

Service-Based Systems Engineering: Consistent Combination of Services

Bernhard Schätz¹, Christian Salzmann²

¹ email: schaezt@in.tum.de

Fakultät für Informatik, TU München, 80290 München, Germany

² email: christian.salzmann@bmw-carit.de

BMW Car IT GmbH, Petuelring 116, 80809 München, Germany

Abstract. Using service-based system descriptions simplifies the specification of complex reactive systems as found in the domain of web-services as well as embedded systems. To support a service-based development process applicable in safety-critical areas, a precise understanding of the notions *service*, *component* and *interface* is introduced as well as methodical steps like composition, and consistency and completeness validation. The applicability of our definitions is demonstrated in the context of tool-supported feature interaction detection.

Keywords: Service, component, specification, behavior, partiality, consistency, completeness, implementation, interaction, formalization, application, tool-support, model checking.

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.

To enable a service-based engineering process, however, in both domains a precise definition of the notion of a service as well as the corresponding methodical steps (checking their compatibility, combining services into components) is required. In Section 3 we will give such a definition, based on the preliminaries introduced in Section 2. Targeting service-based software engineering in general, we furthermore show the relationship between services and components in Section 4 including the implementation of a service (network) by a component (network). Since safety-related issues play an important role in embedded systems, in Section 5 we define methodical properties like completeness and consistency of services; issues more specific to dynamic networks (e.g., dynamic allocation of services) are not discussed here. For the practicability of the introduced concepts it is important to evaluate their applicability in a in a tool supported service-based development process. Therefore, in each section we apply the introduced definitions by translating them into a form suitable for a model checker for the relational μ calculus.

The basic techniques introduced are not specific to the semantic model used here but can be easily transferred to others like I/O-Automata [LT89] or TLA [Lam93] as well as other tools like bounded model checking or CLP based approaches. The contribution of the formalization therefore rather lies in the transfer of the formal techniques than in the introduction of new formal concepts. Basic principles behind this formal model have been applied in a service-engineering environment [Sal02] and in a model-checking approach to detect feature interaction [Sch02].

2 Preliminaries

To relate our definition of service to other forms, we will first look at some definitions found in other work, identifying the essential difference between services and components and the advantages of a service-based approach. Furthermore, we give a short introduction in the semantic model that is used to give a precise definition of the notions of service and component.

2.1 Services vs. Components and Interfaces

There are several different definitions of what a service is; generally more pragmatically described than precisely defined. The web service definition language (WSDL) defines a service “*as collections of network endpoints, or ports*” [CCM+], basically corresponding to a typed interface or signature description. In the Open Systems Interconnection-Reference Model (OSI-RM) a service is defined as, “*a capability of a given layer, and the layers below it, that (a) is provided to the entities of the next higher layer and (b) for a given layer, is provided at the interface between the given layer and the next higher layer*”. A very general definition is “*A service is an abstract protocol*” [BL01].

While these definitions are quite diverse, all of the above-cited definitions have in common that they focus on *interfaces* and *interactions* but exclude *structure*. Therefore, 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* in form of signatures defining the types of messages that flow via these interfaces (e.g., interfaces in Java or the interface of a hardware interfaces). *Components* are defined as reusable units of behavior and structure [BS01,Müh96]. 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; however they also support a stronger form of (behavioral) combination.

In a nutshell, 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.

2.2 Semantic Model:FOCUS

In the following we use the model of stream-processing functions to introduce our definitions. However, the definitions are not specific to this model but directly carry over to other formalisms focusing on system interaction and supporting message-asynchronous communication, concurrent input as well as output actions, and time-ordered interactions.¹ Thus, TLA [Lam93] or Reactive Modules [AH99] are suitable models as well. Furthermore, when applying the concepts, we do not operate on the mathematical model itself. We rather use more structured and intuitive description techniques like state-transition or sequence diagrams as shown in Figure 2. Besides a more structured description, these techniques support reuse of modularized behavior, e.g., by encapsulating services in partial state-transition descriptions [HS99].

Here, we use an adapted version of the general model introduced in [BS01]. Basically, the mathematical model of a component (or service) consists of its externally observable behavior, i.e. the messages received from the environment or sent to it. Messages are sent and received via channels. We use a timed model, splitting the observable behavior into time slots; during each time slot, an arbitrary (but finite) number of messages can be received or sent via each channel. The behavior of a component or service can then be described by channel histories, assigning a sequence of messages to each channel and time slot. In the following, we introduce stream relations to model those forms of behaviors.

For a given set of messages M , the set M^* defines the set of *finite sequences* of messages including the empty sequence consisting of no messages. To model a com-

¹ See [KS03] for classification of models of reactive systems.

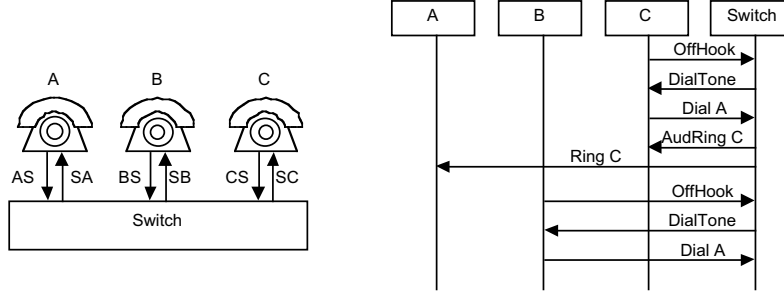


Figure 1: Example of the POTS Interface and Description of an Observation

plete observed interaction of exchanged messages from the set M , we use the set of *infinite streams* of finite sequences, with notation M^ω . Those infinite streams of finite sequences can be identified with functions from the natural numbers Nat to finite sequences M^* , i.e. $M^\omega \equiv Nat \rightarrow M^*$. Thus, for a time slot $t \in Nat$, s_t describes the finite message sequence assigned to time slot t of a stream $s \in M^\omega$.

To combine observation with channels, we introduce the notion of a channel history. Given a set C of channel identifiers, a channel history is a mapping from those identifiers to message streams M^ω . The set of all *channel histories* for a given set of channel identifiers is described by \vec{C} , i.e., $\vec{C} \equiv C \rightarrow M^\omega$. Furthermore, we define the restriction $h \uparrow C'$ of a channel history h to subset $C' \subseteq C$ by standard function restriction. Restriction corresponds to the hiding of channels. Finally, using channel histories, we define the behavior of a component or service with a given interface by a relation of channel histories.

For sets of input and output channels I, O , a *stream relation* s over these input and output channels is defined as a (partial) function $s : \vec{I} \rightarrow \wp(\vec{O})$.² For a stream relation over I, O , we define its *restriction* $s \uparrow (I', O')$ to (I', O') with $I' \subseteq I, O' \subseteq O$ by

$$s \uparrow (I', O') = \{(i \uparrow \vec{I}', o \uparrow \vec{O}') \mid (i, o) \in s\}$$

using restriction of channel histories.

Note that – since we focus on the introduction of a service notion – in this short introduction we did not impose additional requirements to be fulfilled by a stream function or relation to be realizable or implementable, e.g. a weak causality constraint or a strong realizability constraint as defined in [BS01]. In the following, we use the notation $\vec{I} \mapsto \wp(\vec{O})$ for total stream functions respecting those additional properties. Note that the properties defined in [BS01] are union stable, i.e. for functions $s_1, s_2 : \vec{I} \mapsto \wp(\vec{O})$, also $s_1 \cup s_2 \in \vec{I} \mapsto \wp(\vec{O})$. Therefore, for a partial function $s : \vec{I} \rightarrow \wp(\vec{O})$ we can define $\hat{s} \equiv \bigcup_{c \in \vec{I} \mapsto \wp(\vec{O}) \wedge c \subseteq s} c$ as the most general causality re-

specting function implementing s .

² Instead of relational notation, we will use functional notation for this set-valued function for ease of reading.

2.3 Application: Telecommunication Services

Throughout the following sections, we illustrate our approach by applying it to the analysis of feature interaction, using automatic analysis by the symbolic model checker tool μ cke [Bie97]. We illustrate our approach using an example from the telecom domain, because telecom services are more familiar to most readers than, e.g., automotive services; nevertheless they have a similar level of interactions between services, and require a similar safety level concerning completeness and consistency of services without introducing additional aspects like hard real-time bounds. To simplify matters, we will use a more basic formal model as introduced in Section 2.2 allowing only one (or no) message to be transported via a channel in a single time slot.

The left side of Figure 2 shows the Chisel representation of a part of the basic telephone service. Telecordia/BellCor introduced Chisel as a graphical notation to describe telecom features ([AG+98]). Informally, each diagram, as described in [GB+99], represents a behavior of the system as a decision tree, describing a possible course of actions. In following we show some tool-support for service-based specifications using Chisel diagrams. Note that this approach is not Chisel specific. It carries over to other notations describing a course of actions like, e.g., (high level) sequence diagrams or even state-based description of services.

To describe a system with services enhancing the plain ordinary telephone system (POTS), a collection of Chisel diagrams is used. Each diagram describes an additional service by extending the original POTS diagram to describe features like Terminating Call Screening (TCSC) or Call Forward on Busy Line (CFBL):

- Terminating Call Screening (TCSC): A subscribing user can prohibit calls from other users adding their terminals to a Screen List. Calls from screened terminals are not announced at the callee; the caller is informed by a corresponding message.
- Call Forwarding on Busy Line (CFBL): A subscribing user can redirect calls to a third party if a call occurs while his terminal is busy.

However, in general those services are not independent of each other. Thus, when combined to form a complete system description, they influence each other, resulting in feature interaction. In the worst case, the combined services may be incompatible, resulting in an inconsistent specification.

3 Service Formalization

In this section we present a formalization of the notions of service and component, based on the introduced FOCUS model. Furthermore, we apply these notions to the POTS example using relational μ calculus.

3.1 Interfaces, Behaviors, Components, and Services

An *interface* of a system or a service consists of the access points (channels) that are used by the system or service to communicate with its environment. Those access

points are directed (i.e., either *input* or *output* channels) and typed (i.e. have an associated type describing what messages can be sent/received at this channel).

Definition (Interface) An interface (I, O) consists of two disjoint sets of channels I , O that represents the directed input and output channels of the interface. Each channel $(c, M_c) \in I \cup O$ consists of a channel identifier c as well as a set $M_c \subseteq M$ of messages that can flow via the channel.

For reasons of simplicity, in the following we assume an interface has disjoint sets of input and output channel identifiers. Considering the example of the POTS system depicted in the left half of Figure 1, the interface of the switch component contains a input channel with identifier AS and carrying the set of messages $\{OffHook, Dial\ A\ X, OnHook\}$ to component A as well as an output channel with identifier SA carrying messages $\{DialTone, LineBusyTone, Ring\ X, \dots\}$ from component A . To describe this interface information, the FOCUS CASE tool prototype **AutoFOCUS** [HSE97] uses a similar notation (including typing of channels) as shown in the left half of Figure 1. To assign behavior to interfaces, we introduce the notion of a channel history, basically corresponding to a complete observation of messages exchanged over a channel.

Definition (Channel History, Execution, Behavior). Given a set of channel identifiers as well as for each identifier c its type M_c , a channel history is a (total) mapping from each identifier c to a message stream M_c^∞ . The set of all *channel histories* for a given set C of channels is described by \vec{C} . A (*system*) *execution* $e \in \overline{(I \cup O)}$ for a given interface (I, O) consists of a channel history for each channel identifier of the interface. A (*system*) *behavior* for a given interface (I, O) is a set of executions $B \subseteq \overline{(I \cup O)}$.

Since a behavior can also be interpreted as relation (or a set-valued function) between input and output channel histories, we will use the notation $B : \vec{I} \rightarrow \wp(\vec{O})$ in the following. Based on the definition of a behavior, we introduce the notion of a *system/component* as well as a *service*. Since – as shown in Section 3.2 – a system can be broken up into components, we use the terms ‘system’ and ‘component’ synonymously.

Of course, there are different forms of description of behavior. The right side of Figure 2 shows a (incomplete) state-transition-based description of a behavior using a similar notation as in **AutoFOCUS** [HSE97]. Using a state-based description including control state and data variables, e.g., the transition between the *OffHook* and the *Ring* state describes a step where

- in the previous state the data variable *BusyB* has the value false,
- on the input channel *AS* the signal *DialB* is received,
- on the output channel *SA* the signal *AudibleRingB* and the output channel *SB* the signal *RingA* is sent, and
- in the following state the data variable *BusyB* is set to true.

To generate a behavior from such a state-transition-based description, sequences of such steps forming channel and variable assignments are constructed.

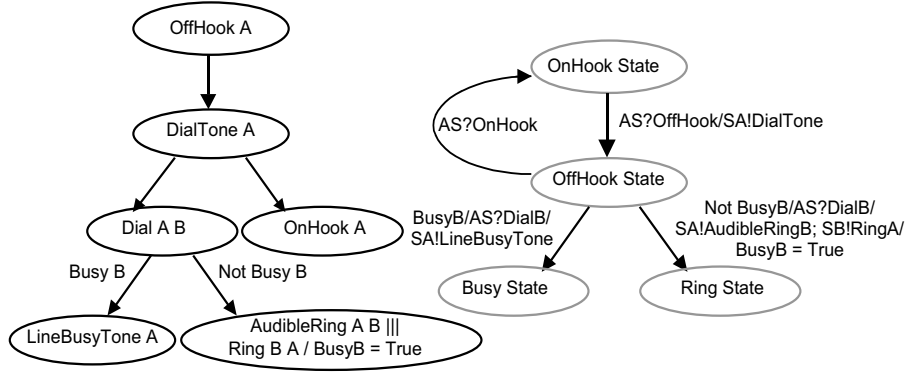


Figure 2: A Chisel Diagram and its State/Transition Representation

A component communicates with its environment via its interface. A component has a *completely specified behavior*: for each behavior of the environment (presenting a history of input messages on the input channels of the component) its reaction (in terms of histories of messages of the output channels) is defined. More formally, this is defined as input completeness or totality, in the following definition.

Definition (Component): A component $c = ((I, O)_c, B_c)$ is defined by its interface $(I, O)_c$ as well as its behavior $B_c : \vec{I} \rightarrow \wp(\vec{O})$. The behavior is input complete, i.e. for each input $i \in \vec{I}$ there exists (at least) one corresponding output $o \in \vec{O}$: $\forall i \in \vec{I}. \exists o \in \vec{O}. o \in B_c(i)$

Since the empty output set corresponds to undefined output behavior, input completeness corresponds to the *totality* of the behavior function. Note that we do not require a component to be deterministic (i.e. $\forall i \in \vec{I}. \exists o \in \vec{O}. o = B_c(i)$). While this is a reasonable requirement for a component to be implemented, on a more abstract level, non-determinism can be helpful when specifying a component.

In contrast to a component, a service behavior needs not be totally defined. For a partial specification, it is possible to have a behavior of the environment where no behavior of the component is defined by the specification. From a formal point of view, we have two different possibilities to deal with those ‘undefined spots’ when constructing a behavior from such a partial specification:

- **“Underspecification = Non-determinism”:** Using this approach, each reaction is considered legal for the ‘undefined spots’ since the specification does not restrict the behavior. As a result, the behavior of the component cannot be distinguished from completely non-deterministic behavior in these spots.

- **“Underspecification = Partiality”**: Here, we explicitly state that no behavior has been defined. As a result, for these inputs from the environment, the empty set of outputs is assigned when constructing a behavior.

Consequently, in the context of services we choose the second interpretation of underspecification and obtain the following definition of a service:

Definition (Service): A service $s = ((I, O)_s, B_s)$ is defined by its interface $(I, O)_s$ as well as its behavior $B_s : \vec{I} \rightarrow \wp(\vec{O})$.

A service describes *partial interaction behavior*. By allowing arbitrary behavior (i.e. partiality), we can use services to describe only a partial behavior offered by a component. This means there are *some inputs* of a service where the specification of the service does *not make any assertion* how the service behaves. According to these definitions a component is also a service, but not the other way round.

Definition (Subinterface, Subinterface Behavior) A (service) interface (I_{s_1}, O_{s_1}) is said to be a *subinterface* of an interface (I_{s_2}, O_{s_2}) if $I_{s_1} \subseteq I_{s_2} \wedge O_{s_1} \subseteq O_{s_2}$, i.e. if the subinterface is a subset (of the channels) of the interface. The behavior $B_{s_1} : \vec{I}_{s_1} \rightarrow \wp(\vec{O}_{s_1})$ is the corresponding subinterface behavior of the service behavior $B_{s_2} : \vec{I}_{s_2} \rightarrow \wp(\vec{O}_{s_2})$ with $B_{s_1} = B_{s_2} \uparrow (I_{s_1}, O_{s_1})$.

3.2 Application: Formalizing Services

The first step of the tool support – besides the modeling of the interfaces and behaviors – consists in supplying a translation of a service description into the μ calculus. Additional information (system interface description, used variables) is needed for a translation as described in [HSE97]. The μ calculus-translator of AutoFocus/Quest [BL+00] can be used to translate the automaton representation of diagrams described by AutoFocus STDs to its μ calculus form. The μ calculus form of the state description is generated from SSD descriptions used by AutoFocus to describe the interface and the data state of a system or component. To formalize a service behavior in a state-based fashion as mentioned in Section 3.1, we use a relation typed according to the interface of the service. In case of the basic telephone service POTS for caller A and callee B, the interface is

$$(\{(AS, In), (BS, In)\}, \{(SA, Out), (SB, Out)\})$$

i.e., input channels with identifiers AS, BS of type In and output channels with identifiers SA, SB of type Out .

Besides the channels, the state of the service is needed as additional argument of the relation. In case of the basic telephone service POTS (for callee B) the type of the control state is $S = \{OnHook\ State, OffHook\ State, Busy, Ring, \dots\}$ and a data state consists of the Boolean variable $BusyB$. Thus this transition relation describing the behavior of the system for one time slot is typed

$$RPOTS_{(A,B)} \subseteq S \times B \times In^\perp \times In^\perp \times Out^\perp \times Out^\perp \times B \times S$$

with B denoting Boolean values, In the input, and Out the output channel type.³ In this simplified example we restrict sending and receiving to at most one message per time slot. Therefore, instead of In^* (or Out^*) we only use In^\perp (or Out^\perp) to describe the message sent/received during a transition.⁴ Furthermore, since this feature considers two parties (initiating/called party), two input and two output lines are needed.

To specify the behavior of a service, we use a notation described in [HSE97], as shown in the right half of Figure 2. To go from *OffHookState* to *Ring* - when A is calling B with B being not busy - we obtain a transition with precondition (*Not BusyB*), input pattern (*AS?Dial B; Dial B* received on channel *AS*), output pattern (*SA!AudibleRing B; SB!Ring A; AudibleRing B* sent on channel *SA* while *Ring B* sent on channel *SB*), and postcondition (*BusyB = True*). While a diagram represents only one call, a system characterized by diagrams accepts an arbitrary sequence of such calls. Thus, the final states of the automaton are identified with initial states, allowing a repetition of a service during system execution. Figure 2 shows an example of such a feedback loop triggered by the *OnHook* signal.

A transition is formalized as the conjunction of pre- and postcondition as well as the channel predicates. Channel predicates are simply the equality between the channel variable and the value assigned to or read from the channel as described by the channel patterns. Thus, for the transition described above with a formal parameter list of

$$State\ C; BusyB\ B; AS\ I; BS\ I; SA\ O; SB\ O; BusyB'\ B; State'\ C$$

we obtain the formalization

$$\begin{aligned} State &= OffHook \wedge \neg BusyB \wedge AS = Dial\ B \wedge \\ SA &= AudibleRing\ B \wedge SB = Ring\ A \wedge BusyB' = True \wedge State' = Busy \end{aligned}$$

with *State* and *State'* denoting the control state as well as *BusyB* and *BusyB'* the data state before and after the transition. The transition relation is constructed via the disjunction of the formalization of each transition. In a state-based description, additionally, the initial state of the transition relation has to be described:

$$Init_{POTS_{(A,B)}}(State, BusyB) \equiv (State = OnHook \wedge BusyB = False)$$

The behavior of a service describes as transition relation is constructed in the usual manner by generating an infinite sequence of transition steps and abstracting from the control and data space. Since in this section we only consider channels transporting at most a single value per time slot, channels and state variables can be treated alike. Thus the associated behavior of a the transition relation can be formalized as

$$\begin{aligned} B_{POTS_{(A,B)}}(\{AS, BS\}, \{SA, SB\}) &\equiv \exists State, BusyB. Init_{POTS_{(A,B)}}(State_0, BusyB_0) \wedge \\ \forall t. R_{POTS_{(A,B)}}(State_t, BusyB_t, AS_t, BS_t, SA_t, SB_t, BState_{t+1}, BusyB_{t+1}) \end{aligned}$$

³ Note that in the example all input channels carry the same messages as the output channels.

⁴ The notation In^\perp is used to describe one-element messages as well as the empty message \perp , i.e. $In^\perp = In \cup \{\perp\}$

4 Combination and Implementation

In the previous section we discussed the differences between services and components. For a service-based engineering process, however, we must relate services to components. Intuitively, a component providing a service reacts as the service on the channels and inputs taken into account by the service. Having introduced the basic notions of service and component, the following questions arise:

- How are components (or services) combined to build composed components (or services)?
- How are services implemented by components?

4.1 Combining Networks

For reasons of simplicity, in the following, we only define binary composition and combination of components or services, resp., which – of course – can be simply generalized to finite sets. *Composition* of components corresponds to combining components by connecting their common channels; Figure 1 shows such a networks consisting of three receivers and a switch including their linking channels.

Definition (Composition): For components c_1, c_2 with interfaces (I_1, O_1) and (I_2, O_2) , resp., we define the *composition* $c_1 \otimes c_2$ to be the stream relation with interface $(I, O) = (I_1 \cup I_2 \setminus (O_1 \cup O_2), O_1 \cup O_2)$ and behavior

$$B_{c_1 \otimes c_2} \equiv \bigcup \{ B \in \vec{I} \mapsto \wp(\vec{O}) \mid B \uparrow (I_1, O_1) \subseteq B_{c_1} \wedge B \uparrow (I_2, O_2) \subseteq B_{c_2} \}$$

given the syntactic compatibility $O_1 \cap O_2 = \emptyset$.

Note that the syntactic compatibility condition $O_1 \cap O_2 = \emptyset$ ensures that the combination of two components again is a component (i.e., a total function). For services, we can also define an analogue notion of a *combination*.

Definition (Combination): For services s_1, s_2 with interfaces (I_1, O_1) and (I_2, O_2) , resp., we define the *combination* $s_1 \oplus s_2$ to be the stream relation with interface $(I, O) = (I_1 \cup I_2 \setminus (O_1 \cup O_2), O_1 \cup O_2)$ and service behavior

$$B_{s_1 \oplus s_2} \equiv \bigcup \{ B \in \vec{I} \mapsto \wp(\vec{O}) \mid B \uparrow (I_1, O_1) \subseteq B_{s_1} \wedge B \uparrow (I_2, O_2) \subseteq B_{s_2} \}.$$

Note that the composition of components is a special case of the combination of services by restricting it to components (complete services) with disjoint output channels. Furthermore note that besides combining them in a network-like fashion, services using the same output channels can also be combined. This form of combination is needed if services with a common subinterface are to be implemented by a single component. Table 1 shows a combination of eight services to be implemented by the switch component of Figure 1; here, e.g., the POTS service for subscriber B (with originator B and further party C) and for subscriber C (with originator C and further party B) both have *AS* and *BS* as input channels and *SA* and *SB* as output channels.

Combination is ‘strict’ in the sense that partial behavior of one service can ‘knock out’ defined behavior of the other service. This is reasonable from a methodical point of view, since we cannot rely on the undefined behavior.

4.2 Implementing Behavior

To implement an abstract component by a more concrete one, we use the notion of behavioral refinement. Basically, behavioral refinement is used to remove non-determinism from a component specification:

Definition (Behavioral Refinement) Given two components c_1, c_2 with the same interface, the behavior of c_1 is said to *refine* the behavior of c_2 , written as $B_{c_1} \leq B_{c_2}$ if c_1 is more deterministic than c_2 . More formally, we require:

$$B_{c_1}, B_{c_2} : \vec{I} \mapsto \wp(\vec{O})$$

$$B_{c_1} \leq B_{c_2} \equiv \forall i \in \vec{I}, o \in \vec{O}. o \in B_{c_1}(i) \Rightarrow o \in B_{c_2}(i)$$

Analogously, c_1 is said to refine c_2 if the behavior of the former is a refinement of the latter.

The refinement relation orders components according to their degree of non-determinism, with the top element being the completely nondeterministic component. Note that behavioral refinement requires respecting the limitations of a component like input closure or the causality restrictions.

To relate components and services, we use the notion of implementation. Intuitively, this corresponds to behavioral refinement extended to services and requiring ‘improved input behavior’:

Definition (Implementation) Given two services s_1, s_2 with the same interface, the behavior of s_1 is said to *implement* the behavior of s_2 , written as $B_{s_1} \prec B_{s_2}$ if s_1 is more deterministic and less partial than s_2 . More formally, we require:

$$B_{s_1}, B_{s_2} : \vec{I} \rightarrow \wp(\vec{O})$$

$$B_{s_1} \prec B_{s_2} \equiv B_{s_1} \leq B_{s_2} \wedge \text{dom}(B_{s_2}) \subseteq \text{dom}(B_{s_1})$$

where

$$\text{dom}(B_S) \equiv \{i \mid \exists o \in \vec{O}. o \in B_S(i)\}$$

Analogously, s_1 is said to implement s_2 if the behavior of the former is an implementation of the latter.

Note that the implementation relation between components corresponds to a refinement relation between them. For a more general version of the notion of implementation we can make use of the notion of subinterfaces. A service with interface (I_{s_1}, O_{s_1}) and corresponding behavior $B_{s_1} : \vec{I}_{s_1} \rightarrow \wp(\vec{O}_{s_1})$ *implements* a service with subinterface (I_{s_2}, O_{s_2}) and behavior $B_{s_2} : \vec{I}_{s_2} \rightarrow \wp(\vec{O}_{s_2})$ if $B_{s_1} \prec_{(I_{s_1}, O_{s_1})} B_{s_2}$ with

Subscriber	Service	Instance: Originator, further parties	Parameters
A	TCSC	B, A	Screen List = {B}
A	TCSC	C, A	Screen List = {B}
A	CFBL	B, A, C	Forward = C
A	CFBL	C, A, C	Forward = C
B	POTS	A, B	-
B	POTS	C, B	-
C	POTS	A, C,	-
C	POTS	B, C	-

Table 1: Service Instantiations for the TCSC/CFBL Interaction

$$B_{s_1} \prec_{(I_{s_1}, O_{s_1})} B_{s_2} \equiv B_{s_1} \leq (B_{s_2} \uparrow (\vec{I}_{s_1}, \vec{O}_{s_1})) \wedge \text{dom}(B_{s_2} \uparrow (\vec{I}_{s_1}, \vec{O}_{s_1})) \subseteq \text{dom}(B_{s_1})$$

Based on this notion of implementation we can define what it means for a component to offer or to require a service.

Definition (Provided Service, Required Service): A service s_1 (or component) is said to *provide a service* s_2 if s_2 is implemented by s_1 . A service s_1 (or component) is said to *require a service* s_2 if the complementary service \bar{s}_2 is implemented by s_1 . A complementary service \bar{s} of a service $s = ((I_s, O_s), B_s)$ is defined by $\bar{s} = ((O_s, I_s), B_{\bar{s}})$ and the behavior $B_{\bar{s}} : \vec{O}_s \rightarrow \wp(\vec{I}_s)$ defined by

$$\forall i \in \vec{I}_s, o \in \vec{O}_s. i \in B_{\bar{s}}(o) \iff o \in B_s(i)$$

If a component c provides and requires a set of services $s_1 \dots s_j$, we demand that these services fulfill additional compatibility constraints. The simplest form of compatibility is called *interface compatibility*, defined by

$$\forall m, n \in \{1, \dots, j\}. m \neq n \Rightarrow (I_m \cap I_n = \emptyset \wedge O_m \cap O_n = \emptyset)$$

This means that the input channels of the services as well as the output channels are pairwise disjoint. Further forms of compatibility are discussed in the following section. Note that formally there is no difference whether a service is provided by an atomic component or by a network of components, since the latter can be substituted by the combined component as defined in Section 4.1.

4.3 Application: Combining Components and Services

As described in Subsection 4.1, from the formal point of view it makes no difference whether we construct a network of components, combine services to create a more complex service, or construct a network of (needed and provided) services. Basically, components and service are combined alike by (interface-adjusted) conjunction of their behaviors. Since in this example services (or rather service schemes) are de-

scribed using parameters like caller, callee, or screen list, these must be instantiated prior to combination. To build a configuration of services, as shown in Table 1, service instances are combined.

As in the case of a single service, we define the specification of the combined services by a transition relation for the complete system. Thus, the type of this relation is defined by the product of

- all variables used by services for each instance (e.g., *BusyA*, *BusyB* and *BusyC* for the terminals A,B, and C), representing the system state prior to the transition
- all input channel variables as defined by the system interface
- all output channel variables as defined by the system interface
- all variables used by services for each instance representing the state after the transition

In a state-based approach, services are defined using input and output channels, and additionally using control and data states. While components can only be combined using communication and thus cannot share their local state space, services support a more general form of combination. Therefore, we additionally allow sharing the state space between services.⁵ Thus, finally, the combination of service instances is simply formed by conjunction of the service transition relation instantiated with the necessary parameters (Screen List, e.g.) and applied to the corresponding elements of the system state and adjusted according to their interfaces. On the automaton level this is equivalent to constructing the product automaton.

5 Methodical issues

From a methodical point of view, the following questions remain when building a component specification from a collection of service specifications:

Which services are (reasonably) combinable?

What makes a service a component?

In this section we investigate how the explicit representation of the domain of a service helps in combining services and constructing complete component behavior. We introduce the notion of *consistency* of a collection of services (basically stating that a collection can be implemented by a component) as well as the notion of *completeness* of a service (stating that each action of the environment is accounted for by a reaction of the service). Since completeness and consistency are essential prerequisites for the robustness and reliability of safety-critical systems, the validation of these properties is a decisive aspect of a service-based development process.

5.1 Completeness

Informally, a service is said to be complete, if its behavior is already detailed enough to define a component (ignoring some causality aspects).

⁵ Note that in this example all service instances make only use of non-overlapping parts of the state space.

Definition (Completeness) A service $s = ((I, O)_s, B_s)$ with interface $(I, O)_s$ as well as behavior $B_s : \vec{I} \rightarrow \wp(\vec{O})$ is called *complete*, if $\text{dom}(B_s) \equiv \vec{I}$.

To form a component specification out of a (combined) service description, the service description must be extended from a partial to a total specification. From a methodical point of view, different forms of canonical completions \hat{s} are possible for a service s . Three prominent examples are:

- *Chaotic completion*: If the service exhibits some undefined behavior at some time point, in the completion any behavior is possible afterwards.
- *Operational completion*: If the service exhibits some undefined behavior at some time point, in the completion the empty output is produced at that time point.
- *Error completion*: If the service exhibits some undefined behavior at some time point, it will produce an error message.

Chaotic completion is associated with a loose interpretation of a specification. Basically, we use a kind of Assumption-Commitment scheme to construct components from services by chaotic completion (c.f. [SDW93]). The domain part of a service forms the assumption; the service specification forms the commitment part. However, since for a given service behavior $B_s : \vec{I} \rightarrow \wp(\vec{O})$, the total function defined by

$$B_{s_{chaos}}(i) \equiv \begin{cases} B_s(i) & \Leftarrow i \in \text{dom}(s) \\ \vec{O} & \Leftarrow i \notin \text{dom}(s) \end{cases}$$

generally is not a causality-respecting function, a component for this service is defined by the behavior $\hat{B}_{s_{chaos}}$ as defined in Section 2.2.

Operational completion is associated with an operational interpretation; it is, e.g., used in [HS01]. As mentioned in the following subsection, completions can be schematically constructed in the μ calculus.

5.2 Application: Detecting Incompleteness

According to our definition of service, the formalization of a service makes explicit the part where no behavior is defined by the description of the service. This explicit representation of the domain of a service (or a combination of services) can be exploited when checking for completeness. When translating the definition of completeness to our μ calculus based description of a service as shown in the previous subsection, a service is incomplete if it has a non-input closed relation. Checking for input-enabledness requires the calculation of all reachable states of the transition. A service is not input enabled if either its initial states relation is unsatisfiable or its transition relation is not input-enabled. Using the above formalization, in the first case we have

$$\forall s, v. \neg \text{Init}(s, v)$$

stating that there is no defined initial (control or data) state, thus leading to the completely undefined behavior. In the second case, we have to check whether there is a reachable state that does not define an output or a successor (control or data) state for a given input. With the set of reachable states used to mark those trace positions identified by s , incompleteness of a transition relation in terms of the μ calculus for transition relation R is

$$\begin{aligned} & \exists s, v. Reach(s, v) \wedge \exists i_1, \dots, i_m. \forall o_1, \dots, o_n, v', t. \\ & \neg R(s, v, i_1, \dots, i_m, o_1, \dots, o_n, v', t) \end{aligned}$$

where *Reach* denotes the set of reachable states defined by

$$\mu Reach(s) \equiv Init(s) \vee \exists i_1, \dots, i_m, o_1, \dots, o_n, t. Reach(t) \wedge R(t, i_1, \dots, i_m, o_1, \dots, o_n, s)$$

with μ denoting the least fixed point used as interpretation of this recursive definition and *Init* the set of initial states.

Note that this definition does not exclude nondeterministic behavior if the nondeterministic behavior is explicitly stated by the service specification. In early phases nondeterminism is often introduced into service specifications by abstracting from internal aspects of a system: an ATM may seemingly nondeterministically provide a customer with cash if abstracting from the current balance of the account. Therefore, from a methodical point of view it is necessary to support both nondeterminism and underspecification but to distinguish between them.

When applying the approach to the example of the POTS, the model checker can detect several incompletenesses. E.g., in the specification is not defined which reaction of the system should occur if the callee picks up the phone in the same instance the caller dials his number.

As mention in Section 5.1, an incomplete service specification can be transformed into a (complete) system behavior by adding a behavior for each possible input sequence to the set of executions of the service. This canonical transformation can also be carried out on the level of the transition relation: simply adding a transition with arbitrary output and successor state for each undefined input to the transition relation leads to a highly nondeterministic relation. To support a more operational interpretation for partial behavior, as, e.g., used in [HS01], undefined output is substituted by *nil* modeling that no signal is send; undefined successor control or data states are defined to remain unchanged. Again, such a canonical transformation can be easily defined on the level of the transition relation.

5.3 Consistency

An important issue when dealing with the combination of services is the problem of service interaction. While each service exposes the intended behavior if used separately, when combined unforeseen interaction patterns may show up.

The combination of services is linked to the concept of compatibility of services: A compatible collection of services can be combined without exhibiting unwanted behavior. Therefore, a notion of compatibility of services is needed to combine services to form a component. In formal approaches dealing with the combination of features or services (e.g., [KB00] or [Sch02]), compatibility of services is defined in terms of consistency of the (specifications of) services. As usual, services are considered to be consistent if the specifications are not contradicting.

Definition (Consistency): Two services s_1, s_2 with interfaces $(I_1, O_1), (I_2, O_2)$ and $dom(B_{s_1}) \subseteq dom((B_{s_1} \oplus B_{s_1}) \uparrow (I_{s_1}, O_{s_1})) \wedge dom(B_{s_2}) \subseteq dom((B_{s_1} \oplus B_{s_1}) \uparrow (I_{s_2}, O_{s_2}))$ are called *consistent*.

From a methodical point of view, consistency of services can be interpreted by the fact that the combined service is not more restricted than each of the services. The immediate methodical consequence is, that the combination of consistent services is an implementation of each service:

$$s_1, s_2 \text{ consistent} \Rightarrow ((s_1 \oplus s_2) \uparrow (I_1, O_1) \prec s_1 \wedge (s_1 \oplus s_2) \uparrow (I_2, O_2) \prec s_2)$$

In the development process this corresponds to the fact that a consistent service can safely be added to a system without leading to unexpected results.

Consistency of services describes the most general intuitive form of compatibility, however rather sophisticated notion to be checked. For practical use, simpler forms of compatibility (ensuring consistency) are useful, e.g.:

- *Syntactic consistency*: This form required the disjointness to the output channels of services. For services s_1, s_2 , their output interface has to be disjoint, i.e. $O_{s_1} \cap O_{s_2} = \emptyset$. While this form is very restricted, it has the advantage that compatibility is guaranteed by a design rule that can be check syntactically.
- *Pointwise consistency*: If service behavior is described in a state-based fashion, consistency of services boils down to consistency of their transition relations. Section 5.4 treats this form of consistency in more detail.

Note that there is a simple relation between consistency of services and the completeness of the combination of their chaos-completions:

$$s_1, s_2 \text{ consistent} \iff s_{1\text{Chaos}} \oplus s_{2\text{Chaos}} \text{ complete}$$

This due to the fact that (completely) chaotic behavior is a ‘neutral element’ when combining services

5.4 Application: Detecting Inconsistency

As introduced in Subsection 5.3, we define a service interaction problem to occur if the behavior defined by the combined services is inconsistent. Therefore, a service configuration is considered to expose a service interaction if the service instances are contradictory for at least one behavior of the environment. Therefore we have to check whether there exists an input channel history that has no behavior assigned by the combined services but has a behavior assigned by the services in isolation.

Instead of checking this property directly, we check for the completeness of the chaotic completion of the services of the configuration, as noted in Subsection 5.2. Thus, services exhibit an interaction problem if their chaotic completions have a non-input closed relation. In the relational μ calculus, chaotic completion CR of a transition relation R can simply be constructed by

$$CR(s, v, i_1, \dots, i_m, o_1, \dots, o_n, w, t) \equiv$$

$$R(s, v, i_1, \dots, i_m, o_1, \dots, o_n, w, t) \vee \forall o_1', \dots, o_n', w', t'. \neg R(s, v, i_1, \dots, i_m, o_1', \dots, o_n', w', t')$$

Using suitable blocking (starting with the input variables) and interleaving when ordering the argument variables, the corresponding OBDD representation of the chaotic completion is of the same complexity.

If services are described using shared variables, a possible source of interaction is the assignment of values to variables. If two services assign different values to a shared variable, there is no interpretation for this assignment. Similar observations hold for communication actions, modeled by values assigned to channels. Note that feature interaction may as well occur combining two instantiations of the same feature (in case of the POTS specification, e.g., the service behaves differently if the initiator is being called prior to service start) as with the combination of two different features like call forwarding and call blocking.

Since this example uses simple signals the system has a finite state space accessible to symbolic verification. Using a schematically generated variable order, OBDDs representing the transition relations can be kept sufficiently small to allow the generation of the set of reachable states. The check for an interaction problem is carried out using the relational μ calculus symbolic model checker μ cke ([Bie97]). Here, all reachable states of the system transition starting from the initial state are checked for input actions missing an appropriate transition. Again, the μ calculus term can be generated schematically from interface and data space of the system. If such a missing transition is found, the counterexample component of μ cke is used to generate execution traces leading to system states with conflicting feature requirements including the input actions that have no defined output actions.

Combining TCSC and CFBL as described by Table we obtain a simple four-step execution trace leading to a feature interaction problem: If user A subscribes to both CFBL with Forward C and TCSC with a Screen List containing B, what happens if B calls A while A is busy? Should B be forwarded resulting in a ring tone played to B or screened resulting in a screening message played to B?

μ cke generates a counterexample given by assignments for the variables and channels of the system for each execution step starting from the initial state of the system and ending with the input action generating the conflict. However, μ cke generates a μ calculus tableau representation. For a transparent use of this formalization, a different representation of the counterexample is needed, for example, in an MSC-like form. Figure 1 shows such a visualization of the counterexample for the TCSC/CFBL conflict.

6 Conclusion

In this article we introduced a formal definition of the notion *service* and related it to formal model of interacting components. Essential aspects of our service notion are its partiality and its separation between functionality and structure. The introduction of different notions of compositionality and combination lays a first foundation for a methodical service-based development process. To support practical application, the formal model is used transparently to the engineer: for the specification of a service (or component) intuitive description techniques like state transition diagrams or sequence diagrams are used; for suitable systems, mechanized techniques like model checking are used to ensure consistency; finally, generic completion techniques are applied to transform the partial behavior of a service into a complete component behavior.

6.1 Service-Based Development Process

Besides defining services, their combination, and their implementation by a component network, the main advantage of the formalization of services is that it enables the support of services as a specification technique in a development process. In the following we give a brief sketch out how a possible process could be structured and what advantages services would bring to the systematic development of software systems.

Services allow modelling of the functionalities of the system without tailoring the structure of the system at the beginning of the development process. They can be extracted from very abstract specifications of the system functionality, for example UML use cases, which do not narrow the later structure of the system either.

After modelling the service functionality – using Sequence Diagrams or a state-based description - the different services can be deployed to component networks by composing services into a component type. During this step, structural attributes are added to the system specification. Since a service specification can be realized by many component networks, different architectures are possible.

As shown in the previous sections, certain aspects of such a deployment step (like checking from completeness or consistency) can be automated. Here, the precise notion of services and components and their composition helps to avoid problems like unwanted service interaction as described above.

A service based development process makes use of the separation of functionality and structural architecture of a system using the following steps:

1. Specification of use cases (i.e. UML)
2. Encapsulation of services out of the use cases
3. Modeling of the service architecture of the system (without defining a technical structure)
4. Deployment of the services on a given set of components
5. Choice of realization and alternatives.

Note that until step four the system is specified without fixing its structure. The result of the deployment is a *set* of various possible realizations that offer the same functionality but differ in the structure (e.g. which component provides which service). So structural issues can be delayed to latter steps in the development process that gives the development more flexibility and better applicability.

6.2 Related Work

As mentioned in Section 1, a precise as well as abstract definition of service has not been widely addressed so far; most work in this area is focused on implementational issues. However, several aspects of services and their methodical treatment have been addressed in other approaches.

The description of partial behavior is, e.g., addressed in the context of description techniques like Message Sequence Charts (MSC) [Krü00]. Here, however, rather the interpretation of sequence diagrams is discussed; the issue of combining partial descriptions of a system is not in the main focus.

As mentioned above, the combination of partial descriptions to form a complete specification is closely related to the problem of feature interaction. Since this prob-

lem has been studied carefully in the context of telecommunication, especially the issue of consistent specifications has been addressed there ([KB+98], [CM00]). However, besides being more complex, those approaches generally add implementation details or other restrictions. Furthermore, the issue of the incompleteness of combined service descriptions is treated here, which is generally not considered in the other approaches.

6.3 Outlook

Services allow modeling of the functionalities of the system without tailoring the structure of the system at the beginning of the development process. They can be extracted from very abstract specifications of the system functionality, for example UML use cases combined with sequence diagrams, which do not narrow the later structure of the system either. After modeling the service functionality the different services can be deployed to component networks by composing services into a component type, adding structural information to the system specification. Furthermore, services allow to structure complex functionalities of a system to modularize its specification. Using consistent composition, services can be used to support reuse on a behavioral rather than an architectural level, enabling a shift from component-based to service-based development. Validation of completeness and consistency makes this approach suitable even for safety-critical systems.

However, several issues – specific to a service-based development process in certain application domains – were not addressed here, e.g.:

- Dynamic networks: How is the dynamic change concerning an interface of a component as well as the services implemented by a component described in the semantic model? While [GS96] gives a general outline, more conceptual support is needed for an application of the approach in this domain.
- Service properties: How are aspects concerning Quality of Service (e.g., response times, loss of signals) incorporated in the semantic model? While the basic model is capable of expressing such aspects like time constraints, special analysis techniques for these properties like [KS01] can be supplied to reduce the complexity of the analysis.

For other domains focusing on the issues of modular description of behavior and the related aspects of completeness and consistency (e.g., in the automotive domain), the results here supply a suitable basis for a methodical development process. Current research is directed on integrating this approach in a support tool for the development of embedded automotive software.

7 References

- [AG+98] Aho, A. Gallagher, N. Griffeth, N. Schell, C. Swayne D. *SCF3/Sculptor with Chisel*. In: Kimbler, K. et al. (eds.) Proc. 5th Feature Interactions in Telecommunications and Software Systems. IOS Press, 1998.
- [AH99] Alur, R. Henzinger, T. *Reactive Modules*. In: Formal Methods in Systems Design. 1(15). Kluwer Academic Publishers, 1999.

- [Bie97] Biere, A. *Effiziente Modellprüfung des μ -Kalküls mit binären Entscheidungsdiagrammen*. Ph.D. Thesis. Universität Karlsruhe, 1997.
- [BL01] Berners-Lee, T. *Are we done yet?*. <http://www.w3.org/2001/Talks/0501-tbl/>
- [BS01] Broy, M. Stoelen, K. *Specification and Development of Interactive Systems*. Springer, 2001.
- [CCM+] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana: *Web Services Description Language (WSDL) 1.1*, W3C Note <http://www.w3.org/TR/wsdl>
- [CM00] Calder, M. et Magill, E. (eds.) Proc. 6th Feature Interactions in Telecommunications and Software Systems. IOS Press, 2000.
- [GB+99] Griffeth, N. et al. Feature Interaction Detection Contest. Instructions. <http://www-db.research.bell-labs.com/user/nancyg/instructions.ps>, 1999.
- [GS96] Grosu, R. Stoelen, K. *Specification of Dynamic Networks*. In: Proceedings of the 8th Nordic Workshop on Programming Theory, Oslo, Norway. University of Oslo, 1996.
- [HS99] Huber, F. Schätz, B. *Integrating Formal Description Techniques*. In: FM'99 -- Formal Methods, Volume II. Wing, J. et al. (eds.). Springer, 1999.
- [HSE97] Huber, F. Schätz, B. Einert, G. *Consistent Graphical Specification of Distributed Systems*. In: Fitzgerald, J. et al. (eds.) Proceedings of FME'97. LNCS Vol. 1313. Springer, 1997.
- [JZ00] Jackson, M. Zave, P. *New Feature Interactions in Mobile and Multimedia Telecommunication Services*. In: Calder, M. et al. (eds.) Proc. 6th Feature Interactions in Telecommunications and Software Systems. IOS Press, 2000.
- [KB00] Koshumi, A. Bevelo, R.J. *A Detection Method Developed after A Thorough Study of the Contest Held in 1998*. In: Calder, M. et al. (eds.) Proc. 6th Feature Interactions in Telecommunications and Software Systems. IOS Press, 2000.
- [KB98] Kimbler, K. et Bouma, L. (eds.) Proc. 5th Feature Interactions in Telecommunications and Software Systems. IOS Press, 1998.
- [Krü00] Krüger, I. *Distributed System Design with Message Sequence Charts*, Dissertation, Technische Universität München, 2000.
- [KS01] Logothetis, G. Schneider, K. *Symbolic Model-Checking of Real-Time Systems*. In: Eighth International Symposium on Temporal Representation and Reasoning (TIME'01). IEEE Computer Society, 2001.
- [KS03] Kof, L. Schätz, B. *Dimensions of Design: Combining Aspects of Reactive Systems*. In: Perspectives Of System Informatics (5th Andrei Ershov International Conference). LNCS Springer, 2003.
- [Lam93] Lamport, L. *Specification and Verification of Concurrent Programs*. In: de Bakker, J.W. et al. (eds.). A Decade of Concurrency. LNCS Vol. 803. Springer, 1993.
- [LT89] Lynch, N. Tuttle, M. *An Introduction to Input/Output Automata*. CWI Quaterly 3(2). 1989.
- [Müh96] Mühlhäuser, M. (Ed.): *Special Issues in Object-Oriented Programming* – Proceedings of WCOP 96, dpunkt Verlag, Heidelberg, Germany, 1997
- [Oas02] OASIS Web Page <http://www.oasis-open.org/committees/wscm>
- [Sal02] Christian Salzmann *Modellbasierter Entwurf spontaner Komponentensysteme*. PhD Thesis Technische Universität München 2002.
- [Sch02] Schätz, B. *Towards Service-Based Systems Engineering: Formalizing and mu-Checking Service Specifications*. Tech. Report TUMI-0602, TU München, 2002
- [SDW93] Stoelen, K. Dederichs, F. Weber, R. *Assumption/Commitment Rules for Networks of Asynchronously Communicating Agents*. Technical Report TUM-I9303, TU München, 1993.