A Framework for Web Services-Based Query Rewriting and Resolution in Loosely Coupled Information Systems

Mahmoud Barhamgi, Pierre-Antoine Champin, and Djamal Benslimane

LIRIS Laboratory, Claude Bernard Lyon1 University
69622 Villeurbanne, France
{mahmoud.barhamgi,pierre-antoine.champin,djamal.benslimane

1 Introduction

In today's loosely-coupled e-collaboration environments (e.g. eHealth, eGov...etc) the access to an increasing number of data sources is made through Web services. Indeed, this is motivated by the need to access and retrieve data items held by autonomous heterogeneous parties involved in the same collaboration scenario irrespectively of the employed proprietary systems. We call this kind of services as "*Data-Providing Services*" as opposed to "*Functionality-Providing services*" since their invocation only returns a piece of information without causing any change in the environment (e.g. charging a credit card, ...etc). Data-Providing services are very common in eHealth collaboration environments, in the same health site, for example, they are used to encapsulate and integrate numerous proprietary data sources that otherwise cannot be integrated, e.g. sensors, equipments equipped with proprietary interfaces...etc. Across different sites, they are used to share patients records, or as a means to transfer outsourced data. Another motivation behind the use of DP services in eHealth is the necessity to constrain the way data is accessed, e.g as a result of privacy constraints [10, 14] or maybe to enforce some access rights. Readers are referred to [5] for further information on the use of services in eHealth.

These e-collaboration environments can be modeled as peer-to-peer environments where every peer holds a collection of DP services, puts them at the disposal of its partners and, in return, they provide it with their DP services. The collaboration implies that some of the peer's data items are outsourced or stored at its partners and that it needs to make use of their DP services to retrieve these items when needed.

Obviously these services must be taken advantage of when answering queries, e.g. a query issued at peer1 (see figure 1) asking some information about a patient admitted at P1, both of P1's local DP services and DP services offered by its partners (i.e. those peers involved in the patient's current treatment scenario, e.g. external laboratories, hospitals...etc) must be exploited in the query answering process.

So far, Peer Data Management and Integration Systems [8, 3, 15, 11, 1, 16] were only concerned with handling traditional data sources (as opposed to services). In these systems, peers are supposed to directly hold and expose their data, either in a syntactic form (XML), or recently in some semantic forms (OWL instances plus some inferencing capabilities), then when they are interrogated, they apply queries squarely

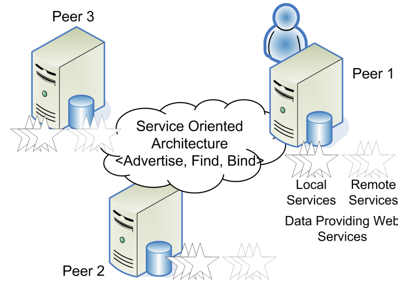

Fig. 1. Data sharing via DP Web services in a loosely coupled environment

to data instances in order to materialize answers. However, none of these systems pay attention to the consequences raised by the adoption of DP services for data sharing. With this form of data accessibility, it becomes impossible to materialize data in any forms or structures before applying, in a subsequent step, queries to it, because data portions are behind services probably offered by different bodies. The query resolution here necessitates to decompose the received query in terms of available services, compose these services, and to coordinate their execution before getting the desired results.

1.1 Our approach

In this paper we suggest a framework to provide better support for data sharing and integration in *e*Systems that make an extensive use of Data-Providing services in their routine collaboration scenarios due to previously seen reasons. In this framework individual peers adopt the three-layer stack presented in figure 2. The lowest layer or the *Data-Providing Services layer* holds (or makes reference to) services that contribute data items pertaining to the peer in question. These services are either local or remote ones (offered by the peer's partners), and they can be picked up based on the SOA[13] model. In the second layer or the *views layer* we model previously selected services (in the first layer) as *RDF parameterized views* over the peer's local ontology. These views serve us for query resolution in the next layer. In the third layer or the *Data-Providing services composition and execution layer*, we make use of previously defined views to decide what are the services whose composition can satisfy a received query (either a local query or a query received from the peer's acquaintances).

Note that this is different from the traditional Web services composition [12] since the latter is "task-driven" as composed services collaborate to achieve a more complex task (or functionality), e.g. a full-package journey reservation service out of plane, hotel and car booking services. In our work, the composition is "data-driven" where composed services collaborate to achieve as much complete answer as possible.

The main contributions of this paper are: 1). Supplementing previously seen P2P systems with the capability to handle Service-Accessed resources by proposing a new approach for answering queries using DP services, this includes modeling services

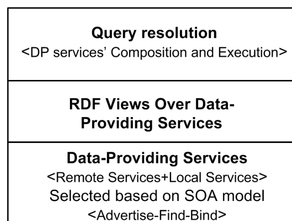

Fig. 2. The Stack adopted by our peers for query resolution in *e*Systems adopting Data-Providing services

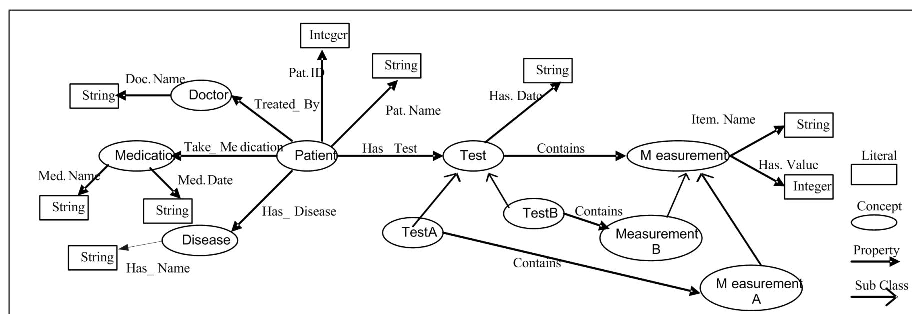

Fig. 3. An example of OWL ontology modeling the peer's local data items and the items provided by its partners

as RDF parameterized views, and a service-based query rewriting algorithm that is capable to compute the possible services compositions answering a given query. 2. A P2P system (under development) that supports the needs encountered in *e*Systems that make an extensive use of DP Web services. In this system, peers which still expose their data in forms such as (OWL, RDFS instances) can still inter-operate with the others since our queries are issued in SPARQL [1], a standard query language suited for querying data in the Semantic Web.

The remainder of the paper is organized as follows. Section 2 describes a motivating example that is used in the rest of the paper to clearly describes various concepts. In section 3, we model both our queries and the DP services. Section 4 is devoted to the query rewriting in terms of services. Section 5 discusses the possible treatments applied on data flow between composed services in the execution time. In section 6 we view the implementation status of our approach. In section 7 we review related works. Finally, in section 8 we conclude the paper and present our future works.

2 Running Example

Consider the case of the peer P1, it holds the ontology (defined with OWL) depicted in figure 3, and it has some services at its disposal to retrieve the different data

[1] http://www.w3.org/TR/rdf-sparql-query/

items modeled by this ontology. In particular, it outsources test information about its patients to two independent laboratories (i.e. where it usually sends its patients to effectuate tests), they provide it with interfaces of two services WS_1 and WS_2 to retrieve *Test A* ,*Test B* (specializations of Test) respectively. It also discloses part of its own database to authorized users in its environment via the services; WS_3: it returns the medications list taken by a given patient. WS_4: it returns patients (their names) who have been administered a given medication. WS_5: it returns patients with a given disease. Now assume a practitioner carrying out some experimental research has issued the following query on P1: *"Q1: what are the tests performed by patients who have been administered a medication termed as "Some Stuff"?*. Obviously the resolution of such a query necessitates the combination of several services, in particular WS_1, WS_2 and WS_4 (local and remote services). Notice that it does not suffice to rely on the services' inputs and outputs to decide whether they can answer the query or not, rather the semantic relation between the service's input and output must be taken in consideration. In the next section, we capture this relation by modeling a service as a *RDF parameterized view*. Queries are rewritten in terms of services by exploiting these views.

3 Modeling issues

This section is devoted to model both our queries and Data-Providing Web services. Based on this modeling we devise an algorithm to rewrite queries in terms of services.

3.1 Ontology and queries

OWL has become the de facto standard for modeling Web resources. OWL primitives include classes, properties and literals (called as Datatypes). Properties break up in two types; Object properties relating classes and Datatype properties relating classes to literals.

Definition In our context an OWL ontology O is a 6-tuple (C, L, DP, OP, SC, SP, X) representing a graph, where:

1. C is the classes set within O.
2. L is the Literals in O. C and L are the nodes in O.
3. DP is the datatype properties set linking a node from C to a node from L. OP is the object properties set holding among nodes from C.
4. SC is a set of directed edges (c_1, c_2) from node c_1 to node c_2, where c_1, $c_2 \in C$. Each edge in SC represents an isA relationship. These edges can be explicitly given by the user or inferred by an OWL inference engine based on the definition of the classes.
5. SP is set of edges representing the the subProperty relations between properties.
6. X is a set of axioms, that describe additional constraints on the ontology.

Materialized OWL instances have similar structure of the previous graph. Users in our framework are allowed to issue queries on the OWL instances ' graphs.

Given the previous definition, a query on the local ontology has the following form:

$\{ \{ ?c_1 . \Psi . p_{1.2} . ?c_2 . \Psi . p_{2.3} .... \Psi . p_{n-1.n} . ?c_n \}, CL, OS \}$, *where:*

1. $?c_1 ... ?c_i ... ?c_n$ *are variables of types defined by classes within C,*
2. $p_{i.j}$ *is the object property linking* $?c_i$ *to* $?c_j$*. Both* $?c_{i:1\to n}$ *and* $p_{i.j}$ *constitute the "**backbone**" of the query.*
3. Ψ *is a linking operator and it is used when one variable* $?c_i$ *is linked to more than one other variable such that each of these variables pose a condition on the selection of* $?c_i$*. The semantics of this operator is that instances of* $?c_i$ *must satisfy all of the conditions specified by the* Ψ*'s outgoing paths.*
4. *CL is the constraints set imposed on datatype properties of* $?c_{i:1\to n}$*.*
5. *OS is the output set, it comprises output variables (and their projected datatype properties).*

We implement this form of queries with SPARQL query language. This form is suitable when matching queries against the services as we shall see next. In the spirit of this definition our query became:
Q1: {*?T1(Test)* . $[Has-Test]^{-1}$. *?P1(Patient)* . *[Take-Medication]* . *?M1(Medication), Ct=* {*$M1(Name="Some Stuff")*}, *Out=* {*?T1(Result)*} }

3.2 Modeling Data-Providing Web services as Views

Web services are usually modeled with the de facto standard for service description OWL-S. In particular, OWL-S's *Service Profile* permits to model the service's functionality, inputs and outputs. On the other hand, Data Providing Services have no explicit functionality, instead, the semantic relation holding between their inputs and outputs must be captured. Therefore OWL-S may not be the best choice for describing them since it does not allow to capture this relation. We model Data-Providing Services in our approach as *RDF Parameterized Views (PVs)* over OWL ontology as they necessitate a particular set of inputs (the parameters values) for their invocation and return a particular set of outputs. Initially a parameterized view is a technique that has been used to describe content and access methods in the widely used Global-as-View (GaV) integration architectures [7], and also recently to describe privacy constraints in [14].

Each PV is a predicate $WS_i(c_i)$***:- 4-tuple** <**Backbone, Ct, In, Out**>* where,

$WS_i(c_i)$ is called the *view head* and it comprises the name of corresponding service and its returned results. The rest is called the *view body* and it has the following contents:

1. ***Backbone*** *it comprises both the variables set* ***C*** *(of classes types) linking the input and the output of the service, and the object properties set* ***OP*** *relating the different variables in C.*
2. ***Ct*** *is the constraints set imposed on the datatype properties of C without being required inputs of the service.*
3. ***In*** *is the necessary literals for the service invocation.*

4. ***Out*** *is the output literals.*

According to this definition, the parameterized views for our example are presented in figure 4. Concretely we establish these views with RDF triples as showed in figure 5.

```
WS1(TestA (result)):-
  {  {?TestA . [Has_Test]^-1 . ?Patient },
   Ct:   {øø}
   In:   {$Patient(Name)}
   Out:  {?TestA(Result)}   }

WS2(Measurement(Name), TestB(Result)):-
  {  { ?TestB . Ψ { . [Has_Test]^-1 ?Patient
                  { . [Contains]^-1 . ?Measurment },
   Ct:   {øø}
   In:   {$Patient(Name)}
   Out:  {?TestB(Result), ?Measurment(Name, Value)} }

WS3(Medication(Name)):-
  {  {?Medication . [Take_Medication]^-1 . ?Patient}
   Ct:   {øø}
   In:   {$Patient(Name)}
   Out:  {?Medication(Name)} }

WS4(Patient(Name)):-
  {  {?Patient . [Take_Medication] . ?Medication},
   Ct:   {øø}
   In:   {$Medication(Name)},
   Out:  {?Patient(Name)}   }
```

Fig. 4. The defined Parameterized Views for the DP services in the running example

4 Web service-based query rewriting

In this section we first pay attention to the conditions under which a composition is considered as valid, then we present the pretreatments we do over our views and finally we present our rewriting algorithm.

For formal discussion assume a query $Q(Q_{backbone}, Ct_Q, OS_Q)$ and a set of services, each has a $PV_i(Ser_{i\ backbone}, Ct_i, In_i, Out_i)$. In order to satisfy Q, backbones union of selected services has to cover the query's backbone, the final output of the composition (the sum of Out_i of selected services) should satisfy OS_Q, and the Q's constraints list Ct_Q is satisfied with the union of Ct_i. However some special cases may occur while the query rewriting process; we review them briefly.

1. The union is larger than the query backbone with provision to all of the asked outputs. Herein it should be verified whether the additional concepts in the union have a corresponding input parameter necessary for the service invocation. In this case the service cannot be invoked as a necessary input will not be available and thus the composition is invalid.
2. OS_Q is not satisfied with the sum of Out_i as some literals do not appear in the output of the composition. Herein if one of the missing outputs is mandated then the composition of these services is invalid.
3. A constraint specified in the Ct_Q was dropped (e.g. patient gender must be male). Herein if dropped constraints were mandated then these services will be rejected. Otherwise these constraints can be enforced on data flow between services.

4. The composition of the selected services enforce an additional constraint that was not specified in the query's Ct_Q. Herein even though obtained results will be specific, they are still relevant ones and thus the composition is valid.
5. There is a conflicting constraint between Q and one of the selected services (e.g. the gender property has conflicting values male vs. female). The composition herein is invalid.
6. The union of the services backbones does not cover the query backbone. In this case these services must be rejected even if they return similar outputs to the demanded ones e.g. two services with one returning the doctors' names who have prescribed a medication, and the second returning the test results which were verified by a given doctor, although the composition here returns an output similar to that of Q(running example) it has not the same semantics as Q.

All of these observations were dealt with in our Web services-based query rewriting algorithm presented next.

```
WS1(?TestA<Result>):-
    Backbone  = { ?Patient . rdf:type .        Op1: Patient
                  ?Patient . Op1:has_Test . ?Test A
                  ?Test A .  rdf:type .         Op1: Test A  }
    Ct        = {    øø    }
    In        = { ?Patient . Op1: Pat_Name . $Name       }
    Out       = { ?Test A .  Op1:has_result .  ?A_Result }

WS2(?TestB<Result>, ?Measurment<Value>):-
    Backbone  = { ?Patient . rdf:type . Op1: Patient
                  ?Patient . Op1:has_Test . ?Test B
                  ?Test B .  rdf:type . Op1: Test B
                  ?Test B . Op1:contains . ?Measurement      }
    Ct        = {    øø    }
    In        = { ?Patient .Op1: Pat_Name.  $Name           }
    Out       = { ?Test B . Op1:has_result . ?B_Result
                  ?Measurement . Op1:hasValue .   ?Value     }

WS3(?Medication<Name>):-
    Backbone = {  ?Patient . rdf:type . Op1: Patient
                  ?Patient . Op1:Take_Medication . ?Medication
                  ?Medication . rdf:type . Op1: Medication        }
    Ct        = {    øø    }
    In        = {  ?Patient  . Op1: Pat_Name . $Name              }
    Out       = {  ?Medication . Op1:Med_Name . ?Med_Name         }

WS4 (?Patient<Name>):-
    Backbone = {  ?Patient . rdf:type .  Op1: Patient
                  ?Patient . Op1:Take_Medication . ?Medication
                  ?Medication . rdf:type . Op1: Medication        }
    Ct        = {   øø    }
    In        = {  ?Medication . Op1:Med_Name . $Name             }
    Out       = {  ?Patient . Op1: Has_Name . ?Patient_Name       }
```

Fig. 5. The Parameterized views defined for the running example's services

4.1 Preprocessing the defined RDF Parameterized Views

Before the rewriting process, the parameterized views should be preprocessed. This includes the following steps.

Step 1. Extending the obtained PVs to reflect OWL "explicit" subclassing statements For those peers which have not the capability to apply some reasoning while matching the query with available PVs, obviously a query making reference to the concept "Test" cannot be answered with a PV if this makes reference to another concept to define the same data item (e.g. the concept "TestA", a specialization of Test) although this PV (or service) returns relevant information. To remedy this, there are two possible solutions. The first is to include in the algorithm the capability

to verify whether a concept is a super/sub class of another (based on the ontology definition) while matching both query and services backbones. This is expensive in terms of the time necessary for the rewritings computation with large ontologies. The other solution is to extend previously defined PVs with the constraints *subClassOf, subPropertyOf* that are explicitly declared in the ontology. For example in (figure 6, case A) a new triple was added to the PV of WS_1 indicating that an instance of "TestA" is also an instance of "Test".

```
WS1(?TestA<Result>):-
  Backbone = {
      ?Patient .rdf:type . Op1: Patient
      ?Patient . Op1:has_Test . SF2(?Id)
      ?Test A. rdf:type . Op1: Test A
      ?Test A .rdf:type . Op1: Test      }
  Ct  ={øø }
  In  = {?Patient . Op1: Pat_Name . $Name  }
  Out={?Test A .Op1:has_result. ?A_Result }

A. Extending PVs with
   subClassing constraints

WS1(?TestA<Result>):-
  Backbone = {
      SF1(?n) . rdf:type . Op1: Patient
      SF1(?n) . Op1:has_Test . SF2(?Id)
      SF2(?Id). rdf:type . Op1: Test A
      SF2(?Id). rdf:type . Op1: Test      }
  Ct  = { øø }
  In  ={SF1(?n) .Op1: Pat_Name. $Name   }
  Out ={SF2(?Id) .Op1:has_result. ?A_Result}

B. Skolemizing Pvs
```

Fig. 6. An extended PV for WS1. A new triple was added (showed in bold)to reflect the relation between Test and Test A

Step 2. Skolemizing triples Variables denoting classes in PVs need to be *skolemized* [2], that is to replace each variable by a *skolem* function helpful to merge instances stemming from different services, e.g. the variable ?Patient (of type Patient) is replaced by the function SF1(Name), that is to say if two instances have the same name then they are considered as being denoting the same entity and thus can be merged. An example of a skolemized PV is shown in (figure 6, case B). The properties of a skolem function for a particular class are chosen by the domain expert.

Query rewriting algorithm

Inputs:
-A query Q $< Q_{backbone}, Ct_Q, OS_Q >$.
-The service List L, where each service $S_i \in L$ has a $PV_i(Ser_{i\ backbone}, Ct_i, In_i, Out_i)$.
1. Populate the list RSL (Relevant Services List) where $S_i \in$ RSL iff
   ($\exists\ ?c_i \in Out_i, \wedge \exists\ ?c_j \in OS_Q$) such that both $?c_i$ and $?c_j \in O : c_i$
1.1 **for** each $S_i \in L$ **do**
1.2 **for** each variable $?c_i \in OS_Q$ (: c_i is its corresponding type class in O) **do**
1.3 **if** (c_i appears in the Out_i) **then** (Add S_i to RSL)
1.4 **else if**(c_i is a subclass to one or more of used classes in OS_Q) **then**
1.5 Reject S_i since it returns generic result.
1.6 **else if** c_i is a superclass to one or more of used classes in OS_Q **then**
1.7 Add S_i to RSL
2 **if** RSL is not empty **then**
2.1 **for** each S_i in RSL **do**
2.1.1 Verifying whether S_i satisfies the Case 1 or not
2.1.1.1 Construct the types schema (sub graph of O) QTS used in Q;

2.1.1.2 Construct the types schema (sub graph of O) STS used in S_i;
2.1.1.3 **if** $\exists c_i$ such that $c_i \in$ STS and $c_i \notin$QTS **then**
2.1.1.4 **if**$\exists$?v $\in c_i$ such that ?v poses a constraint in $Ct_i \cup In_i$ **then**
2.1.1.5 Reject the service and treat another one
2.1.2 Verifying whether S_i's backbone is covered by Q's backbone
Take the backbones of S_i and the Q
Let $?c_1$ be the common output variable between S_i and Q
Let $?c_{in}$ be a variable enclosing some literals necessary for
the service's invocation
Let $?c_i$ be a variable varying from $?c_1$ till $?c_{in}$ in the service backbone
2.1.2.1 **for** ($?c_i$=$?c_1$ till $?c_i$=$?c_{in}$) in S_i's backbone **do**
Let $O : c_i$ be the corresponding class type of $?c_i$
Let $O : c_j$ be the corresponding class type of $?c_j$, where
$?c_j$ is $?c_i$'s analogous variable in Q's backbone
2.1.2.2 **if** ($\neg(c_i \equiv c_j)$ or $\neg(c_i$ subclass $c_j)$)**then**
2.1.2.3 Reject S_i
2.1.3 Let Ct_S and Ct_Q are the constraints sets pertaining
to variables involved in compared backbones
2.1.3.1 **if** exist a conflicting constraint between Ct_S and Ct_Q **then**
2.1.3.2 S_i is rejected
2.1.3.3 **else if** $Ct_S > Ct_Q$ **then**
2.1.3.4 $//S_i$ returns more restricted results, the user has
2.1.3.5 the choice to whether or not accept specific results
2.1.3.6 **else if** $Ct_S < Ct_Q$ **then**
2.1.3.7 S_i returns more general results, the user
has the choice to whether or not accept general results.
2.1.4 **if** S_i is not rejected yet **then**
2.1.4.1 Insert the service predicate in the query
2.1.4.2 Eliminate the service backbone from the query backbone
2.1.4.3 Eliminate the service output from the query output
2.1.4.4 Insert the service's inputs in the Q's outputs and mark
them as mandated ones
2.1.4.5 **if** Q:OS is not satisfied with the Q:Ct **then**
2.1.4.6 Repeat the algorithm on the new Query
3 **else** Q cannot be resolved
Output: The rewritings list.

The algorithm in operation The possible rewritings of our query are shown in figure 7. Our algorithm starts with looking for services which provide at least one of the sought outputs. It finds that two services do provide relevant results (WS_1 and WS_2) since these services return the results of "TestA" and "TestB" respectively (subclasses of "Test" (subClassing constraints were added in the views definitions)). Each of these services corresponds to an independent rewriting. The backbones of WS_1 and WS_2 match part of Q's backbone, and their constraints lists satisfy the involved constraints in the query's Ct. Next, in each rewriting the algorithm eliminates the service's backbone from the query's backbone and its provided output from the Q's output set. Then, it inserts the needed inputs for the service invocation as mandated outputs in the new query Q's OS (Output List). Obtained result after this iteration is shown in figure 7. Then the same algorithm is applied again on the new yielded query in each rewriting. This time, it turned out that WS_4 satisfies the new query as it provides the required outputs, its backbone matches exactly the query backbone, and its inputs are satisfied with query constraints list.

Q: { {?T1(type:Test) . [Has_Test] . ?P1(type:Patient)
. [Take_Medication]. ?M1(type:Medication)},
{ø}
{M1(Name="Some Stuff")},
{?T1(Result)} }

After the first iteration

WS1(Test (result))
Data flow (Patient)
New Q: {{?P1(type:Patient) . [Take_Medication] .
.?M1(type:Medication)},
{ø}
{M1(Name="Some Stuff")},
{?Patient (Name) }

WS2(Test (result))
Data flow (Patient)
New Q: {{?P1(type:Patient) . [Take_Medication] .
.?M1(type:Medication)},
{ø}
{M1(Name="Some Stuff")},
{?Patient (Name) }

After the second iteration

Rewriting 1
WS1(Test (result))
Data flow (Patient)
WS4(Patient)
Data flow (M1(Name= ``Some Stuff"))

Rewriting 2
WS2(Test (result))
Data flow (Patient)
WS4(Patient)
Data flow (M1(Name= ``Some Stuff"))

Fig. 7. The query rewriting process for the running example. There were two yielded rewritings for Q

5 The rewritings execution

The previous algorithm yields a certain number of query rewritings (set of DP services compositions). Before executing these compositions, we apply an extra algorithm (this was left out for space limitation) in order to superimpose these compositions (if possible). The intent of this algorithm is to avoid the duplicate invocation of the same service across several compositions, e.g. instead of invoking WS_4 twice, it suffices to invoke it one time and then to use its results to invoke both of WS_1 and WS_2 (see figure 8, case a). In our system the results of each service invocation are materialized in the form of OWL instances (skolem functions are exploited in this respect) before being used to invoke subsequent services or sent to the requester . This helps in merging and aggregating results stemming from different services and enable us to detect, if needed, data inconsistences (between services) based on some semantic reasoners (e.g. two instances of patient with the same value of a functional property (e.g. national ID) but with different names, probably misspelled, are detected).

We need to apply extra treatment and processing over data flow among combined services. In general, three semantic operators can be applied. They are as follows.

- **Semantic Union** ($WS_i \cup WS_j$): This operator is used to semantically combine the outputs (OWL instances) of the services WS_i, WS_j. The outcome includes the disjoint instances provided by WS_i and WS_j and the semantically equivalent instances provided by both only once. For example suppose that there is a second service ($WS*$) equivalent to WS_4 (depicted with doted lines in figure 8, case b). Here, obtained results of both $WS*$ and WS_4 must be combined with the union operator before invoking the subsequent services.
- **Semantic Intersection** ($WS_i \cap WS_j$): This operator can be used to return the semantically equivalent instances provided by both WS_i and WS_i.
- **Semantic Difference** ($WS_i \oslash WS_j$): It can be used to return the instances provided by WS_i excluding the equivalent instances provided by WS_j.

Note that treatments on data flow are applied in the execution time of the composition. Returning back to our example, the execution of the compositions is done as follows. First, WS_4 is invoked with medication name. The returned patients' information is put automatically in the form of OWL instances. Then, for each obtained instance of patient we invoke both WS_1 and WS_2 and the results are materialized as OWL instances then sent to the requester.

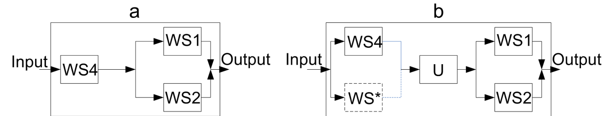

Fig. 8. Data flow in Data driven Web services composition

6 implementation issues

We are focusing on integrating our framework for supporting DP services within a P2P-based data management system that is being developed by our research team for the purpose of integrating proprietary data resources in eHealth. In our system, each peer maps its proper ontology to the neighboring ontologies (in a pair-wise manner) via *OWL mapping constructs*. When a peer receives a query from neighboring peers (or from the peer's user) it tries to resolve it by composing its DP services. We have implemented our algorithm using Jena Framework [6]. Currently we are conducting some experimental tests on our algorithm to measure the impact of PVs number and the ontology volume on the rewritings computation time.

7 Related works

The work presented in this article is closely related to research in several areas. We review them briefly (for space concerns).

7.1 Data integration & interoperation in P2P systems

Previous works in this area like [15, 8, 11, 3, 1, 16, 4] have focused on integrating traditional data sources (as opposed to DP services) in the P2P environment. None of which has dealt with Service-Accessed sources in the P2P sharing environment. Furthermore, data sources are either integrated in a syntactic way, e.g. using *mapping tables*[15], *GaV/LaV-based mappings*[8], or fully transformed into forms like (OWL or RDFS) before applying, in a subsequent step, queries to data in these forms(like in [16, 4]). However, both ways are not suitable to handle DP services. Our services are described as RDF triples-based Views over the peer's ontology and that, in principal, enables for applying some reasoning when matching the query against the views since these triples represent concepts and properties of the underlying ontology.

7.2 Web services description & composition

Concerning the service description aspect, we have modeled DP services as RDF triples-based parameterized views over the peer's ontology. We would have used OWL-S[2] for describing DP services, but effectively OWL-S's service profile does not permit to represent the In/Out's semantic relationship of a DP service w.r.t. the underlying ontology.

Hull et al. attempt in [9] to deal with problem of Query/Service matching of stateless services. However this work is more concerned with the matching decidability issues, and they do not take into account the constraints that would be imposed on services. In addition, their approach was not tested or implemented.

Concerning the composition aspect, as we previously pointed out, the key difference between the traditional Web services composition in research works like [17, 12] and the composition in our work is that while composition in these approaches is task-driven, it is data-driven in our work. That is to say, the ultimate objective of the composition is to provide as much complete answer as possible to the user queries. Also in our composition we need to superimpose obtained compositions and to apply extra treatments on data flow between services (data aggregation, redundancy elimination...etc).

8 Future works

We have several research directions in mind.

1. To adapt the query rewriting algorithm so that it allows for handling privacy constraints.
2. To take measures in order shorten the execution time of obtained composition.
3. To include data-mediating services in our algorithm (to convert data values if a conversion is needed e.g. the conversion of a measurement unit).

References

1. Sonia Bergamaschi, Pablo R. Fillottrani, and Gionata Gelati. The sewasie multi-agent system. In *AP2PC*, pages 120–131, 2004.
2. Huajun Chen, Zhaohui Wu, Heng Wang, and Yuxin Mao. Rdf/rdfs-based relational database integration. In *ICDE*, page 94, 2006.
3. Isabel F. Cruz, Huiyong Xiao, and Feihong Hsu. Peer-to-peer semantic integration of xml and rdf data sources. In *AP2PC*, pages 108–119, 2004.
4. Dimitre A. Dimitrov, Jeff Heflin, Abir Qasem, and Nanbor Wang. Information integration via an end-to-end distributed semantic web system. In *International Semantic Web Conference*, pages 764–777, 2006.
5. A. Dogac, G. Laleci, S. Kirbas, Y. Kabak, S. Sinir, A. Yildiz, and Y. Gurcan. Artemis: Deploying semantically enriched web services in the healthcare domain. *Information Systems Journal (Elsevier)*, 2006.
6. Jena Framwork. http://jena.sourceforge.net/.

[2] http://www.daml.org/services/owl-s/

7. Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.
8. Alon Y. Halevy, Zachary G. Ives, Jayant Madhavan, Peter Mork, Dan Suciu, and Igor Tatarinov. The piazza peer data management system. *IEEE Trans. Knowl. Data Eng.*, 16(7):787–798, 2004.
9. Duncan Hull, Evgeny Zolin, Andrey Bovykin, Ian Horrocks, Ulrike Sattler, and Robert Stevens. Deciding semantic matching of stateless services. In *AAAI*, 2006.
10. Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David J. DeWitt. Limiting disclosure in hippocratic databases. In *VLDB*, pages 108–119, 2004.
11. Alexander Löser, Wolf Siberski, Martin Wolpers, and Wolfgang Nejdl. Information integration in schema-based peer-to-peer networks. In *CAiSE*, pages 258–272, 2003.
12. Zakaria Maamar, Djamal Benslimane, Chirine Ghedira, and Michael Mrissa. Views in composite web services. *IEEE Internet Computing*, 9(4):79–84, 2005.
13. Mike P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE*, pages 3–12, 2003.
14. Shariq Rizvi, Alberto O. Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD Conference*, pages 551–562, 2004.
15. Patricia Rodríguez-Gianolli, Maddalena Garzetti, Lei Jiang, Anastasios Kementsietsidis, Iluju Kiringa, Mehedi Masud, Renée J. Miller, and John Mylopoulos. Data sharing in the hyperion peer database system. In *VLDB*, pages 1291–1294, 2005.
16. Eirini Spyropoulou and Theodore Dalamagas. Sdqnet: Semantic distributed querying in loosely coupled data sources. In *ADBIS*, pages 55–70, 2006.
17. Dan Wu, Bijan Parsia, Evren Sirin, James A. Hendler, and Dana S. Nau. Automating daml-s web services composition using shop2. In *International Semantic Web Conference*, pages 195–210, 2003.