A Prototype for Discovering Compositions of Semantic Web Services[⋆]

Antonio Brogi[*], Sara Corfini[*], José F. Aldana[†], Ismael Navas[†]
[*]Department of Computer Science – University of Pisa, Italy
Email: {brogi,corfini}@di.unipi.it
[†]Department of Computer Languages and Computing Science – University of Málaga, Spain
Email: {jfam,ismael}@lcc.uma.es

Abstract— **Semantic Web services provide semantically enhanced service descriptions which allow the development of automated service discovery and composition. Yet, an efficient semantic-based service discovery remains a major open challenge towards a wide acceptance of semantic Web services. In this paper we present a fully-automated matchmaking system for discovering sets of OWL-S Web services capable of satisfying a given client request.**

I. Introduction

Web services are internet-based, self-describing and platform-independent application components published using standard interface description languages (WSDL [23]) and universally available via standard communication protocols (SOAP [22]). Web services are emerging as the new fundamental elements for developing complex software applications. However, a prerequisite to reusing and composing Web services is the ability to find the right service(s).

The only accepted standard for service discovery is UDDI [21], which performs a keywords-based search on registries of WSDL services. One of the main limitations of the current service standards is that they do not provide an explicit semantics. Indeed, two identical WSDL documents may describe two completely different services. A promising approach seems to be semantically modelling service capabilities in order to enable a discovery based on what the services really provide. The application of the Semantic Web [4] model to the Web service area has recently promoted the development of so-called Semantic Web Services. A semantic Web service (SWS) is a software component which self-describes its functionalities by annotating them with (instances of) concepts formally defined by means of ontologies. One of the most promising languages for describing SWSs is the Ontology Web Language for Web Services (OWL-S [16]), which provides each service with three documents: the *service profile* ("what the service does"), the *process model* ("how the service works") and the *service grounding* ("how to access the service").

The development of semantics-based service discovery mechanisms constitutes a major open challenge in this context, and it raises several important issues. One of them is the ability of coping with different ontologies, as different services are typically described in terms of different ontologies. Another important feature is the capability of discovering service compositions rather than single services. Indeed it is often the case that a client query cannot be fulfilled by a single service, while it may be fulfilled by a suitable composition of services. Last, but not least, efficiency is obviously an important objective of service discovery mechanisms.

In this perspective, we described in [6] the design of a composition-oriented methodology for discovering OWL-S SWSs. Such methodology employs the notion of Semantic Field [14] to cross different ontologies and it achieves efficiency by pre-computing off-line all the query-independent tasks, viz., to determine the dependencies within/among services as well as the relationships among ontologies. An hypergraph is employed to collect all the computed dependencies. The search algorithm of [6] analyses the hypergraph in order to discover the sets of services (candidated to be composed) capable of satisfying a given client request, which specifies inputs and outputs of the desired service.

In this paper we develop the methodology proposed in [6] by introducing the algorithms needed to pre-process ontologies, to pre-compute service dependencies as well as to discover service compositions. We also present a first prototype of our system which interacts with the clients by means of a suitable Web interface.

The rest of the paper is organised as follows. Sections II and III are devoted to present the discovery system. In Section II we define the data structure we use to collect information about ontology concepts, services and their dependencies, while in Section III we describe the search algorithm for discovering Web service compositions. A first prototype of the discovery system is presented in Section IV. Finally, related work is discussed in Section V, while some concluding remarks are drawn in Section VI.

II. The dependency hypergraph

In order to store the knowledge derived from the pre-processing of ontologies and service descriptions, we need a data structure capable of suitably representing service and data dependencies. The hypergraph seems to be a good candidate as dependencies can be naturally modelled by means of hyperedges.

[⋆]Work partially supported by the SMEPP project (EU-FP6-IST 0333563) and by the project TIN2005-09098-C05-01 (Spanish Ministry of Education and Science).

According to [7], a *directed hypergraph* $H = (V, E)$ is a pair, where V is a finite set of vertices and E is a set of directed hyperedges. A directed hyperedge is an ordered pair (X, Y) of (possible empty) disjoint subsets of V, where X and Y denote the tail and the head of the hyperedge, respectively.

In this context, the vertices of the hypergraph correspond to the concepts defined in the ontologies employed by the analysed service descriptions, while the hyperedges represent relationships among such concepts. More precisely, an hyperedge has one of the following three types:

- $E_{\subset} = (D, \{c\}, nil)$ – *subConceptOf relationship*. Let c be a concept defined in an ontology O and let $D \in O$ be the set of the (direct) subconcepts of c. Then, there is a $E_{\subset}$ hyperedge from D to c.
- $E_{\equiv} = (\{e\}, \{f\}, sim)$ – *equivalentConceptOf relationship*. Let e, f be two concepts defined in two separate ontologies and let sim be the similarity between e and f, i.e., the probability that e is (semantically) equivalent to f. If sim is above a given similarity threshold, there is a $E_{\equiv}$ hyperedge from e to f labelled by sim.
- $E_S = (I, O, s)$ – *intra-service dependency*. Let s be (a profile of) a service and let I be the set of inputs that s requires to produce the set O of outputs. Then, there is a E_S hyperedge from I to O labelled by s.

It is worth noting that the proposed hypergraph automatically models other important aspects regarding the registry-published services and ontologies. The hypergraph models an extended notion of *subConceptOf* between two concepts c, d defined in two given ontologies. More specifically, c is *subConceptOf* d (hereafter, $c \mapsto d$) if and only if there exists a path from c to d which consists of *subConceptOf* relationships ($E_{\subset}$ hyperedges) and/or *equivalentConceptOf* relationships ($E_{\equiv}$ hyperedges). Moreover, the hypergraph directly represents the *inter-service dependencies*. Indeed, there is a inter-service dependency between two services s and t if the head of a s-labelled E_S hyperedge and the tail of a t-labelled E_S hyperedge share (at least) a concept. Note that there is a inter-service dependency between s and t also if there exist two concepts c_s and c_t such that $c_s \mapsto c_t$, c_s belongs to the head of the s-labelled E_S hyperedge and c_t belongs to the tail of the t-labelled E_S hyperedge.

In the following subsections we describe how *subConceptOf* relationships, *equivalentConceptOf* relationships and *intra-service* dependencies can be determined.

A. *The* SemFiT *application*

Semantic Fields [14], [1] are groups of interrelated ontologies that may be relevant to a given information request, in which the client needs to find related ontologies, from a set of ontologies, for a given set of concepts of a specific ontology. As searching for relevant concepts from a large number of ontologies is a time-consuming task, one goal of Semantic Fields is to reduce the number of candidate ontologies to offer the user a good solution within a reasonable period of time.

In order to build Semantic Fields, namely to find relationships between ontologies, we firstly compute the similarity between pairs of concepts and next we calculate the distance between pairs of ontologies (i.e., how similar two ontologies are). Mappings between concepts can be determined by means of different ontology matching tools, which check whether two concepts are equivalent with a given probability.

Since the results depend on the employed matching tools, we have hence developed a framework for adapting existent matching tools and combine them in order to improve the obtained results. In the framework instance used in this paper we combine the results of individual matchers which analyses the similarity between pairs of concepts with different strategies: (1) *name similarity*: compares the names of the provided concepts, (2) *data type similarity*: compares the types of the provided concepts, and establishes a similarity value depending on the possibility of producing each type from another type by casting, (3) *WordNet similarity*: checks whether the two concepts are synonyms in WordNet [18], (4) *path similarity*: compares the path from each concept to its root in the ontology, and provides an average between the path length similarity and the name similarity between all concepts in both paths, (5) *property similarity*: compares the similarities among the properties of the concepts compared, and (6) *edit distance similarities*: this family of algorithms is based on the comparison of strings, and calculates the difference between them as the operations that should be performed on one string to obtain another. As these algorithms calculate the distance between two concepts, it is necessary to calculate the similarity from the value provided.

The calculus of mappings is done between pairs of ontologies, hence if we have N ontologies then we have to calculate $N^2 - N$ matrixes. Assuming that the mapping between two ontologies is a symmetric relationship we need to calculate $(N^2 - N)/2$ matrixes. Yet, our tool calculates the similarity matrixes between the new ontology and all the previously published ontologies when the new one is registered. Thus, each insertion only requires $N - 1$ calculations.

Once the mappings between two different ontologies are established, the distance between the ontologies is calculated by making use of some ontology distance measurements. Given a matrix with the similarity between pairs of concepts of two ontologies (O_1 and O_2), we can make use of several measurements based on the definitions of distance in different contexts. First, we define the directed distance (DD) between them (as it is an asymmetric measure):

$$DD(O_1, O_2) = \frac{\#Concepts(O_1)}{\sum_{c \in Concepts(O_1)} max(mappings(c, O_2))}$$

$$D(O_1, O_2) = min(DD(O_1, O_2), DD(O_2, O_1))$$

We can calculate the distance (which is a symmetric measure) between two ontologies (D) in different ways (see for example the second formula) depending on the distance concept used. These measures allow us to establish how similar two ontologies are, and in this way to know whether it will be necessary to include the already calculated similarities

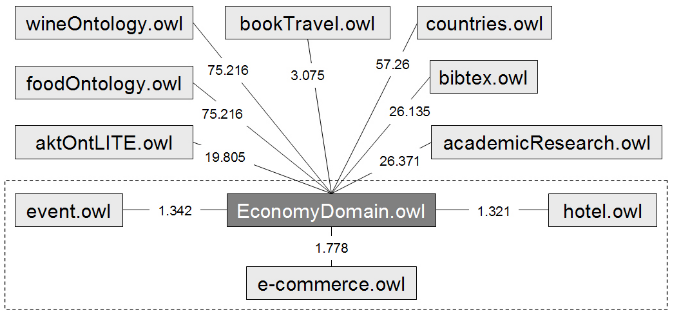


Fig. 1. Semantic Field Calculation.

between their concepts to solve the problem of determining the user Semantic Field.

Let us suppose that we have the set of registered ontologies in Figure 1, in which the edge values mean the already calculated distance between pairs of ontologies. Let us assume that the user selects the `EconomyDomain` ontology, which covers his intended domain, and sets up a distance threshold of 2. In this example the Semantic Field will be composed of the ontologies at a distance less than this value, which are those included in the dotted rectangle of Figure 1, i.e., `event`, `hotel` and `e-commerce`.

Semantic Fields have been implemented as a configurable tool[1] (SemFiT) which can be used directly through a Web Service or Web forms for registered users. The Semantic Field Tool suitably configured to be employed by the matchmaking algorithm provides the following methods:

- startSession()
  returns an identifier for the user session, which will be necessary in other methods;
- insertOntology(`string` $ontologyURI$, `int` $sessionId$)
  inserts a new ontology (if not already present) in the user session. This process includes the calculation of the mappings and distances among the inserted ontology and all the ontologies previously inserted. This calculation implies the use of the framework for ontology matching, and creating a set of similarity matrixes which contain the similarity between concepts of pairs of ontologies (a value between 0 and 1 indicating the probability of two concepts of being the same semantic concept);
- getOntologies(`int` $sessionId$)
  returns the ontologies which belong to the user session;
- searchEquivalentConcepts(`string` $concept$, `string` $ontologyURI$, `float` $minSimilarity$, `int` $sessionId$)
  taking into account the user Semantic Field (the ontologies in which he/she is interested in) this method analyses the similarity matrixes (between pairs of ontologies in his/her Semantic Field) for finding relationships between concepts. Concepts with a similarity value greater that *minSimilarity* are considered equivalent concepts;
- searchParentsForConcept(`string` $ontologyURI$, `string` $concept$, `int` $sessionId$)
  searches in the ontology hierarchy for parents of the given concept, returning a set of vectors with concept, ontology and similarity value. Since the search of hierarchical relationships is a time consuming task, the hierarchy is precalculated from the OWL file by means of the Jena[2] parser and stored in an internal database of SemFiT;
- searchChildrenForConcept(`string` $ontologyURI$, `string` $concept$, `int` $sessionId$)
  searches in the ontology hierarchy for children of the given concept, returning a set of vectors with concept, ontology and similarity value. This method makes use of the pre-calculated hierarchy as well.

B. Computing intra-service dependencies

As described above, an intra-service dependency is a hyperedge connecting two sets of nodes, respectively representing the inputs and the outputs of a service. Yet, a service may behave in different ways and features different functionalities. Consider, for instance, a simple service s which inputs a and produces as output b or c (i.e., a choice process). s can behave in two different ways, namely, as a service s_1 which inputs a and yields b and as a service s_2 which inputs a and yields c. Hence s exposes two different *profiles*, which correspond to the different intra-service dependencies $(\{a\},\{b\},s_1)$ and $(\{a\},\{c\},s_2)$.

The intra-service dependencies of an OWL-S described service can be determined by analysing its process model, which details the complete behaviour of the service. We present next the recursive function DataDep, initially invoked over the root composite process of a service s, which determines all the intra-service dependencies (I,O) of s.

- $\mathsf{DataDep}(\texttt{atomic}(A)) = \{(A.\text{inputs}, A.\text{outputs})\}$;
- $\mathsf{DataDep}(\texttt{ifThenElse}(P_1,P_2))$
  $= \{(I_1,O_1)|(I_1,O_1) \in \mathsf{DataDep}(P_1)\} \cup \{(I_2,O_2)|(I_2,O_2) \in \mathsf{DataDep}(P_2)\}$
- $\mathsf{DataDep}(\texttt{choice}(P_1,...,P_k))$
  $= \bigcup_{i=1}^{k} \{(I,O)|(I,O) \in \mathsf{DataDep}(P_i)\}$
- $\mathsf{DataDep}(\texttt{sequence}(P_1,...,P_k))$
  $= \mathsf{DataDep}(\texttt{split}(P_1,...,P_k))$
  $= \mathsf{DataDep}(\texttt{split+join}(P_1,...,P_k))$
  $= \mathsf{DataDep}(\texttt{any-order}(P_1,...,P_k))$
  $= \{(\bigcup_{i=1}^{k} I_i, \bigcup_{i=1}^{k} O_i)|(I_i,O_i) \in \mathsf{DataDep}(P_i)\}$
- $\mathsf{DataDep}(\texttt{iterate}(P))$
  $= \mathsf{DataDep}(\texttt{choice}(\underbrace{P, \texttt{sequence}(P,P), ..., \texttt{sequence}(P,...,P)}_{\mathsf{Alt}(P)}))$

Before defining Alt(P), we need to specify some assumptions. The proposed algorithm for discovering semantic Web services works on ontology types. As a consequence, the algorithm checks whether an output (i.e., a type) can be produced by some available service, while it does not take care of how many times such output can be produced. This assumption allows us to expand the `iterate` process into a choice of sequences, where each sequence is a possible iterated execution of P. Alt(P), which is defined below, computes how

[1]available at `http://khaos.uma.es/semanticfields/`

[2]`http://jena.sourceforge.net/`

many times the process P has to be executed in order to yield all the outputs producible by its atomic process children.

- Alt(`atomic`(A)) = 1
- Alt(`ifThenElse`(P_1, P_2)) = Alt(P_1) + Alt(P_2)
- Alt(`choice`($P_1, ..., P_n$)) = Alt(P_1) + ... + Alt(P_n)
- Alt(`sequence`($P_1, ..., P_n$)) = Alt(`any-order`($P_1, ..., P_n$)) = Alt(`split`($P_1, ..., P_n$)) = Alt(`split+join`($P_1, ..., P_n$)) = max(Alt(P_1), ..., Alt(P_n))
- Alt(`iterate`(P)) = 0

Let us consider, for example, the HotelService, whose root is a `repeat-until` composite process, illustrated in Figure 2. Alt(`chooseOperation`) returns two, indeed, `chooseOperation` has to be executed twice in order to yield all the outputs producible by its children. Hence, DataDep(`HotelService`) returns three profiles: H_1, which inputs {`state`, `city`} and produces {`infoHotel`}, H_2, which inputs {`hotel`, `beginDate`, `lastDate`, `creditCard`, `contactInformation`} and produces {`accomodationFee`, `hotelInvoice`}, and H_3, which inputs {`state`, `city`, `hotel`, `beginDate`, `lastDate`, `creditCard`, `contactInformation`} and produces {`infoHotel`, `accomodationFee`, `hotelInvoice`}.

Note that DataDep is a recursive function and that if an `iterate` process has an `iterate`-typed ancestor Q, DataDep first expands Q and then P. That is why we set Alt(`iterate`(P)) = 0 in the pseudo-code above. It is also worth observing that the definition of DataDep(`iterate`(P)) is suitable when P has to be executed at least one (i.e., `iterate` instantiated as `repeat-until`) as well as when P may be skipped (i.e., `iterate` instantiated as `repeat-while`). In the latter case, the definition of DataDep(`iterate`(P)) has to be expanded by adding a *nop* branch to the choice among all possible iterate executions of P.

C. Building the hypergraph

After describing how data and service dependencies are determined, we can introduce in this subsection the AddService function, which updates the hypergraph whenever a new service s is added to the registry.

Generally speaking, AddService firstly adds to the hypergraph the concepts defined in the ontologies employed by s and next, it draws the hyperedges representing the *subConceptOf* relationships, the *equivalentConceptOf* relationships and the *intra-service* dependencies between the newly added ontology concepts. Note that AddService does not affect the efficiency of the matching process, as the hypergraph construction is completely query independent and can be pre-computed off-line before query time.

The behaviour of AddService can be summarised by the following pseudo-code, where $sessionID$ is the session number assigned to AddService by the startSession method of SemFiT.

```
1.  AddService(hypergraph $H = (V, E)$, service $s$, int $sessionID$)
2.      forall ontology $O \notin$ getOntologies($sessionID$) referred by $s$ do
3.          insertOntology($O, sessionID$);
4.          forall concept $c$ in $O$ do
5.              Add $c$ to $V$;
6.              forall concept $a$ in searchParentsForConcept($O, c, sessionID$) do
7.                  if $a \in V$ then
8.                      if $\exists(D, \{a\}, nil) \in E_\subset$ then $D = D \cup \{c\}$;
9.                      else Add $(\{c\}, \{a\}, nil) \in E_\subset$ to $E$;
10.             forall concept $b$ in searchChildrenForConcept($O, c, sessionID$) do
11.                 if $b \in V$ then
12.                     if $\exists(D, \{c\}, nil) \in E_\subset$ then $D = D \cup \{b\}$;
13.                     else Add $(\{b\}, \{c\}, nil) \in E_\subset$ to $E$;
14.             forall $e, similarity$ in searchEquivalentsForConcept($c, O,$
15.                                                 $minSimilarity, sessionID$) do
16.                 if $e \in V$ then
17.                     Add $(\{c\}, \{e\}, similarity) \in E_\equiv$ to $E$;
18.                     Add $(\{e\}, \{c\}, similarity) \in E_\equiv$ to $E$;
19.     forall pairs $(I_n, O_n)$ in DataDep(Root($s$)) do
20.         Add($I_n, O_n, s_n$) $\in E_S$ to $E$;
```

For each ontology O employed by s (line 2), the AddService inserts O (if not already present) into the Semantic Field identified by $sessionID$ (line 3). Next, for each concept $c \in O$ (line 4), it adds c to the hypergraph (line 5) and determines the (possibly empty) sets of the parents and children of c by invoking the searchParentsForConcept and searchChildrenForConcept methods of the SemFiT. Then, for each parent a of c (line 6), AddService adds (if $a \in V$) a $E_\subset$ hyperedge (c, a, nil) to E (lines 7–9), while for each child concept b of c (line 10), it inserts (if $b \in V$) a $E_\subset$ hyperedge (b, c, nil) to E (lines 11–13). Next, it determines the (possible empty) set of the equivalent concepts of c by invoking the searchEquivalentsForConcept method of SemFiT, which returns those concepts above a given similarity threshold, i.e., the *minSimilarity* parameter. For each equivalent concept e of c (lines 14–15) AddService adds to E (if $e \in V$) the two $E_\equiv$ hyperedges $(c, e, similarity)$ and $(e, c, similarity)$ (lines 16–18). Note that all the concepts in the ontologies employed by s will be included in the hypergraph, independently of whether they directly occur in the specification of (some process in) s. Finally, AddService inserts to the hypergraph a E_S hyperedge (I_n, O_n, s_n) for each intra-service dependency returned by DataDep (namely, for each profile n of s) (lines 19–20).

D. Example

We present next an example which illustrates the behaviour of the AddService function. Let us consider an empty registry where we want to add two OWL-S described services: HotelService, which allows a client to search for and/or to reserve hotels, and ConferenceService, which allows a client to register to academic events. Their process models, which employ three different ontologies (`hotel`, `e-commerce` and `event`), are shown in Figure 2.

Let us consider first the HotelService. AddService inserts to the hypergraph all the concepts defined in the `hotel` and `e-commerce` ontologies (the light gray and dark gray ellipses in Figure 3) together with the *subConceptOf* and the *equivalentConceptOf* relationships returned by SemFiT. Note, for example, the *subConceptOf* relationships between `hotel#city` and `hotel#physicalLocation`, and between `e-commerce#creditCard` and `e-commerce#paymentMethods` as well as the *equivalentConceptOf* relationships between `e-commerce#invoice` and `hotel#invoice`. At this point, the AddService inserts also the *intra-service* dependencies of HotelService computed by

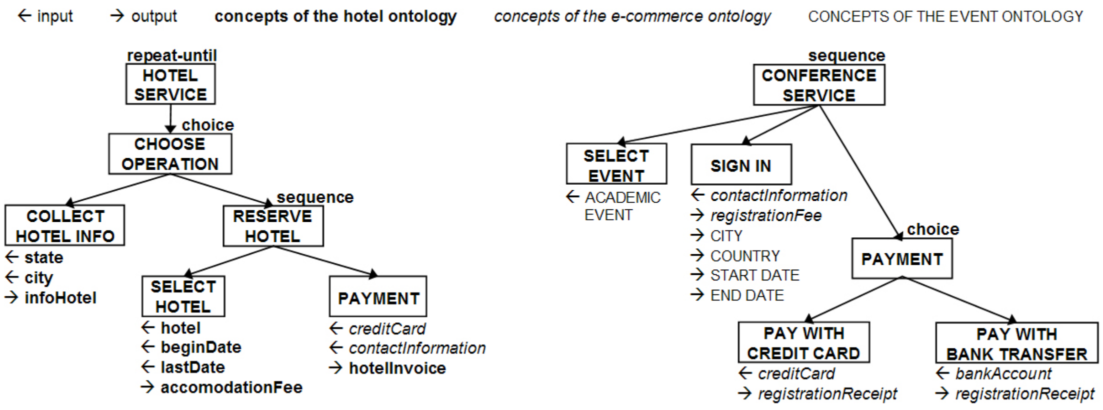


Fig. 2. Process models of the HotelService and ConferenceService.

the DataDep function, i.e., the E_S hyperedges H_1, H_2 and H_3 in Figure 3. Next, also ConferenceService is added to the registry. AddService inserts to the hypergraph the concepts defined in the `event` ontology, the *subConceptOf* and *equivalentConceptOf* relationships computed by SemFiT, as well as the intra-service dependencies of ConferenceService. The resulting hypergraph is shown in Figure 3.

It is worth observing that some of the *equivalentConceptOf* relationships found by SemFiT appear due to the similarity between the syntax of the compared concepts (`e-commerce#invoice` – `hotel#invoice`, `hotel#physicalLocation` – `event#location` and `hotel#city` – `event#city`), while other mappings are the result of applying searches in wordNet (e.g., `event#startDate` – `hotel#beginDate` and `event#endDate` – `hotel#lastDate`). In this last case, the similarity does not derive from direct synonyms in wordNet, but from the tokens that compose the words. Finally, `event#dateTime` and `hotel#date` are similar because of their syntax and their structure (both have children that have same similarity).

III. THE DISCOVERY ALGORITHM

The discovery algorithm takes as input a client query specifying the set of inputs and outputs of the desired service (composition). As we will describe in Section IV, the formulation of the query is eased by a suitable interface that displays the available concepts with an expandable tree structure. Next, the search algorithm explores the hypergraph, by performing a depth-first visit, in order to discover the (compositions of) services capable of satisfying the client request.

The discovery algorithm is synthesised by the following recursive function QuerySolver which inputs five parameters: the hypergraph H, the client query Q, the set $composition$ of the services selected so far (initially empty), the set $neededOutput$ of the outputs to be generated (initially the query outputs), and the set $availableOutput$ of the outputs available (initially the query inputs).

In the pseudo-code listed below, Input(s) denotes the inputs of the service s, that is, the set $\{i|\exists(I,O,s) \in E_S \wedge i \in I\}$ as well as Output(s) denotes the outputs of s, namely $\{o|\exists(I,O,s) \in E_S \wedge o \in O\}$. We also remind you that $\mapsto$ denotes the extended *subConceptOf* relationship.

```
1.  QuerySolver(hypergraph $H$, query $Q$, set $composition$,
2.                        set $neededOutput$, set $availableOutput$)
3.    if ($neededOutput = \emptyset$) then return $composition$
4.    else
5.      $out$ = extract($neededOutput$);
6.      $S = \{s|out \in$ Output$(s) \vee (\exists c \in$ Output$(s)|c \mapsto out)\}$;
7.      if ($S = \emptyset$) then fail
8.      else
9.        forall service $s$ in $S$ do
10.         $composition'$ = $composition \cup \{s\}$;
11.         $availableOutput'$ = $availableOutput \cup$ Output$(s)$;
12.         $neededOutput'$ = $\{x|x \in neededOutput \cup$ Input$(s)$
13.                              $\wedge \nexists a \in availableOutput'$ :
14.                              $a = x \vee a \mapsto x\}$;
15.         QuerySolver($H$, $Q$, $composition'$,
16.                     $neededOutput'$, $availableOutput'$);
```

If $neededOutput = \emptyset$, i.e., there are no outputs to be generated, QuerySolver returns the set $composition$ of the services found (line 3), which satisfies the functional client query. Otherwise (line 4) QuerySolver withdraws an output out from the set $neededOutput$ (line 5) and computes the set S of the services which produce (a sub concept of) out (line 6). If S is empty, that is, out cannot be generated by any registry-published service, then QuerySolver fails (line 7) since the query cannot be fulfilled. Otherwise (line 8) for each service s which generates (a sub concept of) out (line 9), QuerySolver adds s to $composition$ (line 10), updates $availableOutput$ by adding the outputs of s (line 11), and updates $neededOutput$ by adding the inputs of s and by removing the concepts that are now available (lines 12–14). Next, QuerySolver continues recursively (lines 15–16).

The complexity of QuerySolver belongs to the NP space as it determines all the possible solutions by visiting the hypergraph non-deterministically. We have tested QuerySolver on a small repository containing ten services only, as there are few available SWSs on the Web as the SWS area is emerging now. Anyway, the average reply-time of QuerySolver is 0.2 seconds. We imagine that in the future many SWSs will exist, thus, we intend to improve efficiency by operating in the following directions:

- to reduce the large number of services taken into account by the QuerySolver by introducing a suitable service

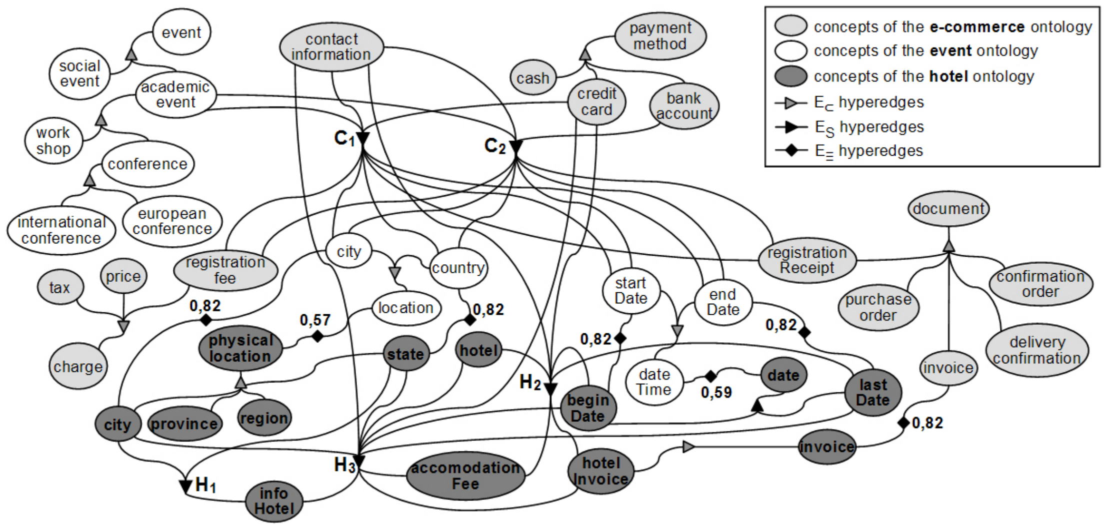


Fig. 3. The hypergraph after registering HotelService and ConferenceService.

pre-selection phase (e.g., using UDDI to filter services not belonging to certain service categories), and by selecting services which produce a needed output with respect to some heuristics (e.g., the number and/or on the quality of the produced outputs) in order to consider the "most promising" services only.

- to introduce caching and indexing techniques in order to sensibly decrease the QuerySolver reply-time.

A. *Example*

Let us continue the example introduced in subsection II-D. Consider now a client wishing to plan its participation in an international conference by registering to the conference and by booking his hotel accomodation, and receiving the conference and hotel confirmations. The client query may be composed by the following parameters:

- **inputs** – `hotel#hotel`, `event#internationalConference`, `e-commerce#creditCard`, `e-commerce#contactInformation`
- **output** – `e-commerce#invoice`, `e-commerce#registrationReceipt`.

As one may note, neither HotelService nor ConferenceService satisfies the given query by itself. Yet, the query can be fulfilled by suitably composing the two available services. QuerySolver takes as parameters the hypergraph depicted in Figure 3, the query inputs (i.e., $availableOutput$) and the query outputs (i.e., $neededOutput$), while $composition$ is an empty set. Suppose that QuerySolver withdraws `e-commerce#invoice` from $neededOutput$. QuerySolver creates two compositions, represented by the hyperedges $\{H_2\}$ and $\{H_3\}$ respectively, as the profiles $\{H_2\}$ and $\{H_3\}$ of HotelService both produce `hotel#hotelInvoice`, which is a *subConceptOf* `e-commerce#invoice` (i.e., the former is linked to the latter by means of $E_\subset$ and $E_\equiv$ hyperedges). Consider composition $\{H_2\}$. QuerySolver adds to $neededOutput$ `hotel#beginDate` and `hotel#lastDate`, since they do not belong to the query inputs. Suppose that, at its second invocation, QuerySolver withdraws `e-commerce#registrationReceipt` from $neededOutput$, which is produced by both the profiles C_1 and C_2 of ConferenceService. QuerySolver creates then the following compositions: $\{H_2, C_1\}$ and $\{H_2, C_2\}$. While the first one fulfills the query, as all the needed outputs are now available, the second one fails, as `e-commerce#bankAccount` cannot be produced by any registry-published service. When all instances of QuerySolver terminate, the functional analyser returns to the client two successful compositions: $\{H_2, C_1\}$ and $\{H_3, C_1\}$.

IV. IMPLEMENTATION

In this section we discuss the implementation of our discovery system, that we named SAM for Service Aggregation Matchmaking.

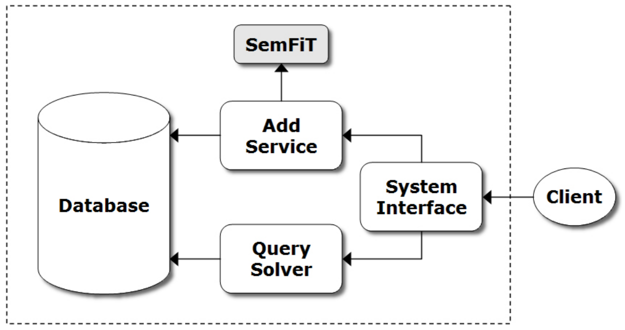


Fig. 4. The system architecture.

Figure 4 presents the high-level architecture of SAM. *AddService* and *QuerySolver* are Java applications which implement the AddService function (Subsection II-C) and the discovery algorithm (Section III). *AddService* makes use of the CMU OWL-S API[3] for parsing OWL-S descriptions.

[3]http://www.daml.ri.cmu.edu/owlsapi/

AddService and *QuerySolver* both access the *database*, which is the local repository where SAM stores service descriptions, ontologies and the dependency hypergraph (Section II). A JDBC-ODBC bridge allows the Java applications to access the repository, implemented as a relational (MS Access, for this first prototype) database. It is worth noting that we do not introduce Java classes to represent ontologies, profiles, concepts and dependencies in order to avoid loading in memory space-consuming Java objects. Indeed, *AddService* and *QuerySolver* perform ontological reasoning and hypergraph handling by means of database invocations. As described in Subsections II-A and II-C, the Java Semantic Field Application (i.e., SemFiT) supports *AddService* to process ontologies and data dependencies.

Finally, we have provided SAM with a suitable Web interface, which allows clients to submit a new service to the system as well as to query the discovery algorithm. Figure 5 shows a screenshot of the system interface. In its left panel the interface provides a tree view of the available ontologies to help the client in specifying its query. The input and output concepts selected by the client are shown in the query inputs and query outputs panels on the right part of the interface. By clicking on the *Submit query* button, the client queries the discovery algorithm which returns the result in the bottom panel of the interface. Moreover, the client can submit a new service to the system by means of the *Submit service* button, after providing the service URL.

AddService and *QuerySolver* are Java servlets which run on Apache Tomcat. To develop the Web interface we employed the Google Web Toolkit, which is a Java development framework that allows to deploy AJAX applications and complex Web interfaces by using the Java language. Google Web Toolkit then translates the Java code to browser-compliant JavaScript and HTML.

Figure 5 catches the Web interface as it appears after submitting the query described in Subsection III-A. Indeed, the results panel lists the two successful compositions which satisfy the example query. Note the result syntax $[S_1(P_{1_1}, \ldots, P_{1_{n1}}), \ldots, S_n(P_{n_1}, \ldots, P_{n_{nn}})]$ where S_i denotes the service name and $P_{i_1}, \ldots, P_{i_{ni}}$ denote the atomic processes of the selected profile of S_i.

V. Related work

The discovery of semantic Web services has been firstly addressed by Paolucci et al. in [17], where they proposed an algorithm performing a functionality matching between service requests and service advertisements described as DAML-S (the predecessor of OWL-S) service profiles. Yet, [17] does not deal with different ontologies, nor it considers service composition, which instead is the goal of [3], [12], [8]. These approaches discover compositions of OWL-S described services by operating in the domain of hypergraphs, finite state automata and interface automata, respectively. However, [3], [12], [8] do not address the task of crossing ontologies.

In [2] Aversano et al. extended [17] by proposing a composition-oriented discovery algorithm capable of coping with different ontologies. Still, the ontology crossing task is computed at query answering time, hence severely affecting the efficiency of the discovery algorithm. An efficient matching of semantic Web service capabilities is the goal of [13], which achieves efficiency by pre-processing the available ontologies and by pre-classifying the registry-published services before query time. Still, [13] does not address the discovery of service compositions.

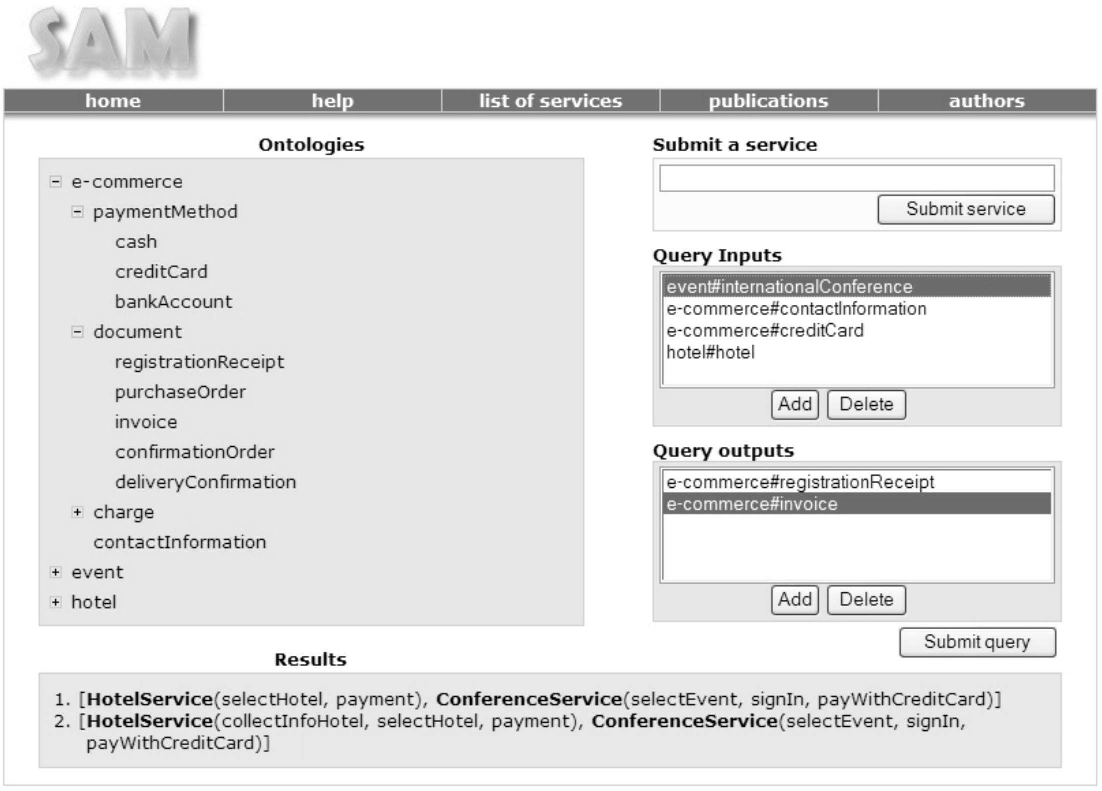
Moreover, it is worth mentioning the METEOR-S project [19], whose objective is the realisation of a framework for annotating, discovering and composing semantic Web services. Yet, METEOR-S is a semi-automated approach requiring a strong participation of the user, which is highly involved in the process of semi-manually discovering and/or composing services.

As composing services requires to cope with different ontologies, several other works have been recently proposed to address the issues of ontology merging and alignment. For instance, it is worth mentioning [11], [15], which exploit the knowledge provided by matching tools that find similarities between concepts interpreted as lexical entries, [10], which makes use of classes labels to compare the taxonomies of two ontologies, and [20], which does not analyse the classes properties, but takes into account the ontology instances to related different classes. In order to achieve better results, SemFiT developed a framework for ontology crossing which takes advantages of different matching algorithms, as the combination of simple matchers has demonstrated its viability in several approaches such as the COMA Framework [9].

VI. Concluding remarks

In this paper, we have presented a new fully-automated matchmaking system for discovering (compositions of) OWL-S semantic Web services capable of satisfying a given client request. The proposed system is the first one – at the best of our knowledge – that addresses three main issues of the semantic service discovery domain, namely, composition-oriented discovery, crossing different ontologies and efficiency. We have also presented a first Web-accessible prototype of our discovery system.

Our plan for future work includes the development of fresh indexing and/or ranking techniques (as search engines do for Web pages) in order to sensibly improve the efficiency of our system. A second line of our future work is to enhance our discovery system by employing a behavioural analyser module, which analyses the behavioural information advertised in the (OWL-S) service descriptions to determine whether the candidate services (selected by the functional analyser) can really be composed together and satisfy the query without dead-locking, in the line of [5]. Finally, we plan to extend SemFiT by employing other existing matching tools to achieve better results for the ontology matching problem. Our long-term goal is to develop a well-founded methodology to support an efficient and fully-automated discovery and composition of Web services.



e-commerce
paymentMethod
cash
creditCard
bankAccount
document
registrationReceipt
purchaseOrder
invoice
confirmationOrder
deliveryConfirmation
charge
contactInformation
event
hotel

event#internationalConference
e-commerce#contactInformation
e-commerce#creditCard
hotel#hotel

e-commerce#registrationReceipt
e-commerce#invoice

1. [**HotelService**(selectHotel, payment), **ConferenceService**(selectEvent, signIn, payWithCreditCard)]
2. [**HotelService**(collectInfoHotel, selectHotel, payment), **ConferenceService**(selectEvent, signIn, payWithCreditCard)]

Fig. 5. The Web interface of SAM.

References

[1] J. F. Aldana-Montes, I. Navas-Delgado, and M. del Mar Roldan-Garcia, “Solving Queries over Semantically Integrated Biological Data Sources,” in *Int. Conf. on Web-Age Information Management. LNCS* 3129, 2004.

[2] L. Aversano, G. Canfora, and A. Ciampi, “An Algorithm for Web Service Discovery through Their Composition,” in *IEEE International Conference on Web Services (ICWS’04)*, L. Zhang, Ed. IEEE Computer Society, 2004, pp. 332–341.

[3] B. Benatallah, M.-S. Hacid, C. Rey, and F. Toumani, “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, 2003, pp. 242–257.

[4] T. Berners-Lee, J. Hendler, and O. Lassila, “The Semantic Web,” in *Scientific American*, 2001.

[5] A. Brogi and S. Corfini, “Behaviour-aware discovery of Web service compositions,” in *University of Pisa, Department of Computer Science - Tech. Rep. TR-06-08*, June 2006.

[6] A. Brogi, S. Corfini, J. Aldana, and I. Navas, “Automated Discovery of Compositions of Services Described with Separate Ontologies,” in *ICSOC 2006.* LNCS *4294*, A. Dan and W. Lamersdorf, Eds. Springer-Verlag, 2006, pp. 509–514.

[7] G. Gallo, G. Longo, S. Nguyen, and S. Pallottino, “Directed hypergraphs and applications,” *Discrete Applied Mathematics*, vol. 42, no. 2, pp. 177–201, 1993.

[8] S. Hashemian and F. Mavaddat, “A Graph-Based Approach to Web Services Composition,” in *SAINT 2005*, IEEE Computer Society, Ed. CS Press, 2005, pp. 183–189.

[9] D. Hong-Hai and R. Herhard, “COMA - A System for Flexible Combination of Schema Matching Approches,” in *Proc. of the* 28^{th} *Int. Conf. on Very Large Databases (VLDB)*, Hongkong, 2002.

[10] A. Maedche and S. Staab, “Measuring Similarity between Ontologies,” in *Proc. Of the European Conference on Knowledge Acquisition and Management - EKAW-2002, LNCS/LNAI* 2473, 2002, pp. 251–263.

[11] D. McGuinness, R. Fikes, J. Rice, and Wilder, “The Chimaera Ontology Environment,” in *Proc. of the 17th Nat. Conf. on Artificial Intelligence*, 2000, pp. 1123–1124.

[12] S. B. Mokhtar, N. Georgantas, and V. Issarny, “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, 2005.

[13] S. B. Mokhtar, A. Kaul, N. Georgantas, and V. Issarny, “Towards Efficient Matching of Semantic Web Service Capabilities,” in *Proceedings of WS-MATE 2006*, 2006.

[14] I. Navas-Delgado, I. Sanz, J. F. Aldana-Montes, and R. Berlanga, “Automatic Generation of Semantic Fields for Resource Discovery in the Semantic Web,” in *16th Int. Conf. on Database and Expert Systems Applications. LNCS* 3588, 2005.

[15] N. Noy and M. Musen, “Prompt: Algorithm and Tool for Automated Ontology Merging and Alignment,” in *Proc. of the Nat. Conf. on Artificial Intelligence*, 2000.

[16] OWL-S Coalition, “OWL-S 1.1,” 2004, http://www.daml.org/services/owl-s/1.1/.

[17] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara, “Semantic Matchmaking of Web Services Capabilities,” in *1th Int. Conf. on The Semantic Web, LNCS 2342*, I. Horrocks and J. Hendler, Eds. Springer-Verlag, 2002, pp. 333–347.

[18] R. Al-Halimi et al., *WordNet - An Electronic Lexical Database*, C. Fellbaum, Ed. MIT Press, 1998.

[19] P. Rajasekaran, J. A. Miller, K. Verma, and A. P. Sheth, “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, 2005, pp. 55–68.

[20] G. Stumme and A. Madche, “Fca-merge: Bottom-up Merging of Ontologies,” in *In 7th Intl. Conf. on Artificial Intelligence (IJCAI ’01)*, 2001, pp. 225–230.

[21] W3C, “The UDDI Technical White Paper,” 2000, http://www.uddi.org/.

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

[23] ——, “Web Service Description Language (WSDL) 1.1,” 2001, http://www.w3.org/TR/wsdl.