Des Autom Embed Syst (2014) 18:199–221
DOI 10.1007/s10617-013-9124-3

Developing self-adaptive automotive systems

On the integration of service-orientation into automotive development processes

Marco Wagner · Ansgar Meroth · Dieter Zöbel

Received: 30 August 2013 / Accepted: 3 December 2013 / Published online: 10 December 2013

Abstract Future Driving Assistance Systems (DAS) will have to react to changes within the system at runtime. This might be the case in Car-to-X systems where the availability of communication partners changes dynamically. Another example are systems like DAS for truck and trailer combinations where a trailer might be disconnected and replaced by another one several times a day. State-of-the-art DAS are not capable of handling these runtime changes. In our approach we make usage of the principles of Service-orientation to generate self-adaptive DAS on architectural level. But this technical approach requires the definition of a development process that fits into the practices within the automotive industry. This paper introduces SOMA4DDAS, a model-based development process based on the UML profile SoaML. SOMA4DDAS describes a tri-phase procedure to transfer an idea for a DAS into a detailed specification of the application and the Services involved. These phases are integrated into the "core process for system and software development" (CPSSD), a standard process within the automotive industry. The paper illustrates the benefits of this approach by developing a truck and trailer DAS consisting of 13 different Services.

Keywords Service-oriented architecture · Driver assistance systems · Automotive development processes · Self-adaptive systems

1 Introduction

1.1 Development and state-of-the-art of advanced driver assistance systems (ADAS)

Driving is a complex set of different processes that constitute control loops on different levels. The most popular classification of driving tasks is according to [9] where the au-

M. Wagner (✉) · A. Meroth
Automotive Systems Engineering, Heilbronn University, Heilbronn, Germany
e-mail: marco.wagner@hs-heilbronn.de

A. Meroth
e-mail: ansgar.meroth@hs-heilbronn.de

D. Zöbel
Faculty of Computer Science, University of Koblenz-Landau, Koblenz, Germany
e-mail: zoebel@uni-koblenz.de

thor distinguishes three tasks: The primary task aims at the stabilization (e.g. longitudinal movement, lateral movement), the route keeping and the navigation of a car, the secondary task aims at operating the car devices (e.g. wipers, headlights etc.) and the tertiary task the operation of information, comfort and entertainment systems, often called infotainment [26]. ADAS developed in all three classes, mainly intended to the dimensions of safety (e.g. longitudinal and lateral stabilization, lane keeping, emergency breaking), comfort (e.g. headlight and wiping assistants) and efficiency (e.g. automated coasting). Automated and assisted driving is achieved by the interaction of different functional and physical instances which are connected via a network on physical level and make use of complex interfaces on the functional and logical level. The reasons for complexity are manifold. First, there is an increasing number of participating functional blocks (not necessarily corresponding to the number of physical units), second, the requirements to the interfaces grow with the functionality. Third, these functional blocks, whether they are in different physical devices or not, often come from different suppliers and not necessarily allow insight into their inner structure (e.g. their code) except for the interface. The big challenge for car manufacturers (often called OEMs) and first tier system suppliers is the integration on the base of a weak knowledge about the components. In [40] the authors find as the main reason of integration problems “Poorly communicated module objectives or requirements”. While the cited study considers mainly aspects of human behavior in integration, the car industry answers to the increasing safety, reliability and traceability requirements with a sophisticated development process organization that will be briefly discussed below. Future ADAS, however, will require totally different paradigms especially regarding situation awareness and changing configurations. Ad-hoc Communication between the so-called Ego-car, other vehicles, traffic participants and the infrastructure, often referred to as Car-to-X communication (See also: [19]) has immediate effect to the functionality and the behavior of hitherto self-contained vehicles [18]. While vehicle systems of the past could be comprehensively described during the design phase, future systems will be open to aftermarket extensions and variations. Typical examples are the retrofitting of infotainment and assistance systems with access to the car’s internal network, the integration of mobile devices (see e.g. [25]), and the varying configuration of installations, as in commercial vehicles, agricultural machinery or in truck and trailer combinations.

1.2 Development processes as a measure for more safety and reliability

In the past ten years, the automotive industry uses basically two approaches to solve the integration problem: The technical and the organizational one. On the organizational side, car industry takes advantage of highly standardized support processes, especially in requirements management, quality assurance, configuration management and project management. Standard models like CMMI [10], Automotive SPICE [27] and ISO TS 16949 [20] are widely spread out between OEMs and their suppliers. Technically, the development follows a systems engineering approach which breaks down the system requirements to subsystem and component levels, of which the functionality and properties are directly derived from those of the preceding system level. On each of them, logical, behavioral and technical models are created that correspond to a set of test cases for the future integration task on that level (see: [35]). This process is widely known as the “V-model”. It allows considering as many aspects of the system as possible in the specification phase and aims at comprehensive testing and integration. As a disadvantage, all requirements and possible systems configurations in one particular systems level should preferably be known at the beginning of the development phase of this level, and requirements changes often lead to disproportionately high additional work and expenses. Note, that the functional requirements of a vehicle are by far exceeded by quality requirements, e.g. regarding efficiency, maintainability, portability, usability and safety. In order to assure, for instance, functional safety, definitions of the safety lifecycle, a hazard analysis and risk assessment, and a functional safety concept in the beginning of the project are mandatory, as well as the subsequent specification of the technical safety requirements on each system level (ISO 26262), each strongly depending on a full understanding of the technical boundary conditions and behavior of the system. With increasing cost, variability and time-to-market demands, a higher re-use rate [22], the access to standardized and highly mature functional modules and a hierarchical model driven design (as e.g. shown in [11]) are necessary to cope with the integration challenge. Many of the above noted challenges have been addressed with the "AUTomotive Open System ARchitecture (AUTOSAR)" approach [21]. AUTOSAR considers a system architecture with highly reusable basic software, a runtime environment and a thoroughly specified application programming interface (API), as well as a development methodology. It strongly supports re-use, configuration and variant management and a homogenous system analysis and description. However, AUTOSAR does not support ad hoc communication with components that were not specified during the system design phase.

1.3 Self-adaptive automotive systems

As mentioned earlier, many future Driver Assistance Systems like the ones basing on Car-to-X communication set up the requirement to allow changes of the software and system architecture at runtime. We call these highly distributed, dynamically changing systems Distributed Driver Assistance Systems (DDAS). In contrast, even cutting-edge development approaches like AUTOSAR do not support the design of such systems. On the other hand runtime adaptive systems have been used in other domains for many years now. One popular method to create dynamically changing, distributed applications is the usage of Service-oriented Architecture (SOA). The basic idea of SOA is, that all functionalities are encapsulated into so called Services. These Services are reachable through a well-defined interface from anywhere in the network. Each Service holds a contract that describes the ways of accessing this functionality. In order to build an application the Services are composed by an orchestration algorithm. These characteristics perfectly match the requirements set up by future DDAS. The heterogeneity of the functionalities is hidden behind the interface. Furthermore, the interfaces ensure compatibility when delegating the development to suppliers. Runtime adaption is carried out by re-orchestrating the Services in the event of a system change. Furthermore SOA-based systems are able to handle another issue coming up with the appearance of DDAS. Since these systems are no longer planned in a central manner, it is quite likely that duplicates of functionalities may be present. The system must be able to handle the presence of these duplicates. In Service-oriented applications the orchestration algorithm is in charge for that. Therefore, these applications distinguish between Services Classes and Service Instances. While a Service Class describes the functionality offered, a Service Instance is an actual software module. In this sense, an application is defined at design time as a combination of Service Classes while it is formed at runtime by the selection of one Service Instance for each Service Class. In order to create the best application possible at that very moment each Service Instance is equipped with a Quality of Service parameter. This property is accessible through the interface of the Service and is used to determine the best composition of Service Instances possible. In order to use SOA to build DDAS we developed the "Service Oriented Driver Assistance" (SODA) framework. This framework merges all the advantages mentioned in a very efficient middleware. It is tailored to automotive systems in the sense of taking resource constraints into account and in basing on state-of-the-art automotive network systems. Besides it is organized in a completely distributed fashion to keep it manageable and to avoid single points of failure.

But fulfilling the requirements set up by future DDAS technically is not enough. A new approach must also be capable of being integrated in the development processes of the automotive industry. Therefore we extended the SODA framework with a phase-oriented model-driven design process. This paper will prove, that this design process can be integrated in today's automotive development schemes. We decided to pick up the so called "core process for system and software development" (CPSSD) published by Schäuffele and Zurawka in [35]. It is a well established model for system development in the automotive industry basing on the popular V-model approach.

The remainder of the paper is organized as follows: Sect. 2 gives an overview on Service-oriented approaches in the automotive domain as well as on development processes for SOA applications. Section 3 introduces our approach to develop self-adaptive Service-based DDAS and how this development process is integrated into CPSSD. In Sect. 4 a case study is presented to validate the development process. Finally, Sect. 5 concludes the paper.

2 Related work

2.1 Service-oriented architectures in the automotive domain

In recent years, there have been several approaches to apply SOA-principles to automotive software systems. Some of these approaches aim on keeping software modules re-usable. Shokry et al. in [36] are presenting an approach to use Service-based computing to manage software product lines in cars. The main idea is to create functionalities in the form of Services that are orchestrated at design time to build an application. However, this approach does not consider any changes at runtime. The approach presented by Krueger et al. in [23] follows the same ideas when describing the development of a Service-based central locking system. Besides the restriction on design-time orchestration the described middleware is using a real time Common Object Request Broker Architecture (RT CORBA) that runs on high level operating systems only. This fact in combination with the strict resource constraints in the automotive industry makes a breakthrough of this approach quite unlikely. Baresi et al. in [4] describe a system using an already existing SOA framework that suffers the same disadvantages. Through basing on Java the system requirements are too high for being deployed on most of the automotive electronic control units (ECU). The necessity of using relatively powerful hardware limits the field of application of these two approaches to for example the infotainment system of a vehicle. The papers [12] by Eichhorn et al. and [8], written by Bohn et al., are describing systems basing on the Device Profile for Web Services (DPWS). DPWS is very interesting for our problem scenario as it is tailored to be used on embedded systems. However, this standard uses IP-based communication which is quite different from the network systems used in today's cars. Additionally, the messages exchanged between the Services are based on XML files which produces a huge amount of traffic considering the transfer rates of automotive networks. Besides, [12] does only allow static, never changing configurations. The two approaches described by Xu and Yan in [39] and Ragavan et al. in [33] are focusing on a different scenario. Instead of implementing the internal functionalities as Services, they describe a gateway approach. These gateways offer internal data of the car to the outside world and vice versa. [39] is using this data to call Web Services located in the cloud, [33] on the other hand sets up an ECU using the Java-based

Open Services Gateway initiative (OSGi) to allow a vehicle to invoke Services offered by the outside world and vice versa. The approach of Gacnik et al. described in [14] follows a similar idea. The authors describe a traveling salesman problem where a connected DAS extends its functionality by using Web Services to for example take train time tables into account when navigating. Although the usage of gateways to encapsulate car functionalities into Services and allows to call Services from outside of the car is a very interesting concept it cannot be transferred into our problem domain. This is because the software components on the vehicle are not implemented as Services and thereby are not runtime adaptive. A last approach which is very interesting in this context is the iLAND project described for example in [15]. Although it is not targeting on automotive systems, it is a very interesting approach to bring Service-oriented computing principles into the embedded domain. Similar to our scenario, the idea is to automatically re-configure a number of embedded Services to create an application. Unfortunately, the process of re-configuration is shaped in a way that does set up the need for a central device that overlooks the whole system. This fact creates a single point of failure scenario which is not acceptable. The number of approaches to bring Service-orientation into the automotive domain proves the potential of this paradigm. However, none of the approaches discussed here completely suites the demands of this problem scenario. In order to close these gaps the SODA middleware has been developed.

2.2 Development of service-oriented applications

Through to the popularity of Service-orientation a huge number of process models to develop such systems has been published in recent years. In 2009 Thomas, Leyking and Scheid identified 21 different approaches in [38]. 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. However, none of them suits to the domain of automotive SOA solutions. Instead of developing yet another model, we decided to find a process model that can be customized to this special scenario and integrated into the CPSSD development process. In order to do so, criteria have been developed and the available approaches have been evaluated based on these. The following criteria have been defined:

1. Completeness of the specification phase
2. Independence of 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

Our first criterion is 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 is that the starting position at the very beginning of the process should be variable. This is important because the process model should allow new developments as well as migrating existing systems into SOA. The fourth criterion is that tool support should be given. Using a tool that for example allows modeling the system graphically reduces development time. In addition, implemented validation functionalities decrease the probability of semantic errors. 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. 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 is that the development process has to be capable of being integrated into CPSSD. Using these criteria, eleven process models are analyzed. The first one is a model proposed by Stein and Ivanov in [37]. 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 belong to the domain of Web Services. A similar model, the Enterprise SOA Roadmap method is presented in [17]. This model also emphasizes on business modeling since only one of the five steps to be executed is technical. Both of the process models violate criteria two that they shouldn't restrict the area of application. Other approaches lack of concrete modeling techniques. Pingel [32] for example, introduces a technology independent five phase model extending well-known approaches. Another approach in this category is a proposal of Mathas [24] which extends the software lifecycle model by adding some SOA-specific tasks and roles while staying coarse-grained. The Service-oriented Modeling Framework developed by Bell is quite generic, too [6]. The idea of the author is to design a concrete process model for every case of application derived from his abstract methodology. Bell also proposes a special design notation which violates the criterion of using a widely-used modeling language. All these models are 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 models "Service-oriented design and development" [31] by Papazoglou and van den Heuvel and "Creating Service-oriented Architectures (CSOA)" [5] developed by Barry are technical in nature. Both of them are phase-oriented and contain practical techniques to be performed in those phases. Through basing on modeling languages like the business-oriented "Business Process Modeling Language (BPML)" or the "Business Process Execution Language for Web Services (WS-BPEL)" they cannot be used for other fields of application without major changes. This fact violates criterion two. Another approach is presented by Nadhan in [28]. 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. In doing so criterion three is violated. 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 in [16]. 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 specifying the architecture of the overall system which violates the criterion of enabling the user to carry out a complete system specification. Another methodology using SoaML introduced in [13] closely follows the processes defined in the Model-driven architecture (MDA) approach published by the Object Management Group. Tool support is granted by the modeling tool "Modelio". This process model defines several specification steps within the computational independent model and the platform independent model of MDA. The approach is very close to "Service-oriented Modeling and Architecture (SOMA)" presented in [2]. This phase-oriented lifecycle model is based on the “Rational Software Architect” and is also free of any restrictions with respect of the area of application. Both of the lastly named methodologies are fitting the criteria set up earlier in this paper. The reasons why SOMA is favored is being more focused on technical issues and offering a more straightforward workflow. Furthermore, the phases of SOMA can be integrated into CPSSD more easily.

3 Developing service-oriented driver assistance systems

In order to develop self-adaptive Distributed Driver Assistance Systems based on the SODA middleware we created a model-based development process based on SOMA. In this section we will explain this development procedure and we will show how this can be integrated into a design process of the automotive industry.

3.1 The core process for system and software development

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. It is basing on the well-known V-model while being tailored to the specific needs of the automotive domain. Just like the V-model it 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 part 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. Compared to similar development models like for example the waterfall model, introduced by Royce [34] or the spiral model, published by Boehm in [7] its main difference is the extension of the integration and testing phase. This addition leads to a better link between the specification and the test proceedings. 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 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 1 presents the V-model in the automotive-specific variant CPSSD. In this paper we focus on the specification of self-adaptive automotive systems. In this sense we restrict the rest of the paper to the left arm of the “V”. According to [35] 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

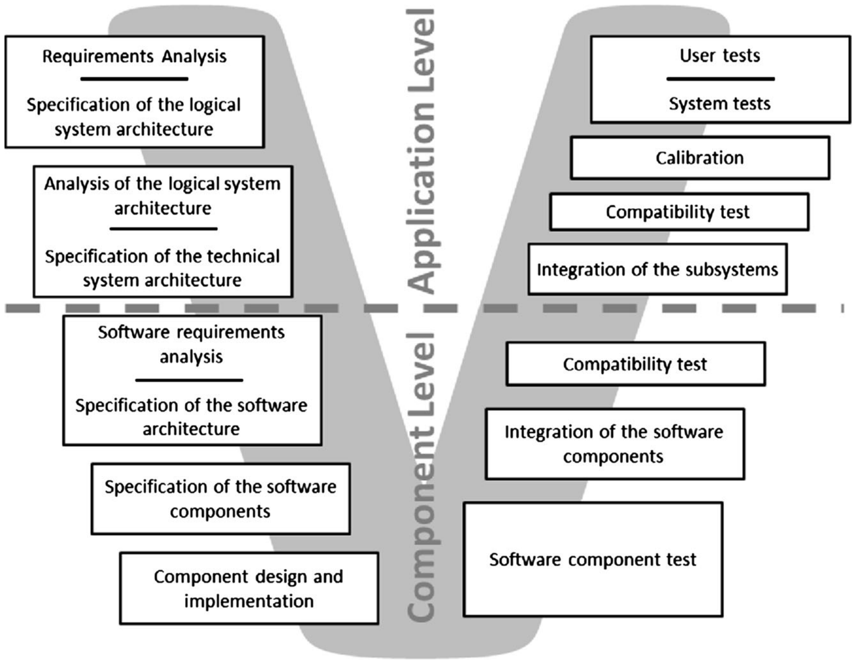
Requirements Analysis
Specification of the logical system architecture
Analysis of the logical system architecture
Specification of the technical system architecture
Software requirements analysis
Specification of the software architecture
Specification of the software components
Component design and implementation
Application Level
Component Level
User tests
System tests
Calibration
Compatibility test
Integration of the subsystems
Compatibility test
Integration of the software components
Software component test

Fig. 1 The CPSSD process [35]

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 propose a two-stage process. In the first part of this process, the software architecture of each component is defined. 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 and 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 these components are even purchased as commercial of the shelf solutions.

3.2 Integrating the development of Service-oriented systems into CPSSD

As mentioned in Sect. 2.2 we have chosen SOMA as a base of our development process for Service-oriented applications. The SOMA methodology is a phase-oriented process model first published by Arsanjani in [1] in 2004. It is based completely on the modeling language SoaML. This ensures a consistent procedure where the outcome of each phase is documented in some kind of model which can directly be used in the next development step. Figure 2 gives an overview of the seven phases of SOMA. The first two of these

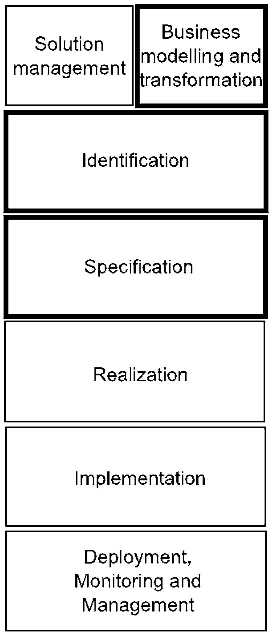
Solution management
Business modelling and transformation
Identification
Specification
Realization
Implementation
Deployment, Monitoring and Management

Fig. 2 The seven phases of the original SOMA methodology [2]

are run simultaneously. While “Business Modeling and Transformation” generates a first, semi-formal description of the requirements, “Solution Management” sets up the project management processes. These two steps are followed by “Identification” to derive possible Service candidates. The “Specification” phase enhances the candidate descriptions and transfers them into Service specifications. In the “Realization” step the focus swaps from functional to non-functional requirements that are now added to the Service description. As the specification work is done now, the “Implementation” phase guides the developer while writing code. In the last phase the software application is put into operation. Although SOMA is meant to be used in any kind of domain there are some issues that have to be worked out in our scenario of usage. These issues are:

– Extension of the domains of usage
– Enabling the development of self-adaptive systems
– Integration into CPSSD

In order to address the first two issues we did some refinements to the steps of the SOMA methodology. Furthermore instead of carrying out the whole lifecycle defined in SOMA we restrict our approach to the three phases marked in Fig. 2. These three steps are integrated into the left arm of CPSSD’s “V”. The reason for the restriction is our focus on the specification of the functional parts of adaptive embedded systems. This functional specification is finished within in this three phases. The remainder of this chapter will describe the refined process model to implement self-adaptive Service-based automotive applications.

The refined SOMA process model to develop Service-based application using the SODA middleware is called SOMA4DDAS. An overview of this process model is presented in Fig. 3. The first one of the three steps of SOMA4DDAS replaces SOMA’s “Business Modeling and Transformation”. We decided to not only change the context of this step but also it’s name to make clear that the refined step is capable of handling a much broader range of systems. In the remaining two steps title and purpose have been left unchanged compared to

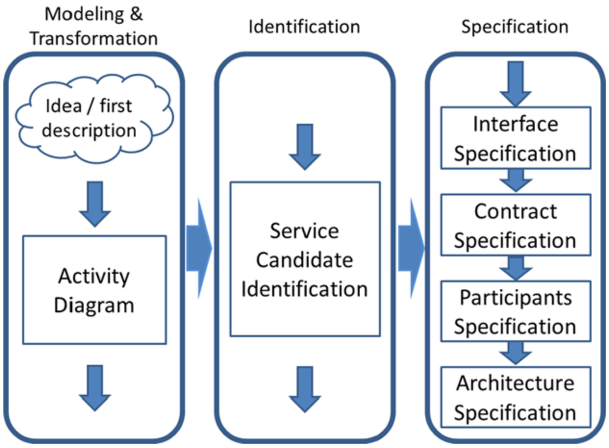

Fig. 3 The three phases of the SOMA4DDAS process model

SOMA. However, the procedures within these three steps have been significantly tailored to the development of self-adaptive DDAS.

Phase 1: Modeling and transformation In the first phase of SOMA4DDAS, called "Modeling and Transformation" the idea for a DDAS is brought into a first semi-formal description. In the corresponding phase in 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 it is not usable in the automotive domain without completely ignoring the semantics of BPMN. On the other hand we want to maintain SOMA4DDAS to be consistent. This means that the result of each phase has to be directly usable for the next one. In this specific case we have to find a replacement for BPMN and especially it's "task" stereotypes that are originally used in the specification phase. We decided to use UML2 Activity Diagrams. Similar to BPMN models workflows 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 4 shows a simplified DDAS modeled as an Activity Diagram. The different functionalities are 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 team conducting it 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 we restricted the number of nodes used in this development step to the six entities pictured in Fig. 4. The main element used here is the action node. It is the fundamental unit of executable functionality [29]. 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 node 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

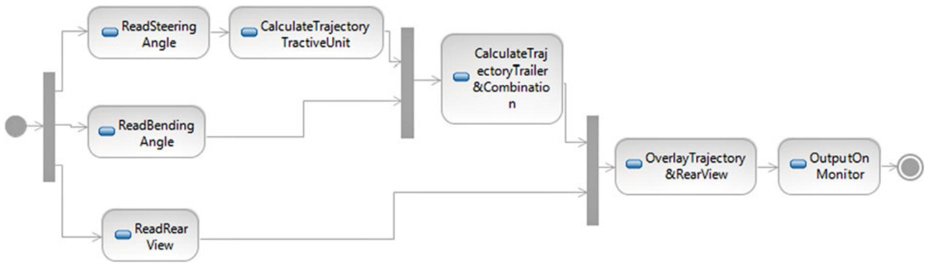
ReadSteering Angle
CalculateTrajectory TractiveUnit
CalculateTrajectoryTrailer &Combination
ReadBending Angle
OverlayTrajectory &RearView
OutputOn Monitor
ReadRear View

Fig. 4 An example for a DDAS modelled as an Activity Diagram

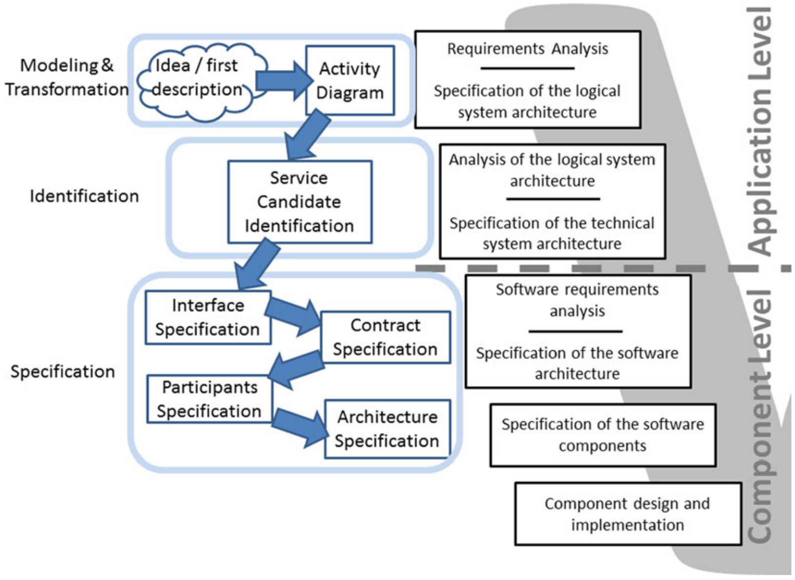
Modeling & Transformation
Idea / first description
Activity Diagram
Identification
Service Candidate Identification
Specification
Interface Specification
Contract Specification
Participants Specification
Architecture Specification
Requirements Analysis
Specification of the logical system architecture
Analysis of the logical system architecture
Specification of the technical system architecture
Software requirements analysis
Specification of the software architecture
Specification of the software components
Component design and implementation
Application Level
Component Level

Fig. 5 Integration of SOMA4DDAS into the V-model of CPSSD

the control or data flow after the system has been invoked. It's 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. We are convinced that this set of nodes is sufficient in the scope of DAS.

This first phase within SOMA4DDAS is to transform an idea for a DDAS into a coarse-grained abstraction. In this sense it is the counterpart to the specification of the Logical System Architecture in CPSSD as shown in Fig. 5. The Activity Diagram fulfills all demands of a LSA for being an abstract solution without any 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. Furthermore, migration of an existing system into the SODA framework is possible. In this scenario code could be analyzed and converted into an Activity Diagram.

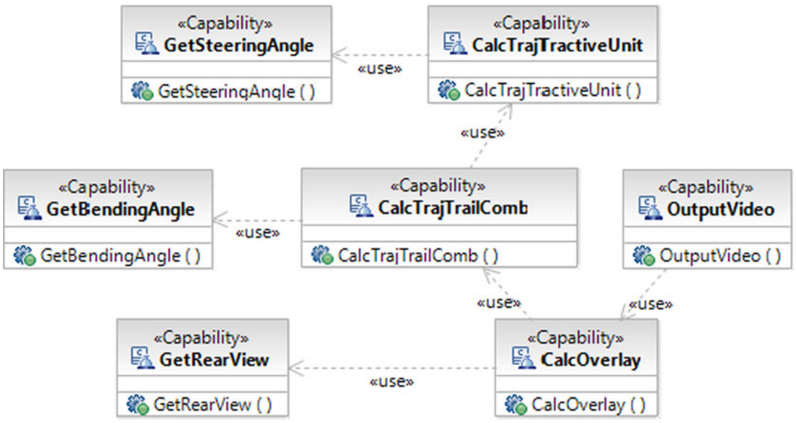

Fig. 6 The Service Candidates derived from the example Activity Diagram

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 in CPSSD as concrete decisions on how the system will be realized are made. For example, in this step the development team determines which Services will be needed and what functionality will be carried out in each of these entities. In this sense we integrated the Identification phase into CPSSD as the definition of the Technical System Architecture as illustrated in Fig. 5. SoaML defines a specialized stereotype for these Service Candidates: 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 high number of functionalities 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 limit the possibilities when assigning them to ECUs in a distributed embedded system. In order to overcome these drawbacks we want our architecture to be rather fine-grained with a relatively high number of Services. In fact, we want to shrink down each Service to offer only one functionality. To achieve this, we go through the Activity Diagram generated in the previous phase and transform every activity into a Capability. 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 Fig. 6. The developed model is now very fine-grained. Each Service 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.

Phase 3: Specification The third and last phase of the SOMA4DDAS 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 components level in CPSSD's "V". For this reason we arrange this phase as a replacement for the software architecture and software component specification as illustrated in Fig. 5. Due to it's extensiveness the Specification phase is split up into four sub-phases. The first one of these defines the Service Interface. It is followed by the Specification of the Service Contracts. 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 Service Architecture.

Phase 3.1: Service interface specification In this first sub-phase a Service Interface is derived from every Capability defined in phase 2. In other words, every future Service is now represented by it's Service Interface. The original SOMA process model recommends to specify a number of sub-interfaces to each Service Interface. 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 our scenario of usage. As we want to use the SoaML specification for runtime adaption it is essential to obtain a common structure. To achieve this the process of Service Interface specification has been refined. First of all, in SOMA4DDAS every Operation offered by a Capability is converted into an UML Interface. As we decided to produce very fine-grained Services this leads to one Interface per Service. This Interface is called a Provided Interface as it reflects the functionality provided by the Service. With this step the functionality offered by a Service is specified. In order to use the specification for decentralized reconfiguration this information is not enough. Therefor, the second part of the refinement demands to generate one additional Interface for every external functionality needed by the Service to execute it's task. These Interfaces are called Requested Interfaces as the Service itself requests their functionality. They are also 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 to SOMA is very important. By adding this information each Service is now self-aware of it's state. This is because, every Service does now have the knowledge of what others Services have to be reachable. It can now explore it's environment by executing the Service Discovery. If at least one implementation of each Service requested is currently available, the Service can put itself into the reachable state. Furthermore, if there is more than one implementation available, it is able to decide on 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 it's 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 Services requested enables the SODA system to re-configure itself in a decentralized manner without the usage of a central configuration entity.

Phase 3.2: Service contract specification In a SOA-based system contracts formalize the exchange of information between the Service and the calling entity. Defining a contract means specifying two things. First of all it is to determine the roles within the contract. Roles define which partners interact with on 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 it's functionality. Again, SOMA is not very precise in this step. In matters of the roles it simply refers to SoaML and states that the developer is free to use any type of description listed there. This opens up three options namely "Service Interface", "Interface" or

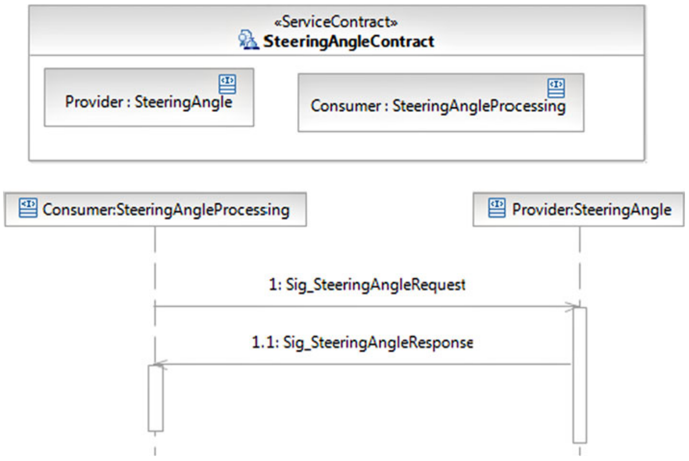
«ServiceContract»
SteeringAngleContract
Provider : SteeringAngle
Consumer : SteeringAngleProcessing
Consumer:SteeringAngleProcessing
Provider:SteeringAngle
1: Sig_SteeringAngleRequest
1.1: Sig_SteeringAngleResponse

Fig. 7 An example contract illustrating to roles and the Sequence Chart of the communication

"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 adaption. This is why we added several constraints to SOMA4DDAS. First of all UML Sequence Charts are chosen to illustrate the communication cycles. Sequence Charts are easily understandable regardless being quite flexible. Furthermore, 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 through to 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 Service Interface possibly combining several Interfaces.

In order to keep the Sequence Charts easily parsable, 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. To illustrate this interaction asynchronous messages are used. They are enriched with signals. These signal represent the data packets exchanged and are defined separately. 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 7 illustrates such a contract. 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 SoaML Participants is quiet vague. The SoaML specification [30] 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". SOMA interprets this stereotype as some kind of computing unit that is able to execute the implementation of such a Service. In this sense it uses this sub-phase to map the different

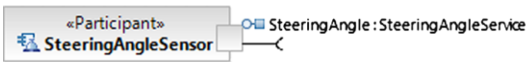

Fig. 8 An example Participant with it's Service Point

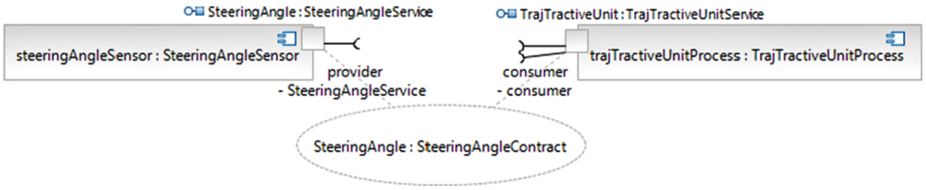
SteeringAngle : SteeringAngleService
steeringAngleSensor : SteeringAngleSensor
provider
- SteeringAngleService
TrajTractiveUnit : TrajTractiveUnitService
trajTractiveUnitProcess : TrajTractiveUnitProcess
consumer
- consumer
SteeringAngle : SteeringAngleContract

Fig. 9 An excerpt from a example Service Architecture

Services to hardware entities, for example to different servers when creating a Web Services application. 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. However, a model-driven development approach needs to be consistent. As the next step within the process model uses Participants we have to introduce them anyway. We use the broadly framed definition of the stereotype at this point. In SOMA4DDAS 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.

Figure 8 illustrates such a Participant. The Service Interface 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 system is defined. It is modeled in form of a SoaML Service Architecture which illustrates the relationships of the Participants involved using their ports and contracts. In the first step of the architecture definition the involved functionalities in form of Participants are selected. As the Services specified are very fine-grained and offer only one functionality each Participant holds only one Provided Interface within it's 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.

Figure 9 shows an excerpt of a Service Architecture. 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 Sequence Charts specified in the contract connecting their ports.

The specification of the Service Architecture 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 an automated tool that analyzes the SoaML model. More precisely, it parses the xml documents containing the Service Interface, Service Contract, Participants and Service Architecture 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. However, it is not within the scope of this paper as it is not addressing the functional specification of the system. Please refer to [3] were we described a process model picking up the information generated to design such a communication stack for more details. 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.

The right part of CPSSD's "V" which defines the testing and integration phases stays unchanged. However, it is planned to customize it as well, to achieve a direct generation and execution of testing procedures within the SOMA4DDAS process model.

4 Case study

The scenario that we focus on with the SODA framework is Driving Assistance for truck and trailer combinations. In this chapter one of these DDAS is presented.

Figure 10 illustrates a possible HMI of such a system. A monitor presents a picture of the area behind the trailer to the driver. Furthermore two trajectories are 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 of the vehicle. The inner, green lines illustrate 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

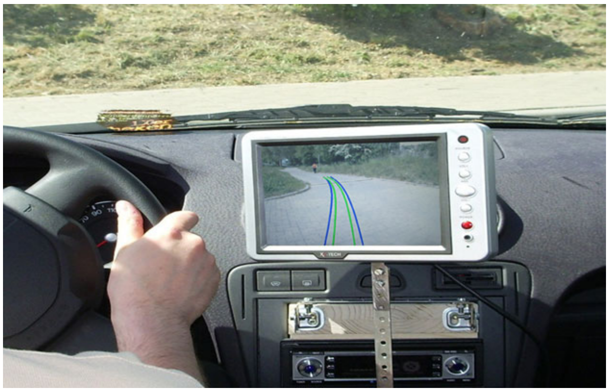
Fig. 10 The HMI of a Visual Assistance System to back up a trailer

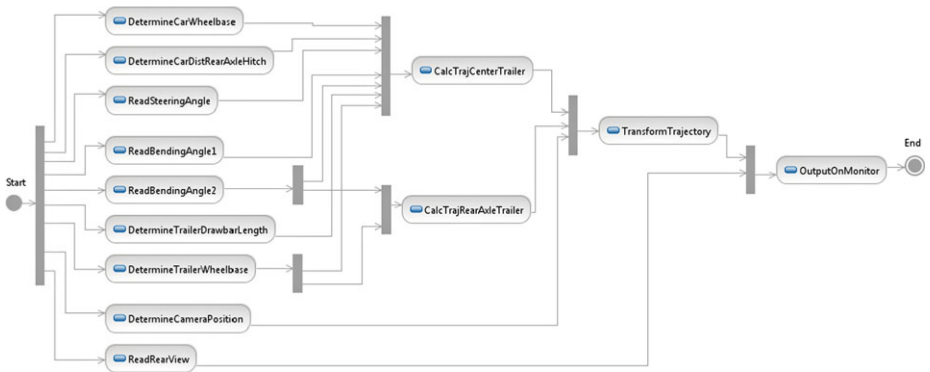

Fig. 11 The Activity Diagram of the DDAS

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. Furthermore, a number of dimensions of truck and trailer are needed like for example the wheelbases. In order to offer the visual HMI as shown in Fig. 10 a camera mounted to the back of the trailer and a monitor to output the video is needed. The rest of this chapter will explain how this kind of system has been developed using the SOMA4DDAS process model.

Phase 1: Modeling and transformation The first phase is to develop the Logical System Architecture, in our approach modeled as an Activity Diagram. The basis for this case study is an implementation of the DDAS described earlier on a driving simulator. The description of the functional properties as well as the code itself is used to define this first coarse-grained description. Figure 11 shows the Activity Diagram developed. The system is built by a combination of 13 functionalities. 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 Fig. 12. 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 one functionality. The relationship between the Activities of the Logical System Architecture are transferred to the technical one by "use" relations connecting the Capabilities.

Phase 3: Specification After defining the Technical System Architecture the development process turns from Application to Component level. In this sense the Capabilities identified are picked up one by one and specified in-depth. This paper 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 trailer attached.

The first step of this phase is to design the Service Interface of each Capability. This is done by first of all deriving one single Service Interface 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 Service Interface specification is to create a Requested Interface for every functionality needed by

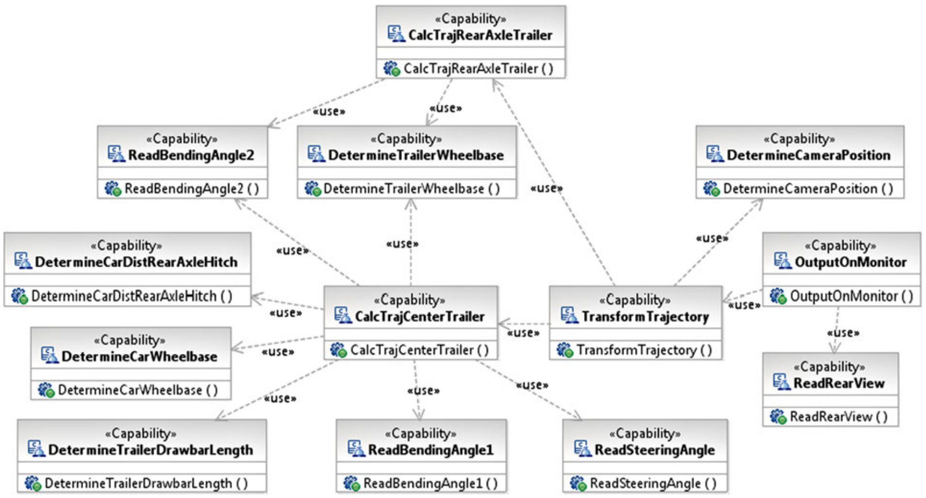
«Capability» CalcTrajRearAxleTrailer
CalcTrajRearAxleTrailer ()
«Capability» ReadBendingAngle2
ReadBendingAngle2 ()
«Capability» DetermineTrailerWheelbase
DetermineTrailerWheelbase ()
«Capability» DetermineCameraPosition
DetermineCameraPosition ()
«Capability» DetermineCarDistRearAxleHitch
DetermineCarDistRearAxleHitch ()
«Capability» CalcTrajCenterTrailer
CalcTrajCenterTrailer ()
«Capability» TransformTrajectory
TransformTrajectory ()
«Capability» OutputOnMonitor
OutputOnMonitor ()
«Capability» DetermineCarWheelbase
DetermineCarWheelbase ()
«Capability» ReadRearView
ReadRearView ()
«Capability» DetermineTrailerDrawbarLength
DetermineTrailerDrawbarLength ()
«Capability» ReadBendingAngle1
ReadBendingAngle1 ()
«Capability» ReadSteeringAngle
ReadSteeringAngle ()
«use»

Fig. 12 Overview of the Capabilities derived for the example application

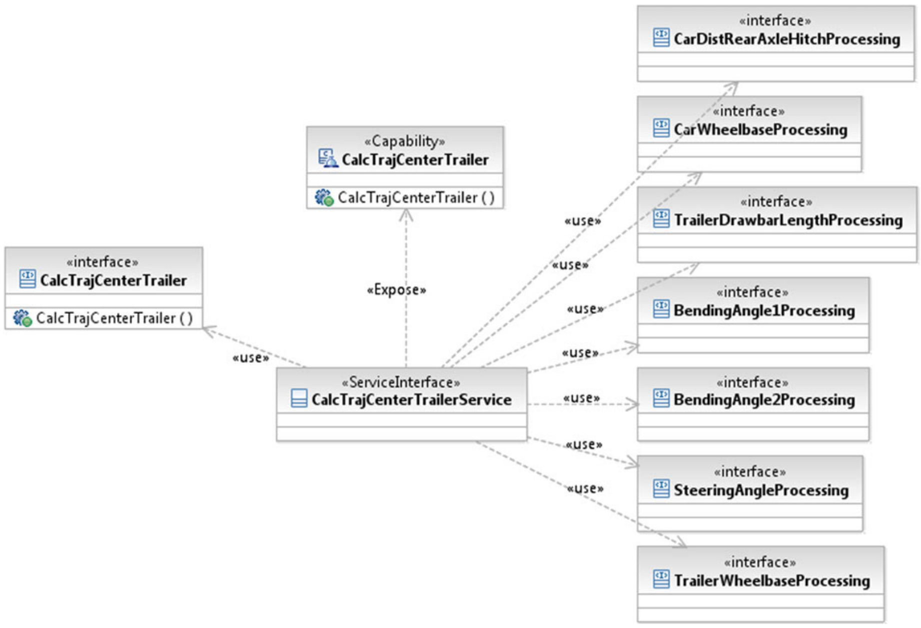
«interface» CarDistRearAxleHitchProcessing
«interface» CarWheelbaseProcessing
«Capability» CalcTrajCenterTrailer
CalcTrajCenterTrailer ()
«interface» TrailerDrawbarLengthProcessing
«interface» CalcTrajCenterTrailer
CalcTrajCenterTrailer ()
«Expose»
«interface» BendingAngle1Processing
«ServiceInterface» CalcTrajCenterTrailerService
«interface» BendingAngle2Processing
«interface» SteeringAngleProcessing
«interface» TrailerWheelbaseProcessing
«use»

Fig. 13 The Service Interface of the Service to calculate the trajectory of the center of the trailer

the future Service to execute it's logic. Figure 13 shows the formed Service Interface of the Service to calculate the trajectory of the trailer center point. In the middle the Service Interface is pictured. It is connected to one Provided Interface on the left and seven Requested Interface 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.

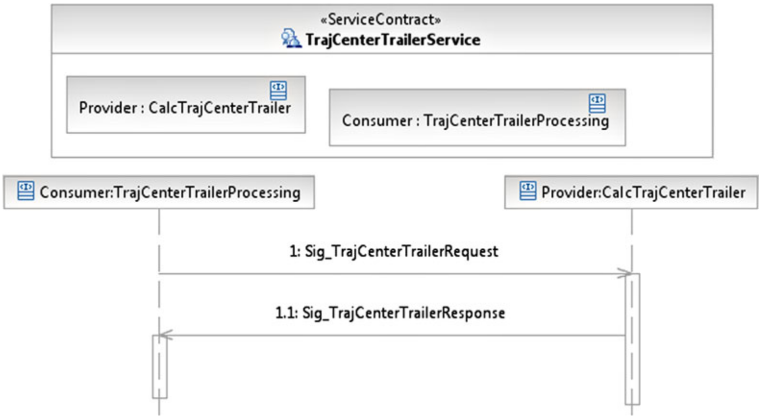
«ServiceContract»
TrajCenterTrailerService
Provider : CalcTrajCenterTrailer
Consumer : TrajCenterTrailerProcessing
Consumer:TrajCenterTrailerProcessing
Provider:CalcTrajCenterTrailer
1: Sig_TrajCenterTrailerRequest
1.1: Sig_TrajCenterTrailerResponse

Fig. 14 The contract of the Service to calculate the trajectory of the center of the trailer

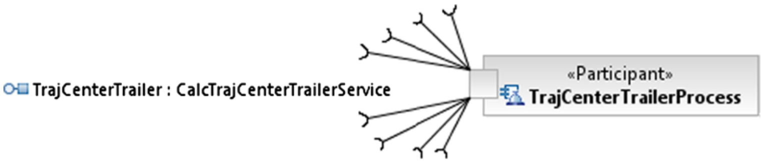

Fig. 15 The Participant of the example Service

Figure 14 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 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 15 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 Service Architecture is defined. In the presented use case all the Services developed are brought together to build the driving assistance application. Therefor 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 to connect two Participants is to define a role binding within the connecting contract. Figure 16 shows the complete Service Architecture of the specified application.

The last step of the left arm of CPSSD's "V" is to implement the specified Services. This has been done using the SoaML model derived through the SOMA4DDAS process.

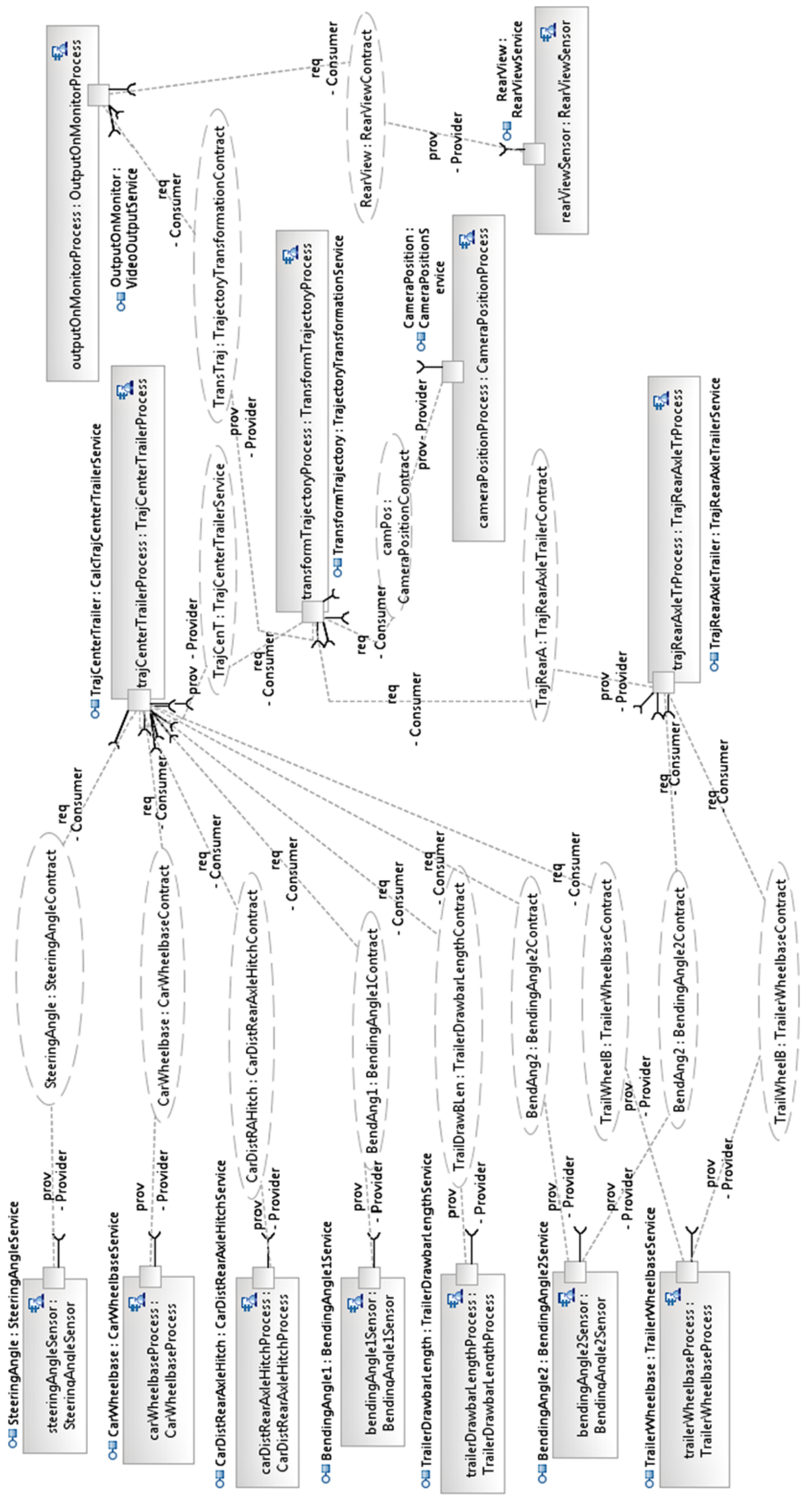
outputOnMonitorProcess : OutputOnMonitorProcess
OutputOnMonitor : VideoOutputService
req - Consumer
RearView : RearViewContract
prov - Provider
RearView : RearViewService
rearViewSensor : RearViewSensor
TransTraj : TrajectoryTransformationContract
transformTrajectoryProcess : TransformTrajectoryProcess
TransformTrajectory : TrajectoryTransformationService
CameraPosition : CameraPositionService
cameraPositionProcess : CameraPositionProcess
camPos : CameraPositionContract
TrajCenterTrailer : CalcTrajCenterTrailerService
trajCenterTrailerProcess : TrajCenterTrailerProcess
TrajCenT : TrajCenterTrailerService
TrajRearA : TrajRearAxleTrailerContract
trajRearAxleTrProcess : TrajRearAxleTrProcess
TrajRearAxleTrailer : TrajRearAxleTrailerService
SteeringAngle : SteeringAngleContract
CarWheelbase : CarWheelbaseContract
CarDistRAHitch : CarDistRearAxleHitchContract
BendAng1 : BendingAngle1Contract
TrailDrawBLen : TrailerDrawbarLengthContract
BendAng2 : BendingAngle2Contract
TrailWheelB : TrailerWheelbaseContract
SteeringAngle : SteeringAngleService
steeringAngleSensor : SteeringAngleSensor
CarWheelbase : CarWheelbaseService
carWheelbaseProcess : CarWheelbaseProcess
CarDistRearAxleHitch : CarDistRearAxleHitchService
carDistRearAxleHitchProcess : CarDistRearAxleHitchProcess
BendingAngle1 : BendingAngle1Service
bendingAngle1Sensor : BendingAngle1Sensor
TrailerDrawbarLength : TrailerDrawbarLengthService
trailerDrawbarLengthProcess : TrailerDrawbarLengthProcess
BendingAngle2 : BendingAngle2Service
bendingAngle2Sensor : BendingAngle2Sensor
TrailerWheelbase : TrailerWheelbaseService
trailerWheelbaseProcess : TrailerWheelbaseProcess

Fig. 16 The overall Service Architecture of the application

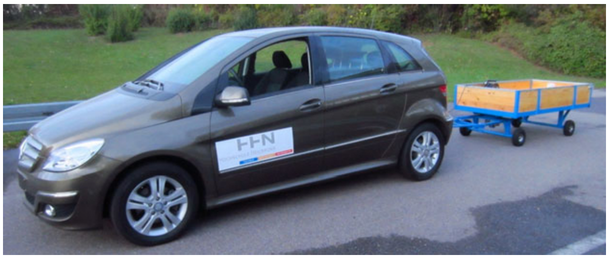

Fig. 17 Picture of the demonstrator vehicle

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 Fig. 17.

5 Conclusions

In our use case we have shown that SOMA4DDAS is suitable for designing self-adaptive Distributed Driving Assistance Systems and is capable of being integrated into the development processes of the automotive industry. Using SOMA4DDAS within the CPSSD model a Service-based system has been developed that supports the driver of a truck and trailer combination while backing up. By going through the three phases the application and the Services have been identified and specified. In the implementation phase afterwards the Services were implemented according to the SoaML model and integrated into the SODA middleware. This step was supported by an automation tool that determines information important to the development of the communication stack and generates a code skeleton. The overall application was installed on a demonstrator and successfully tested. The SOMA4DDAS process model turned out to be a precise, straight forward method to develop such systems.

Furthermore the usage of Service-orientation brought some additional benefit during the development of the example application. In this paper we focused on the left arm of CPSSD's "V". The right arm explains how the components implemented are tested and integrated. The development of the example application has shown that the integration of the components, implemented as Services in our framework, was relatively easy. The main reasons for this were the loose coupling and the well defined interfaces of the Services.

Future work will focus on extending SOMA4DDAS to the integration and testing phases. Furthermore the existing steps will be enriched by procedures to directly generate testing scenarios for the specified Services as well as for the overall application.

References

1. Arsanjani A (2004) Service-oriented modeling and architecture. http://www.ibm.com/developerworks/library/ws-soa-design1/

2. Arsanjani A, Ghosh S, Allam A, Abdollah T, Ganapathy S, Holley K (2008) SOMA: a method for developing service-oriented solutions. IBM Systems Journal 47(3):377–396. doi:10.1147/sj.473.0377
3. Ballesteros A, Wagner M, Dieter Z (2013) SOAcom: designing service communication in adaptive automotive networks. In: 8th IEEE international symposium on industrial embedded systems, Porto
4. Baresi L, Ghezzi C, Miele A, Miraz M (2005) Hybrid service-oriented architectures: a case-study in the automotive domain. In: Proceedings of the 5th international workshop on software engineering and middleware, pp 62–68
5. Barry DK (2003) Web services, service-oriented architectures, and cloud computing (the Savvy manager's guides). Morgan Kaufmann, San Mateo
6. Bell M (2008) Service-oriented modeling framework. In: Service-oriented modeling. Wiley, New York, p 366
7. Boehm B (1988) A spiral model of software development and enhancement. Computer 21(5):61–72
8. Bohn H, Bobek A, Golatowski F (2006) SIRENA-service infrastructure for real-time embedded networked devices: a service oriented framework for different domains. In: International conference on mobile communications and learning technologies
9. Bubb H (2002) Der Fahrprozess Informationsverarbeitung durch den Fahrer. In: Tagungsband Technischer Kongress
10. CMMI Institute: CMMI. http://www.cmmiinstitute.com/
11. Dubois H, Peraldi-Frati M, Lakhal F (2010) A model for requirements traceability in a heterogeneous model-based design process: application to automotive embedded systems. In: 15th IEEE international conference on engineering of complex computer systems (ICECCS). IEEE Press, New York, pp 233–242
12. Eichhorn M, Pfannenstein M, Muhra D, Steinbach E (2010) A SOA-based middleware concept for in-vehicle service discovery and device integration. In: IEEE intelligent vehicles symposium. IEEE Press, New York, pp 663–669. doi:10.1109/IVS.2010.5547977
13. Elvesæter B, Carrez C (2011) Model-driven service engineering with SoaML. Service Engineering 25:25–54
14. Gacnik J, Haeger O (2008) Service-oriented architecture for future driver assistance systems. In: FISITA world congress 2008
15. Garcia-Valls M, Rodríguez-López I, Fernandez-Villar L (2013) iLAND: an enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems. IEEE Transactions on Industrial Informatics 9(1):228–236
16. Gebhart M, Moßgraber J (2010) SoaML-basierter Entwurf eines dienstorientierten Überwachungssystems. 40. Jahrestagung der Gesellschaft für Informatik (1)
17. Hack S, Lindemann M (2008) Enterprise SOA roadmap. Galileo Press, Bonn
18. Hartenstein H, Laberteaux K (2010) vANET: vehicular applications and inter-networking technologies, vol 1. Wiley, New York
19. Hartenstein H, Laberteaux KP (2008) A tutorial survey on vehicular ad hoc networks. Communications Magazine, IEEE 46(6):164–171
20. IATF/ISO: ISO/TS 16949 (2009) Quality management systems. International Automotive Task Force
21. Kindel O, Friedrich M (2009) Softwareentwicklung mit AUTOSAR. dpunkt Verlag, Berlin
22. Krüger A, Hardung B, Kölzow T (2008) Reuse of software in automotive electronics. In: Navet N, Simonot-Lion F (eds) Automotive embedded systems handbook. CRC Press, Boca Raton, pp 283–292
23. Krüger I, Gupta D, Mathew R, Moorthy P (2004) Towards a process and tool-chain for service-oriented automotive software engineering. In: Workshop on software engineering for automotive systems
24. Mathas C (2007) SOA intern. Hanser Fachbuchverlag, Berlin
25. Meroth A (2012) The vehicle in the app opportunities and risks of open interfaces between vehicles and smartphones. In: 2nd CTI forum "Connected Car". CTI
26. Meroth A, Tolg B (2007) Infotainmentsysteme im Kraftfahrzeug: Grundlagen, Komponenten, Systeme und Anwendungen. Springer, Berlin
27. Mueller M, Hoermann K, Dittmann L, Zimmer J (2012) Automotive SPICE in Practice: surviving implementation and assessment. O'Reilly, New York
28. Nadhan E (2004) Seven steps to a service-oriented evolution. Business Integration Journal 1:41–44
29. OMG: OMG Unified Modeling Language 2.4.1—Superstructure specification (2011)
30. OMG: Service oriented architecture Modeling Language (SoaML) Specification v.1.0.1 (2012)
31. Papazoglou MP, Heuvel WJVD (2006) Service-oriented design and development methodology. International Journal of Web Engineering and Technology 2(4):412. doi:10.1504/IJWET.2006.010423
32. Pingel D (2007) Der SOA entwicklungsprozess. In: Starke G, Tilkov S (eds) SOA expertenwissen, pp 187–200
33. Ragavan SV, Ponnambalam SG, Ganapathy V, Teh J (2010) Services integration framework for vehicle telematics. In: Third international conference on intelligent robotics and applications, pp 636–648

34. Royce W (1970) Managing the development of large software systems. In: IEEE WESCON
35. Schäuffele J, Zurawka T (2013) Automotive software engineering, 5th edn. Springer, Heidelberg
36. Shokry H, Babar MA (2008) Dynamic software product line architectures using service based computing for automotive systems. Tech rep
37. Stein S, Ivanov K (2007) Vorgehensmodell zur entwicklung von geschäftsservicen. Tech rep
38. Thomas O, Leyking K, Scheid M (2009) Serviceorientierte Vorgehensmodelle: Überblick, Klassifikation und Vergleich. Informatik-Spektrum 33(4):363–379. doi:10.1007/s00287-009-0399-5
39. Xu Y, Yan J (2011) A cloud based information integration platform for smart cars. In: 2nd international conference on security-enriched urban computing and smart grids, pp 241–250
40. Zafar A, Ali S, Shahzad RK (2011) Investigating integration challenges and solutions in global software development. In: Frontiers of information technology (FIT). IEEE Press, New York, pp 291–297