A Dynamic Service-Oriented Software Architecture for Highly Automated Vehicles

Conference Paper · October 2019

DOI: 10.1109/ITSC.2019.8916841

A Dynamic Service-Oriented Software Architecture for Highly Automated Vehicles

Alexandru Kampmann[1], Bassam Alrifaee[1], Markus Kohout[1], Andreas Wüstenberg[1], Timo Woopen[2]
Marcus Nolte[3], Lutz Eckstein[2] and Stefan Kowalewski[1]

Abstract— **We present an ecosystem comprised of multiple building blocks that are centered around a simple, pragmatic concept for service-oriented architectures (SOA). Due to rigid design-time integration, today's prevailing automotive electric, electronic and software architectures are often unsuitable for infield updates or system reconfigurations. As statically integrated architectures do not provide the flexibility required to keep up with shorter development and technology life cycles of connected and automated vehicles (AVs), service-oriented architectures are a promising way forward. We present a SOA concept that allows for dynamic, runtime integrated software architectures. Our implementation stack equally supports full-scale computers and resource-constrained platforms. We facilitate architecture specification through a web-based description tool, which follows a simple paradigm that has applications beyond software architectures. The evaluation provides benchmarks of our implementation on an automotive-grade embedded system and a trial login to our architecture specification tool.**

I. INTRODUCTION

Advanced Driver Assistance Systems (ADAS) are now mainstream functionalities available in majority of new vehicles [1]. Many ADAS functionalities are built on top of historically grown architectures that were not designed for these increasingly advanced functionalities. According to [2] and [3], these often safety-critical functions not only run through short technology life cycles, but also require stronger functional interconnection across domain boundaries than ever before. Fig. 1 depicts the relation of various automotive architectures in a simplified way. Inspired by [4], the system is decomposed in different architecture views, wherein each view is concerned with a specific system domain. The topmost view consists of various vehicle functions, which may range from driver assistance features, infotainment systems, all the way to low-level sensing and actuation. This view encapsulates the user-facing and non user-facing functions of the vehicle, while purposely abstracting from the actual implementation. Many elements of the functional architecture view are software driven, especially current innovation drivers in the infotainment and driver assistance domain. Second from above, the software view holds all elements that implement the functional components. The bottom view shows the Electronic Control Units (ECUs), that are the execution platform for the software components.

As pointed out in [2] and [5], the overall architecture can be characterized as function-oriented and ECU-centric, resulting in the depicted vertical structures: each function is implemented as one software component, which is executed on a dedicated ECU. The various ECUs, which are often purpose-built for cost minimization, communicate through a number of non-interoperable bus systems, such as Control Area Network (CAN), Local Interconnect Network (LIN) or FlexRay. In the course of the vehicle engineering process, a large number of subsystems are integrated by the original equipment manufacturer (OEM) into a functioning vehicle. This results in a rigid, design-time integrated architecture, where connections and interplay between software components and between ECUs are ultimately frozen in the course of the engineering process [2].

Components in design-time integrated software architectures become difficult to update or replace, as decisions taken during the system integration become inextricable parts of the overall architecture. This may include hard-coded sources for specific information, e.g. the identifier of the message carrying the vehicle speed and the specific hardware port from which that information is to be obtained. Updating a single component can require changes to all other parts dependent on it. Thus, adaptations to the architecture and any changes to the system integration beyond this point become rare and costly [6].

The inflexible nature of the design-time integrated architecture has not been a major concern, possibly even beneficial for OEM business models. Besides updates to the infotainment or navigation system, OEMs rarely add new functionalities to vehicles past start of production, with Tesla being a notable exception. Instead, customers have to purchase new vehicles to get the latest functionalities.

We argue that the ability to adapt software and functional architectures will be important for the success of emerging technologies such as connected and automated vehicles. The innovation cycles of these technologies tend to be short, which becomes evident in the rapid progress that has been made in the areas of image recognition [8], LIDAR processing [9], localization [10] and other perception and control systems in the recent years. In order to ensure that automated vehicles always operate using state-of-the-art technology, updates to the software and overall architecture will become

[1]Alexandru Kampmann, Bassam Alrifaee, Markus Kohout, Andreas Wüstenberg and Stefan Kowalewski are with the Chair for Embedded Software, RWTH Aachen University, 52074 Aachen, Germany {kampmann, alrifaee, kowalewski}@embedded.rwth-aachen.de

[2]Timo Woopen and Lutz Eckstein are with the Institute for Automotive Engineering, RWTH Aachen University, 52074 Aachen, Germany {timo.woopen, lutz.eckstein}@ika.rwth-aachen.de

[3]Marcus Nolte is with the Institute for Control Engineering, TU Braunschweig, 38106 Braunschweig, Germany {nolte}@ifr.ing.tu-bs.de

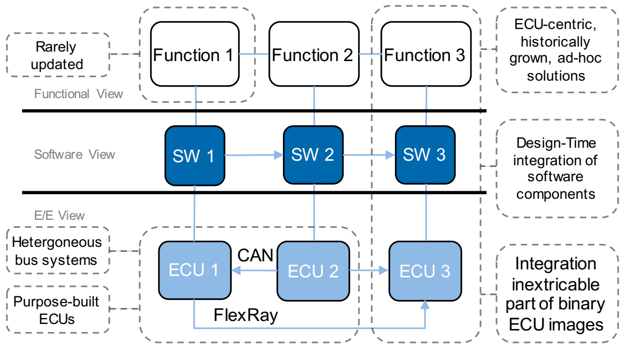

Fig. 1. Function-oriented, design-time integrated architectures that are widespread today, figure adapted in reference to [7].

necessary. In contrast to today's privately owned vehicles, business models behind self-driving car companies such as Waymo or Uber are often centered around taxi or shuttle services [11]. Consequentially, the majority of automated vehicles will potentially remain in the ownership of a commercial AV company. Due to new ownership models, OEMs and operators may start to continuously ship new updates and functions, instead of waiting for the next model year.

The rigid, design-time integrated software architectures of today's vehicles do not have the flexibility to cope with the challenges arising from short technology life cycles and new business models. In contrast, service-oriented architectures are based on runtime integrated services and allow for more flexible software architectures [12]. These architectures are a possible way forward in order to meet the challenges described above [13]. Our contribution is the ecosystem depicted in Fig. 2, which serves as the backbone for the UNICAR*agil* vehicles [14] and consists of the following building blocks:

- a concept for a service-oriented architecture that allows for dynamic runtime integration,
- an implementation stack that is suitable for high-end compute platforms and resource-constrained embedded platforms, and
- a modern, web-based architecture design tool with a new and simple paradigm, that allows to describe architectures beyond software.

The remainder of this paper is structured as follows. Section II provides an overview of related work on automotive service-orientation. The elements of our proposed ecosystem will be presented in Section III. The evaluation in Section IV provides a trial login to our architecture tool and benchmarks of our portable software stack running on an automotive-grade microprocessor.

II. Related Work

Formal models and applications of service-oriented architectures in the automotive context have been investigated in various efforts. In [15], the authors present αSOA, a rich formal model with semantic and syntactic specifications. The applicability of αSOA in the automotive-context has been investigated empirically by Kugele et al. in [5]. The same authors present a concept for elastic service provisioning in [16] with a somewhat similar concept for architecture control but with an additional, neural-network based fallback layer. In [17], Bocci et al. discuss service-oriented modeling of automotive systems using the SENSORIA Reference Modelling Language (SRML), that is also based upon mathematical semantics. Farcas et al. investigate rich services for system integration in avionics and automotive systems in [12]. The project *Controlling Concurrent Change* (CCC) aims at automated integration of software updates in mixed criticality automotive and space systems [18], basing a formal system model on multi-view descriptions [19] in a contract-based fashion. Malkis et al. approach services from a theoretical standpoint [20]. In contrast to the approaches above, we follow a radically simplified approach for service specifications. The AUTOSAR (AUTomotive Open System ARchitecture) is a wide-spread, standardized platform for automotive software development [21]. Service-orientation will become part of AUTOSAR with the introduction of the Adaptive Platform. Among other stacks, AUTOSAR Adaptive will support the Data Distribution Service (DDS) as the middleware for service implementation. DDS is a middleware specification standardized by the Object Management Group (OMG), that is designed for scalable, real-time communication in distributed systems [22]. In order to maintain compatibility, our software stack is based on a portable version of the Real-Time Publish Subscribe (RTPS) protocol, which is the underlying wire protocol for DDS.

III. Ecosystem for Service-Oriented Architectures

This section presents our ecosystem for a service-oriented software architecture in highly automated vehicles. We first present our concept for a service-oriented software architecture, which enables runtime integrated services and provides mechanisms for updates, replacements and repurposing of software components. Second, we present our portable software stack, that interconnects both powerful, full-fledged computers with resource-constrained platforms. Lastly, we introduce a web-based architecture description tool, that allows to specify various automotive architectures.

A. Concept for Service-Orientation

Inspired by [4] and [23] we use the framework depicted in Fig. 2 to describe our architecture. The topmost layer consists of the mode of operation of the vehicle. Each mode of operation consists of a different set of functional components and corresponds to a different intended vehicle behavior. In this simple example, the modes of operation consist of autonomous mode, hypothetical remote operation mode and the safe halt mode. The functional components are implemented as modular, runtime integrated services. We will now define the building blocks of our architecture.

***Definition** 1 (Service):* Let $\mathbb{S}$ be the set of fixed services, with $\mathbb{S} = \{S_1, \dots, S_\sigma\}$, $|\mathbb{S}| = \sigma \in \mathbb{N}^{>0}$. Every service $S_i = (R_i, G_i) \in \mathbb{S}$, $1 \le i \le \sigma$ is defined by a set of fixed requirements R_i and a set of fixed guarantees G_i. The

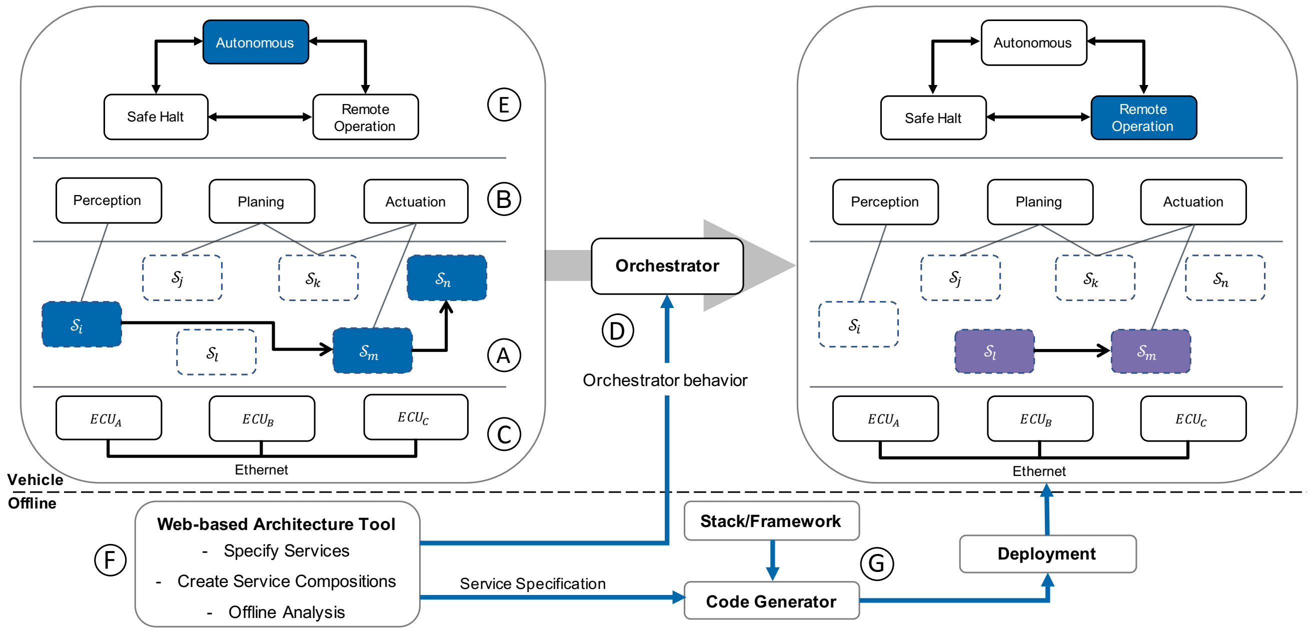

Fig. 2. Overview of our ecosystem centered around a service-oriented software architecture. **A, B** runtime integrated services make up multiple functional components. **C** Homogenous ECUs connected by Ethernet as execution platform. **D** Orchestrator controls the runtime integration. **E** multiple modes of operation that can consist of different service compositions. **F** Service and service composition specification tool. **G** A portable software stack supporting microprocessors as well as full-fledged POSIX-based platforms.

number of guarantees for service S_i is $|G_i| = \kappa(i) \in \mathbb{N}$. Analogous, the number of requirements for service S_i is $|R_i| = \mu(i) \in \mathbb{N}$. Either $G_i = \emptyset$ or $G_i = \{g_i^1, \dots, g_i^{\kappa(i)}\}$ and either $R_i = \emptyset$ or $R_i = \{r_i^1, \dots, r_i^{\mu(i)}\}$ for every $S_i \in \mathbb{S}$.

In contrast to prevalent automotive architectures, system integration information is decoupled from the implementation of our services, i.e., the service $S_j \in \mathbb{S}$ providing a guarantee to $S_i \in \mathbb{S}$ is not a hard-coded part of any S_i. Instead, services are integrated at runtime, thereby becoming interchangeable and reusable software components. This is facilitated by the set of requirements R_i and guarantees G_i that make up every service $S_i \in \mathbb{S}$, which acts like a machine-readable datasheet. Service's guarantees are typed functionalities that are offered by a particular service to other services. A service's requirements are also functionalities required to operate and are available as other service's guarantees.

For example, a trajectory tracking service may have a demand for the current vehicle velocity with a specific frequency. In order to match that particular requirement with a service capable of providing the required information, compatibility checking of requirements and guarantees occurs dynamically at runtime.

We refrain from naming requirements and guarantees inputs or outputs, which in our opinion implies the direction of information flow. Although this is true for publish-subscribe mechanism we currently employ, in the future the guarantee can be a function that is offered by a service, which is invokable by consuming services. In this case, the information would first flow into the invoked service, then back to the invoking service.

***Definition** 2 (Functionality Types):* We define the set of functionality types $\mathbb{T} = \{\tau_1, \dots, \tau_\pi\}$, $\pi \in \mathbb{N}^{>0}$. Every $\tau_i = (I_i, D_i, Q_i, P_i) \in \mathbb{T}$ has a fixed identifier I_i and consists of datatype definitions for Data D_i, Quality Q_i and Parameter P_i. For $1 \leq i \leq \pi$ with $\tau_i \in \mathbb{T}$, we denote the sets of all possible values assignable to D_i, Q_i and P_i as $\mathcal{V}_{\tau_i}^D$, $\mathcal{V}_{\tau_i}^Q$ and $\mathcal{V}_{\tau_i}^P$ respectively.

***Definition** 3 (Type Function):* Every element of non-empty requirements and guarantees are mapped to an element in the set of functionality types by $\mathcal{T}_G$ with $\mathcal{T}_G : \{(i,j) \mid S_i \in \mathbb{S}, g_i^j \in G_i\} \rightarrow \mathbb{T}$ and $\mathcal{T}_R$ with $\mathcal{T}_R : \{(i,j) \mid S_i \in \mathbb{S}, r_i^j \in R_i\} \rightarrow \mathbb{T}$.

***Definition** 4 (Compatibility):* We call a guarantee $g_i^m \in G_i$ of service $S_i \in \mathbb{S}$ *compatible* with a requirement $r_j^n \in R_j$ of service $S_j \in \mathbb{S}$ iff $\mathcal{T}_G(i,m) = \mathcal{T}_R(j,n)$.

We currently follow a data-centric approach for functionalities, where each functionality is essentially information exchanged through publish-subscribe mechanisms. Any $\tau_i \in \mathbb{T}$ only specifies the format of the data and the values are assigned by the providing service. The identifier I_i facilitates for comparison during runtime. D_i describes the format of the core content, e.g. the vehicle velocity. An essential aspect of our proposed architecture is explicit consideration of data quality. Therefore, the datatype Q_i is reserved for variables describing the quality of the data in D_i, e.g. measurement noises or uncertainties. This allows consuming services to adapt to changes in the data quality and may be fed into supervisory mechanisms in order to assess the overall system performance. Finally, P_i facilitates the runtime integration, as it allows services to specify any parametric aspects describ-

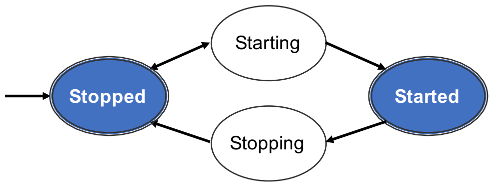

Fig. 3. Service life cycle model with states *stopped* and *started* and transients *starting* and *stopping*.

ing the data or quality portions. While the values assigned to data of type D_i and Q_i can vary during a services' lifetime, the parameters P_i are assigned constant values only once at the beginning of a services' lifetime.

For example, assume that τ_j represents the vehicle velocity, with $I_j = \texttt{egovelocity}$. D_j may consist of three floating point variables that represent the velocity in three dimensions. Component Q_j may consist of covariance matrices representing the current sensor noise in D_j. Lastly, P_j allows to convey fixed a-priori parameters, e.g. the maximum resolution or sensitivity of the sensor, mounting coordinates within the vehicle, sampling frequency, or noise offsets. Note that any two services S_m and S_n may provide guarantees of the same functionality type, e.g., multiple sensors providing the vehicle acceleration. They become differentiable by the values assigned to P_j.

We will now describe the runtime aspects of our architecture. Services follow a simple life cycle model displayed in Fig. 3. The model has states *stopped* and *started*, with transient states *starting* and *stopping*. A service starts in the *stopped* state and transitions to *starting*. In this state, the service can setup any resources, memory allocation or other initializations. After finishing the initialization, a services transitions to *started*. A service transitions to *stopped* by first passing through the *stopping*-state, where the service has the opportunity to release any allocated resources and prepare for suspension.

Definition *5 (Current Life Cycle State):* During runtime, the life cycle state of service $S_i \in \mathbb{S}$ at time $t \in \mathbb{R}^{\geq 0}$ is $\mathcal{M} : \mathbb{S} \times \mathbb{R}^{\geq 0} \rightarrow \{started, stopped\}$.

Definition *6 (Value Assignment):* At any time $t_k \in \mathbb{R}^{\geq 0}$ during runtime, every service $S_i \in \mathbb{S}$ with $\mathcal{M}(S_i, t_k) = started$ sporadically or periodically assigns values to the data and quality components of all it's guarantees $g_i^j \in G_i$, $j \leq \kappa(i)$. For every $S_i \in \mathbb{S}$ and $g_i^j \in G_i$ we denote the value assigned at time $t \in \mathbb{R}$ to the data and quality component of g_i^j as ${}^t\alpha_{ij}^D \in \mathcal{V}_{\mathcal{T}_G(i,j)}^D$ and ${}^t\alpha_{ij}^Q \in \mathcal{V}_{\mathcal{T}_G(i,j)}^Q$ respectively. The value of the parameter component is constant and denoted as $\alpha_{ij}^P \in \mathcal{V}_{\mathcal{T}_G(i,j)}^P$.

Services S_i provide their data-centric guarantees by assigning values to all $g_i^k \in G_i$. The assigned values are transported through the publish-subscribe mechanism facilitated by an appropriate middleware. In this context, we define the connection among services as follows.

Definition *7 (Service Connection):* We call a requirement $r_i^m \in R_i, m \leq \mu(i)$ of service S_i *connected* to a guarantee $g_j^n \in G_j, n \leq \kappa(j)$ of service S_j iff the underlying middleware is configured such that messages corresponding to g_j^n are received by S_i.

We are aware that this definition can only describe the observable behavior of a service. Automatons and mathematical logics can be used to further describe the internals of services on a semantic level. As this becomes highly application specific, we will cover this aspect in future work.

We will now provide details on the runtime integration of the simple model defined above. In other established application domains of SOAs, such as business and web applications, services may be allowed to initiate interactions with other services on their own. As we target safety-critical automated vehicles, we deem this flexibility harmful as unintended, potentially dangerous interactions may occur. Instead, we introduce the *orchestrator* as a designated component for controlling interactions among services at runtime.

Definition *8 (Orchestrator):* The orchestrator is a state transition system $\mathcal{O} = (\mathcal{Q}, \mathcal{E}, \mathcal{A}, \mathcal{R})$ consisting of states $\mathcal{Q}$, set of events $\mathcal{E}$, set of actions $\mathcal{A}$ and transition relation $\mathcal{R} \subseteq \mathcal{Q} \times \mathcal{E} \times \mathcal{G} \times \mathcal{A} \times \mathcal{Q}$ with the set of predicate logic guard conditions $\mathcal{G}$. In state $q \in \mathcal{Q}$, the orchestrator transitions to state q' and executes action a upon event $e \in \mathcal{E}$ if the guard condition g is true, i.e. $(q, e, a, g, q') \in \mathcal{R}$.

The set of events $\mathcal{E}$ consists of services failures, ECUs malfunctioning or non-safety related HMI-inputs and is application specific. The actions in $\mathcal{A}$ consists of `compatible()` for checking compatibility of requirements and guarantees, `setState()` for controlling life cycle model of each service, `connect()` for establishing connections among functionalities of two services, `disconnect()` for disconnecting all requirements of a service and lastly, `establishComposition()` for establish a service composition (cf. 9).

Definition *9 (Service Composition):* Let $\mathbb{C}$ be the set of all service compositions with $\mathbb{C} = \{C_1, \ldots, C_\lambda\}$, $\lambda \in \mathbb{N}^{>0}$. A service composition $C_i \in \mathbb{C}$ is a directed graph $C_i = (V_i, E_i)$. The vertices $V_i \subseteq \mathbb{S}$ are a subset of the set of services $\mathbb{S}$. The edges $e_j \in E_i$ with $j \in \mathbb{N}$ are connections between requirements and guarantees of services in V_i that are of identical functionality type: $E_i = \{(r_i^m, g_j^n) \mid S_i, S_j \in V_i \subseteq \mathbb{S}, r_i^m \in R_i, g_j^n \in G_j, \mathcal{T}_G(g_j^n) = \mathcal{T}_R(r_i^m)\}$.

Each composition $C_i \in \mathbb{C}$ defines a set of services that are in the state *started* and the connections among those services. For the sake of modularity, services are not aware of the current composition they are part of. Note that `establishComposition()` is syntactic sugar and can be defined as a sequence of the other operations.

The orchestrator provides a framework for rule-based system-integration and is designed offline. A simple example consisting of two states is provided in Fig. 4. In this example, we assume that the orchestrator is at state q when event `transitionToRemote` is issued. Hypothetically,

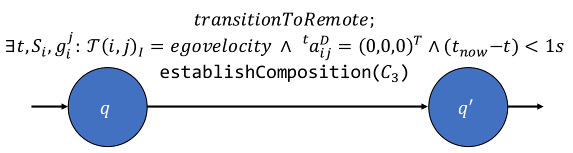
$transitionToRemote;$
$\exists t, S_i, g_i^j : \mathcal{T}(i,j)_I = egovelocity \wedge {}^{t}a_{ij}^{D} = (0,0,0)^T \wedge (t_{now} - t) < 1s$
establishComposition(C_3)
q
q'

Fig. 4. Orchestrator as state transition system reacting to events with guarded transitions and actions.

this event could correspond to the request of a control center for the vehicle to transition to a remote operation mode. The orchestrator only accepts this event under the condition that a service providing the functionality `egovelocity` has assigned $(0,0,0)^T$ to the corresponding guarantee within the last second. Consequentially, the transition will only be executed if the vehicle has been in standstill for at least one second. Note that the guard condition makes no reference to the actual service providing the respective guarantee, which facilitates the replacement and update of services. Finally, the executed action involves starting all services of a service composition C_3, which includes stopping non-required services and starting services that are necessary for remote operation.

In contrast to the vertical structures of function-oriented architectures in Fig. 1, one service can contribute to the implementation of multiple elements of the functional architecture. In the context of our project, the orchestrator allows to implement all vehicle variants with their respective modes of operation while using the same set of services. Furthermore, services become exchangeable, as long as a service is replaced with a service that provides the same formal guarantees. The specification of services and possible compositions is facilitated using the architecture description tool presented in Section III-C.

B. Implementation Stack

This section introduces the software stack that we use for implementing the service-oriented architecture concept described above. Our software stack is designed around a number of key requirements. As mentioned before, different isolated and often non-interoperable bus systems prevail in today's vehicle architectures. This poses a hard constraint on the possible interactions between services, as the communication paths cannot cross these hardware boundaries. Furthermore, the bandwidth required for transporting large amounts of data, e.g. point clouds or camera images across various ECUs cannot be handled by traditional automotive bus systems such as CAN, LIN or FlexRay [24]. In order to simplify the architecture and to maximize throughput and flexibility, the communication architecture in our project is purely Ethernet-based, from high-level perception down to the actuators. As many algorithms for automated driving require hardware with more computational power than prior driver assistance systems, it is obvious that full-fledged POSIX-based computers have to be supported by our software stack. Although low-level microcontrollers are not an appropriate platform for CPU-heavy computations, they can be suitable for safety-critical tasks, system monitoring or real-time demanding control functions. Therefore, our software stack is designed for portability allowing not only high-level computers but also microcontrollers to become first-class participants in our architecture. Finally, the stack has to support publish-subscribe messaging that our previously introduced concept relies upon.

Although the Robot Operating System (ROS) enjoys great popularity as a middleware in robotics, we argue that it is not suitable to fulfill the requirements above. First, ROS relies on Windows or Linux as the underlying operating systems. This makes ROS unsuitable for microcontrollers, which often run none or at most a slim, lightweight operating system. One workaround could be to use high-level computers to provide a bridge for microcontrollers into the ROS network. This comes with severe consequences for safety, as a failure in the bridging computer isolates the microcontroller from the rest of the system.

Instead, we base our software stack on the Real-Time Publish Subscribe (RTPS) protocol [22]. This protocol is the backbone for the Data Distribution Service (DDS), which is an API specification for machine-to-machine data exchange [25]. It has found application in various safety-critical systems such as air-traffic control, transportation systems, medical devices, aerospace and defense [26]. Besides DDS increasingly being used in automated vehicles, in [27] Kugele et al. also concluded the automotive applicability with experimental results. In contrast to ROS, RTPS does not rely on a designated entity for service discovery or binding. The protocol is based on the User Datagram Protocol (UDP) and has a range of configurable quality of service parameters, such as dependable or best-effort communication, heartbeats and fail-over mechanisms. Various commercial and open source implementations of DDS are available[1,2]. To the best of our knowledge, none of them is tailored for resource constrained microprocessors, as Windows or Linux is often assumed as the underlying operating system.

In order to be able to implement services on embedded platforms, we have developed an RTPS implementation that is based on the lightweight IP stack (lwIP) [28] and FreeRTOS as the operating system. The lwIP library is an open-source network stack that supports a broad range of protocols and that is designed for embedded systems with low resources and scarce computation power. Furthermore, lwIP has been ported to a broad range of embedded systems and ports for Linux and Windows are also available. Furthermore, our implementation has constant memory consumption and avoids memory allocation in critical code paths to make the implementation suitable for timing-sensitive applications. By obeying the standard defined by OMG, our implementation is interoperable to existing RTPS implementations. This is demonstrated in the Section IV-C, which also provides timing measurements for our implementation.

[1]http://www.eprosima.org
[2]http://www.opendds.com

C. Architecture Specification Tool

This section presents a web-based architecture description tool that is under continuous development. The tool is suitable for developing service specifications but is also able to capture architectures beyond software. Fig. 5 depicts a snapshot of our tool and a trial-login is provided in Section IV-D.

The tool is based upon a simple, expandable paradigm that is explained in the following. Our proposed solution is designed to avoid the often steep learning curves of existing architecture description tools. Components are organized in a variable amount of architecture views. Each view has a configurable set of *requirements* and *guarantees* that components in that view can offer or demand. In our framework, requirements are generalized inputs and can express more than just information flows, e.g. the need of components to be supplied with power, cooling requirements or a services' need for computation time. Guarantees are generalized outputs of a component, that are consumable by other components. This is analogous to a layered graph, where each layer corresponds to an architecture view and the nodes within each layer correspond to components of that view. An edge that connects nodes in this graph represents a requirement of a component that has been matched by the guarantee of another node. The abstraction of requirements and guarantees is applied to all architecture views. For example, we have configured our ECU architecture view, which also contains networking components of our prototype vehicles, such that each component can express demand for the following:

- **Ethernet connectivity requirement** - allows ECUs to express the need for Ethernet connection, and
- **Power supply requirement** - allows consumer to express need for power supply.

In turn, components in the E/E architecture view make offers of the following types:

- **Computation time guarantee** - allows computers to express the capability to provide computation time, and
- **Ethernet connectivity guarantee** - allows switches to express the gateway into vehicular Ethernet.

Obviously, this paradigm allows to capture services requirements and guarantees as defined in Section III-A. Components on the software architecture view can formulate requirements for computation time, which can then be matched with the analogous computation time guarantees of components in the ECU view. Furthermore, we created a view for the power supply system that holds components like batteries or DC/DC converters. These components have guarantees of **power supply** type, which is linked to all power requirements of components in other architecture views. For further refinement, each requirement or guarantee type can have arbitrary parameters attached to it. For example, parameters specifying the voltage and other power requirements are part of the parameters for the power supply requirement mentioned above. Service guarantees, besides the datatype definitions for D_i, Q_i and P_i, also hold

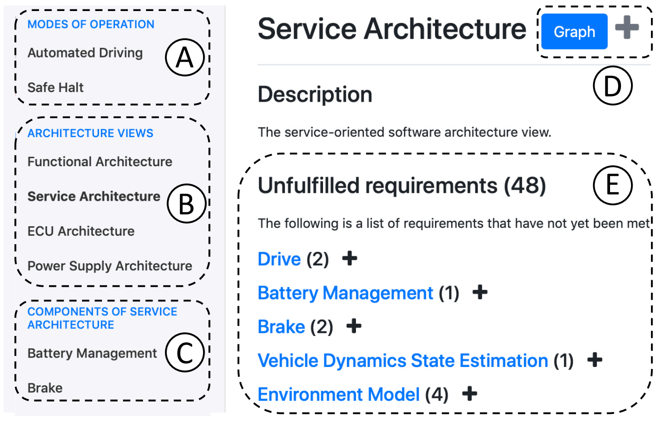
MODES OF OPERATION
Automated Driving
Safe Halt
A
ARCHITECTURE VIEWS
Functional Architecture
Service Architecture
ECU Architecture
Power Supply Architecture
B
COMPONENTS OF SERVICE ARCHITECTURE
Battery Management
Brake
C
Service Architecture
Graph
D
Description
The service-oriented software architecture view.
Unfulfilled requirements (48)
E
The following is a list of requirements that have not yet been met
Drive (2)
Battery Management (1)
Brake (2)
Vehicle Dynamics State Estimation (1)
Environment Model (4)

Fig. 5. Specification Tool Overview. **A** Modes of Operation with Service Compositions. **B** Architecture Views. **C** Components of Current Views. **D** Graph-based View. **E** Various Automatic Analyses.

information on the maximum data size and frequency. The tool allows for flexible configuration of arbitrary architecture views, requirement or guarantee types and their defining parameters.

The resulting connection graph among all components can be used for various automatic analyses. For example, using the connection among services, the specified frequency and data size attributes, we compute communication matrices that summarize the data flow between the ECUs automatically. This functionality is used to design and assess the sufficiency of communication paths in the UNICAR*agil* vehicles.

Due to the transparent and traceable connections among components, we can automatically determine each view's requirements that have not been matched by a respective guarantee, providing a quick overview of gaps or missing components in the overall system. Furthermore, the tool serves as the basis for cooperatively developing the functionality set $\mathbb{T}$ (cf. Definition 2) as the union of all service guarantees specified on the *service architecture* view. The connections among the services specified in the view provide the global constraints for of all *possible* service compositions in $\mathbb{C}$, i.e. no service composition can contain a connection among two services that are not connected in the service architecture view. Within these bounds, compositions can be modeled in the *mode of operation* section, yielding $C_i \in \mathbb{C}$ for the orchestrator. Additionally, specification serves as the basis for code generation of service interfaces. This approach enables runtime validation mechanisms that ensure that the actual software architecture matches the intended system design.

IV. DISCUSSION & EVALUATION

This section first discusses the presented work in the context of safety and automated driving. Afterwards, benchmarks of our implementation stack and a trial login to our architecture specification tool are provided.

A. Impact on Safety

We believe that the dynamic nature of our architecture does not conflict with safety goals. On the contrary, the orchestrator together with the standardized service interfaces allow for systematically handling safety-related events. Due to fact that the deterministic orchestrator behavior is defined offline, we can systematically enumerate and check every possible execution trace for potentially harmful configurations. At the same time, it is easy to verify if all considered failure events are handled. Results from the Hazard Analysis and Risk Assessment (HARA), which is mendatory for ISO 26262 certification, can serve as input for this aspect of the orchestrator design.

We disagree with the authors of [16], who claim that not all possible situations can be considered by a rule-based system. Instead, the authors introduce a neural-network based fallback layer for architecture control. We deem this approach unsuitable and dangerous for safety-critical automotive applications, as the verification of neural network is still an open research topic.

B. Application to Automated Driving

As motivated in Section I, automated vehicles will require updatable software architectures in order to ensure that safest and most advanced algorithms are on board. This is enabled through the modular and decoupled service interfaces of our architecture.

We also believe that our orchestrator can play an important role as a coordination mechanism for different automated driving tasks. For example, an automated delivery vehicle could require a different set of services to be active for package delivery, depending on a sequence of events that occur through the delivery process. Once the delivery has been carried out, a different set of services is required to drive the vehicle to the next customer. Making individual services responsible for coordination contradicts the modular run-time system integration approach, as individual services would require hard-coded information about other services in order to fulfill this coordination role.

C. Software Stack Portability and Performance

This section presents preliminary timing measurements of our software stack. In order to demonstrate the portability, we run the software stack on an Infineon Aurix microprocessor, which is a TriCore-based, ASIL-D certified safety-microcontroller. We benchmark our implementation by timing the message exchange between the Aurix controller and an off the shelf laptop running a Linux 4.18.16 kernel that has been outfitted with real-time capabilities through the `PREEMPT_RT`[1] patch. To demonstrate the interoperability, the laptop sends messages using the eProsima RTPS implementation to the microprocessor, which are processed by our implementation and immediately returned back to the laptop. The respective RTPS readers and writers have been configured for best-effort communication. We measure the round trip time (RTT) of this communication, which we define as the time between the laptop sending a message and receiving the response from the microprocessor. Both platforms are connected directly through Ethernet without a switch or router in-between. The experiment is repeated 10000 times for different message sizes and the results are presented in Table I. Assuming that the end-to-end delay

TABLE I
ROUND TRIP TIMES FOR VARRYING PACKET SIZES IN MICROSECONDS

Bytes	**Samples**	**Min** [μs]	**50%** [μs]	**99%** [μs]	**Max** [μs]
32	10000	460	634	944	1207
64	10000	477	732	914	1500
128	10000	481	735	933	1516
256	10000	569	793	999	1351
512	10000	626	891	1053	1535

for one direction of the communication is roughly half of the numbers reported above, our maximum latency lies well below the strictest end-to-end delay of 2.5 ms demanded for vehicular real-time control data according to research published in [29]. Nevertheless, we plan to reduce the somewhat large gap between the median and the maximum RTT through further optimizations.

D. Architecture Tool

The current version of our architecture specification tool can be accessed under **demo.architecture.unicaragil.embedded.rwth-aachen.de** with login E-Mail **demo@itsc2019.org** and password **demo**. We have populated four different architecture views with dummy components, that demonstrate the paradigm behind the typed architecture views. Various automatic analyses for different architecture views are provided. For example, an overview of all requirements, that are not linked to any guarantee are presented. For the ECU view, a communication matrix is automatically computed based on the connections among services and the use of the computation power guarantee of the components of the ECU view. A graph-based visualization of all components is provided for each architecture view. The tool facilitates the architecture description in UNICAR*agil* and its functionalities have helped us to discover gaps in the system design already early on.

V. FUTURE WORK

We regard the presented work as the foundation for automotive service-oriented architectures that will be further extended in the future. Already with the simple semantics, we were able to quickly and systematically define functional and complex service architectures for our prototype vehicles. Nevertheless, we plan to carefully add more precise semantics to the service specification, while still preserving the lean approach taken so far. We plan to add building blocks for safety measures to the architecture, such as redundancy, automatic failover modules with frequency monitoring that can be attached to any existing software component without having to modify code. Furthermore, we will present more details of our RTPS implementation and plan to demonstrate the portability to other common automotive microprocessors.

[1]https://wiki.linuxfoundation.org/realtime/start

The architecture specification tool has already facilitated the development of prototype vehicles in our project and we plan to add further functionalities. Among others, we plan extensions that enable timing analysis and the ability to systematically track changes to the views and components.

ACKNOWLEDGMENT

This research is accomplished within the project "UNICAR*agil*" (FKZ EM2ADIS002). We acknowledge the financial support for the project by the Federal Ministry of Education and Research of Germany (BMBF).

REFERENCES

[1] R. Schram, A. Williams, M. van Ratingen, S. Ryrberg, and R. Sferco, "Euro NCAP'S First Step to Assess Autonomous Emergency Braking (AEB) for Vulnerable Road Users," in *Proceedings of 24th Enhanced Safety of Vehicles (ESV) Conference*, 2015.

[2] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann, "Engineering Automotive Software," *Proceedings of the IEEE*, vol. 95, pp. 356–373, Feb 2007.

[3] C. Berger, "From Autonomous Vehicles to Safer Cars: Selected Challenges for the Software Engineering," in *Computer Safety, Reliability, and Security* (F. Ortmeier and P. Daniel, eds.), (Berlin, Heidelberg), pp. 180–189, Springer Berlin Heidelberg, 2012.

[4] M. Broy, M. Gleirscher, S. Merenda, D. Wild, P. Kluge, and W. Krenzer, "Toward a holistic and standardized automotive architecture description," *Computer*, vol. 42, pp. 98–101, Dec 2009.

[5] 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)*, pp. 193–202, April 2017.

[6] A. Elfatatry, "Dealing with Change: Components Versus Services," *Commun. ACM*, vol. 50, pp. 35–39, Aug. 2007.

[7] J. Bach, S. Otten, and E. Sax, "A taxonomy and systematic approach for automotive system architectures. from functional chains to functional networks," in *Proceedings of the 3rd International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS 2017), Porto, P, 22.04. - 24.04.2017. Ed.: O. Guskhin*, pp. 90–101, Scitepress, Setúbal, P, 2017.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A Large-scale Hierarchical Image Database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255, Ieee, 2009.

[9] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep Learning on Point Sets for 3D Classification and Segmentation," *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, vol. 1, no. 2, p. 4, 2017.

[10] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[11] T. Litman, *Autonomous Vehicle Implementation Predictions*. Victoria Transport Policy Institute Victoria, Canada, 2017.

[12] C. Farcas, E. Farcas, I. H. Krueger, and M. Menarini, "Addressing the integration challenge for avionics and automotive systems—from components to rich services," *Proceedings of the IEEE*, vol. 98, pp. 562–583, April 2010.

[13] J. Küfen, J. Hudecek, and L. Eckstein, "Automotive Service Oriented System Architecture - A New Concept and its Possibilities for Future Vehicle Systems," *Automotive meets Electronics*, 2014.

[14] T. Woopen, B. Lampe, T. Böddeker, L. Eckstein, A. Kampmann, B. Alrifaee, S. Kowalewski, D. Moormann, T. Stolte, I. Jatzkowski, M. Maurer, M. Möstl, R. Ernst, S. Ackermann, C. Amersbach, S. Leinen, H. Winner, D. Püllen, S. Katzenbeisser, M. Becker, C. Stiller, K. Furmans, K. Bengler, F. Diermeyer, M. Lienkamp, D. Keilhoff, H.-C. Reuss, M. Buchholz, K. Dietmayer, H. Lategahn, N. Siepenkötter, M. Elbs, E. v. Hinüber, M. Dupuis, and C. Hecker, "UNICARagil - Disruptive Modular Architectures for Agile, Automated Vehicle Concepts," in *27th Aachen Colloquium*, (Aachen, Germany), Oct. 2018.

[15] M. Broy, I. H. Krüger, and M. Meisinger, "A formal model of services," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, Feb. 2007.

[16] S. Kugele, D. Hettler, and S. Shafaei, "Elastic service provision for intelligent vehicle functions," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pp. 3183–3190, IEEE, 2018.

[17] L. Bocchi, J. L. Fiadeiro, and A. Lopes, "Service-oriented modelling of automotive systems," in *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pp. 1059–1064, July 2008.

[18] J. Schlatow, M. Möstl, and R. Ernst, "An extensible autonomous reconfiguration framework for complex component-based embedded systems," in *12th International Conference on Autonomic Computing (ICAC 2015)*, (Grenoble, France), pp. 239–242, July 2015.

[19] J. Schlatow, M. Nolte, M. Möstl, I. Jatzkowski, R. Ernst, and M. Maurer, "Towards model-based integration of component-based automotive software systems," in *Annual Conference of the IEEE Industrial Electronics Society (IECON17)*, (Beijing, China), oct 2017.

[20] A. Malkis and D. Marmsoler, "A model of service-oriented architectures," in *2015 IX Brazilian Symposium on Components, Architectures and Reuse Software*, pp. 110–119, Sep. 2015.

[21] O. M. Group, "Autosar: Standardized software framework for intelligent mobility." `https://www.autosar.org`, 2019. [Online].

[22] Object Management Group, "DDS Interoperability Wire Protocol." `https://www.omg.org/spec/DDSI-RTPS/2.2/`, 2014. [Online].

[23] G. Bagschik, M. Nolte, S. Ernst, and M. Maurer, "A System's Perspective Towards an Architecture Framework for Safe Automated Vehicles," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pp. 2438–2445, Nov 2018.

[24] A. Camek, C. Buckl, P. S. Correia, and A. Knoll, "An Automotive Side-View System Based on Ethernet and IP," in *2012 26th International Conference on Advanced Information Networking and Applications Workshops*, pp. 238–243, March 2012.

[25] G. Pardo-Castellote, "OMG Data-Distribution Service: Architectural Overview," in *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pp. 200–206, IEEE, 2003.

[26] Object Management Group, "Who's Using DDS?." `https://www.omgwiki.org/dds/who-is-using-dds-2/`, 2019. [Online].

[27] S. Kugele, D. Hettler, and J. Peter, "Data-Centric Communication and Containerization for Future Automotive Software Architectures," in *2018 IEEE International Conference on Software Architecture (ICSA)*, pp. 65–6509, IEEE, 2018.

[28] A. Dunkels, "Design and Implementation of the lwIP TCP/IP stack," *Swedish Institute of Computer Science*, vol. 2, p. 77, 2001.

[29] R. Steffen, R. Bogenberger, J. Hillebrand, W. Hintermaier, A. Winckler, and M. Rahmani, "Design and Realization of an IP-based In-car Network Architecture," in *1st International ICST Symposium on Vehicular Computing Systems*, 2010.