Service-Based Development of Embedded Systems*

L. Kof, B. Schätz, I. Thaler, and A. Wisspeintner

Institut für Informatik, Technische Universität München
Boltzmannstr.3, 85748 Garching, Germany

Abstract. Services as a basic notion are helpful in two respects: on the one hand, services are used to *structure the specification* of a system easing understand and reasoning about the system; on the other hand, services can also be used as *design principle of the implementation architecture*. This paper presents an approach for developing software systems using services as the central development concept. A concrete case study, namely the development of a control unit for an automotive seat adjustment system, clarifies the proposed service-based development process.

1 Introduction

Using services as basic concept eases the specification of reactive systems with a high degree of interaction with its environment as found, e.g., in the telecommunication or web services domain. This approach allows breaking up complex system functionality into smaller functional modules. This modularity supports a more manageable and comprehensible description of the functionality. This shift from a structural architecture (using components as the main building blocks) to a behavioral architecture (using services instead) is, e.g., applied in the domain of web services. There, systems do not consist of a fixed set of components, but are dynamically composed from services. However, using a service-based engineering process is not only useful in the field of dynamic networks, but also in domains with static structure supplying complex interacting functionalities. In the automotive domain, e.g., a large number of functionalities like ABS (anti-lock braking system) and ABC (active body control) are combined, interacting with each other and resulting in a complex overall behavior requiring a high level of safety. Here, too, services can help to structure the behavior of the complete system and make those interactions more explicit, thus leading to improved safety.

Our description formalism is based on the formalization of interatcion scenarions and is somehow related to [7]. In [7] Krueger defines a service as an interaction pattern. In this paper we go further and combine basic services, defined by their interaction patterns/scenarios, to more complicated services and to a complete application.

* This work was in part supported by the DFG (project InOpSys under reference number Br 887/16-1 and project Inkrea under reference number Br 887/14-1).

2 Services and Components

As defined in [10], a service is a clipping of the behavior of a component that is under-specified concerning internal structure and – making assumptions about the environment – supports the definition of partial behavior. However, there is also a strong relation between services and components: both an abstract component as well as an abstract service can be realized by a network of communication components or services, resp. And – most importantly – a service (or network of services) can be implemented by a component (or network of components) making it an offered service of the component.

2.1 Component-Based Architecture

Component-based architecture uses only the notion of a *component* as the central concept. As defined, e.g., in [2], a component

- has an interface (usually in form of typed ports)
- exhibits some behavior using this interface (usually in form of receiving and sending messages via these ports)
- can be hierarchically structured using networks of sub-components (usually communicating by channels linking their ports)

ROOM [11], SDL [3], or AutoFocus are typical examples for such a component-based approach (with some slight extensions, e.g., multiple instances of components supported by ROOM).

A (rudimentary) component-based development consists of three basic steps:

Component Specification: To specify a basic component, first the interface of a component is defined (e.g., in form of a ROOM capsule). Then, scenarios of interactions between the component and its environment are defined (e.g., in form of UML sequence diagrams).

Component Definition: Based on the component specification, a complete and consistent description of the behavior of the component is defined (e.g., in form of a state transition diagrams) fulfilling the specification (e.g., the sequence diagrams describe possible behavior of the state transition diagrams).

Component Combination: To build complex systems, components are combined by connecting their (shared) interfaces, linking an output port to a set of input ports. In a composed system, components run in parallel, exchanging messages. The composition of components is static: the set of communicating components does not change during the execution of a system. Due to these restrictions, composition is a total operation: combing components does always result in a component.

As a result of their constructive nature, component-based architectures are generally used in a bottom-up development process, focusing on easily reusable parts with limited degree of interaction; this is generally ensured by standard frameworks (e.g., layered architectures or multi-tier approaches) off restricted classes of components with highly-structured interfaces between the elements of the framework. Consequently, they are much less suited for the engineering of product lines, using large sets of interacting (and even conflicting) functionalities to construct variants of behavior.

2.2 Service-Based Architecture

In contrast of component-based architectures, service-based architectures makes use of two central concepts, the notion of a *service* and the notion of a *configuration*. As defined, e.g., in [8], their main characteristics are

Service: Besides the hierarchic structure, a service is rather similar to a component. A service
- has an interface (usually in form of typed ports).
- exhibits some behavior using this interface (usually in form of receiving and sending messages via these ports).
- has a set of configurations (with transitions between), describing which groups of sub-services are can be active simultaneously.

Configuration: Configurations describe the dynamic aspect of a service or a whole system: by changing from configuration to configuration, a service or a system can dynamically change its internal structure. Configurations
- describe sets of simultaneously active sub-services
- are linked by by transitions between them; transitions are triggered by inputs

Since service descriptions support the description to conflicting behavior, they are especially suited for the early phases of development: during the construction of a functional architecture of a system, a major step is the integration of functions, the detection of conflicts between functions, and conflict resolution strategies.[1] Here, services are especially suited to support those steps without enforcing design or implementation

Obviously, component-based architectures are a special case of service-based architectures (with only one configuration per service and only 1:n channels between input and output ports). Current approaches and tools support the definition of service-based architectures only to a very limited extent.

By allowing an arbitrary number of configurations (or services) in a component-like service-based architectures, we obtain a constructive subset of service-based architectures, similar to Statecharts [4]. Here, the sub-services of a configurations correspond to the composition of an AND state; the configurations of a service correspond to the compositions of an OR state.

In the remainder, we focus on the engineering aspect of applying services in the development process. Therefore, for the treatment of the issues of consistency of the combination of services and the completeness of a service, we refer to [10]; for the description of conflict-resolution steps for services and some resolving patterns we refer to [8].

[1] [8] treats those issues in more detail.

In the following, we focus on the construction of components from scenarios of interaction using services as intermediate construction between analysis and design. Therefore, in Section 3, we demonstrate

– how to identify basic services, starting from interaction descriptions in form of Sequence Diagrams,
– how to combine basic services to more complex services by configurations of exclusive services, and finally,
– how to combine services to complex components, using configurations of simultaneously active services and continuations of services.

Section 3 illustrates those steps using an example of [9].

2.3 Comparison

Focusing on interface behavior instead of system structure is the fundamental difference between a component-based and a service-based development process. Components and services own interfaces defined by the types of messages that flow via these interfaces (e.g., interfaces in Java or the interface of a hardware interfaces). Components are generally defined as reusable units of behavior and structure. Structure is defined by assigning subcomponents including their behavior to a component.

To reuse a component, it is structurally composed with other components, restricted only by structural compatibility conditions and supporting only a restricted form of behavioral combination (through communication). In contrast, a service represents a more abstract behavioral specification, its behavior depending on other services in the form of needed services. For reuse, services require a much stronger (behavioral) compatibility; in return, they also support a stronger form of (behavioral) combination.

As mentioned above, the main purpose of services is their ability to describe a coherent piece of behavior and thus support the stepwise integration of basic functionalities to high-level behavior. In contrast to a component-based architecture, a service-based architecture *explicitly models active and inactive services*, using configurations of (active) services. Component-based architectures usually use a static hierarchy of components: components are always active, their communication structure does not change. Furthermore, service-based architectures are more flexible concerning the compositions of services. Component-based architectures usually only support only the combination of

Furthermore, service-based architectures are more flexible concerning the compositions of behavior. Component-based architectures usually only support the combination of behavior with a rather restricted notion of (interface) compatibility, targeting rather a bottom-up design process. Service-based architectures also allow the combination of less restricted behavior, supporting the definition of a functional architecture.

Though targeting the early phases of the development process, a service-based architecture does even influence the later phases like design and implementation.

The service-based architecture:

– avoids the necessity for the introduction of auxiliary design elements, often enforced by component-based architectures (e.g., in form of managing components like arbiters).
– enables the explicit description of modes of operation in form of configurations of active services.
– supports efficient implementations by explicitly describing activation (and deactivation) by configurations of services, instead of using networks of continuously active components.

To illustrate the engineering relevance of a service-based architecture, in the following section we concentrate on the constructive development steps and use description techniques along the lines of ROOM and Statecharts.

3 Service-Based Development

3.1 Identifying Basic Services

This section presents a case study on service–based development of an automotive seat adjustment controller. The case-study has been carried out in a student project [12] using the service-based development process described in [6]. In the case study a part of a DaimlerChrysler automotive controller specification [5] has been modeled. The specification describes an electronically controlled car seat. The seat offers position adjustment in 3 axes. Each adjustment axis is equipped with a separate motor.

There are several constraints on the adjustments:

– Any adjustment must be stopped when the car speed is above 5 km/h.
– Any adjustment must be stopped when the battery voltage falls below 10 V.
– At most two axes can be adjusted simultaneously.

Defining Scenarios The first key element of the service–based approach is the definition of the system interface. In the case of an embedded control system, such as seat adjustment control, the interface consists of a set of input signals "understood" by the system and a set of possible feedback signals. In our case study the inputs are the events of pressing/releasing single seat adjustment buttons, propagated to the system in the form of messages. The outputs are different signals activating and deactivating the adjustment motors.

Furthermore, the actual service specification defines its observable behavior of the service and not its internal structure. A good means to specify observable behavior are Message Sequence Charts (MSCs). An example of the behavior specification is shown in Figure 1. It shows a possible adjustment scenario:

– The seat control gets the signal $S2\_SITZ\_HOR\_vor$.
– It starts the motor for horizontal seat adjustment.

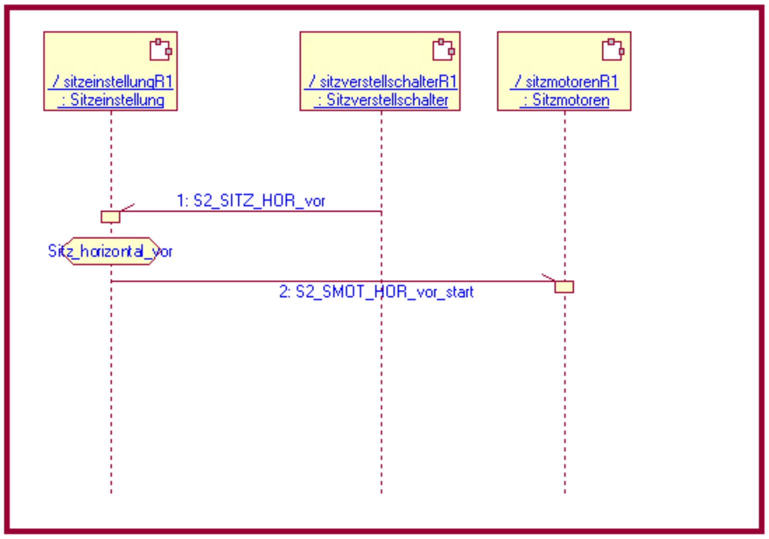

Fig. 1. Example of a basic service "Move seat horizontally forwards"

In the exceptional situation shown in Figure 2 the scenario is more complicated and contains additional steps:

– During the adjustment the seat control gets the signal $CAN\_BATT\_kleiner\_10\_Volt$.
– It stops the adjustment due to low voltage.
– It reports that the adjustment was finished.

Such scenarios become the basic system services. Next subsection explains the transition from scenarios to services.

From Scenarios to Basic Services After identifying the single use case scenarios of our system we want to define basic services. Therefore first of all we identify reoccurring parts in the set of sequence diagrams. These reoccurring parts give useful information for identifying candidates for basic services.

In the preceding paragraph we have shown several sequence diagrams (see Figure 1 and 2). All these sequence diagrams include the message $S2\_SITZ\_HOR\_vor$ indicating the event that the user presses the horizontal seat adjustment button to move the seat forward. Consequently we conclude, that all these scenarios must deal with the functionality of moving the seat forward and we define a basic service *move seat forward* that should realize the different scenarios.

From the set of sequence diagrams defining one single basic service, the basic service interface is extracted. All messages appearing in the different sequence diagrams describing the service are part of the service interface. The service

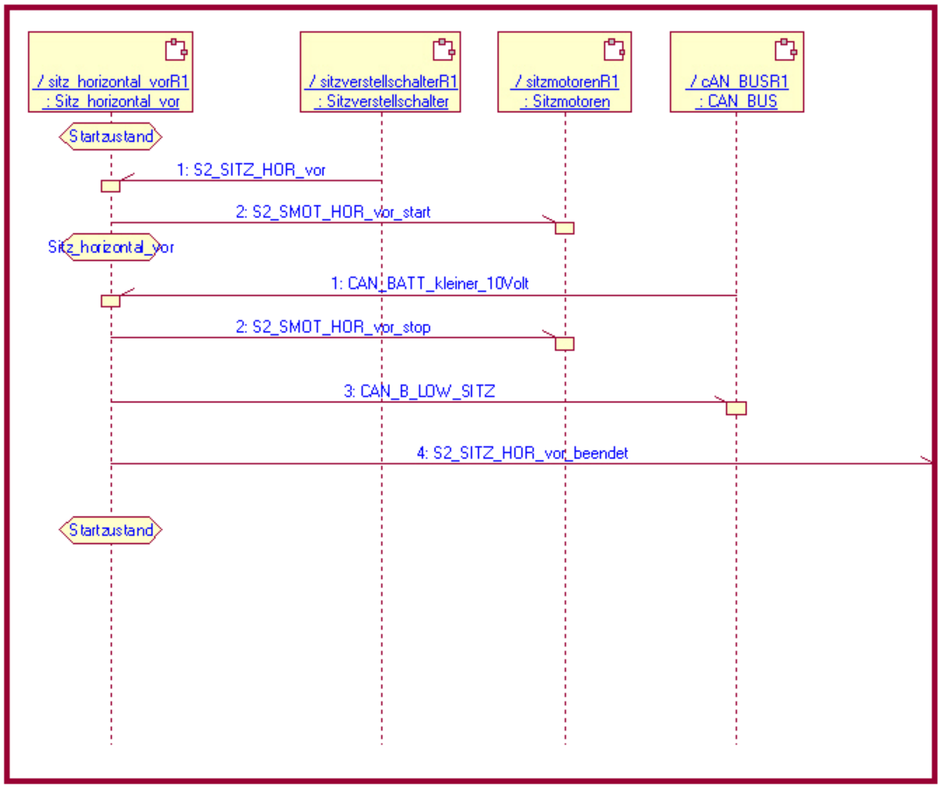

Fig. 2. Example of a basic service "Move seat horizontally forwards", exceptional situation

interface is depicted using a structure diagram. Figure 3 shows the interface definition of the move seat forward service.

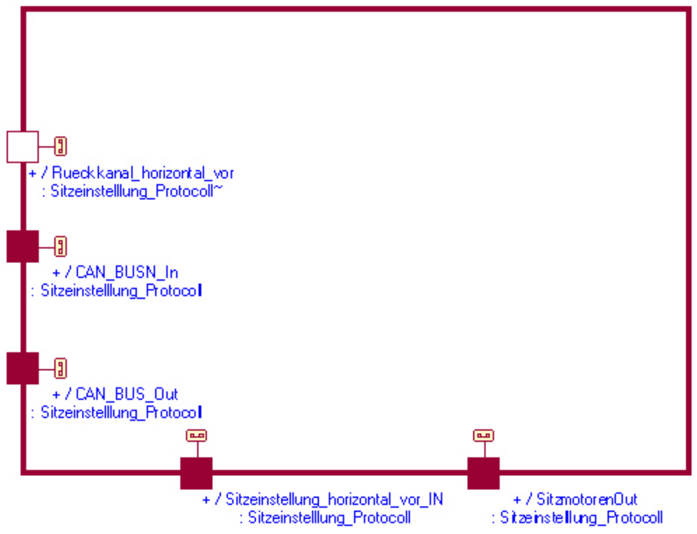
+ / Rueckkanal_horizontal_vor
: Sitzeinstelllung_Protocoll~
+ / CAN_BUSN_In
: Sitzeinstelllung_Protocoll
+ / CAN_BUS_Out
: Sitzeinstelllung_Protocoll
+ / Sitzeinstellung_horizontal_vor_IN
: Sitzeinstelllung_Protocoll
+ / SitzmotorenOut
: Sitzeinstellung_Protocoll

Fig. 3. Interface of the move seat forward service

Based on the different use case scenarios of the move seat forward service we are now building a state chart description (Figure 4). The state chart is tested against the use case scenarios by executing the state machine and providing the inputs and checking the outputs according to the sequence diagrams.

3.2 Building Complex Services

When building a complex behavior, a system is obtained by constructing it from a set of small services, as introduced in the previous subsection. However, a system generally is not just the independent execution of all services: not all the services are simultaneously executable. For example, the services "Move seat forward" and "Move seat backward" are not independent, because they access the same motor.

This conflict is solved by using arbiter services. For each service (or component) that can be used only exclusively, there is one arbiter service monitoring this component. In the case of the services "Move seat forward" and "Move seat backward" the shared hardware component is the adjustment motor. As this motor is used by both services "Move seat forward" and "Move seat backward", we have to introduce the arbiter service for the motor. This service monitors the services and decides which one is allowed to execute.

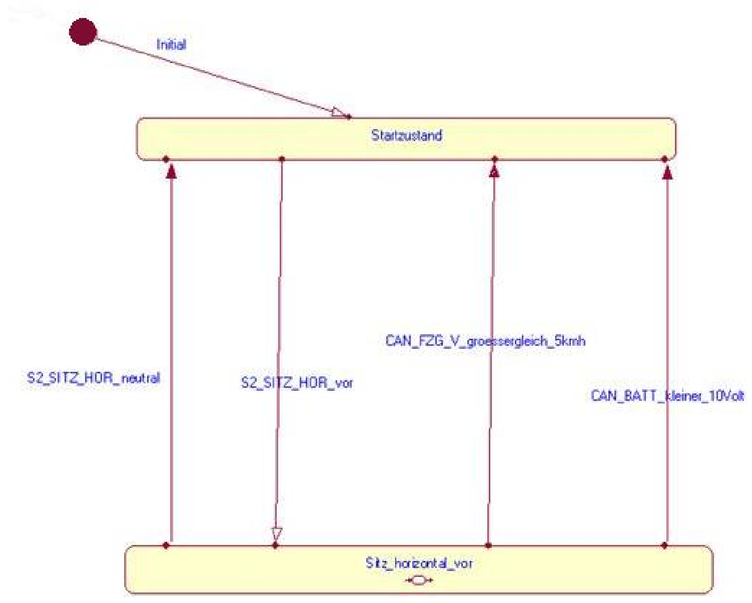

Fig. 4. Behavior of the move seat forward service

An example of the arbiter service is shown in Figure 5. It demonstrates the role of the arbiter service: The simple services themselves are no longer accessible directly. To access them, the user interacts with the arbiter service first. To that end, the arbiter service makes use of three states (*Startzustand*, *Sitz_horizontal_vor*, *Sitz_horizontal_zurueck*). Depending on its state, the arbiter service passes commands to its controlled services. The arbiter service also controls whether subordinate services are finished. Thus teh arbiter controls command execution by dispatching activation signals for both subordinate services.

This “hiding” is desired, because it converts uncontrolled and unwanted feature interaction into controlled one. The undesired feature interaction manifests itself in the usage of the same adjustment motor for two services. The solution is to identify the conflicting services, their shared resources and to introduce an arbiter service for every shared resource. Figures 6 and 7 illustrate this idea of the “controlled service”.

In a component-based architecture, an arbiter component is added to a set of components to avoid conflicts. As a result, we obtain a layered component architecture with three concurrent services: a higher service *Sitz_horizontal*, which is delegating commands to the lower services *Sitz_horizontal_vor* and *Sitz_horizontal_zurueck*. Since all components of this set – including the arbiter – are active at the same time, this composition corresponds to the use of an AND state in Statecharts.

In a service-based architecture, we use alternative configurations of services to avoid conflict of services when constructing complex behavior. Similar to

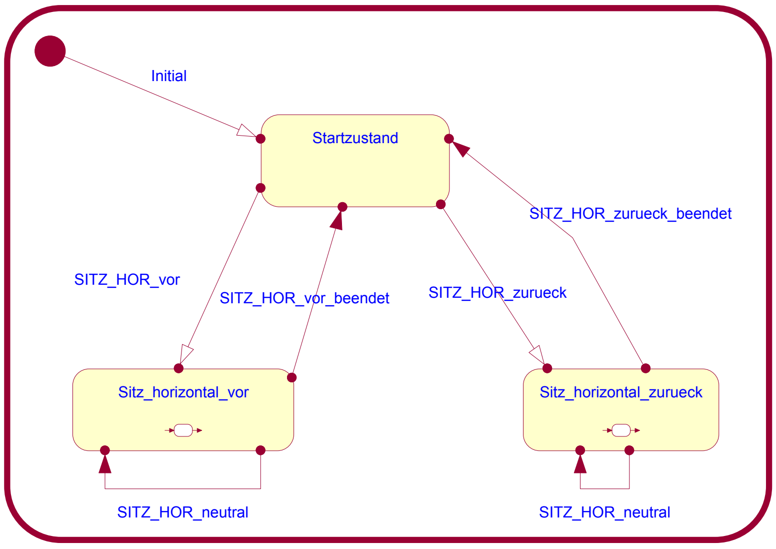

Fig. 5. Example of an arbiter service, service configuration diagram (see also section 3.3)

the use of OR states in Statecharts, this form of compositions corresponds to describing non-simultaneous activation of services. For the horizontal movement service, we achieve this composition by constructing a new service consisting of

- three configurations (*Startzustand*, *Sitz_horizontal_vor*, *Sitz_horizontal_zurueck*), corresponding to the states of the arbiter service described in Figure 5
- an embedding of the two basic services (*Sitz_horizontal_vor*, *Sitz_horizontal_zurueck*) in the corresponding configurations, and activated/deactivated by entering/exiting the configuration

In contrast to the component-based architecture (where we obtained a layered component architecture with three concurrent services), here the services are directly embedded in the configurations.

Note that this from of construction is not supported by component-based approaches like ROOM. Even Statecharts-based approaches support this form of construction only to a limited extent: in Statecharts there is no explicit description of the communication interface of a service associated with a state; furthermore, embedding a service by reference into a state is not supported by all Statecharts variants; finally, since services are activated through configurations, inter-level transitions must broken up at state borders.

3.3 Building Applications

After defining the single services, we want to build whole applications based on them. When combining several services to one application the developer has to

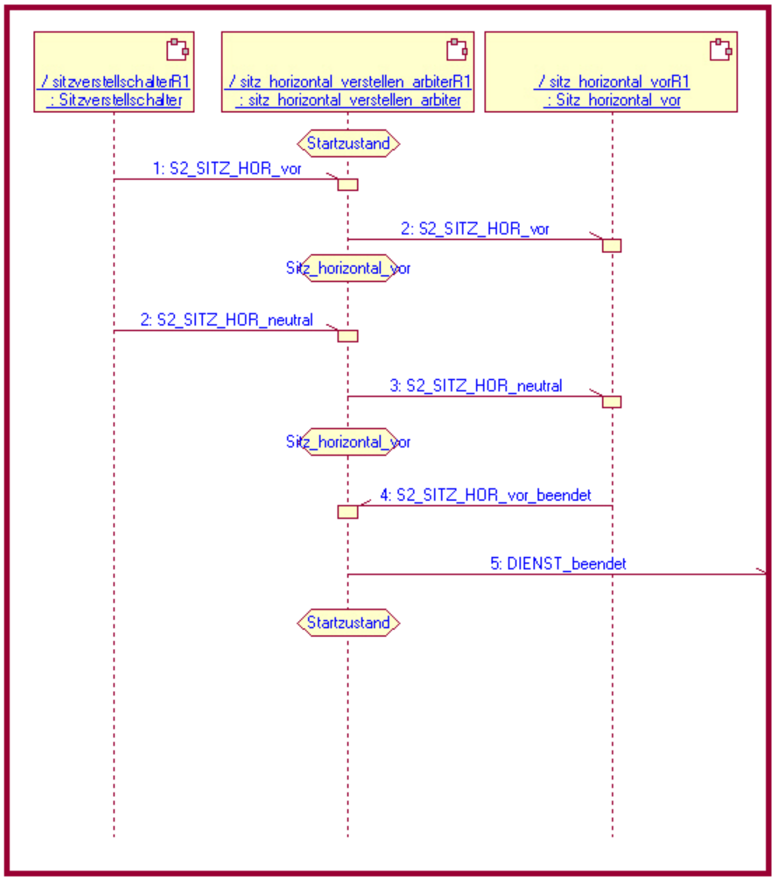

Fig. 6. Arbiter service controlling the service “Move seat forward”

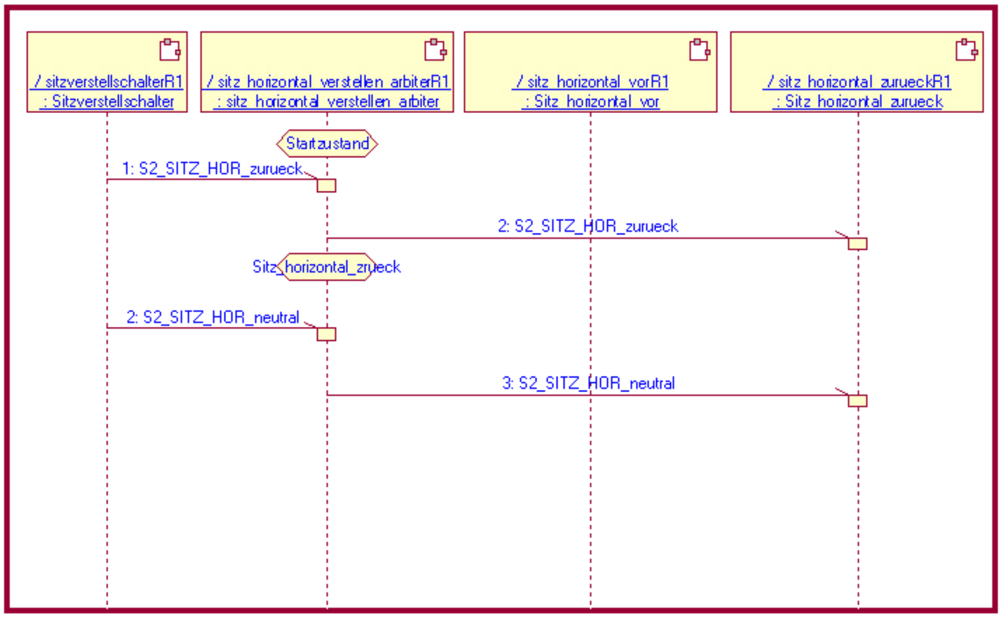
/ sitzverstellschalterR1 : Sitzverstellschalter
/ sitz horizontal verstellen arbiterR1 : sitz horizontal verstellen arbiter
/ sitz horizontal vorR1 : Sitz horizontal vor
/ sitz horizontal zurueckR1 : Sitz horizontal zurueck
Startzustand
1: S2_SITZ_HOR_zurueck
2: S2_SITZ_HOR_zurueck
Sitz_horizontal_zurueck
2: S2_SITZ_HOR_neutral
3: S2_SITZ_HOR_neutral

Fig. 7. Arbiter service controlling the service "Move seat backward"

think about the dynamic system structure during the execution of the application. Typically at a certain time during execution only a subset of the available services of the system is active. The currently inactive services can be suspended to save computing power.

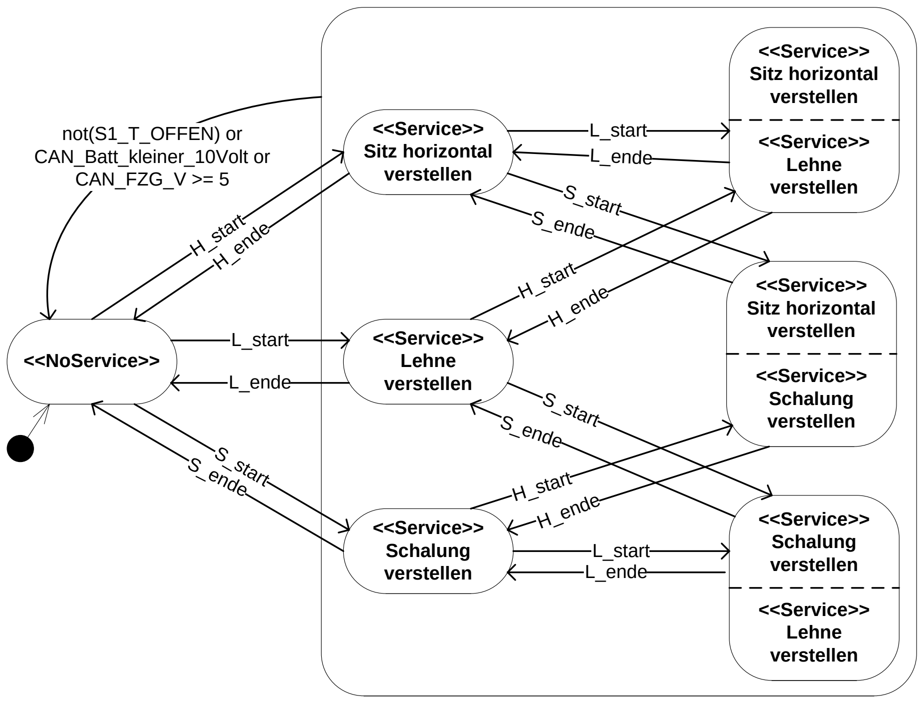
not(S1_T_OFFEN) or
CAN_Batt_kleiner_10Volt or
CAN_FZG_V >= 5
<<NoService>>
<<Service>> Sitz horizontal verstellen
<<Service>> Lehne verstellen
<<Service>> Schalung verstellen
<<Service>> Sitz horizontal verstellen
<<Service>> Lehne verstellen
<<Service>> Sitz horizontal verstellen
<<Service>> Schalung verstellen
<<Service>> Schalung verstellen
<<Service>> Lehne verstellen
H_start
H_ende
L_start
L_ende
S_start
S_ende

Fig. 8. Service configuration diagram of the seat adjustment system

In Section 2.2 we have introduced the concept of service configurations. One configuration describes the set of services that is simultaneously active at a certain time during system execution. Figure 8 shows the service configuration diagram of our seat adjustment system. The diagram describes the available service configurations combining the services to adjust the seat in three different axes.

The service configuration diagram shows one configuration with no active services, three configurations representing the adjustment of a single axis and three configurations representing the simultaneously activation of two adjustment services. One user requirement of the shown seat adjustment system is, that only two axes of the seat can be adjusted simultaneously. The service configuration diagram does not include a configuration activating all three adjustment services at the same time. Consequently the previously mentioned user requirement is fulfilled.

At system startup no service is active at all (configuration $<< NoService >>$). When the user presses a button to adjust the horizontal seat position (H_start event) our system switches to the $Sitz\_horizontal\_verstellen$ configuration. In this configuration the horizontal seat adjustment service is active. This fact is described by a reference between the configuration in the service configuration diagram and the state machine describing the single service. In our example the configuration $Sitz\_horizontal\_verstellen$ holds a reference to the state machine describing the horizontal adjustment service depicted in Figure 5.

Now if the user presses a button to adjust the back of the seat (event L_start) without releasing the button of the horizontal seat position adjustment, a configuration change occurs. In the new active configuration the two services horizontal seat adjustment ($Sitz\_horizontal\_verstellen$) and back adjustment ($Lehnenwinkel\_verstellen$) are running simultaneously. Because the horizontal seat adjustment service is part of the previous and current configuration, the referenced state machine describing the horizontal seat adjustment is not stopped and restarted during the configuration change. In both configurations the same instance of the state machine of this service is referenced and the state machine simply continues working in its current state. This interpretation of configuration changes shows one main difference of the common semantics of hierarchical state charts and the service configuration diagrams.

The service configuration diagram in Figure 8 also describes exception handling. Whenever some seat adjustment services are active, these services are deactivated (configuration change towards the configuration $<< NoService >>$) when the battery has low voltage ($CAN\_Batt\_kleiner\_10Volt$ event) or the car speed is higher or equal than 5 km/h ($CAN\_FZG\_V >= 5$ condition) or the car door is closed ($not(S1\_T\_OFFEN)$ condition).

4 Conclusion

As shown in Subsections 3.2 and 3.3, components and services differ essentially concerning the construction of complex systems: component-based architectures only support concurrent composition to form hierarchic structures; service-based architectures support a more flexible form of reusing behavior by offering composition with concurrent and alternative configurations.

As a result, service-based architecture:

- supports a description of the functional structure of the system using explicit description of modes of operation in form of configurations of active services.
- avoids design-oriented description of the system enforced by component-based architectures.
- supports efficient implementations by explicitly describing active and inactive functionality.

Since service-based architectures target the functional structure of a system, an important aspect of the development process is the systematic detection of conflicts and underspecifications during service composition, as well as the transition from service-based to component-based architectures. Besides the issues covered in [8], current investigations include tool-based refactoring steps supporting conflict elimination and transition from service- to component-based architectures. Furthermore, suitable variants for the constructive design of embedded software are investigated (see, e.g., [1], focusing on issues like explicit description of timing constraints and efficient deployment to hardware.

References

1. Andreas Bauer, Jan Romberg, and Bernhard Schätz. Integrierte Entwicklung von Automotive-Software mit AutoFOCUS. In *2. Workshop Automotive Software Engineering, 34. Jahrestagung der Gesellschaft für Informatik*, 2004.
2. Manfred Broy and Ketil Stoelen. *Specification and Development of Interactive Systems*. Springer, 2001.
3. CCITT. *Specification and Design Language SDL Z.100*, 1983.
4. David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts*. MacGraw-Hill, 1998.
5. Frank Houdek and Barbara Paech. Das Türsteuergerät - eine Beispielspezifikation. Technical Report IESE-Report Nr. 002.02/D, Fraunhofer Institut Experimentelles Software Engineering (IESE), 2002.
6. Leonid Kof. Formales Service Engineering für Eingebettete Systeme. Master's thesis, Technische Universität München, November 2001.
7. Ingolf Heiko Krueger. Service Specification with MSCs and Roles. In M. H. Hamza, editor, *Proceedings of the IASTED International Conference on Software Engineering*, Innsbruck, Austria, February 17-19 2004.
8. Bernhard Schätz. Service-based development of embedded software. TUM-I 0411, TU München, 2004.
9. Bernhard Schätz, Tobias Hain, Frank Houdek, Wolfgang Prenninger, Martin Rappl, Jan Romberg, Oscar Slotosch, Martin Strecker, and Alexander Wisspeintner. Case-tools for embedded systems. Tum-i, TU München, 2003.
10. Bernhard Schätz and Christian Salzmann. Service-Based Systems Engineering: Consistent Combination of Services. In *Proceedings of ICFEM 2003, Fifth International Conference on Formal Engineering Methods*. Springer, 2003. LNCS 2885.
11. B. Selic, G. Gullekson, and P. Ward. *Real-Time Object Oriented Modelling*. Wiley, 1994.
12. Ingomar Thaler. Service basierter Entwurf eines Steuergeräts für Kraftfahrzeuge, 2004. Systementwicklungsprojekt, Institut für Informatik, Technische Universität München.