: *5th Workshop on AdaPtive and Reconfigurable Embedded Systems (APRES 2013)*. Philadelphia, 2013, pp. 42–45.

[13] Alberto Ballesteros, Marco Wagner, and Dieter Zöbel. "SOAcom : Designing Service communication in adaptive automotive networks". In: *8th IEEE International Symposium on Industrial Embedded Systems*. Porto, 2013. ISBN: 9781479906581.

[14] Luciano Baresi et al. "Hybrid service-oriented architectures: a case-study in the automotive domain". In: *Proceedings of the 5th international workshop on Software Engineering and Middleware*. 2005, pp. 62–68. URL: http://dl.acm.org/citation.cfm?id=1108487.

[15] Douglas K. Barry. *Web Services, Service-Oriented Architectures, and Cloud Computing (The Savvy Manager's Guides)*. Morgan Kaufmann, 2003, p. 264. ISBN: 1558609067. URL: http://www.amazon.com/Services-Service-Oriented-Architectures-Computing-Managers/dp/1558609067.

[16] Basil Becker, Holger Giese, and Stefan Neumann. "Model-based extension of autosar for architectural online reconfiguration". In: *Models 09*. 2009, pp. 123–137. URL: http://www.springerlink.com/index/V72R656310462M4N.pdf.

[17] Michael Bell. *Service-Oriented Modeling Framework*. Wiley Publishing Inc., 2008. Chap. 1, p. 366. ISBN: 978-0-470-14111-3.

[18] Gorka Benguria and Parque Tecnologico De Zamudio. "A platform independent model for service oriented architectures". In: *Enterprise Interoperability* (2007), pp. 23–32. URL: http://www.springerlink.com/index/JX7567507X101H6M.pdf.

[19] Uwe Berg, Philipp Wojke, and Dieter Zöbel. *Projekt: Visuelle Rückfahrassistenz für Gespanne*. Koblenz, 2007. URL: http://www.uni-koblenz-landau.de/koblenz/fb4/ist/AGZoebel/downloadbereich/flyer/EZlenk_Flyer.pdf/at_download/file.

[20] Uwe Berg and Dieter Zöbel. "Gestaltung der Mensch-Maschine-Interaktion von Lenkas- sistenzsystemen zur Unterstützung der Rückwärtsfahrt von Fahrzeugen mit Anhänger". In: *Mechatronik 2007 - Innovative Produktentwicklung* 1971 (2007), pp. 575–588.

[21] Uwe Berg and Dieter Zöbel. "Visual Steering Assistance for Backing-Up Vehicles with One-axle Trailer". In: *Vision in Vehicles 11*. Dublin, 2006.

[22] Christian Berger and Matthias Tichy. "Towards Transactional Self-Adaptation for AUTOSAR on the Example of a Collision Detection System". In: *GI Jahrestagung 2012*. 2012, pp. 853–862.

[23] Sascha Berkessel. "Überarbeitung und Erweiterung des Fahrsimulators für den prototypischen Test unterschiedlicher Rückfahrassistenzsysteme". Master Thesis. University of Koblenz-Landau, 2012, p. 111.

[24] Barry Boehm. "A Spiral Modell of Software Development and Enhancement". In: *Coumputer* 21.5 (1988), pp. 61–72.

[25] Hendrik Bohn, Andreas Bobek, and Frank Golatowski. "SIRENA-Service Infrastructure for Real-time Embedded Networked Devices". In: *International Conference on Mobile Communications and Learning Technologies*. 2006. ISBN: 0769525520. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1628289.

[26] David Booth et al. *Web Services Architecture*. 2004. URL: http://www.w3.org/TR/ws-arch/wsa.pdf.

[27] David Bridges and Shervin Mostashfi. "Dynamic Orchestration of the Sensor Web (DOSW)". In: *2008 International Symposium on Collaborative Technologies and Systems*. Ieee, May 2008, pp. 88–94. ISBN: 978-1-4244-2248-7. DOI: 10.1109/CTS.2008.4543917. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4543917.

[28] Antonio Brogi et al. “A service-oriented model for embedded peer-to-peer systems”. In: *Electronic Notes in Theoretical Computer Science* 194.4 (Apr. 2008), pp. 5–22. ISSN: 15710661. DOI: 10.1016/j.entcs.2008.03.096. URL: http://linkinghub.elsevier.com/retrieve/pii/S1571066108002016.

[29] H Bubb. “Der Fahrprozess Informationsverarbeitung durch den Fahrer”. In: *Tagungsband Technischer Kongress 2002*. 2002. URL: http://scholar.google.com/scholar?hl=en\&btnG=Search\&q=intitle:Der+Fahrprozess+Informationsverarbeitung+durch+den+Fahrer\#0.

[30] Edmund Burke. *Reflections on the Revolution in France*. London: James Dodsley, 1790. URL: http://www.gutenberg.org/ebooks/15679.

[31] Valeria Cardellini et al. “Flow-Based Service Selection forWeb Service Composition Supporting Multiple QoS Classes”. In: *IEEE International Conference on Web Services (ICWS 2007)*. Icws. Ieee, July 2007, pp. 743–750. ISBN: 0-7695-2924-0. DOI: 10.1109/ICWS.2007.91. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4279667.

[32] Marlene Caroselli. *Leadership Skills for Managers*. McGraw-Hill, 2000, p. 169. ISBN: 0071364307. URL: http://www.goodreads.com/book/show/1508582.Leadership_Skills_for_Managers.

[33] S. Cavalieri. “Meeting Real-Time Constraints in CAN”. In: *IEEE Transactions on Industrial Informatics* 1.2 (May 2005), pp. 124–135. ISSN: 1551-3203. DOI: 10.1109/TII.2005.844429. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1430655.

[34] WC Chang and CS Wu. “Optimizing the Dynamic Composition of Web Service Components”. In: *Journal of Information Technology and Applications* 2.4 (2008), pp. 227–234. URL: http://140.126.5.184/jita_web/publish/vol2_num4/OptimizingtheDynamicCompositionofWebServiceComponents.pdf.

[35] CiA. *CANopen*. Tech. rep. CiA. URL: http://www.can-cia.org/index.php?id=canopen.

[36] CiA. *DeviceNet*. URL: http://www.can-cia.org/index.php?id=176.

[37] Arthur C. Clarke. *Profiles of the Future*. 1st ed. SCIENTIFIC BOOK CLUB, 1962. ISBN: B001GKPECC.

[38] RI Davis et al. “Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised”. In: *Real-Time Systems* 35.0 (2007), pp. 239–272. URL: http://www.springerlink.com/index/8N32720737877071.pdf.

[39] Steffen Dienst and Stefan Kühne. *Realisierung fachlicher Services auf Basis von Android-Konzepten*. Tech. rep. Leipzig: Leipzig University, 2011, pp. 1–6.

[40] Marco Dorigo and Gianni Di Caro. “The ant colony optimization meta-heuristic”. In: *New Ideas in Optimization*. Ed. by David Corne, Marco Dorigo, and Fred Glover. New York: McGraw-Hill, 1999, p. 450. ISBN: 0077095065. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.184.955.

[41] M. Dröschel and M. Wiemers. *Das V-Modell 97: der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz*. München: Oldenburg Wissenschaftsverlag, 2000, p. 710. ISBN: 3486250868. URL: http://books.google.de/books/about/Das_V_Modell_97.html?id=lcHvAAAACAAJ\&pgis=1.

[42] Surekha Durvasula et al. “SOA Practitioners’ Guide, Part 2, SOA Reference Architecture”. In: *Combined Effort* (2006), pp. 1–52. URL: http://scholar.google.com/scholar?hl=en\&btnG=Search\&q=intitle:SOA+Practitioners+?+Guide+Part+2+SOA+Reference+Architecture\#0.

[43] Edsger W. Dijkstra. "The Humble Programmer - ACM Turing Award Lecture". In: *Communications of the ACM* 15.10 (1972), pp. 895–866. URL: http://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD340.html.

[44] Michael Eichhorn, Martin Pfannenstein, and Eckehard Steinbach. "A flexible in-vehicle HMI architecture based on web technologies". In: *Proceedings of the 2nd international workshop on Multimodal interfaces for automotive applications - MIAA 2010* (2010), pp. 9–12. DOI: 10.1145/2002368.2002374. URL: http://portal.acm.org/citation.cfm?doid=2002368.2002374.

[45] Michael Eichhorn et al. "A SOA-based middleware concept for in-vehicle service discovery and device integration". In: *2010 IEEE Intelligent Vehicles Symposium*. Ieee, June 2010, pp. 663–669. ISBN: 978-1-4244-7866-8. DOI: 10.1109/IVS.2010.5547977. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5547977.

[46] Brian Elvesæ ter and Cyril Carrez. "Model-driven Service Engineering with SoaML". In: *Service Engineering* 25 (2011), pp. 25–54. URL: http://books.google.com/books?hl=en\&lr=\&id=piuG4bzu9doC\&oi=fnd\&pg=PA25\&dq=Model-driven+Service+Engineering+with+SoaML\&ots=Dc7bz5capG\&sig=IAi6XocTJttE6MjkXBQmKyr1qqE.

[47] Vina Ermagan et al. "Towards tool support for service-oriented development of embedded automotive systems". In: *Tagungsband Dagstuhl-Workshop MBEES : Modellbasierte Entwicklung eingebetteter Systeme III*. 2007, pp. 1–23.

[48] JA Estefan et al. *Reference Architecture Foundation for Service Oriented Architecture*. 2009. URL: http://scholar.google.com/scholar?hl=en\&btnG=Search\&q=intitle:Reference+Architecture+Foundation+for+Service+Oriented+Architecture\#2.

[49] Claudiu Farcas et al. "Addressing the Integration Challenge for Avionics and Automotive Systems-From Components to Rich Services". In: *Proceedings of the IEEE* 98.4 (2010), pp. 562–583. URL: http://ieeexplore.ieee.org/xpl/login.jsp?tp=\&arnumber=5433050\&url=http\%3A\%2F\%2Fieeexplore.ieee.org\%2Fxpls\%2Fabs_all.jsp\%3Farnumber\%3D5433050.

[50] P. Finger and K. Zeppenfeld. *SOA und WebServices*. Ed. by O. Günther et al. Heidelberg: Springer Berlin Heidelberg, 2009, p. 128. ISBN: 978-3-540-76990-3.

[51] Howard Foster et al. "A model-driven approach to dynamic and adaptive service brokering using modes". In: *Lecture Notes in Computer Science* 5364 (2008), pp. 558–564. URL: http://www.springerlink.com/index/u42977p04750840r.pdf.

[52] Viktor Friesen. *Dynamically Self-Configuring Automotive Systems - Scenario and System Requirements*. Tech. rep. DySCAS PMC, 2007, p. 134.

[53] J Gacnik and O Haeger. "Service-oriented architecture for future driver assistance systems". In: *FISITA World Congress 2008*. 2008. URL: http://en.scientificcommons.org/34613146.

[54] Yan Gao et al. "Optimal Web Services Selection Using Dynamic Programming". In: *11th IEEE Symposium on Computers and Communications (ISCC'06)*. Ieee, 2006, pp. 365–370. ISBN: 0-7695-2588-1. DOI: 10.1109/ISCC.2006.116. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1691056.

[55] H Garcia-Molina. "Elections in a Distributed Computing System". In: *IEEE Transactions on Computers* C-31.1 (Jan. 1982), pp. 48–59. ISSN: 0018-9340. DOI: 10.1109/TC.1982.1675885. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1675885.

[56] M. García Valls and P. Basanta Val. "A real-time perspective of service composition: Key concepts and some contributions". In: *Journal of Systems Architecture* 59.10 (Nov. 2013), pp. 1414–1423. ISSN: 13837621. DOI: 10.1016/j.sysarc.2013.06.008. URL: http://linkinghub.elsevier.com/retrieve/pii/S1383762113001239.

[57] M Garcia-Valls, I Rodríguez-López, and L Fernandez-Villar. "iLAND An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time Systems". In: *IEEE Transactions on Industrial Informatics* 9.1 (2013), pp. 228–236. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6198329.

[58] Michael Gebhart et al. "SoaML-basierter Entwurf eines dienstorientierten Überwachungssystems". In: *40. Jahrestagung der Gesellschaft für Informatik* 1 (2010). URL: http://www.researchgate.net/publication/221386038_SoaML-basierter_Entwurf_eines_dienstorientierten_berwachungssystems/file/79e41503f3b08e096f.pdf.

[59] Holger Giese et al. *Modular design and verification of component-based mechatronic systems with online-reconfiguration*. 2004. DOI: 10.1145/1041685.1029920.

[60] Google. *Google Maps JavaScript API v3: Elevation Service*. 2014. URL: https://developers.google.com/maps/documentation/javascript/elevation.

[61] Michael Götz and Fabian Keicher. "Konzeption und Konstruktion von Winkelsensorik für die Vermessung von Zweiachs-Drehschemel-Anhängern". Student Research Project. Heilbronn University, 2013, p. 33.

[62] Jeff Gray et al. "DSLs: the good, the bad, and the ugly". In: *Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA Companion '08*. New York, New York, USA: ACM Press, Oct. 2008, p. 791. ISBN: 9781605582207. DOI: 10.1145/1449814.1449863. URL: http://dl.acm.org/citation.cfm?id=1449814.1449863.

[63] Georg Grossmann, Michael Schrefl, and Markus Stumptner. "Model-driven framework for runtime adaptation of web service compositions". In: *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems - SEAMS '11* (2011), p. 184. DOI: 10.1145/1988008.1988034. URL: http://portal.acm.org/citation.cfm?doid=1988008.1988034.

[64] Hugo Haas and Allen Brown. *Web Services Glossary, W3C Working Group Note 11 February 2004*. 2004. URL: http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/.

[65] Stefan Hack and Markus Lindemann. *Enterprise SOA roadmap*. Bonn: Galileo Press, 2008. URL: http://www.galileo-press.de/download/dateien/1436/sappress_enterprise_soa_roadmap.pdf.

[66] Handelsblatt. *Lebensdauer eines Autos steigt kaum noch*. Feb. 2008. URL: http://www.handelsblatt.com/auto/nachrichten/lebensdauer-eines-autos-steigt-kaum-noch/2918710.html.

[67] Steven Helmis. "Datenqualität". In: *Webbasierte Datenintegration*. 2009, pp. 7–23. ISBN: 978-3-8348-9280-5.

[68] Hermann Hesse. *Steps*. 1941. URL: http://mindmastery.wordpress.com/2007/02/17/hermann-hesse-steps-stufen/.

[69] David Hollingsworth. *The Workflow Reference Model*. Winchester, 1995.

[70] Zhenqiu Huang et al. "Effective Pruning Algorithm for QoS-Aware Service Composition". In: *2009 IEEE Conference on Commerce and Enterprise Computing*. i. Ieee, July 2009, pp. 519–522. ISBN: 978-0-7695-3755-9. DOI: 10.1109/CEC.2009.41. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5210745.

[71] International Organization for Standardization. *ISO 15765-2:2011 Road vehicles - Diagnostic communication over Controller Area Network (DoCAN) - Part 2: Transport protocol and network layer services*. 2011. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=46045.

[72] Akihito Iwai and Mikio Aoyama. "Automotive Cloud Service Systems Based on Service-Oriented Architecture and Its Evaluation". In: *2011 IEEE 4th International Conference on Cloud Computing* (July 2011), pp. 638–645. DOI: 10.1109/CLOUD.2011.119. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6008765.

[73] Isabell Jahnich, Ina Podolski, and Achim Rettberg. "Towards a Middleware Approach for a Self-configurable Automotive Embedded System". In: *Lectue Notes in Computer Science* 5287 (2008), pp. 55–65.

[74] Nicolai Josuttis. *SOA in Practice: The Art of Distributed System Design*. 1st ed. O'Reily Media, 2007, p. 342. ISBN: 978-0596529550.

[75] JY Jung, BD Adelstein, and SR Ellis. "Discriminability of prediction artifacts in a time-delayed virtual environment". In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 44.5 (2000), pp. 499–502. URL: http://pro.sagepub.com/content/44/5/499.short.

[76] Jörg Kaiser, Cristiano Brudna, and Carlos Mitidieri. "COSMIC: A real-time event-based middleware for the CAN-bus". In: *Journal of Systems and Software* 77.1 (July 2005), pp. 27–36. ISSN: 01641212. DOI: 10.1016/j.jss.2003.12.037. URL: http://linkinghub.elsevier.com/retrieve/pii/S0164121204001347.

[77] K Kim, G Geon, and Seongsoo Hong. "Resource-conscious customization of CORBA for CAN-based distributed embedded systems". In: *Proceedings Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000) (Cat. No. PR00607)* (2000), pp. 34–41. DOI: 10.1109/ISORC.2000.839509. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=839509.

[78] Xenofon Koutsoukos et al. "OASiS : A Service-Oriented Architecture for Ambient-Aware Sensor Networks". In: *Lectue Notes in Computer Science* 4888 (2007), pp. 125–149.

[79] H Kreger. "Web services conceptual architecture (WSCA 1.0)". In: *IBM Software Group* May (2001). URL: http://www.csd.uoc.gr/~hy565/newpage/docs/pdfs/papers/wsca.pdf.

[80] Kim LEMON. "Introduction to the Universal Measurement and Calibration Protocol XCP". eng. In: *SAE transactions* 112.7 (2003), pp. 482–488. ISSN: 0096-736X. URL: http://cat.inist.fr/?aModele=afficheN\&cpsidt=16125195.

[81] Stefan Lankes, Andreas Jabs, and T Bernmerl. "Integration of a CAN-based connection-oriented communication model into Real-Time CORBA". In: *International Parallel and Distrubuted Processing Symposium*. Vol. 00. C. 2003. ISBN: 0769519261. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1213239.

[82] Yang Li et al. "QoS-aware Service Composition in Service Overlay Networks". In: *IEEE International Conference on Web Services (ICWS 2007)*. Icws. Ieee, July 2007, pp. 703–710. ISBN: 0-7695-2924-0. DOI: 10.1109/ICWS.2007.148. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4279662.

[83] Ying Li et al. "QoS-Driven Dynamic Reconfiguration of the SOA Based Software". In: *2010 International Conference on Service Sciences*. Ieee, 2010, pp. 99–104. ISBN: 978-1-4244-6603-0. DOI: 10.1109/ICSS.2010.58. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5494320.

[84] Jin Liang and Klara Nahrstedt. "Service Composition for Advanced Multimedia Applications". In: *ACM Multimedia Computing and Networking (MMCN'05)*. Ed. by Surendar Chandra and Nalini Venkatasubramanian. Vol. 1. c. San Jose, CA, Jan. 2005, pp. 228–240. DOI: 10.1117/12.592307. URL: http://proceedings.spiedigitallibrary.org/proceeding.aspx?articleid=858192.

[85] S Liu et al. "A dynamic web service selection strategy with QoS global optimization based on multi-objective genetic algorithm". In: *Lectue Notes in Computer Science* 3795 (2005), pp. 84–89. URL: http://www.springerlink.com/index/Y67U055622245Q6U.pdf.

[86] Juan López et al. "A Middleware Architecture for Unmanned Aircraft Avionics". In: *Proceedings of the 2007 ACM/IFIP/USENIX international conference on Middleware companion*. 2007, p. 24. ISBN: 9781595939357.

[87] Jochen Ludewig. "Models in software engineering - an introduction". In: *Software and Systems Modeling* 2.1 (Mar. 2003), pp. 5–14. ISSN: 1619-1366. DOI: 10.1007/s10270-003-0020-3. URL: http://dblp.uni-trier.de/db/journals/sosym/sosym2.html\#Ludewig03.

[88] C. M. MacKenzie et al. *Reference Model for Service Oriented*. Tech. rep. October. OASIS, 2006, p. 31.

[89] Salvatore T. March and Gerald F. Smith. "Design and natural science research on information technology". In: *Decision Support Systems* 15.4 (Dec. 1995), pp. 251–266. ISSN: 01679236. DOI: 10.1016/0167-9236(94)00041-2. URL: http://linkinghub.elsevier.com/retrieve/pii/0167923694000412.

[90] H Martorell et al. "Towards Dynamic Updates In AUTOSAR". In: *SAFECOMP 2013 - Workshop CARS (2nd Workshop on Critical Automotive applications : Robustness & Safety)*. Toulouse, 2013. URL: http://hal.archives-ouvertes.fr/hal-00848361/.

[91] Christoph Mathas. "Der Service-Lebenszyklus". In: *SOA intern*. 1st ed. Carl Hanser Verlag, 2007. Chap. 3.8, p. 292. ISBN: 3446411895.

[92] P Mayer, A Schroeder, and N Koch. "MDD4SOA: Model-Driven Service Orchestration". In: *Enterprise Distributed Object Computing Conference 2008 EDOC 08 12th International IEEE*. Ieee, 2008, pp. 203–212. ISBN: 9780769533735. DOI: 10.1109/EDOC.2008.55. URL: http://portal.acm.org/citation.cfm?id=1437901.1438849.

[93] Philip Mayer, Andreas Schroeder, and Nora Koch. "A Model-Driven Approach to Service Orchestration". In: *2008 IEEE International Conference on Services Computing*. Ieee, July 2008, pp. 533–536. ISBN: 978-0-7695-3283-7. DOI: 10.1109/SCC.2008.91. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4578572.

[94] Willis E. McNelly. *Interview with Frank and Beverly Herbert*. Fullerton, CA, 1969. URL: http://www.sinanvural.com/seksek/inien/tvd/tvd2.htm.

[95] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and how to develop domain-specific languages". In: *ACM Computing Surveys* 37.4 (Dec. 2005), pp. 316–344. ISSN: 03600300. DOI: 10.1145/1118890.1118892. URL: http://dl.acm.org/citation.cfm?id=1118890.1118892.

[96] Microsoft. *System Requirements for Windows Server 2012 R2 Essentials*. 2013. URL: http://technet.microsoft.com/de-de/library/dn383626.aspx.

[97] Megha Mohabey et al. "A Combinatorial Procurement Auction for QoS-Aware Web Services Composition". In: *2007 IEEE International Conference on Automation Science and Engineering*. Ieee, Sept. 2007, pp. 716–721. ISBN: 978-1-4244-1153-5. DOI: 10.1109/COASE.2007.4341711. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4341711.

[98] E.G. Nadhan. "Seven Steps to a Service-oriented Evolution". In: *Business Integration Journal* 1 (2004), pp. 41–44.

[99] ODVA. *DeviceNet - Technical Overview*. Tech. rep. The Open DeviceNet Vendor Association (ODVA), 2004, pp. 1–8. URL: http://onlinelibrary.wiley.com/doi/10.1002/9781118445983.ch2/summary.

[100] OMG. *OMG Unified Modeling Language 2.4.1 - Superstructure specification*. 2011.

[101] OMG. *Service oriented architecture Modeling Language (SoaML) - Specification for the UML Profile and Metamodel for Services*. Needham, MA, 2009.

[102] OMG. *Service oriented architecture Modeling Language (SoaML) Specification v.1.0.1*. 2012.

[103] Mark Panahi, Weiran Nie, and Kwei-Jay Lin. "The Design of Middleware Support for Real-Time SOA". In: *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing* (Mar. 2011), pp. 117–124. DOI: 10.1109/ISORC.2011.24. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5753599.

[104] Michael P. Papazoglou and Willem-Jan Van Den Heuvel. "Service-oriented design and development methodology". In: *International Journal of Web Engineering and Technology* 2.4 (2006), p. 412. ISSN: 1476-1289. DOI: 10.1504/IJWET.2006.010423. URL: http://www.inderscience.com/link.php?id=10423.

[105] Dirk Pingel. "Der SOA Entwicklungsprozess". In: *SOA Expertenwissen*. Ed. by G. Starke and S. Tilkov. 2007, pp. 187,200.

[106] Klaus Pohl et al., eds. *Model-Based Engineering of Embedded Systems*. Springer Berlin Heidelberg, 2012, p. 229. ISBN: 9783642346132.

[107] Fang Qiqing et al. "A Global QoS Optimizing Web Services Selection Algorithm Based on MOACO for Dynamic Web Service Composition". In: *2009 International Forum on Information Technology and Applications*. Ieee, May 2009, pp. 37–42. ISBN: 978-0-7695-3600-2. DOI: 10.1109/IFITA.2009.91. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5231511.

[108] S Veera Ragavan et al. "Services Integration Framework for Vehicle Telematics". In: *Third international conference on Intelligent robotics and applications*. 2010, pp. 636–648.

[109] Toni Reichelt et al. "IP Based Transport Abstraction for Middleware Technologies". In: *International Conference on Networking and Services (ICNS '07)*. Ieee, June 2007, pp. 39–39. ISBN: 978-0-7695-2858-9. DOI: 10.1109/ICNS.2007.76. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4438288.

[110] H. Renkewitz and J. Conradi. "On the effects of tracking errors and latency for Augmented Reality interaction". In: *Virtuelle und Erweiterte Realität, 2. Workshop der GI-Fachgruppe VR/AR*. Ed. by T. Kuhlen, L. Kobbelt, and S. Müller. Aachen: Shaker Verlag, 2005, pp. 95–106.

[111] D. Rocco et al. "Domain-specific Web service discovery with service class descriptions". In: *IEEE International Conference on Web Services (ICWS'05)*. Ieee, 2005. ISBN: 0-7695-2409-5. DOI: 10.1109/ICWS.2005.49. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1530838.

[112] M Röckl. "Integration of Car-2-Car communication as a virtual sensor in automotive sensor fusion for advanced driver assistance systems". In: *FISITA 2008*. 1. 2008. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.140.5067.

[113] Pablo Rossi and Zahir Tari. "Software Metrics for the Efficient Execution of Mobile Services". In: *Emerging Web Services Technology*. Ed. by Cesare Pautasso and Christoph Bussler. Birkhäuser Basel, 2007, pp. 135–152.

[114] Winston Royce. "Managing the Development of Large Software Systems". In: *IEEE WESCON*. 1970.

[115] Antoine de Saint-Exupéry. "L'Avion". In: *Terre des Hommes*. 1939. Chap. III.

[116] Hans-Werner Schaal. "Ethernet und IP im Kraftfahrzeug". In: *Elektronik Automotive* 4 (2012), pp. 38–41. URL: http://www.vector.com/portal/medien/cmc/press/PON/Ethernet_IP_ElektronikAutomotive_201204_PressArticle_DE.pdf.

[117] Jörg Schäuffele and Thomas Zurawka. *Automotive Software Engineering*. 5th. Heidelberg: Springer Vieweg, 2013. ISBN: 9783834824691.

[118] August-Wilhelm Scheer. "ARIS House of Business Engineering - Konzept zur Beschreibung und Ausführung von Referenzmodellen". In: *Entwicklungsstand und Entwicklungsperspektiven der Referenzmodellierung*. Ed. by Jörg Becker, Michael Rosemann, and Reinhard Schütte. 52. Münster: Institut für Wirtschaftsinformatik der Westfälischen Wilhelms-Universität Münster, 1997. Chap. 1, pp. 3–15.

[119] Roland Schmelzer. *zapthink's Service-Oriented Architecture Roadmap*. 2005. URL: http://www.zapthink.com/2005/10/31/zapthinks-service-oriented-architecture-roadmap/.

[120] D.C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". en. In: *Computer* 39.2 (Feb. 2006), pp. 25–31. ISSN: 0018-9162. DOI: 10.1109/MC.2006.58. URL: http://www.computer.org/csdl/mags/co/2006/02/r2025.html.

[121] Andreas Scholz et al. "eSOA - Service Oriented Architectures adapted for embedded networks". In: *2009 7th IEEE International Conference on Industrial Informatics*. 1. Ieee, June 2009, pp. 599–605. ISBN: 978-1-4244-3759-7. DOI: 10.1109/INDIN.2009.5195871. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5195871.

[122] H Shokry and M Ali Babar. *Dynamic software product line architectures using service based computing for automotive systems*. Tech. rep. 2008. URL: http://ulir.ul.ie/handle/10344/1897.

[123] Jan Sonnenberg. "A distributed in-vehicle service architecture using dynamically created web Services". In: *IEEE International Symposium on Consumer Electronics (ISCE 2010)*. Ieee, June 2010, pp. 1–5. ISBN: 978-1-4244-6671-9. DOI: 10.1109/ISCE.2010.5523715. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5523715.

[124] David Sprott. *Cloud-SOA Meta Model L1*. 2011. URL: http://davidsprottsblog.blogspot.de/2011/06/cloud-soa-meta-model-l1.html.

[125] Thomas Stahl. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt Verlag; Auflage: 1, 2005, p. 410. ISBN: 3898643107. URL: http://www.amazon.com/Modellgetriebene-Softwareentwicklung-Techniken-Engineering-Management/dp/3898643107.

[126] Sebastian Stein and Konstantin Ivanov. *Vorgehensmodell zur entwicklung von geschäftsservicen*. Tech. rep. 2007. URL: http://sebstein.hpfsc.de/publications/stein2007ie.pdf.

[127] Anja Strunk. "QoS-Aware Service Composition: A Survey". In: *2010 Eighth IEEE European Conference on Web Services*. 1. Ieee, Dec. 2010, pp. 67–74. ISBN: 978-1-4244-9397-5. DOI: 10.1109/ECOWS.2010.16. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5693246.

[128] Longji Tang, Jing Dong, and Tu Peng. "A Generic Model of Enterprise Service-Oriented Architecture". In: *2008 IEEE International Symposium on Service-Oriented System Engineering* (Dec. 2008), pp. 1–7. DOI: 10.1109/SOSE.2008.37. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4730454.

[129] Fei Tao et al. "FC-PACO-RM: a parallel method for service composition optimal-selection in cloud manufacturing system". In: *IEEE Transactions on Industrial Informatics* 9.4 (2013), pp. 2023 –2033. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6376181.

[130] The AUTOSAR development partnership. *AUTOSAR - The Worldwide Automotive Standard for E/E Systems*. Tech. rep. The AUTOSAR development partnership, 2011, p. 4.

[131] Oliver Thomas, Katrina Leyking, and Michael Scheid. "Serviceorientierte Vorgehensmodelle: Überblick, Klassifikation und Vergleich". In: *Informatik-Spektrum* 33.4 (Nov. 2009), pp. 363–379. ISSN: 0170-6012. DOI: 10.1007/s00287-009-0399-5. URL: http://www.springerlink.com/index/10.1007/s00287-009-0399-5.

[132] Bruce Tognazzini. "Principles, Techniques, and Ethics of Stage Magic and Their Application to Human Interface Design". In: *InterCHI '93*. 1993, pp. 355–362. ISBN: 0-89791-575-5. DOI: 10.1145/169059.169284.

[133] Wolfgang Trumler, Markus Helbig, and Andreas Pietzowski. "Self-configuration and Self-healing in AUTOSAR". In: *Proceedings of the 14th Asia Pacific Automtive Engineering Conference*. SAE International, Aug. 2007. DOI: 10.4271/2007-01-3507. URL: http://www.sae.org/technical/papers/2007-01-3507https://www.informatik.uni-augsburg.de/lehrstuehle/sik/publikationen/papers/2007_apac_tru/2007_apac_tru.pdf.

[134] Wei-Tek Tsai et al. "Service-oriented system engineering (SOSE) and its applications to embedded system development". In: *Service Oriented Computing and Applications* 1.1 (Mar. 2007), pp. 3–17. ISSN: 1863-2386. DOI: 10.1007/s11761-007-0003-2. URL: http://www.springerlink.com/index/10.1007/s11761-007-0003-2.

[135] VDE. *Hightech-Autos stellen Elektronik-Branche vor neue Herausforderungen*. 2007. URL: http://www.vde.com/de/fg/ITG/Aktuelles/Seiten/KFZ-Positionspapier.aspx.

[136] Vijay Vaishnavi and William Kuechler. *Design research in information systems*. Boca Raton, FL: Auerbach Publications, 2004, p. 248. ISBN: 978-1-4200-5932-8.

[137] Dung Vi. "Middleware for Dynamically Self-Configuring Automotive Systems". PhD thesis. Jönköping University, 2006, p. 48.

[138] Lars Völker. "SOME/IP - Die Middleware für Ethernet- basierte Kommunikation". In: *HANSER automotive networks* Special Issue 2013 (2013), pp. 17–19. URL: http://www.hanser-automotive.de/uploads/media/FA_BMW_stemp.pdf.

[139] Marco Wagner, Ansgar Meroth, and Dieter Zöbel. "Developing self-adaptive automotive systems". In: *Design Automation for Embedded Systems* (Dec. 2013). ISSN: 0929-5585. DOI: 10.1007/s10617-013-9124-3. URL: http://link.springer.com/10.1007/s10617-013-9124-3.

[140] Marco Wagner, Dieter Zöbel, and Ansgar Meroth. "A CAN-based Communication Model for Service- Oriented Driver Assistance Systems". In: *Proc. of the IEEE Vehicular Networking Conference (VNC) 2012*. Seoul, Korea: IEEE, 2012.

[141] Marco Wagner, Dieter Zöbel, and Ansgar Meroth. "Model-driven development of SOA-based Driver Assistance Systems". In: *Proc. of the 4th Workshop on Adaptive and Reconfigurable Embedded Systems (APRES '12) in conjunction with the IEEE / ACM CPSWeek '12*. Beijing, China: IEEE, 2012, pp. 27–32. URL: http://cps.kaist.ac.kr/apres2012/apres12.pdf.

[142] Marco Wagner, Dieter Zöbel, and Ansgar Meroth. "Model-driven development of SOA-based Driver Assistance Systems". In: *SIGBED Review* 10.1 (2013).

[143] Marco Wagner, Dieter Zöbel, and Ansgar Meroth. "Re-configuration in SOA-based adaptive Driver Assistance Systems". In: *in Proc. of the 6th Workshop on Adaptive and Reconfigurable Embedded Systems (APRES '14) in conjunction with the IEEE / ACM CPSWeek '14*. Berlin: IEEE Computer Society, 2014.

[144] Marco Wagner, Dieter Zöbel, and Ansgar Meroth. "SODA: Service-oriented Architecture for runtime adaptive Driver Assistance Systems". In: *Proceedings of the 17th IEEE Computer Society symposium (ISORC)*. Reno, NV: IEEE Computer Society, 2014.

[145] Marco Wagner, Dieter Zöbel, and Ansgar Meroth. "Towards an adaptive Software and System Architecture for Driver Assistance Systems Introducing a new approach to address dynamically changing automotive software systems". In: *Proc. of the 4th IEEE International Conference on Computer Science and Information Technology (ICCSIT '2011)*. figure 2. Chengdu, China: IEEE, 2011, pp. 174–178.

[146] Marco Wagner, Dieter Zöbel, and Ansgar Meroth. "Towards runtime adaptation in AUTOSAR: adding Service-orientation to automotive software architecture". In: *9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'2014)*. Barcelona: IEEE Computer Society, 2014.

[147] Changlin Wan et al. "On Solving QoS-Aware Service Selection Problem with Service Composition". In: *2008 Seventh International Conference on Grid and Cooperative Computing*. Ieee, Oct. 2008, pp. 467–474. ISBN: 978-0-7695-3449-7. DOI: 10.1109/GCC.2008.75. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4662903.

[148] Mark Wilkins and Oracle. *Oracle Reference Architecture*. Tech. rep. September. Oracle Corporation, 2010, p. 74. URL: http://www.oracle.com/technetwork/topics/entarch/oracle-ra-soa-foundation-r3-1-176715.pdf.

[149] Matthias Wissmann. *carIT, eine strategische Aufgabe in der Automobilindustrie*. Frankfurt, 2011. URL: http://www.youtube.com/watch?v=WtCpxIoOKBM.

[150] Minghui Wu et al. "QoS-driven Global Optimization Approach for Large-scale Web Services Composition". In: *Journal of Computers* 6.7 (July 2011), pp. 1452–1460. ISSN: 1796-203X. DOI: 10.4304/jcp.6.7.1452-1460. URL: http://ojs.academypublisher.com/index.php/jcp/article/view/4879.

[151] Yi Xu and Jun Yan. "A Cloud Based Information Integration Platform for Smart Cars". In: *2nd International Conference on Security-enriched Urban Computing and Smart Grids*. 2011, pp. 241–250.

[152] Yadunandana Yellambalase and Min Choi. "Automatic Node Discovery in CAN (Controller Area Network) Controllers using Reserved Identifier Bits". In: *2007 IEEE Instrumentation & Measurement Technology Conference IMTC 2007* (May 2007), pp. 1–3. ISSN: 1091-5281. DOI: 10.1109/IMTC.2007.379338. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4258116.

[153] Tao Yu, Yue Zhang, and Kwei-Jay Lin. "Efficient algorithms for Web services selection with end-to-end QoS constraints". In: *ACM Transactions on the Web* 1.1 (May 2007), 6–es. ISSN: 15591131. DOI: 10.1145/1232722.1232728. URL: http://portal.acm.org/citation.cfm?doid=1232722.1232728.

[154] Xin Yuan and Xingming Liu. "Heuristic algorithms for multi-constrained quality of service routing". In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)* 2 (2001), pp. 844–853. DOI: 10.1109/INFCOM.2001.916275. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=916275.

[155] Alexander Zeeb. "Plug-And-Play-Lösung für AUTOSAR-Software-Komponenten". In: *Kompendium ausgewählter Fachartikel zur Elektronik-Entwicklung in verteilten Systemen*. 4th Editio. Stuttgart: Vector Informatik GmbH, 2013, pp. 6/22–6/26.

[156] Marc Zeller et al. "Towards Runtime Adaptation in AUTOSAR". In: *5th Workshop on AdaPtive and Reconfigurable Embedded Systems (APRES 2013)*. Philadelphia, 2013, pp. 38–41.

[157] Liangzhao Zeng and Boualem Benatallah. "QoS-aware middleware for web services composition". In: *IEEE Transactions on Software Engineering* 30.5 (2004), pp. 311–327. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1291834.

[158] Chun-Jie Zhou et al. "Self-organization of reconfigurable protocol stack for networked control systems". In: *International Journal of Automation and Computing* 8.2 (May 2011), pp. 221–235. ISSN: 1476-8186. DOI: 10.1007/s11633-011-0577-1. URL: http://www.springerlink.com/index/10.1007/s11633-011-0577-1.