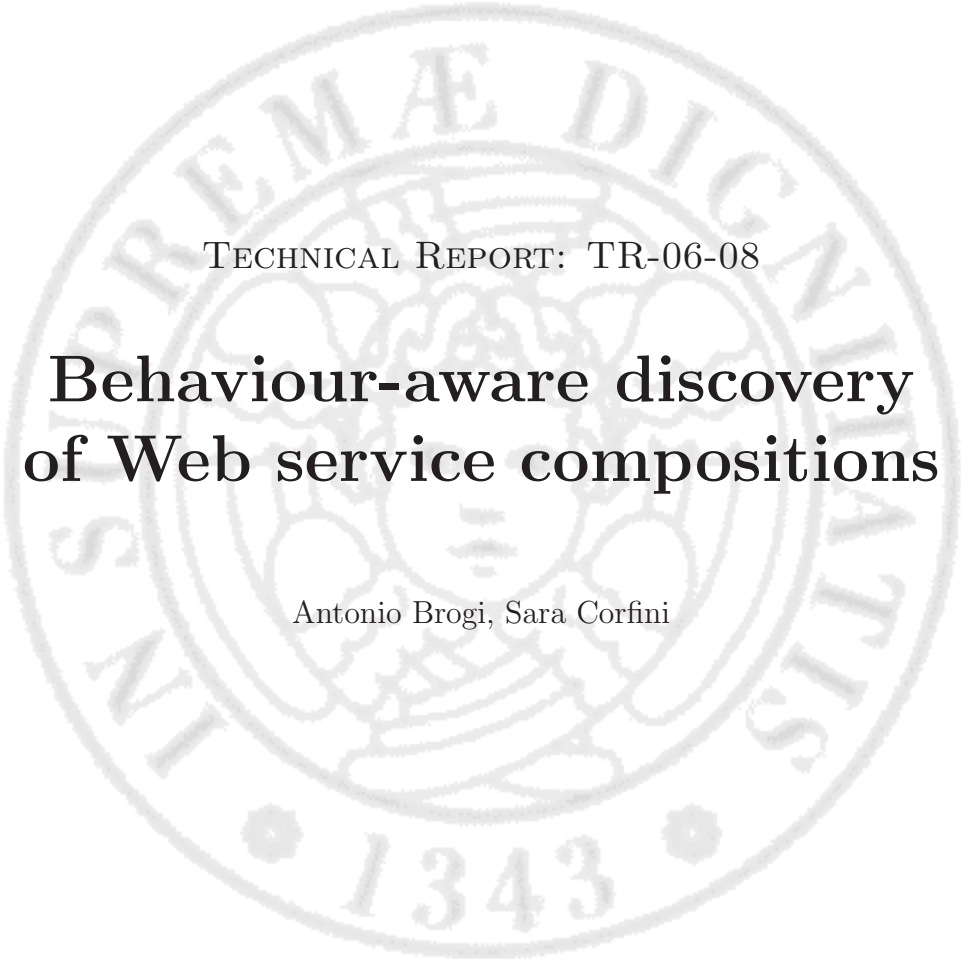
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-06-08

Behaviour-aware discovery of Web service compositions

Antonio Brogi, Sara Corfini

June 23, 2006

Address: via F. Buonarroti 2, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

Behaviour-aware discovery of Web service compositions

Antonio Brogi, Sara Corfini

June 23, 2006

Abstract

A major challenge for Service-oriented Computing is how to discover and compose (Web) services to build complex applications. We present a matchmaking system that exploits both semantics and behavioural information to discover service compositions capable of satisfying a client request.

Keywords: *Web service discovery*, *Web service composition*, *Petri nets*, *OWL-S ontologies*.

1 Introduction

The Web is rapidly evolving from being a collection of static information to a collection of services which interoperate through the Internet. Recently, increasing attention is devoted to Service-oriented Computing (SoC) [24], a new emerging paradigm for distributed computing whose best-known instantiation is represented by Web services. Web services are software components that, thanks to their platform neutral and self-describing nature, should allow to construct complex applications faster and with less programming efforts. The current Web services infrastructure relies on WSDL [38], SOAP [36] and UDDI [33]. WSDL is a XML-based language for describing what a service does and how to invoke it. SOAP is a standard protocol for exchanging messages over HTTP between applications. UDDI allows for the definition of global registries where information about services are published. Currently, UDDI is the only universally accepted standard for Web service discovery.

Unfortunately, the current service infrastructure suffers from two main limitations: it does not support service composition and it does not account for semantics information. On the one hand, assuming that for each service request there exists a single Web service that perfectly satisfies it on its own, is rather unrealistic. In many cases, composing functionalities offered by different services may be needed to satisfy a client request. On the other hand, the availability of machine-understandable service descriptions is a must for automatising the processes of service discovery and composition. Regrettably, available WSDL

interfaces provide neither semantics information to describe the service functionality nor behavioural information to describe the service interaction behaviour.

The problem of how to automate the composition of Web services has recently attracted quite some attention, as witnessed for instance by the definition of BPEL4WS [2] and OWL-S [22], which are two XML-based languages for describing services. Both BPEL4WS and OWL-S allow to describe behavioural information about services, and OWL-S also allows to specify semantics information about them. Generally speaking, most approaches aim at overcoming either of the two above mentioned limitations. Some of them introduce semantics information to improve service discovery (not considering service composition issues), while others focus on composition issues (not considering semantics).

We argue that both semantics and behavioural information should be taken into account in order to automate the discovery of service compositions. Semantics information can be fruitfully exploited for discovering (candidate) services, while behavioural information can be fruitfully exploited to compose them in a correct way.

In this perspective, in [10] we presented an algorithm for the composition-oriented discovery of Web services. Such algorithm performs a flexible matching over a registry of OWL-S service advertisements – considering both semantics and behavioural information – and determines whether there exists a service composition capable of satisfying a client request. The algorithm in [10], however, has two drawbacks. The first one is efficiency, as a dependency graph representing the behaviour of each service in the registry must be constructed at query answering time. The second drawback is that the algorithm guarantees neither the deadlock-freedom nor the minimality of the returned service composition.

In this paper we present a Petri net-based matchmaking system that overcomes the above mentioned drawbacks of [10]. Our system takes advantage of both semantics and behavioural information advertised in OWL-S service descriptions. The main features of the proposed matchmaking system can be summarised as follows:

- Our system is strongly based on behavioural information, as it models services as Petri nets. The expressive power of Petri nets allows to easily model complex service compositions as well.
- Petri net representations of services can be pre-computed and stored together with service descriptions, without affecting the efficiency of the matchmaking process.
- The control flow verification of Petri nets allows to determine whether or not the services in a composition terminate correctly.
- Last, but not least, the returned composition does not contain services that are not strictly necessary to satisfy the query.

The rest of the paper is organised as follows. In Section 2 we briefly introduce OWL-S together with a motivating example, that we will use to illustrate our

approach throughout the paper. In Section 3 we show how OWL-S behavioural descriptions can be translated into Petri nets. Section 4 is devoted to present the architecture and the behaviour of the matchmaking system for the discovery and composition of services. Related works are discussed in Section 5, while some concluding remarks are drawn in Section 6.

2 Matching services with OWL-S

The currently adopted standards for Web services (UDDI [33], SOAP [36] and WSDL [38]) do not deal with semantics information. To overcome the consequent inaccuracy of service discovery, the W3C consortium promotes the adoption of new semantic-based formalisms for describing services. OWL-S [22] is an ontology for semantically describing Web services. An OWL-S advertisement of a service consists of the following three parts:

1. *Service profile* – which provides a high level description of the service, containing information such as its inputs and outputs (i.e., the *functionality* of the service), other extra-functional attributes as well as a further human readable service description;

2. *Process model* – which describes the service behaviour providing a view of the service in terms of process compositions. OWL-S defines three types of processes:

   - *atomic* processes, which have associated inputs and outputs and can be directly invoked by the client,
   - *composite* processes, which consist of other composite and atomic processes, and
   - *simple* processes, which are an abstract and simplified view of a composite process;

3. *Service grounding* – which describes how to access and interact with the service by specifying protocol and message format information.

By adopting OWL-S as a language for describing services, we aim at deploying a matchmaking system which allows to satisfy a client query with a (composition of) service(s). Given a query, the matchmaker should be able of selecting the services that can be useful to satisfy the query, as OWL-S profiles provide a semantic description of the functional attributes of services. Moreover, the matchmaker should be capable to find a composition of such services that effectively fulfills the query, as OWL-S process models provide a behavioural description of services.

Example. Let us consider a registry containing three services described by OWL-S: `Photo_Service`, `Online_Bank` and `Prepaid_Cards`. The first prints photos in different formats and delivers them to the address specified by the client. The second allows the client to obtain virtual credit cards via a bank

transfer, by specifying the desired card capacity. Finally, the third service releases prepaid credit cards of fixed capacity through a bank transfer. The process models of `Photo_Service`, `Online_Bank` and `Prepaid_Cards` are shown in Figure 1, 2 and 3, respectively.

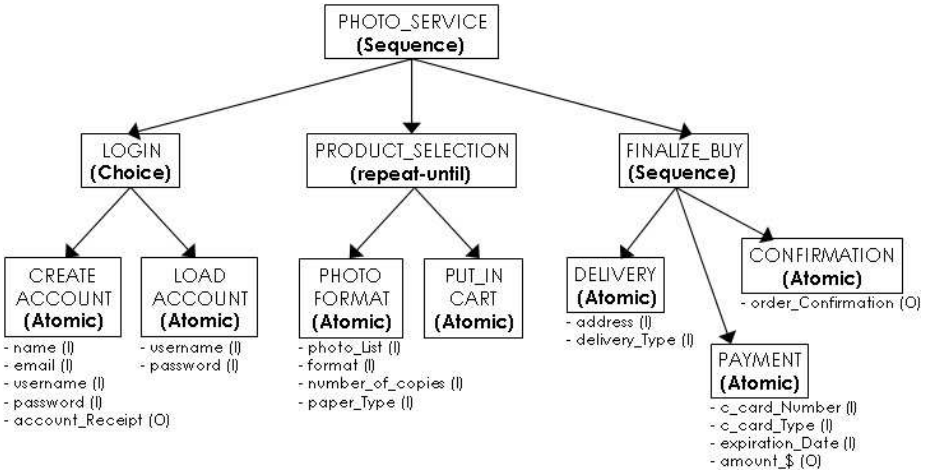


Figure 1: Process model of `Photo_service`.

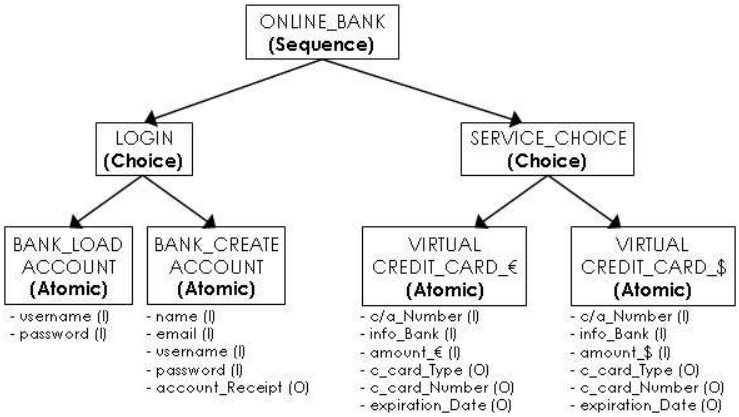


Figure 2: Process model of `Online_Bank` service.

`Photo_Service` starts with the authentication of the client, that can choose between logging in to an existing account or creating a new account. Next, the service continues with the printing phase, during which the client provides its print preferences and adds the preferred photos to its order. Finally, the service performs the selling phase asking the user for the delivery type and payment. From the point of view of its OWL-S process model, `Photo_Service` is a

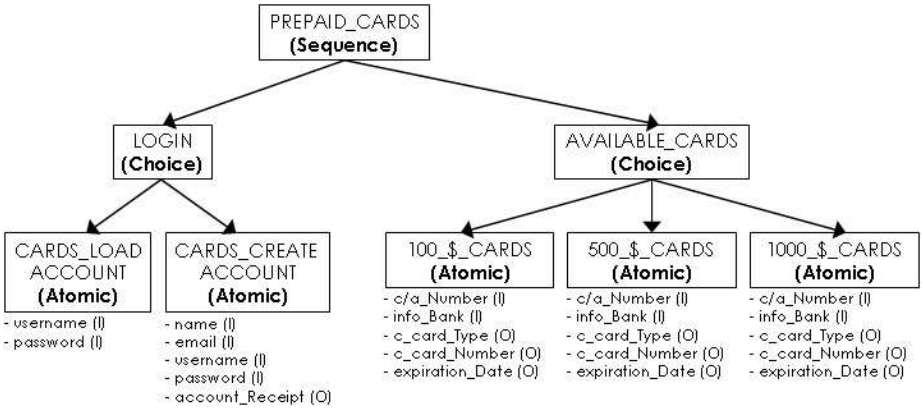


Figure 3: Process model of `Prepaid_Cards` service.

`sequence` process composed of a `choice` process, a `repeat_until` process and a `sequence` process. The `choice` process, corresponding to the logging phase, is composed of two atomic processes, `create_account` and `load_account`. The `repeat_until` process modelling the printing phase, is composed in turn of two atomic processes, `photo_format` and `put_in_cart`. The `sequence` process representing the selling finalisation, is composed of three atomic processes, `delivery`, `payment` and `confirmation`.

`Online_Bank` starts with the authentication of the client, that, again, can choose between logging in to an existing account or creating a new account. Next, the client can obtain a virtual credit card, providing its bank coordinates and choosing the preferred currency. `Online_Bank` is a `sequence` process composed of two `choice` processes. The former consists of `bank_load_account` and `bank_create_account` atomic processes and the latter is composed of `virtual_credit_card_€` and `virtual_credit_card_$` atomic processes.

`Prepaid_Cards` starts with the authentication of the client, and continues with the release of prepaid credit cards. After providing its bank coordinates, the client can choose among different sizes of prepaid cards. `Prepaid_Card` is modelled as a `sequence` of two `choice` processes. The former consists of `cards_load_account` and `cards_create_account` atomic processes and the latter is composed of `100_$_cards`, `500_$_cards` and `1000_$_cards` atomic processes.

Consider now the query specifying:

- inputs: `username`, `password`, `c/a_Number`, `info_Bank`, `photo_List`, `paper_Type`, `format`, `number_of_copies`, `delivery_Type`, `address`, and
- output: `order_Confirmation`.

As one may notice, while none of the services satisfy the query, the latter can be fulfilled by composing the three presented services. To be more exact, the query can be satisfied by a composition of `Photo_Service` and `Prepaid_Cards`. Indeed, the other composition that seems to be able of satisfying the query,

that is, `Photo_Service` and `Online_Bank`, fails as some inputs (`c_card_Type`, `c_card_Number` and `expiration_Date`) required by `Photo_Service` are produced as output by `Online_Bank`, that in turn requires as input an output (`amount_$`) generated by `Photo_Service`. Therefore, the two services end up in deadlock. As we will see, thanks to the semantic matching and to the analysis of service behaviour, our matchmaking system is capable of returning a *lock-free* (composition of) service(s) that is really able of satisfying the client query.

◇

3 From OWL-S to Petri nets

In this section, before showing how OWL-S behavioural descriptions can be mapped into Petri nets, we briefly recall the essence of Petri nets.

Petri nets [25] have been introduced by Carl Adam Petri for modelling concurrent behaviour of a distributed system. We hereafter include the formal definition of a Petri net, as described in [17].

Def. 1 *A* place/transition Petri net *is a 5-tuple,* $PN = \{P, T, F, W, M_0\}$ *where:*

1) $P = \{p_1, p_2, ..., p_m\}$ *is a finite set of* places,
2) $T = \{t_1, t_2, ..., t_n\}$ *is a finite set of* transitions,
3) $F \subseteq (P \times T) \cup (T \times P)$ *is a set of* arcs,
4) $W : F \to \mathbb{N}^+$ *is a* weight function,
5) $M_0 : P \to \mathbb{N}$ *is the* initial marking,
6) $P \cap T = \varnothing$ *and* $P \cup T \neq \varnothing$,

A Petri net is a directed, weighted and bipartite graph whose nodes can be distinguished in two non-empty and disjoint sets (6), namely *places* (1) and *transitions* (2). Places are connected to transitions as well as transitions are connected to places by means of directed (3) and weighted (4) *arcs*. Hence, an arc can connect only two (differently typed) nodes. For each $x \in P \cup T$, we denote by ${}^\bullet x = \{y | (y, x) \in F\}$ the *pre-set* of x and by $x^\bullet = \{y | (x, y) \in F\}$ the *post-set* of x. By convention, places and transitions are graphically represented by circles (or ellipses) and rectangles, respectively.

Petri nets simulate the dynamic behaviour of a system by introducing *tokens*, which are objects residing inside places and graphically represented by solid dots. Places can hold an arbitrary number of tokens. In place/transition Petri nets, a *token* is a marker whose presence/absence indicates the availability/unavailability of whatever it represents, e.g., a condition, a resource, a signal and so on.

A *marking* of a Petri net is a mapping $P \to \mathbb{N}$ which assigns a non-negative integer number of tokens to each place of the net. A marking represents the state of the Petri net, that changes whenever tokens modify their distribution.

The initial state of the Petri net corresponds to the provided initial marking (5). A marking M evolves according to the following *transition firing rules*, where $w(p,t)$ denotes the weight of the arc (p,t):

- A transition t is *enabled* if for each place $p \in {}^\bullet t$, $w(p,t) \leq M(p)$.
- An enabled transition t can *fire*. Then, it removes $w(p,t)$ tokens from each place $p \in {}^\bullet t$, and adds $w(p,t)$ tokens to each place $p \in t^\bullet$.

To represent an OWL-S process model with Petri nets, we consider atomic processes as transitions and we model both data flow and control flow relations among processes with Petri nets transition firing rules.

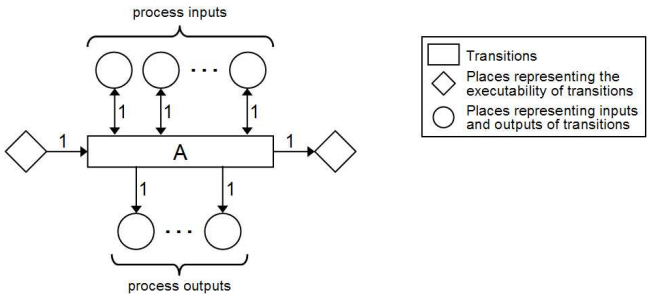


Figure 4: Modelling OWL-S atomic processes as Petri net transitions.

An atomic process can be executed only if the following two conditions occur:

1) all of its inputs are available, and

2) all processes to be executed before it have been completed.

Both conditions are represented by the availability of tokens in those places which belong to the pre-set of a transition. Indeed, as illustrated in Figure 4, we represent an atomic process A as a transition t having a place $p \in {}^\bullet t$ for each input of A as well as a place $p \in t^\bullet$ for each output of A. We also introduce two further places: one in the pre-set ${}^\bullet t$ to denote that t is executable, and the other in the post-set $t^\bullet$ to denote that t has completed its execution. When the transition t is enabled to fire, that is, when each place $p \in {}^\bullet t$ is marked with at least $w(p,t)$ tokens, it means that both the conditions (1) and (2) occur.

It is worth noting that in Figure 4 places representing inputs and outputs of transitions (i.e., of atomic processes) are depicted as circles, whereas places representing the executability of transitions are depicted as diamonds. Yet, it is important to stress that circles and diamonds are only a graphical convention which we adopt to help the reader and that there is no distinction between places (and tokens) in the proposed Petri net representation of OWL-S process models.

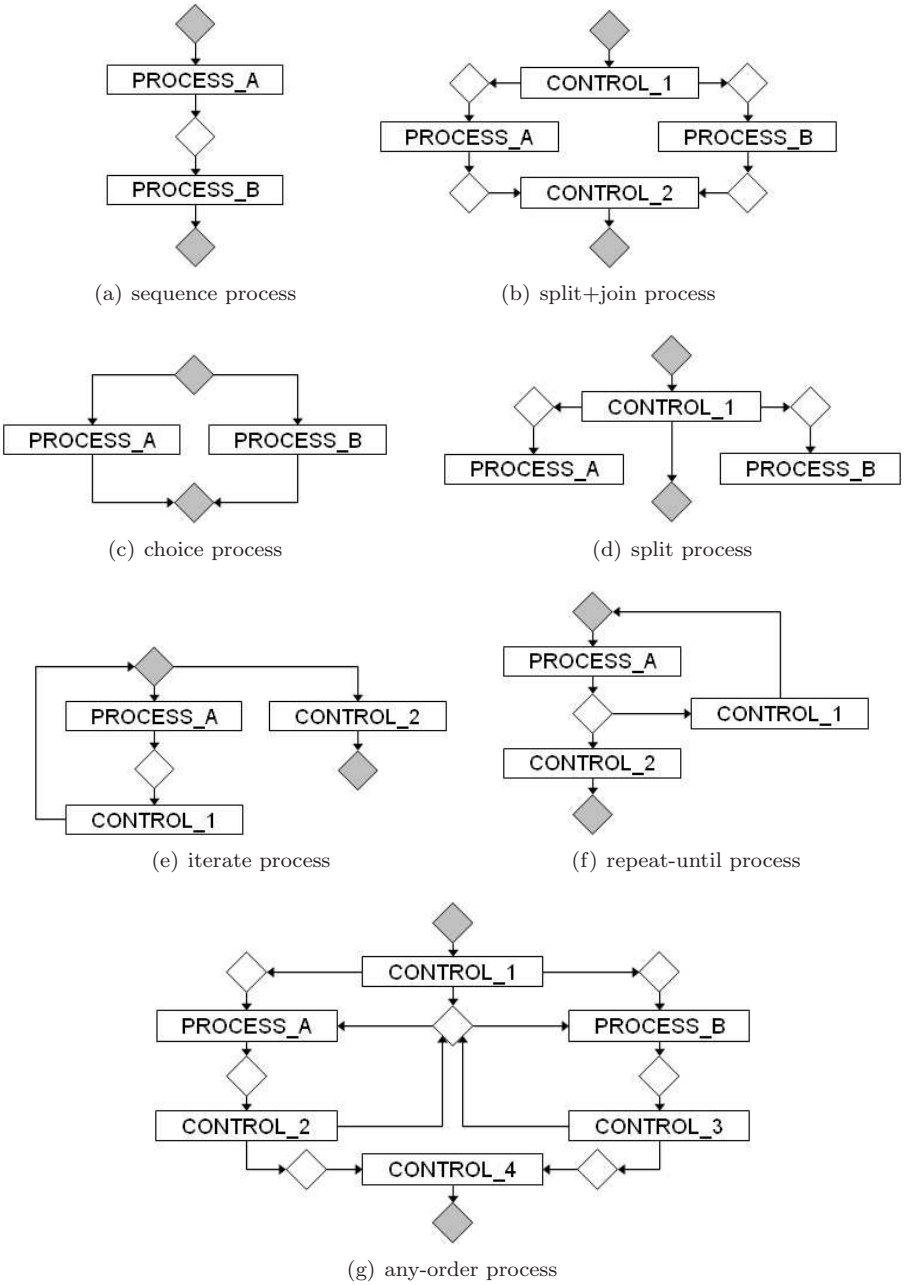
It is also worth noting that a transition t is linked to each circle-shaped place which belongs to ${}^\bullet t$ through a double arc. A double arc is a shorthand

for two opposite directed arcs sharing the same weight. More precisely, we have inserted the directed arc (t, p) in order to make the input which p represents also available after the firing of t. As we will better explain in the following sections, a circle-shaped place p could belong to the pre-set of $n > 1$ transitions, and each of them could need the input represented by p. Finally, it is worth observing that we use *ordinary* Petri nets, as they involve only 1-weighted arcs. We will hence omit the arc weights in the figures included in the rest of the paper.

The control flow of a service presented by an OWL-S advertisement is described in the process model part. The service is depicted as a composite process composed of other composite or atomic processes. OWL-S defines the following types of composite processes [22]:

- a `sequence` process is a list of processes to be executed in order;
- an `any-order` process is a bag of processes to be executed in some unspecified order but not concurrently;
- a `split` process is a bag of processes to be executed concurrently;
- a `split+join` process is a bag of processes to be executed concurrently with barrier synchronization;
- a `choice` process is a bag of processes out of which only one can be chosen for execution;
- an `if-then-else` process is a bag of two processes out of which one is chosen for execution according to the value of a condition;
- an `iterate` process is a processes to be executed, without specifying how many iterations have to be done;
- a `repeat-while` process is a process to be executed zero or more times, until a condition becomes false.
- a `repeat-until` process is a process to be executed at least once, until a condition becomes true.

OWL-S composite processes can be directly represented as Petri nets, as illustrated in Figure 5. As previously described, diamond-shaped places represent the executability of transitions and they contribute to the control flow management. We have emphasised in light gray those used for composing different Petri nets. Instead, circle-shaped places representing inputs and outputs of transitions have been omitted to simplify reading. Transitions indicated as PROCESS_*X* identify processes. Transitions named CONTROL_*N* identify empty processes added for managing the control flow. The Petri net representing a `choice` process is equivalent to the one for an `if-then-else` process as in both cases only one process is chosen for execution. In the `choice` case, the process is extracted from a bag of two or more processes, while in the `if-then-else`



(a) sequence process

(b) split+join process

(c) choice process

(d) split process

(e) iterate process

(f) repeat-until process

(g) any-order process

Figure 5: Translation diagrams for building a Petri net.

case it is extracted from a bag of exactly two processes. Moreover, the Petri

net for an `iterate` process is equivalent to the one for a `repeat-while` process. Indeed a `repeat-while` process differs from an `iterate` process because of the condition specifying the number of iterations to be done. Nevertheless, a Petri net defines the control flow of a process, which is identical for both `iterate` and `repeat-while` processes, as it is independent from the number of iterations.

4 Architecture of the matchmaker

The matchmaking system we propose consists of three independent modules (Figure 6): a *translator* from OWL-S process models to Petri nets, a *functional analyser* which filters services taking into account their functional attributes, and a *behavioural analyser* which after merging together a set of (selected) Petri nets, checks for a positive match by animating the composite Petri net.

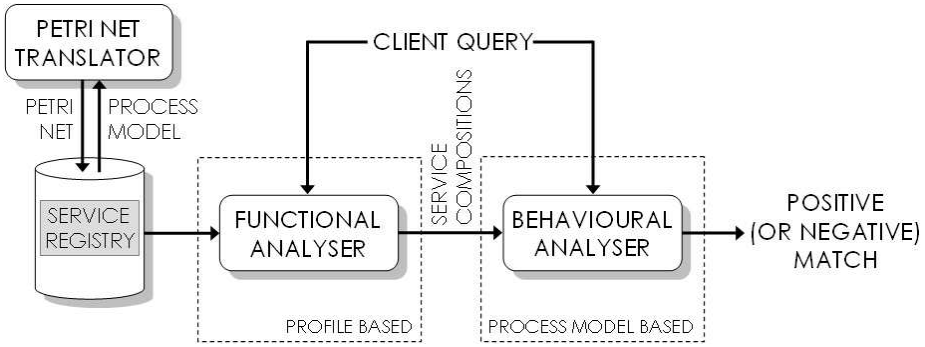


Figure 6: Matchmaking system architecture.

In general terms, the system behaviour is the following. We consider a registry that for each stored service contains its OWL-S description and its Petri net representation. Every time a service is added to the registry, the translator loads the OWL-S process model of the service and generates its Petri net representation according to the translation diagrams shown in Figure 5. The functional analyser takes as input a client query, analyses the OWL-S profiles of all services contained in the registry and returns an ordered list of all the possible sets of services that may be useful to satisfy the query. Next, the matchmaking system analyses (the Petri net representation of) such sets of services until it finds one (if any) capable to fulfill the request. In order to establish whether a service set can really satisfy the query, the behavioural analyser firstly merges together the Petri nets of the services contained in the candidate set. Next, thanks to the control flow verification of the global Petri net, it determines whether the services in the set can really be composed together and generate the query outputs without dead-locking. If so, the behavioural analyser returns a positive match to the client, otherwise it analyses the next service set.

The following subsections are devoted to a detailed explanation of the behaviour of both the functional and behavioural analysers.

4.1 The functional analyser

The functional analyser takes as input a client query, which consists of the required inputs and the produced outputs of the service which the client is searching for. This module selects the services which may be useful to satisfy the query by taking into account their functional attributes (i.e., inputs and outputs described in their service profiles) and it returns as output an ordered list of all the possible compositions of such services. The functional analyser performs two main steps.

Step 1. The first step performs a registry inspection in order to synthesise the information about the outputs produced by the available services. Such information is stored in a data structure, called $outputRegistry$, which associates each output o with the set of services $outputRegistry[o]$ that generate it.

As previously mentioned, the functional analyser takes into account the semantic information exposed in the profiles advertising the registered services. Indeed, it exploits the semantic relationships existing among the functional service attributes and defined in the ontologies referred by the service profiles.

According to the OWL-S specification [22], two processes are type compatible if for each output of one that flows to the input of the other, the type of the output is a subtype of the type of the input, that is, the output is a *sub-concept* of the input.

We now extend the notion of sub-concept over different ontologies. In the following definition, *sub-concept-of*$_O$ denotes the (reflexive) sub-concept-of relation defined within an ontology O, while $\equiv$ denotes the equivalence relation among concepts belonging to different ontologies.

Def. 2 *A concept c is a* sub-concept-of *a concept d, and we write $c \sqsubseteq d \iff$*

1) c sub-concept-of$_O$ d, or

2) $c \equiv c'$ *and* c' sub-concept-of$_O$ d, or

3) c sub-concept-of$_O$ d' *and* $d' \equiv d$

Hence, c is compatible with d if c is a sub-concept of d in some ontology O (1), or if c is equivalent to c' and c' is a sub-concept of d in some ontology O (2), or if c is a sub-concept of a d' in some ontology O and d is equivalent to d' (3).

It is worth noting that the notion of equivalence we consider is a *semantic* equivalence, i.e., the equivalence existing among concepts belonging to one or more ontologies. For instance, the *country* concept defined in an ontology should be equivalent to the *nation* concept defined in an another ontology. Semantic equivalence can be computed for example by using the method of *semantic fields* proposed in [19, 18] by Aldana et al. and defined in terms of the notions of semantic distance and ontology neighbourhood.

The behaviour of the first step can be roughly summarised as follows:

```
1. forall service s in Registry do
2.   forall output o in s.outputs do
3.     if (o ∉ outputRegistry) then
4.       forall service t in Registry do
5.         forall output q in t.outputs do
6.           if q ⊑ o then outputRegistry[o] = outputRegistry[o] ∪ {t};
```

This step hence associates each output o produced by the registered services (lines 1–2) with those services that generate (at least) one output which is a subconcept of o (line 4–6). On the other hand, as different services may use different ontologies, there are generally services that produce syntactically different, but semantically compatible, outputs.

It is worth observing that while a service is associated with an unique process model, it may be described by several service profiles [22]. Indeed, because of the non-determinism modelled by its process model, a service may behave in different ways and feature different functionalities. For instance, consider a service S_1 taking as input either A or B (i.e., `choice` process) and producing as output C. In order to correctly advertise S_1, two profiles must be exposed: one describing a service which takes A and returns C, and the other describing a service which takes B and returns C. Obviously the process model of S_1 is unique and it corresponds to a `choice` process composed of two atomic processes. Therefore, different profiles of the same service are to be considered as different services during the functional analyses (even if they correspond to the same process model).

Finally, it is important to observe that this first step is completely independent of the client query. As a consequence, all the information collected in *outputRegistry* can be pre-computed before query time. Namely, *outputRegistry* is updated every time a service is added to the registry.

Step 2. Once the information about outputs is available, the second step of the functional analyser can start. Its aim is to compute all possible sets of services capable to satisfy the client query.

We first formally define when a set of services may satisfy a query.

Def. 3 *A set of services* S may satisfy *a query* $Q \iff$

1) $\forall o \in Q.outputs,\ \exists x \in \bigcup_{s \in S} s.outputs : x \sqsubseteq o$, and

2) $\forall i \in \bigcup_{s \in S} s.inputs,\ \exists x \in (\bigcup_{s \in S} s.outputs \cup Q.inputs) : x \sqsubseteq i$

Namely, we say that a set of services S may satisfy a query Q if and only if (1) every query output is *subsumed* by a concept produced by some service in S, and (2) every service input is *subsumed* either by a query input or by an output of some service in S.

Note that Definition 3 refers to the functional attributes of services (i.e., service profiles) and it does not consider service behaviour, which will be considered by the next module (viz., the behavioural analyser). Therefore, a set of services that may satisfy a query (according to Definition 3) may possibly lock during their interaction.

The functional analyser explores all the sets of services by means of a recursive function SELECT. As we will show later, such function determines all the minimal sets of services that may satisfy the query.

The following pseudo-code summarises the behaviour of SELECT, which inputs five parameters: the *outputRegistry*, the query Q, the set of services found so far (*serviceSet*, initially empty), the set *o_needed* of outputs to be generated (initially the query outputs), and the set *o_available* of outputs available (initially the query inputs).

```
1.  SELECT(outputRegistry, Q, serviceSet, o_needed, o_available)
2.      if (o_needed = ∅) then return serviceSet;
3.      else
4.          out = extract(o_needed);
5.          if (out ∉ outputRegistry) then fail;
6.          else
7.              forall service s in outputRegistry[out] do
8.                  serviceSet' = serviceSet ∪ {s};
9.                  forall service t in serviceSet do
10.                     R = serviceSet' \ {t};
11.                     if (∄x ∈ t.outputs :
12.                         ∃y ∈ (Q.outputs ∪ {u|u ∈ ⋃_{r∈R} r.inputs ∧ ∄v ∈ Q.inputs : v ⊑ u}) :
13.                             x ⊑ y ∧ ∄w ∈ ⋃_{r∈R} r.outputs : w ⊑ y) then fail;
14.                 o_available' = o_available ∪ s.outputs;
15.                 o_needed' = {x|x ∈ o_needed ∪ s.inputs ∧ ∄y ∈ o_available' : y ⊑ x};
16.                 SELECT(outputRegistry, Q, serviceSet', o_needed', o_available');
```

If there are no outputs to be generated ($o\_needed = \emptyset$) then SELECT returns the set of services found (line 2). Otherwise (line 3), it withdraws an output *out* from the set *o_needed* (line 4). If there is no entry for *out* in *outputRegistry* (i.e., *out* can not be generated by any service in the registry) then SELECT fails since the query can not be satisfied (line 5). Otherwise (line 6) for each service s that generates *out* (line 7), SELECT adds s to the set of services (line 8), adds the set *s.outputs* to the outputs available (line 14), and updates the outputs needed (line 15) by adding *s.input* and by removing the concepts that are now available. After this, SELECT continues recursively (line 16).

Let us ignore for a moment the loop at lines 9–13 whose role is, as we will see later, to discard (by failing) all the non-minimal sets of services that may satisfy the query.

We first observe that whenever a recursive call SELECT(*outputRegistry*, Q, *serviceSet*, *o_needed*, *o_available*) is issued, the following invariant property holds:

$$o\_needed = \{x | x \in (Q.outputs \cup \{u | u \in \bigcup_{t \in S} t.inputs \wedge \nexists v \in Q.inputs : v \sqsubseteq u\}) \wedge \nexists y \in \bigcup_{t \in S} t.outputs : y \sqsubseteq x\}$$

where S is a shorthand for *serviceSet*.

Hence, when $o\_needed = \emptyset$ we have that:

1) $\forall x : Q.outputs \Rightarrow \exists y \in \bigcup_{t \in S} t.outputs : y \sqsubseteq x$, and
2) $\forall x : \bigcup_{t \in S} t.inputs \Rightarrow \exists y \in (Q.inputs \cup \bigcup_{t \in S} t.outputs) : y \sqsubseteq x$

that is, exactly the two conditions of Definition 3 hold. Hence the above invariant property guarantees that when $o\_needed = \emptyset$ then *serviceSet* is a set of services that may satisfy the query.

It is also worth observing that if there is an entry for $out \in o\_needed$ in *outputRegistry* (line 6), then none of the services in *outputRegistry*[*out*] (line 7) is already part of *serviceSet* (otherwise *out* would not still belong to *o_needed*). Finally, it is easy to observe that since the set of services and the set of their inputs and outputs are all finite, then SELECT always terminates.

Let us now consider the remaining lines 9–13 of the pseudo-code of SELECT. As already anticipated, the role of this loop is to discard (by failing) any non-minimal set of services that may satisfy the query.

Let us first formalise the (obvious) notion of minimality.

Def. 4 *Let Q be a query and let S be a set of services that may satisfy Q. S is* minimal $\iff \nexists S' \subset S : S'$ *may satisfy Q.*

To illustrate the nature of non-minimal sets of services, consider the following simple example. Consider a query taking as input E and requiring as outputs A, B, C and D, and three services S_1, S_2 and S_3 (Figure 7). S_1 takes as input E and returns as output A, B and C; S_2 takes as input E and produces as output A and D, while S_3 takes as input E and returns as output B and D. The set of services consisting of S_1, S_2 and S_3 may satisfy the query but it is not minimal because the outputs produced by S_2 are contained in the set of outputs produced by S_1 and S_3 (viz., $\{A, D\} \subset \{A, B, C, D\}$). On the other hand, both $\{S_1, S_3\}$ and $\{S_1, S_2\}$ are minimal sets of services that may satisfy the query.

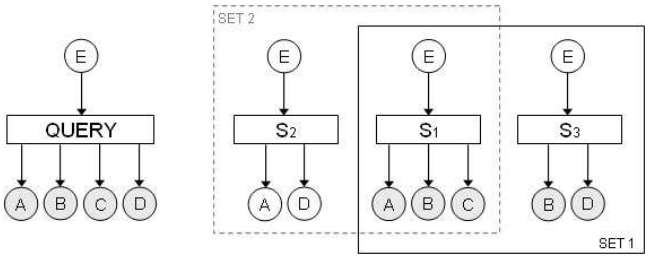


Figure 7: Example of *minimal* sets of services.

Intuitively speaking, the loop at lines 9–13 checks whether the inclusion of the new service s in the set of services *serviceSet* makes some other service in *serviceSet* not strictly necessary to satisfy the query.

SELECT hence checks, for each service t in *serviceSet* (line 9), that the condition at lines 11–13 does not hold. Such condition holds if all the "useful" outputs produced by a service t are already produced by the other services in $serviceSet \cup \{s\} \setminus \{t\}$. An output of t is considered useful if it is a sub-concept of a query output or of an input needed by some other service (and not part of the

query inputs). Therefore, if the condition at lines 11–13 holds, this means that the inclusion of s in the set of services has made service t not strictly necessary to achieve the goal. If this is the case, then SELECT fails (line 13) in order to avoid constructing non-minimal sets of services.

It is worth noting that, while the condition at lines 11–13 is verbose, its verification simply reduces to a few trivial operations over (small sized) sets of data.

Finally, we prove that the condition employed at lines 11–13 is both necessary and sufficient to establish the minimality of a set of services that may satisfy a query.

Property 1 *Let Q be a query and let S be a set of services that may satisfy Q. S is* minimal $\iff \forall t \in S, \exists x \in t.outputs$:

$$(\exists y \in Q.outputs : x \sqsubseteq y \wedge \nexists z \in \bigcup_{r \in S \setminus \{t\}} r.outputs : z \sqsubseteq y) \vee$$

$$(\exists y \in \bigcup_{r \in S \setminus \{t\}} r.inputs : x \sqsubseteq y \wedge \nexists z \in \bigcup_{r \in S \setminus \{t\}} r.outputs \cup Q.inputs : z \sqsubseteq y)$$

Proof ($\Longrightarrow$) Suppose that S is not minimal.
Then $\exists t \in S : S \setminus \{t\}$ may satisfy Q, that is:
(1) $\forall o \in Q.outputs, \exists x \in \bigcup_{r \in S \setminus \{t\}} r.outputs : x \sqsubseteq o$
(2) $\forall i \in \bigcup_{r \in S \setminus \{t\}} r.inputs, \exists y \in \bigcup_{r \in S \setminus \{t\}} r.outputs \cup Q.inputs : y \sqsubseteq i$
Now:
(1) $\Rightarrow \nexists o \in Q.outputs : (\exists x \in t.outputs : x \sqsubseteq o) \wedge (\nexists z \in \bigcup_{r \in S \setminus \{t\}} r.outputs : z \sqsubseteq o)$
(2) $\Rightarrow \nexists i \in \bigcup_{r \in S \setminus \{t\}} r.inputs : (\exists x \in t.outputs : x \sqsubseteq i) \wedge (\nexists w \in \bigcup_{r \in S \setminus \{t\}} r.outputs \cup Q.inputs : w \sqsubseteq i)$
Therefore
$\exists t \in S : \nexists x \in t.outputs$:
$(\exists y \in Q.outputs : x \sqsubseteq y \wedge \nexists z \in \bigcup_{r \in S \setminus \{t\}} r.outputs : z \sqsubseteq y) \vee$
$(\exists y \in \bigcup_{r \in S \setminus \{t\}} r.inputs : x \sqsubseteq y \wedge \nexists z \in \bigcup_{r \in S \setminus \{t\}} r.outputs \cup Q.inputs : z \sqsubseteq y)$
Hence, we obtain a contradiction.

($\Longleftarrow$) Suppose that:
$\exists t \in S : \nexists x \in t.outputs$:
$(\exists y \in Q.outputs : x \sqsubseteq y \wedge \nexists z \in \bigcup_{r \in S \setminus \{t\}} r.outputs : z \sqsubseteq y) \vee$
$(\exists y \in \bigcup_{r \in S \setminus \{t\}} r.inputs : x \sqsubseteq y \wedge \nexists z \in \bigcup_{r \in S \setminus \{t\}} r.outputs \cup Q.inputs : z \sqsubseteq y)$
Now:
$\forall o \in Q.outputs, \exists y \in \bigcup_{r \in S} r.outputs : y \sqsubseteq o$
$\Longrightarrow$ {since $\nexists x \in t.outputs : \exists o \in Q.outputs : x \sqsubseteq o \wedge \nexists y \in \bigcup_{r \in S \setminus t} r.outputs : y \sqsubseteq o$}
$\forall o \in Q.outputs, \exists y \in \bigcup_{r \in S \setminus t} r.outputs : y \sqsubseteq o$
Moreover:
$\forall i \in \bigcup_{r \in S} r.inputs, \exists y \in (\bigcup_{r \in S} r.outputs \cup Q.inputs) : y \sqsubseteq i$
$\Longrightarrow$ {since $\nexists x \in t.outputs : \exists z \in \bigcup_{r \in S \setminus t} r.inputs : x \sqsubseteq z \wedge \nexists w \in \bigcup_{r \in S \setminus t} r.outputs \cup Q.inputs : w \sqsubseteq z$}
$\forall i \in \bigcup_{r \in S \setminus t} r.inputs, \exists y \in (\bigcup_{r \in S \setminus t} r.outputs \cup Q.inputs) : y \sqsubseteq i$

Hence $S \setminus \{t\}$ may satisfy Q, and S is not minimal. Therefore we obtain a contradiction. □

Summing up, Property 1 ensures that the SELECT function determines all the minimal sets of services that may satisfy a query. The functional analyser then organises the sets of services returned by SELECT into an ordered list, which is the output of the whole module. Such list can be ordered according to the client's preferences (specified together with the query), as for instance minimal number of selected services.

As one may expect, this second step of the functional analyser has a high worst-case complexity, as finding all compositions that satisfy the request is a NP problem [5]. Executing function SELECT over a registry of S services will generate $S!$ sequences of recursive invocations in the worst case, and each sequence will perform $O(S^2)$ times the minimality comparisons of lines 11–13 (which may be assumed to take constant time) in the worst case.

It is however worth observing that a more efficient implementation of the matchmaker can be obtained by orchestrating the functional and behavioural analysers in a generate-and-test pipeline. Namely, the behavioural analyser can check each candidate composition as soon as it is determined by the functional analyser, without having to wait for all candidates to be determined and compared. Yet, returning the first successful composition may affect the quality of the overall result, as the functional analyser cannot select the service composition satifying some specific requirement (e.g., the successful composition employing the minimal number of service).

Another factor that significantly influences the efficiency of the functional analyser is (obviously) the number of services considered by the function SELECT. Such number can be sensibly reduced both by well-specifying the query and by introducing a suitable pre-selection phase (for instance using UDDI to filter services not belonging to certain service categories, classified in taxonomies like NAICS or UNSPSC [37]).

Example. We present next an example that illustrates the behaviour of the functional analyser. Let us consider again the registry containing the three services described in Section 2: `Photo_Service`, `Online_Bank` and `Prepaid_Cards`, whose Petri net representations are respectively depicted in Figures 8, 9 and 10 (as before, places for control tokens are coloured in gray).

Consider again the query specifying:

- inputs: `username`, `password`, `c/a_Number`, `info_Bank`, `photo_List`, `paper_Type`, `format`, `number_of_copies`, `delivery_Type`, `address`, and
- output: `order_Confirmation`.

The first step of the functional analyser analyses each service in the registry and generates the *outputRegistry*. One may note that all the presented services have multiple profiles, as their process models contain non-deterministic composite processes (i.e., `choice` processes). Therefore, for example, the `username` concept is generated by two profiles corresponding to `Photo_Service` as well as by four and two profiles corresponding to `Online_Bank` and `Prepaid_Cards`,

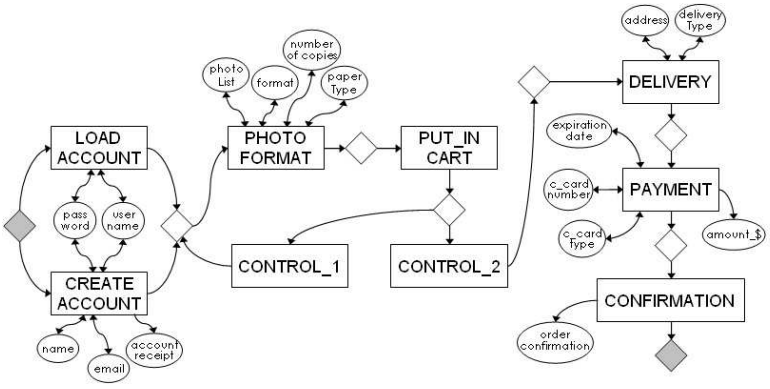


Figure 8: Petri net representation of the `Photo_Service`.

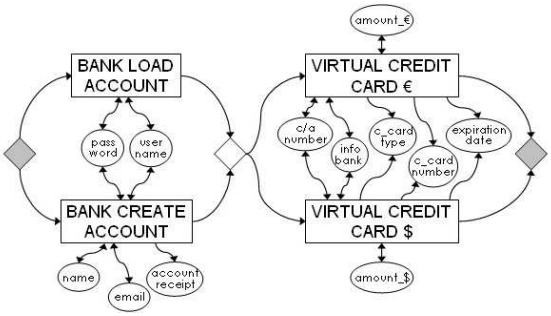


Figure 9: Petri net representation of the `Online_bank` service.

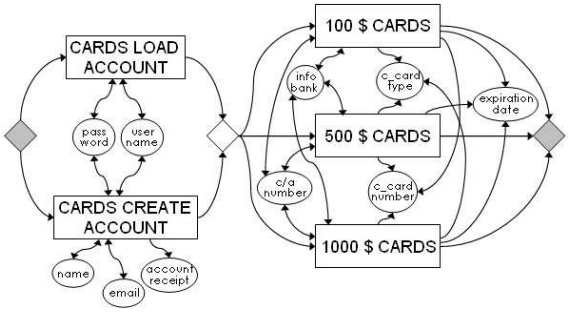


Figure 10: Petri net representation of the `Prepaid_Cards` service.

respectively. Once *outputRegistry* has been constructed, the second step can start. The first invocation of SELECT takes as *o_needed* (i.e., the goal set) the set of query outputs. *o_needed* contains only `order_Confirmation`, which is produced by the two profiles of `Photo_Service`. The functional analyser withdraws `order_Confirmation` from *o_needed* and it creates two sets of services,

everyone containing a Photo_Service profile. Let us consider the *serviceSet* which contains the Photo_Service profile advertising load_account atomic process (the other *serviceSet* fails). Next, c_card_Type, c_card_Number and expiration_Date are added to *o_needed*. As *o_needed* is not empty, SELECT is invoked with the updated parameters. Now, let us suppose that SELECT withdraws from *o_needed* c_card_Number, which is generated by the profiles of both Online_Bank and Prepaid_Cards. In this case, SELECT generates six recursive instances. All such instances fail (as they do not satisfy the query), with the exception of two of them. In the first one, SELECT adds Online_Bank (profile corresponding to bank_load_account and virtual_credit_card_$) to the set of services (up till now, it contains Photo_Service), whereas in the second one, it adds Prepaid_Cards (profile corresponding to cards_load_account and (100_$_cards or 500_$_cards or 1000_$_cards)) to the set of services. In both instances, c_card_Type and expiration_Date are withdrawn from *o_needed*, as they are produced as output by both services Online_Bank and Prepaid_Cards. Moreover, in the first instance, amount_$ is not added to *o_needed*, as it belongs to the set of the outputs produced by the already selected services. When all instances of SELECT terminate, two set of services have been constructed: the first contains (a profile of) Photo_Service and (a profile of) Online_Bank, the second contains (a profile of) Photo_Service and (a profile of) Prepaid_Cards. These two sets of services are taken as input by the behavioural analyser. ◇

4.2 Behavioural analyser

When the functional analyser module terminates, it returns an ordered list of all minimal sets of services which *may* achieve the query satisfaction. Indeed, the previous module builds such sets selecting services with respect to their functional attributes, as it analyses only their OWL-S service profiles. In order to verify whether services belonging to a set can really be composed together, their behaviour must be analysed. This is the task of the behavioural analyser, which consists of two steps. The first step takes as input a set of services and merges together their Petri net representations. The result is a global Petri net which implements the whole composition. From the point of view of process composition, merging services means creating a new composite process which executes the services concurrently. To be more exact, according to the OWL-S vision, the service composition is a split+join process. It is worth observing that, while other types of composition could be possible, parallel composition is the "most general" in the sense that it allows services to freely interleave but for the constraints imposed by data dependencies.

The second step analyses the generated global Petri net and replies to the client with a positive or negative match.

Step 1. The aim of the first step of the behavioural analyser is to merge together the Petri nets contained in a given set. It returns as output the global Petri net whose transitions and places respectively consist of all transitions and all places belonging to the Petri nets in the given set.

One may notice that whereas the transitions are all distinct, as each of them represents an atomic process identified by an unique URI, different (circle-shaped) places can instead represent related concepts, namely concepts linked together by means of the *sub-concept-of* relation (Def. 2). For each couple of places p_1, p_2 representing c, d concepts such that $c \sqsubseteq d$, we insert into the global Petri net an empty transition t and three directed 1-weighted arcs (p_1, t), (t, p_1), (t, p_2). Hence when c is available (p_1 is marked), t can fire and d will be available (p_2 will be marked) as well. Note that the directed arc (t, p_1) is necessary to keep c available also after the firing of t.

Furthermore, the produced global Petri net has two (diamond-shaped) additional places, the new starting control place and the new ending control place, and two additional transitions, called `start_activity` and `end_activity`. The former activity is linked to the new starting control place by an incoming arc and to the starting control places of all services contained in the given set by outgoing arcs. The latter activity is linked to the ending control places of all services in the initial set by incoming arcs and to the new ending control place by an outgoing arc. The new additional transitions are needed for implementing the two synchronization points, respectively the initial one and the final one, as requested in a `split+join` process.

Step 2. The second step of the behavioural analyser inputs a client query and a Petri net representing a (composition of) service(s), and it checks whether the given composition is capable of fulfilling the query. A composition satisfies the query if it generates all the outputs requested by the query and if all of its services terminate correctly. From the point of view of Petri nets, this means that a Petri net satisfies a query if it *reaches* a state (i.e., a marking) where the ending control place as well as those places corresponding to the query outputs are marked with at least one token. If this condition occurs, the module returns a positive match to the client.

This second step starts the Petri net analysis by determining the *initial* state of the net, that is, the marking in which all the net places are unmarked with the exception of the starting control place as well as of the places corresponding to the query inputs, that hold one token each. Next, this step performs a *state space* analysis in order to check whether a target state is reachable starting from the initial marking.

A target state is the one capable of satisfying the query. Yet, since the non-deterministic behaviour of a Petri net, several target states can exist. Target states satisfy the following properties:

- diamond-shaped places are unmarked, with the exception of the ending control place which holds one token;
- circle-shaped places corresponding to the query inputs and to the query outputs are marked with (at least) one token;
- circle-shaped places which do not belong to the query inputs and which have no incoming arcs are unmarked.

Due to the non-determinism of Petri nets, it is not possible to foresee the marking of the non-mentioned places without animating the net.

A marking M_n is *reachable* from a marking M_0 if there exists a sequence of firings, denoted by $\sigma = t_1, t_2, ..., t_n$, that transforms M_0 in M_n. In this case M_n is reachable from M_0 by σ and we write $M_0[\sigma > M_n$. The set of all possible markings reachable from M_0 is denoted by $R(M_0)$.

Hence, given a query and the Petri net representation N of a set of services, the behavioural analyser determines whether there exists a target state M_t of N which is reachable from the initial state M_0 of N, that is, $M_t \in R(M_0)$.

The *reachability* (or *coverability*) *graph* and the *matrix-equation* approaches are two of the main methods for analysing the behavioural (or marking-dependent) properties of a Petri net, namely for determining whether there exists a $M_t \in R(M_0)$. The first method generates a labeled directed graph whose nodes represent all the markings reachable from the initial marking and whose arcs, labeled with a single transition t_k, represent all possible firings such that $M_i[t_k < M_j$, where M_i and M_j belong to the graph nodes. Still, this method can be applied only to limited-size nets due to the complexity of the space-state explosion.

To avoid this problem, the second method describes the dynamic behaviour of a Petri net by means of a linear algebraic approach. Consider a transition t, let t^- and t^+ be $P \times 1$ vectors defined as follows:

$$t^-(p) = \begin{cases} w(p,t) & p \in {}^\bullet t \\ 0 & \text{otherwise} \end{cases} \qquad t^+(p) = \begin{cases} w(t,p) & p \in t^\bullet \\ 0 & \text{otherwise} \end{cases}$$

where $w(p,t)$ and $w(t,p)$ denotes the weight of arcs (p,t) and (t,p), respectively. The *incident matrix* C of a Petri net is a $P \times T$ matrix of integers, where each column t is equal to the vector $(t^+ - t^-)$. Now, let σ a firing sequence. The *count vector* $\Psi(\sigma)$ is a $T \times 1$ vector that assigns to each t its number of occurrence in σ.

Given a marking M_n reachable from a marking M_0 by σ, the resulting *state equation* is:

$$M_n = M_0 + C \cdot \Psi(\sigma)$$

The state equation can be used in a non-reachability test. Indeed, if M_n is reachable from M_0 then the following system has an integer solution:

$$C \cdot x = M_n - M_0,\ x \geq 0$$

Yet, as there can be solutions of the state equation that do not correspond to any executable firing sequence, it is necessary to check whether a solution is a firing sequence by animating the net.

In the literature, there exist various proposals which aim at performing an efficient state space analysis, because of the key role played by the reachability problem. For instance, Schmidt presented in [28] a search strategy that firstly explores sequences corresponding to a minimal solution of the state equation. More precisely, given a (possibly partially defined) target state, when the state

equation is not able to qualify it as unreachable, [28] exploits the information concerning the computed state equation in order to narrow the state space search by exploring first the promising firing sequences.

Currently, there also exist several Petri net tools which provide a graphical modelling interface, support interactive simulation and perform reachability analysis. For example, Gaševic et al. proposed in [29] a tool capable of analysing the behavioural properties of a given Petri net by means of the reachability tree, matrix equations and matrix invariants. A Petri net tool generating coverability graphs and checking for invariance and consistence properties was also described in [9]. Varpaaniemi et al. presented in [35] a Petri net tool which performs the reachability analysis by relying on the stubborn set method [34] in order to avoid the state space explosion.

Any of these approaches can be plugged in our proposal in order to complete an efficient behavioural Petri net analysis. At the end of the marking-dependent analysis, if there exists at least a reachable target state, the behavioural analyser can reply to the client with a positive match.

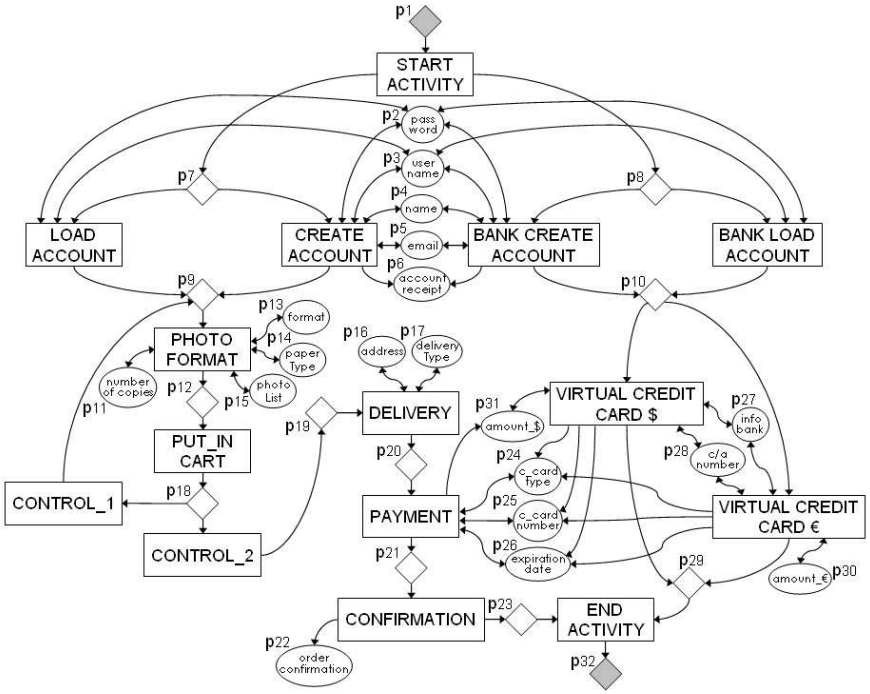


Figure 11: The global Petri net for the first composition.

Example. Consider again the example described in the previous subsection. The functional analyser module returns two compositions containing the Petri net representation of the services needed to fulfill the client query. The first composition contains `Photo_Service` and `Online_Bank` and the behavioural

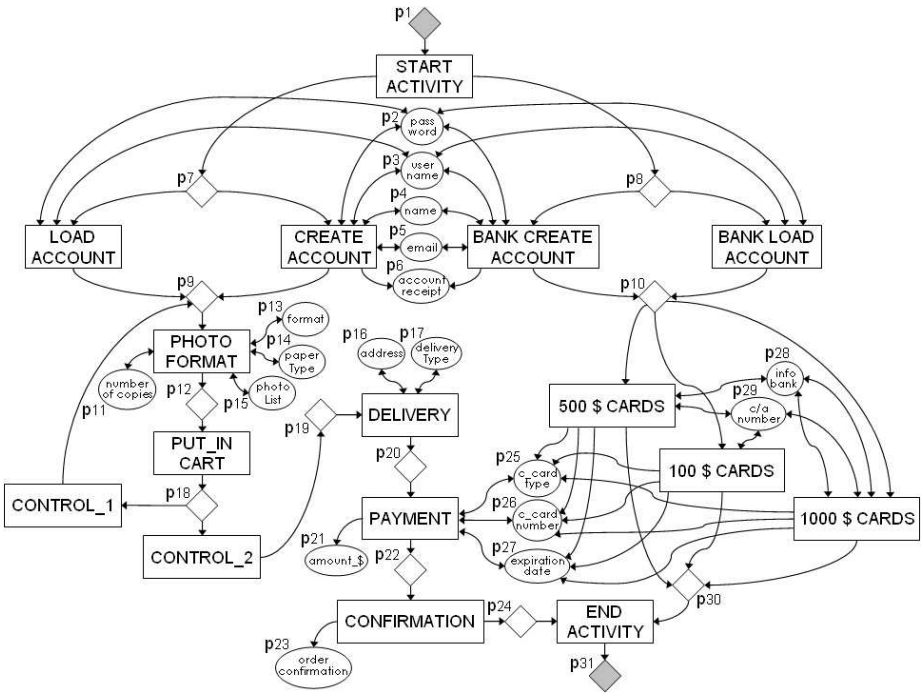


Figure 12: The global Petri net for the second composition.

analyser, during its first step, merges together these two services generating as output the global Petri net illustrated in Figure 11. The global Petri net contains all the transitions as well as all the places of `Photo_Service` and `Online_Bank` with the addition of the new starting and ending transitions (i.e., `start_activity` and `end_activity`) as well as of the new starting and ending control places. One may note that the `username` concepts of the two services have been merged in an unique place. Indeed these concepts are both syntactically and semantically equivalent. For the same reason, the `password` concept of `Photo_Service` has been unified with the `password` concept of `Online_Bank` and so on for the following concepts: `name`, `email`, `account_Receipt`, `amount_$`, `c_card_Type`, `c_card_Number` and `expiration_Date`. The second composition returned by the functional analyser contains `Photo_Service` and `Prepaid_Cards`. Similarly, the behavioural analyser merges together these two services generating the global Petri net shown in Figure 12.

Next, the behavioural analyser continues with the second step. We first analyse the global Petri net produced for the first composition, depicted in Figure 11. In order to verify if the composition consisting of `Photo_Service` and `Online_bank` services is capable to fulfill the example query, the second step checks whether the target state M_t is reachable from the initial state M_0. In the initial state the places corresponding to the query inputs (i.e., `photo_List`,

paper_Type, format, number_of_copies, delivery_Type, address, c/a_Number, info_Bank, username and password) as well as the starting control place are marked with one token. M_0 is described by the following $P \times 1$ vector:

$$M_0 = [1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0]^T$$

In the target state the ending control place as well as the places corresponding to the query inputs and outputs hold one token, whereas the other diamond-shaped places and the places having no incoming arcs and not belonging to the query inputs are unmarked. The following $P \times 1$ vector describes the target state M_t, where "*" denotes those places whose marking cannot be foreseen.

$$M_t = [0\ 1\ 1\ 0\ 0\ *\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ *\ *\ *\ 1\ 1\ 0\ 0\ *\ 1]^T$$

Next, the behavioural analyser starts the state space analysis in order to check whether $M_t \in R(M_0)$. In this example the target state cannot be reached and, therefore, the set of services {Photo_Service, Online_Bank} is not capable of satisfying the query. For example, consider the firing sequence {start_activity, bank_load_account, load_account, photo_format, put_in_cart, control_1, control_2, delivery}. At this point, the net reaches a dead state, where no more transitions can fire. Indeed, payment is waiting for c_card_Number, c_card_Type and expiration_Date as well as virtual_credit_card_$ and virtual_credit_card_€ are waiting for amount_$ and amount_€, respectively.

Consider now the second composition consisting of Photo_Service and Prepaid_Cards, whose Petri net representation is illustrated in Figure 12. The following two $P \times 1$ vectors describe the initial state M_0 and the target state M_t respectively.

$$M_0 = [1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0]^T$$

$$M_t = [0\ 1\ 1\ 0\ 0\ *\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ *\ 0\ 1\ 0\ *\ *\ *\ 1\ 1\ 0\ 1]^T$$

In this second example, the state space analysis produces a positive result, indeed the set of services {Photo_Service, Prepaid_Cards} is capable of satisfying the query. For instance, consider the following firing sequence σ defined as follows: {start_activity, cards_load_account, load_account, photo_format, put_in_cart, control_1, control_2, delivery, 100_$_cards, payment, confirmation, end_activity}. The condition $M_0[\sigma > M_t$ holds. Therefore, the services belonging to this second composition can really be composed together and satisfy the client query.

The behavioural analyser replies the client with a positive match.

◇

5 Related work

During the last years, various efforts have been devoted to developing service discovery algorithms capable of overcoming the limitations of the available UDDI

search mechanisms. We briefly mention above some of such proposals, focussing on those which take into account semantics information, and address behavioural and/or composition issues of service discovery. As already anticipated in the Introduction, our proposal is – at the best of our knowledge – the first one to address all the above three issues.

The first semantics-based algorithm for Web service discovery was proposed by some of the authors of OWL-S in [23]. The algorithm of Paolucci et al. performs a functionality matching between service requests and service advertisements described as DAML-S (the predecessor of OWL-S) service profiles. A request matches a service advertisement, if all the outputs of the request are matched by the outputs of the advertisement as well as all the inputs of the advertisement are matched by the inputs of the request. The algorithm in [23] however does not deal with service behaviour, nor it considers service composition.

Li and Horrocks described in [14] a service discovery algorithm based on a description logic reasoner, which speeds up the matching process by employing an off-line classification of DAML-S service advertisements. As [23], [14] does not address behavioural and composition issues.

Aversano et al. and Benatallah et al. respectively proposed in [3] and [5] two approaches which extend [23] with the discovery of service compositions. [3] searches for (compositions of) services able to fulfill the client request by featuring a cross-ontology matching (over service descriptions employing different ontologies). [5] addresses the discovery of service compositions that match a given request by reducing such issue to a best covering problem in the domain of hypergraph theory. Since both [3] and [5] focus on DAML-S service profiles only, they do not deal with service behaviour.

Bansal and Vidal [4] were the first to propose a service discovery algorithm that takes into account service behaviour. Their algorithm analyses DAML-S process models (rather than service profiles as [23, 3, 5]). However [4] only addresses the problem of single service discovery and it does not consider the issue of service composition.

Sahin et al. presented in [27] a P2P-based Web service discovery framework supporting keyword-based, ontological-based and behaviour-based search operations. However, [27] features neither combinations of the discovery mechanisms it provides nor service composition.

Mokhtar et al. have recently proposed in [16] an algorithm for Web service discovery and composition based on both OWL-S service profile and process model. They model both services and the client request as finite state automata and their goal is to reconstruct the client query automaton by using fragments of the available services. A similar work has been presented by Hashemian and Mavaddat in [12]. They propose a graph-based approach for composing Web services based on the OWL-S process model, and they formally model both services and the client query as interface automata [11]. Both [16] and [12] address the composition of OWL-S services by focussing on analysing input/output dependencies among services. On the other hand, they do not consider the ordering of atomic processes (within services) which is crucial in order to determine the

behaviour of a service composition, e.g., to determine whether it may deadlock or not.

Traverso and Pistore described in [32] a composition-oriented service discovery approach which takes into account service behaviour and complex goals. The target of [32] is to find a plan that satisfies a given goal over a planning domain rendered as the state transition system that combines all the transition systems corresponding to (the OWL-S process models of) the available services. A tool which implements [32] is presented in [31]. [32] focuses on behavioural and composition issues, however, it seems to disregard semantics information, as the quality of the discovered composition mainly depends on the goodness of the given goal (expressed by means of the EaGLe language), rather than on the functional attributes of services.

Berardi et al. addressed in [7] the composition issue by developing a FSM-based framework where services are specified by means of finite state machines, the target service (i.e., the query) included. They proposed an exponential-timed algorithm that checks for the existence of a service composition by reducing such problem into the satisfiability of a suitable DPDL (Deterministic Propositional Dynamic Logic) formula. In [8] they extended the approach of [7] by modelling services as nondeterministic finite state machines in order to deal with services not fully controllable by the orchestrator. [7] and [8] address behavioural and composition issues, however they do not focus on semantic aspects.

The METEOR-S Team [15] is working to the realisation of a framework [30, 26] for annotating, discovering and composing semantic Web services. Yet, METEOR-S is a semi-automated approach requiring a strong participation of the user, which is highly involved in the process of semi-manually discovering and/or composing services.

Finally, it is worth observing that the adoption of Petri nets (and their extensions) to model Web services (compositions) has been advocated by many authors. Hamadi and Benatallah defined in [6] a Petri net-based algebra for modelling Web service control flows. They use ordinary Petri nets to represent services and their compositions. Yet, by modelling a service by means of a single transition and two places, one for absorbing information and the other for emitting information, they do not consider the inner behaviour of services.

Alvares et al. addressed in [1] Web service composition and coordination by modelling complex conversations and workflows by means of the Nets-within-Nets paradigm, which is an extension of the coloured Petri nets. A weak aspect of [1] is the data flow representation that could not suffice to allow services to communicate, since some data flow information could be lost in the mapping of services into the Nets-within-Nets paradigm.

Yi and Kochut proposed in [13] a unified coloured Petri nets-based specification model suitable for both service composition and conversation protocol. Their model enabled both automated detection of problem hidden in compositions (e.g., deadlocks) and formal verification of service (compositions) properties (e.g., reachability of any expected state). However, [13] as well as [6, 1] intentionally focus on behavioural and composition issues, without taking into

account semantics information.

Still, several Petri net-based tools performing static verification of some service composition properties have been recently proposed, as [20] which supports the analysis of BPEL processes.

6 Concluding remarks

We have presented a new matchmaking system based on OWL-S ontologies and Petri nets for discovering lock-free compositions of Web services. Our system consists of three independent modules:

- a *translator* – which models OWL-S services as Petri nets,
- a *functional analyser* – which filters services taking into account their functional attributes,
- a *behavioural analyser* – which, after merging together a set of (selected) Petri nets, checks for a positive match by analysing the Petri net of the service composition found.

As already mentioned in the Introduction, the main features of our system are to discover *minimal* compositions of services, that is, compositions containing the number of services strictly necessary to satisfy a request, and to feature a matching strongly based on behaviour of services.

In this paper, we have assumed that services are provided with an OWL-S description. On the other hand, one may argue that only few OWL-S service descriptions are currently available and that, more generally, a "de facto" new standard for service description has not emerged yet. Roughly speaking, while academic research seems to focus more towards ontology-based descriptions like OWL-S, industry seems to focus more on WSDL and BPEL.

This very situation has motivated the modular design of the architecture of our matchmaking system. Indeed both the functional and the behavioural analysis module can be configured so as to deal with different service description languages. For instance, BPEL services can be dealt with by translating their behaviour in Petri nets (as shown in [21]) and by performing only syntactic matching during their functional analysis. The deployment of this multi-language capability of the matchmaker is the next step we intend to make. Our first goal here is to feature a full integration with BPEL, supporting both the inclusion of BPEL services in registries and the deployment in BPEL of discovered compositions.

A second line for our future work is to improve the semantic matching performed by the functional analyser so as to feature full-fledged cross-ontology matchings over service descriptions employing different ontologies, by plugging-in existing "ontology-crossers", such as [19].

Our long-term goal is to develop a well-founded methodology to support the discovery, composition, and – when necessary – adaptation of Web services.

References

[1] P. Álvares, J. Bañares, and J. Ezpelata. Approaching Web Service Coordination and Composition by Means of Petri Nets. The Case of the Nets-within-Nets Paradigm. In B. Benatallah, F. Casati, and P. Traverso, editors, *Service-Oriented Computing – ICSOC 2005, LNCS* 3826, pages 185–197. Springer-Verlag, 2005.

[2] T. Andrews and et al. Business Process Execution Language for Web Services (version 1.1). May 2003. http://www-106.ibm.com/developerworks/library/ws-bpel.

[3] L. Aversano, G. Canfora, and A. Ciampi. An algorithm for web service discovery through their composition. In L. Zhang, editor, *IEEE International Conference on Web Services (ICWS'04)*, pages 332–341. IEEE Computer Society, 2004.

[4] S. Bansal and J. Vidal. Matchmaking of Web Services Based on the DAML-S Service Model. In T. Sandholm and M. Yokoo, editors, *Second International Joint Conference on Autonomous Agents (AAMAS'03)*, pages 926–927. ACM Press, 2003.

[5] B. Benatallah, M.-S. Hacid, C. Rey, and F. Toumani. Request Rewriting-Based Web Service Discovery. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *The Semantic Web - ISWC 2003, LNCS* 2870, pages 242–257. Springer-Verlag, 2003.

[6] B. Benatallah and R. Hamadi. A Petri Net-based Model for Web Service Composition. In *Proceedings of the 14th Australasian Database Conference (ADC 2003)*, pages 191–200, 2003.

[7] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of *e*-Services that Export their Behavior. In M. E. Orlowska, S. Weerawarana, M. P. Papazoglou, and J. Yang, editors, *Service-Oriented Computing – ICSOC 2003, LNCS* 2910, pages 43–58. Springer-Verlag, 2003.

[8] D. Berardi, D. Calvanese, G. D. Giacomo, and M. Mecella. Composition of Services with Nondeterministic Observable Behaviour. In B. Benatallah, F. Casati, and P. Traverso, editors, *Service-Oriented Computing – ICSOC 2005, LNCS* 3826, pages 520–526. Springer-Verlag, 2005.

[9] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42(14), 2004.

[10] A. Brogi, S. Corfini, and R. Popescu. Composition-oriented Service Discovery. In T. Gschwind, U. Aßmann, and O. Nierstrasz, editors, *Software Composition, LNCS* 3628, pages 15–30. Springer-Verlag, 2005.

[11] L. de Alfaro and T. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–102. ACM Press, 2001.

[12] S. Hashemian and F. Mavaddat. A Graph-Based Approach to Web Services Composition. In I. C. Society, editor, *The 2005 Symposium on Applications and the Internet (SAINT'05)*, pages 183–189. CS Press, 2005.

[13] K. J. Kochut and X. Yi. CPNet Model for BPEL4WS Workflow. In *University of Georgia, Computer Science department - technical report*, November 2004.

[14] L. Li and I. Horrocks. A Software Framework for Matchmaking Based on Semantic Web Technology. *International Journal of Electronic Commerce*, 8(4):39–60, 2004.

[15] METEOR-S Team. METEOR-S: Semantic Web Services and Processes, 2004. `http://lsdis.cs.uga.edu/projects/meteor-s/`.

[16] S. B. Mokhtar, N. Georgantas, and V. Issarny. Ad Hoc Composition of User Tasks in Pervasive Computing Environment. In T. Gschwind, U. Aßmann, and O. Nierstrasz, editors, *Software Composition, LNCS* 3628. Springer-Verlag, 2005.

[17] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[18] I. Navas-Delgado, M. del Mar Rojano-Munoz, and J. F. Aldana-Montes. An Architecture to Semantically Compose Web Services. In *DOA 2005 (Poster)*, 2005.

[19] I. Navas-Delgado, I. Sanz, J. F. Aldana-Montes, and R. Berlanga. Automatic Generation of Semantic Fields for Resource Discovery in the Semantic Web. In *16th International Conference on Database and Expert Systems Applications (DEXA 2005). LNCS* 3588, 2005.

[20] C. Ouyang, E. Verbeek, W. van der Aalst, S. Breutel, M. Dumas, and A. ter Hofstede. WofBPEL: A Tool for Automated Analysis of BPEL Processes. In B. Benatallah, F. Casati, and P. Traverso, editors, *Service-Oriented Computing – ICSOC 2005, LNCS* 3826, pages 484–489. Springer-Verlag, 2005.

[21] C. Ouyang, E. Verbeek, W. M. van der Aalst, S. Breutel, M. Dumas, and A. H. ter Hofstede. Formal Semantics and Analysis of Control Flow in WS-BPEL. Technical Report BPM-05-13, 2005.

[22] OWL-S Coalition. OWL-S 1.1 release, 2004. `http://www.daml.org/services/owl-s/1.1/`.

[23] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic Matchmaking of Web Services Capabilities. In I. Horrocks and J. Hendler, editors, *First International Semantic Web Conference on The Semantic Web, LNCS* 2342, pages 333–347. Springer-Verlag, 2002.

[24] M. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *WISW2003*, pages 3–12, December 2003.

[25] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[26] P. Rajasekaran, J. A. Miller, K. Verma, and A. P. Sheth. Enhancing Web Services Description and Discovery to Facilitate Composition. In J. Cardoso and A. Sheth, editors, *Semantic Web Services and Web Process Composition, LNCS* 3387, pages 55–68. Springer-Verlag, 2005.

[27] O. Sahin, C. Gerede, D. Agrawal, A. Abbadi, O. Ibarra, and J. Su. SPiDeR: P2P-Based Web Service Discovery. In B. Benatallah, F. Casati, and P. Traverso, editors, *Service-Oriented Computing – ICSOC 2005, LNCS* 3826, pages 157–169. Springer-Verlag, 2005.

[28] K. Schmidt. Narrowing Petri Net State Spaces Using the State Equation. *Fundamenta Informaticae*, 47(3-4):325–335, IOS Press, 2001.

[29] D. G. sevic, V. D. zic, and N. Veselinovic. P3 - Petri Net Educational Software Tool for Hardware Teaching. In *Proceedings of The 10th Workshop on Algorithms and Tools for Petri Nets*, pages 111–120, 2003.

[30] K. Sivashanmugam, J. A. Miller, A. P. Sheth, and K. Verma. Framework for Semantic Web Process Composition. *International Journal of Electronic Commerce (IJEC), Special Issue on Semantic Web Services and Their Role in Enterprise Application Integration and E-Commerce*, 9(2):71–106, Winter 2004-05.

[31] M. Trainotti, M. Pistore, G. Calabrese, G. Zacco, G. Lucchese, F. Barbon, P. Bertoli, and P. Traverso. ASTRO: Supporting Composition and Execution of Web Services. In B. Benatallah, F. Casati, and P. Traverso, editors, *ICSOC 2005, LNCS 3826*, pages 495–501. Springer-Verlag, 2005.

[32] P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *International Semantic Web Conference (ISWC)*, pages 380–394. IEEE Computer Society, 2004.

[33] UDDI. The UDDI Technical White Paper. 2000. http://www.uddi.org/.

[34] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.

[35] K. Varpaaniemi, K. Heljanko, and J. Liliu. PROD 3.2 an advanced tool for efficient reachability analysis. In O. Grumberg, editor, *Computer Aided Verification: 9th International Conference, CAV'97, LNCS* 1254, page 472475. Springer-Verlag, 1997.

[36] W3C. Simple Object Access Protocol (SOAP) 1.2, W3C working draft, 17 December 2001. 2001. http://www.w3.org/TR/2001/WD-soap12-part0-20011217/.

[37] W3C. UDDI Core tModels - Taxonomy and Identifier Systems. 2001. http://www.uddi.org/taxonomies/Core_Taxonomy_OverviewDoc.htm.

[38] W3C. Web Service Description Language (WSDL) 1.1. 2001. World Wide Web Consortium, http://www.w3.org/TR/wsdl.