Semantics-based Composition-oriented Discovery of Web Services

ANTONIO BROGI, SARA CORFINI, RAZVAN POPESCU
Department of Computer Science, University of Pisa, Italy

Service discovery and service aggregation are two crucial issues in the emerging area of Service-Oriented Computing (SOC). We propose a new technique for the discovery of (Web) services that accounts for the need of composing several services to satisfy a client query. The proposed algorithm makes use of OWL-S ontologies, and explicitly returns the sequence of atomic process invocations that the client must perform in order to achieve the desired result. When no full match is possible, the algorithm features a flexible matching by returning partial matches and by suggesting additional inputs that would produce a full match.

Categories and Subject Descriptors: H.3.5 [**Information Storage and Retrieval**]: Online Information Services – *Web-based services*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods

General Terms: Algorithms

Additional Key Words and Phrases: Web service discovery, Web service composition, matchmaking algorithms, OWL-S ontologies

1. INTRODUCTION

Service-Oriented Computing (SOC) [Papazoglou and Georgakopoulos 2003] is emerging as a new, promising computing paradigm that centres on the notion of *service* as the fundamental element for developing software applications. According to [Papazoglou and Georgakopoulos 2003], services are self-describing components that should support a rapid and low-cost composition of distributed applications. Services are offered by service providers, which procure service implementations and maintenance, and supply service descriptions. Service descriptions are used to advertise service capabilities, behaviour, and quality, and should provide the basis for the discovery, binding, and composition of services. Services possess the ability of engaging other services in order to complete complex transactions, like checking credit, ordering products, or procurement. The platform-neutral nature of services creates the opportunity for building composite services by composing existing elementary or complex services, possibly offered by different service providers [Yang 2003].

The Web service model includes three component roles — clients, providers and registries — where providers advertise their services to registries, and clients query registries to discover services. In this scenario, two prominent issues involved in the development of next generation distributed software applications can be roughly synthesised as:

(1) discovering available services that can be exploited to build a needed application, and

(2) suitably aggregating such services in order to achieve the desired result.

The current Web services infrastructure relies on WSDL (Web Services Description Language) [W3C 2001b], SOAP (Simple Object Access Protocol) [W3C 2001a] and UDDI (Universal Description & Discovery Interface) [UDDI 2000]. 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.

The standard service infrastructure has two main limitations: it does not support (automatic) service composition, and it does not account for semantics information. Indeed, service descriptions are expressed by means of WSDL, by declaring a set of message formats and their direction (incoming/outgoing). The resulting description is purely syntactic, very much in the style of Interface Description Languages (IDLs) in component-based software engineering. Consequently, UDDI, which supports the definition of service registries in the style of yellow pages, is able to feature only keyword-based matches that often give poor performance.

Given the pivotal importance of service discovery for SOC, several attempts to enhance the discovery process are currently being pursued. One of the major efforts in this direction is promoted by the OWL-S coalition which aims at enriching service descriptions with semantics and behavioural information. The OWL-S coalition proposes a semantics-based description of Web services, based on the use of OWL-S (formerly DAML-S) ontologies [OWL-S Coalition 2004], where each service is provided with a description consisting of three documents: *service profile* ("what the service does"), *service model* ("how the service works"), and *service grounding* ("how to access the service").

The process of Web service discovery — often referred to as service matchmaking — then takes as input a query as well as a service registry consisting of (service) advertisements, and returns as output a list of matched services. A matchmaking query typically specifies the functionality of the desired service (composition) in terms of its inputs and outputs[1].

In this paper we present a new algorithm for the composition-oriented discovery of Web services. The algorithm — called SAM (for Service Aggregation Matchmaking) — can be used to match queries with service registries making use of OWL-S ontologies. SAM extends a matchmaking algorithm proposed in [Bansal and Vidal

[1]Although OWL-S permits to specify also preconditions and effects of service executions, we decided not to consider their possible employment within client queries as their effectiveness in the matchmaking process is not quite clear.

2003] by featuring a more flexible matching and, more importantly, by accounting for service compositions. Indeed, queries that cannot be satisfied by a single service might be frequently satisfied by composing several services together. An immediate example of this is a client wishing to plan its holidays by booking flight tickets as well as hotel accommodation while taking into account various parameters such as weather, season prices, special offers, and so on.
The main features of the proposed algorithm can be summarised as follows:

– *Flexible matching.* The proposed algorithm (SAM) features a more flexible matching w.r.t. [Bansal and Vidal 2003] as:
  • SAM performs a fine-grained matching at the level of atomic processes, or sub-services (rather than at the level of entire services as in [Bansal and Vidal 2003]).
  • Rather than returning only full matches (when a single service can fully satisfy the client request by itself), SAM also returns (when no full match is possible) a list of *partial* matches. A partial match is a (composition of) sub-service(s) that can provide only some of the outputs requested by the client. It is important to stress that a partial match can be a valuable answer for the client, which may have over-specified its query or may decide to use the selected services even if its query will be only partially satisfied.
  • When no full match is possible, SAM — besides returning partial matches — is also capable to suggest to the client additional inputs that would suffice to get a full match.
– *Composition-oriented matching.* More importantly, SAM is the first algorithm (at the best of our knowledge) to provide a composition-oriented matchmaking based on semantic descriptions of queries and services by taking into account the service behaviour.
  • When no single service can satisfy the client query, SAM checks whether there are service compositions that can satisfy the query, possibly including multiple executions of services.
  • When SAM finds a match, it explicitly returns the sequence of atomic process invocations that the client must perform in order to achieve the desired result.

The rest of the paper is organised as follows. Section 2 is devoted to introduce OWL-S ontologies for service discovery, while in Section 3 we briefly discuss the limitation of existing matchmaking algorithms based on OWL-S. In Section 4 we describe the new algorithm for the composition-oriented discovery of services, and we analyse its complexity, correctness and completeness in Section 5. Related work is presented in Section 6. Some concluding remarks are drawn in Section 7.

2. BACKGROUND: A SHORT INTRODUCTION TO OWL-S

UDDI is the only universally accepted standard for the discovery of Web services. UDDI allows for the creation of online service registries in the style of yellow pages and employs a keyword-based matching system that often leads to poor performance (because of the current availability of interface-level descriptions, i.e. WSDL). According to the Semantic Web vision [Berners-Lee et al. 2001], the introduction of semantic information in Web service descriptions is therefore strongly required. A step in this direction has been made by the OWL-S coalition, that defined a high

level ontology-based language for describing services, named OWL-based Web Service ontology (OWL-S). Since OWL-S is a computer-interpretable semantic markup language providing all the needed information for describing services, OWL-S allows for discovering, invoking and composing services in an automatic way. According to [OWL-S Coalition 2004], a Web service is defined by an OWL-S *description* which is structured in three parts providing three different views of a service: the *service profile* (which describes what the service does), the *service model* (which describes how the service works) and the *service grounding* (which describes how to interact with the service).

The *service profile* provides a high-level description of a service and it consists of three types of information regarding the organisation that provides the service, the function computed by the service as well as some other characteristics of the service. The provider information consists of contact information and refers to the entity that provides the service. Service functionalities are represented by listing the inputs required as well as the outputs produced by the service. The last type of information is a list of service parameters containing information such as an estimate of the maximum response time, geographic availability, cost or quality of the service.

The *service model* has a process model subclass which provides a different perspective of a Web service, that can be viewed as a process. It is important to note, that in this context a process is not a program to be executed, but it is a specification of the ways a client may interact with a service. OWL-S defines three types of processes: atomic, simple and composite. An atomic process is executed in a single step (from the point of view of the client of the service). It can not be decomposed further and it has an associated grounding. Atomic processes have associated inputs and outputs (IOs) and they are the only processes that can be directly invoked by the client. A simple process is similar to an atomic one but it can not be invoked directly and it does not have an associated grounding. It is a simplified and abstract view of a composite process. Finally, a composite process consists of other processes, the composition being made with the following control constructs: `sequence`, `choice`, `split`, `split+join`, `if-then-else`, `any-order`, `iterate` and `repeat-while/until`. The meaning of such control constructs is the usual meaning that they have in the conventional programming languages. More precisely [OWL-S Coalition 2004]:

- a `sequence` process is a list of processes to be executed in order;
- a `choice` process is a bag of processes out of which only one can be chosen for execution;
- 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 synchronisation;
- 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 `any-order` process is a bag of processes to be executed in some unspecified order but not concurrently;

- an `iterate` process is a process 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.

The inputs and the outputs of a composite process correspond to the set of all inputs and respectively to the set of all outputs of its sub-processes.

While both the service profile and the service model are abstract representations of a service, the *service grounding* is the only concrete specialisation of the service. It provides detailed information about how to access and how to interact with the service by specifying protocol as well as message format information. The main purpose of the OWL-S grounding is to show the way to concretely organise the inputs and the outputs of an atomic process (that is, the elements needed for interacting with the service) as a message. To this end, OWL-S exploits the extensive work made in the area of concrete message specification by relying on WSDL and SOAP, that is a standard protocol for information exchange in decentralised and distributed environments.

In the rest of the paper we concentrate on the OWL-S process model because it provides essential information for discovering and composing services, which is the goal of our algorithm. Indeed, even if the information contained in the service profile would be sufficient for (single) Web service discovery, it would not suffice for composing services. More precisely, the profile-based service discovery/composition does not suffice in the case of complex services, while in the case of atomic services the information provided by the profile coincides with the information advertised by the process model. The service profile consists of a list of inputs and outputs and it does not provide any information about the order in which the inputs are requested, or the outputs are returned. Thus, we argue that it is not possible to compose services in a semi-automatic engineered way using their profile contents only.

We present next the process model of an `Electronics_Store` service that sells electronic items like notebooks or digital cameras. The service firstly asks the client for the preferred country and returns in response the service availability in such country. Next, it continues with the authentication of the client, that can choose between logging in to an existing account or creating a new account. Finally, the service starts the selling phase, during which the client can buy a notebook or a digital camera.

The tree structure of the `Electronics_Store` service is illustrated in Figure 2, while (a fragment of) its OWL-S code is presented in the Appendix. The root composite process is `Electronics_Store` which is a `sequence` process composed of an atomic process, `country_choice`, as well as two `choice` processes, `login` and `product_choice`. The `login` process is composed of two atomic processes, `create_account` and `load_account`, while the `product_choice` process is composed of two `sequence` processes, `digital_camera_buy_sequence` and `notebook_buy_sequence`. Both `digital_camera_buy_sequence` and `notebook_buy_sequence` processes are composed of two atomic processes, that are `digital_camera_buy` and `digi-`

`tal_camera_payment`, as well as `notebook_buy` and `notebook_payment` respectively.

3. WEB SERVICE DISCOVERY USING OWL-S ONTOLOGIES

Paolucci et al. propose in [Paolucci et al. 2002] the first matchmaking approach based on the DAML-S service profile. Their algorithm takes as input a client query, a repository of DAML-S Web services, as well as the shared type ontology, and searches for services able to satisfy the query. The client specifies the query as a list of provided inputs and requested outputs. An advertisement *matches* the request [Paolucci et al. 2002] if all the outputs of the request match outputs of the advertisement, and dually, all the inputs of the advertisement match inputs of the request. The matched services have associated a degree of match: `exact` (when the inputs/outputs of the advertisement are equivalent to the inputs/outputs of the request), `plug-in` (when the inputs/outputs of the advertisement include the inputs/outputs of the request), or `subsumes` (when the inputs/outputs of the request include the inputs/outputs of the advertisement). For example, considering the ontology fragment shown in Figure 1, we have a `plug-in` match if the client asks for a *piano* and the provider replies with *musical instruments* as well as we have a `subsumes` match if the client asks for *musical instruments* and the provider replies with a *piano*. [Paolucci et al. 2002] sorts matched services by selecting first the match with the highest score in the outputs. Input matching is used only as a secondary score to tie breaks between equally scoring outputs.

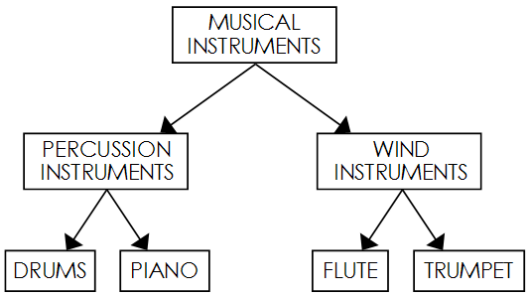

Fig. 1. A fragment of a musical instruments ontology.

Unfortunately, the matching based on the service profile is similar somehow to matching two black boxes, and it allows to match a service request asking for two outputs o_1 *and* o_2 with a service advertisement that provides *either* o_1 *or* o_2 but *not necessarily both* o_1 *and* o_2 (e.g., a `choice` process). Indeed, in order to clearly specify the behaviour of such service one would have to provide two service profiles corresponding to the two alternatives. As one may note this would lead to advertising a large number of profiles, even for simple services. Moreover, analysing Web services only through their service profile (i.e., their IOs), severely affects the process of discovery of service compositions that satisfy a request (in the case of complex services). Indeed, as explained in Section 2, the service profile does not describe the internal behaviour of a service and hence it does not provide valuable information needed for composing services.

Bansal and Vidal present in [Bansal 2002; Bansal and Vidal 2003] an improvement to the matchmaking process by using an algorithm based on the DAML-S process model. Similarly to Paolucci et al., their algorithm takes as input a query specifying the desired IOs as well as a repository of DAML-S Web services and returns one of the following degrees of match: `exact`, `plug-in`, `subsumes`, or `failed`. A service request matches a service advertisement if the request provides all the inputs (possibly more) needed by the advertisement while the advertisement generates all the outputs (possibly more) needed by the requester. The algorithm of Bansal and Vidal constructs a tree for each process model analysed during the matchmaking process. Indeed, as described in Section 2, the process model decomposes a service into its constituent processes, which can be either composite (i.e., they consist, in turn, of other processes) or atomic (i.e., they are not decomposable). Due to the hierarchical nature of a service, Bansal and Vidal represent it as a tree, where composite and atomic processes respectively correspond to intermediary nodes and leaves. The root of the tree is the composite process representing the entire Web service.

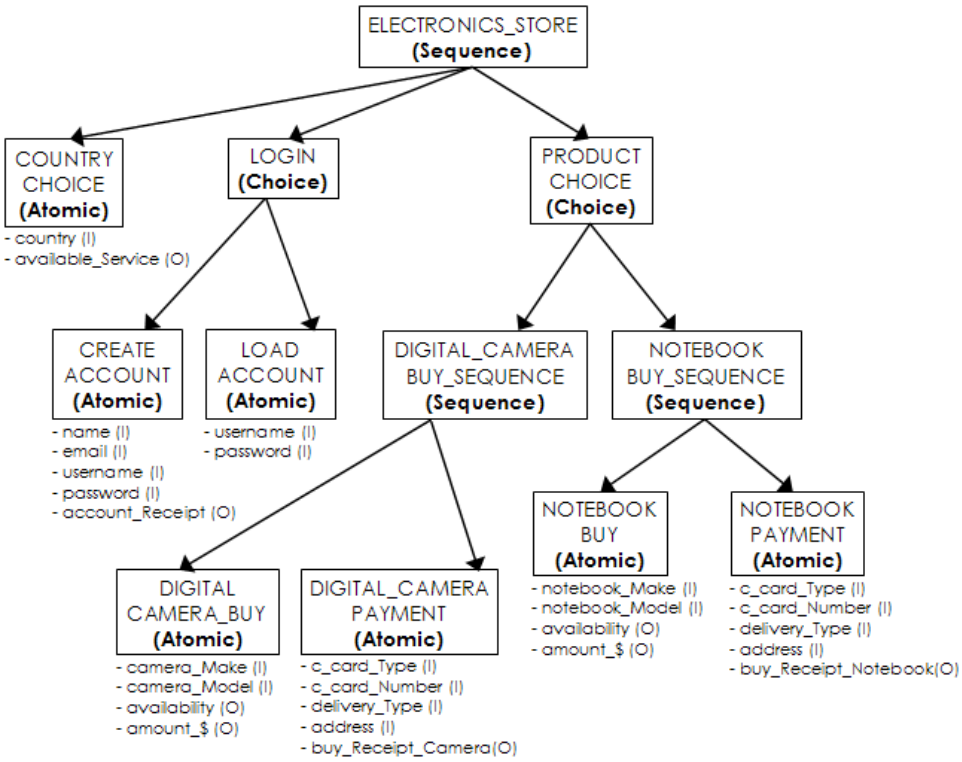

Fig. 2. Process model of an Electronics Store Service.

Let us consider the `Electronics_Store` service presented in Section 2: it consists of a `sequence` process composed by an atomic process, `country_choice`, as well as of two `choice` processes, `login` and `product_choice`. The root of the generated

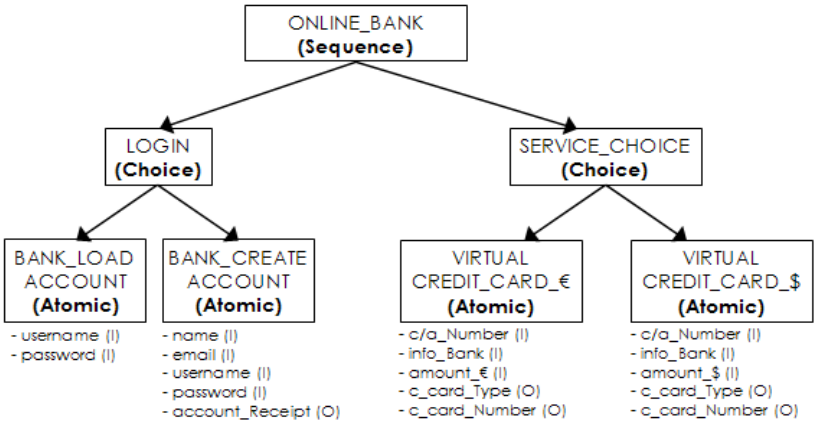

Fig. 3. Process model of an Online Bank Service.

tree corresponds to the `Electronics_Store` process. The root has three children: a leaf corresponding to the `country_choice` process, and two intermediary nodes corresponding to the `login` and the `product_choice` processes. The `login` process is composed in turn of two atomic processes, `create_account` and `load_account`. In the generated tree, the intermediary node `login` has two leaves corresponding to the `create_account` and the `load_account` processes. At the end of this process the tree shown in Figure 2 is obtained. The algorithm of Bansal and Vidal takes into account the process model trees of the advertisements as well as the ontological relations between matched IOs. The matchmaking algorithm begins at the root of the advertisement tree and recursively visits all its subtrees finishing at the leaves. For each composite node (e.g., `sequence`, `choice`, and so on) as well as for atomic nodes a corresponding matching algorithm is employed. For example, in the case of a `sequence` process, if the outputs requested by the query can be satisfied by all its children collectively then we have a success, otherwise a failure. In the case of a `choice` process we get a success or a failure depending on whether there exists at least one child able to provide by itself all the outputs desired by the query. In the case of an atomic node, we have a match if all its inputs are contained in the query and if its outputs contain at least one query output. A detailed description of the matching algorithms corresponding to several composite processes can be found in [Bansal 2002].

Two of the main limitations of the approaches proposed by Paolucci et al. as well as by Bansal and Vidal are single service discovery and single service execution. Indeed, their algorithms are not considering neither compositions of services nor multiple executions of services, and look (inside a repository) for a *single* service capable to fulfil the request by itself.

For example, let us consider a registry containing two services: `Electronics_Store` (Figure 2) and `Online_Bank` (Figure 3). The former sells electronic items like notebooks or digital cameras. The latter is able to create virtual credit cards: a client obtains a credit card number and a credit card type through a bank transfer. We assume for simplicity that all concepts contained in the OWL-S descriptions

are defined in a single shared ontology. Consider next the query specifying:

– inputs: `username`, `password`, `country`, `camera_Model`, `camera_Make`, `notebook_Model`, `notebook_Make`, `c/a_Number`, `info_Bank`, `delivery_Type` and `address`, and
– outputs: `buy_Receipt_Camera` and `buy_Receipt_Notebook`.

The algorithms of Paolucci et al. as well as of Bansal and Vidal give a `failed` match because there is no service in the registry able to fulfil the request by itself. One may note that `c_card_Type` and `c_card_Number` are needed as inputs by both atomic processes `Digital_Camera_Payment` and `Notebook_Payment` of the `Electronics_Store` service, yet they are not provided by the query. Still, they can be obtained by executing the `Online_Bank` service. In other words, while the first service cannot satisfy the query, a suitable composition of the two services can. Moreover, the two atomic processes `Digital_Camera_Payment` and `Notebook_Payment` are contained in two distinct subtrees whose common root is a `choice` process. For this reason, such processes cannot be both executed in a single run. Hence, both the `Electronics_Store` and `Online_Bank` services have to be executed *twice*.

In the following section we present SAM – a composition-oriented algorithm for service discovery that overcomes the above described limitations, on the one hand by analysing the OWL-S process model of the advertised services and, on the other hand by performing a fine-grained matchmaking. Our algorithm searches for a suitable composition capable to satisfy the request. SAM is able to cope with multiple executions of services and it returns a precise sequence of atomic processes that the client has to invoke in order to obtain the desired outputs.

4. SERVICE AGGREGATION MATCHMAKING (SAM)

Before showing in detail the behaviour of our approach, it is worth to specify some assumptions. We firstly suppose that SAM is able to access a (local) service registry storing OWL-S descriptions. Moreover, we do not deal with cross-ontology issues, since, for simplicity, we suppose that all the concepts referenced by the OWL-S descriptions are defined in a single shared ontology. Although we left this feature as future work, it is worth mentioning that full-fledged cross-ontology matchings over service descriptions employing different ontologies could be addressed, for example, by plugging-in existing "ontology-crossers", such as [Navas-Delgado et al. 2005]. Finally, we assume that SAM input queries are expressed in terms of required inputs and produced outputs of the service the client is searching for. We leave the inclusion of additional information such as preconditions, effects or Quality of Service (as in [Kritikos 2005]) as future work.

SAM adopts the matchmaking strategy proposed in [Bansal and Vidal 2003], as it involves the process model in the phase of service discovery. However, SAM extends the approach of Bansal and Vidal by introducing a more flexible matching. Its goal is to determine whether a query can be satisfied by a (composition of) service(s), advertised in an OWL-S registry.

The algorithm consists of three main phases (which will be described in the next three subsections):

(1) Translation of each service in the registry into a tree structure;

(2) Construction of a graph representing the dependencies among atomic processes of the matched services;
(3) Analysis of such dependency graph to determine a service composition capable to satisfy the query (or part of it, when no service composition can fully satisfy the query).

4.1 Translation of services into trees

The first phase of the algorithm consists of two steps: in the first one, SAM translates services into trees, and in the second one, it computes dependencies among their leaves.

Step 1. According to [Bansal and Vidal 2003], SAM stores services as trees, by exploiting the hierarchical nature of their process models. As previously described in Section 3, intermediary nodes correspond to composite processes and may have the type `sequence`, `choice`, `if-then-else`, `split`, `split+join`, `any-order`, `iterate`, `repeat-until` and `repeat-while`, while leaves correspond to atomic processes. However, in order to simplify the next phase of the algorithm (i.e., the construction of the dependency graph), such trees have to be translated further. The following recursive function Transf generates trees containing only `sequence`-, `choice`-, `split+join`- and `split`-typed intermediary nodes.

- $\mathsf{Transf}(\texttt{atomic}(A)) = A$;
- $\mathsf{Transf}(\texttt{sequence}(P_1, ..., P_n)) = \texttt{sequence}(\mathsf{Transf}(P_1), ..., \mathsf{Transf}(P_n))$;
- $\mathsf{Transf}(\texttt{choice}(P_1, ..., P_n)) = \texttt{choice}(\mathsf{Transf}(P_1), ..., \mathsf{Transf}(P_n))$;
- $\mathsf{Transf}(\texttt{if-then-else}(P, Q)) = \texttt{choice}(\mathsf{Transf}(P), \mathsf{Transf}(Q))$;
- $\mathsf{Transf}(\texttt{split}(P_1, ..., P_n)) = \texttt{split}(\mathsf{Transf}(P_1), ..., \mathsf{Transf}(P_n))$;
- $\mathsf{Transf}(\texttt{split+join}(P_1, ..., P_n)) = \texttt{split+join}(\mathsf{Transf}(P_1), ..., \mathsf{Transf}(P_n))$;
- $\mathsf{Transf}(\texttt{any-order}(P_1, ..., P_n)) =$
  $\texttt{choice}(\texttt{sequence}(P'_1, ..., P'_n), ..., \texttt{sequence}(P'_n, ..., P'_1))$
  where $P'_i = \mathsf{Transf}(P_i)$ **with** $i = 1, ..., n$;

- $\mathsf{Transf}(\texttt{iterate}(P)) =$
  $n = \mathsf{Alt}(P)$;
  $P' = \mathsf{Transf}(P)$;
  if $(n = 1)$ **then return** P'
  else return$(\texttt{choice}(P', \texttt{sequence}(P', P'), ..., \texttt{sequence}(\underbrace{P', ..., P'}_{n})))$;

As one may note, Transf does not affect `sequence`, `choice`, `split+join` and `split` nodes, while it defines how `if-then-else`, `any-order` and `iterate` have to be translated. More precisely, `if-then-else` nodes are replaced by `choice` nodes. Indeed, `if-then-else` behaves as `choice`, except for the specified condition, which, being a run-time issue, is not taken into account by SAM. `Any-order` is expanded into a `choice` among all possible permutations of its child nodes. Figure 4, for example, shows how Transf translates an `any-order` with two atomic children.

It is important to note that SAM tackles the discovery process by matching the concepts (i.e., types) corresponding to the inputs/outputs of the query with the

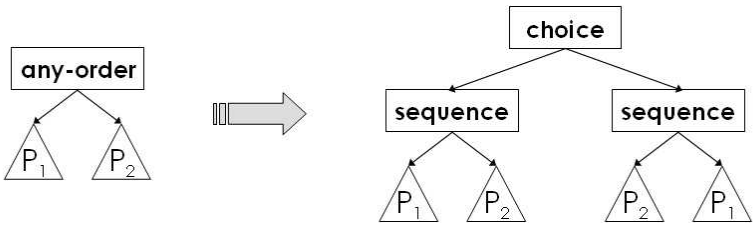

Fig. 4. Translation of `any-order`.

concepts corresponding to the inputs (offered)/outputs (requested) by the advertisements. (For example, SAM matches a query asking for two *car* outputs with an advertisement that outputs a *car*, even though the query specifies multiple data outputs.) Consequently, SAM expands the `iterate` into a `choice` among all possible iterated executions of its child node, as illustrated in Figure 5. In other words, SAM transforms an `iterate`(P) process into a `choice` of sequences, where each sequence is a possible iterated execution of P. The auxiliary function Alt (given hereafter) supports Transf during the translation of `iterate`(P) and it computes the number of times the process P has to be executed in order to yield all the concepts producible by its atomic process children.

Namely, $\mathsf{Alt}(\texttt{atomic}(A)) = 1$;
$\mathsf{Alt}(\texttt{choice}(P_1, ..., P_n)) = \mathsf{Alt}(P_1) + ... + \mathsf{Alt}(P_n)$;
$\mathsf{Alt}(\texttt{sequence}(P_1, ..., P_n)) = \mathsf{Alt}(\texttt{any-order}(P_1, ..., P_n))$
$= \mathsf{Alt}(\texttt{split}(P_1, ..., P_n)) = \mathsf{Alt}(\texttt{split+join}(P_1, ..., P_n))$
$= \mathsf{Alt}(P_1) \times ... \times \mathsf{Alt}(P_n)$;
$\mathsf{Alt}(\texttt{iterate}(P)) = 0$;

Note that if an `iterate` process P has an `iterate`-typed ancestor Q, then the algorithm transforms first Q and then P. That is why we set $\mathsf{Alt}(\texttt{iterate}(P)) = 0$ in the pseudo-code above. It is important to note that this definition of $\mathsf{Transf}(\texttt{iterate}(P))$ is suitable when P has to be executed at least once (i.e., `iterate` instantiated as `repeat-until`). If P may be skipped (i.e., `iterate` instantiated as `repeat-while`), the definition of $\mathsf{Transf}(\texttt{iterate}(P))$ can be easily expanded by adding a *nop* branch to the `choice` among all possible iterated executions of P.

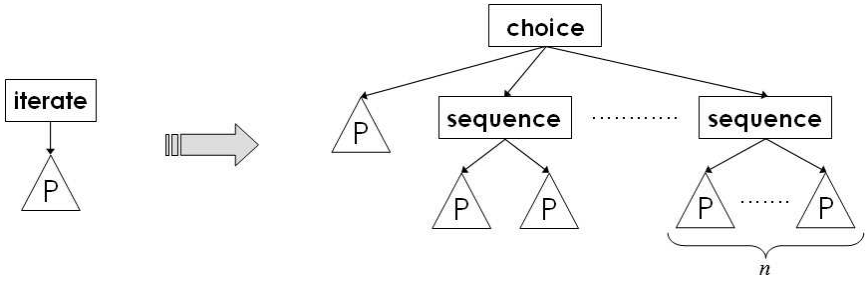

Fig. 5. Translation of `iterate`.

Let us consider, for example, an `iterate` process (see Figure 6) whose child is a `choice` process (call it C), which consists, in turn, of two atomic processes A_1 and A_2. In order to produce the outputs of both A_1 and A_2, the `choice` process has to be executed twice, first to generate the outputs of A_1, and second to generate the outputs of A_2 (or vice-versa). Indeed, $\mathsf{Alt}(C)$ returns 2, as $\mathsf{Alt}(C) = \mathsf{Alt}(A_1) + \mathsf{Alt}(\mathsf{A}_2) = 1 + 1$.

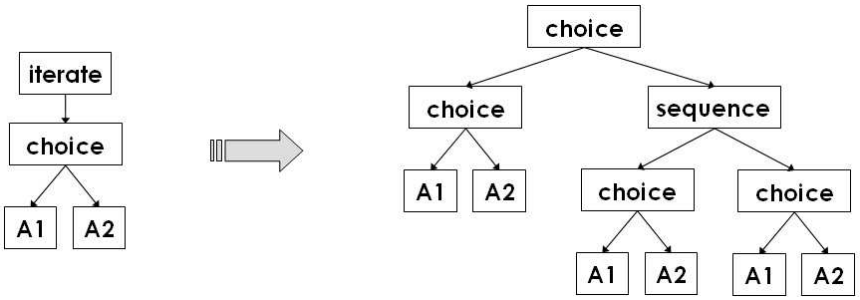

Fig. 6. Example translation of an `iterate` process.

Step 2. At the end of the first step, services are stored as trees including `sequence`, `choice`, `split+join`, `split` and `atomic` nodes. Each tree summarises the interaction protocol (i.e., the execution order of the operations) of the service it describes. The second step computes for each stored tree T and for each atomic node A of T the predecessors of A as well as those nodes which are in a mutual exclusion relationship with A, that is, the nodes that cannot be executed if A is executed.

The set of predecessors of an atomic node A is a set of sets, where each set contains the nodes corresponding to atomic processes that must be completed before executing A. SAM computes the predecessors of A by means of two recursive functions summarised next – Prev, initially invoked over A and its parent, and Last. Intuitively speaking, Prev searches for the closest ancestor of A which is a non left-most child of a `sequence` node. Let R be the left sibling of such ancestor. The predecessors of A are the last executable atomic processes in R, computed by means of the function Last.

- $\mathsf{Prev}(P, nil) = \emptyset$;
- $\mathsf{Prev}(P,\ X) = \mathsf{Prev}(X,\ parent(X))$
  where $X = \texttt{choice}(Q_1, ..., Q_n) \vee$
  $X = \texttt{split}(Q_1, ..., Q_n) \vee$
  $X = \texttt{spli+join}(Q_1, ..., Q_n)$;
- $\mathsf{Prev}(Q_i,\ \texttt{sequence}(Q_1, ..., Q_n)) = \mathsf{Last}(Q_{i-1})$ **with** $i \geq 2$;
- $\mathsf{Prev}(Q_1,\ \texttt{sequence}(Q_1, ..., Q_n)) =$
  $\mathsf{Prev}(\texttt{sequence}(P_1, ..., P_n),\ parent(\texttt{sequence}(P_1, ..., P_n)))$;
- $\mathsf{Last}(A) = \{\{A\}\}$;
- $\mathsf{Last}(\texttt{sequence}(P_1, ..., P_n)) = \mathsf{Last}(P_n)$;
- $\mathsf{Last}(\texttt{choice}(P_1, ..., P_n)) = \mathsf{Last}(P_1) \bigcup ... \bigcup \mathsf{Last}(P_n)$;

- $\mathsf{Last}(\texttt{split}(P_1, ..., P_n)) = \mathsf{Prev}(\texttt{split}(P_1, ..., P_n), \mathit{parent}(\texttt{split}(P_1, ..., P_n)))$;
- $\mathsf{Last}(\texttt{split+join}(P_1, ..., P_n)) = \{t_1 \cup ... \cup t_n \mid t_1 \in \mathsf{Last}(P_1) \wedge ... \wedge t_n \in \mathsf{Last}(P_n)\}$;

For instance, let us consider the `Electronics_Store` service illustrated in Figure 2. In order to compute the predecessors of `Notebook_Buy` atomic node, we invoke the function Prev with parameters `Notebook_Buy` and `Notebook_Buy_Sequence` (i.e., the parent of `Notebook_Buy`). In this example, `Prev` returns the last executable atomic processes of `Login` node (i.e., Last(`Login`)), namely, $\{\{$ `Create_Account`$\}$, $\{$`Load_Account`$\}\}$. Therefore, either `Create_Account` or `Load_Account` must be completed before executing `Notebook_Buy`.

The set of nodes which are in a mutual exclusion relationship with an atomic node A consists of those nodes that cannot be executed if A is executed. SAM computes such set by means of two recursive functions, FindChoice, initially invoked over an atomic node and its parent, and Leaf, defined below. The intuitive behaviour of the function FindChoice is the following. Given an atomic node A, for each `choice`-typed ancestor P of A, FindChoice collects the leaves of each child node of P (computed by the auxiliary simple function Leaf) with the exception of the child including A. Such collected leaf nodes are in a mutual exclusion relationship with A.

- $\mathsf{FindChoice}(P, nil) = \emptyset$;
- $\mathsf{FindChoice}(P, \texttt{choice}(Q_1, ..., Q_n)) =$
  $\mathsf{FindChoice}(\texttt{choice}(Q_1, ..., Q_n), \mathit{parent}(\texttt{choice}(Q_1, ..., Q_n))) \cup$
  $\bigcup_{i=1}^{n} \mathsf{Leaf}(\mathsf{Q_i})$ **with** $(Q_i \neq P)$;
- $\mathsf{FindChoice}(P, X) = \mathsf{FindChoice}(X, \mathit{parent}(X))$
  where $X = \texttt{sequence}(Q_1, ..., Q_n) \vee$
  $X = \texttt{split}(Q_1, ..., Q_n) \vee$
  $X = \texttt{spli+join}(Q_1, ..., Q_n)$;
- $\mathsf{Leaf}(A) = \{A\}$;
- $\mathsf{Leaf}(\texttt{sequence}(Q_1, ..., Q_n) = \mathsf{Leaf}(\texttt{choice}(Q_1, ..., Q_n)$
  $= \mathsf{Leaf}(\texttt{split}(Q_1, ..., Q_n) = \mathsf{Leaf}(\texttt{split+join}(Q_1, ..., Q_n)$
  $= \bigcup_{i=1}^{n} \mathsf{Leaf}(\mathsf{Q_i})$;

For example, let us consider again the `Electronics_Store` service depicted in Figure 2. In order to compute the set of leaf nodes in exclusive relationship with the `Notebook_Buy` atomic process, we invoke FindChoice with parameters `Notebook_Buy` and `Notebook_Buy_Sequence` (i.e., the parent of `Notebook_Buy`). In this case, FindChoice returns the leaves of the `Digital_Camera_Buy_Sequence` node (i.e., Leaf(`Digital_Camera_Buy_Sequence`)), namely, {`Digital_Camera_Buy`, `Digital_Camera_Payment`}. This means that if `Notebook_Buy` is executed, both `Digital_Camera_Buy` and `Digital_Camera_Payment` cannot be executed. Dually, `Notebook_Buy` cannot be executed if `Digital_Camera_Buy` and `Digital_Camera_Payment` have been executed.

Finally, it is worth observing that the two steps of this phase are both completely query-independent. Hence, they can be pre-computed before query answering time, without affecting the efficiency of the matchmaking process.

4.2 Construction of the Dependency Graph

The matchmaking process starts with the second phase, whose objective is to construct a *BF-hypergraph* representing the dependencies among matched services. We hereafter include the defintions of hypergraph, directed hypergraph and BF-hypergraph, as described in [Gallo et al. 1993].

Def. 4.1. A *hypergraph* is a pair $H = (V, E)$, where $V = v_1, v_2, ..., v_n$ is the set of vertices (or nodes) and $E = E_1, E_2, ..., E_m$, with $E_i \subseteq V$ for $i = 1, ..., m$, is the set of *hyperedges*. Note that when $|E_i| = 2$, $i = 1, ..., m$, the hypergraph is a standard graph.

Def. 4.2. A *directed hypergraph* is a hypergraph with directed hyperedges. A *directed hyperedge* is an ordered pair, $E = (X, Y)$, of (possibly empty) disjoint subsets of vertices; X is the tail of E, denoted by $T(E)$, while Y is its head, denoted by $H(E)$.

Def. 4.3. A *BF-hypergraph*, or simply *BF-graph* is a hypergraph whose hyperedges are either B-edges or F-edges. A *backward hyperedge*, or simply *B-edge*, is a hyperedge $E = (T(E), H(E))$ with $|H(E)| = 1$ (Figure 7a). A *forward hyperedge*, or simply *F-edge*, is a hyperedge $E = (T(E), H(E))$ with $|T(E)| = 1$ (Figure 7b).

Fig. 7. A *B-edge* (a) and a *F-edge* (b).

Using the definition of the BF-hypergraphs, we can now describe the behaviour of the second phase. In the matchmaking process performed by SAM, the match regards exclusively atomic processes – the only processes that can be directly invoked by the client. As a consequence, the hypergraph produced as the result of the matchmaking phase has two node types: *process* nodes and *data* nodes, the former corresponding to matched atomic processes and the latter to data taken as input or produced as output by such processes. An hyperedge $E = (T(E), H(E))$ of the dependency graph has one of the following five types:

- $E_{pd} = (\{p\}, O)$: there is a F-edge from a process node p to the set O of data nodes which are outputs of p.
- $E_{dp} = (I, \{p\})$: there is a B-edge from a set I of data nodes which are inputs of p to the process node p.
- $E_{dd} = (D, \{d\})$: there is a B-edge from a set D of data nodes which are subtypes of d to the data node d.
- $E_{sc} = (P, \{p\})$: there is a B-edge from a set P of process nodes which are predecessors of p to the process node p. A hyperedge of type E_{sc} is a *sequencing constraint*.

– $E_{xc} = (\{p\}, P)$: there is a F-edge from a process node p to the set P of process nodes which are in a mutual exclusion relationship with p. A hyperedge of type E_{xc} is an *excluding constraint*.

Initially, the graph contains only the data nodes corresponding to the inputs and the outputs of the query (line 2 in the following pseudo-code). The graph is built during the matchmaking phase that is implemented by a recursive function Match.

The behaviour of Match is summarised by the pseudo-code listed in Figure 8, where I_A and O_A denote the inputs and the outputs of an atomic process A, respectively. Let also PRED_A be the set of atomic processes which have to be executed before A, and CHOICE_A be the set of atomic processes which can be executed only if A is not executed. PRED_A and CHOICE_A are both pre-computed as described previously in Subsection 4.1. The expression $d_1 \prec d_2$ means that d_1 is a sub-concept of d_2 as well as the expression $d \sqsubset H$ means that d is a sub-concept of at least one data node belonging to the dependency graph H. Similarly, the expression $d \sqsupset H$ means that H contains at least one data node that is a sub-concept of d.

The function Match is invoked over each service (line 4) contained in the service registry. Match starts its execution at the root of the process model tree (line 4) and it is recursively invoked over children nodes (lines 15, 23). The execution finishes at leaf nodes, where Match verifies the compatibility between the inputs and the outputs of the corresponding atomic process and the data nodes currently present in the graph. Match deals with the following types of OWL-S nodes: `sequence`, `choice`, `split`, `split+join` and `atomic`. For atomic nodes (line 6), Match checks whether the corresponding atomic process is already contained in the graph (line 7). If this is not the case, Match verifies the *compatibility* between the inputs and the outputs of the atomic node and the data nodes currently contained in the graph (line 9). An atomic process matches if and only if:

– either all its inputs are available because they are part of the query or because they are returned as outputs by other previously matched atomic processes,
– or at least one of its outputs is part of the query or it is an input for some previously matched atomic process.

According to the OWL-S specification [OWL-S Coalition 2004], we assume that an output o is *compatible* with an input i if and only if either o and i represent the same concept (*exact match*), or o represents a sub-concept of i ("o plugs-in i", or equivalently "i subsumes o"). For example, if there is a match between an output o of an atomic process P and a data node d contained in the graph, as o is a subtype of d, we create and insert in the graph a new process node P, a new data node o, a E_{dd}-typed hyperedge from o to d, and a E_{pd}-typed hyperedge from p to o.

When the algorithm finds a new matched atomic process, it invokes the function Add (line 10), which firstly creates a corresponding process node A and adds it to the graph (line 26). For each output o (line 27) of A, Add creates and inserts in the graph (if not already present) a new data node (line 29), and for each data node d in the graph which is a super-type of o (line 30), it adds o to the the tail of the E_{dd}-typed hyperedge, if it exists, whose head is d (line 31). Otherwise, Add inserts a new directed E_{dd}-typed hyperedge from o to d (line 32). Next, Add inserts a new

```
1. Build_DG(ServiceRegistry SR, Query Q, HyperGraph H)
2.     forall data d in Q do add a new data node d to H;
3.     repeat
4.         forall services s in SR do Match(Root(s), H);
5.     until no process node is added to H;

6. boolean Match(Atomic node A, HyperGraph H)
7.     if (A ∈ H) then return true
8.     else
9.         if (∀i ∈ I_A : i ∈ H ∨ i ⊐ H) ∨ (∃o ∈ O_A : o ∈ H ∨ o ⊏ H) then
10.            Add(A, H);
11.            return true;
12.        else return false;

13. boolean Match(Sequence node sequence(P_1, ..., P_n), HyperGraph H)
14.    forall P_i in {P_1, ..., P_n} do
15.        if (Match(P_i, H)) then
16.            forall atomic node A in ({P_1, ..., P_n} \ {P_i}) do
17.                if (A ∉ H) then Add(A, H);
18.            return true;
19.    return false;

20. boolean Match(Choice node choice(P_1, ..., P_n), HyperGraph H)
21.    boolean matched = false;
22.    forall P_i in {P_1, ..., P_n} do
23.        matched = matched ∨ Match(P_i, H);
24.    return matched;

25. Add(Atomic node A, HyperGraph H)
26.    add a new process node A to H;
27.    forall outputs o in O_A do
28.        if (o ∉ H) then
29.            add a new data node o to H;
30.            forall data node d in H : o ≺ d do
31.                if (∃E : E ∈ E_dd ∧ H(E) = {d}) then T(E) = T(E) ∪ {o}
32.                else add a new directed hyperedge ({o}, {d}) ∈ E_dd;
33.    add a new directed hyperedge ({A}, O_A) ∈ E_pd to H;
34.    forall inputs i in I_A do
35.        if (i ∉ H) then
36.            add a new data node i to H;
37.            D = ∅;
38.            forall data node d in H : d ≺ i do D = D ∪ {d};
39.            if (D ≠ ∅) then add a new directed hyperedge E_dd = (D, {i});
40.    add a new directed hyperedge (I_A, {A}) ∈ E_dp to H;
41.    if (R = {P : P ∈ CHOICE_A ∧ P ∈ H} ≠ 0) then
42.        add a new directed hyperedge ({A}, R) ∈ E_xc to H;
43.    forall choice node C in CHOICE_A do
44.        if (C ∈ H) then
45.            if (∃E ∈ E_xc : T(E) = {C}) then H(E) = H(E) ∪ {A}
46.            else add a new directed hyperedge ({C}, {A}) ∈ E_xc to H;
47.    forall sets of predecessors Q in PRED_A do
48.        if (R = {P : P ∈ Q ∧ P ∈ H} ≠ 0) then
49.            add a new directed hyperedge (R, {A}) ∈ E_sc to H;
```

Fig. 8. Pseudo-code of the dependency graph construction.

directed E_{pd}-typed hyperedge from A to the outputs of A (line 33). Dually, for each input i (line 34) of A, Add creates and inserts (if not already present) a new data node (line 36) and a new directed E_{dd}-typed hyperedge from the nodes in the graph which are sub-types of i to i (line 37–39). Next, it adds a new directed E_{dp}-typed hyperedge from the inputs of A to A (line 40). Moreover, if A is in a mutual exclusion relationship with some process nodes included in the graph (line 41), then Add inserts a new directed E_{xc}-typed hyperedge from A to such nodes (line 42). For each process node C in a mutual exclusion relationship with A included in the graph (line 43–44), the algorithm adds A to the head of the E_{xc}-typed hyperedge, if exists, whose tail is C (line 45). Otherwise, Add inserts a new directed E_{xc}-typed hyperedge from C to A (line 46). Finally, for each set of predecessors of A (line 47), the algorithm inserts a new directed E_{sc}-typed hyperedge from such set to A (lines 48–49).

In the case of a `sequence` node (line 13), Match verifies if the corresponding sub-tree (line 14) contains at least one matched atomic node (lines 15). If so, all (matched and unmatched) atomic nodes contained in the sub-tree (line 16) are inserted in the dependency graph, by invoking the function Add (line 17), together with their IOs and all the necessary dependencies and constraints (lines 25–45). For a `choice` node (line 20) Match adds to the graph only the matched atomic node children (line 23).

It is important to observe that in the case of `split` and `split+join` nodes, Match behaves as previously described for `sequence` nodes (lines 13–19). The matchmaking phase cycles over the registry until no more process nodes can be added to the dependency graph (line 5).

Example. Let us consider now the motivating example presented in Section 3, that we will use to illustrate our algorithm throughout the rest of this section. Hence, the registry contains the two services `Electronics_Store` and `Online_Bank`, while the query is the following:

- inputs: `username`, `password`, `country`, `camera_Model`, `camera_Make`, `notebook_Model`, `notebook_Make`, `c/a_Number`, `info_Bank`, `delivery_Type` and `address`, and
- outputs: `buy_Receipt_Camera` and `buy_Receipt_Notebook`.

Let us suppose that Match is invoked first over the `Electronics_Store` service. The root of the `Electronics_Store` advertisement tree is a `sequence` node, so Match verifies if the tree contains at least one matched atomic process. For instance the `Country_Choice` atomic process matches the query, as its input (`country`) belongs to the query inputs. Therefore, all (matched and unmatched) atomic processes of `Electronics_Store` are inserted in the dependency graph. Next, Match is invoked over the `Online_Bank` service, whose root is a `sequence` node. Again, Match verifies if `Online_Bank` contains at least a matched atomic process. For example, `Bank_Load_Account` atomic process matches the query as both its inputs (`username` and `password`) belong to the query inputs. Hence, all the atomic processes of `Online_Bank` are inserted in the dependency graph, as well. The dependency graph produced by SAM is depicted in Figure 9, where data and process nodes are graphically represented by ellipses and rectangles, respectively. One may note, for instance, the excluding constraints between the `Load_Account` and `Create_Account` atomic processes as well as between the `Bank_Load_Account` and

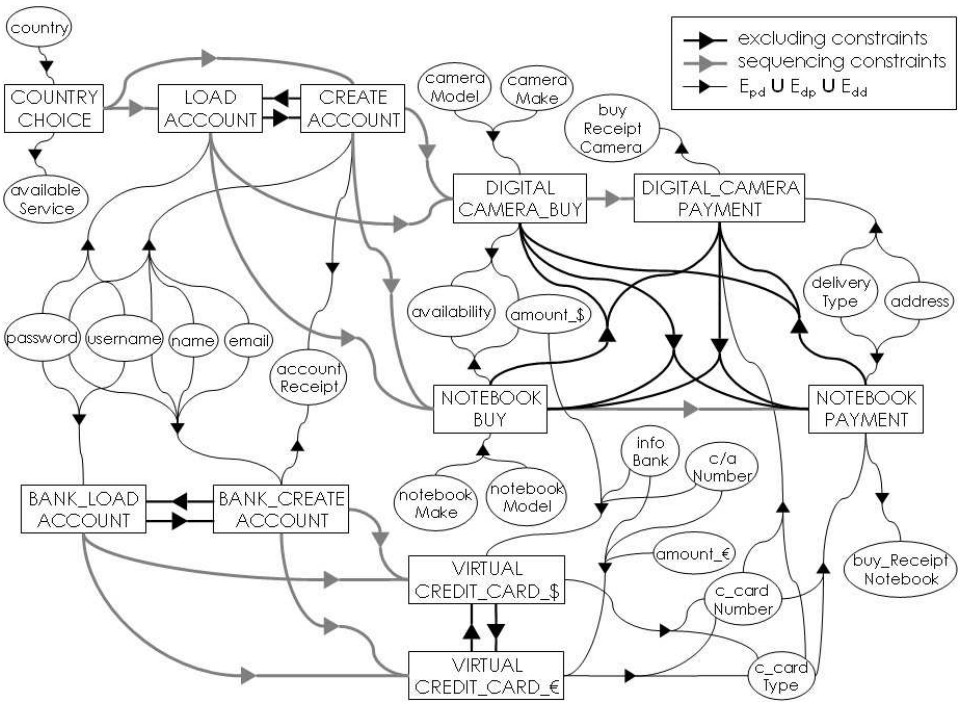

Fig. 9. Dependency graph.

`Bank_Create_Account`. Indeed, all such atomic processes are leaves of subtrees whose common root is a `choice` process. One also may notice the sequencing constraints between `Country_Choice` and `Load_Account`, between `Load_Account` and `Digital_Camera_Buy`, between `Digital_Camera_Buy` and `Digital_Camera_Payment` and so on. Indeed, all these atomic processes are leaves of a subtree whose root is a `sequence` process.

◇

4.3 Analysis of the Dependency Graph

The second phase of the algorithm analyses the dependency graph constructed during the first phase and it consists of the five steps described next. The corresponding pseudo-code, which is referenced in the following paragraphs (by indicating the referenced pseudo-code line number) is presented at the end of this subsection.

Step 1. **Reachability of query outputs**. The dependency graph includes a data node for each query input and output, regardless of whether or not these data have been matched during the first phase. SAM firstly checks whether there are query output nodes in the graph H that do not have incoming hyperedges from process nodes (line 1). Indeed, such disconnected query outputs cannot be produced as no service in the registry can generate them. If there are disconnected query outputs in the graph (line 2), the client has to choose (line 3) whether the matchmaking process should nevertheless continue, by discarding such outputs from the query (line 4), or abort (line 5). In the latter case SAM terminates with a FAILURE (line

5). In the former case SAM removes the disconnected query outputs (line 4) and continues with Step 2.

Example. The dependency graph illustrated in Figure 9 contains no disconnected query outputs. Indeed, in this example, all query outputs are produced by at least an atomic process. ◇

Step 2. **Yellow Colouring**. In this step SAM identifies — by colouring them in yellow — all processes which *may* be useful for generating the query outputs. Initially all nodes in the graph are white (line 6). The white colour is used to denote all process and data nodes that do not have yet proved to be useful for satisfying the query. SAM first colours in yellow all the query outputs (line 7). It then recursively paints in yellow all process and data nodes that are white and that belong to the tail of an hyperedge whose head includes at least one yellow coloured node (lines 8–9). Note that the yellow paint spreads over subtype data nodes (i.e., if d_1 is yellow, d_2 is white and d_2 is a subtype of d_1, then d_2 is coloured in yellow as well). On the other hand, excluding constraints are not taken into account here (i.e., the yellow paint does not spread over excluding constraints). The process of painting in yellow finishes when there is no other node that can be coloured. At the end of this step all yellow process nodes correspond to processes that might have to be executed to generate the query outputs. Dually, yellow data nodes correspond to data that "might be useful as input"/"might be generated as output" to/by yellow processes in order to generate the query outputs. All nodes that are still white at the end of this phase are not needed for fulfilling the request (and could be removed from the graph). One may note that more nodes than necessary may have been painted. The algorithm then continues with Step 3.

Example. During the second step, SAM paints in yellow all the data nodes corresponding to the query outputs and it recursively paints in yellow all process and data nodes linked to other yellow nodes. At the end of this step all nodes contained in the graph are painted in yellow with the exception of `available_Service`, `account_Receipt` and `availability` (data nodes). Indeed, such data nodes are not part of the query outputs and they are not taken as input by a process node useful for satisfying the client request. The dependency graph at the end of the second step is illustrated in Figure 10. ◇

Step 3. **Red&Black Colouring**. The goal of this step is to identify — by painting them in red — the processes which contribute to generate the query outputs and which can actually be executed if the query inputs are provided. To describe this step it is convenient to introduce the notion of *firable* process. *A process node P is* firable *in a hypergraph H if P is yellow and all its input data nodes are red, and if there is at least one set of predecessors of P then at least one set of predecessors of P is red coloured.* (A set S of processes is red coloured if and only if all processes in S are red coloured). The algorithm firstly paints in red all data nodes corresponding to the query inputs (line 11). While there is at least one yellow query output node and at least one *firable* process (line 12), the algorithm selects a *firable* process for execution (line 14). If there are several *firable* processes linked through excluding constraints then SAM non-deterministically chooses one such

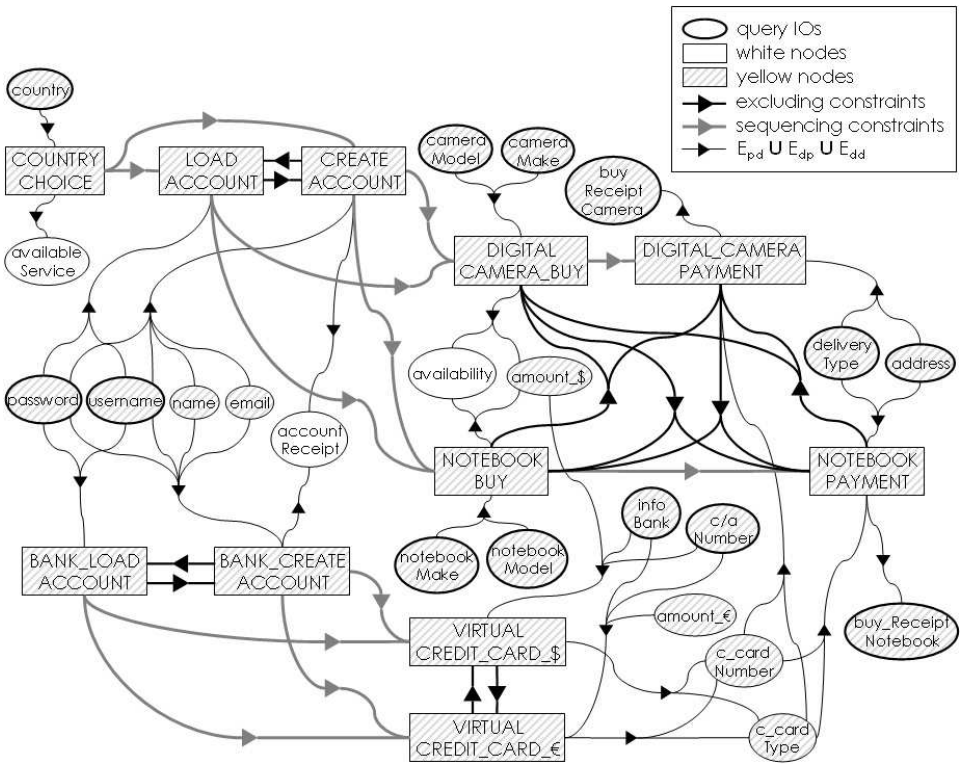

Fig. 10. Dependency graph at the end of the second step.

firable process node (line 15) and paints it in red (line 16). Every non-deterministic choice corresponds to a "fork" into several instances. After painting a process in red, all its output data nodes are coloured in red (line 17) and all their supertype data nodes are coloured in red (line 18) as well (i.e., the red paint spreads over outgoing data to data hyperedges). Moreover all the process nodes linked to it by excluding constraints are inhibited by painting them in black (line 19). (We do so as, for example, by colouring in red a `Pay_with_Credit_Card` process we should inhibit another `Pay_with_Cash` process linked to it by an excluding constraint.) When painting in red a process node, the algorithm adds it to an initially empty PROCESS_SEQUENCE list (line 16).

It is worth noting that, once a process node is coloured in red, it can not be coloured in black at a later moment, as proved in the following.

PROPERTY 4.4. *A red process node cannot be coloured in black.*

PROOF. Consider a red node R. R might be coloured in black at a later moment by another node B such that B *is yellow* and there is an excluding constraint between R and B. Furthermore, due to the excluding constraint between R and B, and due to the fact that R has been coloured in red, we infer that B *has been coloured in black as well* (together with all the other processes having excluding constraints with R). However, this is a contradiction as B cannot be yellow and

black at the same time. □

Each instance of this step finishes either with a SUCCESS if all query outputs become red (line 21), or with a FAILURE if there are no more *firable* processes but there is still at least one yellow query output (line 20). It is important to note that if there are several *firable* processes linked through excluding constraints then the non deterministic CHOOSE operator (line 15) splits the current execution of this step into a number of instances equal to the number of *firable* processes, each such instance corresponding to painting in red the respective process node and further on its outputs as well as to inhibiting the processes linked to it by excluding constraints. As a result of this step we shall obtain a set of triples

<SUCCESS/FAILURE, coloured graph H, PROCESS_SEQUENCE>.

Example. During the third step SAM starts by painting in red all yellow data nodes corresponding to the query inputs. At this point, only `Country_Choice` and `Bank_Load_Account` process nodes are *firable*, as all their data inputs are red and they have no predecessors. Let us consider that SAM chooses to execute the `Country_Choice` process. By doing so, SAM paints it in red and adds it to the PROCESS_SEQUENCE list. Moreover, the `Load_Account` process becomes *firable* as its predecessor is now red. By further assuming that SAM selects the `Load_Account` process for execution, SAM paints `Load_Account` in red and then inhibits the `Create_Account` process by painting it in black. One may note that in this case the algorithm does not need to split the execution in two instances as the `Load_Account` is linked through an excluding constraint to a process node which is not *firable*. Next, SAM paints in red the `Bank_Load_Account`, adds it to the PROCESS_SEQUENCE list and inhibits the `Bank_Create_Account` process. At this moment both `Digital_Camera_Buy` and `Notebook_Buy` process nodes are *firable*. Given that they are linked through excluding constraints, SAM splits the execution in two instances: the first one paints in red the `Digital_Camera_Buy` process and it paints in black the `Notebook_Buy` and `Notebook_Payment` processes, while the second paints in red the `Notebook_Buy` process and it paints in black the `Digital_Camera_Buy` and `Digital_Camera_Payment` processes. The first instance continues by painting in red `Virtual_Credit_Card_$` and `Digital_Camera_Payment` and by inhibiting the `Virtual_Credit_Card_€`. Next, it finishes with a FAILURE as it generates only the `buy_Receipt_Camera` query output. Similarly, the second instance paints in red `Virtual_Credit_Card_$` and `Notebook_Payment`, it inhibits the `Virtual_Credit_Card_€` and it terminates with a FAILURE as it produces the `buy_Receipt_Notebook` query output only. Hence, the PROCESS_SEQUENCE list resulting from the first instance is [`Country_Choice`, `Load_Account`, `Bank_Load_Account`, `Digital_Camera_Buy`, `Virtual_Credit_Card_$`, `Digital_Camera_Payment`]. The second instance produces the following PROCESS_SEQUENCE list: [`Country_Choice`, `Load_Account`, `Bank_Load_Account`, `Notebook_Buy`, `Virtual_Credit_Card_$`, `Notebook_Payment`]. The dependency graph resulting from the first instance at the end of the third step is shown in Figure 11. ◇

Step 4. **Analysis of Triples**. The algorithm further checks whether there exists at least one tuple <SUCCESS, H, PROCESS_SEQUENCE> (line 22). If so, it returns

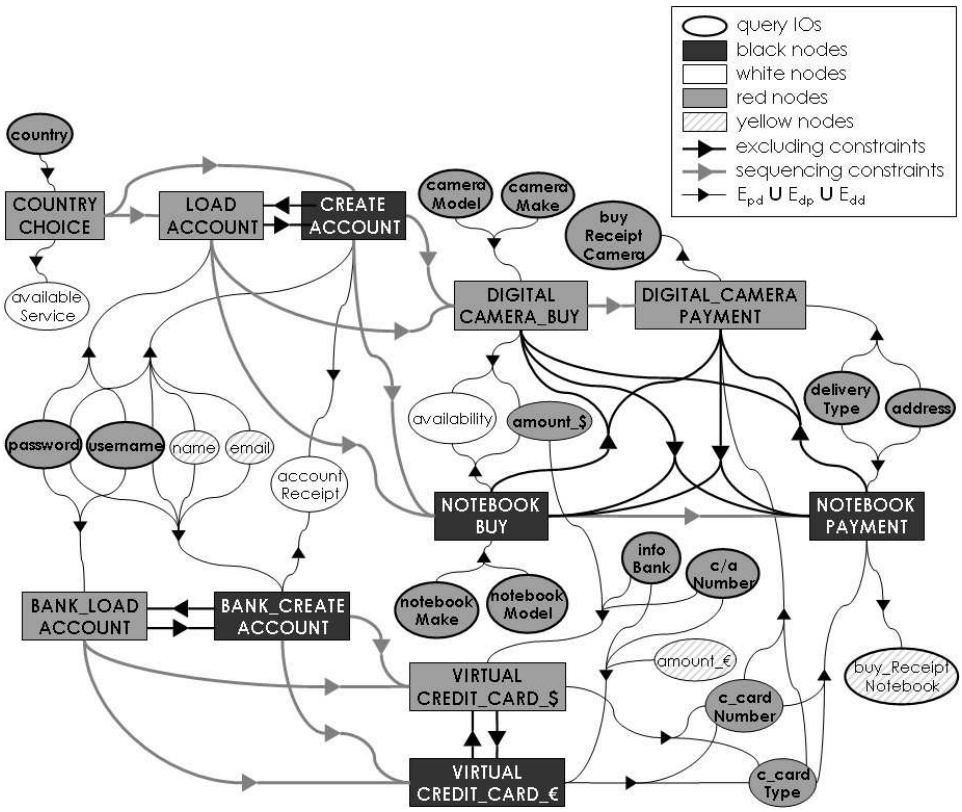

Fig. 11. Dependency graph at the end of the third step.

to the client an ordered list of all the successful tuples $T_i = \{<$SUCCESS, H_i, PROCESS_SEQUENCE$_i>\}$ (line 23). Such list can be ordered by taking into account the client's preferences (expressed together with the query). Such preferences can include minimal number of matched services, minimal PROCESS_SEQUENCE length, and so on.

Now, in the case that all the triples generated by Step 3 are FAILUREs (line 24), SAM checks whether there exists a set of FAILUREs that together are able to generate all outputs requested by the query (line 25). This means that each query output has to be produced (i.e., has to be coloured in red) by at least one failure in such set. If so, the request can be satisfied by simply considering one sequence of FAILUREs in this set. It is important to note again that the choice of such set is made with respect to client's preferences. If such a set exists, the process finishes by returning to the client a sequence obtained by the concatenation of all PROCESS_SEQUENCEs corresponding to the considered FAILUREs in the set (line 26). In this case we have a SUCCESS obtained from the aggregation of a set of FAILUREs.

If instead there is no such set of FAILUREs that can collectively satisfy the query (line 27), it means that there are query outputs that remain yellow in all graphs obtained at the end of Step 3. In such case, the algorithm returns a partial match, namely,

a set of FAILUREs able to collectively produce some, yet not all query outputs. The algorithm then computes the intersection of the sets of all the unsatisfiable query outputs for all FAILUREs (line 30). Next, similarly to the previous case, it considers a set of FAILUREs able to collectively satisfy the producible outputs (line 32) (i.e., the query outputs less the unsatisfiable ones (line 31)). The algorithm further asks the client whether it wishes more information with respect to what is needed to completely satisfy the request (line 28). This information consists of the additional inputs that are needed in order to be able to unlock and to execute other processes so as to fully satisfy the request. If the client agrees (line 29) then SAM continues with Step 5. Otherwise it terminates (line 39).

Example. At the fourth step SAM starts by checking whether there exists at least one instance returning a SUCCESS. Due to that both instances generated during the previous step return a FAILURE, SAM checks whether their union is able to generate all the requested outputs. Consequently, SAM obtains a SUCCESS from the aggregation of the PROCESS_SEQUENCEs corresponding to the two FAILUREs. SAM finishes by returning to the client the following PROCESS_SEQUENCE: [[`Country_Choice`, `Load_Account`, `Bank_Load_Account`, `Digital_Camera_Buy`, `Virtual_Credit_Card_$`, `Digital_Camera_Payment`], [`Country_Choice`, `Load_Account`, `Bank_Load_Account`, `Digital_Notebook_Buy`, `Virtual_Credit_Card_$`, `Notebook_Payment`]]. ◇

Step 5. **Individuating Additional Inputs**. During this last step the algorithm looks for additional inputs that have to be provided in order to have further *firable* processes that help generating the unsatisfiable query outputs. Hence, for each FAILURE (line 33) and for each unsatisfied output (line 34), SAM looks for yellow process nodes that generate this output (line 36). The set of additional inputs needed for producing this output in the respective FAILURE firstly comes from considering all yellow input data nodes of these processes (line 37). Morever, each such process could be the head of a chain of sequencing constraints. This means that all processes on such chain have to be executed before the head process, as they are its predecessors. Hence, the set of additional inputs will contain also the yellow input data nodes of all such predecessors (lines 40–46). Then, since an output can be generated by two or more processes, and since a process can have more than one yellow predecessors, several sets of additional inputs may be computed. In such a case SAM returns an ordered list comprising all the possible sets of additional inputs (lines 35,43). Such list can be ordered by taking into account some client preferences, such as minimal number of matched services, minimal number of suggested additional inputs and so on.

Example. For this example, the last step of the algorithm is not executed as the request has been fulfilled.

To illustrate the behaviour of this step, let us now consider the same query presented in Section 3 where the `country` concept has been removed. At the end of the second step, the produced dependency graph is the same as the one illustrated in Figure 10. At the beginning of the third step only the `Bank_Load_Account` process is *firable*. Hence, SAM paints it in red and adds it to the PROCESS_SEQUENCE list, while the `Bank_Create_Account` process is painted in black. At this point, there are no more *firable* processes, but there are still yellow query outputs. Therefore,

— STEP 1 → Reachability of Query Outputs —
1. $U = \{d \mid d \in O_Q \wedge \nexists P, D : (\{P\}, D) \in E_{pd} \wedge d \in D)\}$;
2. **if** $U \neq \emptyset$ **then**
3. Query client whether to go ahead ignoring U;
4. **if** client says yes **then** $O_Q = O_Q \setminus U$;
5. **else return**(*"The query cannot be satisfied"*);
6. Paint in white all X s.t. $X \in N$;
— STEP 2 → Yellow Colouring —
7. Paint in yellow all d s.t. $d \in O_Q$;
8. **while** $(\exists E \in E_{dp} \cup E_{pd} \cup E_{dd} \cup E_{sc} \wedge (\exists Y \in H(E) : Yyellow) \wedge (\exists X \in T(E) : Xwhite))$ **do**
9. **forall** X **in** $T(E)$ **do** Paint X in yellow;
— STEP 3 → Red & Black Colouring —
10. Initialise ProcessSequence;
11. Paint in red all d s.t. $d \in I_Q \wedge d$ yellow;
12. **while** $(Firable(H) \neq \emptyset \wedge \exists d \in O_Q$: d yellow) **do**
13. **if** $\exists P' : (P' \in Firable(H)) \wedge (\nexists E = (\{P'\}, H(E)) \in E_{xc} : H(E) \cap Firable(H) \neq \emptyset)$ **then**
14. $P = P'$;
15. **else** $P =$ CHOOSE$(Firable(H))$;
16. Paint P in red and add P to ProcessSequence;
17. $\forall d : d$ yellow $\wedge \exists E = (\{P\}, H(E)) \in E_{pd} : d \in H(E)$: paint d in red;
18. $\forall d' : d'$ yellow $\wedge \exists E = (T(E), \{d'\}) \in E_{dd} : d \in T(E) \wedge d$ red : paint d' in red;
19. $\forall P' : P'$ yellow $\wedge \exists E = (\{P\}, H(E)) \in E_{xc} : P' \in H(E)$: paint P' in black;
20. **if** $\exists d : d$ yellow $\wedge d \in O_Q$ **then** FAILURE;
21. **else** SUCCESS;
— STEP 4 →Analysis of Triples —
22. **if** there exists at least one SUCCESS **then**
23. **return** *an ordered list of (successful) results*;
24. **else**
25. **if** ∃ a set S of FAILUREs s.t. $\forall d \in O_Q \; \exists F \in S : d \in O_F$ **then**
26. **return** *a concatenation of the ProcessSequences of all graphs in* S;
27. **else**
28. Query client whether it wants info on additional inputs;
29. **if** client says yes **then**
30. $NonProducibleOutputs = \{d | d \in O_Q \wedge \forall$ FAILURE $F : d$ yellow in $F\}$;
31. $ProducibleOutputs = O_Q \setminus NonProducibleOutputs$;
32. Let S be a set of FAILUREs s.t.$\forall d \in ProducibleOutputs \; \exists F \in S : d \in O_F$;
— STEP 5 → Individuating Additional Inputs —
33. **forall** FAILURE F **in** S **do**
34. **forall** $d \in NonProducibleOutputs$ **do**
35. **print** *"Additional inputs nedeed for"* + d + *"in"* + F + *":"*;
36. $P = \{Q \mid \exists E = (\{Q\}, H(E)) \in E_{pd} \wedge d \in H(E)\}$;
37. **forall** P' **in** P **do** AdditionalInputs$(P', F, \emptyset)$;
38. **return** (*a concatenation of the ProcessSequences of all graphs in* S);
39. **else** FAILURE;

40. AdditionalInputs(set of process nodes P, Failure F, set of data nodes AI)
41. **forall** P' **in** P **do** $AI = AI \cup \{d \mid (d \in I_{P'} \wedge d$ yellow$)\}$;
42. **if** $(\nexists E = (T(E), H(E)) \in E_{sc} : H(E) = \{Q\}, Q \in P)$ **then print** "–" + AI;
43. **else**
44. **forall** $(t_1, ..., t_n)$ **in** $\{\{T(E_1) \cup ... \cup T(E_n)\} \mid$
45. $\exists E_i = (T(E_i), \{P_i\}) \in E_{sc} : P_i \in P, i \in (1...n), n = |P|\}$ **do**
46. AdditionalInputs$((t_1, ..., t_n), F, AI)$;

Fig. 12. Pseudo-code of the dependency graph analysis.

the third step ends with a FAILURE. As the only available triple is a FAILURE, the fourth step is unable to find a set of FAILUREs that collectively are able to provide all query outputs. Next, SAM asks the client whether it wants more information about the generated process sequence list and about how it may be possible to fully satisfy the query. If it agrees, SAM continues with the fifth step when it looks for each unsatisfied output for yellow process nodes that generate it. In our case, the process nodes `Digital_Camera_Payment` and `Notebook_Payment` can respectively generate the outputs `buy_Receipt_Camera` and `buy_Receipt_Notebook`. Yet, their executions are conditioned by the execution of their predecessors and moreover, such processes need `c_Card_Number` and `c_Card_Type` to be provided as inputs. Hence, a possible list of such additional inputs (for both the unsatisfied outputs) is: {`country`, `c_Card_Number`, `c_Card_Type`}. Indeed, all inputs needed for the execution of predecessor processes of `Digital_Camera_Payment` and `Notebook_Payment` are contained in the query with the exception of `country`.

◇

The pseudo-code listed in Figure 12 summarises the analysis of the dependency graph described so far, where $H = (N, E)$ denotes the dependency graph produced by the second phase (as explained in Subsection 4.2, $E = E_{dp} \cup E_{pd} \cup E_{dd} \cup E_{sc} \cup E_{xc}$). Let I_p and O_p denote the inputs and the outputs of a process P respectively, and let $Q = \{I_Q, O_Q\}$ denote the query. Let also O_F be the set of data output nodes that are red in failure F.

5. ANALYSIS OF COMPLEXITY, CORRECTNESS AND COMPLETENESS

In this section we discuss the time complexity as well as the correcteness and completeness of SAM.

5.1 Complexity analysis

Let us first sketch the worst-case analysis of the time complexity $T_{\mathsf{SAM}}(S)$ of executing SAM on a registry containing $|S|$ services. $T_{\mathsf{SAM}}(S)$ derives from the time needed to perform the two phases at query answering time, namely $T_{GC}(S)$ to construct the dependency graph and $T_{GA}(S)$ to analyse it. To construct the dependency graph, SAM will cycle over the service registry at most $|AP|$ times, where $|AP|$ is the number of atomic processes contained in the services in the registry. At each cycle, SAM will perform at most $|AP|$ invocations to the base case of function Match to determine whether some atomic process P matches. Each of such calls will compare the set D_P of inputs and outputs of the process with the data nodes N_D already present in the graph. Since $|D_P|$ can be maximised by a constant and $|N_D| \in O(|AP|)$, each cycle will perform at most $|AP|^2$ comparisons. Since inserting the matched process requires also at most $|AP|^2$ operations, and since $|AP| \in O(|S|)$, we have that $T_{GC}(S) \in O(|S|^3)$. In the analysis of the dependency graph, the red&black colouring step is the most expensive task, by performing $|N| + |E|$ steps (where $|N|$ and $|E|$ are the number of nodes and edges in the graph, respectively) for each instance. Each non-deterministic choice involves a set of atomic processes $C_i = \{P_1, ..., P_n\}$ such that there is an excluding constraint outgoing from each $P_x \in C_i$ and reaching a $P_y \in C_i$. Consequently, the dependency graph employs a number k of non-deterministic choice sets C_i,

where $k \in [0, \left\lceil \frac{|AP|}{2} \right\rceil]$. Let $c_{max} = max(|C_i|)$, and dually, $c_{min} = min(|C_i|)$, where $i \in [1, k]$. Then SAM can generate at most $c_{max}^{\frac{|AP|}{c_{min}}}$ instances. For each instance the red&black colouring step performs $|N| + |E| \in O(|AP|^2)$ steps. The time complexity of the red&black colouring step is $T_{R\&B}(S) = c_{max}^{\frac{|AP|}{c_{min}}} \cdot O(|AP|^2)$. Hence, the analysis of the dependency graph requires exponential time, namely $T_{GA}(S) \in NP$, and the overall time complexity $T_{\mathsf{SAM}}(S) \in NP$ as well.

Finally, it is worth mentioning the time complexity of the additional inputs step, performed in case of query unsatisfaction. Similarly to the red&black colouring step, it performs $|N| + |E|$ steps for each instance (i.e., failure). Hence, the additional inputs step requires exponential time as well.

While the asymptotic complexity of the version of SAM described in this paper is high, we reckon that the efficiency of SAM can be sensibly improved in the following directions:

- The efficiency of the graph analysis can be improved by enumerating the nondeterministic choices rather than by splitting several instances whenever a CHOOSE operation is executed. While this does not affect the worst-case asymptotic complexity, it does sensibly improve the average response time of SAM, and it has been already implemented in the last version of SAM.
- Reducing the number of candidate services to be considered (in particular during the graph construction phase) would obviously improve the efficiency of SAM. This can be achieved by means of a preselection phase – e.g., based on UDDI as well as on some functional analysis. For instance, UDDI can be used to filter services that match a requested value in a taxonomy such as NAICS or UNSPSC. A reason for doing so comes from the ambiguity of the service parameters. For example, queries requesting travel services that output a "date" would match services from the banking domain, health systems, travel agencies, and so on. This would lead to a large number of services being added to the dependency graph. Still, by firstly filtering services from the "travel domain", for example, services that are of no interest to the client would be discarded.
- Fresh indexing and/or ranking techniques (as search engines do for Web pages) help to reduce the complexity at query time (e.g., anticipating the construction of (part of) the dependency graph). However, the problem in this case seems to be a bit harder than for text indexing due to caching issues raised for at least two reasons: (a) adding/removing one parameter in a query may radically change the results associated to a query, and (b) Web services are typically non-uniform (i.e., not constantly offered/available in the same way over time).

5.2 Correctness

As detailed in Subsection 4.3, in the case of a success, SAM outputs a list of atomic processes which represents the sequence of process invocations to be performed in order to fulfill the given query. Let us now formally define when a successful list of atomic processes satisfies both the *data-flow* and the *control-flow* requirements.

Def. 5.1. Let $R = \{S_1, \ldots, S_m\}$ be a set of services. A list of atomic processes

(APL[2] for short) $a_1.a_2 \ldots a_n$, where each a_i belongs to a service $S_j \in R$, *satisfies* a given query Q if and only if:

- it satisfies the *control-flow contraints* – the sequence of executions of $a_1, \ldots, a_n$ does not violate the process models of services in R, namely, it violates neither sequencing nor excluding constraints, and
- it satisfies the *data-flow constraints* – the query outputs are a subset of the outputs of $a_1 \ldots a_n$, and the inputs of every $a_i \in APL$ are provided either by the query inputs or by the outputs of $a_1 \ldots a_{i-1}$.

The following proposition establishes the correctness of SAM, namely, that each list of atomic processes returned by SAM respects Definition 5.1.

PROPOSITION 5.2. *Let $APL = a_1.a_2 \ldots a_n$ be a list of atomic processes returned by* SAM *for a given query Q. Then, APL satisfies Q.*

PROOF. (*sketch*) Let APL be a list of atomic processes returned by SAM for Q, and suppose that APL does not satisfy Q. This means that either (1) APL violates the control-flow constraints, or (2.a) at least an input of $a_i \in APL$ is not available, or (2.b) at least an output of Q is not produced by any $a_i \in APL$.

We can easily prove that (2.a) and (2.b) cannot hold. Indeed, if a_i belongs to APL, then a_i was firable, since SAM added it to APL. Hence, all the inputs of a_i either belong to Q or they are produced as output by a process that SAM has previously added to APL. Furthermore, if APL has been successfully returned by SAM, then all the query outputs are red, that is, each of them is produced by at least a process in APL.

Next, we prove that (1) cannot hold, either. Indeed, after applying the Transf function (see Subsection 4.1), the process models of the available services employ `sequence`, `choice`, `split` and `split+join` composite processes, which yield sequencing and excluding constraints only. The APL returned by SAM is either (i) a success (i.e., a simple APL), or (ii) a concatenation of several failures (i.e., of several $APLs$).

We consider first (i). If APL violates the sequencing contraints, then there exists $a_i \in APL$ such that there is no set of predecessors $P \in \mathsf{Prev}(a_i, parent(a_i))$ (see Subsection 4.1) such that $P \subseteq \{a_1, \ldots, a_{i-1}\}$. On the other hand, if SAM added it to APL, then a_i had to be executable. Hence, at least a set of predecessors of a_i had been executed and consequently added to APL before a_i. Similarly, if APL violates the excluding constraints, then there exists $a_i \in APL$ and $a_k \in APL$, such that $k < i$ and $a_k \in \mathsf{FindChoice}(a_i, parent(a_i))$ (see Subsection 4.1). However, if a_k belongs to APL, then a_k had been coloured in red and a_i would have had been coloured in black, since there is an excluding constraint between a_k and a_i. Hence we have a contradiction because a_i is red, since it belongs to APL.

We now consider (ii). Let $F = APL_1.APL_2. \ldots APL_n$ be a concatenation of $APLs$. Since every $APL_k \in F$ corresponds to an independent execution trace of the services in the registry, F violates the process model constraints if there exists at least $APL_i \in F$ such that APL_i violates such constraints. However, this

[2]For the sake of simplicity we assume that all process names in an APL are all distinct, since they actually denote atomic process invocations.

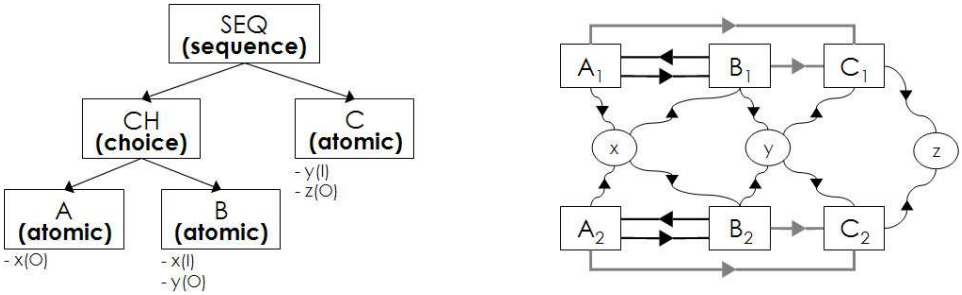

Fig. 13. Example showing process model multiple instances in the dependency hypergraph.

contradicts the previous proof of (i), since APL_i can not violate the process model constraints.

Hence, SAM is correct. □

5.3 Completeness

Consider the example in Figure 13 (left hand-side), the query asking for z as output, as well as the $APL = A.B.C$. Note that the x output of process A is needed as input by process B. Furthermore, C inputs the y output of B, and it produces z. Hence, in order to satisfy the query, one should execute first A, then B, and then C. However, this cannot be done in a single instance of the process model in the Figure, since A and B are mutually excluding processes. The desired output could be produced however by instantiating the process model two times – first for obtaining x through the execution of A, and second for obtaining y from the execution of B. It is important to note that the invoker should act as a sort of "proxy" by getting the x output of A from an instance, and feeding it as input to B in the other instance. Note further that, by assuming a proxy-like invoker for which data is persistent, C could be executed in either of the two instances.

Currently, SAM does not cope with scenarios as the above one, which involves exchanging data across multiple instances of the same service. This is because, during the red colouring phase SAM can only paint in red A, and this leads to painting in black (viz., "burning") B. As a result SAM returns a failure, as it cannot paint in red the output z of C requested by the query.

However, SAM can be slightly modified to cope with such scenarios by duplicating, for each matched service containing a `choice` process, the processes that are added to the hypergraph, together with their sequencing and excluding constraints, and their data dependencies. Informally, the duplicates correspond to instances of the matched process models. In this way all the non-deterministic executions of services containing `choice` processes are introduced in the hypergraph. As explained in Subsection 4.1, the Alt function applied to a process tree node P computes the number of times P has to be executed in order to yield all the concepts producible by its atomic process children. Hence, SAM can duplicate process models P containing `choice` processes "$\mathsf{Alt}(Root(P)) - 1$" times in order to represent explicitly in the hypergraph all the possible distinct alternative executions of P needed to obtain all the outputs of its children atomic processes. This can be simply implemented in the pseudocode of Figure 8, by adding a parameter (i.e., `int` alt) to the

Match function (lines 6, 13, 20) and by invoking Match($Root(s), H,$ Alt(s)) for each service s in the registry (line 4), as well as by invoking the Add function Alt(s) times (line 10, 17).

For our example, Alt(SEQ) gives a value of two, and the hypergraph one obtains is given in Figure 13 (right hand-side). Note that we have used indices for the process names in the hypergraph so as to explicitly show the two instances. Now, during the red colouring phase, both A_1 and A_2 are firable. Assume SAM paints in red A_1. This leads to painting in black B_1. However, SAM does not terminate now with a failure, since B_2 is firable, and by painting it in red the y data node becomes red as well. Consequently, both C_1 and C_2 are firable, and by executing either of them, SAM paints in red the z datum requested by the query.

With this extension, we argue that SAM would be able to generate an APL whenever there exists an APL for a given query. However, this extension decreases further the efficiency of SAM, as the size of the hypergraph linearly increases with Alt. Given the high complexity of SAM, we plan to devote our immediate future efforts to try to improve the algorithm efficiency, rather than sacrificing further efficiency in order to achieve completeness.

6. RELATED WORK

In the Introduction we argued that WSDL does not provide neither semantics nor behavioural information, hence affecting the process of automating both the discovery and composition of Web services. The problem has recently attracted quite some attention, as witnessed by the definition of several new service description languages. Some of them focus on behavioural aspects, such as BPEL [Andrews, T., et al. 2003], WS-CDL [Kavantzas et al. 2004], YAWL [van der Aalst and ter Hofstede 2005], while others centre on semantics features such as WSDL-S [Akkiraju et al. 2005], OWL-S [OWL-S Coalition 2004] (which we briefly described in Section 2), WSMO [J. de Bruijn et al. 2005] and SWSO [Battle, S., et al. 2005]. Given the recent availability of richer service descriptions, many proposals to discover and/or to compose services have been advanced. According to the different kinds of information they exploit, such proposals can be classified in three groups: *sematics-based* approaches, *behaviour-based* approaches as well as *semantics-* and *behaviour-based* approaches.

Paolucci et al. proposed in [Paolucci et al. 2002] the first semantics-based algorithm for Web service discovery. Their algorithm performs an interface-level matching between service requests and service advertisements described as DAML-S service profiles. As already noted in Section 3, the algorithm described in [Paolucci et al. 2002] is however limited to discovering a single service, and it does not address the issue of discovering service compositions.

To overcome the drawbacks of UDDI, Kawamura et al. developed in [Kawamura et al. 2004] a matchmaker that enhances the search functionality of UDDI by enriching WSDL service descriptions with semantic descriptions in the style of DAML-S service profiles. This proposal tries to take advantage of the wide spread use of UDDI, as well as of the semantic power of the ontology descriptions of Web services. However, it suffers from the same limitations of [Paolucci et al. 2002].

Li and Horrocks describe in [Li and Horrocks 2004] an algorithm for matching

DAML-S service descriptions based on a description logic reasoner. The authors mention that an off-line classification of service advertisements (in a DL TBox) could be employed to speeds up the matching process. Still, they note that removing advertisements from the TBox is more difficult. An important limitation of their approach is that they do not address composition-oriented service discovery. Furthermore, we argue that it is quite difficult for the client to express a query matching a service S without prior knowledge about the description of S.

Kifer et al. [Kifer et al. 2004] employ WSMO to define a logical framework based on *Flora-2* for service discovery. The two main stages of locating a service are (1) *discovery* – which finds services that might satisfy the request, and (2) *contracting* which checks whether the request can be actually fulfilled. A feature of their approach is the usage of *wgMediators* for matching client goals with service capabilities. Still, they do not tackle the discovery of service compositions. Furthermore, we argue that client requests may be difficult to be modelled by ordinary clients.

Aversano et al. and Benatallah et al. respectively proposed in [Aversano et al. 2004] and [Benatallah et al. 2003] two approaches which extend [Paolucci et al. 2002] with the discovery of service compositions. The algorithm of Aversano et al. analyses DAML-S service profiles (as [Paolucci et al. 2002]) and, by performing a cross-ontology matching (over service descriptions employing different ontologies), it searches for service compositions capable of satisfying the client request (when no single service can fulfill the request). The algorithm of Benetallah et al. [Benatallah et al. 2003] computes the combinations of Web services that best match a given request by resolving a *best covering problem* in the domain of hypergraphs theory. Each available service becomes a vertex in the hypergraph while each query output O_Q becomes an edge populated by those services that produce an output equivalent to O_Q. The problem of computing the best service combinations can be reduced to the computation of the minimal transversals (i.e., covers) with the minimal cost of the hyperpgraph, where the notion of cost is defined in terms of the missing information of the request with respect to the considered service combination. Both the approaches of Aversano et al. and of Benatallah et al. are based on analysing DAML-S service profiles only, and hence they do not consider the service behaviour. Moreover, whilst Aversano et al. consider the possibility that the missing input of a service (i.e., the inputs not contained in the client request) could be produced as output by other services, Benetallah et al. do not take into account this feature. Comparing SAM with [Aversano et al. 2004] and [Benatallah et al. 2003], one may note that SAM analyses the process model of services in order to perform a finer-grained matchmaking, at the level of atomic processes inside services rather than at the level of entire services. Moreover, when no service composition can satisfy the query, SAM is also capable of suggesting additional inputs that would suffice to get a full match.

The algorithm proposed by Bansal and Vidal in [Bansal and Vidal 2003] belongs to the third category, namely to the semantics- and behaviour-based approaches, as it analyses OWL-S process models. As we already discussed in Section 3, SAM extends [Bansal and Vidal 2003] by considering both compositions and multiple executions of services.

Ben Mokhtar et al. have recently proposed in [Mokhtar et al. 2005] 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 [Hashemian and Mavaddat 2005]. 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 [de Alfaro and Henzinger 2001]. Both the approaches of Ben Mokhtar et al. and of Hashemian and Mavaddat 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 (inside services) which is crucial in order to determine the behaviour of a service composition, e.g., to determine whether it may deadlock or not. Moreover, the approaches of Ben Mokhtar et al. and of Hashemian and Mavaddat assume that the user query is expressed as an automaton or as an interface automaton respectively. Instead,SAMonly requires a query formed by a list of inputs and outputs (belonging to an ontology) of the desired service. We believe that for a common user the former approach is quite complicated.

Miller et al. have recently contributed to the realization of METEOR-S [METEOR-S Team 2004] which is an active research initiative that aims at annotating, discovering and composing semantic Web services. METEOR-S consists of two parts: the front-end [Rajasekaran et al. 2005], which deals with development, annotation and discovery of services, and the back-end [Sivashanmugam et al. 2004], which addresses the Web service composition issue. However, 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, the proposals which belong to the second category (i.e., behaviour-based approaches) intentionally focus on how to aggregate services as well as on how to express service compositions. A common limitation of these approaches is that services useful for the composition have to be manually selected by a service orchestrator. Indeed, semantics information is mandatory in order to automate the process of Web service discovery. It is worth mentioning [Álvares et al. 2005], [Benatallah and Hamadi 2003] and [Kochut and Yi 2004], which model services and their compositions as Petri nets, and [Berardi et al. 2005] which represents service compositions as nondeterministic transition systems.

7. CONCLUDING REMARKS

We have presented a new algorithm — called SAM (for Service Aggregation Matchmaking) — for the composition-oriented discovery of Web services. The version of SAM presented in this paper substantially improves and extends the version presented in the first report [Brogi et al. 2005]. On the one hand, the new version employs hypergraphs (rather than graphs) to better capture the dependencies among processes in the matched services. On the other hand, the new version copes with arbitrary OWL-S service descriptions (while the first version only dealt with OWL-S `sequence` and `choice` constructs).

As already mentioned in the Introduction, the main novel features of SAM are:

(1) to perform a fine-grained matching (at the level of atomic processes of services

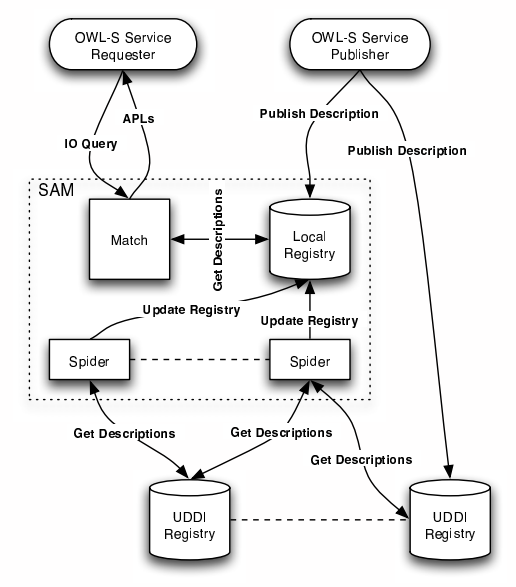

Fig. 14. SAM using a local registry. *APL* stands for *A*tomic *P*rocess *L*ist.

rather than at the level of entire services),

(2) to feature a flexible matching by returning partial matches and by suggesting additional inputs (when some query output cannot be produced by the services in the registry),

(3) to discover service compositions capable of satisfying a query, when no single service can satisfy it. In such cases SAM also explicitly returns the sequence of atomic process invocations that the client must perform in order to achieve the desired result.

It is worth discussing some architectural issues here. The first issue mainly concerns the service registry used by SAM, that is, what service registry SAM uses, and how is it built and updated. We assume that OWL-S service descriptions will be made available by service providers. Two possibilities for deploying SAM are:

(1) A first possibility is for SAM to use a (local) registry consisting of OWL-S descriptions fetched from UDDI registries by means of spiders (or uploaded directly by service providers) as shown in Figure 14. In this scenario, the providers firstly publish OWL-S service descriptions. The requester then queries SAM which matches the request with the service descriptions stored in its (local) registry. It is important to note that OWL-S descriptions consisting both of service profile and service model are to be encoded in the registries.

(2) A second possibility for the deployment of SAM is to embed it in UDDI registries similarly to the approach of Srinivasan et al. [Srinivasan et al. 2004]. Again, a prerequisite of our approach is that SAM should have access to both the service profile and service model information of the OWL-S descriptions. As for [Srinivasan et al. 2004], UDDI registries should provide a special port through

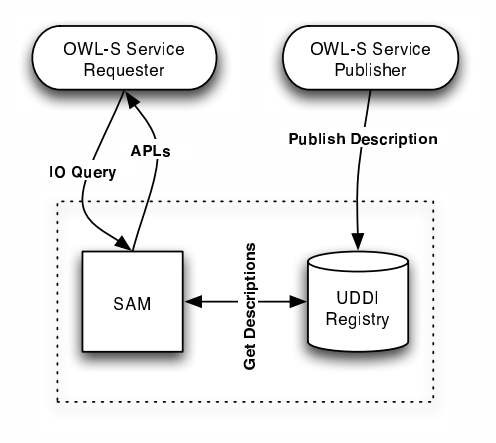

Fig. 15. SAM embedded in UDDI registries.

which SAM can be invoked in order to deal with semantic queries. Figure 15 depicts this scenario.

It is worth noting that in (1) and (2) the discovery takes place on the server-side. The actual composition process is performed on the client-side by employing the atomic process list discovered by SAM.

To conclude, we hereafter present some remarks and comments regarding some of the current issues on ontologies and ontological languages for service descriptions (in general). One general open issue is the adoption of a "de facto" service description language as standard. Service signature and behaviour are tackled on the one hand by the industry which sustains WSDL and respectively BEPL-like (i.e., syntactic) proposals, while on the other hand academic research is more oriented towards ontology-based (i.e., semantic) languages such as WSDL + OWL (e.g., OWL-S Service Profile) or WSDL-S and OWL-S (Service Model) respectively. Currently, WSDL and BPEL-like services are far more spread than semantic ones, yet this is not an indication that such languages are more appropriate for describing Web services. The "popularity" of a proposal does not depend exclusively on its scientific quality as its success on the market also relies on the "forces" pushing it.

A criticism often made to ontologies is that it is not reasonable to imagine everybody using a single ontology for all purposes everywhere. This calls for the need of crossing/relating ontologies whenever we have services described by means of different ontologies. One could however imagine the emergence of "de facto" standard ontologies as well as suitable ontology crossing algorithms – if ontologies will be the way chosen.

Some other concerns on the applicability of ontologies may be summarised as follows:

– how hard is it for the common client to write OWL-S queries?
– how easy is it to under/over/mis- specify a query?
– will the availability of supporting tools be enough to support the penetration of this technology?

Although such issues are far from trivial, we argue that ontologies are a fundamental ingredient of a successful service discovery framework and, should standardised ontologies appear, tools easing the job of the client (e.g., that assist the client to specify queries, that cope with ontology crossing, a.s.o.) will be developed. For example, a under-specified query looking for a service selling flight tickets – Q = {inputs: `destination`; outputs: `flightTicket`} – might be assisted as follows: "Did you mean: Q = {inputs: `destination`, `travelPeriod`; outputs: `flightTicket`}?". Such hints might be based on query history (record tracking) and service ranking.

Our plans for future work include assessing SAM by experimenting it on a large number of queries and service advertisements. While we have tested our Java implementation of SAM on several examples, an obstacle to running massive experiments is the lack of available OWL-S descriptions of services (only a few are publicly available on the W3C Web site). A promising approach to ease the generation of OWL-S descriptions of services may be to publicly deploy (to UDDI registries) supporting tools that facilitate such descriptions, as done for instance by Kawamura et al. [Kawamura et al. 2004] to promote the generation of DAML-S service profiles. Another direction for future work is to extend the matching featured by SAM in order to deal with multiplicity of data (e.g., client may request more than one output having the same data type) as well as with other attributes of services (including non-functional ones) and the use of different ontologies. As for efficiency, we intend to develop a functional analysis module to be pugged-in "before" SAM to restrict the set of candidate services to be considered for a given query. Our long-term goal is to develop a well-founded methodology to support the discovery, aggregation, and —when necessary— adaptation [Bracciali et al. 2005] of services.

ACKNOWLEDGMENTS

This work has been partially supported by the SMEPP project (EU-FP6-IST 0333563).

REFERENCES

Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M.-T., Sheth, A., and Verma, K. 2005. Web Service Semantics – WSDL-S technical note (version 1.0). http://lsdis.cs.uga.edu/library/download/WSDL-S-V1.pdf.

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

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

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

Bansal, S. 2002. *Matchmaking of Web Services Based on the DAML-S Service Model, Master Thesis*. University of South Carolina.

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

Battle, S., et al. 2005. Semantic Web Service Ontology (SWSO), W3C Member Submission, 9 September 2005. http://www.w3.org/Submission/SWSF-SWSO/.

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

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

Berardi, D., Calvanese, D., Giacomo, G. D., and Mecella, M. 2005. Composition of Services with Nondeterministic Observable Behaviour. In *ICSOC 2005, LNCS 3826*, B. Benatallah, F. Casati, and P. Traverso, Eds. Springer-Verlag, 520–526.

Berners-Lee, T., Hendler, J., and Lassila, O. 2001. The Semantic Web. In *Scientific American*.

Bracciali, A., Brogi, A., and Canal, C. 2005. A formal approach to component adaptation. *Journal of Systems and Software 3*, 45–54.

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

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

Gallo, G., Longo, G., Nguyen, S., and Pallottino, S. 1993. Directed hypergraphs and applications. *Discrete Applied Mathematics 42,* 2, 177–201.

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

J. de Bruijn et al. 2005. Web Service Modeling Ontology (WSMO), W3C Member Submission, 3 June 2005. http://www.w3.org/Submission/WSMO/.

Kavantzas, N., Burdett, D., and Ritzinger, G. 2004. Web Service Choreography Description Language Version 1.0, W3C Working Draft, 27 April 2004. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/.

Kawamura, T., Blasio, J. D., Hasegawa, T., Paolucci, M., and Sycara, K. 2004. Public Deployment of Semantic Service Matchmaking with UDDI Business Registry. In *Third International Semantic Web Conference (ISWC'04), LNCS 3298*, S. McIlraith and D. Plexousakis, Eds. Springer-Verlag, 752–766.

Kifer, M., Lara, R., Polleres, A., Zhao, C., Keller, U., Lausen, H., and Fensel, D. 2004. A Logical Framework for Web Service Discovery. In *ISWC 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications*. Vol. 119. CEUR Workshop Proceedings, Hiroshima, Japan.

Kochut, K. J. and Yi, X. 2004. Specification and Analysis of Service Oriented Distributed Systems using Coloured Petri Nets: Models, Algorithms and Tools. In *University of Georgia, Computer Science department - technical report*.

Kritikos, K. 2005. Extending OWL for QoS-based Web Service Description and Discovery. In *IBM Research Report. Proceedings of the IBM PhD Symposium at ICSOC 2005*, A. Hanemann, Ed. 73–78.

Li, L. and Horrocks, I. 2004. A software framework for matchmaking based on semantic web technology. *Int. J. of Electronic Commerce 8,* 4, 39–60.

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

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

Navas-Delgado, I., Sanz, I., Aldana-Montes, J. F., and Berlanga, R. 2005. 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*.

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

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

Papazoglou, M. and Georgakopoulos, D. 2003. Service-oriented computing. *Communications of the ACM 46,* 10, 25–28.

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

Sivashanmugam, K., Miller, J. A., Sheth, A. P., and Verma, K. 2004. 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 (Winter), 71–106.

Srinivasan, N., Paolucci, M., and Sycara, K. P. 2004. An Efficient Algorithm for OWL-S Based Semantic Search in UDDI. In *SWSWPC*. 96–110.

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

van der Aalst, W. M. P. and ter Hofstede, A. H. M. 2005. Yawl: yet another workflow language. *Information Systems 30,* 4, 245–275.

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

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

Yang, J. 2003. Web service componentization. *Communications of the ACM 46,* 10, 35–40.

A. APPENDIX

Hereafter we partially present the OWL-S code of the `Electronics_Store` service, depicted in Figure 2. For each composite process which is part of the service, the process model describes its structure as well as its inputs and outputs. Since the complete process model is quite verbose, we only present the OWL-S code of the root composite process of the `Electronics_Store` service.

```
<process:CompositeProcess rdf:ID="Electronics_Store">

  <process:hasInput>
    <process:Input rdf:ID="country">
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        An_Ontology_URI#country
      </process:parameterType>
    </process:Input>
  </process:hasInput>

  ... other inputs and outputs ...

  <process:composedOf>
    <process:Sequence>
      <process:components>
        <process:ControlConstructList>
          <objList:first>
            <process:Perform rdf:ID="Country_ChoicePerform">
              <process:process rdf:resource="#Country_Choice"/>
              <process:hasDataFrom><process:InputBinding>
                <process:toParam rdf:resource="#Country_Choice_country"/>
                <process:valueSource><process:ValueOf>
                  <process:theVar rdf:resource="#country"/>
                  <process:fromProcess rdf:resource="&process;#TheParentPerform"/>
                </process:ValueOf></process:valueSource>
              </process:InputBinding></process:hasDataFrom>
            </process:Perform>
          </objList:first>
          <objList:rest>
            <process:ControlConstructList>
              <objList:first>
                <process:Perform rdf:ID="LoginPerform">
                  <process:process rdf:resource="#Login"/>
                  <process:hasDataFrom><process:InputBinding>
                    <process:toParam rdf:resource="#Login_username"/>
                    <process:valueSource><process:ValueOf>
                      <process:theVar rdf:resource="#username"/>
                      <process:fromProcess rdf:resource="&process;#TheParentPerform"/>
                    </process:ValueOf></process:valueSource>
                  </process:InputBinding></process:hasDataFrom>

                  ... other inputs ...

                </process:Perform>
              </objList:first>
              <objList:rest>
                <process:ControlConstructList>
                  <objList:first>
                    <process:Perform rdf:ID="Product_ChoicePerform">
                      <process:process rdf:resource="#Product_Choice"/>
                      <process:hasDataFrom><process:InputBinding>
                        <process:toParam rdf:resource="#Product_Choice_camera_Make"/>
                        <process:valueSource><process:ValueOf>
                          <process:theVar rdf:resource="#camera_Make"/>
                          <process:fromProcess rdf:resource="&process;#TheParentPerform"/>
                        </process:ValueOf></process:valueSource>
                      </process:InputBinding></process:hasDataFrom>

                      ... other inputs ...
```

```
                    </process:Perform>
                  </objList:first>
                  <objList:rest rdf:resource="&objList;#nil"/>
                </process:ControlConstructList>
              </objList:rest>
            </process:ControlConstructList>
          </objList:rest>
        </process:ControlConstructList>
      </process:components>
    </process:Sequence>
  </process:composedOf>

  <process:hasResult>
    <process:Result>
      <process:inCondition rdf:resource="&expr;#AlwaysTrue"/>
      <process:withOutput>
        <process:OutputBinding>
          <process:toParam rdf:resource="#Electronics_Store_available_Service"/>
          <process:valueSource>
            <process:ValueOf>
              <process:theVar rdf:resource="#Country_Choice_available_Service"/>
              <process:fromProcess rdf:resource="#Country_ChoicePerform"/>
            </process:ValueOf>
          </process:valueSource>
        </process:OutputBinding>
      </process:withOutput>
    </process:Result>
  </process:hasResult>

  ... other results ...

</process:CompositeProcess>
```

The description of an atomic process is very simple because it only consists of a list of the inputs and the outputs of the process. We present next the OWL-S code for describing the `country_choice` atomic process, which takes as input a country and returns the service availability in that country. We would like to underline that all the concepts used in an OWL-S description (e.g., `country` or `available_service`) have to be previously declared in one or more type ontologies shared by the services in the registry.

```
<process:AtomicProcess rdf:ID="Country_Choice">

  <process:hasInput>
    <process:Input rdf:ID="country">
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        An_Ontology_URI#country
      </process:parameterType>
    </process:Input>
  </process:hasInput>

  <process:hasOutput>
    <process:Output rdf:ID="available_Service">
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        An_Ontology_URI#available_Service
      </process:parameterType>
    </process:Output>
  </process:hasOutput>

</process:AtomicProcess>
```