On the Nature of Automotive Service Architectures

Preprint · March 2019

CITATIONS
0

READS
373

2 authors:

Vadim Cebotari
Technische Universität München
5 PUBLICATIONS **13** CITATIONS
SEE PROFILE

Stefan Kugele
Technische Hochschule Ingolstadt
64 PUBLICATIONS **301** CITATIONS
SEE PROFILE

This text is a preprint.
The final paper is accepted for publication in *Proceedings of the 2019 IEEE International Conference on Software Architecture Companion, ICSA Companion.*

On the Nature of Automotive Service Architectures

Vadim Cebotari
Department of Informatics
Technical University of Munich
Garching b. München, Germany
vadim.cebotari@tum.de

Stefan Kugele
Department of Informatics
Technical University of Munich
Garching b. München, Germany
stefan.kugele@tum.de

***Abstract*—*Context*: Current trends in the automotive industry like automated driving, machine intelligence, and digitisation introduce more and more complex functions and their interplay in in-vehicle architectures. To cope with this complexity, car makers are introducing particularly tailored service-oriented architectures. *Aim*: We aim at developing a *formal* and *methodological* approach to model and structure automotive service architectures. *Method*: We present a formal model to *specify* and *group* automotive service-oriented architectures following a classification scheme. Furthermore, a *well-defined property model* is introduced facilitating the specification and verification of deployment and run-time requirements. The approach is illustrated using a realistic case example. *Conclusion*: The presented approach works well for the used case example but needs to be further evaluated in depth in the future. However, we strongly believe that the framework is promising and will receive further attention in future work.**

***Index Terms*—service-oriented architecture, automotive, service classification, property model**

I. Introduction

During the last decades, thousands of mostly software-controlled functions running on a large number of electronic control units (ECU) were included in modern cars. Their particular characteristics reach from non-safety-critical to high safety and real-time critical functions. Driving forces are: (i) safety requirements, (ii) customer demand for more comfort and the newest infotainment systems, and (iii) advanced driver assistance systems allowing to reach hitherto unrivalled levels of driving automation (cf. [1]). Sophisticated smart capabilities and highly personalised functions are no longer a future vision.

Automated driving poses the highest functional safety requirements (i. e., ASIL D according to ISO 26262 [2]). Regulatory authorities will by law require to improve, i. e., (i) fix possible errors and (ii) update crucial components to the state-of-the-art, continuously. Vehicles are in use for more than a decade on average. During that time-span, information technology improves rapidly: E. g. an encryption algorithm used for backend communication cannot be considered as secure and state-of-the-art after 15 years anymore. Hence, possible attack surfaces need to be mitigated continuously.

Rooted in the consumer electronics industry, customers demand new levels of *personalisation* (and individualisation) for their vehicles. This extends the traditional aspects of personalisation such as vehicle colour, interior equipment, or options by far. Both customer demand and innovation push in addition to a quick reaction to competitors require exceptionally fast and light-weight software updates and, by that, enables the use of continuous integration (CI) technologies in the context of a fast-moving development process. *Continuous service integration and delivery* (CI/CD) helps to improve the quality and speed at which automotive software-based innovations are delivered to the customers' vehicles—going along with the DevOps way of thinking.

Traditional software development processes and methods are hardly capable of coping with these challenges. When designing a software architecture of a system, the primary goals are to develop flexible, adaptive and maintainable systems. *Service-oriented architectures* (SOA) are known for supporting the design of flexible systems. Still, having many services freely communicating with each other poses additional challenges. These challenges become especially evident when applying SOA to security- and safety-critical cyber-physical systems.

We formulate the following challenges concerning SOA in the context of security- and safety-critical cyber-physical systems: • *Visibility:* Should any service be able to access any other service? • *Deployment:* Is each service independently deployable? Are there any dependencies (e.g. versioning dependencies) that require services to be deployed together? Are there any restrictions that require services to be deployed on the same ECU (e. g., real-time requirements), or on different ECUs (e. g., safety requirements such as dislocality)?

In order to deal with these challenges, a well-defined structuring of services in logical abstraction (sub-)levels is required. Kugele et al. [3] address the problem of categorisation of automotive services and introduce the concepts of *service framework* and *service kit*, thus defining two logical abstraction levels for service structuring. Further, they describe some criteria for building service kits.

In this paper we aim, on the one side, to analyse in more depth the criteria for grouping services in logical abstraction (sub-)levels and, on the other side, to develop a formal model for specification of properties to be assigned to service groups. Hence, we pose the following research questions:

RQ1 Which criteria can be used for grouping services into logical abstraction (sub-)levels in security- and safety-critical service-oriented cyber-physical systems?

RQ2 How does a formal property model for service groups in order to fulfil above-mentioned visibility and deployment requirements look like?

Outline: The remainder of this paper is structured as follows. Section II summarises related work followed by the main approach in Section III. In Section IV, we present the case example and discuss the introduced classification scheme. Finally, we conclude in Section V and point to future work.

II. Related Work

Broy et al. [4] introduce a formal model for services by defining the *syntactic interface* of a service as a set of typed input and output channels, and the *semantic interface* by utilising a strictly causal function.

We extend the formal model for services defined by Broy et al., in particular, the *syntactic interface*, by *service interface elements*. We define the *service interface element* by a set of typed input and output channels, and the *service interface* as the union of its interface elements.

Kugele et al. [3] introduce the concept of a *service kit*, which is defined as a logical collection of *technically* or *conceptually* related services. We extend *service kits* and introduce the concept of *service groups*. They are a composition of services with specific properties. This approach introduces a formal *property model* to specify the properties of *service groups*.

Franca+ is a Component Definition Language (FCDL) [5] that provides support for the specification of *service groups* and service group properties. A *service group* is called in Franca+ a *component*. Components can be composed of other components to build more complex components. Properties of service groups can be expressed in Franca+ either by *component attributes* or by the definition of *custom tags* that are declared within the structured comments of Franca+. We will use Franca+ to demonstrate how the concepts introduced in this paper can be applied using a modelling framework in use.

Service groups and their properties are specified at development time, thus representing the development model of the service-oriented system. At run-time, an appropriate runtime model that enforces property adherence is required.

OSGi Service Platform [6] is an excellent runtime framework that supports visibility constraints between OSGi bundles. An OSGi bundle is a Java archive or a Web application archive file. Using the *Export-Package* header in the bundle's manifest file `MANIFEST.MF`, it is possible to declare which packages of the bundle are visible outside the bundle. If a package is not declared in this header, it is visible only within the bundle. Still, *OSGi Service Platform* does not support visibility constraints on different abstraction levels. The only level, on which visibility control is provided, is the level of bundles.

Approaches for service categorisation and classification are discussed in the literature (e. g., [7]–[9]). Most of them, though, address business information systems and focus on service categorisation and classification from a functional perspective.

We introduce a service classification scheme for safety-critical cyber-physical systems. The primary goal of our classification scheme is to support the building of service groups according to both well-defined classification criteria, and deployment and run-time system requirements.

III. Approach

A. Formal Specification

In the following, we formulate a formal model for service grouping on different levels of abstraction. Before presenting the model, we first introduce and explain needed concepts and definitions. The basic building blocks of the model are *services* and *service groups*. The concept of service that we use in our model is based on the definition given by Broy et al. [4].

Each service is described through its *service interfaces*. Let `SERVICE` denote the set of all services and `INTERFACE` the set of all interfaces. The mapping of services to a set of associated interfaces is defined by the function $\iota\colon \texttt{SERVICE} \to \mathcal{P}(\texttt{INTERFACE})$.

Each service interface $\mathfrak{i} \in \texttt{INTERFACE}$ is described through its *service interface elements*. We denote by `ELEMENT` the set of service interface elements. The mapping of service interfaces to associated service interface elements is given by the function $\epsilon\colon \texttt{INTERFACE} \to \mathcal{P}(\texttt{ELEMENT})$. Each service interface element $\mathfrak{e} \in \texttt{ELEMENT}$ is described by a set of typed input and output channels. Let C be the set of all channels, i. e., $\mathrm{C} = I \cup O$, where I denotes the set of typed input and O the set of typed output channels. Each channel $c \in \mathrm{C}$ is associated with a specified data type, and only messages of this data type can be exchanged through the channel c. Let `TYPE` denote the set of all data types. The assignment of data types to channels is described by the function $\mathrm{type}\colon \mathrm{C} \to \texttt{TYPE}$.

Definition 1 (Service Interface Element): A *service interface element* $\mathfrak{e} \in \texttt{ELEMENT}$ is denoted by $(I^{\mathfrak{e}} \triangleright O^{\mathfrak{e}})$, where $I^{\mathfrak{e}} \subseteq I$ and $O^{\mathfrak{e}} \subseteq O$.

In practical terms, the service interface element $\mathfrak{e}$ denotes a method of the service interface. The subset $I^{\mathfrak{e}}$ corresponds to the argument types and $O^{\mathfrak{e}}$ to the return types, respectively.

Example 1: Let us consider the service *HandsOffDetection-Service* that provides the interface *HandsOffDetectionInterface* (cf. Fig. 5). The service interface consists of three methods:

- `getStatusHandsOffDetection`,
- `deactivateHandsOffDetection`, and
- `activateHandsOffDetection`.

These methods represent the service interface elements of the service interface *HandsOffDetectionInterface*.

Definition 2 (Service Interface): A *service interface* $\mathfrak{i} \in \texttt{INTERFACE}$ is denoted by $\left(I^{\mathfrak{i}} \triangleright O^{\mathfrak{i}}\right)$ and defined by the union of its service interface elements: $\mathfrak{i} \stackrel{\text{def}}{=} \bigcup_{\mathfrak{e} \in \epsilon(\mathfrak{i})} \mathfrak{e}$.
Since a service interface element $\mathfrak{e} = (I^{\mathfrak{e}} \triangleright O^{\mathfrak{e}})$ is described by a set of typed input and output channels, we can rewrite the service interface definition as follows:

$$\mathfrak{i} \stackrel{\text{def}}{=} \bigcup_{\mathfrak{e}=(I^{\mathfrak{e}} \triangleright O^{\mathfrak{e}}) \in \epsilon(\mathfrak{i})} (I^{\mathfrak{e}} \triangleright O^{\mathfrak{e}}) = \left(\bigcup_{\mathfrak{e} \in \epsilon(\mathfrak{i})} I^{\mathfrak{e}}, \bigcup_{\mathfrak{e} \in \epsilon(\mathfrak{i})} O^{\mathfrak{e}} \right) = \left(I^{\mathfrak{i}} \triangleright O^{\mathfrak{i}}\right), \quad (1)$$

where $I^{\mathfrak{i}} \subseteq I$ and $O^{\mathfrak{i}} \subseteq O$.

Broy et al. [4] specify a service through its syntactic and semantic interfaces. The definition of the syntactic interface corresponds to our definition of a service interface. The semantic interface is defined by means of a strictly causal

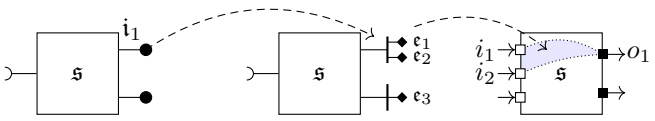

Fig. 1. Different views from left to right: Service $\mathfrak{s}$ provides a service interface $\mathfrak{i}_1$. It consists of the two service interface elements $\mathfrak{e}_1$ and $\mathfrak{e}_2$ (also known as methods from OOP). E. g. element $\mathfrak{e}_1$ consumes two arguments via the input channels i_1 and i_2 and returns the result via output channel o_1.

function that describes the interface behaviour. In this paper, the object of interest is the syntactic interface of a service. Thus, we refer always to the syntactic interface of a service when we use, in the following, the term service interface.

Definition 3 (Service (syntactic interface)): The *syntactic interface* $(I^{\mathfrak{s}} \triangleright O^{\mathfrak{s}})$ of a *service* $\mathfrak{s} \in \texttt{SERVICE}$ is defined by the union of its service interfaces:

$$(I^{\mathfrak{s}} \triangleright O^{\mathfrak{s}}) \stackrel{\text{def}}{=} \bigcup_{\mathfrak{i}=(I^{\mathfrak{i}} \triangleright O^{\mathfrak{i}}) \in \iota(\mathfrak{s})} \mathfrak{i}$$

From the representation of a service interface as a set of typed input and output channels follows:

$$= \bigcup_{(I^{\mathfrak{i}} \triangleright O^{\mathfrak{i}}) \in \iota(\mathfrak{s})} (I^{\mathfrak{i}} \triangleright O^{\mathfrak{i}}) = \left(\bigcup_{(I^{\mathfrak{i}} \triangleright O^{\mathfrak{i}}) \in \iota(\mathfrak{s})} I^{\mathfrak{i}}, \bigcup_{(I^{\mathfrak{i}} \triangleright O^{\mathfrak{i}}) \in \iota(\mathfrak{s})} O^{\mathfrak{i}} \right) \quad (2)$$

where $I^{\mathfrak{s}} \subseteq I$ and $O^{\mathfrak{s}} \subseteq O$.
As a result, (1) and (2) yield the following inclusion relations for all $\mathfrak{s} \in \texttt{SERVICE}, \mathfrak{i} \in \iota(\mathfrak{s}), \mathfrak{e} \in \epsilon(\mathfrak{i})$:

$$I^{\mathfrak{e}} \subseteq I^{\mathfrak{i}} \subseteq I^{\mathfrak{s}} \subseteq I \text{ and } O^{\mathfrak{e}} \subseteq O^{\mathfrak{i}} \subseteq O^{\mathfrak{s}} \subseteq O \quad (3)$$

Thus, service interface elements can be viewed as projections on the service interface as depicted in Fig. 1.

The second important concept, we examine in the following, is the concept of *service groups*. Service groups represent compositions of services with certain properties. First, we formalise the concept of *property*. Then, we introduce the concept of *abstract service group* and its two specialisations *atomic* and *composite groups*.

A property is described through a property identifier and corresponding property values. Let IDENTIFIER denote the set of all property identifiers and VALUE the set of all property values. Property identifiers and property values are arbitrary datasets. Not every property value $v \in \texttt{VALUE}$ can be associated with every property identifier $id \in \texttt{IDENTIFIER}$. We denote by $\texttt{TARGET}(id) \subseteq \texttt{VALUE}$ the subset of all property values that can be assigned to the property identifier id. Thus, the set of property values VALUE can be represented as a union of subsets $\texttt{TARGET}(id)$ for all $id \in \texttt{IDENTIFIER}$: $\texttt{VALUE} = \cup_{id \in \texttt{IDENTIFIER}} \texttt{TARGET}(id)$.

Let GROUP denote the set of all abstract service groups. An *abstract group* is a generalisation of *atomic* and *composite groups*. Atomic and composite groups differ in their containment relationship. Atomic groups consist of services, while composite groups can contain only abstract subgroups.

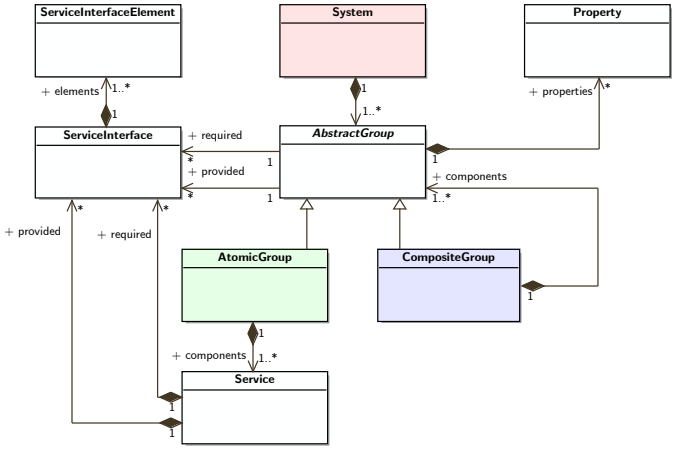

Fig. 2. Class diagram of the conceptual model.

Definition 4 (Atomic Group): An *atomic group* $a \in \texttt{GROUP}$ is defined by the pair (S, π), where $S \subseteq \texttt{SERVICE}$ is the set of contained services, and $\pi\colon \texttt{IDENTIFIER} \nrightarrow \mathcal{P}(\texttt{VALUE})$ is a partial function that associates a set of values to a property identifier. For the property function π holds the following restriction: $\forall id \in \texttt{IDENTIFIER} : \pi(id) \subseteq \texttt{TARGET}(id)$.

Definition 5 (Composite Group): A *composite group* $g \in \texttt{GROUP}$ is defined by the pair (G, π), where $G \subseteq \texttt{GROUP}$ is the set of contained abstract subgroups and π is a partial function defined as in Definition 4.

The property function π is used to specify the properties to be applied on services of an atomic group resp. subgroups of a composite group at deployment time and run-time. Next, we present examples of defining group properties. Fig. 2 depicts the class diagram of our conceptual model for service grouping.

B. Property Specification

Properties are specified on the level of abstract groups and apply to their components, which can be services or subgroups. Let PROPERTY denote the set of all properties. The property $p \in \texttt{PROPERTY}$ is specified by an identifier id and a set of corresponding values ϑ as follows: $p = (id, \vartheta)$, where $id \in \texttt{IDENTIFIER}$ and $\vartheta = \{v_i \mid i \in \mathbb{N}_+, v_i \in \texttt{VALUE}\} = \texttt{TARGET}(id)$. An instantiation of property p is given by the property function π. Each abstract group is described by a property function π, which specifies the set of properties that apply to the abstract group. If no properties are specified for a certain abstract group, then this abstract group has no property function π associated with it. As an example, we specify the following three properties: (i) visibility of service interface elements, (ii) qualified provision of service interfaces, and (iii) deployment of services, and demonstrate them on the sample system s outlined in Fig. 3. The system s is composed of two composite groups g_1 and g_2, four atomic groups a_1, a_2, a_3, and a_4, as well as six services $\mathfrak{s}_1, \mathfrak{s}_2, \mathfrak{s}_3, \mathfrak{s}_4, \mathfrak{s}_5$, and $\mathfrak{s}_6$. The system has two abstraction levels, the level of composite groups and that of atomic groups. Due to the recursive definition of

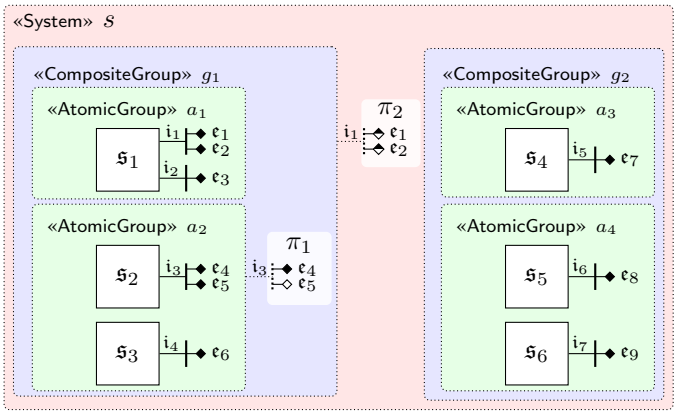

Fig. 3. System properties example. π_1 is applied to the "delegated" interface $\mathfrak{i}_3$; π_2 is applied to the "delegated" interface $\mathfrak{i}_1$

composite groups, the number of abstraction levels is not restricted. Note, that the lowest level in the hierarchy will only consist of atomic groups. As a rule, the number of abstraction levels depends, on the one side, on the necessity to structure services according to well-defined criteria, and, on the other side, on deployment and run-time requirements of the system.

In Section III-C we examine several criteria for structuring services in a service-oriented software architecture.

1) Visibility of Service Interface Elements: By default, a service interface element can be accessed by any service in any abstract group. By this property we define restrictions in the service interface *provision/consumption relation* between services. We set $p_1 = (private, \vartheta_1) \in \texttt{PROPERTY}$, where $\vartheta_1 = \{\text{name}(\mathfrak{s}).\text{name}(\mathfrak{i}).\text{name}(\mathfrak{e}) \mid \mathfrak{s} \in \texttt{SERVICE}, \mathfrak{i} \in \iota(\mathfrak{s}), \mathfrak{e} \in \epsilon(\mathfrak{i})\}$ and name is a function that returns the name of a system component, i. e., abstract group, service, service interface, or service interface element.

In system s, the service interface element $\mathfrak{e}_5$ of interface $\mathfrak{i}_3$ of service $\mathfrak{s}_2$ can only be accessed by services within the same atomic group a_2. An invocation of the service interface element $\mathfrak{e}_5$ by services outside of a_2 is not allowed. We define the property function π_1 for the atomic group a_2 and express the *access restriction* on service interface element $\mathfrak{e}_5$ by setting $\pi_1(private) = \{\mathfrak{s}_2.\mathfrak{i}_3.\mathfrak{e}_5\}$. In Fig. 3 this is expressed by a white diamond in front of the interface element $\mathfrak{e}_5$ of the interface $\mathfrak{i}_3$ at the boundary of the atomic group a_2. The dashed line rendering of the interface $\mathfrak{i}_3$ at the boundary of the atomic group a_2 expresses the "delegate" relationship between the service $\mathfrak{s}_2$ and the atomic group a_2, i. e., the service $\mathfrak{s}_2$ *delegates* the interface $\mathfrak{i}_3$ to the boundary of the atomic group a_2.

2) Qualified Provision of Service Interfaces: By default, after publishing a service interface is accessible by any other service in the system. Sometimes we want to restrict the access to a service interface to selected groups. This type of service interface access restriction is called *qualified provision of service interfaces*. We define $p_2 = (providedTo, \vartheta_2) \in \texttt{PROPERTY}$, where $\vartheta_2 = \{\text{name}(\mathfrak{s}).\text{name}(\mathfrak{i}), \text{name}(g)) : \mathfrak{s} \in \texttt{SERVICE}, \mathfrak{i} \in \iota(\mathfrak{s}), g \in \texttt{GROUP}\}$.

In the example system s, we restrict the access to the service interface $\mathfrak{i}_1$ of service $\mathfrak{s}_1$ from outside the composite group g_1 only to the atomic group a_3. Within the composite group g_1, all services have unrestricted access to the interface $\mathfrak{i}_1$. We express the qualified provision of the service interface $\mathfrak{i}_1$ to the atomic group a_3 outside the composite group g_1 by defining the property function π_2 for the composite group g_1 and setting $\pi_2(providedTo) = \{(\mathfrak{s}_1.\mathfrak{i}_1, a_3)\}$. Fig. 3 illustrates this by half black, half white diamonds on the service interface $\mathfrak{i}_1$ *delegated* to the boundary of the composite group g_1.

3) Deployment of Services: Sometimes it is necessary to deploy particular services together due to special dependencies between them (e. g. versioning dependencies) or to deploy them on the same ECU or different ECUs due to special run-time and safety requirements (e. g. hard real-time requirements, redundancy, dislocality). These deployment constraints can be expressed by means of properties. By building appropriate service groups and defining appropriate property functions for these groups we can achieve effective control over the deployment process of services in a service-oriented architecture. We set $p_3 = (deploy, \vartheta_3) \in \texttt{PROPERTY}$, where $\vartheta_3 = \{\text{group}, \text{ecu}, \text{core}, \text{partition}, \text{network}, \text{dislocal}\}$.

By default, services contained in an abstract group are independently deployable. The property value "group" expresses that services contained in the abstract group must be deployed together. The property value "ecu" stands for the deployment of services contained in an abstract group on the same ECU. The "core" property indicates that services of an abstract group have to be run on the same CPU core, whereas the "partition" property indicates that services of an abstract group have to be mapped to the same partition of an ECU. The property value "network" specifies that the services of an abstract group have to be deployed within the same sub-network. Finally, the property value "dislocal" indicates that services contained in an abstract group need to be deployed onto different ECUs.

In the example system s services $\mathfrak{s}_2$ and $\mathfrak{s}_3$ contained in the atomic group a_2 must always be deployed together due to versioning dependencies. We set $\pi_1(deploy) = \{\text{group}\}$, where π_1 is the property function defined in Section III-B1 for the abstract group a_2.

In order to meet the deployment and run-time requirements of the system, we need appropriate deployment and run-time models that enforce the fulfilment of these requirements at deployment time resp. run-time. Thus, having a compelling development model that provides excellent support for documentation, modelling and design of deployment and run-time requirements is not enough.

We need effective ways to translate with minimal effort development models into deployment and run-time models, i. e., automatic generation of deployment descriptors from development model, automatic verification, and generation of a run-time model that fulfils all run-time requirements of the system under consideration. Moreover, we need to check the structural properties of the service architecture such (i) strictness of a layered architecture and (ii) fully specified entities, i. e., services contain service interfaces which

in turn are built up from interface elements statically. The automatic generation of deployment and run-time models from the development model is a topic of our future research, and is not further examined in this paper.

C. Service Classification Scheme

Next, we discuss several criteria for service grouping.

1) Hardware Dependency: Hardware dependency describes the extent to which a service depends on the underlying hardware environment. In automotive software systems, obvious candidates for such hardware-dependent services are services that provide the interface to sensors and actuators. The main reasons for classifying hardware-dependent and hardware-agnostic services into separate logical groups are reusability and changeability. Since hardware changes less frequently than software, hardware-dependent services are also expected to be more stable, thus being characterised through lower change rates and increased maintainability. Further, hardware-dependent services provide basic functionality that is not confined to a certain vehicle domain, thus being characterised through high reusability rates.

2) Runtime Environment: Runtime environment describes the technical environment in which services are deployed and run. Concerning the runtime environment services are divided into (i) onboard services, i. e., services that are running within the vehicle, and (ii) off-board services, i. e., services that are running outside the vehicle and are provided either by the OEM itself or by third-party service providers.

Access control and performance requirements build the main reasons for grouping onboard and off-board services into different logical groups. The notion of *access control* has different meanings. In this paper, we examine the concept of access control from the point of visibility of service interfaces to other services. E. g. the access to services which have access to and operate with sensitive data such as personal driver records needs to be restricted. Performance requirements also play an essential role in logically separating onboard from off-board services. As a rule, onboard services expose lower latency and higher availability as off-board services.

3) Service Binding: Service binding describes the way services are discovered and invoked at run-time. There are two types of service binding: (1) static and (2) dynamic service binding. Static service binding is defined at development time. Technically, static service bindings are resolved at compile-time. After the service deployment, invocation relations to services that are statically bound are fixed and cannot be changed anymore at run-time. Hot deployments for services that are statically bound are not possible. In contrast to static service binding, dynamic service binding is configured/parameterised at development time. At run-time, the service invocation of dynamically bound services is realised through the mechanism of *Service Discovery* which provides support for hot deployments of services. The main reasons for classifying statically and dynamically bound services into different logical groups are related to safety and real-time requirements. Statically bound services provide much better support for the enforcement of such requirements.

4) Functional Scope: According to [10], the overall automotive E/E system can be divided into four functional domains: (1) powertrain, (2) chassis & driver assistance, (3) interior, and (4) telematics. Efforts to increase reusability of services in SOAs, as well as the development of innovative functions like fully automated driving, lead to services that span the aforementioned "classical" functional domains and cannot be clearly assigned to one of them (cf. [3]). For this reason and for service classification into logical groups we add *cross-functional* domain to the four above-mentioned domains.

The main reason for classifying services by functional domains, and thus building five different logical groups within a logical abstraction level, is related to different requirements pertaining to these domains. Services with strict real-time and safety requirements are in the foreground within powertrain and chassis and driver assistance domains. In the interior domain, with the emergence of AI-based functions great importance is attached to memory storage requirements. With the deployment of services in the OEM backend, high performance communication and security requirements become more and more important. The telematics domain consists mainly of multimedia and infotainment applications which also pose performance requirements on communication networks.

5) Safety Criticality: Safety criticality describes the extent to which safety requirements for services exist. The *ISO 26262* [2] standard defines a risk classification scheme (Automotive Safety Integrity Level, ASIL) to identify the safety requirements for automotive functions. Performing a hazard and risk analysis yields the ASIL classification. In order to better control and ensure different safety requirements posed on services, classification of services into different logical groups according to ASILs is appropriate and reasonable.

6) Security Criticality: Security criticality describes the extent to which security-critical requirements for services exist. There are different levels for enforcing security requirements: (i) wire-level security, (ii) user authentication and authorisation, and (iii) service-level security. The *Web Services Security OASIS* standard [11] provides an example of service-level security. The standard specifies security measures in terms of security tokens combined with digital signatures to protect and authenticate SOAP messages. In order to better control and enforce security requirements posed on services, classification of services into different logical groups according to security levels can be appropriate and reasonable.

7) Real-Time Requirements: In automotive systems (and generally, in safety-critical cyber-physical systems) a guarantee of timely execution of many functions (e. g. powertrain) is crucial. The recent trend to highly automated and autonomous driving will further increase the number of automotive functions with strict real-time requirements. Thus, a classification of services into logical groups depending on the requirements for real-time behaviour can be appropriate and reasonable.

Criteria mentioned above for service grouping are not entirely independent of each other. The developer has the responsibility

TABLE I
REQUIREMENTS

Req.	Description
RE1	*VehicleConfigurationService* provides through the interface *VehicleIdNumberInterface* the car's chassis number, which is considered sensitive data and should not be accessible to services *CallListService* and *MessageService*.
RE2	*VehicleConfigurationService* provides through the interface *VehicleTypeInterface* sensitive information about car configuration. This information should be exposed only to services *HandsOffDetectionService* and *OBDHandsOffDetectionService*. All other services should not be able to access this information.
RE3	The service *HandsOffDetectionService* provides through the interface *HandsOffDetectionInterface* methods to activate and deactivate the Hands Off Detection feature. The method *deactivateHandsOffDetection* should be made accessible only to the onboard diagnostic service *OBDHandsOffDetectionService*. All other services should not be able to invoke this method.
RE4	The service *OBDHandsOffDetectionService* is a dedicated service for continuous state checking of the Hands Off Detection system. It must have the same version number as the service *HandsOffDetectionService*. For this reason, both services have to be deployed always together.

to perform service classification in a consistent and meaningful way. In our future work, we plan to develop a tool to support the developer in performing service classifications. One of the main tasks of the tool will be to check the consistency of the service grouping and to warn the developer about possible inconsistencies and contradictions, e. g. dynamic service binding vs real-time requirements. Further, the tool should be able to recognise and warn the developer about contradicting property definitions, e. g. assignment of both values ecu and dislocal to the property identifier *deploy*.

IV. CASE EXAMPLE

We apply the concepts of Section III to a realistic automotive example. The services, their interfaces, and *provision/consumption relations* are depicted in Fig. 4a.

VehicleConfigurationService provides two interfaces, *VehicleIdNumberInterface* and *VehicleTypeInterface*. All other services provide one interface. An exemplary illustration of service interfaces with provided service interface elements is given in Fig. 4b. Services *RemoteDoorUnlockService* and *VehicleFinderService* are off-board services that are deployed and run in the OEM backend. All other services are deployed onboard. The example system has to fulfil deployment and run-time requirements stated in Table I.

First, we structure services into abstract groups using the service classification scheme from Section III-C and above-mentioned requirements. Then, we examine how the deployment and run-time requirements can be specified using the property model described in Section III-B.

Since services *VehicleConfigurationService*, *HandsOffDetectionService*, and *OBDHandsOffDetectionService*, on the one side, and services *RemoteDoorUnlockService*, *VehicleFinderService*, *CallListService*, and *MessageService*, on the other side, belong to different functional domains, we apply the *Functional Scope* classification criterion (cf. Section III-C4) and group these services into two different composite groups, *Interior* and *Comfort*. As a result, the *Comfort* composite group contains two onboard services, *CallListService* and *MessageService*, and two off-board services, *RemoteDoorUnlockService* and *VehicleFinderService*. By applying the *Runtime Environment* classification criterion (cf. Section III-C2), we further group the services, contained in the *Comfort* group, into two different atomic subgroups, *VehicleRemoteControl* and *Office*.

The requirement **RE4** states that services *HandsOffDetectionService* and *OBDHandsOffDetectionService*, contained in the *Interior* group together with the service *VehicleConfigurationService*, have to be always deployed together. For this reason, we group services *HandsOffDetectionService* and *OBDHandsOffDetectionService* into a separate atomic subgroup, called *HandsOffDetection*. The remaining service *VehicleConfigurationService* is then placed in its own subgroup, called *VehicleData*. The other three requirements **RE1**, **RE2**, and **RE3** are not used for service grouping performed above. These requirements are used togther with the requirement **RE4** to specify property functions which are depicted in Table II.

The final result of the service grouping is illustrated in Fig. 5. As we see, *Interior* and *Comfort* groups are composite groups, since their components are atomic subgroups. *VehicleData*, *HandsOffDetection*, *VehicleRemoteControl*, and *Office* groups are atomic groups, since their components are services. This way, we created two levels of abstraction, the level of composite groups and the level of atomic groups.

The specification of deployment and run-time requirements formulated in Table I using the property model described in Section III-B is illustrated in Table II. For each requirement, we first indicate the abstract group of the property function to be specified and then declare the property function. Visibility restrictions specified by requirements **RE1** to **RE3** are visualised in Fig. 5. By default, if there are no visibility restrictions specified for a service group, then the interfaces and their elements are accessible by all other services of the system, independent of the group they are located in.

In the following, we want to demonstrate, using our simplified service-oriented system, how services, service groups and their properties can be specified in Franca+. E. g., we examine the property derived from **RE1** for the group *Interior*.

The Franca+ component model for the composite group *Interior* is illustrated in Example 2.

Example 2:

```
<** @description: composite group Interior **>
component Interior {
  provides VehicleNumberInterface as
    VehicleNumberInterface
  provides VehicleTypeInterface as
    VehicleTypeInterface
  // ...
  contains VehicleData as VehicleData
  // ...
  delegate provided VehicleNumberInterface to
    VehicleData.VehicleNumberInterface
  delegate provided VehicleTypeInterface to
    VehicleData.VehicleTypeInterface
  // ...
}
```

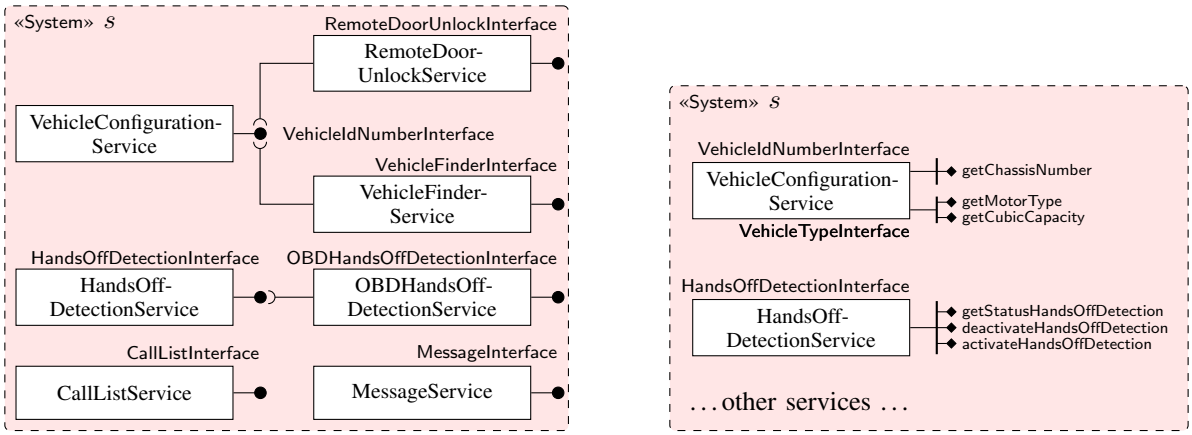

Fig. 4. (a) The system s consisting of seven services, and (b) for an exemplary selection their interface elements.

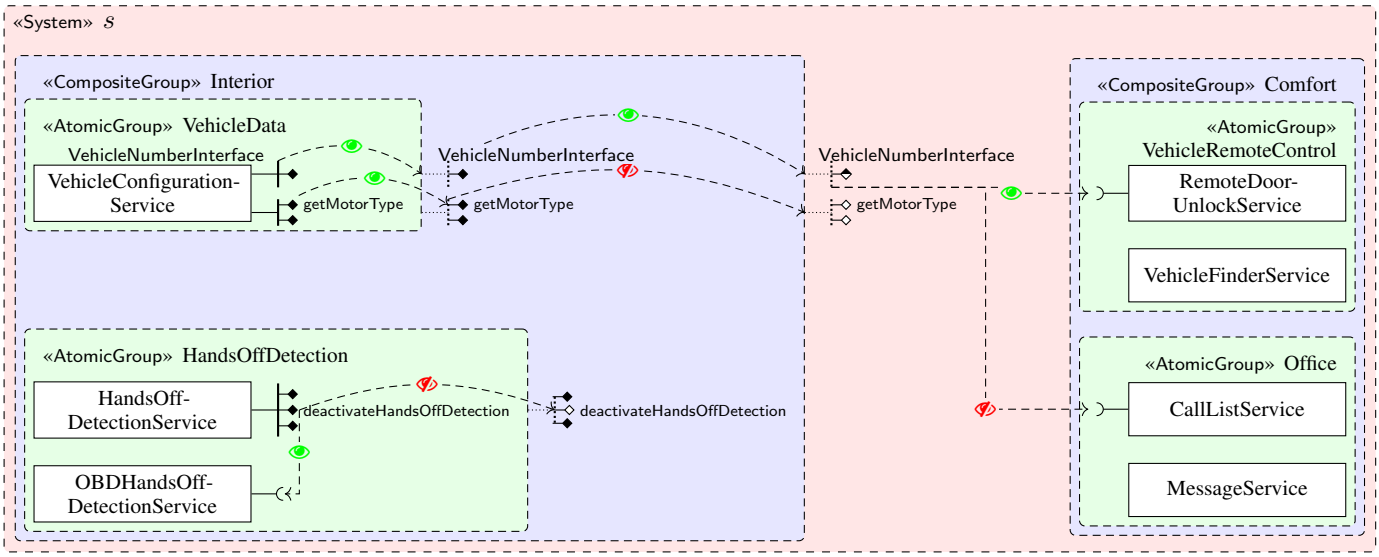

Fig. 5. Case Example: Visibility. For the sake of brevity, only relevant service interfaces are depicted.

TABLE II
PROPERTY FUNCTIONS

→Req.	Property function specification
RE1	$Interior = (\{VehicleData, HandsOffDetection\}, \pi_1)$, with $\pi_1(providedTo) = \{(VehicleConfigurationService.VehicleIdNumberInterface, VehicleRemoteControl)\}$
RE2	$Interior = (\{VehicleData, HandsOffDetection\}, \pi_1)$, with $\pi_1(private) = \{VehicleConfigurationService.VehicleTypeInterface.getMotorType, VehicleConfigurationService.VehicleTypeInterface.getCubicCapacity\}$
RE3	$HandsOffDetection = (\{HandsOffDetectionService, OBDHandsOffDetectionService\}, \pi_2)$, with $\pi_2(private) = \{HandsOffDetectionService.HandsOffDetectionInterface.deactivateHandsOffDetection\}$
RE4	$HandsOffDetection = (\{HandsOffDetectionService, OBDHandsOffDetectionService\}, \pi_2)$, with $\pi_2(deploy) = \{\text{group}\}$

```
<** @description: atomic group VehicleData **>
component VehicleData {
  provides VehicleNumberInterface as
    VehicleNumberInterface
  provides VehicleTypeInterface as
    VehicleTypeInterface
  contains VehicleConfigurationService as
    VehicleConfigurationService
  delegate provided VehicleNumberInterface to
    VehicleConfigurationService.VehicleNumberInterface
  delegate provided VehicleTypeInterface to
    VehicleConfigurationService.VehicleTypeInterface
}

<** @description: service Veh.Conf.Service **>
service component VehicleConfigurationService {
```

```
  provides VehicleNumberInterface
  provides VehicleTypeInterface
}
```

The composite group property can be specified by defining a custom tag such as `tag String @to`. The tag is then declared in the component model as shown in Example 3.

Example 3:

```
<** @description: composite group Interior **>
component Interior {
  <** @to VehicleRemoteControl **>
  provides VehicleNumberInterface as
    VehicleNumberInterface
  provides VehicleTypeInterface as
    VehicleTypeInterface
  // ...
}
```

Property specification using custom tags is error-prone, since we declare the components that can access the interface *VehicleNumberInterface* by their string representations in comments.

Another way would be to extend the keyword `provides` by an optional keyword `to` and to declare the components directly in the component model as illustrated in Example 4.

Example 4:

```
<** @description: composite group Interior **>
component Interior {
  provides VehicleNumberInterface to
    VehicleRemoteControl
  provides VehicleTypeInterface as
    VehicleTypeInterface
  // ...
}
```

This approach would imply an extension of Franca+.

V. Conclusion

In this paper, we extended the formal model of services defined in [4] by *service interface elements* and introduced the concepts of *service groups* and their *properties*. By defining an appropriate *property model* we created the conceptual foundation for defining any *properties* for *service groups* in a service-oriented architecture answering **RQ2**.

Further, we introduced an adequate *service classification scheme* (**RQ1**) for security- and safety-critical cyber-physical systems facilitating the derivation of appropriate *service groups*.

By structuring a service-oriented software system into service groups, we do not only create a good foundation for the efficient definition of properties to meet deployment and run-time system requirements, but also increase the overall maintainability of the software system.

We demonstrated concepts and formal models with a simplified but yet realistic automotive SOA. Moreover, we showed that the *property model* is well-suited for the definition of critical requirements.

Still, we have not examined all possible types of properties, and we have not evaluated the application of the proposed *service classification scheme* with complex automotive SOAs consisting of hundreds of services. Our strong expectation is that the *property model* remains feasible for other types of service group properties not examined in this paper and that the defined *service classification scheme* can be efficiently applied also to complex scenarios.

As future work, we plan to design and implement a modelling framework for service-oriented architectures with support for service grouping according to well-defined criteria as described by our *service classification scheme* (cf. Section III-C). The framework will also provide support for the definition of any properties (cf. Section III-B), since existing frameworks (e. g. Franca+) do not provide convenient and full support for any property specifications.

In this paper, we perform the service grouping using a well-defined service classification scheme. In addition, we plan in our future work to analyse how *service groups* can be automatically derived using software metrics (cf. [12]–[15]). Our work will also include an empirical assessment of the quality of the resulting *service groups* based on software metrics.

Acknowledgment

This work was supported by the BMW Group. We thank our project partners for putting our research into an innovative practical context.

References

[1] Society of Automotive Engineers, *Taxonomy and Definitions for Terms Related to On-road Motor Vehicle Automated Driving Systems, SAE Standard J3016*, 2014.

[2] ISO, "Road vehicles–Functional safety (ISO 26262)," 2011.

[3] S. Kugele, P. Obergfell, M. Broy, O. Creighton, M. Traub, and W. Hopfensitz, "On service-orientation for automotive software," in *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*. IEEE, 2017, pp. 193–202.

[4] M. Broy, I. H. Krüger, and M. Meisinger, "A formal model of services," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, Feb. 2007.

[5] GENIVI, "Component Definition Language Franca+," 2018. [Online]. Available: https://github.com/GENIVI/franca_plus

[6] OSGi Alliance, "OSGi Service Platform," 2015. [Online]. Available: https://www.osgi.org

[7] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.

[8] D. S. Dirk Krafzig, Karl Banke, *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall, 2004.

[9] N. M. Josuttis, *SOA in Practice*, ser. Theory in Practice. O'Reilly Media, Inc., 2007.

[10] K. Reif, *Bosch Autoelektrik und Autoelektronik*, sixth edition ed., ser. Bosch Fachinformation Automobil. Vieweg+Teubner Verlag, 2011.

[11] OASIS, "Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)," 2006. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

[12] M. Hirzalla, J. Cleland-Huang, and A. Arsanjani, "A metrics suite for evaluating flexibility and complexity in service oriented architectures," in *Service-Oriented Computing - ICSOC 2008 Workshops, ICSOC 2008 International Workshops, Sydney, Australia, December 1st, 2008, Revised Selected Papers*, ser. LNSC, vol. 5472. Springer, 2008, pp. 41–52.

[13] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari, "Coupling metrics for predicting maintainability in service-oriented designs," in *18th Australian Software Engineering Conference (ASWEC 2007), April 10-13, 2007, Melbourne, Australia*. IEEE Computer Society, 2007, pp. 329–340.

[14] M. Perepletchikov, C. Ryan, and K. Frampton, "Cohesion metrics for predicting maintainability of service-oriented software," in *Seventh International Conference on Quality Software (QSIC 2007), 11-12 October 2007, Portland, Oregon, USA*. IEEE Computer Society, 2007, pp. 328–335.

[15] R. Sindhgatta, B. Sengupta, and K. Ponnalagu, "Measuring the quality of service oriented design," in *Service-Oriented Computing, 7th International Joint Conference, ICSOC-ServiceWave 2009, Stockholm, Sweden, November 24-27, 2009.*, ser. LNCS, vol. 5900, 2009, pp. 485–499.