Behavioral Constraints for Services

Niels Lohmann[1,⋆], Peter Massuthe[1], and Karsten Wolf[2]

[1] Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany
{nlohmann,massuthe}@informatik.hu-berlin.de
[2] Universität Rostock, Institut für Informatik,
18051 Rostock, Germany
karsten.wolf@informatik.uni-rostock.de

Abstract. In service-oriented architectures (SOA), deadlock-free interaction of services is an important correctness criterion. To support service discovery in an SOA, *operating guidelines* serve as a structure to characterize all deadlock-freely interacting partners of a services. In practice, however, there are *intended* and *unintended* deadlock-freely interacting partners of a service. In this paper, we provide a formal approach to express intended and unintended behavior as *behavioral constraints*. With such a constraint, unintended partners can be "filtered" yielding a customized operating guideline. Customized operating guidelines can be applied to validate a service and for service discovery.

Keywords: Business process modeling and analysis, Formal models in business process management, Process verification and validation, Petri nets, Operating guidelines, Constraints.

1 Introduction

Services are an emerging paradigm of interorganizational cooperation. They basically encapsulate self-contained functionalities that interact through a well-defined interface. A service can typically not be executed in isolation — services are designed for being invoked by other services or for invoking other services themselves. *Service-oriented architectures* (SOA) [1] provide a general framework for service interaction. Thereby, three roles of services are distinguished. A *service provider* publishes information about his service to a public repository. A *service broker* manages the repository and allows a *service requester* (also called client) to find an adequate published service. Then, the provider and the requester may bind their services and start interaction.

In [2,3] we introduced the notion of an *operating guideline* (*OG*) of a service as an artifact to be published by a provider. The operating guideline OG_{Prov} of a service *Prov* characterizes all requester services *Req* that interact deadlock-freely with *Prov*. Operating guidelines therefore enable the broker to return only those published services *Prov* to a querying *Req* such that their interaction

⋆ Funded by the BMBF project "Tools4BPEL".

G. Alonso, P. Dadam, and M. Rosemann (Eds.): BPM 2007, LNCS 4714, pp. 271–287, 2007.

is guaranteed to be deadlock-free. Additionally, an operating guideline is an operational description and can therefore be used to *generate* a deadlock-free interacting service *Req*.

In practice, there are *intended* and *unintended* clients for a service. For example, an online shop is intendedly designed for selling goods to its customers. After an update of its functionality, it might introduce the possibility to abort the ordering at any time. The *OG* of such a shop would now also characterize customers that abort after the first, second, etc. step of the ordering process. These interactions with aborting partners are deadlock-free. However, the owner of the shop is interested whether it is still possible to actually purchase goods in his shop.

On the other hand, a service broker might classify provider services as intended or unintended. For example, he may want to assure certain features, like payment with certain credit cards only. Finally, a client requesting for a travel agency might want to exclude going by train and thus is only interested in flights. Even more involved, he could prefer arranged communication such that certain actions occur in a given order (first hotel reservation, then flight booking, for instance).

In this paper we study *behavioral constraints* (constraints for short) that have to be satisfied in addition to deadlock freedom. We provide a formal approach for steering the communication with *Prov* into a desired direction and extend operating guidelines to *customized* operating guidelines. A customized *OG* of *Prov* characterizes all those services *Req* that communicate deadlock-freely with *Prov* satisfying a given constraint.

We identified four scenarios involving behavioral constraints.

1. *Validation.* Before publishing, the designer of a service *Prov* wants to check whether a certain feature of *Prov* can be used.
2. *Restriction.* A specialized repository might require a certain constraint to be fulfilled by published services. To add a service *Prov* to this repository, its behavior might have to be restricted to satisfy the constraint.
3. *Selection.* For a service *Req*, the broker is queried for a matching provider service *Prov* satisfying a given constraint.
4. *Construction.* A requester does not have a service yet, but expresses desired features as a constraint. The broker returns all operating guidelines providing these features. With this operational description, the requester service can then be constructed.

In the first two scenarios, the operational description — in this paper given as a Petri net — of *Prov* itself is available. This has the advantage that constraints are not restricted to communication actions but may involve internal behavior of the service. This way, a service can, for instance, be customized to legal requirements (publish, for example, an operating guideline where only those partners are characterized, for which the internal action "add added value tax" has been executed). In contrast, in the last two scenarios, a customized operating guideline is computed from a given general operating guideline of *Prov*, without having access to an operational description of *Prov* itself.

In this paper, we propose solutions for all four scenarios. That is, we show how to compute customized operating guidelines (a) from a given operational description of *Prov* itself, or (b) from a given general operating guideline.

It is worth being mentioned that our approach is fully residing on the behavioral (protocol) level. That is, we abstract from semantic as well as nonfunctional issues. Our approach is not meant to be a competitor to those approaches but rather a complement. In fact, a proper treatment of semantic discrepancies between services is a prerequisite of our approach, but does not replace the necessity to send and receive messages in a suitable order. Policies and nonfunctional criteria can be integrated into our approach as far as they can be reduced to behavioral constraints. They are, however, not the focus of this paper. We chose deadlock freedom as the principle notion of "correct interaction". There are certainly other correctness criteria which make sense (for instance, additional absence of livelocks) but our setting is certainly the most simple one and part of any other reasonable concept of correctness. Thus, our setting can be seen as an intermediate step towards more sophisticated settings.

The rest of this paper is organized as follows. In Sect. 2, we set up the formal basis for our approach and introduce an online shop as running example throughout the paper. Section 3 is devoted to implement constraints into the operational description of a service. Our solutions to compute a customized operating guideline from a general one are presented in Sect. 4. We conclude with a presentation of related work in Sect. 5 and a discussion of future work in Sect. 6.

2 A Formal Approach to SOA

Our algorithms are based on open workflow nets (oWFNs) [4] and descriptions of their behavior (service automata [2]). Suitability of oWFNs for modeling services has been proven through an implemented translation from the industrial service description language WS-BPEL [5] into oWFNs [6]. Service automata form the basis of our concept of an operating guideline. The presentation of our constructions on this formal level simplifies the constructions and makes our approach independent of the evolution of real-world service description languages. As our approach is to a large extend computer-aided, the formalisms can, however, be hidden in real applications of our methods.

2.1 Open Workflow Nets

Open workflow nets (oWFNs) are a special class of Petri nets. They generalize the classical workflow nets [7] by introducing an interface for asynchronous message passing. oWFNs provide a simple but formal foundation to model services and their interaction.

We assume the usual definition of (place/transition) Petri nets. An *open workflow net* is a Petri net $N = [P, T, F]$, together with an *interface* $P_i \cup P_o$ such that $P_i, P_o \subseteq P$, $P_i \cap P_o = \emptyset$, and for all transitions $t \in T$: $p \in P_i$ (resp. $p \in P_o$) implies $(t, p) \notin F$ (resp. $(p, t) \notin F$); a distinguished marking m_0, called the *initial*

marking; and a distinguished set Ω of *final markings*. P_i (resp. P_o) is called the set of *input* (resp. *output*) places.

We require that the initial or a final marking neither marks input nor output places. We further require that final markings do not enable a transition. A nonfinal marking that does not enable a transition is called *deadlock*.

Throughout this paper, consider an online shop as running example. An open workflow net N_{shop} modeling this online shop is depicted in Fig. 1. The only final marking is [final]; that is, only place final is marked. Though the online shop is a small toy example, it allows to demonstrate the results of this paper.

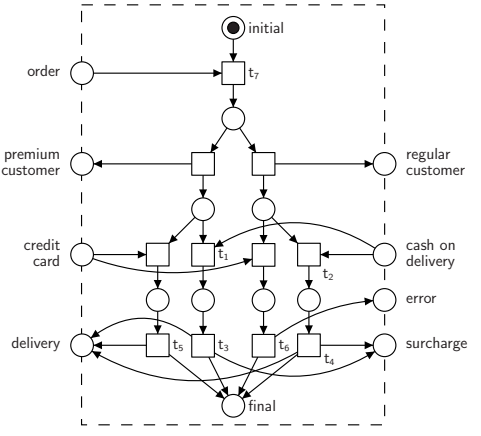


Fig. 1. An oWFN N_{shop} modeling an online shop. It initially receives an order from a customer. Depending on the previous orders, the customer is classified as premium or regular customer. Premium customers can pay with credit card or cash on delivery, whereas regular customers can only pay cash on delivery: If a regular customer tries to pay with credit card, the online shop will respond with an error message. Otherwise, a delivery notification — and in case of cash on delivery payment a surcharge notification — is sent.

Open workflow nets — like common Petri nets — allow for diverse analysis methods of computer-aided verification. The explicit modeling of the interface further allows to analyze the communicational behavior of a service [6,8].

The interaction of services is modeled by the composition of the corresponding oWFN models. For composing two oWFNs M and N, we require that M and N share interface places only. The composed oWFN $M \oplus N$ can then be constructed by merging joint places and merging the initial and final markings. All interface places of M and N become internal to $M \oplus N$.

For a given oWFN *Prov* of a service provider we are particularly interested in the set of oWFNs *Req* of service requesters, such that $Req \oplus Prov$ is deadlock-free. Each such *Req* is called a *strategy* for *Prov*. We write $Strat(Prov)$ to denote the set of strategies for *Prov*. The term strategy originates from a control-theoretic

point of view (see [9,10], for instance): We may see *Req* as a controller for *Prov* imposing deadlock freedom of $Req \oplus Prov$.

As an example, consider a client of our online shop firstly placing an order and then receiving either premium customer or regular customer. In any case, he pays cash on delivery and then receives the surcharge notice and the delivery. Obviously, the described client is a strategy for N_{shop}.

In this paper, we restrict ourselves to oWFNs with bounded state space. That is, we require the oWFNs M, N, and $M \oplus N$ to have finitely many reachable markings.

2.2 Service Automata

In the following, we recall the concepts of *service automata* and *operating guidelines*, which were introduced in [2] and generalized in [3]. Operating guidelines are well-suited to characterize the set of all services *Req* for which $Req \oplus Prov$ is deadlock-free. Since absence of deadlocks is a behavioral property, two oWFNs which have the same behavior but are structurally different have the same strategies. Thus, we may refrain from structural aspects and consider the behavior of oWFNs only. Service automata serve as our behavioral model.

A *service automaton* $A = [Q, I, O, \delta, q_0, \Omega]$ consists of a set Q of states; a set I of input channels; a set O of output channels, such that $I \cap O = \emptyset$; a nondeterministic transition relation $\delta \subseteq Q \times (I \cup O \cup \{\tau\}) \times Q$; an initial state $q_0 \in Q$; and a set of final states $\Omega \subseteq Q$ such that $q \in \Omega$ and $(q, x, q') \in \delta$ implies $x \in I$. A service automaton is finite if its set of states is finite.

As an example, the service automaton modeling the described client of the online shop is depicted in Fig 2.

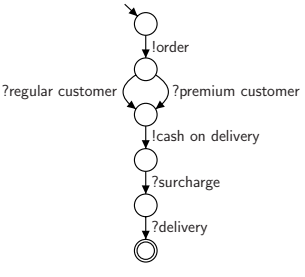


Fig. 2. An automaton describing a strategy for the online shop. The client first sends an order message to the shop, then either receives the regular customer or the premium customer message. In either case he decides for cash on delivery, receives the surcharge note, and finally waits until he receives his delivery. As a convention, we label a transition sending (resp. receiving) a message x with !x (resp. ?x).

The translation of an oWFN into its corresponding service automaton is straightforward [3]. Thereby, we consider the *inner* of the oWFN, easily constructed by removing the interface places and their adjacent arcs, and compute its reachability graph. Transitions of the oWFN that are connected to interface

places correspond to transition labels of the service automaton. Similarly, a service automaton can be retranslated into its corresponding oWFN model, using the existing approach of region theory [11], for instance.

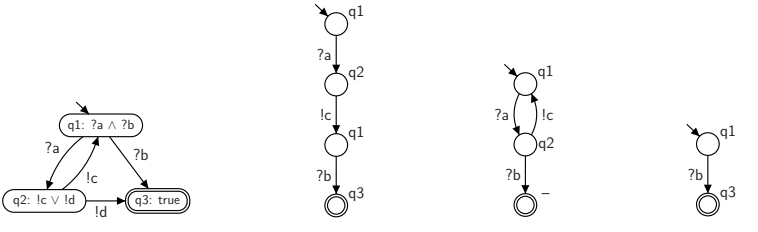
The composition of two service automata is the service automaton $A \oplus B$ where the shared interface channels become internal. To reflect our proposed model of asynchronous communication, a state of $A \oplus B$ is a triple of a state of A, a state of B, and a multiset of currently pending messages (see [3] for details). The notions of deadlocks and strategies can be canonically extended from oWFNs to service automata.

2.3 Operating Guidelines

Given a service automaton A, consider now a function Φ that maps every state q of A to a Boolean formula $\Phi(q)$. Let the propositions of $\Phi(q)$ be labels of transitions that leave q in A. Φ is then called *annotation* to A. An *annotated automaton* is denoted by A^Φ. We use annotated automata to represent sets of automata. Therefore, we need the concept of *compliance* defined in Def. 1.

Let A and B be two service automata. Then, $R_{(A,B)} \subseteq Q_A \times Q_B$, the *matching relation of A and B*, is inductively defined as follows: $(q_{0_A}, q_{0_B}) \in R_{(A,B)}$. If $(q_A, q_B) \in R_{(A,B)}$, $(q_A, x, q'_A) \in \delta_A$ and $(q_B, x, q'_B) \in \delta_B$, then $(q'_A, q'_B) \in R_{(A,B)}$. Let furthermore β_{q_A} denote an assignment at state q_A of A that assigns *true* to all propositions x for which there exists a transition $(q_A, x, q'_A) \in \delta_A$ and *false* to all other propositions.

Definition 1 (Compliance). *Let A be a service automaton, let B^Φ be an annotated service automaton and let $R_{(A,B)}$ and β be as described above. Then, A* complies to *B^Φ iff for every state $q_A \in Q_A$:*
- *there exists a state $q_B \in Q_B$ with $(q_A, q_B) \in R$, and*
- *for every state $q_B \in R(q_A)$ holds: β_{q_A} satisfies the formula $\Phi(q_B)$.*



(a) A^Φ (b) B (c) C (d) D

Fig. 3. (a) An annotated service automaton A^Φ. The annotation $\Phi(q)$ is depicted inside a state. (b)–(d) Three service automata B, C, and D. A state q of A^Φ attached to a state s of B, C, or D represents the element (q, s) in the corresponding matching with A^Φ. Since the final state of C has no matching state in A, C does not comply to A^Φ. D does not comply to A^Φ, because it violates the annotation $\Phi(\mathtt{q1})$ in D's initial state.

Let $Comply(B^\Phi)$ denote the set of all service automata that are compliant to B^Φ. This way, the annotated automaton B^Φ characterizes the set $Comply(B^\Phi)$ of service automata.

As an example, Fig. 3(a) shows an annotated service automaton A^Φ. The service automaton B from Fig. 3(b) complies to A^Φ whereas the automata C and D (cf. Fig. 3(c) and 3(d)) do not comply to A^Φ.

Finally, the operating guideline OG_{Prov} of a service *Prov* is a special annotated service automaton that represents the set $Strat(Prov)$ of strategies for *Prov*.

Definition 2 (Operating guideline). *Let A be a service automaton. An annotated service automaton B^Φ with $Comply(B^\Phi) = Strat(A)$ is called* operating guideline of A, *denoted OG_A.*

In [3], we presented an algorithm to compute operating guidelines for finite-state service automata. If there is no single strategy for a service *Prov*, then OG_{Prov} is empty. In that case, *Prov* is obviously ill-designed and has to be corrected. In [6], we demonstrated that even very small changes of a service *Prov* can have crucial effects on the set of strategies for *Prov*. The calculation of operating guidelines is implemented in the tool Fiona[1], giving the designer of services the possibility to detect and repair errors, that would have been hard or impossible to find manually.

As an example, Fig. 4 depicts the operating guideline for our online shop. Obviously, there are some interleavings in which an error message is received by the client (?e). These interleavings describe deadlock-free interactions though cannot be regarded as successful interactions by the owner of the online shop. Constraints can help to, for example, exclude this unwanted behavior.

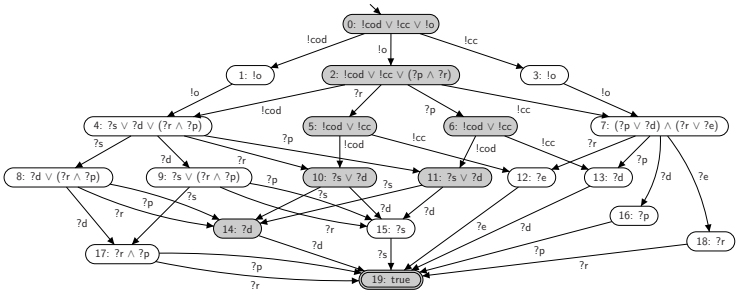


Fig. 4. The operating guideline OG_{shop} of the online shop. For reasons of space we abbreviated message names by its first letter (!o means !order, for instance). !cod abbreviates !cash on delivery, !cc means !credit card. It is easy to see that the client of Fig. 2 complies to OG_{shop}. The matching involves the highlighted states of OG_{shop}.

We propose to use operating guidelines as an artifact generated by the owner of a provider service *Prov* to be published to the service broker. The broker can

[1] Available at `http://www.informatik.hu-berlin.de/top/tools4bpel`

then check whether or not a given requester service *Req* will have deadlocks with *Prov* even before actually plugging them together.

In Sect. 4, we will use the operating guideline for *Prov* to characterize all requester services for which their composition with *Prov* satisfies a given constraint, thus realizing the third and the fourth scenario from the introduction.

3 Adding Constraints to Open Workflow Nets

As stated in the introduction, we aim at putting constraints on the *behavior* of two oWFNs *Prov* and *Req* in their interaction. Therefore we consider the notion of a *run* of an oWFN N: A run of N is a transition sequence $t_0 \dots t_n$ starting in the initial marking of N and ending in a final marking of N.

We distinguish two effects of constraints: *exclusion* of unwanted behavior and *enforcement* of desired behavior. Using oWFN service models, these effects can be expressed by sets of transitions that are either not permitted to fire, or that are required to fire.

A service *Prov* in isolation usually deadlocks (e. g., the shop in Fig. 1 deadlocks in its initial state). Hence, investigating *Prov* in isolation does not make sense in general. Instead, we consider the composition of *Req* and *Prov*, and check whether this composition satisfies the given constraint. This leads to the following definition of exclude and enforce.

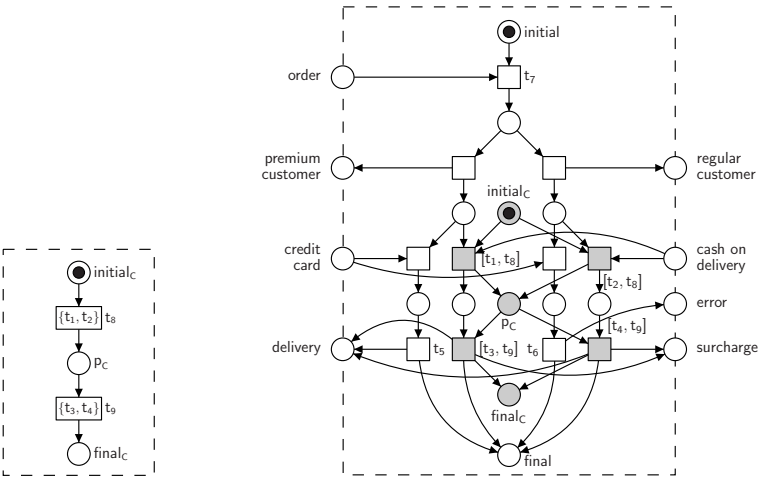
Definition 3 (Exclude, enforce). *Let Req and Prov be two oWFNs and let t be a transition of Prov. $Req \oplus Prov$* excludes *t iff no run of $Req \oplus Prov$ contains t. $Req \oplus Prov$* enforces *t iff every run of $Req \oplus Prov$ contains t.*

Definition 3 can be canonically extended to *sets* of excluded or enforced transitions. As an example, the composition of the online shop and the client described above excludes the transitions t_5 and t_6, because it has no run where the client sends a credit card message. Furthermore, this composition enforces transition t_7, because in every run an order is sent by the client.

When two services *Req* and *Prov* are given, the exclusion or enforcement of transitions can be checked with the help of the runs of $Req \oplus Prov$. Therefore, standard model checking techniques could be used. However — coming back to the scenarios described in the introduction — when a service provider wants to validate his service *Prov*, there is no fixed partner service *Req*. Hence, we follow a different approach: We suggest to change the oWFN *Prov* according to a given constraint in such a way that the set $Strat(Prov')$ of the resulting oWFN $Prov'$ will be exactly the set of requester services *Req* for which the composition of *Req* and the original oWFN *Prov* satisfies the constraint.

To formulate constraints, we propose *constraint oWFNs*. A constraint oWFN is an oWFN with an empty interface whose transitions are labeled with transitions of the oWFN to be constrained.

Definition 4 (Constraint oWFN). *Let N be an oWFN. Let N' be an oWFN with $P_{i_{N'}} = P_{o_{N'}} = \emptyset$ such that $P_N \cap P_{N'} = \emptyset$ and $T_N \cap T_{N'} = \emptyset$. Let L be a labeling function $L : T_{N'} \to 2^{T_N}$. Then $C = [N', L]$ is a* constraint oWFN *for N.*



(a) C_1 (b) $N_{\text{shop}} \otimes C_1$

Fig. 5. (a) The constraint oWFN C_1 for the online shop of Fig. 1. As a convention, the labels of the labeling function L are written inside a transition. (b) The product of the online shop and C_1. Gray nodes highlight the changes of the original oWFN.

Constraint oWFNs are a general means to describe constraints: The exclusion and enforcement of transitions can also be expressed by constraint oWFNs.

Figure 5(a) depicts a constraint oWFN, C_1, for the online shop N_{shop} of Fig. 1. The transitions are labeled with sets of transitions of N_{shop}. Intuitively, C_1 is satisfied if the online shop receives a cash on delivery message (i. e., t_1 or t_2 fires) and then sends a surcharge message (i. e., t_3 or t_4 fires). C_1 is an example for constraining the *order* of transitions and therefore cannot be expressed by exclude/enforce constraints as defined in Def. 3.

To implement a constraint in an oWFN, we construct the product of the respective constraint oWFN and the oWFN to be constrained. Intuitively, labeled transitions of the constraint oWFN are merged (i. e., "synchronized") with the transitions of the considered oWFN. So, the product reaches a final marking if both the constraint and the considered oWFN reach a final marking. Define $L(T) = \bigcup_{t \in T} L(t)$ to be the set of all transitions used as a label.

Definition 5 (Product of oWFN and constraint oWFN). *Let N be an oWFN and let $C = [N', L]$ be a constraint oWFN for N. The* product of N and C *is the oWFN $N \otimes C = [P, P_i, P_o, T, F, m_0, \Omega]$, defined as follows:*

- $P = P_{N \oplus N'}$, $P_i = P_{i_N}$, $P_o = P_{o_N}$,
- $T = (T_N \setminus L(T_{N'})) \cup \{(t, t') \mid t' \in T_{N'}, t \in L(t')\} \cup \{t' \in T_{N'} \mid L(t') = \emptyset\}$,

- $F = F_N \setminus ((P_N \times L(T_{N'})) \cup (L(T_{N'}) \times P_N))$
  $\cup \{[(t,t'),p] \mid t' \in T_{N'}, t \in L(t'), p \in t^\bullet \cup t'^\bullet\}$
  $\cup \{[p,(t,t')] \mid t' \in T_{N'}, t \in L(t'), p \in {}^\bullet t \cup {}^\bullet t'\}$,
- $m_0 = m_{0_N} \oplus m_{0_{N'}}$,
- $\Omega = \Omega_{N \oplus N'}$.

The product of the online shop of Fig. 1 and the constraint oWFN C_1 of Fig. 5(a) is depicted in Fig. 5(b). The marking $[\mathsf{final}, \mathsf{final_c}]$ is the only final marking of the product. Only when composed to a requester who chooses cash on delivery and then receives the surcharge note, the product can reach this marking.

To check whether an oWFN N satisfies a constraint oWFN $C = [N', L]$, the runs of N and N' have to be considered. A run σ' of N' induces an ordered labeled transition sequence. Each label consists of a set of transitions of N. Thus, σ' describes which transitions of N have to be fired in which order. However, σ' might contain unlabeled transitions, and a run σ of N might contain transitions that are not used as a label in N'. Let $\sigma'_{|L}$ be the transition sequence σ' without all transitions with an empty label. Similarly, let $\sigma_{|L}$ be the transition sequence σ without all transitions that are not used as labels.

Definition 6 (Equivalence, satisfaction). *Let N be an oWFN an $C = [N', L]$ a constraint oWFN for N. Let σ and σ' be a run of N and N', respectively. σ and σ' are* equivalent *iff $\sigma_{|L} = t_1 \dots t_n$ and $t_i \in L(t'_i)$ for all $1 \leq i \leq n$. N* satisfies *the constraint oWFN C, denoted $N \models C$, iff for every run of N there exists an equivalent run of N'.*

We now can link the satisfaction of a constraint with the product:

Theorem 1. *Let Prov and Req be two oWFNs and $C = [N, L]$ a constraint oWFN for Prov.*

Then, Req is a strategy for $Prov \otimes C$ iff Req is a strategy for Prov and $Req \oplus Prov \models C$.

Proof (sketch)

($\rightarrow$) Every run of $Req \oplus (Prov \otimes C)$ can be "replayed" by $Req \oplus Prov$. This run satisfies C — if not, C would deadlock in $Req \oplus (Prov \otimes C)$, contradicting the assumption that Req is a strategy for $Prov \otimes C$.

($\leftarrow$) As $Req \oplus Prov \models C$, there exists a sub-run of C for every run of $Req \oplus Prov$. From a run of $Req \oplus Prov$ and its sub-run of C, a run of $Req \oplus (Prov \otimes C)$ can be derived. □

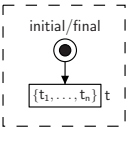
Theorem 1 underlines the connection between the product of an oWFN with a constraint oWFN and the runs satisfying a constraint. This connection justifies more efficient solutions for the first two scenarios described in Sect. 1. In the first scenario, a service provider wants to validate his service *Prov*. In particular, he wants to make sure that for all strategies *Req* for *Prov* the composition $Req \oplus Prov$ satisfies certain constraints, for example that payments will be made, or no errors occur. We suggest to describe the constraint as a constraint oWFN C. Then,

Theorem 1 allows to analyze the product of *Prov* and C, $Prov \otimes C$, instead of *Prov*. The operating guideline of $Prov \otimes C$ characterizes all strategies *Req* for *Prov* such that $Req \oplus Prov$ satisfies C. The benefit of this approach is that instead of calculating all strategies *Req* and checking whether $Req \oplus Prov$ satisfies the constraint C, it is possible to characterize *all* C-satisfying strategies *Req*. In addition, the latter calculation usually has the same complexity as calculating all strategies for *Prov*.

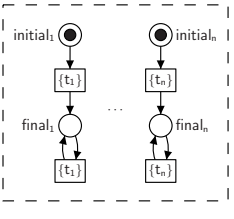
Similarly, the problem of the second scenario, the publication of a provider service in special repositories, can be solved. We assume that the constraints that have to be satisfied by a provider service to be published in a special repository — for example, the required acceptance of credit cards — are published by the service broker. We again suggest to describe these constraints as a constraint oWFN C. The service provider can now calculate the operating guideline $OG_{Prov \otimes C}$ of the product of his service *Prov* and the constraint C. Theorem 1 states that this operating guideline characterizes all strategies *Req* for *Prov* such that $Req \oplus Prov$ satisfies the constraint C. If the set of these strategies is not empty, the service provider can publish $OG_{Prov \otimes C}$ in the service repository.

The service *Prov*, however, can remain unchanged. This is an advantage as — instead of adjusting, re-implementing, and maintaining several "versions" of *Prov* for each repository and constraint — only a single service *Prov* has to be deployed. From this service the customized operating guidelines are constructed and published. If, for example, *Prov* supports credit card payment and cash on delivery, then only the strategies using credit card payments would be published to the repository mentioned above. Though there exist strategies *Req* for *Prov* using cash on delivery, those requesters would not match with the published operating guideline.

To conclude this section, we return to the exclude/enforce constraints defined in Def. 3. As mentioned earlier, they can be expressed as constraint oWFNs.



(a) *exclude*($\{\mathsf{t}_1, \ldots, \mathsf{t}_n\}$)



(b) *enforce*($\{\mathsf{t}_1, \ldots, \mathsf{t}_n\}$)

Fig. 6. In the constraint oWFN to exclude a set of transitions (a), the initial and final marking coincide. Thus, the final marking of the composition becomes unreachable when transition t fires. The constraint oWFN to enforce a set of transitions (b) consists of n similar nets. Each net consists of two labeled transitions to ensure that the enforced transition may fire arbitrary often. To enforce every of the n transitions to fire at least once, the final marking is $[\mathsf{final}_1, \ldots, \mathsf{final}_n]$.

The canonic constraint oWFNs expressing the exclusion and enforcement of transitions are depicted in Fig. 6(a) and Fig. 6(b), respectively.

To enforce a certain communication event (the sending of the delivery notification by the online shop, for example), however, a constraint oWFN like the one in Fig. 6(b) cannot be used: enforcing, for instance, *all* three transitions sending delivery (t_3, t_4, and t_5) would require *each* of the transitions to fire in every run of $Req \oplus N_{\text{shop}}$ for a client *Req*. This is not possible as at most one delivery message is sent. Still, the constraint can be expressed by a constraint oWFN similar to C_1 (cf. Fig. 5(a)) with the set $\{\mathsf{t}_3, \mathsf{t}_4, \mathsf{t}_5\}$ used in a label, meaning one of these transition has to fire.

4 Customized Operating Guidelines

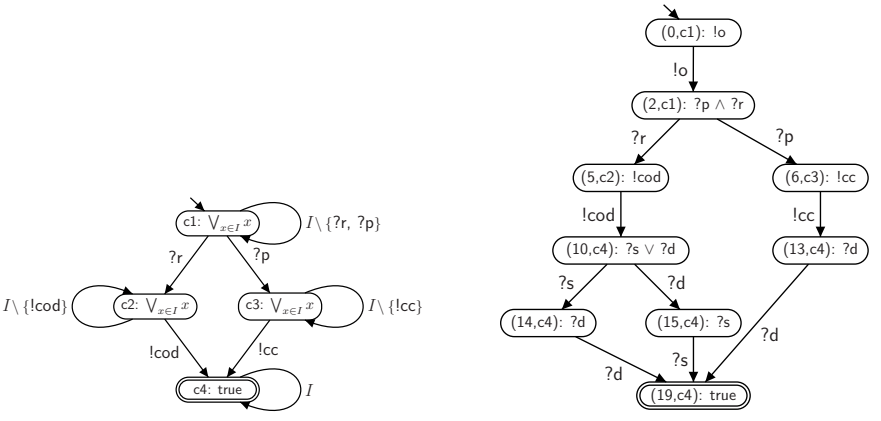
The last section was devoted to implementing constraints at build time. We changed the service by building the product of the oWFN model of the service and the constraint oWFN describing the required behavioral restrictions. The presented solutions can be used to validate a given provider service or to publish the service in special repositories.

In a service-oriented approach, however, we also want to be able to dynamically bind provider and requester services *Prov* and *Req* at runtime without the need to change an already published *Prov*. Thus, the question arises whether it is still possible to satisfy a given constraint after publishing the service *Prov*. In this section, we extend our operating guideline approach to this regard. We show that it is possible to describe a constraint as an annotated automaton C^Ψ, called *constraint automaton*, and apply it by building the product of C^Ψ and the operating guideline OG_{Prov}. The resulting *customized* operating guideline guideline $C^\Psi \otimes OG_{Prov}$ will describe the set of all requester services *Req* such that $Req \oplus Prov$ satisfies the constraint.

An advantage of this setting is that we do not need the original oWFN model of *Prov*. A drawback, however, is that for the same reason we are not able to enforce, exclude, or order concrete transitions of the oWFN any more. C^Ψ may only constrain send or receive actions as such. For example, if two or more transitions send a message a, then a C^Ψ excluding a means that all the original transitions are excluded.

Definition 7 (Constraint automaton). *Let OG_{Prov} be an operating guideline with input channels I_{OG} and output channels O_{OG}. Let $C = [Q, I, O, \delta, q_0, \Omega]$ be a service automaton such that $I \subseteq I_{OG}$ and $O \subseteq O_{OG}$ and let Ψ be an annotation to C. Then, C^Ψ is a* constraint automaton *for OG_{Prov}.*

The product $A^\Phi \otimes B^\Psi$ of two annotated automata A^Φ and B^Ψ can be constructed as follows. The states of $A^\Phi \otimes B^\Psi$ are pairs (q_A, q_B) of states $q_A \in Q_A$ and $q_B \in Q_B$. The initial state is the pair (q_{0_A}, q_{0_B}). A state (q_A, q_B) is a final state of $A^\Phi \otimes B^\Psi$ iff q_A and q_B are final states of A and B, respectively. There is a transition $((q_A, q_B), x, (q'_A, q'_B))$ in $A^\Phi \otimes B^\Psi$ iff there are transitions (q_A, x, q'_A) in A *and* (q_B, x, q'_B) in B. The interface of the product is defined as



(a) A constraint automaton C^{Ψ}. (b) The product $OG_{\text{shop}} \otimes C^{\Psi}$.

Fig. 7. (a) A constraint automaton for OG_{shop} of Fig. 4. A transition labeled with a set means a transition for each element. $I = \{!o, !cc, !cod, ?r, ?p, ?s, ?e, ?d\}$ is the set of all messages that can be sent to or received by the online shop. (b) The product $OG_{\text{shop}} \otimes C^{\Psi}$. The annotations were simplified for reasons of better readability.

$I_{A^{\Phi}\otimes B^{\Psi}} = I_A \cap I_B$ and $O_{A^{\Phi}\otimes B^{\Psi}} = O_A \cap O_B$. The annotation of a state (q_A, q_B) is the conjunction of the annotations of q_A and q_B.

As an example, a constraint automaton C^{Ψ} for OG_{shop} of Fig. 4 is depicted in Fig. 7(a). It assures that premium customers pay with credit card and regular customers pay cash on delivery. Thereby, we exclude the error message and avoid a surcharge where possible. The product of OG_{shop} with C^{Ψ} is depicted in Fig. 7(b). It can easily be seen that every requester service that complies to $OG_{\text{shop}} \otimes C^{\Psi}$ sends !cc after receiving ?p and therefore avoids the surcharge. A regular customer (message ?r) sends !cod, avoiding an error message. Hence, the interaction of the original online shop with a service requester complying to the new OG satisfies the constraint.

The following theorem justifies the construction.

Theorem 2. *Let OG_{Prov} be an operating guideline and let C^{Ψ} be a constraint automaton for OG_{Prov}.*

Then, $Req \in Comply(OG_{Prov} \otimes C^{\Psi})$ iff $Req \in Comply(OG_{Prov})$ and $R \in Comply(C^{\Psi})$.

Proof (sketch). Let OG_{Prov} be equal to A^{Φ}.

($\rightarrow$) $Req \in Comply(A^{\Phi} \otimes C^{\Psi})$ means that Req matches with $A^{\Phi} \otimes C^{\Psi}$ structurally (first item of Def. 1) and each state of Req fulfills the annotation of the matching state(s) of $A^{\Phi} \otimes C^{\Psi}$ (second item of Def. 1). Let q_{Req} be an arbitrary state of Req. Then, if $(q_{Req}, (q_A, q_C)) \in R_{(Req, A^{\Phi}\otimes C^{\Psi})}$, then $(q_{Req}, q_A) \in R_{(Req, A^{\Phi})}$ and $(q_{Req}, q_C) \in R_{(Req, C^{\Psi})}$. Hence, Req matches with A and Req matches with C. By assumption, each assignment $\beta_{q_{Req}}$ fulfills the annotation

$\Phi(q_A) \wedge \Psi(q_C)$ for each $(q_A, q_C) \in R_{(Req, A^\Phi \otimes C^\Psi)}$. Hence, $\beta_{q_{Req}}$ fulfills $\Phi(q_A)$ and $\beta_{q_{Req}}$ fulfills $\Psi(q_C)$.

($\leftarrow$) By assumption, *Req*'s states match with the states of A and of C. Let q_A and q_C be the corresponding states in $R_{(Req,A)}(q_A)$ and $R_{(Req,C)}(q_A)$, respectively. Then, the state (q_A, q_C) is in $A^\Phi \otimes C^\Psi$ and $(q_{Req}, (q_A, q_C)) \in R_{(Req, A^\Phi \otimes C^\Psi)}$. Hence, *Req* matches with $A^\Phi \otimes C^\Psi$. Finally, since the assignment $\beta_{q_{Req}}$ fulfills the annotation $\Phi(q_A)$ and the annotation $\Psi(q_C)$ of matching states in A or C, $\beta_{q_{Req}}$ fulfills their conjunction as well. □

With the result of Theorem 2 we are able to come back to the last two scenarios described in the introduction. As already seen in our example, in these scenarios the constraint is modeled as a constraint automaton C^Ψ. C^Ψ characterizes the set of accepted behaviors and can be formulated without knowing the structure of the *OG* needed later on. Only the interface (i. e., the set of input and output channels of the corresponding service automaton) must be known.

In the third scenario, the (general) operating guidelines of all provider services are already published in the repository and a requester *Req* queries for a matching service *Prov* under a given constraint C^Ψ. Theorem 2 allows that the broker computes the customized operating guideline of a provider first and then matches *Req* with the customized *OG*. That way, the consideration of constraints refines the "find" operation of SOAs: Instead of finding *any* provider service *Prov* such that the composition with a requester service *Req* is deadlock-free, only the subset of providers *Prov* for which $Req \oplus Prov$ satisfies the constraint is returned. To speed up the matching, the two steps of building the product and matching *Req* with the product can easily be interleaved. Additionally, the broker could prepare customized *OG*s for often-used constraints.

In the fourth scenario, the requester service *Req* is yet to be constructed. Therefore, the desired features of *Req* are described as a constraint automaton. For example, consider a requester who wants to book a flight paying with credit card. If these features are expressed as a constraint automaton C^Ψ, it can be sent to the broker who returns all operating guidelines of provider services *Prov* offering these features (i. e., where the product of OG_{Prov} with C^Ψ is not empty). From this operational descriptions, the service *Req* can easily be constructed.

(a) *enforce*(a) (b) *exclude*(a)

Fig. 8. The constraint automata for enforcing or excluding a communication action a

To conclude this section, we depict generic constraint automata for enforcing or excluding a communication action a in Fig. 8. The broker could apply such

constraint automata and store customized *OG*s enforcing or excluding the most common typical communication actions (as payments, errors, etc.) in addition to the general *OG* version of the provider services.

5 Related Work

There is a lot of research being done to enforce constraints in services. The originality of this paper lies in the application of constraints to the communication between a requester and a provider service. Furthermore, the presented model of constraints allows us to refine "find" operation in SOAs.

The idea to constrain the behavior of a system by composing it with an automaton is also used in the area of model checking. When a component of a distributed system is analyzed in isolation, it might reach states that are unreachable in the original (composed) system. To avoid these states, [12] introduce an *interface specification* which is composed to the considered component and mimics the interface behavior of the original system. In [13], *cut states* are added to the interface specification which are not allowed to be reached in the composition. These states are similar states with *false* annotation in Fig. 8(b).

In [14], services are described with a logic, allowing the enforcement of constraints by logical composition of a service specification with a constraint specification. Similarly, several protocol operators, including an intersection operator are introduced in [15]. Though these approaches consider synchronous communication, they are similar to our product definition of Sect. 3.

An approach to describe services and desired (functional or nonfunctional) requirements by *symbolic labeled transition systems* is proposed in [16]. An algorithm then selects services such that their composition fulfils the given requirements. However, the requirements have to be quite specific; that is, the behavior of the desired service have to be specified in detail. In our presented approach, the desired behavior can be described by a constraint instead of a specific workflow. However, the discovery of a composition of services to satisfy the required constraint is subject of future work. Another approach is presented in [17], where a *target service* is specified which is then constructed by composing available services. Again, this approach bases on synchronous communication.

6 Conclusion

We presented algorithms for the calculation of customized operating guidelines from different inputs. Computing a customized *OG* from a general one is useful for scenarios where a requester wants to explore specific features of a service that is published through its (general) operating guideline. Computing a customized *OG* from the service description itself may be useful for registration in specialized repositories as well as for validation purposes.

We implemented all results of this paper in our analysis tool Fiona. Fiona's main functionality is to compute the *OG* of a service from the oWFN description of that service. Suitability of oWFNs for modeling services has been proven

through an implemented translation from the industrial service description language WS-BPEL [5] into oWFNs [6]. Additionally, Fiona can (1) apply a constraint given as a constraint oWFN C to N and calculate the OG of the product $N \otimes C$ and (2) read an OG and apply a constraint automaton C^{Ψ} to the OG to compute the customized OG. First case studies with real-life WS-BPEL processes show that both approaches have the same complexity.

In ongoing work, we are further exploring the validation scenario. In case that a service turns out not to have partners, we are trying to produce convincing diagnosis information to visualize *why* a constraint cannot be satisfied. In addition, we work on an extension of the set of requirements that can be used for customizing an operating guideline. For instance, we explore the possibility of replacing the finite automata used in this paper with Büchi automata, thus being able to handle arbitrary requirements given in linear time temporal logic.

References

1. Gottschalk, K.: Web Services Architecture Overview. IBM whitepaper, IBM developerWorks (2000), `http://ibm.com/developerWorks/web/library/w-ovr`
2. Massuthe, P., Schmidt, K.: Operating Guidelines – An Automata-Theoretic Foundation for the Service-Oriented Architecture. In: Cai, K.Y., Ohnishi, A., Lau, M. (eds.) Proceedings of the Fifth International Conference on Quality Software (QSIC 2005), Melbourne, Australia, pp. 452–457. IEEE Computer Society, Los Alamitos (2005)
3. Lohmann, N., Massuthe, P., Wolf, K.: Operating Guidelines for Finite-State Services. In: Kleijn, J., Yakovlev, A. (eds.) 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland. LNCS, vol. 4546, Springer, Heidelberg (2007)
4. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. Annals of Mathematics, Computing & Teleinformatics 1(3), 35–43 (2005)
5. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0. Committee Specification, Organization for the Advancement of Structured Information Standards (OASIS) (2007)
6. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing Interacting BPEL Processes. In: Dustdar, S., Fiadeiro, J.L., Sheth, A. (eds.) BPM 2006. LNCS, vol. 4102, pp. 17–32. Springer, Heidelberg (2006)
7. Aalst, W.M.P.v.d.: The application of Petri nets to workflow management. Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
8. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing Interacting WS-BPEL Processes Using Flexible Model Generation. In: Dustdar, S., Fiadeiro, J.L., Sheth, A. (eds.) BPM 2006. LNCS, vol. 4102, Springer, Heidelberg (2006)
9. Cassandras, C., Lafortune, S.: Introduction to Discrete Event Systems. Kluwer Academic Publishers, Dordrecht (1999)
10. Ramadge, P., Wonham, W.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. 25(1), 206–230 (1987)
11. Badouel, E., Darondeau, P.: Theory of Regions. In: Reisig, W., Rozenberg, G. (eds.) Lectures on Petri Nets I: Basic Models. LNCS, vol. 1491, pp. 529–586. Springer, Heidelberg (1998)

12. Graf, S., Steffen, B.: Compositional Minimization of Finite State Systems. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 186–196. Springer, Heidelberg (1991)
13. Valmari, A.: Composition and Abstraction. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 58–98. Springer, Heidelberg (2001)
14. Davulcu, H., Kifer, M., Ramakrishnan, I.V.: CTR-S: a logic for specifying contracts in semantic web services. In: Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E. (eds.) Proceedings of the 13th international conference on World Wide Web, WWW 2004, pp. 144–153. ACM, New York (2004)
15. Benatallah, B., Casati, F., Toumani, F.: Representing, analysing and managing web service protocols. Data Knowl. Eng. 58(3), 327–357 (2006)
16. Pathak, J., Basu, S., Honavar, V.: Modeling Web Services by Iterative Reformulation of Functional and Non-functional Requirements. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 314–326. Springer, Heidelberg (2006)
17. Berardi, D., Calvanese, D., Giacomo, G.D., Mecella, M.: Composition of Services with Nondeterministic Observable Behavior. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 520–526. Springer, Heidelberg (2005)