2020-01-1364 Published 14 Apr 2020

Model-Based Design of Service-Oriented Architectures for Reliable Dynamic Reconfiguration

Florian Oszwald and Philipp Obergfell BMW Group

Bo Liu and Victor Pazmino Betancourt FZI Research Center for Information Technology

Juergen Becker Karlsruhe Institute of Technology

Citation: Oszwald, F., Obergfell, P., Liu, B., Pazmino Betancourt, V. et al., "Model-Based Design of Service-Oriented Architectures for Reliable Dynamic Reconfiguration," *SAE Int. J. Advances & Curr. Prac. in Mobility* 2(5):2938-2947, 2020, doi:10.4271/2020-01-1364.

This article is from WCX™ 2020 SAE World Congress Experience.

Abstract

Service-oriented architectures (SOAs) are well-established solutions in the IT industry. Their use in the automotive domain is still on the way. Up to now, the automotive domain has taken advantage of service-oriented architectures only in the area of infotainment and not for systems with hard real-time requirements. However, applying SOA to such systems has just started but is missing suitable design and verification methodologies. In this context, we target to include the notion of model-based design to address fail-operational systems. As a result, a model-based approach for the development of fail-operational systems based on dynamic reconfiguration using a service-oriented architecture is illustrated. For the evaluation, we consider an example function of an automatically controlled braking system and analyze the reconfiguration time when the function fails. The reconfiguration time, together with the worst-case execution time (WCET), was determined by a model-based calculation and by simulation. In summary, the proposed approach applied to dynamically reconfigurable systems can meet the design requirements of the ISO 26262.

Introduction

One of the significant challenges for the automotive industry is the realization of fail-operational functions for highly or fully automated driving vehicles. Furthermore, future automotive systems may not be limited to the physical boundaries of modern vehicles but are instead interacting beyond this point with the cloud. These kinds of systems may also make intensive use of dynamically reconfigurable systems, also in fail-operational context. With such systems, hardware cost-down, weight reduction, and maybe even limiting energy consumption can be achieved. But dynamically reconfigurable systems need to be validated and verified, especially in fail-operational context. Currently, there is a lack of suitable design and verification methodologies to proof a development according to the ISO 262626. For example, the fulfillment of timing constraints is one topic for which evidence has to be provided. In this paper, we investigate this research topic and provide a toolbox and a methodology for the model-based design of dynamically reconfigurable systems in a fail-operational context using service-oriented architectures.

At first, we describe the related work on service-oriented architectures, model-based design, and in the field of dynamic reconfiguration in a fail-operational context. In the next section, we describe a toolbox consisting of modular elements to derive dynamically reconfigurable systems in SOAs. We introduce basic components and illustrate design patterns. After that, evaluation criteria based on metrics are depicted. We pinpoint a design methodology and a systematic on how to proceed in the development of systems with reliable dynamic reconfiguration. Next, we continue with an evaluation, illustrating a simulative way and a static analysis for the assessment of timing properties. Finally, we provide some details on possible future work and complete with a conclusion. With all this, we developed a full concept on how to bring reliable dynamic reconfiguration into the automotive domain.

In the proposed approach, the following questions for the design of reliable dynamic reconfigurable systems based on SOA arise.

1. What are the atomized basic components?
2. What are the possible design patterns?
3. What are the evaluation criteria to design a system architecture?
4. What does the development methodology look like across different levels of abstraction?
5. How can timing constraints be evaluated?

The proposed methodology helps to bring reconfigurable systems into the automotive domain.

Related Work

There are no established and known approaches to handle the design and verification of reliable dynamic reconfiguration in automotive fail-operational systems. Nevertheless, some research work exists. In [1], a conceptual framework is provided. AI-controlled and automatically triggered measures, as well as rule-based strategies together with publish-subscribe middleware, are provided to respond to inconsistent vehicle environment conditions. The main difference to this paper is that we provide a model-based approach and explain the steps for dynamic reconfiguration on a system level that can be easily adapted during development in terms of software reuse and amount of effort. In the work presented in [2] the authors investigated the applicability of SOA for safety-critical embedded automotive software systems. The conclusion of this is that the first results are indeed suitable for automotive software engineering and can cope with complexity and safety requirements. The work done in that previous study is used in this paper and enhanced with the model-based approach and is then expanded to dynamically reconfigurable systems. Furthermore, we bridge the gap between the model-based design of SOAs and dynamical reconfiguration. The work described in [3] gives a first setup and a design procedure for dynamic reconfiguration in a fail-operational context.

Two main topics involved in this challenge is the model-based design of service-oriented architectures and middleware concepts. There exists work which is based on these two topics.

Model-based Service Oriented Architecture Design

There are already several studies on the model-based SOA design in the IT industry. In [4], a theory and a formal model of services for SOAs are introduced. A framework with an SOA layered architecture using model-based techniques is presented in [5]. There is also some research going on in the automotive domain. A comprehensive overview of different aspects, including SOAs and model-based design, is put together in [6]. Domain-specific modeling for automotive services is suggested in [7] to fulfill the specific requirements in the automotive domain. However, dynamic reconfiguration is yet not considered in this research.

Middleware Concepts

The technologies and middleware concepts are, for example, Scalable Service-Oriented Middleware over IP Protocol (SOME/IP) [8], AUTOSAR Adaptive [9], Data Distribution Service (DDS) [10], GENIVI [11] and Rich Services [12]. SOME/IP is an automotive middleware solution that was designed to be integrated into AUTOSAR. It supports a wide range of middleware features such as Serialization, Remote Procedure Call, Service Discovery, Publish/Subscribe, and Segmentation of UDP messages. The AUTOSAR Adaptive platform aims to support the dynamic deployment of customer applications. It provides an environment for applications that require high-end computing power while preserving features that originated in embedded systems like safety determinism and real-time capabilities. DDS is a Data-Centric Publish-Subscribe (DCPS) model for distributed application communication and integration. The specification enables the efficient delivery of information from information producers to matching consumers and targets real-time applications. The GENIVI Development platform is an Open Source project for automotive. Rich Services is an architectural pattern suitable for systematically designing and implementing cyber-physical systems. It specific places emphasis on dynamic change.

Elements of Dynamic Service-Oriented Systems

In this section, we provide a toolbox consisting of three major modular elements for developing a dynamically reconfigurable system in a fail-operational context. As the first modular element, we introduce and explain the basic components. The second modular element consists of different design patterns. The third modular element contains evaluation criteria to enable the system designer to choose the right pattern.

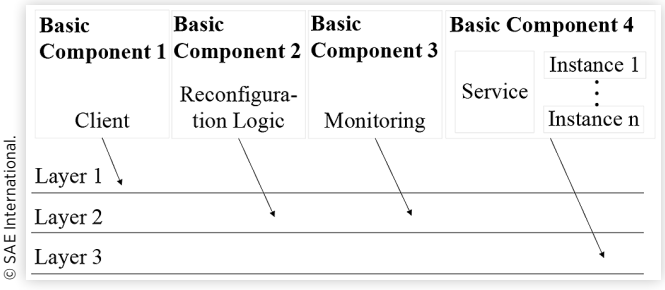
First Modular Element - The Basic Components

Here we give a breakdown list of the first modular element, and we introduce three different layers. To begin, we specify four basic components,

1. Client,
2. reconfiguration logic,
3. monitoring,
4. and service with n instances.

Fig. 1 shows the four basic components. The first component is the client. The client is the application itself, which makes use of different services, such as sensors, actuators, or trajectory planning. An example application is the steering function or the braking function, as described in [3]. Since we want to reconfigure the function dynamically, we need a second component, the reconfiguration logic. The reconfiguration logic executes the reconfiguration. The third component is the monitoring component, which detects failure

FIGURE 1 First Modular Element - The Basic Components.



events and is necessary as a trigger for the dynamic reconfiguration process. The fourth component is the provider of a service. This component may have several instances at least two.

Besides the four basic components, we also show three different layers and assign the basic components onto them, as given in Fig. 1. On layer 1, we assigned the client, on layer 2, the reconfiguration logic and the monitoring, and on layer 3, the services. With this concept, a separation of single software parts is supported, and therefore reuse of software parts is encouraged.

Second Modular Element - The Design Patterns

Next, we give an overview of design patterns for dynamically reconfigurable systems. We explain these patterns and provide different alternatives. For each, we depict the advantages and disadvantages. These design patterns are the following:

1. centralized architecture, central broker
2. decentralized architecture, each with monitoring
3. domain-oriented architecture, one broker per domain

We compile them by taking the four basic components from the previous section. Depending on how they are implemented, these three design patterns can be extracted. The right bookend is the centralized approach, while the left bookend is the decentralized approach. These are two design patterns, and the third one is a mixture of both.

Design Pattern of a Centralized Architecture

Overviews of centralized architectures in the automotive domain are provided in [13, 14, 15]. The design pattern of a centralized architecture is shown in Fig. 2. There are three services shown with n instances each. The logic for monitoring and reconfiguration is implemented in the central broker. The central broker negotiates the service subscriptions between the instances of the service providers and the clients. For example, there are several instances of service provider 1. In this case for trajectory planning. We assume two instances. One instance is running on one electrical control unit (ECU), and another one is running on another ECU. The client connects to the central broker and asks for the trajectory planning. The central broker transfers this request like a gateway and asks the different instances of the service provider to provide the trajecotry planning. Then the central broker establishes a connection to one of them. The connection is made, via a service protocol, which can be implemented with SOME/IP.

FIGURE 2 Design Pattern - Centralized Architecture.

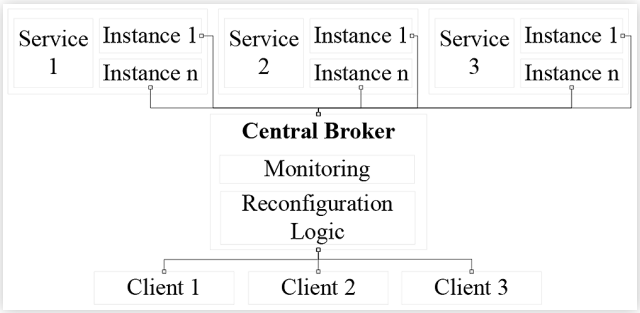
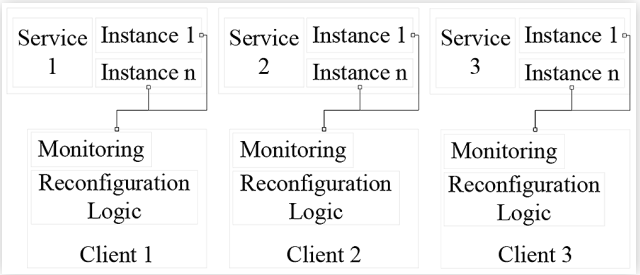


FIGURE 3 Design Pattern - Decentralized Architecture.



In case of a failure of an instance, the central broker is informed by the monitoring logic. Now the central broker asks the second instance of the service provider for the trajectory planning to provide the service to the client. The client does not notice that another instance now provides the trajectory planning.

Design Pattern of a Decentralized Architecture

An example of a decentralized architecture is described in [15]. The design pattern is depicted in Fig. 3. This pattern has no central broker. There are again two instances of service provider 1. The connection is also established via SOME/IP. But in this case, this is done by the client itself. Also, the monitoring and the reconfiguration logic are running inside the client because there is no central broker. The client now checks to see if the remaining instances of the service provider are available and subscribes to one. In each client, the monitoring and the reconfiguration logic needs to be implemented. Therefore, this design pattern causes a higher development effort in the client.

Design Pattern of a Domain Architecture

In [15] and [16], automotive domain architectures are illustrated. In these architectures, we usually find the following automotive domains:

1. Powertrain
2. Chassis
3. Body
4. Telematics

A domain architecture uses aspects of the centralized and decentralized architecture. In a domain architecture, there are domain control units (DCU) on which high-level functions are running, just as in the centralized architecture but only for a single domain. Other functions are running on ECUs, which are located below the DCU. To select the right design patterns, an evaluation is required.

Third Modular Element - Design Evaluation

In a centralized design pattern, there is only one reconfiguration logic. Whereas in a decentralized design pattern, the

TABLE 1 Evaluation Criteria for Design Patterns.

Evaluation Criteria	Centralized Architecture	Decentralized Architecture	Domain Architecture
reconfiguration logics	1	# of clients	# of domains
fail-operational	+	+	+
development effort	+	--	-
test effort	+	--	-
maintainability	+	--	-

number of these logics is corresponding to the number of clients. In a domain architecture, the number of reconfiguration logics is equal to the number of domains. Fail-operational requirements can be fulfilled with all three design patterns. The development effort and testing effort for the centralized design patterns need only be done once, while for the design patterns in a decentralized and a domain architecture, this metric is equal to the number of clients or domains. In terms of maintainability, there is a correlation with the number of reconfiguration logics within the architecture being used. The three earlier described design patterns are classified into these metrics, as can be seen in Table 1.

Model-Based Design Methodology for Dynamic Reconfiguration

For the introduced design patterns based on the idea of service-orientation, we now approach a methodology that targets the corresponding system design, runtime architecture, and End-to-End Latency Assessment.

Development Process

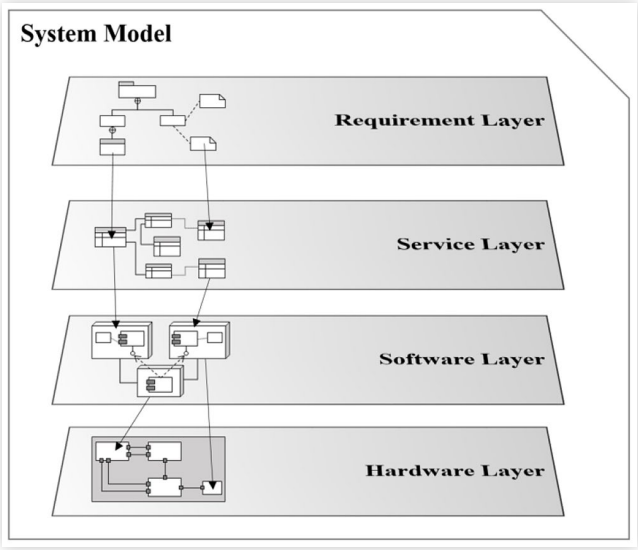
As shown in Fig. 4, a development process is proposed for designing and developing a reliable dynamically reconfigurable fail-operational system. Firstly, the system should be designed using the model-based approach. Relevant system elements should be modeled precisely for later verification. Using the modeled system as inputs, a first verification for the reconfiguration process could be performed. After that, another verification could be done using the simulative approach, ensuring the reliability of the dynamic reconfiguration process. The implementation should only be carried out if the verification results of both static analysis and simulative assessment meet the corresponding requirements.

Modeling Service-Oriented Systems

The modeling of an SOA-based system combining dynamic components is a challenging task. As shown in Fig. 5, there are different layers to be considered when designing a system model. The approach in this paper focuses on modeling the system in different levels of abstraction, which are most relevant for dynamic reconfiguration: service, software, hardware. In addition, there are other levels (such as the requirements, the functional level, etc.) that are not considered in this document. This approach allows the exact description of the several levels so that the reconfiguration process can be verified based on the modeled information.

FIGURE 4 Methodology – Process Steps.

FIGURE 5 Layered Architecture for System Modeling



Service Modeling Service modeling on the service layer depicts the abstract level within our system design approach. The primary task in this respect is the definition of service interfaces to describe how provided services can be accessed by clients and how the services are mapped to the software and hardware layer.

Service Interface Design: Service interfaces describe the exchange of data between services and clients using different interaction patterns. In Fig. 6, the publish-subscribe interaction paradigm and the request-response interaction paradigm are illustrated. Both describe bidirec-tional data channels between services and clients: For the publish-subscribe paradigm, a client subscribes to a service that was pub-lished beforehand and retrieves notification messages. These notifica-tion messages might be event-based, i.e., fired when a status changes or periodically. For the request-response

FIGURE 6 Publish-subscribe and request-response interaction paradigm.

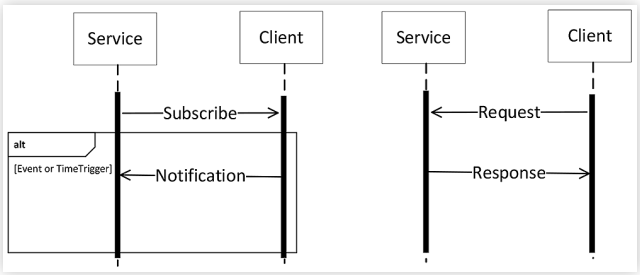
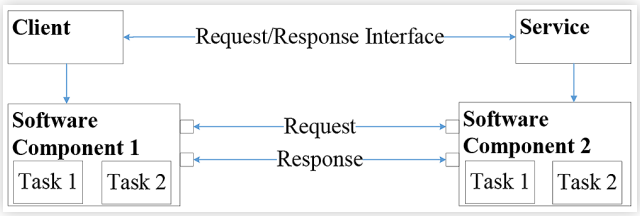


FIGURE 7 Relation between Services/Clients and Software Components.



paradigm, the client calls a service and retrieves an answer either synchronously or asynchro-nously because of being blocked by another client.

Software Modeling On the software layer, the instantiation of services and clients is implemented. Software components[1] give instances of services and clients. As an example, Fig. 7 represents one service/client dependency in the form of a request-response communication pattern that is instantiated by two software components.

In addition to software components, Fig. 7 depicts tasks within software components. A task of a software component reflects requirements on resource and real-time requirements and can be scheduled by computing resources. Necessary parameters like worst-case execution time (WCET) should be described as a part of a software component, which can be used for the verification process.

Hardware Modeling The used hardware elements for the deployment of software components and the relation between these elements should be described on the hardware layer. The deployment of software components within physical networks, i.e. the network topology, is achieved by the deployment of tasks from the software architecture model on the modeled hardware elements. In this respect, the selection of appropriate computing resources is derived by focusing on the resource and real-time requirements of tasks. In the case of having at least one deployment target for each task, the system architecture model is statically defined.

[1] For automotive software architectures, the most common approach is given by AUTOSAR, which also captures the idea of service-orientation within its newest release (cf. [9]).

Runtime Architecture

Based on the derived static system architecture model, we now describe behavioral aspects of the resulting runtime architecture. Since we focus on service-oriented architectures, runtime aspects are primarily captured by the idea of dynamic communication binding.

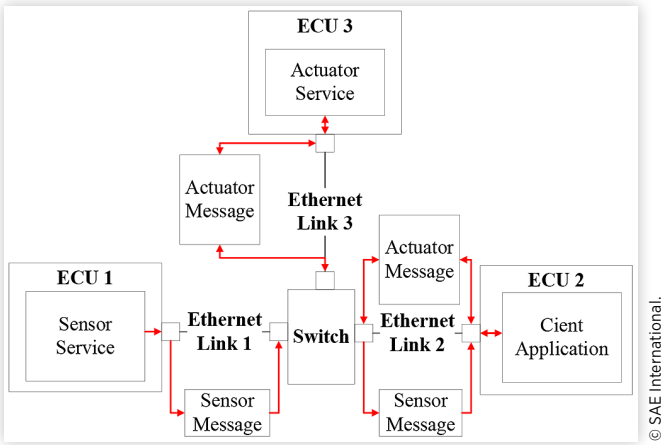
Service Discovery Service discovery implements the dynamic binding of communication using a protocol that resolves service/client dependencies at runtime. For this purpose, client instances receive for certain services offering messages. Afterward, the subscription in the form of a response message from the client to the service takes place. Fig. 8 depicts a simple illustration of the service discovery mechanism with one client and one service.

Service Usage After accomplishing the service discovery, clients can use their requested services. In this respect, the usage of a service is often embedded into a chain of client/service dependencies. A typical chain of client/service dependencies starts with the usage of a sensor service by a client application (e.g., a driver assistance application). After a processing step within the client application, subsequent usage of an actuator service (e.g., a motor for controlling the brake or the steering) is executed.

In Fig. 9, one example for a network of one client and two services is presented. The network depicts an Ethernet network that links three ECUs using a switch. The switch is equipped with ports that allow duplex transmission of data. The deployed software on the ECUs illustrates a sensor service, a client application, and an actuator service. An event-based

FIGURE 8 Service Discovery.

FIGURE 9 Analysis of System Design.



interaction pattern achieves the interaction between the sensor service and the client application. In contrast, for the actuator service and the client application, a request-response interaction pattern is applied.

End-to-End Latency Assessment Especially for safety-critical functions, the amount of time that passes while receiving data from a sensor service, processing it in a client application, and finally controlling the service interface of an actuator service needs to remain below a specific boundary. To assess this property, we establish an end-to-end latency assessment within our design approach. Two steps achieve the assessment:

1. Static Analysis: The posing of an end-to-end latency requirement on a network of interacting services and clients, calculating the reconfiguration time based on the static system models.
2. Simulative Assessment: The verification of the requirement by simulation-based verification techniques. Both approaches contribute to implementing recommendations from the ISO 26262 [17] (Part 4 of the system level and part 6 of the software level).

Static Analysis In the static analysis process, the system to be developed should be modeled using a model-based development tool. This system model is then used as input for the static analysis.

The execution times are determined from the software layer in combination with the selected hardware. The remaining communication components, such as bus systems and other network latencies, are also modeled in the hardware layer. This alone provides only the base timing information in the system. To calculate the end-to-end latency of the reconfiguration of a service, we also need to model the behavioral aspects. We propose to describe the reconfiguration process explicitly using a sequence diagram, where the actors are the software components, and the activities are the execution of functions. With this mapping, we calculate the reconfiguration time relying on the modeled execution times and the determined behavior of services and functions at reconfiguration.

Using this approach, we obtain an explicit description of the reconfiguration process. The added value is not only the ability to estimate and calculate the reconfiguration times, but also the explanation, justification, and traceability of the process. All of these points may be relevant in later verification and certification of such a system.

Simulative Assessment In the simulative assessment process, the modeled system should be simulated using a simulation tool so that values like the execution time of the simulated tasks could be derived for the verification.

Possible tools for doing the system design and assessment are already applied in practice. One option in that respect is given by the tool suite chronSIM/chronVAL (cf. [18]), which focuses on real-time properties of embedded system networks. As key strengths of the tool, simulation-based evaluation of timing properties can be achieved as well as an evaluation based on an analytic approach.

For the simulative assessment the following process steps are necessary. First, the system has to be modeled. Second, the software code has to be written. Next, the code is included in the simulation tool. After this, the timing properties are set. And finally, the simulation has to be run.

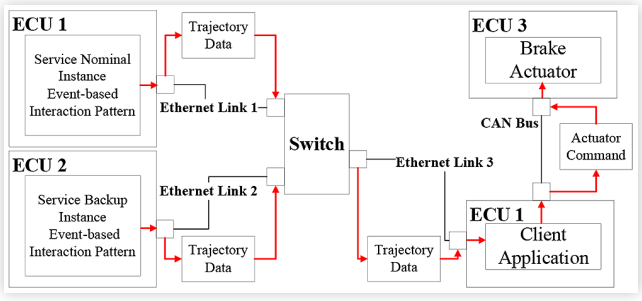
Implementation and Evaluation In the following, we apply our proposed design approach on an example vehicle function in the form of an automatically controlled brake.

Use Case for Dynamic Reconfiguration For our assessment, we emphasize the end-to-end latency assessment in case of reconfiguration. To do so, we assume that a running network of services and clients (i.e., each client is in use of its corresponding services) may degrade (i.e., clients are no longer able to use the services to which they were initially bound). In that case, the service discovery has to be executed again to identify backup service instances and consequently reconfigure the architecture (i.e., each client is again bound to its corresponding services. This use case uses the trajectory planning as the service in two instances, and the client is the braking function. The reconfiguration and the monitoring logic are implemented on the client.

System Architecture As an example architecture, we consider the topology in Fig. 10.

The topology has service-oriented functions within an Ethernet network as well as non-service-oriented functions that communicate via CAN bus. It has three ECUs. For the service-oriented contributions, one client application is introduced that is capable of using two instances of a trajectory planning service. Since there is no broker, this system implements a decentralized architecture pattern. The trajectory planning service supplies data about obstacles using an event-based interaction pattern. The client application evaluates the service information to control the brake actuator. The control messages may command the brake to do nothing (in case of having no information to brake) or to brake to avoid an accident. With respect to the software architecture model, a software component is required for each of the service instances as well as for the client application and the brake actuator, which are refined into individual tasks.

FIGURE 10 Evaluation Example.



Implementation for Simulation Our implementation is based on the realization of various types of tasks provided by the network participants in Fig. 10. In this respect, we distinguish between **Service Tasks** of the trajectory planning, **Client Tasks** of the braking function, and the **Actuator Task** of the brake actuator. For the service, the following tasks are introduced.

Service Tasks **Task 1: Offer Service Task:** The offer service task transmits periodically Ethernet messages to the client application.

Task 2: Interrupt Service Routine: The interrupt service routine handles incoming Ethernet messages that represent service subscriptions by the client application. The subscription is achieved by extracting the Media Access Control (MAC) address of the client.

Task 3: SendEventTask: After a client has subscribed to a service, the sendEventTask transmits periodically events that represent the current sensor value to the client application. For sending events, the Media Access Control (MAC) address of the client, which is extracted by the interrupt service routine, is used to transmit each 10 ms one event.

Client Tasks **Task 1: Interrupt Service Routine:** The interrupt service routine of the client application handles incoming Ethernet messages that represent service offers and triggers the subscription task.

Task 2: Subscription Task: The subscription task evaluates the received Ethernet messages from the interrupt service routine by filtering the destination of the service instances in the form of their MAC address. Afterward, this MAC address is inserted into a subscription message and sent to the corresponding service instance.

Task 3: Monitoring Task: The monitoring task evaluates whether a certain sensor service instance provides the events within the upraised period. In that respect, a service instance is considered to be degraded when less than seven events are received within an interval of 100 ms. As a result, the client must subscribe to the trajectory planning service's backup instance by re-executing the subscription task.

Task 4: Request Action Task: The request action task transmits periodically requests to the brake actuator task. These requests may command the brake actuator to brake or to do nothing.

Actuator Task The brake actuator is responsible for receiving and processing commands from the client application.

Static Modeling In the proposed development process, it is necessary to model the system to calculate the reconfiguration time and verify it in the first step. For this purpose, a custom model-based development tool derived from the project SysKit_HW [19] has been implemented using the eclipse modeling framework. With the custom-defined metamodels, different layers could be modeled according to the specific requirements. In this case, we need to model the hardware architecture, software architecture, service layer, and the reconfiguration scenario.

FIGURE 11 Hardware Diagram.

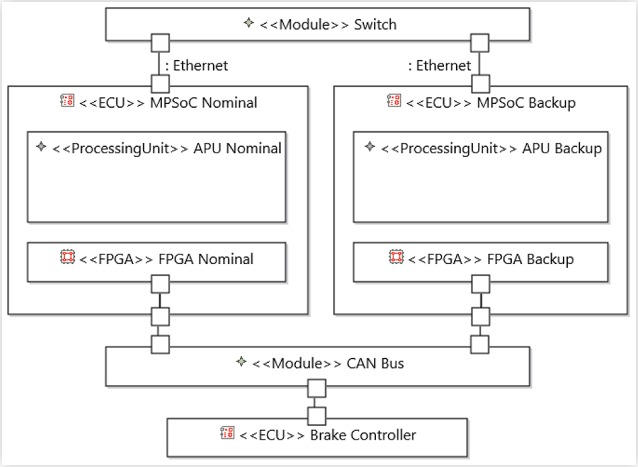


Fig. 11 shows the hardware diagram of the modeled example system. In this setup, there are two MPSoCs (representing an UltraScale+ Board) used for the trajectory planning task. This task is executed in the APU. The service consumer is also executed in the APU. One MPSoC is considered as the nominal instance, and the other one is considered as the backup instance. These two MPSoCs communicate through the switch using an ethernet connection. The FPGA part of the MPSoC is used to redirect information to the CAN Bus.

Fig. 12 shows the modeled software layer of the example system. The trajectory planning application is implemented on both MPSoCs, providing the same service (trajectory planning service). The client application is also implemented of both MPSoCs as the service consumer, which can use each trajectory planning application from the service providers. The SOME/IP application realizes the service discovery and the service provider functionality. To calculate the reconfiguration time as accurately as possible, the SOME / IP middleware function and the routing function of the switch should be considered and also modeled in the software layer. The next step is to determine the WCET of these applications that

FIGURE 12 Software Diagram.

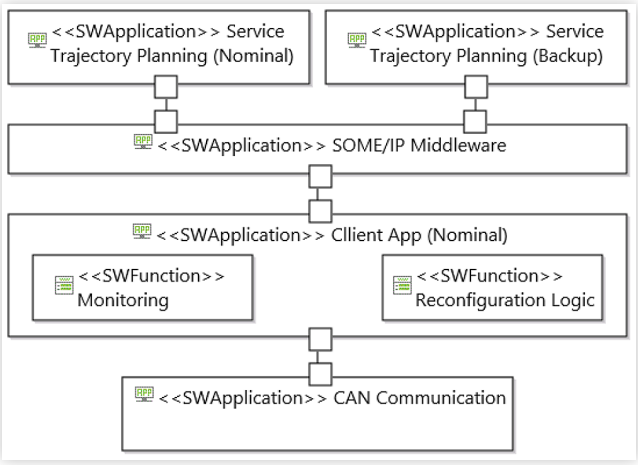
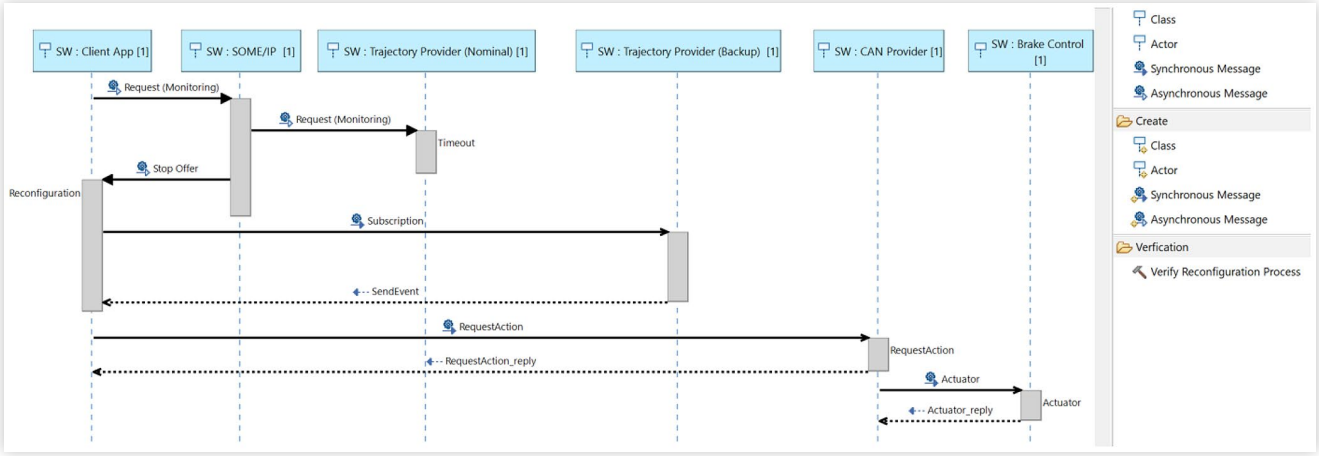


FIGURE 13 Sequence Diagram.



should be made by the designer as input to the reconfiguration time calculation step. For this, he can either use a separate tool to estimate timing or use empirically derived values.

Fig. 13 shows the reconfiguration process as a sequence diagram. First, the client application requests the service from the nominal instance with the monitoring task also running. If the nominal service is not available, the client application does not receive any response. After the timeout, the SOME/IP middleware notifies the client app, that the current service provider stops offering the service. After the reconfiguration, the client app subscribes to the backup trajectory planning provider. At this time, the client app requests the service from the backup instance. After the client gets and processes the result, it sends the commands to the CAN provider to control the brake. Using the custom implemented button "verify reconfiguration process", it is possible to calculate the reconfiguration process and verify if it meets the requirements. Only if the requirements are met, the next implementation and simulation steps should be executed.

End-to-End Latency Assessment For our example architecture, we pose an end-to-end latency requirement, which describes that the provision of trajectory planning, the processing of its data, and the transmission of the control message to the brake actuator is not allowed to exceed the value of 150 ms. This requirement has to be also met in case of reconfiguration where the message path involves an additional service discovery procedure. The message protocol in case of reconfiguration is illustrated in Fig. 11.

As a trigger for the reconfiguration, the nominal instance of the sensor service is considered to be degraded. The degradation happens on the client by the monitoring task that does not receive the requested amount of events within a given time interval of 100 ms. Then, a service discovery protocol is executed before finally sending a command message to the actuator.

End-to-End Latency Assessment Based on Simulation We simulated our example architecture within the tool chronSIM to derive the execution times of the introduced tasks. In this respect, we could derive worst-case execution times of single tasks for which 100 percent of the simulation runs met the posed end-to-end latency requirement. In total, we evaluated more than 6000 simulations runs.

TABLE 2 WCET for each task and total reconfiguration time

Task	Worst-Case Execution Time
Request (Monitoring)	40 ms
Reconfiguration (Interrupt Service Routine)	100 µs
Send Event Task	100 µs
Subscription Task	1 ms
Request Action Task	2 ms
Actuator Task	4.2 ms
Total Reconfiguration Time	**147.3 ms**

End-to-End Latency Assessment Based on a Static Model In the reconfiguration scenario described in Fig. 13, we use the "verify reconfiguration process" function to calculate the total time between the client application sends the request, and the brake control receives the message. The used WCET for each task, and the total reconfiguration time are shown in Table 2. The total time is less than the required worst-case time, so that the static verification for this reconfiguration process is successful. The next step of a simulative assessment could be carried out.

Summary and Conclusion

A current challenge for reliable dynamic systems in automotive is to achieve a design and verification process that ensures functionality in terms of safety-critical systems. One approach to realize such systems is using service-oriented architectures. Nevertheless, in today's electric/electronic (E/E) architectures, not every function is implemented in a service-oriented way. At the moment, no methods or tools exist that provide

an overall solution to develop and verify a reliable dynamic reconfiguration in a service-oriented system. Our approach shows a method to support the model-based design using different abstraction layers such as service, software, hardware, and also modeling the behavior of the reconfiguration scenario. Using this approach, we suggest evaluating the end-to-end latency of reconfiguration based on the system models and verifying it using a simulation-based approach. We present an implementation and evaluation of the advantages to calculate the reconfiguration time in a scenario of a braking function.

The main benefit for the system designer is the model-based support to have an explicit description of the reconfiguration scenario and to verify and validate the timing requirements. This support is based on the analysis of simulation results together with system models and the scenario description. Therefore, the approach is a step forward in allowing the design and verification of reliable dynamic reconfiguration, which can be relevant for validation and certification of such systems.

References

1. Kugele, S., Hettler, D., and Shafaei, S., "Elastic Service Provision for Intelligent Vehicle Functions," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC), 2018 IEEE International Conference on Intelligent Transportation Systems (ITSC)*, Maui, HI, USA, 04.11.2018 - 07.11.2018, IEEE, 2018, 3183-3190, ISBN 978-1-7281-0321-1.
2. Kugele, S., Obergfell, P., Broy, M., Creighton, O. et al., "On Service-Orientation for Automotive Software," in *2017 IEEE International Conference on Software Architecture (ICSA), 2017 IEEE International Conference on Software Architecture (ICSA)*, Gothenburg, Sweden, 03.04.2017 - 07.04.2017, IEEE, 2017, 193-202, ISBN 978-1-5090-5729-0.
3. Oszwald, F., Becker, J., Obergfell, P., and Traub, M., "Dynamic Reconfiguration for Real-Time Automotive Embedded Systems in Fail-Operational Context," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Vancouver, BC, Canada, Canada, 21-25 May 2018, 2018, 206-209, ISBN 978-1-5386-5555-9.
4. Broy, M., Krüger, I.H., and Meisinger, M., "A Formal Model of Services," *ACM Trans. Softw. Eng. Methodol* 16(1):5, 2007, doi:10.1145/1189748.1189753.
5. Cubo, J. and Pimentel, E., "DAMASCo: A Framework for the Automatic Composition of Component-Based and Service-Oriented Architectures," Crnkovic, I., Gruhn, V., and Book, M., in *Software Architecture*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, 388-404, ISBN 978-3-642-23798-0.
6. Broy, M., Krüger, I.H., and Meisinger, M., *Model-Driven Development of Reliable Automotive Services* (Berlin, Heidelberg: Springer Berlin Heidelberg, 2008), 4922, ISBN 978-3-540-70929-9.
7. Iwai, A., Oohashi, N., and Kelly, S., "Experiences with Automotive Service Modeling," in *Proceedings of the 10th Workshop on Domain-Specific Modeling, DSM' 10, ACM*, New York, NY, USA, 2010, 6, ISBN 978-1-4503-0549-5:1:1-1.
8. Völker, L., "Scalable Service-Oriented MiddlewarE over IP," http://some-ip.com/, February 1, 2019.
9. Fürst, S. and Bechter, M., "AUTOSAR for Connected and Autonomous Vehicles: The AUTOSAR Adaptive Platform," in *DSN-W Volume: 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks: 28 June-1 July 2016*, Toulouse, France: proceedings, Toulouse, France, IEEE, Piscataway, NJ, 2016, 215-217, ISBN 978-1-5090-3688-2.
10. The Object Management Group, "Data Distribution Service (DDS)," USA, https://www.omg.org/spec/DDS/1.4/, February 1, 2019.
11. GENIVI Alliance, "GENIVI Development Platform (GDP)," https://at.projects.genivi.org/wiki/pages/viewpage.action?pageId=11567210, February 1, 2019.
12. Krüger, I., "Rich Services - A SOA Pattern for Dynamic Change in Cyber-physical Systems," *IT- Information Technology* 55(1):10-18, 2013, doi:10.1524/itit.2013.0002.
13. Sommer, S., Camek, A., Becker, K., Buckl, C. et al., "RACE: A Centralized Platform Computer Based Architecture for Automotive Applications," in *Electric Vehicle Conference (IEVC), 2013 IEEE International, 2013 IEEE International Electric Vehicle Conference (IEVC)*, Santa Clara, CA, USA, IEEE, 2013, 1-6, ISBN 978-1-4799-1451-7.
14. Traub, M., Maier, A., and Barbehön, K.L., "Future Automotive Architecture and the Impact of IT Trends," *IEEE Softw.* 34(3):27-32, 2017, doi:10.1109/MS.2017.69.
15. Oertel, M., "Architectures of High Performance Computing," in *9th Vector Congress 2018*, Stuttgart, 2018.
16. Streichert, T. and Traub, M., "Elektrik/Elektronik-Architek\-turen im Kraftfahrzeug: Modellierung und Bewertung von Echtzeitsystemen," VDI-Buch, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, ISBN 3-642-25478-0.
17. International Organization for Standardization, "Road Vehicles -- Functional Safety," ISO 26262:2011, 1st ed., Rev. Nov. 2011.
18. INCHRON GmbH, INCHRON Tool-Suite (2.9.0.125 Build 20181108), INCHRON GmbH, http://www.inchron.com, 2018.
19. Liu, B., Glock, T., Sax, E., Pazmino Betancourt, V. et al., "Model-Driven Design of Tools for Multi-Domain Systems with Loosely Coupled Metamodels [in press]," in *Annual IEEE International Systems Conference*, Orlando, USA, 8-11 April 2019.

Contact Information

Florian Oszwald from BMW Group, Research, New Technologies, Innovations, Germany, can be contacted by florian.oszwald@bmw.de.

Definitions/Abbreviations

APU - Application Processing Unit

CAN - Controller Area Network

DCU - Domain Control Unit

E/E - Electrical/Electronic

ECU - Electrical Control Unit

MPSoC - Multi-Processor System-On-Chip

SOA - Service-Oriented Architecture

SOME/IP - Scalable Service-Oriented Middleware over Internet Protocol

WCET - worst-case execution time

e-ISSN 2641-9645