An Algorithm for Matching Nondeterministic Services with Operating Guidelines

Peter Massuthe[1] and Karsten Wolf[2]

[1] Humboldt–Universität zu Berlin
Institut für Informatik
Unter den Linden 6
10099 Berlin, Germany
massuthe@informatik.hu-berlin.de

[2] Universität Rostock
Institut für Informatik
18051 Rostock, Germany
karsten.wolf@uni-rostock.de

Abstract. Interorganizational cooperation is more and more organized by the paradigm of *services*. *Service-oriented architectures* (SOA) provide a general framework for service interaction. SOA describe three roles of services, the *service provider*, the *service requester*, and the *service broker*, together with the three operations *publish*, *find*, and *bind*.
We provide a formal method based on *nondeterministic automata* to model services and their interaction. In this paper, we restrict ourselves to finite and acyclic automata. We suggest *operating guidelines* as a convenient and intuitive artifact to realize the *publish* operation. In our approach, the *find* operation reduces to a matching problem between the requester's service and the published operating guidelines. If matching services are actually bound together, our approach guarantees deadlock-free communication. In this paper, matching of deterministic as well as nondeterministic automata with operating guidelines is presented.

Key words: Services, SOA, Formal Methods, (Nondeterministic) Automata, Operating guidelines, Matching

1 Introduction

Nowadays, cooperation across borders of enterprises is increasingly important. Functionalities are outsourced or so-called virtual enterprises for specific tasks are formed. In this setting, *services* play an important role. A service basically encapsulates self-contained functions that interact through a well-defined interface. Recent publications apply the term service in different contexts with varying denotations (see [1] for a survey). In this paper, we assume the essentials of a service to include its *identifier* (id), its *interface*, and its internal *operational behavior*. We abstract from other aspects of services such as real-time constraints, cost models, underlying middleware, etc.

The well-known class of *web services* is an implementation of services with an interface specified in WSDL (*Web Services Description Language*) [2] and an id given by an URI (*Uniform Resource Identifier*).

In the following, we concentrate on services with operational behavior described as a *workflow*, i.e. an implemented *business process*. Such services have become particularly important since the establishment of BPEL[3] as a widely accepted language to describe web services. BPEL provides control structures that typically occur in workflows.

As a running example we consider the service of a beverage vending machine as outlined in Fig. 1. The service provided by this machine expects a coin (€) to be inserted and one of the buttons T or C being pressed. The service then reacts by delivering a beverage, i.e. a cup of tea (in case T has been pressed) or a cup of coffee (in case C has been pressed).

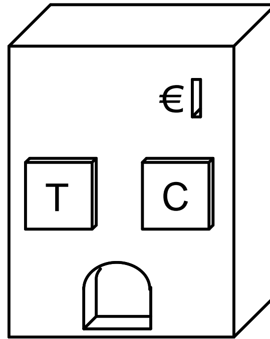


Fig. 1. A vending machine that sells, for 1 Euro, either a cup of tea (button T), or a cup of coffee (button C).

Typically, a service is not executed in isolation, but in cooperation with other services. So, a network of interacting services arises. *Service-oriented architectures* (SOA) [4] provide a general framework for service interaction. SOA distinguish three roles of services: the *service provider*, the *service requester*, and the *service broker*. A service provider *publishes* information about his service to a repository. The service broker manages the repository and allows a service requester to *find* an adequate service provider. Then, the requester may initiate the *bind* operation and both service requester and provider can start interacting.

The three roles of services and the three operations *publish*, *find*, and *bind* are depicted in Fig. 2.

The interaction of services may cause nontrivial communication between a requester and a provider. In our example, a requester of the vending machine service must insert a coin, press a button, and will finally receive his beverage.

[3] *Business Process Execution Language for Web Services* [3], also known as BPEL4WS or WS-BPEL.

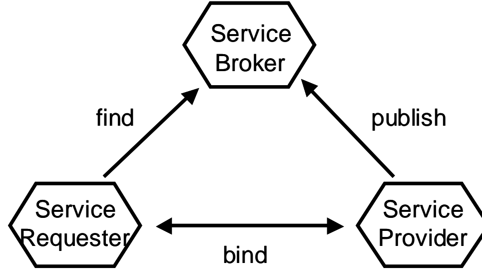


Fig. 2. The service-oriented architecture triangle.

Considering the service broker, it is obviously desirable that a querying service requester gets assigned only such a service provider that their services do not ill-communicate with each other (such as running into a deadlock or sending unanticipated messages).

In our example, a broker must not deliver our vending machine service to a requester who wants to pay in other currencies than € or to a requester who expects the beverage before paying.

For this purpose, the service broker needs information about the internal control structure of the provider's service – the service's interface only (like its WSDL specification, for instance) is not sufficient. Publishing the whole internal control to the service broker would solve the problem. This is, however, not feasible for complexity reasons. Additionally, the service provider may want to keep the internal structure of his service secret.

In a currently quite popular approach, the information published about a provider's service P is a so-called *public view* [5,6] of P, i.e. an abstract version P' with a communication behavior equivalent to P. In this setting, the broker must perform a *compliance check*, i.e. a check whether the system composed of P' and a requesting service R behaves well in the above described manner. The compliance check is mainly a verification task for the (non-)reachability of states in the composed system. It is supposed that compliance of R and P' induces compliance of R and P.

In contrast, our approach is based on a different perspective: A provider does not publish information about internals of *his* service P, but information about all accepted ways of *interacting* with P. This information implicitly describes all well-communicating requester's services R for P and is called *operating guideline* for P (OG_P for short) [7,8].

In our operating guideline setting, the broker must solve a *matching problem*. If a requester's service R matches OG_P then R's interaction with P is accepted. Thus, it is guaranteed that P and R interact well.

As an advantage, matching R with OG_P is less complex than a compliance check of R with the public view P' of P: The complexity of the compliance check is in the order of the *product* of the sizes of R and P'. In contrast, the complexity of matching R with OG_P is basically in the order of the size of R only.

In this paper, we present a formal approach to realize a service-oriented architecture with the help of operating guidelines. In its current shape, it is restricted to acyclic services, i.e. services which cannot return to a state they have been in before. In our approach, a service is modeled as an acyclic, possibly nondeterministic *service automaton*. A possible way of interacting with a service is modeled as an automaton as well.

We establish a theory that enables us to characterize *all* accepted interactions R with a given service P. Our main result in this regard is the separation of the criterion "accepted behavior" into local conditions which, for every state s of R, just talk about presence or absence of transitions leaving s. These conditions can be translated into Boolean annotations to a particular automaton $\mathcal{S}_P$ for characterizing the set of all accepted interactions with P. The *annotated* $\mathcal{S}_P$ then serves as the operating guideline.

The rest of the paper is organized as follows. First we introduce our formal foundation for services, service automata, and their interaction via asynchronous communication in Sect. 2. Then, in Sect. 3 we formally define the operating guideline for a service automaton and develop an algorithm to compute operating guidelines. In Sect. 4 we show how operating guidelines can be used to derive all well-communicating service requesters for a given provider. We present an algorithm to match both deterministic and nondeterministic automata with operating guidelines. Finally, we summarize the results of the paper and sketch our plans for further work.

2 A Formal Model for Services

In this section we introduce *service automata*, a class of communicating, nondeterministic automata, as a formal model for services. A service automaton reflects the internal control of a service as well as its communication behavior via its interface. Service automata provide a model for services of requesters and providers.

Service automata are essentially a simplification of classical I/O automata [9] towards the handling of asynchronous messages. Using I/O automata, the content of mailboxes and message channels would be modeled explicitly as part of the state of an automaton. In our approach, the mailboxes are not considered to be part of the automaton itself. They occur implicitly through the definition of the interaction between service automata. This approach leads to smaller and thus more readable automata. Other versions of automata models for services were proposed by [10] and [1], for instance. In [10] communication is modeled as occurrences of labels with no explicit representation of pending messages, whereas [1] use bounded and unbounded queues to store such messages.

In our service automata model the communication behavior of a service is modeled as labels to transitions of the service automaton. A label $!x$ represents sending a message via channel x (which can represent a message or a real trade item). In contrast, a label $?x$ represents receiving a message from channel x. The label τ represents a silent (i.e. internal) transition. We require that, inside

one and the same automaton, a letter x occurs either everywhere with question mark or everywhere with exclamation mark. For the sake of simplicity, we abstract from data and do not consider the content of a message. For data with finite domain, important message content can, however, be represented in our approach. For instance, a message with Boolean values can be represented through the separation into two channels, one for messages with content *true*, one for messages with content *false*.

The interaction of services is represented by a composite transition system of the corresponding service automata. A state of the transition system consists of a state of each communicating service automaton and the currently pending messages. Well-communication of two services R and P is then expressed as a property of the transition system.

In the following, we assume a finite set MC of message channels.

Definition 1 (Service automaton). *A* service automaton *is a nondeterministic automaton $A = [I, Q, T, q_0, \Omega]$ that consists of*

- *an* interface $I = I_{in} \cup I_{out}$ *such that* $I_{in}, I_{out} \subseteq MC$, $I_{in} \cap I_{out} = \emptyset$*,*
- *a finite set Q of* states*,*
- *a finite set $T \subseteq Q \times L \times Q$ of* transitions *where* $L = \{?x \mid x \in I_{in}\} \cup \{!x \mid x \in I_{out}\} \cup \{\tau\}$*,*
- *an* initial state $q_0 \in Q$*, and*
- *a set $\Omega \subseteq Q$ of* final states*.*

L is called the set of labels *of A.* ⌟

In the sequel, we study *acyclic* service automata only, i.e. automata such that the transitive closure of the transition relation is irreflexive.

We denote service automata by A, B, P, and R. A and B are used for arbitrary automata, P and R are used if we want to emphasize the role as service provider and service requester, respectively. If not clear from the context, we denote the ingredients of a service automaton A by I_A, I_{in_A}, I_{out_A}, Q_A, T_A, q_{0_A}, Ω_A.

As an example, Fig. 3 shows a service automaton P_V modeling the vending machine of Fig. 1. The automaton can receive a coin (label ?€) and accepts one of the two buttons being pressed (labels ?T and ?C). The vending machine then generates a beverage (labels $!\mathsf{B}_\mathsf{C}$ and $!\mathsf{B}_\mathsf{T}$). Let p1 be the initial state of P_V and let p5 and p6 be the final states of P_V.

Consider now two requesters who want to use the provided vending machine service. Their services are modeled as service automata R_C and R_E, depicted in Fig. 4. Let q4 and t3 be the only final states of R_C and R_E, respectively. R_C models a customer who wants coffee, whereas the customer modeled by R_E apparently "forgets" to press one of the buttons. Thus, R_E is an erroneous customer of the vending machine.

Two communicating automata R and P must have interfaces such that one automaton sends only those messages that can be received by the other one, and vice versa. Without loss of generality, we assume $I_{in_R} = I_{out_P}$ and $I_{out_R} = I_{in_P}$ to be given from now on.

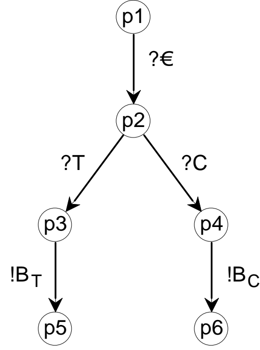
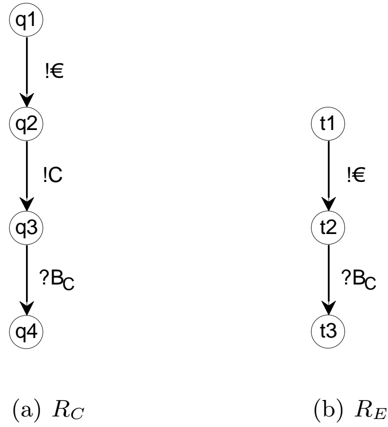


Fig. 3. A service automaton P_V for the provider's vending machine.



(a) R_C (b) R_E

Fig. 4. Two service automata R_C and R_E modeling requesters of the vending machine.

We consider an asynchronous model of message passing between R and P. This model is formalized through the following definition of the transition system $R \oplus P$ representing the behavior of two service automata in interaction. As we assume asynchronous communication, this interaction involves messages pending in channels (namely those which have already been sent but not yet received). Since more than one message of a kind may be pending, we use the concept of *multisets* for modeling pending messages. A multiset is basically an extension of the concept *set* which allows multiple occurrences of one and the same element. Formally, a multiset over a domain A is a mapping $M : A \to \mathbb{N}$. $M(a)$ is called the *multiplicity* of a in M and stands for the number of occurrences of a. In all following definitions, $bags(A)$ denotes the set of all multisets over A. $M + a$ stands for incrementing the multiplicity of a in M by 1 (adding an element to M), and $M - a$ for decrementing the multiplicity of a in M by 1 (removing an

element from M). $a \in M$ is *true* if the multiplicity of a in M is at least 1. We use $\{\}$ to denote the empty multiset where, for all $a \in A$, $\{\}(a) = 0$.

A state of the transition system $R \oplus P$ represents a state of P, a state of R, and a multiset M of pending messages. Every transition of $R \oplus P$ corresponds either to a transition in R or to a transition in P. A transition labeled $!a$ produces a message in channel a, whereas a transition labeled $?a$ consumes a message from channel a which can only occur if a is present in the message bag. τ-transitions do neither create nor consume messages.

Definition 2 (Interaction of service automata). *Let P and R be two service automata. Without loss of generality, let $Q_R \cap Q_P = \emptyset$. The* transition system *$R \oplus P = [Q, T, q_0, \Omega]$ consists of*
- *a set of states $Q \subseteq Q_R \times Q_P \times bags(MC)$,*
- *a set of labeled transitions $T \subseteq Q \times (L_P \cup L_R) \times Q$,*
- *an* initial state *q_0, and*
- *a set $\Omega \subseteq Q$ of* final states.

Q and T are defined inductively as follows:
Basis: *$q_0 = [q_{0_R}, q_{0_P}, \{\}]$ is a state of the transition system.*
Step: *If $q = [q_R, q_P, M]$ is a state and there is a transition*

- *$t = [q_R, !a, q'_R] \in T_R$, then $q' = [q'_R, q_P, M + a] \in Q$ and $[q, !a, q'] \in T$,*
- *$t = [q_P, !a, q'_P] \in T_P$, then $q' = [q_R, q'_P, M + a] \in Q$ and $[q, !a, q'] \in T$,*

- *$t = [q_R, ?a, q'_R] \in T_R$ and $a \in M$, then $q' = [q'_R, q_P, M - a] \in Q$ and $[q, ?a, q'] \in T$,*
- *$t = [q_P, ?a, q'_P] \in T_P$ and $a \in M$, then $q' = [q_R, q'_P, M - a] \in Q$ and $[q, ?a, q'] \in T$,*

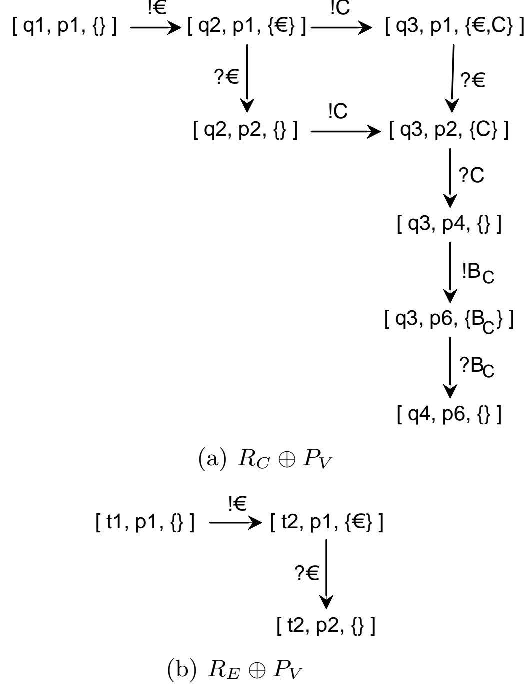
- *$t = [q_R, \tau, q'_R] \in T_R$, then $q' = [q'_R, q_P, M] \in Q$ and $[q, \tau, q'] \in T$,*
- *$t = [q_P, \tau, q'_P] \in T_P$, then $q' = [q_R, q'_P, M] \in Q$ and $[q, \tau, q'] \in T$.*

A state $q = [q_R, q_P, M]$ of $R \oplus P$ is defined to be a final state, i.e. $q \in \Omega$, if and only if $q_P \in \Omega_P$, $q_R \in \Omega_R$, and $M = \{\}$. ⌟

The definition formalizes the intuition that R and P move independently in $R \oplus P$. A receive action is only possible if the message to be received is present in the message bag (it is then removed). A send action adds a message to the bag.

As an example, the transition systems $R_C \oplus P_V$ and $R_E \oplus P_V$ are depicted in Fig. 5. The node [q1, p1, {}] in the transition system in Fig. 5(a) means that the coffee requester R_C of Fig. 4(a) is in its state q1, the vending machine P_V is in its state p1, and there are no messages pending. Since it was possible for the requester to send a coin in state q1, it is possible in [q1, p1, {}], too. Thereby, a new node is reached, where the requester is in state q2 and there is an € pending to be consumed by the vending machine.

In this paper, we concentrate on a correctness criterion called *weak termination.* Intuitively, weak termination means that from every reachable state of the



(a) $R_C \oplus P_V$

(b) $R_E \oplus P_V$

Fig. 5. The two transitions systems $R_C \oplus P_V$ and $R_E \oplus P_V$.

transition system a final state of the transition system is reachable. This criterion is derived from the notion of *weak soundness* of components of distributed workflows [6]. Weak soundness itself is based on the notion of *soundness* of (monolithic) workflows introduced by van der Aalst [11].

Definition 3 (Deadlocks, weak termination). *Let P and Q be two acyclic service automata and let $P \oplus R$ be the corresponding transition system as defined above. A non-final state q, i.e. $q \in Q \setminus \Omega$, without any successor in $R \oplus P$ is a* deadlock. *$P \oplus R$ is* weakly terminating *iff $R \oplus P$ does not have deadlocks.* ⌟

It is easy to see that the only state without successors in $R_C \oplus P_V$ is the state $[\mathsf{q4}, \mathsf{p6}, \{\}]$, which is a final state of $R_C \oplus P_V$: Fig. 5(a) is an example of a transition system which is weakly terminating.

In contrast, the state $[\mathsf{t2}, \mathsf{p2}, \{\}]$ of the transition system $R_E \oplus P_V$ (see Fig. 5(b)) is a deadlock but is no final state: $R_E \oplus P_V$ is *not* weakly terminating.

In this paper, for a given service automaton P, we are interested in the set of *all* service automata R such that the composed system of R and P behaves well. In technical terms, we are looking for all R, s.t. $R \oplus P$ is weakly terminating. Each such R is called a *strategy* for P.

Definition 4 (Strategy). *Let P be a service automaton. A service automaton R is a* strategy *for P iff $R \oplus P$ is weakly terminating.* ⌟

In our example, R_C is a strategy for P_V but R_E is no strategy for P_V.

The term strategy originates from a control-theoretic point of view (see [12,13], for instance): We may see R as a controller for P enforcing the weak termination property.

From now on, a service automaton is denoted by S if we want to emphasize its role as a strategy. The set of all strategies for P is denoted by $Strat(P)$.

Several results in the forthcoming sections are based on a state-by-state characterization of those service automata which are strategies. This characterization uses a mapping that we call *knowledge*.

Definition 5 (Knowledge). *Let R and P be two service automata. The knowledge function $k_{(R,P)}$ is a mapping $k_{(R,P)} : Q_R \to \wp(Q_P \times bags(MC))$, such that $k_{(R,P)}(q_R) = \{[q_P, M] \mid [q_R, q_P, M] \in Q_{R\oplus P}\}$.* ⌟

Informally, $k_{(R,P)}(q_R)$ represents the set of possible states in $R \oplus P$ that P and the message bag can be in, while R is in q_R.

For given services R and P, $k_{(R,P)}$ can be easily computed by constructing $R \oplus P$. This is possible in time proportional to the size of $R \oplus P$ which is at most the size of R times the size of P. For acyclic services R, $k_{(R,P)}$ can as well be computed incrementally since the value $k_{(R,P)}(q)$ just depends on the direct predecessors of q in R.

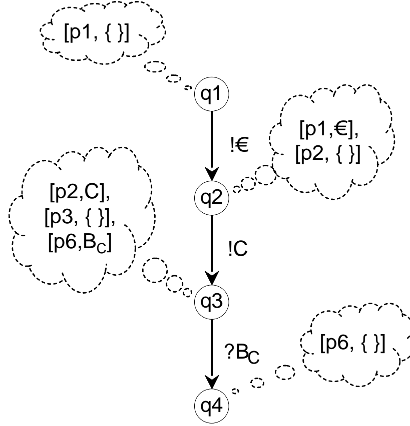


Fig. 6. The service automaton R_C with its knowledge about the vending machine.

Figure 6 shows the coffee requester's service R_C of Fig. 4(a) with its k-values depicted in the bubbles. If R_C is in state $\mathtt{q1}$ then P_V must be in state $\mathtt{p1}$ and the message bag is empty, hence $k_{(R_C,P_V)}(\mathtt{q1}) = \{\,[\mathtt{p1}, \{\}]\,\}$. After sending a coin (i.e. R_C is in state $\mathtt{q2}$), P_V can still be in state $\mathtt{p1}$ and the coin is in the bag, or P_V has already received the payment and is in state $\mathtt{p2}$.

Using the knowledge $k_{(R,P)}$, we can characterize strategies as follows.

Lemma 1. *R is a strategy for P iff, for all $q_R \in Q_R$ and all $[q_P, M] \in k_{(R,P)}(q_R)$, at least one of the following conditions holds:*

- *$q_R \in \Omega_R$, $q_P \in \Omega_P$, and $M = \{\}$, i.e. $[q_R, q_P, M]$ is a final state;*
- *there is a transition $t = [q_P, l, q'_P]$ in T_P, such that there is a transition in $R \oplus P$ that leaves $[q_R, q_P, M]$ and is labeled with l;*
- *there is a transition $t = [q_R, l, q'_R]$ in T_R, such that there is a transition in $R \oplus P$ that leaves $[q_R, q_P, M]$ and is labeled with l.* ⌟

We omit the proof since the result is basically a reformulation of the definition of weak termination. Nevertheless, the lemma shall turn out to be useful since it describes explicitly the obligations for strategies R: Whenever P has no transition leaving a state in $R \oplus P$ (which is determined since we assume P to be given) then R is obliged to have one (i.e. we have to design R such that it is capable of leaving such states).

It is again easy to check the service automata R_C and R_E for these criteria. Whereas R_C fulfills Lemma 1, R_E violates all three criteria in state t2: There is a tuple $[\mathsf{p2}, \{\}]$ in $k_{(R_E,P_V)}(\mathsf{t2})$ such that neither $[\mathsf{t2}, \mathsf{p2}, \{\}]$ is an end state, nor is there a transition possible in $R_E \oplus P_V$: All transitions leaving t2 (p2) are consuming transitions in R_E (P_V), but M is empty.

3 Operating Guidelines

In this section, we develop our notion of operating guidelines. An operating guideline for a service P is an artifact computed from P that is suitable for deciding for arbitrary services R, whether or not R is a strategy for P. In other words, our aim is to characterize the set of all strategies $Strat(P)$. In our approach, the operating guideline OG_P for a service P will turn out to be an annotated automaton. It is built from a specific strategy $\mathcal{S}_P$ and Boolean annotations as in [14]. The strategy $\mathcal{S}_P$ as well as the annotations are selected such that a *deterministic* automaton is a strategy for P if and only if it is a sub-automaton of $\mathcal{S}_P$ that complies to the annotations in a sense yet to be defined. The case of nondeterministic requesters is left to the next section.

A service automaton (and thus a strategy) is *deterministic* if it does not contain τ-transitions and does not have states that are left by multiple transitions with equal labels. The set of all deterministic strategies of P is denoted by $Strat_d(P)$. Obviously, it holds $Strat_d(P) \subseteq Strat(P)$.

We proceed as follows. We formally define the notions sub-automaton, annotation, and compliance. Then we exhibit, based on Lemma 1, the particular strategy $\mathcal{S}_P$ as well as the particular annotations that form the operating guideline OG_P for a given service P. We conclude with an algorithm and remarks on complexity issues.

Definition 6 (Sub-automaton). *An automaton A' is a sub-automaton of an automaton A, $A' \sqsubseteq A$, iff $Q_{A'} \subseteq Q_A$, $T_{A'} \subseteq T_A$, $q_{0_{A'}} = q_{0_A}$, and $\Omega_{A'} \supseteq Q_{A'} \cap \Omega_A$.* ⌟

The set of all sub-automata of an automaton A is denoted by $Sub(A)$.

If P is acyclic, we may unroll every deterministic strategy R of P to an equivalent strategy R' which is a *tree shaped automaton*, i.e. a service automaton where each state has at most one incoming transition. Since P is acyclic, R' has limited depth. Thus, matching non-tree-shaped automata with OG_P can be reduced to matching tree shaped automata with OG_P. It is thus sufficient to characterize the set $Strat_{d,t}(P) \subset Strat_d(P)$ of all (deterministic) tree shaped automata which are strategies for P.

Consider a deterministic tree shaped automaton A and 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 an *annotation* to A. An automaton with an annotation is called *annotated automaton* and is denoted by A^Φ. As an example, Figure 8 shows an annotated automaton.

Definition 7 (Φ-compliance). *Let A^Φ be an annotated automaton, A' a sub-automaton of A, and $q \in Q_{A'}$ (and therefore $q \in Q_A$).*
A state q is compliant *with $\Phi(q)$ iff $\Phi(q)$ is* true *under the assignment assigning* true *to all propositions that are labels of transitions leaving q in A', and* false *to all other propositions.*
A' is compliant *with A^Φ (denoted $A' \models A^\Phi$) iff all states $q \in Q_{A'}$ are compliant with $\Phi(q)$.* ⌟

Having a compliant sub-automaton A' of A, we call every automaton A'' that is isomorphic to A' compliant to A^Φ, too.

Let $Comply(A^\Phi) = \{A' \mid A' \sqsubseteq A, A' \models A^\Phi\}$ denote the set of all automata that are compliant to A^Φ. This way, a single annotated automaton A^Φ represents a set of automata, i.e. the set $Comply(A^\Phi)$.

Our goal for the remaining part of this section is to derive an automaton $\mathcal{S}_P$ and an annotation Σ such that $Comply(\mathcal{S}_P^\Sigma) = Strat_{d,t}(P)$. We will then use $\mathcal{S}_P^\Sigma$ as the operating guideline for P. Both our choice of $\mathcal{S}_P$ and Σ depends on Lemma 1 presented at the end of the previous section.

Let, throughout the remainder of this section, P be a service automaton. First, we derive a suitable automaton $\mathcal{S}_P$. Since only sub-automata can be compliant with $\mathcal{S}_P$, $\mathcal{S}_P$ must be at least as large as the largest deterministic tree shaped strategy of P. As pointed out earlier, there is a depth limit d for deterministic interaction with P. Furthermore, a deterministic tree shaped automaton can have, in every state, at most one transition per message channel. Thus, every deterministic tree shaped strategy must be a sub-automaton of the automaton $\mathcal{F}$ which is a tree of sufficient depth where all states at non-maximum depth have *exactly* one successor for each message channel. Since $\mathcal{F}$ may become as large as c^d where c is the number of message channels of P and d the depth limit, it is desirable to find a smaller automaton that fits our needs. Such an automaton $\mathcal{S}_P$ can be found by applying Lemma 1 to $\mathcal{F}$. The following pseudo-code algorithm illustrates the idea. It first computes $\mathcal{F}$ and then iteratively removes states which violate Lemma 1. The result is the service automaton $\mathcal{S}_P$. In this algorithm, we encode a state of $\mathcal{F}$ as a sequence over message channels. This results in unique

naming of states. The initial state is coded as the empty sequence λ, the successor of state q reached by the transition labeled m is denoted by qm. Let I^* denote the set of finite sequences over I.

```
01 const  d : Integer; // the assumed depth limit
02 var    Q_{S_P} : states;
03        E_{S_P} : edges;
04 begin
05    // start with automaton F described above:
06    Q_{S_P} := {w | w ∈ I_P*, length(w) ≤ d};
07    E_{S_P} := {[q, m, qm] | qm ∈ Q_{S_P}};
08    compute k-values;   // as outlined in the previous section
09    while (exists q ∈ Q_{S_P} such that k_{(S_P,P)}(q) violates Lemma 1)
10       // remove q and its adjacent edges:
11       Q_{S_P} := Q_{S_P} \ {q' | q is prefix of q'};
12       E_{S_P} := E_{S_P} ∩ (Q_{S_P} × L_P × Q_{S_P});
13    // set corresponding final states of S_P
14    Ω_{S_P} := {q ∈ Q_{S_P} | k_{(S_P,P)}(q) ∩ (Ω_P × bags(MC)) ≠ ∅};
15    return S_P := [I_P, Q_{S_P}, E_{S_P}, λ, Ω_{S_P}];
16 end
```

Fig. 7. The algorithm for constructing the strategy $\mathcal{S}_P$ out of $\mathcal{F}$ by removing states violating Lemma 1.

Since, by Lemma 1, states being removed by this procedure cannot be a member of any strategy of P, we immediately have, that every deterministic tree shaped strategy of P is a sub-automaton of the resulting automaton $\mathcal{S}_P$. Formally:

Lemma 2. $\mathit{Strat}_{d,t}(P) \subseteq \mathit{Sub}(\mathcal{S}_P)$. ⌟

Justified by this lemma, we shall refer to the computed $\mathcal{S}_P$ as the *most permissive (tree shaped) strategy* for P.

The outlined algorithm involves starting out with the automaton $\mathcal{F}$. This can be substantially improved by constructing and k-annotating $\mathcal{F}$ incrementally. This way, it is not necessary to descend beyond states that violate Lemma 1 anyway. This idea has been detailed out in [15].

In the next step, we aim at constructing a particular annotation Σ to an arbitrary tree shaped automaton R such that $\mathit{Comply}(R^{\Sigma})$ is exactly the set of those sub-automata of R which are in $\mathit{Strat}_{d,t}(P)$. For this purpose, let $q_R \in Q_R$. Then the formula $\Sigma(q_R)$ is built as a straight coding of the criteria in Lemma 1: $\Sigma(q_R)$ is the *conjunction* of sub-formulae $\sigma_{(q_R,q_P,M)}$, for all $[q_P, M] \in k_{(R,P)}(q_R)$. If $k_{(R,P)}(q_R) = \emptyset$ then let $\Sigma(q_R) = \mathit{true}$.

The sub-formula $\sigma_{(q_R,q_P,M)}$ is

- *true* if $q_R \in \Omega_R$, $q_P \in \Omega_P$, and $M = \{\}$;

- *true* if there is a transition of P leaving $[q_R, q_P, M]$ in $R \oplus P$;
- the disjunction of all l occurring as labels of transitions in R that leave $[q_R, q_P, M]$ in $R \oplus P$, otherwise.

Please note that if there is no l in the third item, then $\sigma_{(q_R,q_P,M)}$ is the empty disjunction which is equivalent to $false$.

Lemma 3. *Let R, P, Σ be as described above. Then $Comply(R^{\Sigma})$ is the set of those sub-automata of R which are in $Strat_{d,t}(P)$.* ⌟

Proof (Sketch). For a tree shaped automaton R and a sub-automaton R' it is easy to verify that, for all $q \in Q_{R'}$, $k_{(R',P)}(q) = k_{(R,P)}(q)$. Then, by the construction of Σ, $R' \in Comply(R^{\Sigma})$ iff every state of R' satisfies the criteria stated in Lemma 1. □

Combining the previous two lemmas, we obtain:

Theorem 1 (Characterization of strategies). *Let P be an arbitrary service automaton. Let $\mathcal{S}_P$ be its most permissive strategy. Then, $Comply(\mathcal{S}_P^{\Sigma}) = Strat_{d,t}(P)$.* ⌟

Thus, the following definition of operating guidelines is justified:

Definition 8 (Operating guideline). *Let P be an arbitrary service automaton. Let $\mathcal{S}_P$ be its most permissive strategy. Let Σ be annotations to $\mathcal{S}_P$ as described above. Then $\mathcal{S}_P^{\Sigma}$ is called the* operating guideline OG_P *for P.* ⌟

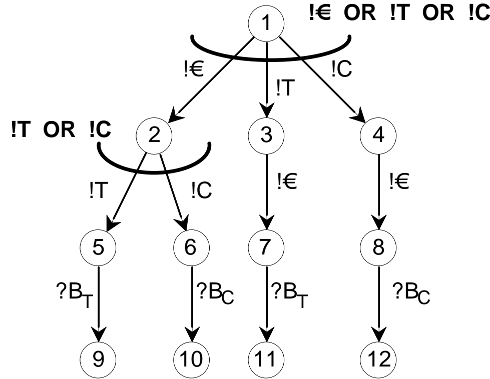


Fig. 8. The operating guideline OG_{P_V} for the vending machine service automaton P_V.

As an example, we recall our vending machine service automaton P_V. The operating guideline OG_{P_V} for P_V (Fig. 3) is depicted in Fig. 8. It is constructed

by removing states from the complete automaton of depth 3 with labels !€, !T, !C, ?$\mathsf{B_C}$, and ?$\mathsf{B_T}$ leaving each state. The annotation of a state with less than two successors is skipped since it coincides with the label of the outgoing transition (in case of one successor) or is *true* if there are no successors.

It is easy to see that the service automaton R_C of Fig. 4(a) is compliant with the annotations. The automaton R_E of Fig. 4(b), instead, is not compliant since its state t2 (which is corresponding to state 2 of OG_{P_V}) violates the formula attached to that state: There is no transition in R_E leaving t2 that is labeled with !T or !C.

Computing OG_P consists of first computing the most permissive strategy $\mathcal{S}_P$ for P, and second computing the annotations for all states of $\mathcal{S}_P$. Computing $\mathcal{S}_P$ can be done in time $c^d \cdot |P|$ where c is the number of message channels of P, d is the maximum number of interactions in any run of P, and $|P|$ is the number of states of P. In one pass, the full automaton $\mathcal{F}$ of size c^d is generated. In a second pass, $\mathcal{F}$ is annotated with k-values which takes $|\mathcal{F}| \cdot |P|$. In a third pass, states of $\mathcal{F}$ are removed if they violate Lemma 1. This can be done in a single depth-first search through $\mathcal{F}$ which is linear in $|\mathcal{F}|$. The accumulated costs of investigating the k-values of $\mathcal{F}$ is proportional to $|\mathcal{F}| \cdot |P|$. Thus, all passes need at most $c^d \cdot |P|$. Computing the annotations takes $c \cdot |\mathcal{S}_P| \cdot |P|$. For every state q of $\mathcal{S}_P$, we need to traverse through $k_{(\mathcal{S}_P,P)}(q)$ and build a disjunction of at most c elements. Thus, the accumulated costs are $O(c \cdot |\mathcal{S}_P| \cdot |P|)$. The computational efforts can be significantly reduced by the use of technologies known from the area of *model checking*. There, several powerful techniques for alleviating state explosion have been developed.

We propose to use OG_P as an artifact generated by the owner of a provided service P which can be published to the service broker.

4 Matching Service Automata with OG_P

In the previous section, we proposed the operating guideline $OG_P = \mathcal{S}_P^{\Sigma}$ as a characterization of all deterministic strategies for a given service automaton P, i.e. $Strat_{d,t}(P) = Comply(OG_P)$. The result is a very easy check to find out whether a querying requester's service R will weakly terminate with the provided service P. For a *deterministic* automaton R, the broker must match R with OG_P and decide whether R is an element of the set $Comply(OG_P)$. This can easily be done in two steps: First, the broker needs to check if $R \sqsubseteq \mathcal{S}_P$. If the test fails, then R can be no strategy. Otherwise the broker must, in a second step, check the compliance of R with the annotations, i.e. decide if $R \models \mathcal{S}_P^{\Sigma}$. Only if both tests succeed, R is a strategy. If at least one test fails, R and P are not weakly terminating and hence R is no strategy.

Thus, matching a deterministic service automaton R with OG_P amounts to unrolling R to a tree, to map the nodes of R to nodes of OG_P, and to evaluate the annotations in OG_P. All steps can be performed during a single depth-first search through the unrolled version of R and is thus linear in the size of R.

A typical example of a requester that already fails in the first step is our vending machine requester R_E (Fig. 4(b)) who forgets to press a button. It is easy to see that R_E is no sub-automaton of the most permissive strategy $\mathcal{S}_{P_V}$ underlying the operating guideline OG_{P_V} of our vending machine.

A requester who first inserts a coin and then presses the button for tea, but who then leaves his beverage in the machine is a typical example for failing in the second step of the matching process. The corresponding service automaton would be a sub-automaton of $\mathcal{S}_{P_V}$ in the first step. In the second step, however, the automaton must have a transition labeled $?\mathsf{B_T}$ to comply with the annotation of state $\mathsf{5}$ of OG_P – which is not present in the automaton.

In the rest of this section, we are interested in the matching of *all* strategies, including the nondeterministic ones. Fortunately, OG_P is, without any change, capable of characterizing nondeterministic strategies, too. In the sequel, we present an algorithm that receives an arbitrary (deterministic or nondeterministic) acyclic service automaton R and the operating guideline OG_P of P as input, and is capable of deciding whether R is a strategy for P or not. Without loss of generality, we assume R to be given in its unrolled shape, i.e. as a *tree shaped* automaton. We proceed with presenting the algorithm, followed by a justification of its correctness.

Our algorithm is based on a coordinated depth-first traversal of R and OG_P. Let again $\mathcal{S}_P$ be the most permissive strategy underlying the operating guideline OG_P, i.e. $OG_P = \mathcal{S}_P^{\Sigma}$. Hence, a state $q_{\mathcal{S}_P}$ is a state in OG_P and a state q_R is a state in R. The algorithm assigns, to each state q_R in R, a "fitting" state $q_{\mathcal{S}_P}$ in OG_P. Then, we evaluate the annotation of the assigned state $q_{\mathcal{S}_P}$ under the assignment given by q_R. Only those states of R that do not have leaving τ-transitions must be evaluated. We claim that R is a strategy if and only if all executed evaluations yield *true*. This amounts to the following pseudo-code for our matching algorithm, depicted in Fig. 9.

The number of calls to dfs is at most the number of states of R. Our algorithm is thus an efficient instrument for matching an acyclic service automaton R with an operating guideline OG_P.

As an example, consider again our vending machine. The relevant part of its operating guideline is depicted on the right hand side of Fig. 10(a) and Fig. 10(b). Please note the state $\mathsf{5}$ of the operating guideline. This node will never be reached by any requester since the vending machine P_V will not return a coffee after button T was pressed. Thus, the knowledge value $k(\mathsf{5})$ is empty and the annotation of state $\mathsf{5}$ is equal to *true*. This node is not removed while constructing the operating guidelines (but was not shown in the figures before for reasons of better readability).

Consider now two new nondeterministic requesters of the vending machine. The requester on the left hand side of Fig. 10(a) is capable of receiving a coffee or to internally decide for a tea in state $\mathsf{p3}$. The call $\text{dfs}(\mathsf{p3}, \mathsf{3})$ results in hasTau being *true* and hence the formula $\Sigma(\mathsf{3}) = ?\mathsf{B_T}$ is not evaluated for $\mathsf{p3}$. All executed evaluations in Fig. 10(a) yield *true*. Thus, Fig. 10(a) is an example for positive matching.

```
01 proc main(R: strategy, OG_P: operating guideline)
02     dfs(q0_R, q0_{S_P});
03     exit("yes")
04
05 proc dfs(q_R, q_{S_P})
06     hasTau := false;
07     for all [q_R, l, q'_R] ∈ T_R do
08         if l = τ then dfs(q'_R, q_{S_P}); hasTau := true;
09         else if ¬∃q'_{S_P} : [q_{S_P}, l, q'_{S_P}] ∈ T_{S_P} then exit("no");
10         else dfs(q'_R, q'_{S_P}); /* note that q'_{S_P} is unique if it exists */
11     if ¬ hasTau then
12         match := evaluate Σ(q_{S_P}) with assignment defined by q_R;
13         if ¬ match then exit("no");
14     return.
```

Fig. 9. The matching algorithm for matching a requester R with an operating guideline OG_P.

In contrast, in Fig. 10(b) the algorithm returns "no". The depicted requester is very similar to the left one with one difference: In state q3, he is capable of receiving a tea or, e.g. after a timeout, to decide for a coffee. In call dfs(q5, 3), there is no τ-transition leaving q5 and thus the formula $\Sigma(3) = ?\mathsf{B_T}$ is evaluated and the evaluation returns *false*.

The first observation involved in justifying this algorithm is:

Lemma 4. *For each called instance* $\mathrm{dfs}(q_R, q_{\mathcal{S}_P})$*:* $k_{(R,P)}(q_R) = k_{(\mathcal{S}_P,P)}(q_{\mathcal{S}_P})$. ⌟

Proof. (Sketch, induction over the transition relation of R)

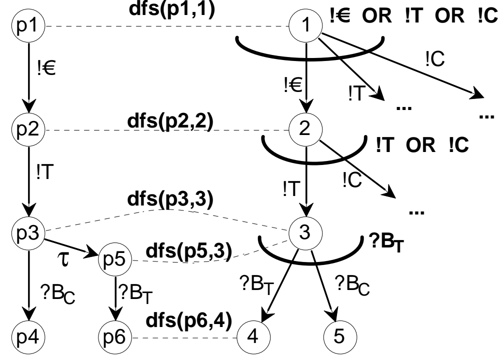
Basis: The k-values of the initial states coincide since they correspond to the states reachable in P without any transition of R or $\mathcal{S}_P$.

Step: Let $[q_R, l, q'_R] \in T_R$ and $k_{(R,P)}(q_R) = k_{(\mathcal{S}_P,P)}(q_{\mathcal{S}_P})$. If $l = \tau$ then we call $\mathrm{dfs}(q'_R, q_{\mathcal{S}_P})$ and it holds $k_{(R,P)}(q_{R'}) = k_{(R,P)}(q_R) = k_{(\mathcal{S}_P,P)}(q_{\mathcal{S}_P})$ since the τ-transition does not change the status of message channels and it does neither enable nor disable any transition in $\mathcal{S}_P$. If $l \neq \tau$, it can be shown that, for the unique $q'_{\mathcal{S}_P}$ holding $[q_{\mathcal{S}_P}, l, q'_{\mathcal{S}_P}] \in T_{\mathcal{S}_P}$, $k_{(R,P)}(q_{R'}) = k_{(\mathcal{S}_P,P)}(q_{\mathcal{S}'_P})$ since, in a tree automaton, the k-value of the target of a transition is uniquely determined by the k-value of the source state of the transition and its label. □

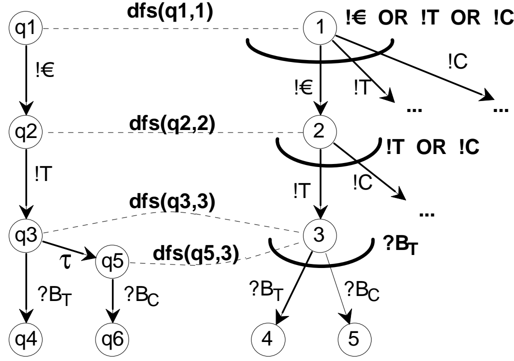
With this observation, it is easy to show

Theorem 2 (Justification I). *If the above algorithm exits with "yes", then R is a strategy for P.* ⌟

Proof. For every state $q_R \in Q_R$, the conditions of Lemma 1 are satisfied: Either, there is a τ-transition leaving q_R. Then the third condition is *true* since the τ-transition is executable in q_R independently from q_P and M. Or, there is no τ-transition leaving q_R. Then, the conditions hold since the state $q_{\mathcal{S}_P}$ assigned to



(a) Positive matching.



(b) Negative matching.

Fig. 10. Two examples for the matching algorithm calls of dfs$(q_R, q_{\mathcal{S}_P})$.

q_R by the coordinated depth-first search has the same k-value as q_R (Lemma 4), and so the annotation of $q_{\mathcal{S}_P}$ correctly encodes the conditions of Lemma 1 for q_R. □

For the justification, it remains to show that, whenever the algorithm exits with "no", then R is not a strategy. We use the following lemma.

Lemma 5. *Let R be a tree shaped strategy that has a transition $[q, \tau, q']$. Let $Q'_R = Q_R \setminus \{q\}$. If $q = q_{0_R}$ then let $q_{0_{R'}} = q'$, otherwise let q'' be the unique state such that $[q'', l, q] \in T_R$ and let $T_{R'} = (T_R \cup \{[q'', l, q']\}) \setminus \{[q'', l, q], [q, \tau, q']\}$. Then it holds: R' is a tree shaped strategy, too.* ⌟

Proof (Sketch). Since the occurrence of a τ-transition in R is not constrained by P, R may decide to execute it whenever it is possible. If R is a strategy then

it is still one if it obeys this "τ first" rule. R' describes basically R under this rule for a particular τ-transition. □

Assume, our algorithm exits with "no" in line 9 during a call dfs$(q_R, q_{\mathcal{S}_P})$ with R being a strategy. Then, using Lemma 5, there would exist a deterministic strategy that has a state q_R corresponding to $q_{\mathcal{S}_P}$ with a transition leaving q_R labeled l. Since $q_{\mathcal{S}_P}$ does not have this transition but is known to contain all deterministic strategies as sub-automaton, we obtain a contradiction.

Assume that our algorithm answers with "no" in line 13 during a call dfs$(q_R, q_{\mathcal{S}_P})$, even though R is a strategy. Then, with the same argument as above, we can derive a deterministic strategy that runs into the same situation and contradicts the relation between OG_P and deterministic strategies established in the previous section. Consequently:

Theorem 3 (Justification II). *If the above algorithm exits with "no", then R is not a strategy of P.* ⌟

5 Conclusion

We introduced a formal approach to service-oriented architectures that is based on automata. The model is called service automata, a class of communicating, nondeterministic automata. Both service provider *and* service requester are modeled as service automata. The composition of two service automata results in a transition system, modeling the asynchronous communication between services.

Then, we introduced the concept of annotated automata as a condensed form to represent sets of automata. We argued that the operating guideline for a service provider is a suitable and elegant artifact to be published to the service broker. With the help of operating guidelines, the broker can easily match a provider with a querying requester.

Even though the guidelines originally cover only *deterministic* strategies, they can, without change, be used for a characterization of *nondeterministic* strategies, too. The matching algorithm is linear in the size of the requester's service.

We provided an approach on a solid theoretic basis. All constructions and matchings can be performed fully automatically and without giving away the internal structure of the provider.

In this paper, we only studied acyclic systems and restricted ourselves to the interaction between one provider and one requester. Current research activities are concerned with operating guidelines for multiple partners and systems with cycles and the extension to service composition. Furthermore, we want to apply the concept of operating guidelines to other problems like the exchangeability of services.

References

1. Hull, R., Benedikt, M., Christophides, V., Su, J.: E-services: A Look Behind the Curtain. In: PODS '03: Proceedings of the twenty-second ACM SIGMOD-

SIGACT-SIGART symposium on Principles of database systems, New York, NY, USA, ACM Press (2003) 1–14
2. Christensen, E., Curbera, F., Meredith, G., Weeravarana, S.: Web Service Discription Language (WSDL) 1.1. Technical report, Ariba, International Business Machines Corporation, Microsoft (2001)
3. Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Weerawarana, S.: Business Process Execution Language for Web Services, Version 1.1. Specification, BEA Systems, IBM, Microsoft, SAP, Siebel (2003) http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbiz2k2/html/bpel1-1.asp.
4. Gottschalk, K.: Web Services Architecture Overview. IBM whitepaper, IBM developerWorks (2000) http://ibm.com/developerWorks/web/library/w-ovr/.
5. Leymann, F., Roller, D., Schmidt, M.: Web Services and Business Process Management. IBM Systems Journal **41**(2) (2002)
6. Martens, A.: Verteilte Geschäftsprozesse - Modellierung und Verifikation mit Hilfe von Web Services. PhD thesis, Institut für Informatik, Humboldt-Universität zu Berlin (2004)
7. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. In: 2nd South-East European Workshop on Formal Methods 2005 (SEEFM05), Ohrid, Republic of Macedonia (2005)
8. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. Annals of Mathematics, Computing & Teleinformatics **1**(3) (2005) 35–43 To appear.
9. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
10. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic Composition of e-Services that Export their Behavior. In: Proc. of the 1st Int. Conf. on Service Oriented Computing (ICSOC 2003). Volume 2910 of Lecture Notes in Computer Science., Springer (2003) 43–58
11. Aalst, W.: The Application of Petri Nets to Workflow Management. Journal of Circuits, Systems and Computers **8**(1) (1998) 21–66
12. Cassandras, C., Lafortune, S.: Introduction to Discrete Event Systems. Kluwer Academic Publishers (1999)
13. Ramadge, P., Wonham, W.: Supervisory Control of a Class of Discrete Event Processes. SIAM J. Control and Optimization **25**(1) (1987) 206–230
14. Wombacher, A., Fankhauser, P., Mahleko, B., Neuhold, E.: Matchmaking for Business Processes Based on Choreographies. International Journal of Web Services **1**(4) (2004) 14–32
15. Weinberg, D.: Analyse der Bedienbarkeit. Diplomarbeit, Humboldt-Universität zu Berlin (2004)