T U M

I N S T I T U T F Ü R I N F O R M A T I K

A Formal Framework for Integrating Functional and Architectural Views of Reactive Systems

Jewgenij Botaschanjan and Alexander Harhurin

TUM-I0904
Januar 09

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-01-I0904-0/0.-FI

Druck: Institut für Informatik der
Technischen Universität München

Abstract. An integrated model-based development approach has to capture the relationship between requirements, design, and implementation models. In the requirements engineering phase, the most important view is the functional one, which specifies functionalities offered by the system and relationships between them. In the design phase, the component-based view describes the system as a network of interacting components. Via their interaction, they have to realize the black-box behavior specified in the functional view. To ensure the consistency between both views, a formal integration of them is necessary.
The presented formal framework captures both function- and component-based models. In particular, we provide a correct-by-construction procedure, which transforms a functional specification into a component-based architecture. Applicability of the method is evaluated on an industrial case study in a CASE tool.

1 Introduction

Rapid increase in the amount and importance of different software-based functions as well as their extensive interaction are just some of the challenges that are faced during the development of embedded reactive systems. As a consequence, the focus of concerns in software engineering shifts from individual software components to the cross-cutting functions these components offer.

The service-oriented approach is an efficient way to manage the complexity in large-scale model-based development involving many distributed teams. Thereby, functions (services), rather than structural entities (components), are emphasized as the basic building blocks for system composition [10]. In contrast to components, services are partial models which describe only fragmented aspects of the overall system behavior. Following the separation-of-concerns principle, a stakeholder can describe a single purpose as a service independent of other services.

A significant factor behind the difficulty of model-based developing complex software is the wide conceptual gap between the problem (function/service) and the implementation (component) domains of discourse [16]. Currently, the step from service-oriented specifications to component-based architectures requires a *manual* transformation. Testing is the main method for consistency checking between both models.

In light of this observation, we introduce a mathematical framework to model multi-functional reactive systems during the early phases of a model-based development process. Thereby, we focus on the formal definitions of services provided by a system, and show how our service-oriented specification can be transformed into a component-based architecture. The term "service" is used in a variety of meanings, and on various levels of abstraction in the software engineering community. Our notion of service captures functional requirements on a system.

Our approach provides two interrelated views of a software system. *Service-Based Specification* (*SBS*) specifies the system functionality as a set of partial black-box models of the overall behavior. In other words, the overall behavior is specified from different viewpoints [15] by a set of scenarios – causal relations between input and output messages from/to actors in the system environment. We call such scenarios *services* and formalize them by means of I/O automata. Scenarios/services are in certain relationships with each other. The combination of these services according to their rela-

tionships yields the overall specification of the system. A further view of the system is the component-based *Logical Architecture*. It decomposes the system behavior into a network of communicating components. Via their interaction, they have to realize the black-box behavior specified by the SBS [9].

The presented framework forms the basis for the comprehensive development process. On the one hand, both views establish a clean separation between the services provided by the system and the software architecture implementing the services. On the other hand, based on the formal foundation, various correct-by-construction methods for the transition between both models can be defined. The fact that our service- and component-based models are based on the same notation and implemented in the same CASE tool facilitates this process. In particular, we present a property preserving procedure, which transforms an SBS into a network of components automatically. Thereby, several services together with their functional dependencies (relationships) are mapped to a component in a schematic manner. Subsequently, the resulting components can be regrouped arbitrarily and distributed according to quality requirements [4]. However, this is out of the scope of this paper.

Contributions Our contributions in this paper are twofold. First, in the presented operational framework both function- and component-based views of the system can be modeled in an integrated manner. On the one hand, the total behavior of a component can be specified from different viewpoints by a set of partial services. On the other hand, a (hierarchical) service can be realized by a component network. The consistency between both models can formally be proved. Furthermore, due to a clear link between services and components, it is possible to identify the components which are affected by the changes in the service-based specification.

Second, the presented approach bridges the conceptual gap in our model-based development framework. The development process introduced in [17] consists of three phases: requirements analysis yielding a consistent service-based specification [8, 19], transformation from specifications to component-based architectures, and deployment of components onto a network of communicating electronic control units [7]. The transition procedure proposed in this paper guarantees a seamless integration of our models at the top of a model chain closing the gap between requirements and design.

Running Example The concepts introduced in the remainder of the paper will be illustrated on a fragment of a specification originally written and implemented for "Advanced Technologies and Standards" of Siemens, Sector Industry. The considered bottling plant system comprises several distributed subsystems: to transport empty bottles from a storehouse to the bottling plant, fill bottles with items, seal them, and transport them back to the storehouse. All these systems are operated by a central control unit (CU) which provides a user interface to receive commands and display the system status as well as a device interface to send/receive control signals to/from the subsystems. Although there are over 70 scenarios of system behavior, in this paper we consider only a small subset concerning the interplay between the CU and the conveyor belt. Among other things, the user can start and stop the conveyor. There is also an emergency brake available. When the emergency brake is activated, the CU immediately switches the

conveyor off. In this case, the CU is not allowed to switch the system on and the emergency lamp flashes red until an abolition of the emergency command is received.

Outline The remainder of this paper is organized as follows. In Section 2, we compare our work to related approaches. Section 3 introduces the notion of a component and that of a service and explains the methodological difference between the service-based specification and component-based architecture. In Section 4, a formal framework for modeling specifications and software architectures is presented. Section 5 introduces the notion of system modes. In Section 6, we show how a service-based specification can be transformed into a component-based architecture. Finally, we show how our formal approach is implemented in a CASE tool in Section 7 before we conclude the paper in Section 8.

2 Related Work

The presented work is based on a theoretical framework introduced by Broy et al. [10] where the notion of service behavior, decomposition, and refinement are formally defined. This framework proposes to model services as partial stream processing functions. However, it does not cover several relevant issues such as an operational semantics, behavior extension by new aspects, or automatic combination of services into components. In [22], Krüger et al. introduced a methodological approach to defining services and mapping them to component configurations. From use cases, roles and services as interaction patterns among roles are derived. Subsequently, the roles are refined into a component configuration, onto which services are mapped to yield an architectural configuration. Due to the applied algorithm for the synthesis of state machines from MSCs [21], this approach assumes the services to be complete descriptions of the system behavior. This completeness assumption is limiting, considering that interaction-based specifications are inherently partial [30]. Our approach, in contrast, supports a combination of *partial* aspects of the same system behavior.

Several approaches study the model merging in specific domains including requirements [28] and software architecture [2]. Also, a wide range of techniques for supporting synthesis of component models from declarative requirements [13] or scenarios [18] exist. In the last years, SCESM community has studied a number of further synthesis approaches that turn scenario descriptions into state machines [24]. The closest approach to our work is the framework introduced by Uchitel and Chechik [30], who defined the combination of partial descriptions of the same system. In contrast to this work, where different automata are merged into a global one, we aim at a network of components, which can be grouped and distributed according to given quality requirements. Due to a clear link between services and components, it is possible to identify the components which are affected by the changes in the service-based specification.

The relationship between requirements and architectures has received increased attention recently [5]. The goal-oriented approach, as proposed by van Lamsweerde [31], aims at refining high-level goals to requirements and modeling architectures based on underlying system goals. This process relies on a set of predefined refinement patterns, and is not automated. Use Case Maps, introduced by Buhr [11], combine behavior and

structure into one view by allocating use cases (responsibilities) to architectural components. This approach requires human intervention for the allocation of use cases.

The combination of services has traditionally been studied in the telecommunication domain [12]. In this field, services (known as features) are considered as increments of the basic functionality. Approaches to this problem (e.g., by Jackson and Zave [20]) propose analytical techniques to detect and to eliminate feature interactions in the implementation domain. Here, in contrast, we propose a correct-by-construction approach to bridging the gap between requirements and architecture models.

Current approaches in the domain of aspect-oriented modeling [29] are concerned with specifying features in modeling views and analyzing interactions across the views at high levels of abstractions (e.g., [27, 3]). However, in contrast to our work, these approaches do not take the semantical combination of aspects into consideration, and do not support overlapping aspects based on shared messages.

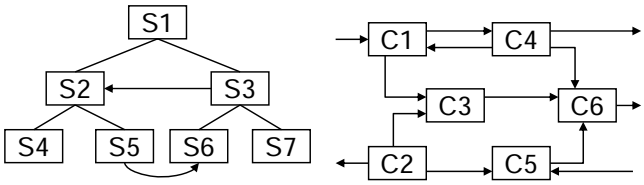
3 Functional and Architectural Views of the System Behavior

In this section, we introduce the notion of a service and that of a component and explain the methodological difference between the service-based specification and component-based architecture.

A central question in software engineering is how to adequately structure the functionalities of the system at different levels of abstraction. A specification consists of a set of requirements, which usually deal with only fragmented aspects of the system behavior. Different scenarios and system modes (e.g., initialization, shutdown, or normal-case behavior) are usually described by separate requirements. Each further requirement adds a new aspect of the specified behavior. Specifications relate functionalities to each other, e.g., a scenario should always precede another one, or it has a higher priority than the other one. These relationships exclusively determine the structure of the functional specification. On the other hand, the functionality has to be realized by a software architecture, which is typically structured into components. There may be many possible component-based architectures that implement the given functionality. Such architectures may be based on various architectural styles such as layered architecture or client-server. The pure functional structure has usually to be redistributed according to quality requirements and other criteria.

In light of this observation, we propose two orthogonal concepts for structuring and decomposing functionality of the system under consideration (cf. Figure 1).

Service-Based Specification The Service-Based Specification defines the *function view* of the system and structures the user functionality into services without any architectural details. The specification consists of a set of services and relationships between them. A service specifies a partial and non-deterministic relation between certain inputs and outputs of the system, which interacts with its environment within a number of scenarios. In other words, a service is a fragmented aspect of the system behavior. The specification does not define the internal data flow within a system – every service obtains inputs from and sends outputs directly to the environment. Usually, services describe system reactions for only a certain subset of the inputs. This partial description



(a) Service-Based Specification (b) Logical Architecture

Fig. 1. Different Views of the System Behavior

allows the distribution of the system functionality over different services and/or leaving the reaction to certain inputs unspecified. In general, adding a service to a specification results in *extending* the system behavior.

Logical Architecture This model defines the *architectural view* of the system and decomposes the functionality into a network of communicating components. Typically, the collaboration between the components realizes the black-box behavior specified at the functional level. The component composition *reduces* the number of possible behaviors of individual components w.r.t. the open-world assumption. The Logical Architecture forms the basis for grouping and distribution of the system functionality over components according to quality constraints rather than functional relationships.

Specification vs. Architecture There are two main differences between both views. Methodologically, it is the difference between the black-box and white-box views of the system. While the SBS provides a hierarchy of functions observable at the outer boundaries of the system, the Logical Architecture focuses on the inside component-based structure of the system. The formal difference between both views lies in the relationships between their building-blocks. In general, inter-service relationships are non-associative, which makes regrouping of services very limited. However, an architecture aims at clustering functionalities into components according to quality requirements, which play no role in the service-based specification. In Section 6 we realize the inter-service relationships by the commutative and associative inter-component communication, that allows us arbitrary restructuring of the component-based architecture.

4 Service-Oriented Development

This section introduces a formal framework for modeling specifications and software architectures of a system. Thereby, the basic building block of both models is a service – a formal representation of a system functionality. Mathematically, a total input-enabled component is a special case of a partial input-disabled service. In the following, we show how services can be combined to service-based specifications and composed to component-based architectures.

4.1 Service

A service has a *syntactic interface* consisting of the sets of typed input and output ports, which represent the system's I/O devices (sensors and actuators) according to Parnas et al. [26]. Figure 2(a) depicts the syntactic interface of a service from our running example. There, input and output ports are depicted by empty and filled cycles, respectively.

The semantics of a service is described by an I/O automaton. This is a tuple $S = (V, \mathcal{I}, T)$ consisting of variables V, initial states $\mathcal{I}$, and a transition relation T. V consists of mutually disjoint sets of typed variables I, O, L. The type of a variable $v \in V$ is denoted by the function $ty(v)$, which maps v to the set of all possible valuations. The variables from I and O are the input and output ports of the service interface, respectively. L is a set of local variables. A *state* of S is a valuation α that maps every variable from V to a value of its type. $\Lambda(V)$ is the set of all type-correct valuations for a set of variables V, i.e., for all $\alpha \in \Lambda(V)$ and all $v \in V$ holds $\alpha(v) \in ty(v)$.

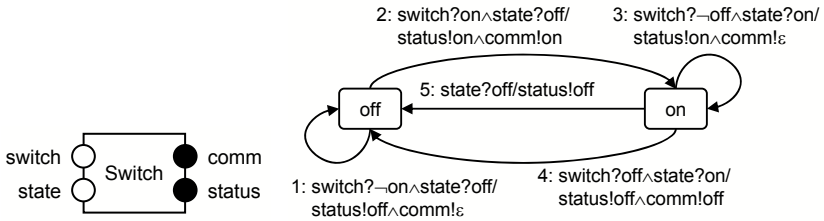
We define following relations on variable valuations: for $\alpha, \beta \in \Lambda(V)$ and $Z \subseteq V$, $\alpha \overset{Z}{=} \beta$ denotes the equality of variable valuations from Z, i.e., $\forall v \in Z : \alpha(v) = \beta(v)$. For an assertion Φ with free variables from V and $\alpha \in \Lambda(V)$, we say that α *satisfies* Φ, written as $\alpha \vdash \Phi$, iff Φ yields true after replacing its free variables with values from α. Finally, the priming operation on a variable name v yields a new variable v' (the same applies for variable sets). Priming of valuation functions yields a mapping of equally valued primed variables, i.e., for given $\alpha \in \Lambda(V)$, α' is defined by $\forall v \in V : \alpha(v) = \alpha'(v')$. Priming is used to argue about the current and next state within the same logical assertion. For Φ with free variables from $V \cup V'$ and $\alpha, \beta \in \Lambda(V)$ we also write $\alpha, \beta' \vdash \Phi$ to denote that Φ yields true after replacing free unprimed variables by values from α and primed ones by values from β.

$\mathcal{I}$ is an assertion over V characterizing the *initial states* of the system. It is allowed to constrain output and local variables only, i.e., the set of possible initial inputs is not constrained by $\mathcal{I}$.

T is a set of transition assertions over $V \cup V'$. In a transition $t \in T$ the satisfying valuations of unprimed variables describe the current state while the valuations of the primed ones constrain the possible successor states. By enabling several satisfying successor state valuations for one current state, we can model non-determinism. A transition is not allowed to constrain primed input and unprimed output variables. By this, we disallow a service to constrain its own future inputs, and enforce the clear separation between the local state (read/write) and the outputs (write only).

We instantiated the above service model for an extended version of I/O automata used in our CASE tool. As in the classical I/O automata [25], a transition leads from one control state to another and might consist of four logical parts: precondition, input pattern, post-condition and output pattern. In our concrete syntax `i?v` denotes an input pattern, which evaluates to true if the variable $i \in I$ has the value v and `o!v` an output pattern, which is satisfied by an assignment of value v to the output variable $o' \in O'$. Figure 2 shows the specification of service `Switch` from our running example, which formalizes the following scenario of the CU. The user can switch the conveyor on/off, by putting one of the two commands (`on` or `off`) in. Additionally, the CU receives the state of the conveyor through the port `state`. If the conveyor is in state `off` and the user switches it on, in the next step the CU sends command `on` through its port

comm to the conveyor, as well as message on through port status to the user display (cf. Transition 2). Note, Transition 5 does only reference two of the four existing ports. This means that the remaining ports are allowed to have arbitrary values within their respective type domains when the automaton executes this transition.



(a) Syntactical Interface (b) I/O Automaton

Fig. 2. Service Switch

In order to be able to reason about transition steps, we define the successor state of some valuation α as $\mathrm{Succ}(\alpha) \stackrel{\text{def}}{=} \{\beta \mid \exists t \in T : \alpha, \beta' \vdash t\}$. The complementary predicate En yields true if a service can make a step: $\mathrm{En}(\alpha) \stackrel{\text{def}}{\Leftrightarrow} \mathrm{Succ}(\alpha) \neq \emptyset$. If $\mathrm{En}(\alpha)$ holds, we say that the service is *enabled* in state α.

The language of a service automation consists of valuation sequences $\alpha_0\alpha_1\ldots$, called *runs*, such that $\alpha_0 \vdash \mathcal{I}$ holds and for all $i \in \mathbb{N}$ either $\alpha_{i+1} \in \mathrm{Succ}(\alpha_i)$ or $\neg\mathrm{En}(\alpha_i)$ and α_i is the last element in the sequence. By this, the semantics of our service is *input-disabled*.

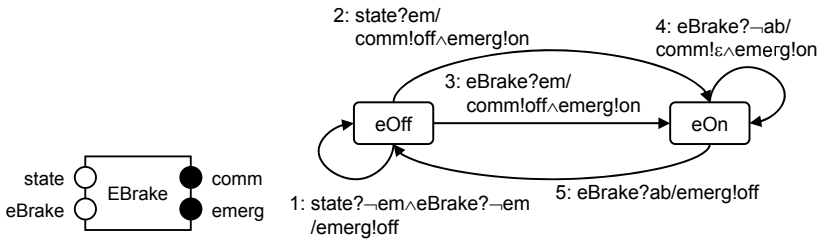
4.2 Service Combination

Single services can be combined to a composite service. This directly reflects the idea that each further requirement/scenario adds a new aspect of the specified behavior. The combination of these fragmented aspects yields the overall system behavior.

Unlike the classical notion of composition (e.g., [25, 14]), which reduces the number of possible behaviors of individual automata, we are interested in obtaining a mechanism for the *extension* of the system behavior. The service combination accepts all inputs, which the single services can deal with as long as the outputs produced by these services are unifiable (not contradictory). The reaction of the combination accords with the reactions specified by the single services.

The combination of two services is defined only if input ports of one service and the output ports of the other do not coincide: $(I_1 \cup L_1) \cap (O_2 \cup L_2) = (O_1 \cup L_1) \cap (I_2 \cup L_2) = \emptyset$ and their common variables $V_1 \cap V_2$ have the same type. Then, we speak about *combinable* services.

For two combinable services S_1 and S_2, their combination $C \stackrel{\text{def}}{=} S_1 \parallel S_2$ is defined by $C \stackrel{\text{def}}{=} (V_C, \mathcal{I}_C, T_C)$, where $I_C \stackrel{\text{def}}{=} I_1 \cup I_2$, $O_C \stackrel{\text{def}}{=} O_1 \cup O_2$, $L_C \stackrel{\text{def}}{=} L_1 \cup L_2$, $V_C \stackrel{\text{def}}{=} I_C \cup L_C \cup O_C$, $\mathcal{I}_C \stackrel{\text{def}}{=} \mathcal{I}_1 \wedge \mathcal{I}_2$. T_C is described by the successor function below. The



(a) Syntactical Interface (b) I/O Automaton

Fig. 3. Service `EBrake`

combined automaton makes a step if either the current input can be accepted by both single services, and their reactions are not contradictory, or the input can be accepted by one of both services only. In the latter case, the local variables of the not enabled service (i.e., the service with $\neg\mathrm{En}(\alpha)$) are not modified, and its output variables (not common with the first service) are unrestricted. Formally, the set of successors of the combination is defined for all $i, j \in \{1, 2\}$, $i \neq j$ as follows:

$$\begin{aligned}\mathrm{Succ}(\alpha) \stackrel{\text{def}}{=} &\{\beta \mid \exists t_1 \in T_1, t_2 \in T_2 : \alpha, \beta' \vdash t_1 \wedge t_2\} \\ &\cup \{\beta \mid \exists t_i \in T_i : \alpha, \beta' \vdash t_i \wedge \neg\mathrm{En}_j(\alpha) \wedge \alpha \stackrel{L_j}{=} \beta\},\end{aligned}$$

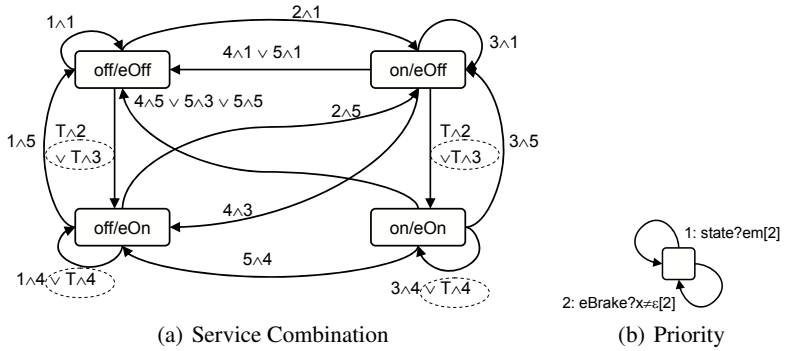
where $\mathrm{En}_i(\alpha)$ is true iff S_i is enabled in state α.

To illustrate the concept of combination, we consider a further scenario concerning the emergency brake from our example. The CU switches the system off if the user puts the emergency brake on (message `em` on port `eBrake`) or a critical state message is received from the conveyor (message `em` on port `state`). The CU is not allowed to switch the system on and the emergency lamp flashes until an abolition of the emergency (`ab`) is received on `eBrake`. The service from Figure 3 formalizes this scenario.

The combination of services `Switch` and `EBrake` results in the automaton from Figure 4(a) (without transitions marked by dashed ovals). There, the labels of transitions are of the form $t_s \wedge t_e$, where t_s and t_e are the transition numbers from Figures 2(b) and 3(b), respectively. A label of the form $T \wedge t_e$ identifies situations where service `Switch` is not enabled. A transition with a label $l_1 \vee l_2$ is an abbreviation of two transitions with labels l_1 and l_2, respectively.

4.3 Prioritized Combination

Usually, some events or scenarios explicitly have a higher priority in specifications than others. For example, the system reaction in the case of emergency has higher priority than the normal-case behavior. In order to be able to reflect this in our service model, we introduce the notion of a *prioritized combination*. It allows an individual service to take control over other services depending on specific input histories. Thereby, we can express different relationships between services without any modifications on them.



(a) Service Combination (b) Priority

Fig. 4. Behavior Specifications

The prioritized combination temporally allows a service to take priority over another service. Thereby, the prioritized combination is controlled by a special service S_P with the interface consisting of all input ports of both services and no output ports. Transitions of S_P might prioritize one of both services. If the current input enables a transition of S_P and this transition prioritizes service S_2, only S_2 is *executing* – the local state of S_1 remains unmodified, the output variables exclusively controlled by S_1 are not subject to any restrictions. If the current input does not enable any transition of S_P or the enabled transition prioritizes no service, the combination behaves like the un-prioritized one. Thus, the priority determines certain inputs for which the system behavior should coincide with the behavior of one of both services only.

The prioritized combination $PC \stackrel{\text{def}}{=} S_1 \parallel^{S_P} S_2$ is defined for a pair of combinable services S_1, S_2 and a special priority service $S_P \stackrel{\text{def}}{=} (I_P \uplus L_P, \mathcal{I}_P, T_P, p)$ by $PC \stackrel{\text{def}}{=} (V_{PC}, \mathcal{I}_{PC}, T_{PC})$, where $I_P \stackrel{\text{def}}{=} I_{PC} \stackrel{\text{def}}{=} I_1 \cup I_2$, $O_{PC} \stackrel{\text{def}}{=} O_1 \cup O_2$, $L_{PC} \stackrel{\text{def}}{=} L_1 \cup L_2 \cup L_P$, $V_{PC} \stackrel{\text{def}}{=} I_{PC} \cup L_{PC} \cup O_{PC}$, $\mathcal{I}_{PC} \stackrel{\text{def}}{=} \mathcal{I}_P \wedge \mathcal{I}_1 \wedge \mathcal{I}_2$. The function $p\colon T_P \to \{0,1,2\}$ defines whether S_1, S_2 or no service is prioritized by a certain transition from T_P. T_{PC} is described by the successor function below. It is defined over the transition set T_C of the unprioritized combination of S_1 and S_2 for all $i, j \in \{1,2\}$, $i \neq j$:

$$\begin{aligned}\mathrm{Succ}(\alpha) \stackrel{\text{def}}{=} &\{\beta \mid \exists t \in T_C : \forall t_P \in T_P : (\alpha, \beta' \vdash t \wedge \neg t_P) \wedge \alpha \stackrel{L_P}{=} \beta\} \\ &\cup \{\beta \mid \exists t \in T_C, t_P \in T_P : (\alpha, \beta' \vdash t \wedge t_P) \wedge p(t_P) = 0\} \\ &\cup \{\beta \mid \exists t_i \in T_i, t_P \in T_P : (\alpha, \beta' \vdash t_i \wedge t_P) \wedge p(t_P) = i \wedge \alpha \stackrel{L_j}{=} \beta\}.\end{aligned}$$

The first subset describes the case when no transition of S_P is enabled, the second one – when the enabled transition of S_P prioritizes none of both services. In both cases, the behavior of $S_1 \parallel^{S_P} S_2$ coincides with the behavior of $S_1 \parallel S_2$. The last subset contains the common behaviors of prioritized service S_i and S_P, i.e., the behavior of service S_1 or S_2 which is temporally prioritized by a currently enabled transition of S_P.

In our running example, it makes sense to prioritize the emergency break signals `eBrake?em` and `state?em`. We require that the combined system must behave like service `EBrake` if one of these signals arrives. The priority service which priori-

tizes emergency signals is depicted in Figure 4(b) (both transitions prioritize service `EBrake`). The prioritized combination of `Switch` and `EBrake` with regard to the priority service results in the automaton from Figure 4(a) (including formulas enclosed in dashed ovals). Whenever an emergency signal has arrived, this combination behaves like service `EBrake` (transitions marked by dashed ovals), otherwise the behavior is identical to the unprioritized combination from the last section.

The un-prioritized combination from the previous section is a special case of the prioritized one. The prioritized combination is in general non-associative, however, it is commutative[1] and distributive. These properties are shown in [8].

4.4 Composition

A component – the architectural building block – can be seen as a (sub)system, which provides a number of services. A component can be specified by a set of services combined according to their interrelationships as described above. In the architectural view, a system is usually described by a network of communicating components. The inter-component communication is not supported by the combination operators. Consequently, we define a *composition* operator on services and, thus, integrate the architectural view into our framework. As a reminder, mathematically, the total component is a special case of the partial service.

The composition operator permits two services to communicate directly via homonymous input/output port pairs. Two services are composable if $L_1 \cap V_2 = L_2 \cap V_1 = I_1 \cap I_2 = O_1 \cap O_2 = \emptyset$ and their shared variables $V_1 \cap V_2$ have the same type.

For two composable services S_1 and S_2, the composition $C \stackrel{\text{def}}{=} S_1 \oplus S_2$ is defined by $C \stackrel{\text{def}}{=} (V_C, \mathcal{I}_C, T_C)$, where $I_C \stackrel{\text{def}}{=} (I_1 \cup I_2) \setminus (O_1 \cup O_2)$, $O_C \stackrel{\text{def}}{=} (O_1 \cup O_2) \setminus (I_1 \cup I_2)$, $L_C \stackrel{\text{def}}{=} L_1 \cup L_2 \cup (V_1 \cap V_2)$, $V_C \stackrel{\text{def}}{=} I_C \cup L_C \cup O_C$, $\mathcal{I}_C \stackrel{\text{def}}{=} \mathcal{I}_1 \wedge \mathcal{I}_2$. T_C is described by the successor function below. The composite automaton makes a step if both services accept their current inputs:

$$\mathrm{Succ}(\alpha) \stackrel{\text{def}}{=} \{\beta \mid \exists t_1 \in T_1, t_2 \in T_2 \wedge \alpha, \beta' \vdash t_1 \wedge t_2\}.$$

The composite service C is enabled if both services in the composition are enabled, i.e., $\mathrm{En}(\alpha) \stackrel{\text{def}}{=} \mathrm{En}_1(\alpha) \wedge \mathrm{En}_2(\alpha)$. The service composition is very similar to those introduced in [14] and, therefore, sketched very briefly. The composition is synchronous, strong causal as well as associative and commutative as shown in [6].

5 System Modes

The prioritized and un-prioritized combination operators with their properties enable a modular and distributed development of service models and allow us to build up service hierarchies. Furthermore, the priority operator is a natural approach to structuring a specification into system modes. System modes are a useful and popular structuring principle for requirements. Complex scenarios might be divided into several modes, or

[1] Provided that priority function p uniquely identifies every transition with combined services.

the overall system behavior might be divided into modes in which different scenarios are specified. Thus, for building intuitive and comprehensive models it is important to support the notion of system modes explicitly. Moreover, the notion of system modes contributes to the efficiency of consistency checks, especially with large requirements specifications.

The system modes are defined over transitions of priority services and specify a set of efficacious services for a given valuation state α. A priority service S_P in $S_1 \parallel^{S_P} S_2$ subdivides the system behavior into three priority modes, in which S_1, S_2, or both of them are efficacious. The logical predicate $isAct_i$ (with $i \in [1..2]$) determines whether service S_i is currently efficacious, $isAct_0$ – whether both services are efficacious:

$$isAct_0(S_P, \alpha) \overset{\text{def}}{\Leftrightarrow} \neg \mathrm{En}_P(\alpha) \vee \exists t \in T_P, \beta \in \mathrm{Succ}_P(\alpha) : \alpha, \beta' \vdash t \wedge p(t) = 0,$$
$$isAct_i(S_P, \alpha) \overset{\text{def}}{\Leftrightarrow} \exists t \in T_P, \beta \in \mathrm{Succ}_P(\alpha) : \alpha, \beta' \vdash t \wedge p(t) = i.$$

Now, we describe the set of efficacious sub-services of a system Sys in a state α: these are the services which execute a transition during the next system step. This set denoted by $Act(Sys, \alpha)$ is inductively defined over the service notation:

$$S \in Act(S, \alpha) \Leftrightarrow \mathrm{En}_S(\alpha),$$
$$S \in Act(S_1 \oplus S_2, \alpha) \Leftrightarrow S \in Act(S_1, \alpha) \vee S \in Act(S_2, \alpha),$$
$$S \in Act(S_1 \oplus^{S_P} S_2, \alpha) \Leftrightarrow \begin{cases} S \in Act(S_i, \alpha) & \text{if } i \in \{1,2\} \wedge isAct_i(S_P, \alpha), \\ S \in Act(S_1 \oplus S_2, \alpha) & \text{if } isAct_0(S_P, \alpha). \end{cases}$$

A *system mode* of a system Sys in a state α corresponds to a subset of its efficacious sub-services. Formally, a system mode is defined by a predicate $mode(\alpha)$ which describes a subset of $Act(Sys, \alpha)$. In our example, the emergency mode is defined by $mode_e(\alpha) \overset{\text{def}}{=} (EBrake \in Act(CU, \alpha))$. We use system modes in the next section to specify properties which need not hold globally but in particular system modes only.

To simplify matters, we assume that all priority services are deterministic. This is a reasonable assumption since different modes of operation in real-life systems bundle mutually exclusive behaviors, e.g., error-recovery vs. normal-case behavior. However, our definition of modes can be generalized to capture the non-deterministic case also by redefining $isAct_0$, $isAct_i$, and Act over two consecutive valuation states.

6 From Specification to Logical Architecture

In this section we show how a service-based specification can be transformed into a component-based architecture. Therefore we propose a property preserving transition procedure. This means, the presented procedure transforms a given service specification $S = S_1 \parallel^{S_P} S_2$ into its realizing network of components C_S, which provably preserves the behavior of S up to stuttering [23].

6.1 Transition Procedure

Figure 5 depicts a synthesized network of components whose overall behavior coincides with the behavior specified by the prioritized combination of services `Switch`

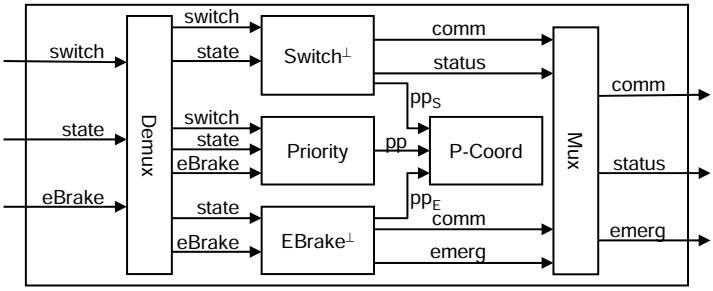


Fig. 5. Logical Architecture

and `EBrake` from Section 4. Component `Demux` splits up the values on the input ports and forwards them to components $\texttt{Switch}^{\perp}$, $\texttt{EBrake}^{\perp}$ and `Priority`. Components `Priority` and `P-Coord` temporally disable $\texttt{Switch}^{\perp}$ according to the semantics of the prioritized service combination from Section 4.3. Finally, the outputs of $\texttt{Switch}^{\perp}$ and $\texttt{EBrake}^{\perp}$ are merged to common values by component `Mux`. According to our strong causal semantics [6], the processing of messages in a component causes a unit delay. Thus, in order to keep inputs and outputs in-sync, each of them are processed by a (de-)mux. The main advantage of the presented solution is that `Priority` immediately disables/enables $\texttt{Switch}^{\perp}$, i.e., without a delay between current inputs and the prioritization effect (see below). Therefore, we take use of our input-disabled semantics to enforce the correct behavior of (un-)prioritized components.

The transition procedure consists of the following steps: (1) Partial services are *totalized* to components. (2) To ensure a correct composition of components, their homonymous variables are renamed in a schematic manner. (3) To preserve the behavior of the service combination, new coordinating components are synthesized. (4) Renamed components and synthesized coordinators are composed to a network. In the following the transition steps are explained in detail.

Totalization Given a partial service $S = (V, \mathcal{I}, T)$. To totalize it to a component $S^{\perp} = (V, \mathcal{I}, T^{\perp})$ by a special value $\perp$, all variable types from V are extended by $\perp$ and $T^{\perp}$ is defined by $\mathrm{Succ}_{S^{\perp}}(\alpha) \stackrel{\text{def}}{=} \mathrm{Succ}_S(\alpha) \cup \{\beta \mid \neg\mathrm{En}_S(\alpha) \wedge \alpha \stackrel{L}{=} \beta \wedge \forall o \in O : \beta(o) = \perp\}$. The behavior of $S^{\perp}$ coincides with the behavior of S for all defined inputs, otherwise $S^{\perp}$ makes a "stuttering step": issues $\perp$ and preserves the local variables valuations.

Renaming In contrast to the combination, in the composition, components are not allowed to share a common input or output port (this ensures the associativity of the composition). In order to establish the required composability between components, their ports are renamed in a uniform manner. The renamed component $S[w/v] = (V_w, \mathcal{I}_w, T_w)$ is defined by $V_w \stackrel{\text{def}}{=} (V \setminus \{v\}) \cup \{w\}$, $\mathcal{I}_w \stackrel{\text{def}}{=} \mathcal{I}[w/v]$, and $T_w \stackrel{\text{def}}{=} \{t[w/v] \mid t \in T\}$, where $v \in V$ and $w \notin V$ is a pair of equal-typed variables. $\Phi[w/v]$ denotes the replacement of all occurrences of v by w and v' by w' in a given assertion Φ. Obviously, the behaviors of S and $S[w/v]$ are equal up to renaming. We lift the renaming operator

for variable sets by $S[v_i/v]_{v\in U}$ for some subset $U \subseteq V$, and define it as a successive renaming of all variables in U by new ones.

I/O Coordination The renaming of a common port p shared by a pair of services (S_1, S_2) yields two ports p_1 and p_2 contained in the interfaces of $S_1[p_1/p]$ and $S_2[p_2/p]$, respectively. To synchronize the values on these ports, a *mux/demux* is synthesized for each input/output port (in Figure 5 three single demuxes are composed to `Demux` and three muxes are composed to `Mux`).

For a pair (S_1, S_2) and their common input port ip, the interface of the demux $dem(S_1, S_2, ip, \bot) \stackrel{\text{def}}{=} (V_d, \mathcal{I}_d, T_d)$ contains one input port $I_d \stackrel{\text{def}}{=} \{ip\}$ and two output ports $O_d \stackrel{\text{def}}{=} \{ip_1, ip_2\}$, where $ip_1, ip_2 \notin (V_1 \cup V_2)$ and $ty(ip) = ty(ip_1) = ty(ip_2)$. Thus, $V_d \stackrel{\text{def}}{=} I_d \uplus O_d$. The semantics of a demux is very simple. It copies each value on the input port to both output ports: $T_d \stackrel{\text{def}}{=} \{ip_1' = ip_2' = ip\}$ and $\mathcal{I}_d \stackrel{\text{def}}{=} ip_1 = ip_2 = \bot$.

For a pair (S_1, S_2) and their common output port op, the interface of the mux $mux(S_1, S_2, op, \bot) \stackrel{\text{def}}{=} (V_m, \mathcal{I}_m, T_m)$ is defined by $V_m \stackrel{\text{def}}{=} I_m \uplus O_m$, $I_m \stackrel{\text{def}}{=} \{op_1, op_2\}$, and $O_m \stackrel{\text{def}}{=} \{op\}$. Thereby, $op_1, op_2 \notin (V_1 \cup V_2)$, $ty(op) = ty(op_1) = ty(op_2)$. The mux forwards the input values to the output port only if both values are equal or one of them is $\bot$. Formally, $T_m \stackrel{\text{def}}{=} \{(op' = op_1 = op_2) \vee (op' = op_1 \wedge op_2 = \bot) \vee (op' = op_2 \wedge op_1 = \bot)\}$. For $\mathcal{I}_m$ we demand that the valuation of op in the mux and in the service combination must be equal: $\alpha \vdash \mathcal{I}_m \stackrel{\text{def}}{\Leftrightarrow} \exists \beta \vdash \mathcal{I}_1 \wedge \mathcal{I}_2 : \beta(op) = \alpha(op)$. Since the reaction to unequal inputs is not specified, the behavior of a mux is not total.

The above synthesis procedures are easily generalized for an arbitrary number of services sharing common I/O ports or for non-common ports contained in the interface of only one service. In the latter case the coordinators become simple unit-delay identity functions.

Synthesis In the following the synthesis procedure is explained in detail. The service combination $S = S_1 \parallel S_2$ is transformed into the following component composition:

$$C_S \stackrel{\text{def}}{=} DE \oplus S_1^{\bot}[v_1/v]_{v\in V_1} \oplus S_2^{\bot}[v_2/v]_{v\in V_2} \oplus MU,$$

where $S_i^{\bot}[v_1/v]_{v\in V_i}$ is the totalization of S_i with variables indexed by $i \in \{1, 2\}$;

$$DE \stackrel{\text{def}}{=} \bigoplus_{ip\in I_1\cup I_2} dem(S_1, S_2, ip, \bot) \quad \text{and} \quad MU \stackrel{\text{def}}{=} \bigoplus_{op\in O_1\cup O_2} mux(S_1, S_2, op, \bot)$$

are the compositions of (de-)muxes for all I/O ports, respectively. Obviously, all components in C_S are mutually composable.

Regarding the prioritized combination, it is important to ensure that the priority component immediately disables/enables one of both affected components, i.e., without a unit delay between current inputs and the prioritization effect. Otherwise, the prioritization is based on outdated inputs and the composition behavior does not coincide with the combination behavior. For example, if components $\texttt{Switch}^{\bot}$, $\texttt{EBrake}^{\bot}$ and `Priority` are connected directly, the prioritization signal generated by `Priority` arrives at the components one unit later than the inputs this signal is based on. Therefore, two components are connected with their priority via an additional coordinator

(cf. `P-Coord` in Figure 5). Due to the input-disabled semantics of our components, $\texttt{Switch}^{\perp}$ and $\texttt{EBrake}^{\perp}$ can execute a transition only if their outputs can be consumed by `P-Coord`. It consumes the outputs from a component only if it receives the prioritization signal for this component from `Priority`. Thus, we realize the *immediate* prioritization of components without any delays.

Formally, for a combined service $S = S_1 \parallel^{S_P} S_2$ with priority $S_P = (I_P \uplus L_P, \mathcal{I}_P, T_P, p)$ we construct a component $S_P^{ext} \stackrel{\text{def}}{=} (I \uplus L \uplus \{pp\}, \mathcal{I}_P, \overline{T}_P)$ with a new output port $pp \notin (I \cup L)$ and $ty(pp) = \{0, 1, 2, \perp\}$. S_P^{ext} behaves like the original service S_P and, additionally, sends its current prioritization decision through the new port pp. According to the semantics from Section 4.3, this decision is defined by the function $p(t)$. If the component S_P^{ext} executes a transition and this transition prioritizes one of both components ($p(t) \in \{1, 2\}$), value $p(t)$ is sent through pp. If no component is prioritized ($p(t) = 0$), a $\perp$ is sent through the port. Formally, $\overline{T}_P \stackrel{\text{def}}{=} \{t \wedge pp' = p(t) \mid t \in T_P \wedge p(t) \in \{1, 2\}\} \cup \{t \wedge pp' = \perp \mid t \in T_P \wedge p(t) = 0\}$. Note, after totalization S_P^{ext} also sends a $\perp$ if S_P is not enabled.

Corresponding output ports are added to the interfaces of S_1 and S_2. For $S_i = (V_i, \mathcal{I}_i, T_i)$ with $i \in \{1, 2\}$ we obtain $S_i^{ext} \stackrel{\text{def}}{=} (V_i \uplus \{pp\}, \mathcal{I}_i, \overline{T}_i)$. Each transition of S_i^{ext} can non-deterministically send the index i or the value 0 through the port pp: $\overline{T}_i \stackrel{\text{def}}{=} \{t \wedge (pp' = i \vee pp' = 0) \mid t \in T\}$. These output ports are used to connect components with their priority via a priority coordinator.

The priority coordinator is a mux $mux(S_1, S_P, S_2, pp, \perp)$ with three input ports pp, pp_1, and pp_2. As a reminder, a mux can execute a transition only if all inputs except for $\perp$ are equal. Thus, the coordinator executes a transition in the following cases. (a) The mux receives value 1 through ports pp and pp_1 and a $\perp$ through pp_2. This means, component S_1 is prioritized and executes a transition. The unprioritized component S_2 is not able to execute a transition because values 2 or 0 on its port pp_2 can not be consumed by the mux. Thus, it sends a $\perp$ and does not modify its local variables. (b) The same goes for $pp = 2$, $pp_1 = \perp$ and $pp_2 = 2$. Here, component S_2 is prioritized. (c) The mux receives a $\perp$ through pp, and 0 through both pp_1 and pp_2. This means, no component is prioritized. S_1 and S_2 execute their transitions and send 0 through pp_1 and pp_2, respectively. In all other cases, the priority coordinator can not execute a transition. Thus, due to the input-disabled semantics, the coordinator prevents other components from executing transitions which are not in accordance with the service combination semantics.

The component composition with priority is defined as

$$C_S \stackrel{\text{def}}{=} DE \oplus \bigoplus_{i \in \{1,2,P\}} (S_i^{ext})^{\perp}[v_i/v]_{v \in V_i} \oplus mux(S_1, S_P, S_2, pp, \perp) \oplus MU.$$

The behavior of the synthesized component network simulates the behavior of the service-based specification. Furthermore, due to the commutativity and associativity of the composition operator, the components can subsequently be regrouped and distributed according to further quality requirements.

6.2 Property Preservation

Now we show the behavior preservation of the corresponding service- and component-based models. For this purpose, we define the following simulation relation.

Definition 1 (Simulation between services and components). *Given a component $C = (V_C, \mathcal{I}_C, T_C)$ and a service $S = (V_S, \mathcal{I}_S, T_S)$, such that $V_C = I \uplus L_C \uplus O$ and $V_S = I \uplus L_S \uplus O$ and $L_S \subseteq L_C$. An infinite component run $\rho_C = \beta_0\beta_1\ldots$* simulates *an infinite service run $\rho_S = \alpha_0\alpha_1\ldots$, denoted by $sim(\rho_C, \rho_S)$, iff $\alpha_0 \overset{V_S}{=} \beta_0$, $\alpha_i \overset{I}{=} \beta_i$, $\alpha_{i+1} \overset{L_S}{=} \beta_{i+2}$, and $\alpha_{i+1} \overset{O}{=} \beta_{i+3}$ hold for all $i \in \mathbb{N}$.*

C simulates *S* up to stuttering*, denoted by $sim(C, S)$, iff for any run $\rho_C = \beta_1\beta_2\ldots$ with $\beta_i(v) \neq \bot$ for all $i \in \mathbb{N}$ and $v \in V_S$ a run ρ_S of S exists such that $sim(\rho_C, \rho_S)$.* □

The definition of $sim(C, S)$ is based on the well-known *stutter reduction* [23], which builds equivalence classes of system runs, consisting of the same sequences of transition steps augmented by arbitrary, but finite sequences of $\bot$-transitions. The representative element of such a class is the run without any $\bot$-transitions.

Theorem 1. *Given a service $S = S_1 \parallel S_2$. Then the corresponding component*

$$C \overset{\text{def}}{=} DE \oplus S_1^{\bot}[v_1/v]_{v\in V_1} \oplus S_2^{\bot}[v_2/v]_{v\in V_2} \oplus MU$$

simulates S, i.e., $sim(C, S)$ holds.

Proof. We prove that every stutter-free trace of C is contained in the language of S. Given some run $\rho_C = \alpha_0\alpha_1\ldots$ such that

$$\alpha_0 \vdash \mathcal{I}_C \qquad \textit{and} \qquad \forall i \in \mathbb{N} : (\alpha_{i+1} \in \mathrm{Succ}_C(\alpha_i) \wedge \forall v \in V_S : \alpha_i(v) \neq \bot)$$

we show that there exists a run S, ρ_S, such that $sim(\rho_S, \rho_C)$ by induction over the valuation index i.

The initial valuations of input variables are unconstrained in both S and C. The consequence is that the initial predicates of C's subservices do not interfere — every initial valuation can be calculated independently. The local variables of S are initially constrained by the same predicate $\mathcal{I}_1 \wedge \mathcal{I}_2$ in both cases. Finally, the initial valuation of every output variable in C is constrained by the initial predicate of the corresponding coordinator, which permits exactly the valuations described by $\mathcal{I}_1 \wedge \mathcal{I}_2$. Thus, $\alpha_0 \vdash \mathcal{I}_S$.

Induction Step Assuming that there is some valuation $\beta \in \Lambda(V_S)$ reachable in S, such that $\beta \overset{I}{=} \alpha_i$, $\beta \overset{L_S}{=} \alpha_{i+1}$ and $\beta \overset{O}{=} \alpha_{i+2}$ for some $i \geq 0$. In other words, the finite run from an initial state to β is the prefix of ρ_S. We need to consider two cases:

$\underline{\neg \mathrm{En}(\beta)}$: When β is the last valuation in a finite run, then the input of α_i (and of β) will serve as the input of $S_1^{\bot}[v_1/v]_{v\in V_1}$ and $S_2^{\bot}[v_2/v]_{v\in V_2}$ in the step $i+1$. In this step, according to the assumption, the valuations of local variables from L_S in C will coincide with the corresponding valuations of S in state β. By the totalization of the both services they will either issue $\bot$-values on all their output ports, or produce

non-unifiable (contradictory) non-$\bot$ outputs in the step $i+2$. In the former case, the mux-services will also issue $\bot$ as the output of C in the step $i+3$. In other words, we obtain a stutter step, which contradicts to the assumption about ρ_C. In the latter case, α_{i+2} is the last valuation of ρ_C, which also cannot be true.

$\underline{\mathrm{En}(\beta)}$*: We must show that there exists $\gamma \in \mathrm{Succ}_S(\beta)$ with $\gamma \overset{I}{=} \alpha_{i+1}$, $\gamma \overset{L_S}{=} \alpha_{i+2}$, and $\gamma \overset{O}{=} \alpha_{i+3}$.*

The first equation follows directly from the fact that the inputs are subject to no restriction in both, S and C. The input valuations of α_i (and, equally, of β) become input valuations of $S_1^{\bot}[v_1/v]_{v\in V_1}$ and $S_2^{\bot}[v_2/v]_{v\in V_2}$ in the step $i+1$. In the same step the local sates of variables from L_S coincide for C and S. Thus, for every successor, γ, of S_i in the state β there exists an equal (up to renaming) successor of $S_i^{\bot}[v_i/v]_{v\in V_i}$ in the state α_{i+1} for $i \in \{1,2\}$. At most one successor state (either $\mathrm{Succ}_1(\beta)$ or $\mathrm{Succ}_2(\beta)$) may be empty. Then, the corresponding successor set of $S_1^{\bot}[v_1/v]_{v\in V_1}$ or $S_2^{\bot}[v_2/v]_{v\in V_2}$ will contain $\bot$-valuations only. Otherwise, the $\bot$-valuation is not *in the successor state. Since S is enabled in β, there exist either at least one pair of unifiable successors of S_1 and S_2 in this state, or, if one of the services is not enabled, the corresponding C's subservice produces a $\bot$-valuation. Only unifiable outputs are accepted by the mux-services. Thus, outputs of C in step $i+3$ coincide with the outputs of the combination of S_1 and S_2, since the behavior of the mux-coordinators reproduces the semantics of the unprioritized combination-operator.* □

Theorem 2. *Given a service $S = S_1 \parallel^{S_P} S_2$. Then the corresponding component*

$$C \overset{\text{def}}{=} DE \oplus \bigoplus_{i\in\{1,2,P\}} (S_i^{ext})^{\bot}[v_i/v]_{v\in V_i} \oplus mux(S_1, S_P, S_2, pp, \bot) \oplus MU$$

simulates S, i.e., $sim(C, S)$ holds.

Proof. We prove that every stutter-free trace of C is contained in the language of S. Given some run $\rho_C = \alpha_0\alpha_1\ldots$ such that

$$\alpha_0 \vdash \mathcal{I}_C \qquad \textit{and} \qquad \forall i \in \mathbb{N} : (\alpha_{i+1} \in \mathrm{Succ}_C(\alpha_i) \wedge \forall v \in V_S : \alpha_i(v) \neq \bot)$$

we show that there exists a run S, ρ_S, such that $sim(\rho_S, \rho_C)$ by induction over the valuation index i.

The initial valuations of input variables are unconstrained in both S and C. The consequence is that the initial predicates of C's subservices do not interfere — every initial valuation can be calculated independently. The local variables of S are initially constrained by the same predicate $\mathcal{I}_P \wedge \mathcal{I}_1 \wedge \mathcal{I}_2$ in both, S and C. Finally, the initial valuation of every output variable in C is constrained by the initial predicate of the corresponding coordinator, which permits exactly the valuations described by $\mathcal{I}_1 \wedge \mathcal{I}_2$ (S_P cannot influence the initial output valuations). Thus, $\alpha_0 \vdash \mathcal{I}_S$.

Induction Step Assuming that there is some valuation $\beta \in \Lambda(V_S)$ reachable in S, such that $\beta \overset{I}{=} \alpha_i$, $\beta \overset{L_S}{=} \alpha_{i+1}$ and $\beta \overset{O}{=} \alpha_{i+2}$ for some $i \geq 0$. In other words, the finite run from an initial state to β is the prefix of ρ_S. We need to consider two cases:

$\underline{\neg \mathrm{En}(\beta)}$: *When β is the last valuation in a finite run, then the input of α_i (and of β) will serve as the input of $(S_1^{ext})^{\perp}[v_1/v]_{v\in V_1}$, $(S_2^{ext})^{\perp}[v_2/v]_{v\in V_2}$, and $(S_P^{ext})^{\perp}[v_P/v]_{v\in V_P}$ in the step $i+1$. In this step, according to the assumption, the valuations of local variables from L_S in C will coincide with the corresponding valuations of S in state β. By the totalization of $(S_1^{ext})^{\perp}[v_1/v]_{v\in V_1}$ and $(S_2^{ext})^{\perp}[v_2/v]_{v\in V_2}$ they will either (1) issue $\perp$-values on all their output ports, or (2) produce non-unifiable (contradictory) outputs in the step $i+2$. In the same step, $(S_P^{ext})^{\perp}[v_P/v]_{v\in V_P}$ will either (a) issue a $\perp$ on pp-port, or (b) prioritize of the subservices. In cases (1a) and (1b), the mux-services will also issue $\perp$ as the output of C in the step $i+3$. In other words, we obtain a stutter step, which contradicts to the assumption about ρ_C. In cases (2a) and (2b), α_{i+2} is the last valuation of ρ_C, which also cannot be true.*

$\underline{\mathrm{En}(\beta)}$: *We must show that there exists $\gamma \in \mathrm{Succ}_S(\beta)$ with $\gamma \overset{I}{=} \alpha_{i+1}$, $\gamma \overset{L_S}{=} \alpha_{i+2}$, and $\gamma \overset{O}{=} \alpha_{i+3}$.*

The first equation follows directly from the fact that the inputs are subject to no restriction in both, S and C. The input valuations of α_i (and, equally, of β) become input valuations of $(S_1^{ext})^{\perp}[v_1/v]_{v\in V_1}$, $(S_2^{ext})^{\perp}[v_2/v]_{v\in V_2}$, and $(S_P^{ext})^{\perp}[v_P/v]_{v\in V_P}$ in step $i+1$. In the same step the local sates of variables from L_S coincide for C and S. Thus, for every successor, γ, of S_i in the state β there exists an equal (up to renaming) the successor of $(S_i^{ext})^{\perp}[v_i/v]_{v\in V_i}$ in the state α_{i+1} for $i \in \{1,2,P\}$. The prioritization for the transition from β to γ in S corresponds to the output on the pp-port of $(S_P^{ext})^{\perp}[v_P/v]_{v\in V_P}$ in step $i+1$. At most one successor state either $\mathrm{Succ}_1(\beta)$ or $\mathrm{Succ}_2(\beta)$ may be empty. Then, the corresponding successor set of $(S_1^{ext})^{\perp}[v_1/v]_{v\in V_1}$ or $(S_2^{ext})^{\perp}[v_2/v]_{v\in V_2}$ will contain $\perp$-valuations only. Otherwise, the $\perp$-valuation is not *in the successor state. Also, at least two of the outputs on the pp-, pp_1-, and pp_2-ports will coincide.*

$\underline{\alpha_{i+2}(pp) = \perp}$: *Since S is enabled in β, there exist either at least one pair of unifiable successors of S_1 and S_2 in this state and $\alpha_{i+2}(pp_1) = \alpha_{i+2}(pp_2) = 0$, or, if one of the services (w.l.o.g. S_1) is not enabled, the corresponding C's subservice produces a $\perp$-valuation. In the latter case $\alpha_{i+2}(pp_2) = 2$.*

$\underline{\text{W.l.o.g. } \alpha_{i+2}(pp) = 1}$: *According to the combination semantics, there must exist a transition $t_P \in T_P$ with $p(t_P) = 1$. Then, there exists a successor of S_1 in state β and a simulating successor of $(S_1^{ext})^{\perp}[v_1/v]_{v\in V_1}$ in state α_{i+1} with $\alpha_{i+2}(pp_1) = 1$. Further on, there is* no *unifiable successor of $(S_2^{ext})^{\perp}[v_2/v]_{v\in V_2}$, since it cannot issue a 1 on its port pp_2. Thus, $(S_2^{ext})^{\perp}[v_2/v]_{v\in V_2}$ can only make a stutter step.*

Only unifiable outputs are accepted by the mux-services. Thus, outputs of C in step $i+3$ coincide with the outputs of the combination of S_1 and S_2, since the behavior of the mux-coordinators together with $mux(S_1, S_P, S_2, pp, \perp)$ reproduces the semantics of the prioritized combination-operator. □

7 Tool Support

Both service- and component-based perspectives are integrated in a CASE tool. AutoFOCUS [1] is a tool for the model-based development of reactive systems. It supports

graphical description of the developed system in both functional and architectural perspectives. Each perspective consists of three views (cf. Figure 6). In the *Project Explorer*

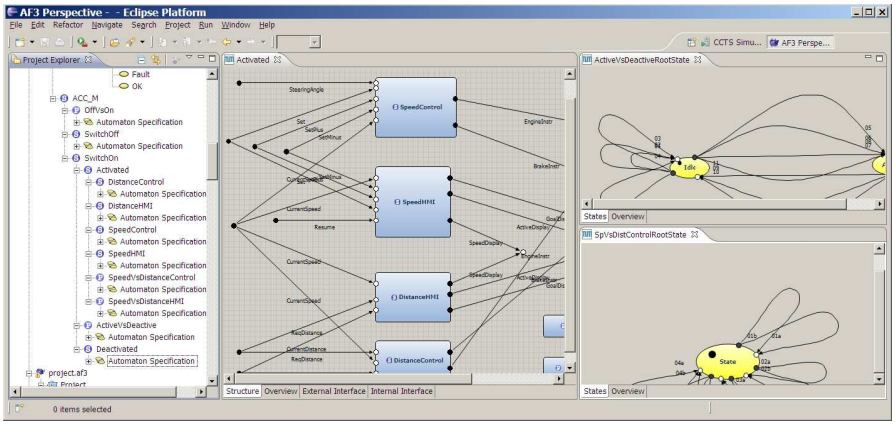


Fig. 6. Different AutoFOCUS Views for SBS Approach

view services/components are structured hierarchically. In the *Structure Diagrams* view syntactical interfaces are defined. *State Transition Diagrams* describe the behavior of services/components using our I/O automata. The simulation environment allows us to validate service-based as well as component-based models. We are currently working on an automatic transformation of service-based specifications into logical architectures using the transition procedure from Section 6.

8 Conclusion and Outlook

We have presented an automata-based framework for modeling multi-functional reactive systems during the early phases of a model-based development process. Two orthogonal perspectives (functional and architectural) on the system behavior are formally defined. The proposed Service-Based Specification combines single (fragmented and incomplete) scenarios, formalized as services, to an overall system behavior. Thereby, we focus on a combination of partial behaviors concerning the same (sub-)system but specified from different viewpoints. The Logical Architecture defines the architectural view and decomposes the functionality into collaborating components. Via their interaction, the components realize the black-box behavior specified at the functional level. The formal integration of service- and component-based models in a mathematical framework is the main contribution of our approach. The proposed property-preserving step from service-based specifications to logical architectures forms the basis for the comprehensive development process that can be supported by a tool in the sense of model transformation techniques. Thus, our models integrate seamlessly at the top of a model chain closing the formal gap between requirements and design.

We see the application area of the proposed approach in the model-based development of reactive systems. With a scenario-based specification as an input, this specification is checked for consistency by verification and simulation [8, 19]. Subsequently, the specification is transformed into a component-based architecture. Finally, the components are deployed onto a network of electronic control units [7]. We are currently working on automating the second step. The fact that both specification and architecture models have the same mathematical foundation facilitates this process.

References

1. AutoFOCUS 3. `http://af3.in.tum.de/`.
2. M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, and D. Garlan. Differencing and merging of architectural views. In *Proceedings of ASE'06*. IEEE Computer Society, 2006.
3. E. Baniassad and S. Clarke. Theme: an approach for aspect-oriented analysis and design. In *Proceedings of ICSE'04*, 2004.
4. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.
5. D. M. Berry, R. Kazman, and R. Wieringa. Second international workshop on from software requirements to architectures (STRAW'03). *SIGSOFT Softw. Eng. Notes*, 29(3), 2004.
6. J. Botaschanjan. *Techniques for Property Preservation in the Development of Real-Time Systems*. PhD thesis, TU München, 2008.
7. J. Botaschanjan, A. Gruler, A. Harhurin, L. Kof, M. Spichkova, and D. Trachtenherz. Towards Modularized Verification of Distributed Time-Triggered Systems. In *Proceedings of FM'06: Formal Methods*. Springer, 2006.
8. J. Botaschanjan, A. Harhurin, and L. Kof. Service-based Specification of Reactive Systems. Technical Report TUM-I0815, Technische Universität München, 2008.
9. M. Broy, I. Krüger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2), 2007.
10. M. Broy, I. H. Krüger, and M. Meisinger. A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1), 2007.
11. R. J. A. Buhr. Use case maps as architectural entities for complex systems. *IEEE Trans. Softw. Eng.*, 24(12), 1998.
12. M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comput. Networks*, 41(1), 2003.
13. C. Damas, B. Lambeau, and A. van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proceedings of FSE'14*. ACM, 2006.
14. L. de Alfaro and T. A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5), 2001.
15. A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1), 1992.
16. R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Proceedings of FOSE'07*. IEEE Computer Society, 2007.
17. A. Gruler, A. Harhurin, and J. Hartmann. Development and configuration of service-based product lines. In *Proceedings of SPLC'07*. IEEE Computer Society, 2007.
18. D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. In *Proceedings of CIAA'00*. Springer, 2001.
19. A. Harhurin and J. Hartmann. Towards consistent specifications of product families. In *Proceedings of FM'08: Formal Methods*, volume 5014 of *LNCS*. Springer, 2008.

20. M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Trans. Softw. Eng.*, 24(10), 1998.
21. I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In *Proceedings of the Distributed and Parallel Embedded Systems*. Kluwer Academic Publishers, 1999.
22. I. H. Krüger and R. Mathew. Systematic development and exploration of service-oriented software architectures. In *Proceedings of WICSA'04*, 2004.
23. L. Lamport. What good is temporal logic. In R. E. A. Mason, editor, *Proceedings of the IFIP 9th World Congress*, 1983.
24. H. Liang, J. Dingel, and Z. Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proceedings of SCESM'06*. ACM, 2006.
25. N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3), 1989.
26. D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1), 1995.
27. A. Rashid, A. Moreira, and J. Araújo. Modularisation and composition of aspectual requirements. In *Proceedings of AOSD'03: Aspect-oriented Software Development*. ACM, 2003.
28. M. Sabetzadeh and S. Easterbrook. View merging in the presence of incompleteness and inconsistency. *Requir. Eng.*, 11(3), 2006.
29. A. Solberg, D. M. Simmonds, R. Reddy, S. Ghosh, and R. B. France. Using aspect oriented techniques to support separation of concerns in model driven development. In *Proceedings of COMPSAC'05*. IEEE Computer Society, 2005.
30. S. Uchitel and M. Chechik. Merging partial behavioural models. *SIGSOFT Softw. Eng. Notes*, 29(6), 2004.
31. A. van Lamsweerde. From system goals to software architecture. *Formal Methods for Software Architectures*, 2804/2003, 2003.