A Framework for Building Privacy-Conscious DaaS Service Mashups*

Mahmoud Barhamgi, Djamal Benslimane, Chirine Ghedira, Salah-Eddine Tbahriti, Michael Mrissa
Claude Bernard Lyon1 University, 69622 Villeurbanne, France
firstname.lastname@liris.cnrs.fr

***Abstract*—Data Mashup is a special class of mashup application that combines information on the fly from multiple data sources to respond to transient business needs. Data mashup is a difficult task that would require an important programming skill on the side of mashups' creators, and involves handling many challenging privacy and security concerns raised by data providers. This situation prevents non-expert users from mashing up data at large. In this paper, we present a declarative approach for mashing-up data. The approach allows data mashup creators to build data mashups without any programming involved. The approach builds the mashups automatically and takes into account the data's privacy concerns. We evaluate the efficiency of the approach via a thorough set of experiments. The results show that handling data privacy introduces only a negligible cost in the mashup building time.**

***Keywords*-Privacy, Data Mashup, DaaS Web Services.**

I. INTRODUCTION

Mashup is a Web application that integrates data, computation and UI elements provided by several applications to create on-the-fly new applications. *HousingMaps.com* is an example of a Web site that "mashes-up" two other Web sites: *CraigsList* and *Google Maps*; it takes housing information from *CraigsList* and displays them on *Google*'s maps. The *ProgramableWeb.com* site lists more than /4800/ online mashups created by Web users.

Data mashup is a special class of mashup application that combines information from several data sources (typically provided through Web Services; this type of services is known as *DaaS Data-as-a-Service Web services* [1], [2]) to meet user requests [3]. Data mashup has become so popular over the last few years; its applications vary from addressing transient business needs in modern enterprises [4] to conducting scientific research in e-science communities [3]. However, in spite of its popularity, current data mashup applications are still limited to very primitive information integration. This is due to many challenges introduced by data mashup both for mashup users (i.e. mashup's creators) and data providers (i.e. data service providers). On the side of mashup users, mashing-up data involves carrying out many challenging tasks including: selecting the data services that are relevant to user's needs, mapping their inputs and outputs to each other (and probably adding some mediation services/functions when inputs/outputs don't fit each other) and performing some processing on intermediate results (e.g. joining the outputs of two services, projecting some attributes, etc). In addition, data mashup are usually written in some procedural programming languages such as *JavaScript*, and the code rarely separates the user interface layer (dynamic HTML) from the data integration layer. These challenging tasks hinder average users from building data mashup applications at large. On the side of data providers, mashing-up data raises many concerns related to data privacy and security [5]. Indeed, data providers are often reluctant to engage in data mashup scenarios for the fear that their data may be disseminated to untrusted parties or used for unintended purposes.

A. Motivating Example

Let us assume the following scenario from the healthcare domain. Assume the physician *Alice* would like to study the effects of a given medication on the cholesterol level of patients. Assume she has at her disposal the services in Table-I; these services access and manipulate the electronic healthcare records (EHRs) of patients and are provided by different healthcare facilities in a private healthcare collaboration environment.

Assume that the medical and the personal information of the patient *Cathy* are accessed by the services from above. *Cathy* was prompted at each healthcare facility to enter her privacy preferences. *Cathy* has agreed to share the results of her medical tests and diseases, but not personal information such as name and address, with third-party scientific organizations for research purposes. *Mike*, another patient, has agreed to share all his personal and medical information with scientific organizations for research purposes.

Alice can use the data services in Table-I to meet her needs as follows: she invokes S_1 and S_2 with the given medication; then she invokes S_3 with the obtained patients to retrieve their personal information. Then she invokes S_4 to retrieve the tests whose type is *Cholesterol Test*.

Alice is faced to the following challenges in this example. First, she needs to delve into the data service space and understand the semantics of each individual service in order to identify the services that may contribute to the resolution of her request. Many services may have the same input's and output's types, but completely different semantics. For

* This research work is supported by the French National Research Agency under grant number ANR-09-SEGI-008

978-0-7695-4463-2/11 $26.00 © 2011 IEEE
DOI 10.1109/ICWS.2011.91

Table I
AVAILABLE DATA WEB SERVICES

Service	Semantics
$S_1(\$a, ?b)$	*Returns the patients "b" that have been administered a given medication "a"*
$S_2(\$a, ?b)$	
$S_3(\$a, ?b, ?c)$	*Returns the personal information (name "b" and address "c") about a given patient "a"*
$S_4(\$a, ?b, ?c)$	*Returns the tests ("b" their types, "c" their value) performed by a given patient "a"*
$S_5(\$a, ?b)$	*Returns the diseases "b" of a given patient "a"*
$S_6(\$a, ?b)$	*Returns the diseases "b" against which a given patient "a" is vaccinated*

example, the services S_5 and S_6 have the same input and output (*Patient* and a *Disease*, respectively), the first returns the patient's diseases while the second returns the diseases against which the patient is vaccinated. Second, *Alice* needs to select the participant services and build the data mashup application. She should realize that the services S_1 through S_4 are necessary for her needs, figure out their execution order and construct the mashup's execution plan. For example, she should realize that S_1 and S_2 can be executed in parallel and write some programming code to unify their outputs (to eliminate redundant tuples); then she should map the obtained output to the inputs of the services S_3 and S_4 (she should realize that these two services can be executed in parallel) and write some programming code to join their outputs. The output of the join will be the output of the constructed data mashup. Non-expert users like *Alice* (a physician) are not able to conduct the previous tasks that require important technical and programming skills.

In addition to these challenges, data in data mashup application are often private and sensitive. Its usage is often subject to privacy and security constraints imposed by data providers. For example, the lab A (S_4's provider) may specify that the test information can be accessed "*unconditionally*" by some healthcare authority; "*conditionally*" by a scientific organization conducting some research (i.e. the purpose for which the tests are requested). An example of conditions could be the respect of patients' preferences as to the disclosure of their data. It may also specify that the tests are "*forbidden*" for an organization needing them for doing publicity.

In this paper we propose a declarative and privacy preserving approach for mashing up DaaS web services. Based on "declarative" mashup queries over domain ontologies and a set of privacy and security polices provided by service providers, our proposed data mashup system generates detailed descriptions of the mashup that fulfills those queries and preserves data privacy. We summarize bellow our major contributions in this paper:

First, we propose to model DaaS services as RDF views over domain ontologies. An RDF view allows capturing the semantics of the associated DaaS service in a "declarative" way based on concepts and relationships whose semantics are formally defined in domain ontologies. Second, we propose to use query rewriting techniques for mashing-up data. The use of these techniques to mashup data enables average users to mashup data as all what they need to do is just specifying their mashup queries declaratively. Third, we propose a privacy aware data mashup model. Our model, given a set of privacy policies defined on domain ontologies, rewrites received mashup queries to accommodate pertaining privacy conditions (from privacy policies) before their resolution by the core mashup algorithm.

The remainder of this paper is organized as follows. In Section II, we model data mashup queries, data web services and privacy policies over domain ontologies. In Section III, we present our declarative approach to construct data mashups. In Section IV, we evaluate the proposed approach in the healthcare application domain. In Section V, we compare our approach to related works. Finally, Section VI summarizes and concludes the paper.

II. DATA SERVICES, QUERIES AND POLICIES

A. Data Mashup Queries

In the proposed approach, mashup creators formulate their data mashup queries against a Domain Ontology Ω. We consider the class of conjunctive queries with arithmetic comparisons expressed in SPARQL query language over RDFS domain ontologies. Formally, a mashup query is defined as follows:

Definition 1 (Mashup Queries)

$$Q(\overline{X}) :- < G(\overline{X}, \overline{Y}), C >$$

where $Q(\overline{X})$ is called the head of the query; it has the form of a relational predicate. $\overline{X}$ and $\overline{Y}$ are called the head (or distinguished) and existential variables, respectively. $G(\overline{X}, \overline{Y})$ is called the body of the query; it contains a set of RDF triples where each triple is of the form (subject.property.object). C is a set of constraints on the body variables, each constraint is of the form: $x\ f\ a$ where x is variable, $f \in \{=, >, <, \leq, \geq\}$ and a is a constant. □

Figure1 (Part-A) shows the graphical representation of our running example query Q_1. A query can be seen as a graph with two types of nodes: *class* and *literal nodes*. Class-nodes refer to classes in Ω (e.g., M, P and T are class-nodes). They are linked via object properties and represent existential variables in the query. Literal nodes represent data-types (e.g., x, w, z). They are linked with class nodes via data-type properties. Literal nodes may correspond to both existential and distinguished variables in a query. The blue ovals in Figure1 (Part-A) represent concepts in Ω (e.g. Medication, Patient and Test). The variables preceded by the symbol \$ represent the mashup's inputs (e.g. x) and the variables preceded by ? represent the mashup's outputs.

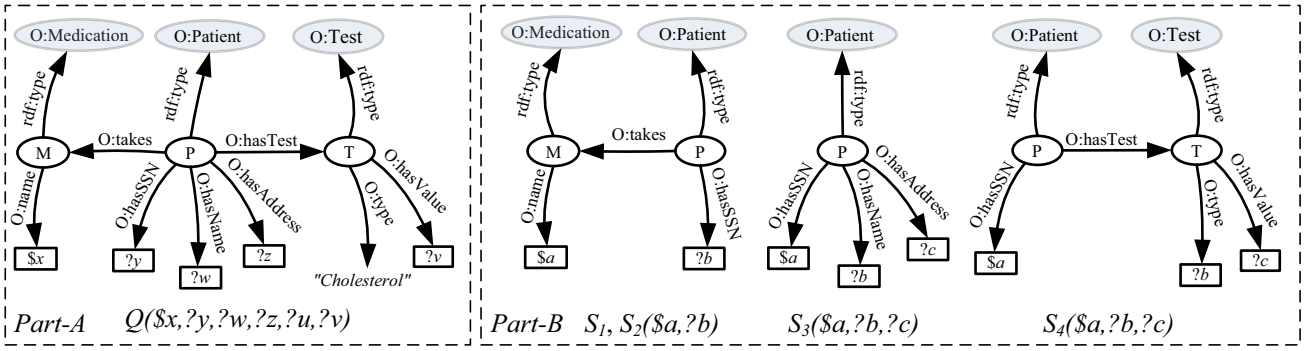

Figure 1. Part-A: The Query of the Running Example, Part-B: the RDF Views of Available Data Services

B. Data Web Services

Contrary to SaaS Web services, the semantics of a Data Web service (a.k.a. DaaS Web service) cannot be captured based solely on its inputs and outputs, preconditions and effects, rather this requires capturing the semantics of the relationship that holds between its inputs and outputs. For this reason, we model DaaS services as *RDF Parameterized Views* ($RPVs$) over domain ontologies Ω. RPVs use concepts and relations from Ω to capture the semantic relationships between input and output sets of a DaaS service.

Formally, a DaaS Web service S_i is described over a Ω as a predicate $S_i(\$\overline{X_i}, ?\overline{Y_i}) : - < RPV_i(\overline{X_i}, \overline{Y_i}, \overline{Z_i}), C_i >$, where:

- $\overline{X_i}$ and $\overline{Y_i}$ are the sets of input and output variables of S_i, respectively. Input and output variables are also called as distinguished variables. They are prefixed with the symbols "$"and "?" respectively.
- $RPV_i(\overline{X_i}, \overline{Y_i}, \overline{Z_i})$ represents the semantic relationship between input and output variables. $\overline{Z_i}$ is the set of existential variables relating $\overline{X_i}$ and $\overline{Y_i}$. $RPV_i(\overline{X_i}, \overline{Y_i}, \overline{Z_i})$ has the form of RDF triples where each triple is of the form (subject.property.object).
- C_i is a set of data value constraints expressed over the $\overline{X_i}, \overline{Y_i}$ or $\overline{Z_i}$ variables.

Figure1 (Part-B) shows the graphs corresponding to the RPVs of the services in the running example. An RPV requires a particular set of inputs (the parameter values) in order to retrieve a particular set of outputs; outputs cannot be retrieved unless inputs are bound. For example, one cannot invoke the service S_1 from above without specifying a medication for which it is needed to learn the patients that have been taking it. Therefore, a parameterized view indicates in its head which parameters are inputs, and which parameters are outputs.

C. Privacy Policies

In our data mashup model, upon publishing a new DaaS service to the mashup server, service providers provide also the privacy policies regulating the usage of their published services. A privacy policy is a set of rules specifying to whom the provided data may be disclosed (a.k.a *recipients*) and how the data may be used (a.k.a. *purposes*). A privacy policy may further personalize data disclosures by defining conditions under which a data item is disclosed. For example, the medical test information may be disclosed to given recipient if patients have opted in to approve the disclosure.

We suppose that privacy policies are defined over domain ontologies. For each datatype property of a data concept within a domain ontology, privacy rules specify the *recipients* that have access to the value of the property, the *purpose* for which the access is granted, and a set of *conditions* that must be met.

Formally, a privacy rule is a $4\text{-}tuple < R, P, S, PC >$, where: R is the class of recipients for which the authorization is specified, P is the purpose for which the data can be accessed, S is the data class whose data properties will be accessed, PC is a set of $Property\text{-}Conditions$ (P_i, C_i) couples; the semantics of each couple is that the property P_i of the concept S can be accessed if the set of conditions C_i is satisfied.

Each condition C_i is expressed against concepts and relations in domain ontology using RDF queries (e.g. using SPARQL query language). The Rule-1 below specifies that personal data such as *name* can be released to a *Researcher* for the purpose of *Conducting Research*, provided that the data subject (i.e. the patient) has *consented to this disclosure* (the same condition applies to the property *hasAddress*). The condition is specified using the ontological concepts *Patient* and *PatientPrivacyPrefernces* from Ω (the concept *PatientPrivacyPrefernces* is defined in the considered domain ontology to model the user's privacy preferences as to the disclosure of his personal data, e.g. name, address, etc).

```
Rule-1:

[R :Researcher,
 P :Research,
 S :Patient,
 PC:{<hasName, "?P  rdf:type O:Patient,
                ?P  hasPrivacyPreferences  ?PP,
                ?PP rdftype O:PatientPrivacyPrefernces,
                ?PP hasPrivacyPrefName  'yes'">,
```

```
   <hasAddress, "?P  rdf:type O:Patient,
                 ?P  hasPrivacyPreferences  ?PP,
                 ?PP rdftype O:PatientPrivacyPrefernces,
                 ?PP hasPrivacyPrefAddress  'yes'">,
     }
]
```

The Rule-2 below specifies that the property *hasValue* of the concept *Test* may be disclosed to a *Researcher* for *conducting research* provided that the patient has consented to this disclosure. Note that the patient's privacy preferences regarding his medical tests are modeled by the ontological concept *TestPrivacyPrefernces*, the object property *hasTest-PrivacyPreferences* links the concept *Patient* to *TestPrivacyPrefernces* in Ω.

```
Rule-2:

[R :Researcher,
 P :Research,
 S :Test,
 PC:{<hasValue, "?P  rdf:type O:Patient,
                 ?P  hasTestPrivacyPreferences  ?TP,
                 ?TP rdftype O:TestPrivacyPrefernces,
                 ?PP PrefValue  'yes'">,

     }
]
```

III. A Declarative Approach to Data Mashups

In this section, we describe our privacy-preserving data mashup approach. Our approach consists of three steps: (*i*) rewriting the mashup query to accommodate pertaining privacy constraints, (*ii*) rewriting the modified mashup query in terms of available data web services, and (*iii*) constructing the data mashup plan.

A. Accommodating Privacy Constraints in the Mashup Query

In this step declarative data mashup queries are rewritten to accommodate pertaining privacy conditions from data privacy policies. Since privacy constraints have the forms of RDF queries, they can easily be incorporated in the posed data mashup queries. For example, SPARQL allows for the incorporation of such constraints at the *datatype property level* by using the $OPTIONAL$ construct. The semantics of the $OPTIONAL$ construct is as follows: in a conjunctive RDF query, all query variables must bind to values in the matched RDF graph in order for the query to return results. If a query variable is defined as optional (i.e. it is defined inside an $OPTIONAL$ block), then the query may still be resolved if that variable is left unbound (i.e., when some RDF triples are missing in the matched graph.). In this case, Null values will be returned for unbound variables. Privacy can be enforced at the datatype property level (of each data concept from Ω) by putting each datatype property that is subject to privacy conditions along with the conditions that must be met to disclose the property value inside the same $OPTIONAL$ block. If these conditions are false, the datatype property will be withheld independently of other datatype properties requested in the query. For example, the mashup query Q_1 from above can be rewritten to include the privacy constraints specified in Rules-1 and Rule-2 as follows (Parts A and B of Figure 2 show Q_1 before and after the modification respectively): (i) the datatype property *hasSSN* is prohibited for *Researchers* and as a result the distinguished variable $?y$ is deleted from the SELECT clause, (ii) the property *hasName* can be accessed by *Researcher* for *conducting scientific researches* provided the patient's consent; the property *hasName* along with its privacy condition (*?PP hasPrivacyPrefName "Yes"*) are put inside an $OPTIONAL$ block. If the property *hasPrivacyPrefName* does not bind to the value "yes", the variable $?w$ will be assigned the Null value independently of the other datatype properties in Q_1. The same applies to the properties *hasAddress* and *hasValue* (the latter represents the medical Test's value). However note that in data mashup applications mashup queries are not matched directly against data, rather they are only accessible through a set of data services and therefore mashup queries need to be rewritten in terms of available services. In the next section we propose an RDF-oriented query rewriting algorithm to rewrite the data mashup query in term of DaaS Web services. Our RDF query rewriting algorithm handles conjunctive queries (i.e. all RDF triples in the query are "implicitly" linked by the AND operator). The presence of the $OPTIONAL$ constructs in the modified query makes it a non-conjunctive one. To keep the mashup query processable by the query rewriting algorithm, all privacy conditions are added in the *conjunctive form* to the mashup query but without enforcing any of the specific data values that are used in those conditions.

For each datatype property p that is subject to privacy conditions C_{p_i}, we conjunctively extend the mashup query Q with RDF triples representing C_{p_i} without enforcing specific data value constraints (i.e. equality and order data value constraints, e.g. $x = 10$). For example, the datatype property $hasName$ in Q_1 has the following privacy condition:

```
<<hasName,   "?P  rdf:type O:Patient,
              ?P  hasPrivacyPreferences  ?PP,
              ?PP rdftype O:PatientPrivacyPrefernces,
              ?PP hasPrivacyPrefName  'yes'">,} ]
```

The condition prescribes that the instance of the class $PatientPrivacyPrefernces$ that is associated with the patient whose name is requested in the query must have the value "yes" for its datatype property $hasPrivacyPrefName$. Therefore, Q_1 is rewritten to project out the value of the property $hasPrivacyPrefName$ as follows:

```
<<hasName,   "?P  rdf:type O:Patient,
              ?P  hasPrivacyPreferences  ?PP,
              ?PP rdftype O:PatientPrivacyPrefernces,
              ?PP hasPrivacyPrefName  ?w1 ">,} ]
```

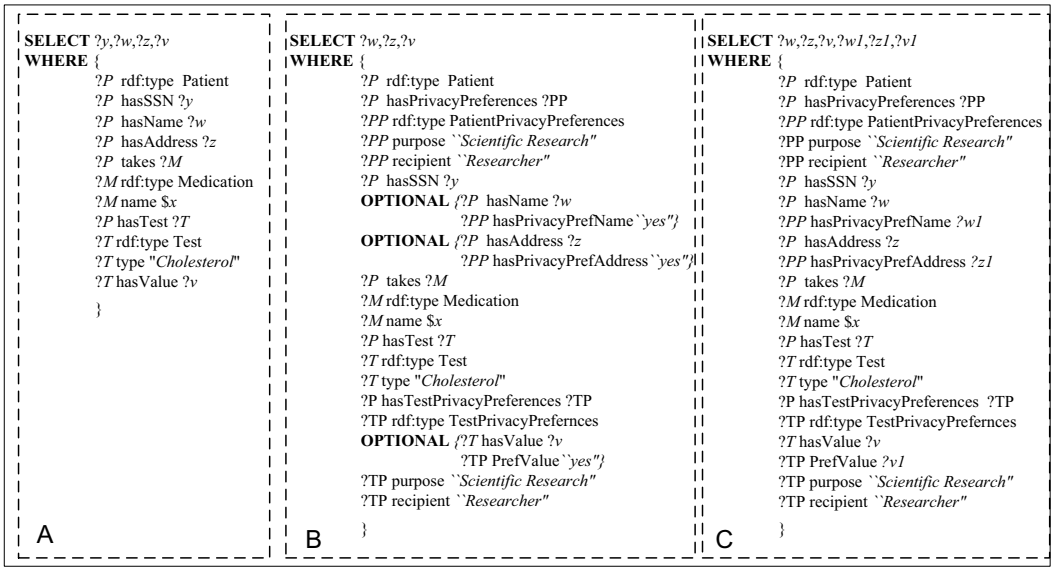

Figure 2. Part-A shows the original query (Q_1), Part-B shows the modified query with the optional constructs, Part-C shows the modified query in the conjunctive form

The modified query does not enforce any specific value for the newly added property $hasPrivacyPrefName$. It just binds it to a new distinguished variable w_1 (i.e., variable appearing in the query head). Specific value enforcement, such as the constraint w_1="yes", will be carried out in a later step, e.g. the constraint w_1="yes" will be tested in the later step to decide whether or not the patient's name shall be disclosed to the recipient. The same applies to the rest of datatype properties that are subject to privacy constraints in Q_1. The modified query at the end of this step is shown in Figure 2 (Part-C). Q_1 then becomes a conjunctive RDF query that can be rewritten in terms of available services.

B. Mashup Query Rewriting

In a previous work [17] we proposed an efficient RDF query rewriting algorithm. Given a data mashup query Q and a set of DaaS services represented by their corresponding RPVs $V = v_1, v_2, v_i$, the algorithm rewrites Q as a composition of DaaS services whose union of RDF graphs (denoted to by G_V) covers the RDF graph of Q (denoted to by G_Q).

The rewriting algorithm has two phases:

1) Phase-I: Finding Relevant Sub-Graphs: In the first phase, our data mashup system compares G_Q to every RPV v_i in V and determines the class nodes and object properties in G_Q that are covered by v_i. The system stores information about covered class nodes and object properties as a partial containment mapping in a *mapping table*. The mapping table points out the different possibilities of using an RPV to cover parts of G_Q.

Example: Let us illustrate this phase using our running example. We consider the following candidate services:

- The Services S_1 and S_2- S_1 has a matching object property *takes*. The class nodes $S_1.P$ and $S_1.M$ linked by this property map to the corresponding class nodes in Q_1 (i.e. to $Q_1.P$ and $Q_1.M$). The functional data-type properties of the concepts *Patient* and *Medication* are projected by S_1 (i.e. they correspond to distinguished variables in S_1). Therefore S_1 is considered as covering the object property *takes*. The covered property *takes*($Q_1.M$, $Q_1.P$)is inserted in the mapping table (Table 2). The same discussion applies to S_2.
- Service S_3- S_3 has a class node $S_3.P$ that can be matched with $Q_1.P$. All the data-type properties of $Q_1.P$ that bound to distinguished variables in Q_1 also bound to distinguished variables in S_3. Furthermore, $Q_1.P$ is involved in object properties in Q_1. However, S_3 has the functional property *hasSSN* of *Patient* bound to a distinguished variable in its RDF view. Therefore, S_3 can be used to cover $Q_1.P$.
- Service S_4- has a matching object-property *hasTest*. The class-nodes linked by *hasTest* $S_4.P$ and $S_4.T$ map to the corresponding classes in Q_1 ($Q_1.P$ and $Q_1.T$). S_4 binds the functional properties of *Patient* (i.e. *hasSSN*) to distinguished variables. The properties *type* and *hasValue* of *Test* are not functional. Therefore, S_4 must also cover the class-node $Q_1.T$, which is possible. The covered class-node and object-property are inserted in the partial mapping table.
- Services S_8 and S_9- S_8 in Figure 3 has a matching object-property *hasPatientPrivacyPreferences*. The

Service	Covered classnodes & properties
$S_1(\$x, ?y)$	$takes(P, M)$
$S_2(\$x, ?y)$	$takes(P, M)$
$S_3(\$y, ?w, ?z)$	$P(y, w, z)$
$S_4(\$y, `cholesterol', ?v)$	$hasTest(P, T)T('cholesterol', v)$
$S_8(\$y, `Researcher',$ $`Research'\ ?w_1, ?z_1)$	$hasPatientPrivacyPreferences(P, PP)$ $PP(`Researcher', `Research',\ w_1, z_1)$
$S_9(\$y, `Researcher',$ $`Research'\ ?v_1)$	$hasTestPrivacyPreferences(P, TP)$ $TP(`Researcher', `Research',\ v_1)$

Table II
MAPPING TABLE

class-nodes linked by this property map to the corresponding class-nodes in Q_1. S_8 does not bind the functional properties of the concept *PatientPrivacyPreferences* to distinguished variables and therefore it has to cover the class-node $Q_1.PP$ as well which is possible. The same discussion applies to S_9 in Figure 3 with replacing the object property *hasPatientPrivacyPreferences* by *hasTestPrivacyPreferences* and the class-node $Q_1.PP$ by $Q_1.TP$.

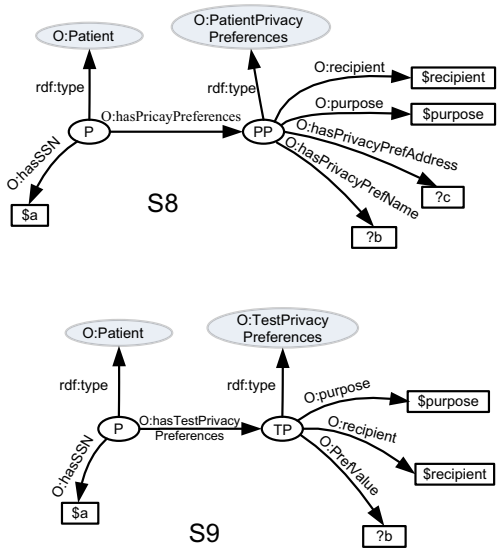

Figure 3. Two Additional DaaS Services

2) Phase-II: Generating DaaS service Compositions: After the construction of the mapping table in the previous phase, the mashup system explores the different combinations from that table. It considers the combination of disjoint sets of covered object properties and class nodes. A combination is said to be a valid rewriting of Q (also a valid composition) if (*1*) it covers the whole set of class-nodes and object-properties in Q, and (*2*) it is executable. A composition is said to be executable if all input parameters necessary for the invocation of its component services are bound or can be made bound by the invocation of primitive services whose input parameters are bound.

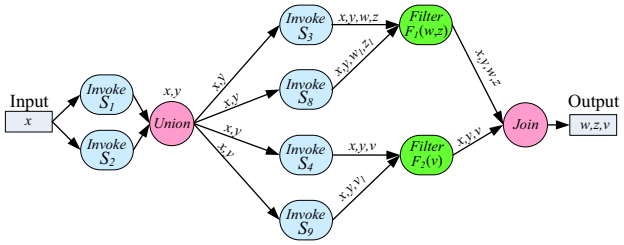

Figure 4. The Data Mashup's Plan

Example- Continuing with the running example, there are two possible combinations $C_1 = \{S_1, S_3, S_4, S_8, S_9\}$ and C_2 $\{S_2, S_3, S_4, S_8, S_9\ \}$. Let us now consider the combination C_1; only $S_1(\$x, ?y)$ can be invoked at the beginning as its input parameter is bound. After the invocation of S_1, the variable y become available; hence, the services S_3, S_4, S_8, S_9 become invokable. Consequently C_1 is executable and is considered as a valid composition. The same applies to C_2.

C. Constructing the Mashup

1) Arranging Services in the Mashup: Component services in a composition must be mashed up in a particular order depending on their access patterns (i.e. the ordering of their inputs and outputs). If a service S_j has an input x that is obtained from an output y of S_i then S_j must be preceded by S_i in the mashup plan; we say that there is a dependency between S_i and S_j (S_j depends on S_i). We define a dependency graph as a directed acyclic graph G in which nodes correspond to services and edges correspond to dependency constraints between component services. The mashup plan must reflect G. Figure4 shows the mashup plan for C_1 and C_2 (they are superposed); there is a dependency constraint between the service S_1 and all of the services S_3, S_8, S_4 and S_9, therefore these later services are preceded by S_1 in the plan (the same applies between S_2 and the services S_3, S_8, S_4 and S_9).

2) Enforcing Privacy Constraints: In previous steps, the datatype properties that participate in validating privacy constraints were projected out along with the initial data items requested in the original mashup query Q. In this step we augment the mashup plan with privacy filters that take into account the values of these additional datatype properties to evaluate the privacy constraints for individual datatype properties that are subject to privacy constraints in the initial query. Null values will be returned for datatype properties whose privacy constraints evaluate to False. Privacy filters are added on the outputs of services returning some privacy sensitive data. The semantics of a privacy filter is defined as follows:

Let t (resp., t_p) be a tuple in the output table T (resp., T_p) of a service S returning some privacy sensitive data, $t[i]$ and $t_p[i]$ be the projected datatype properties that are subject to privacy constraints, and $constraint(t[i])$ be a boolean function that evaluates the privacy constraints associated with $t[i]$. A tuple t_p is inserted in T_p as follows:

For each tuple $t \in T$
  For i = 1 to n / n is the number of columns in T */*
    if const(t[i]) = true Then t_p *[i] = t[i]*
      else t_p [i] = null
Discard all tuples that are null in all columns in T_p

Continuing with our running example, as Figure 4, two privacy filters F_1 and F_2 are added on the outputs of the services S_3 and S_4 respectively. The filter F_1 computes the values of w and z as follows:

w = w if w_1 = 'yes', otherwise w = null

z = z if z_1 = 'yes', otherwise z = null

The filter F_2 computes the values v as follows:

v = v if v_1 = 'yes', otherwise v = null

The obtained mashup plan after the insertion of privacy filters represents the data mashup that will be returned to the user.

IV. EVALUATION

To illustrate the viability of our approach to data mashup, we applied it to the healthcare domain. We were provided with access to /411/ medical Web services defined on top of /23/ different medical databases storing medical information (e.g. diseases, medical tests, allergies, etc) about more than /30,000/ patients. The usage of these medical data services was conditioned by a set of /47/ privacy and security rules. For each patient in these databases, we have randomly generated data disclosure preferences with regard to /10/ medical actors (e.g. researcher, physician, nurse, etc) and different purposes (e.g., scientific research). These preferences are stored in an independent database and accessed via 10 Web services, each giving the preferences relative to a particular type of medical data (e.g., ongoing treatments, Allergies).

We conducted a set of experiments to measure the cost incurred in privacy preservation. We considered two sets of mashup queries. The first one included queries about a given patient, each with a different size: Q_1 requests the "Personal information" of the patient Alice (1 class-node in the query graph), Q_2 requests the "Personal information", "Allergies" and "Ongoing Treatments" of Alice (3 class-nodes), and Q_3 requests the "Personal information", "Allergies", "Ongoing Treatments", "Cardiac Conditions" and "Biological Tests" of Alice. (5 class-nodes). The second set uses the same queries Q_1, Q_2 and Q_3 but for all of patients living in Lyon. All queries were posed by the same actor (researcher) and for the same purpose (medical research). Figure 5 depicts the results

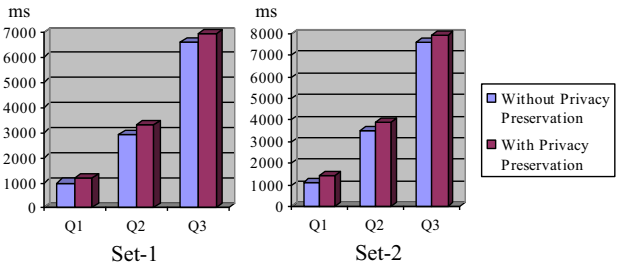

Figure 5. The Experimental Results

obtained for the queries in sets 1 and 2,(the time shown includes both the mashup construction time and the mashup execution time). Set-2 (as opposed to Set-1) amplifies the cost incurred by set-1 at the mashup "*execution phase*" by a factor equals to the number of returned patients. The results for Set 1 show that privacy handling adds only a slight increase in the query rewriting time (note that the mashup execution time is neglected for one patient). This is due to the fact that the number of services used to retrieve privacy preferences is limited compared to the number of services used to retrieve data (10 versus 411 in our experiments). The results for Set 2 show that the extra time needed to handle privacy in the added privacy filters is still relatively low if compared to the time required for answering queries without addressing privacy concerns.

V. RELATED WORKS

A. Mashup Systems and Tools

Several mashup editors have been introduced by the industry with the objective of making the process of mashups creation as simple and "programmable-free" as possible. Examples include Yahoo Pipes [6], Google Mashup Editor [7], Intel Mash Maker [8]. These products allow average users to create mashups without any programming involved; the users need just to drag and drop services, operators, feeds and/or user inputs and to visually connect them. However, the knowledge required from users is not trivial because they are still expected to know exactly what the mashup inputs and outputs are, and to figure out all the intermediate steps needed to generate the desired outputs from the inputs. This includes selecting the needed services/data sources, mapping their inputs and outputs to each other and probably adding some mediation services/functions when inputs/outputs don't fit each other. Compared to these industrial mashup editors and to other academic mashup systems [9], [10], [11], users of our system are not required to select the services manually, connect them to each other and drop code (in JavaScript) to mediate between incompatible inputs/outputs of involved services. This is completely carried out by the system in a transparent fashion. That is, our approach is *declarative*; users need just to specify the

information they need without specifying how this information is obtained. Furthermore, although data privacy and security are two crucial issues that must be addressed in data integration applications, these approaches have not, as far as we are aware, provided mechanisms and solutions to address data privacy and security concerns, whereas in our data mashup system data privacy and security are considered as central issues.

B. Web Service Composition

A considerable body of recent work addresses the problem of composition (or orchestration) of multiple Web services to carry out a particular business task, e.g. [12], [13], [14]. However, these works consider only *SaaS Web services* (*Software-as-a-Service Web services*) and focus only on describing workflow-oriented applications, rather than applications coordinating data obtained from multiple data sources exported as Web services as addressed in this paper. In these approaches, the exploited composition algorithms (which are largely inspired by AI planning techniques) regard services as *actions* and therefore assume that the capability of a Web service (i.e. a SaaS Web service) can be modeled by representing the service's inputs, outputs, preconditions and effects (IOPEs) [15]. This assumption makes these approaches inapplicable to DaaS Web services whose capabilities (i.e. semantics) can only be represented by capturing the semantic relationships between the service's inputs and outputs in relation with the schemes of underlying data sources. In contrast, we model services are RDF views over domain ontologies to capture the semantic relationships between their inputs and outputs sets. We exploit these views to mashup available DaaS Web services on the fly. Our solution can be applied to both types of Web services (i.e. SaaS and DaaS services).

VI. CONCLUSION

In this paper, we have presented a declarative and privacy preserving approach to mashup data Web services on the fly while preserving data privacy. We modeled data Web services as parameterized RDF views over domain ontologies; defined views are then used to annotate the service descriptions files (e.g. WSDLs files). We proposed to use query rewriting techniques to rewrite data mashup queries in terms of available data Web service. Specifically, mashup queries are first modified to accommodate data privacy constraints from privacy policies; then are rewritten in terms of available services using an RDF-oriented query rewriting algorithm. We applied the proposed approach to mashup /411/ data Web services from the healthcare application domain; the obtained results are very promising. As a future work, we intend to test the proposed approach in different application domains like the e-Government and e-Tourism.

REFERENCES

[1] H. L. Truong and S. Dustdar, "On analyzing and specifying concerns for data as a service," in *APSCC*, 2009, pp. 87–94.

[2] M. J. Carey, "Declarative data services: This is your data on soa," in *IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2007*, 2007, p. 4.

[3] D. Butler, "Mashups mix data into global service," in *Nature*, January 2006.

[4] A. Jhingran, "Enterprise information mashups: Integrating information, simply," in *VLDB*, 2006, pp. 3–4.

[5] S. S. Bhowmick, L. Gruenwald, M. Iwaihara, and S. Chatvichienchai, "Private-iye: A framework for privacy preserving data integration," in *ICDE Workshops*, 2006, p. 91.

[6] Yahoo inc. yahoo pipes. [Online]. Available: http://pipes.yahoo.com/pipes/

[7] Google inc. google mashup editor. [Online]. Available: http://code.google.com/gme/

[8] Intel. intel mash maker. [Online]. Available: http://mashmaker.intel.com/web/

[9] J. Tatemura, S. Chen, F. Liao, O. Po, and D. Agrawal, "Uqbe: uncertain query by example for web service mashup," in *SIGMOD Conference*, 2008, pp. 175–180.

[10] J. Tatemura, A. Sawires, O. Po, S. Chen, D. Agrawal, and M. Goveas, "Mashup feeds: : continuous queries over web services," in *SIGMOD Conference*, 2007, pp. 128–130.

[11] A. H. H. Ngu, M. P. Carlson, Q. Z. Sheng, and H. young Paik, "Semantic-based mashup of composite applications," *IEEE T. Services Computing*, vol. 3, no. 1, pp. 2–15, 2010.

[12] T. Weise, S. Bleul, D. E. Comes, and K. Geihs, "Different approaches to semantic web service composition," in *Third International Conference on Internet and Web Applications and Services, ICIW 2008*, 2008, pp. 90–96.

[13] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed, "Deploying and managing web services: issues, solutions, and directions," *VLDB J.*, vol. 17, no. 3, pp. 537–572, 2008.

[14] M. A. Eid, A. Alamri, and A. El-Saddik, "A reference model for dynamic web service composition systems," *IJWGS*, vol. 4, no. 2, pp. 149–168, 2008.

[15] D. Martin, M. Paolucci, and M. Wagner, "Bringing semantic annotations to web services: Owl-s from the sawsdl perspective," in *ISWC/ASWC*, 2007, pp. 340–352.

[16] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and randomized optimization for the join ordering problem," *VLDB J.*, vol. 6, no. 3, pp. 191–208, 1997.

[17] M. Barhamgi, D. Benslimane, and B. Medjahed, "A query rewriting approach for web service composition," *IEEE T. Services Computing*, vol. 3, no. 3, pp. 206–222, 2010.