Towards deciding policy violation during service discovery

Jan Sürmeli

Institut für Informatik
Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
suermeli@informatik.hu-berlin.de

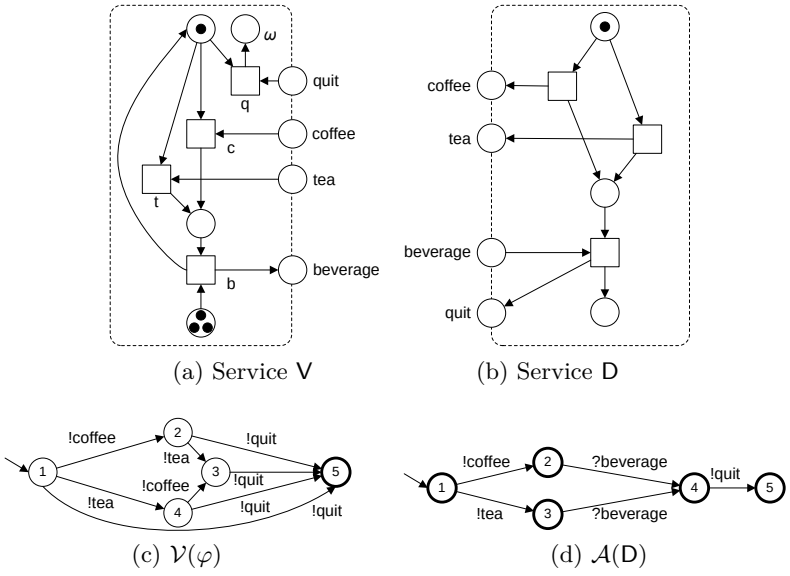
Abstract. In a service-oriented architecture, a *provider* publishes its service in a service repository. A *requester* approaches a *broker* which returns a service S matching the requester's service R. Then, S and R are coupled. The provider of S may require a specific relation between the expenses and rewards for an execution of S, summarized in a *policy* φ. The control flow of S may contain both internal and external decisions: By sending messages, R may trigger a certain execution path. Based on models of S and R, the broker may decide if R *violates* φ before coupling. If so, the broker may not couple S and R. In this paper, we provide a formal framework to model policies, and introduce a decision procedure for policy violation based on open net models of the services.

1 Setting and problem

We understand a *service* as a component with an inner control flow and an interface to exchange messages asynchronously with other services. Thereby, it provides a certain functionality which may be used by other services. A *provider* publishes its service in a repository. A *requester* approaches a *broker* for accessing a previously published service. The provider earns a reward for providing its service. This reward may manifest as a usage fee, or a provision from the repository owner, or from any third party. Usually, a provider desires some beneficial relation between this reward and the expenses for providing its service. As an example, a provider might want the expenses to be covered by the reward. We specify such requirements as *policies*. A partner either *violates* a policy or not. The provider aims at its service being coupled only with non-violating partners. Both reward and expenses may have fixed and variable components. This is a quite usual problem in economics and solutions for this problem are known for a long time. However, in our case, we encounter another difficulty: We consider *stateful* services. A stateful service has its own control flow which is influenced by internal and external decisions. External decisions are made through asynchronous message exchange. Therefore, reward and expenses for providing a service vary from requester to requester.

As a running example, consider a vending machine which sells coffee and tea, modeled as an *open net* [1] in Fig. 1(a). In its initial state, it waits for one of

three messages: Either an order for coffee, an order for tea, or a quit message. To receive an order it executes the respective transition c or t. Subsequently, it serves the beverage by executing b. A quit message may be consumed by executing q, resulting in a final state ω. The machine may serve up to three beverages, as indicated by the three tokens in the place in the bottom. The provider of the vending machine may have fixed expenses of 10 units for providing its service and variable expenses for each served beverage depending on the type: 20 units for coffee and 10 units for tea. As a reward, the provider collects a fixed amount of 5 units and additionally 25 units per served beverage. Assume the provider desires the expenses to be fully covered by the reward, specified in a policy φ_{V}. We find that a customer ordering at least one beverage is a good customer, whereas a customer ordering nothing and simply quitting is not. However, asynchronous message exchange induces a subtle problem: A customer ordering a beverage and then sending the quit message before receiving the beverage is a bad customer: The vending machine might receive the quit message first. A simple partner for V is shown in Fig. 1(b): D orders either a tea or a coffee, receives the beverage, and sends a quit message. Obviously, D does not violate φ_{V}.



(a) Service V (b) Service D

(c) $\mathcal{V}(\varphi)$ (d) $\mathcal{A}(\mathsf{D})$

Fig. 1. Services V and D, modeled as open nets, a finite representation $\mathcal{V}(\varphi)$ of all φ-violating partners of V, and a finite automaton $\mathcal{A}(\mathsf{D})$ characterizing D.

There exists work on *quality of service (QoS)*, e.g. [2] and pricing of services, e.g. [3], focusing on non functional properties of *stateless* services. Such services do not have their own control flow. Therefore, for each dimension, for example *costs*, a value is given. Research is centered on finding composites of many services

which reach a specific goal in the least expensive way, e.g. [4]. For stateful services, there exists work on over-approximating the costs incurring while running the service with a given partner service or a set of partner services described as constraints [5]. This analysis gives the service provider an idea on the QoS of its service. However, it does not provide a sufficient basis to decide if a partner violates the provider's policy or not. Likewise, [6,7] compute the costs for running a business process. In our setting, the closed system is not known beforehand. Instead, we consider open systems.

We sketch our solution. Upon publishing, the provider states its requirements in form of a *policy* φ. Upon requesting, the broker decides whether the requester *violates* the policy or not. This procedure is similar to that of Operating Guidelines [8] which allows the broker to decide policy violation: We use an automaton characterizing partner behavior which may lead to policy violation.

We illustrate our approach on the running example: Figure 1(c) shows automaton $\mathcal{V}(\varphi_{\mathsf{V}})$ which finitely represents violating partner behavior. For instance, the sequence !tea!quit (read: *send* tea, *send* quit) may result in expenses that are not covered by the reward and therefore *violates* φ_{V}. By comparing $\mathcal{V}(\varphi_{\mathsf{V}})$ with an abstract model of a requester, the broker may decide policy violation. For example, automaton $\mathcal{A}(\mathsf{D})$ in Fig. 1(d) is an abstract model of open net D in Fig. 1(b). Comparing $\mathcal{V}(\varphi)$ and $\mathcal{A}(\mathsf{D})$, we find no common path to a final state. Therefore, D does not violate φ_{V}. As a consequence, the broker may return V.

The rest of the paper is structured as follows: Section 2 shortly recalls basic formal concepts, especially open nets. We introduce a framework for policies in Sect. 3. We sketch our approach to compute a finite representation of violating behavior in Sect. 4. Finally, we conclude our paper and present ideas for future work in Sect. 5.

2 Basic notions

As usual, $\mathbb{Z}$ denotes the set of all *integers*. We write Σ^* for the set of all *finite sequences* over an alphabet Σ. For $\sigma \in \Sigma^*$, we write $\sigma(i)$ for the *i-th character* in σ. We denote the *restriction* of σ to $\Sigma' \subseteq \Sigma$ with $\sigma_{|\Sigma'}$. We recall the basic notions of *Petri nets*: A *Petri net* is a tuple $N = \langle P, T, F, m_0 \rangle$ of *places* P, *transitions* T, *arcs* F and *initial marking* m_0. We denote the set of all markings of N with $\mathcal{M}_N$. We denote the *preset* and *postset* of $x \in P \cup T$ with ${}^\bullet x$ and $x^\bullet$, respectively. We canonically extend these notions to sets of net elements by union. We call a sequence $\sigma \in T^*$ firing sequence if the transitions in σ may be fired subsequently starting in m_0. We write $beh(N, m)$ for the set of all firing sequences resulting in a marking m.

Open nets are Petri nets with an interface declaration and a set of final markings: We define an *open net* as a tuple $N = \langle P, T, F, m_0, I, O, \Omega \rangle$ where $\langle P, T, F, m_0 \rangle$ forms a Petri net, I, O are disjoint subsets of P with ${}^\bullet I \cup O^\bullet = \emptyset$, called *input* and *output places*, respectively, and $\Omega \subseteq \mathcal{M}_N$ is a set of *final markings*. We call the Petri net $ip(N) = \langle P', T, F', m_0' \rangle$ the *inner process* of N where $P' = P \setminus (I \cup O)$, $F' = F \cap ((P' \times T) \cup (T \times P'))$, and $m_0'(p) = m_0(p)$

for all $p \in P'$. We call two open nets N_1, N_2 *partners* if their inner processes are component-wise disjoint, $I_1 = O_2$, and $O_1 = I_2$. We compose two partners N_1, N_2 by accordingly merging the interface places, yielding $N_1 \oplus N_2$.

Example 1. Figure 1(a) shows an open net V with input places quit, coffee and tea, and output place beverage. The set Ω_V of final markings cannot be seen from the figure. We define $\Omega_\mathsf{V} = \{m \mid m \in \mathcal{M}_\mathsf{V} \wedge m(\omega) > 0\}$. The open net D in Fig. 1(b) and V are partners.

3 A formal framework for policies

A *policy* specifies the allowed behavior of a provided service N in composition with an arbitrary partner Q. The main building blocks of a policy are *cost functions* and *constraints*. For the following definitions, we assume a given open net N with transitions T_N.

Cost functions. There are different approaches to define cost functions based on behavior. The most general is to define a cost function as a mapping from transition sequences to some value domain. Throughout this paper, we use the set of integers $\mathbb{Z}$ for this purpose. In our approach, we specify the costs for executing a *single* transition after having executed a (finite) *history*. This covers varying costs for executing a single transition based on the knowledge which transitions have been fired. A cyclic service usually has infinitely many and arbitary long (finite) runs. To ease up analysis, we encode histories into *hash values*.

A hashing from a set A into a set B is a function $f : A \to B$. Usually, the idea is that the elements of B, called hash values, are more lightweight than those of A. Thus, a hash function may be used for efficient table lookups and the like. Typically, a hash function is required to fulfill a number of properties ensuring its usability. In this paper, we use *finite histories* as input: a finite sequence of transitions. We define a *history hashing* h as a hashing from T_N^* into some set H having two properties: *Continuity* and *Finiteness*.

Definition 1 (Continuity, finiteness, history hashing). *We define two properties for functions $h : T_N^* \to H$:*

1. Continuity. *Let $\sigma, \sigma', \sigma'' \in T_N^*$. If $h(\sigma) = h(\sigma')$, then $h(\sigma\sigma'') = h(\sigma'\sigma'')$.*
2. Finiteness. *H is finite.*

We call h a history hashing *if h has both properties* continuity *and* finiteness.

Intuitively, *continuity* demands that, given two histories with the same hash value, each equal continuation of the two results in the same hash value again. *Finiteness* restricts history hashings to finite sets of hash values. We elaborate on the value of these properties for analysis in Sect. 4. For the definitions of this section, we assume a given history hashing h into a set of hash values H.

As mentioned above, we define *cost functions* for executing single transitions based on the hash values. Thus, the domain of such a function is the cross product of the set of transitions and the set of hash values. Induction yields the semantics of the cost function: The costs for a complete transition sequence.

Definition 2 (Cost functions). *We call a function* $f : T_N \times H \to \mathbb{Z}$ cost function. *We define the* semantics *of* f *as the function* $\|f\| : T_N^* \to \mathbb{Z}$ *with*

- $\|f\|(\epsilon) = 0$, *and*
- $\|f\|(\sigma t) = f(t, h(\sigma)) + \|f\|(\sigma)$ *if* $t \in T_N, \sigma \in T_N^*$.

Example 2. We define the cost functions mentioned in Sect. 1 for open net V in Fig. 1(a). For both functions, we need to know whether a coffee or a tea order has been received last. We cover those two cases by the hash values coffee and tea. For totality, we introduce a third hash value, other. We define a hash function $\mathsf{h} : T_\mathsf{V}^* \to \mathsf{H}$ with $T_\mathsf{V} = \{\mathsf{c}, \mathsf{t}, \mathsf{b}, \mathsf{q}\}$ and $\mathsf{H} = \{\mathsf{coffee}, \mathsf{tea}, \mathsf{other}\}$. We define the sets of occurrences of c and t in $\sigma \in T_\mathsf{V}^*$: $\mathsf{OC}_\sigma = \{i \mid \sigma(i) = \mathsf{c}\}$ and $\mathsf{OT}_\sigma = \{i \mid \sigma(i) = \mathsf{t}\}$.

$$\mathsf{h}(\sigma) = \begin{cases} \mathsf{other} & \text{if } \mathsf{OC}_\sigma = \mathsf{OT}_\sigma = \emptyset, \\ \mathsf{coffee} & \text{if } \mathsf{OC}_\sigma \neq \emptyset \wedge (\mathsf{OT}_\sigma \neq \emptyset \Rightarrow \max(\mathsf{OC}_\sigma) > \max(\mathsf{OT}_\sigma)), \\ \mathsf{tea} & \text{otherwise.} \end{cases}$$

Based on this hashing, we define the cost functions f and g over T_V with hashing h to specify the *expenses* and the *reward* for the provider to execute V:

- $\forall t \in T_\mathsf{V} \setminus \{\mathsf{b}\}, a \in \mathsf{H} : \mathsf{f}(t, a) = \mathsf{g}(t, a) = 0$,
- $\mathsf{f}(\mathsf{b}, \mathsf{coffee}) = 20$, $\mathsf{f}(\mathsf{b}, \mathsf{tea}) = 10$, $\mathsf{f}(\mathsf{b}, \mathsf{other}) = 0$, and
- $\mathsf{g}(\mathsf{b}, \mathsf{coffee}) = \mathsf{g}(\mathsf{b}, \mathsf{tea}) = 25$, $\mathsf{g}(\mathsf{b}, \mathsf{other}) = 0$.

The choice of the history hashing determines the class of cost functions that may be build. We propose to make use of *deterministic finite automata (DFA)*. A history may be interpreted as a word. Using a history σ as input for a DFA, the resulting state q may be understood as a hash value for σ. Such a history hashing obviously satisfies *continuity* as a DFA is deterministic and total. Additionally, since its set of states is finite, *finiteness* holds. Utilizing such a hashing, we can express any cost function where conditions consist of checking membership of the history in regular languages.

Constraints. We introduce *constraints* as restrictions on behavior by specifying *conditional bounds* for cost functions. As conditions, we use markings. As bounds, we use integer intervals. Intuitively, a transition sequence meeting the condition *satisfies* a constraint if its costs are inside given bounds. If the transition sequence results in a different marking or cannot be fired, it trivially satisfies the constraint.

Definition 3 (Constraints). *We call a pair* $p = \langle m, \tau \rangle$ constraint *over a set* G *of cost functions, iff* $m \in \mathcal{M}_{ip(N)}$ *is a marking of the inner process of* N *and* τ *maps each cost function* f *to an integer interval.*

A transition sequence $\sigma \in T_N^*$ satisfies $p = \langle m, \tau \rangle$, *written* $\sigma \models p$, *iff* $\sigma \in beh(ip(N), m) \Rightarrow \forall f \in G : \|f\|(\sigma) \in \tau(f)$.

Example 3. We model the constraint informally described in Sect. 1: Upon reaching a final marking, all expenses should be covered by the reward. We define this as one constraint p_m per final marking $m \in \Omega_{\mathsf{V}}$. Each p_m is defined over cost function $(\mathsf{g} - \mathsf{f})$ as defined in Example 2. At first glance, the acceptance interval for this cost function is $[0, \infty)$. However, we did not model the fix costs yet. We thus shift the acceptance interval by the difference of the fix costs yielding $\mathsf{i} = [5, \infty)$. We inspect some example firing sequences and decide constraint satisfaction for each. The firing sequence cb trivially satisfies each p_m since it does not result in a final marking. The firing sequences $\sigma = \mathsf{cbq}$ and $\sigma' = \mathsf{q}$ result in a final marking. While σ satisfies each p_m, σ' does not: $(\mathsf{g} - \mathsf{f})(\sigma) = 5 \in \mathsf{i}$, $(\mathsf{g} - \mathsf{f})(\sigma') = 0 \notin \mathsf{i}$.

Policies. A *policy* is basically a collection of constraints over a given set of cost functions. Policy violation requires a change of the viewpoint: Policies are defined over the behavior of a fixed open net N. However, policy violation is not a property of N but of a partner Q of N. By sending messages to N, Q may influence the control flow of N. Intuitively, Q *violates* a policy φ, if it sends messages, such that N *may* choose a firing sequence which does not satisfy all constraints in φ. Formally, we define policy satisfaction for a transition sequence of N and based thereon policy violation.

Definition 4 (Policies). *We define a* policy *as a tuple $\varphi = \langle N, h, G, C\rangle$ where G is a set of cost functions and C is a set of constraints over G. A transition sequence $\sigma \in T_N^*$* satisfies *φ, written $\sigma \models \varphi$, iff $\forall p \in C : \sigma \models p$. A partner Q of N* violates *φ iff there exists a firing sequence σ of $N \oplus Q$, such that $\sigma_{|T_N} \not\models \varphi$.*

Example 4. We combine the open net V from Fig. 1(a), the hashing h from Example 2, the cost functions f, g from Example 2, and the constraints p_m ($m \in \Omega_{\mathsf{V}}$) from Example 3 to *policy* $\varphi_{\mathsf{V}} = \langle \mathsf{V}, \mathsf{h}, \{(\mathsf{g} - \mathsf{f})\}, \{\mathsf{p}_m \mid m \in \Omega_{\mathsf{V}}\}\rangle$. We find $\mathsf{cbq} \models \varphi_{\mathsf{V}}$ and $\mathsf{q} \not\models \varphi_{\mathsf{V}}$. Consider open net D from Fig. 1(b) as a partner for V. We answer the question if D violates φ_{V} or not: There are two firing sequences σ, σ' of $\mathsf{V} \oplus \mathsf{D}$ which result in a final marking of V: $\sigma_{|T_{\mathsf{V}}} = \mathsf{cbq}$, and $\sigma'_{|T_{\mathsf{V}}} = \mathsf{tbq}$.

To decide whether D violates φ_{V}, we need to decide $\sigma_{|T_{\mathsf{V}}} \models \varphi_{\mathsf{V}} \wedge \sigma'_{|T_{\mathsf{V}}} \models \varphi_{\mathsf{V}}$ which boils down to deciding $\forall m \in \Omega_{\mathsf{V}} : \sigma_{|T_{\mathsf{V}}} \models \mathsf{p}_m \wedge \sigma'_{|T_{\mathsf{V}}} \models \mathsf{p}_m$. D does not violate φ_{V} because $\{(\mathsf{g} - \mathsf{f})(\sigma_{|T_{\mathsf{V}}}), (\mathsf{g} - \mathsf{f})(\sigma'_{|T_{\mathsf{V}}})\} = \{5, 15\} \subseteq [5, \infty)$.

4 Toward deciding policy violation

In this section, let N be an open net and $\varphi = \langle N, h, G, C\rangle$ be a policy. Our approach follows three steps: (1) Compute the *φ-state space* $\mathcal{S}(\varphi)$. (2) Finitely represent all φ-violating behavior, yielding $\mathcal{V}(\varphi)$. (3) Decide policy violation utilizing $\mathcal{V}(\varphi)$.

The φ-state space. The state space of a system is usually a directed graph where each vertex represents a state of the system and each edge stands for a transition

from the source state to the target state. In case of a Petri net, a state is a marking and an edge is a transition. Many properties may be decided on the state space by exploring the set of all states or their order.

Our property of interest is the following: Which firing sequences satisfy the given policy? The problem is that even a finite state space generally may represent infinitely many firing sequence due to cyclic behavior. This can be easily overcome if it is possible to transform the property into a state property, i.e. if it is sufficient to inspect a state to conclude if the firing sequences resulting in this state have the property or not.

We intend to do the same trick for our property of interest. In a first step, we enrich states with the so far incurred costs for each cost function. However, this is not sufficient: Let σ, σ' be firing sequences resulting in the same costs and marking. Let σ'' be a transition sequence, such that $\sigma\sigma''$ and $\sigma'\sigma''$ are firing sequences again. Then, $\sigma\sigma''$ and $\sigma'\sigma''$ do not necessarily result in the same costs again.

Example 5. Consider the firing sequences $\sigma_1 = \mathsf{cbc}$ and $\sigma_2 = \mathsf{cbt}$ of the inner process of open net V from Fig. 1(a). According to cost function f from Example 2, both σ_1 and σ_2 result in the same state: Firing yields obviously the same marking and the same costs of 20 units. However, continuing with b, we find that $f(\mathsf{cbcb}) = 40 \neq f(\mathsf{cbtb}) = 30$.

We thus add the hash value of the history. Formally, we define a φ-state as a triple $q = \langle m, x, \tau_q \rangle$ consisting of a marking m, a hash value x, and a mapping $\tau_q : G \to \mathbb{Z}$.

Example 6. We continue Example 6. According to the history hashing h from Example 2, the hash values for σ_1 and σ_2 are different: $\mathsf{h}(\sigma_1) = \mathsf{coffee} \neq \mathsf{h}(\sigma_1) = \mathsf{tea}$. We can distinguish the resulting states of σ_1 and σ_2 by their hash values.

Given a φ-state q, it is trivial to decide if the firing sequences resulting in q satisfy the policy or not: We check membership of the current values given by τ_q with the intervals given by τ for each constraint $\langle m, \tau \rangle$. Additionally, if the *φ-state space* $\mathcal{S}(\varphi)$ is finite, we may compute it with a depth first search, thereby exploiting property *Continuity*. Property *Finiteness* ensures that the set S of reachable states is finite iff $\{\langle m, \tau_q \rangle \mid \langle m, x, \tau_q \rangle \in S\}$ is. Given $\mathcal{S}(\varphi)$, we may compute $\mathcal{V}(\varphi)$, similarly as in [8]. So far, we do not have a solution for infinite φ-state spaces.

Deciding φ-violation. We finitely represent the *φ-violating partner behavior* as a finite automaton $\mathcal{V}(\varphi)$. Thereby, a word represents partner behavior: $?a!b?c$ stands for receiving a, followed by sending b and receiving c. We can represent the set of traces of the inner process of any open net Q as a finite automaton $\mathcal{A}(Q)$, if it is finite state. By setting the set of final states to the complete state set, policy violation may be decided by comparing the languages of $\mathcal{V}(\varphi)$ and $\mathcal{A}(Q)$. If their intersection is non-empty, Q *violates* φ.

Example 7. Figure 1(c) shows $\mathcal{V}(\varphi_{\mathsf{V}})$ of φ_{V} from Example 4, Fig. 1(d) shows $\mathcal{A}(\mathsf{D})$ of D from Fig. 1(b). $\mathcal{V}(\varphi_{\mathsf{V}})$ and $\mathcal{A}(\mathsf{D})$ do not share an accepting run, φ_{V} is not violated.

Similarly, we believe that $\mathcal{V}(\varphi)$ may be used as a *constraint automaton* as introduced in [9] to compute policy-aware operating guidelines. The result may then be used to decide policy violation and behavioral compatibility in one step.

5 Conclusion and future work

We provided a formal framework to specify policies φ, describing acceptable behavior of a partner based on cost functions and constraints. We explained how a φ-state space may be computed and processed if it is finite. We sketched a decision procedure based on this representation. In the future, we aim at solving the problem that the φ-state space is not necessarily finite. We intend to apply techniques similar to the coverability graph for Petri nets. We plan to extend our proof of concept implementation to evaluate the practical usability of our approach with a case study.

References

1. Kindler, E.: A compositional partial order semantics for Petri net components. In: ATPN'97. Volume 1248 of LNCS. (1997) 235–252
2. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Trans. Software Eng. **30**(5) (2004) 311–327
3. Ding, W.: Services pricing through business value modeling and analysis. In: IEEE SCC. (2007) 380–386
4. Schuller, D., Miede, A., Eckert, J., Lampe, U., Papageorgiou, A., Steinmetz, R.: Qos-based optimization of service compositions for complex workflows. In Maglio, P.P., Weske, M., Yang, J., Fantinato, M., eds.: ICSOC. Volume 6470 of Lecture Notes in Computer Science. (2010) 641–648
5. Gierds, C., Sürmeli, J.: Estimating costs of a service. In Gierds, C., Sürmeli, J., eds.: Proceedings of the 2nd Central-European Workshop on Services and their Composition, ZEUS 2010, Berlin, Germany, February 25–26, 2010. Volume 563 of CEUR Workshop Proceedings., CEUR-WS.org (2010) 121–128
6. Sampath, P., Wirsing, M.: Computing the cost of business processes. In: UNISCON. (2009) 178–183
7. Magnani, M., Montesi, D.: BPMN: How much does it cost? An incremental approach. In: BPM. (2007) 80–87
8. Wolf, K.: Does my service have partners? LNCS ToPNoC **5460**(II) (March 2009) 152–171 Special Issue on Concurrency in Process-Aware Information Systems.
9. Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services. In Alonso, G., Dadam, P., Rosemann, M., eds.: BPM. Volume 4714 of Lecture Notes in Computer Science., Springer (2007) 271–287