A Query Rewriting Approach for Web Service Composition

Mahmoud Barhamgi, Djamal Benslimane, and Brahim Medjahed, *Member*, *IEEE*

Abstract—Data-Providing (DP) services allow query-like access to organizations' data via web services. The invocation of a DP service results in the execution of a query over data sources. In most cases, users' queries require the composition of several services. In this paper, we propose a novel approach for querying and automatically composing DP services. The proposed approach largely draws from the experiences and lessons learned in the areas of service composition, ontology, and answering queries over views. First, we introduce a model for the description of DP services and specification of service-oriented queries. We model DP services as RDF views over a mediated (domain) ontology. Each RDF view contains concepts and relations from the mediated ontology to capture the semantic relationships between input and output parameters. Second, we propose query rewriting algorithms for processing queries over DP services. The query mediator automatically transforms a user's query (during the query rewriting stage) into a composition of DP services. Finally, we describe an implementation and provide a performance evaluation of the proposed approach.

Index Terms—Services integration framework, advanced services invocation framework, services delivery platform, composite web services.

1 Introduction

Recent years have witnessed a growing adoption of web services as a medium for enabling cross-organizational collaborations on the web [29]. Modern enterprises are increasingly embracing the service-oriented paradigm to provide interoperable and programmatic interactions with their internal systems [28]. Such interactions are generally performed via two types of services: *Effect-Providing* (EP) services and *Data-Providing* (DP) services. *EP services* implement organizations' business functions. The execution of an EP service produces effects that may change the state of the world. For instance, a book-selling service has as effects charging the customer's credit card and physical transfer of the book from the bookstore's warehouse to the customer's address. *DP services* allow query-like access to organizations' data sources. The invocation of a DP service results in the execution of a query over the data sources' schema [6], [32], [1]. However, in contrast to EP services, such invocation has no effect on the state of the world. For instance, a pharmaceutical DP service may return the generic equivalent of a given brand medication; a homology search DP service scours a DNA sequence database to find sequences that have a common ancestor with a given sequence.

DP services provide bridges to access (or query) enterprise data sources [6], [32], [1]. However, in most cases, users' queries require the invocation of several services. For instance, let us consider the following query: "*what are the tests performed in ABC Lab by patients who have been administered Glucophage in XWZ hospital?*" Let us assume that ABC Lab and XWZ hospital provide two DP services S_{ABC} and S_{XYZ}, respectively: S_{ABC} returns the tests performed by a given patient in ABC Lab and S_{XWZ} returns the list of patients that have been administered a given drug in XWZ hospital. The execution of the above-mentioned query involves the *composition* of S_{ABC} and S_{XYZ} services. Web service composition is a powerful solution for building value-added services on top of existing ones [46], [23].

Composing web services is not a new problem; significant research has been devoted to service composition in the past years [14], [41], [42], [45], [5], [23], [24]. However, current approaches mostly focus on EP services and hence are not directly applicable to DP services [20], [22]. The capabilities of EP services were modeled in the aforementioned approaches through 1) *input*, *output*, *effects*, and *preconditions* (a.k.a. IOPE) and/or 2) concepts defined within a taxonomy/ontology (e.g., RosettaNet) enumerating all potential actions/functions that may exist in a particular domain. Both ways are inappropriate to represent the capability of DP services. Indeed, DP services have no preconditions and produce no effects, and as a result, representing them as state transformation (i.e., via their IOPEs) would be reduced to representing their input and output. However, DP services may have similar input and output, yet completely different semantics. For example, two services may have the same input and outputs, say a *Drug*, the first returns the drugs that would interact with the given one whereas the second retrieves the drugs that are equivalent to a given one. That is, the semantics of a DP service should capture the way input parameters relate to output parameters, i.e., the *input/output semantic relationship*. Assuming a domain ontology, the relationships in the previous example might be defined by relations like, for example, "*isEquivalentTo*," "*interactsWith*" that relate the concept "*Drug*" to itself in the ontology. It should be noted that input/output semantic relationships may be complex and, as such, cannot be conveniently

- *M. Barhamgi and D. Benslimane are with the LIRIS Lab, Claude Bernard University Lyon1, LIRIS, Bâtiment Nautibus, 8 Bd Niels Bohr, 69100 Villeurbanne, France. E-mail: {mahmoud.barhamgi, djamal.benslimane}@liris.cnrs.fr.*
- *B. Medjahed is with the Department of Computer and Information Science, University of Michigan-Dearborn, 4901 Evergreen Road, Dearborn, MI 48128. E-mail: brahim@umd.umich.edu.*

Manuscript received 11 June 2009; revised 15 Oct. 2009; accepted 20 Dec. 2009; published online 11 Feb. 2010.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSCSI-2009-06-0159. Digital Object Identifier no. 10.1109/TSC.2010.4.

represented by exhaustively enumerating them in taxonomies/ontologies as is done for actions/functions in the EP services' case. Consider the example of DP services that return the medications administered to a given patient; one service may specify that the medications it returns are prescribed by doctors, as opposed to over-the-counter medications that can be bought without prescriptions. Input and output—the concepts "*Patient*" and "*Medication*" here—are not only linked via a relation such as "*takes*," but also via additional concepts like "*Doctor*" that is linked to the input via a relation like "*treats*" and to the output concept via a relation like "*prescribedBy*."

In this paper, we propose a novel approach for querying DP services. The proposed approach largely draws from the experiences and lessons learned in the areas of service composition, ontology, and answering queries over views. It assumes the existence of a mediated ontology (MO) specified in Resource Description Framework (RDF) to capture consensual and shared knowledge in a given domain (e.g., healthcare). RDF is a W3C-supported language for ontology representation [11]. The contributions of this paper are summarized below:

- *Query Model for DP Services*—We introduce an RDF-based model for the 1) description of DP services and 2) specification of service-oriented queries. We model DP services as *RDF views* over domain ontologies. Input/output relationships are declaratively represented based on concepts and relations that are semantically defined in a mediated ontology. We adopt SPARQL language for posing queries over DP services.
- *Processing DP Service Queries*—We propose *query rewriting* algorithms for processing queries over DP services. The idea behind query rewriting is the following: given a query over the mediated ontology and a set of RDF views of DP services, reformulate the query into an expression that refers only to the RDF views and provides the answer to the query. The proposed approach automatically transforms a user's query (during query rewriting) into a composition of DP services (modeled as RDF views) that are selected, orchestrated, and invoked to execute the posed query.
- *Implementation and Evaluation*—We describe an implementation and provide a performance evaluation of the proposed approach.

The rest of this paper is organized as follows: In Section 2, we motivate the need for querying DP services; discuss the underlying challenges, and overview the proposed approach. In Section 3, we describe our query model for DP services. In Section 4, we propose a query rewriting approach for processing queries over DP services. In Section 5, we extend our approach to handle parameterized queries. In Section 6, we describe our implementation and evaluate our approach. We overview related work in Section 7. We provide concluding remarks in Section 8.

2 Querying DP Services: Motivation and Challenges

In this section, we first describe a motivating scenario from the e-prescription domain. Then, we discuss the challenges to be addressed for querying DP services and give an overview of the proposed approach.

TABLE 1
DP Services in the E-Prescription Reference Scenario

Service	Functionality	Constraints
S_1($a, ?b)	Returns medications (***b***) taken by a given patient (***a***)	a≥x888
S_2($a, ?b)	Returns medications (***b***) taken by a given patient (***a***)	a≤x888
S_3($a, ?b)	Returns medications (***b***) that interact with a given one (***a***)	
S_4($a, ?b, ?c)	Returns information (***b*** and ***c***) about a given medication (***a***)	a≥p660
S_5($a, ?b, ?c)	Returns information (***b*** and ***c***) about a given medication (***a***)	a≤x8999
S_6($a, ?b)	Returns medications (***a***) equivalent to a given one(***b***)	

2.1 The E-Prescription Scenario

As a running scenario, let us consider the e-prescription system that provides access to the set of services described in Table 1. S_1 and S_2 return the medications taken by a given patient identified by a Social Security Number (or SSN). Both services have a similar signature and semantics but impose different constraints on their inputs/outputs as specified in the "Constraints" column of Table 1. S_4 and S_5 return information (name and URL to more details) about a given medication; S_4 gives information about medication with a code higher than "p660" while S_5 gives information about medications with a code lower than "x8999."

Let us now assume that a physician Alice would like to submit the following query Q_1: "check whether the drug d identified by code "x9999" to be prescribed to patient John interacts with the ones currently taken by that patient." Alice uses the services described in Table 1 to obtain such information. As depicted in Fig. 1a, Alice invokes S_1 to retrieve the list of drugs (called L_1) currently taken by John (step a.1); this assumes that John's SSN satisfies the condition "a≥ x888." Alice also invokes S_3 to get the list of drugs that interact with d (step a.2). We refer to the list of drugs returned in step a.2 as L_2. Note that steps a.1 and a.2 may be executed in parallel. The result set of Q_1 contains the drugs that belong to $L_1 \cap L_2$ (step a.3). Alice may get additional information about d and the drugs in the result set by invoking S_4 and S_5 services (step a.4); if a drug's code verifies the condition "a≥ p660," then S_4 is invoked; if the code verifies the condition "a≤ x8999," then S_5 is invoked.

To make the scenario even more challenging, Alice may obtain the results of query Q_1 in a different way as depicted in Fig. 1b. Steps b.1 and b.4 are similar to steps a.1 and a.4, respectively. For each drug in L_1, Alice invokes S_3 to update the list L_2 of drugs that interact with those in L_1 (step b.2). L_2 contains tuples (A: taken drug, B: interacting drug). Alice finally needs to filter out L_2 and look for tuples whose second part B refers to d (step b.3).

2.2 Challenges

Our reference scenario shows that *Alice* needs to perform several tasks to execute her query. These tasks may

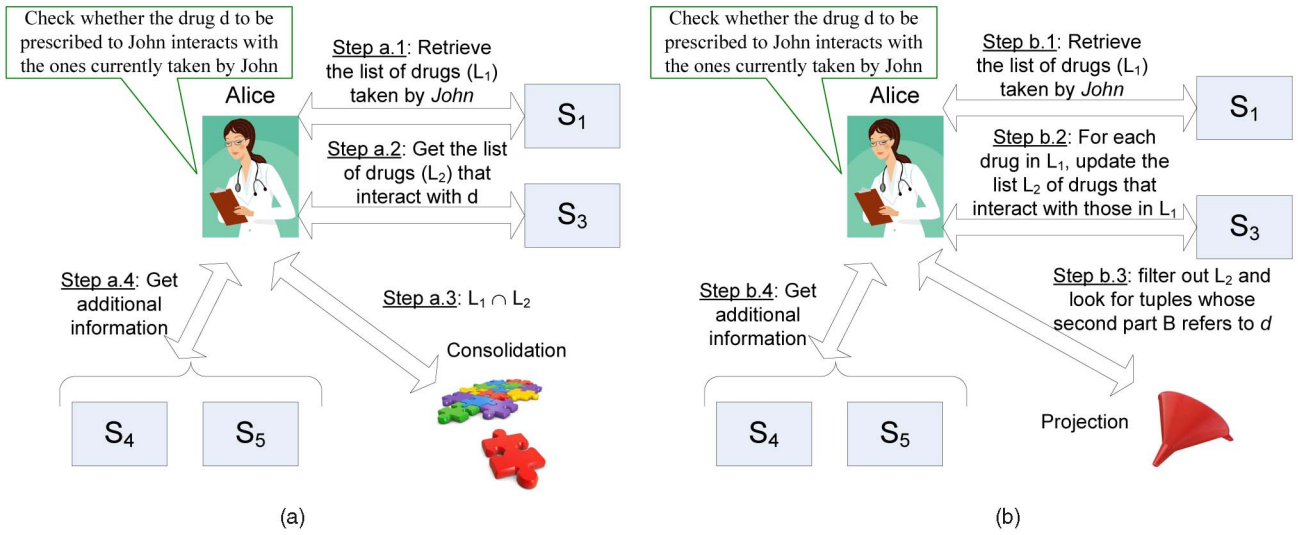

Fig. 1. The e-prescription reference scenario.

especially be tedious in application domains such as bioinformatics where the last count of DP services approximates three thousands [20]. First, *Alice* needs to understand the semantics of the existing services and the relationships between the input and output parameters of each DP service. For example, S_3 and S_6 have the same input and output parameters (i.e., drug). However, S_3 returns the medications that interact with a given drug d whereas S_6 returns the medications that are equivalent to d. Unfortunately, *Alice* will not be able to differentiate between these two services as current web service description languages provide little or no support for declaring input/output semantic relationships. Second, *Alice* needs to manually select the services that are relevant to her query and invoke them in the right order. For instance, she has to figure out the execution plan for her query (Fig. 1a or 1b). Third, *Alice* needs to consolidate results and manually perform potential joins as mentioned in step 3 of the scenario. The above-mentioned tasks fall under the following two challenges: developing 1) a query model for DP services and 2) techniques for processing queries over DP services.

2.2.1 Developing a Query Model for DP Services

The *query model* allows the declarative specification of DP service semantics as well as queries over DP services. Semantic web services are usually modeled using languages such as WSMO [33], OWL-S [22], and WSDL-S [36]. A survey of major service description languages is given in [45]. For example, OWL-S's Service Profile permits the modeling of the service input, output, the preconditions for invoking the service and the produced effects. It also allows the categorization of services according to their functionality and domain. While such semantic properties are important for describing web services in general, they are not sufficient in the case of DP services. DP services are mostly concerned with retrieving the appropriate outputs given specific inputs. They do not provide any functionality, beyond retrieval, and have no external effects. However, it is important to capture the semantic relationship that may exist between the inputs and outputs of DP services. For instance, S_3 and S_6 have the same input and output sets but different semantics linking the two sets ("interacts with" and "is equivalent to"). In this paper, we propose an RDF-based model for DP services. We represent DP services as *RDF views* over a mediated ontology. Each RDF view contains concepts and relations from that ontology. We adopt SPARQL, the *de facto* query language for RDF, for posing queries over DP services.

2.2.2 Developing Techniques for Processing Queries over DP Services

Users should be relieved from the burdensome task of selecting, composing, and invoking DP services. Given a query specified according to our query model, a *query mediator* will automatically execute the query by selecting and orchestrating the right DP services. DP services should be treated as first-class objects that could be transparently selected, invoked, and composed much like database tables are transparently located, queried, and joined in traditional DBMS systems [45]. The query mediator proposed in this paper adopts a *query rewriting* approach: given a SPARQL query specified over the mediated ontology and a set of RDF views corresponding to DP services, reformulate the query into an expression that refers only to the RDF views and provides the answer to the query. The query mediator automatically transforms a user's query (during the query rewriting stage) into a composition of DP services (modeled as RDF views) that are selected, orchestrated, and invoked to execute the posed query.

2.2.3 Overview of the Proposed Approach

Fig. 2 gives an overview of the proposed approach. DP services are modeled as RDF Views over a mediated ontology. RDF views capture the semantic relationships between input and output parameters using ontological concepts defined in the mediated ontology. They are incorporated within WSDL description files as annotations. Users pose their queries over the mediated ontology using SPARQL query language. The DP service query mediator uses an RDF query rewriting module and the existing RDF

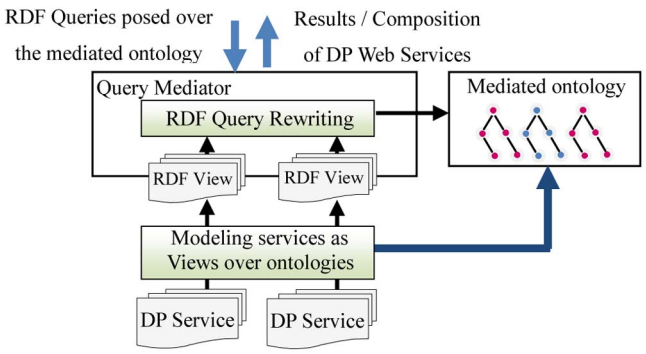

Fig. 2. Overview of the proposed approach.

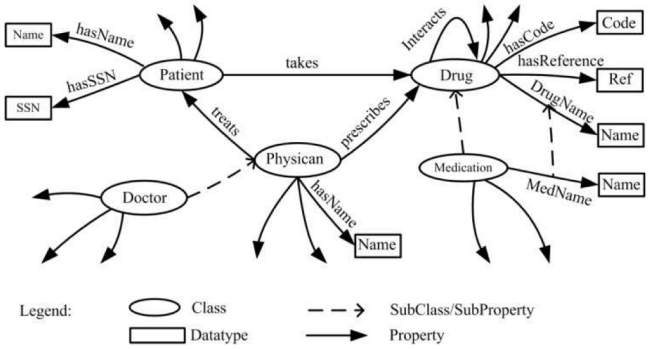

Fig. 3. Example of mediated ontology.

views to select the services that can be combined to answer the posed query. Then, it generates a composite service as an execution plan for the query, execute the composite service, and returns data to the user. The generated composite service may also be deployed as a new DP service and used to answer subsequent parameterized queries.

3 The DP Service Query Model

In this section, we describe the proposed model for DP service queries. We first introduce some preliminaries. Then, we use RDF views to describe DP services. Finally, we present preprocessing steps that need to be performed prior to query resolution.

3.1 Preliminaries

The model presented in this section relies on three major concepts: mediated ontology, query, and query containment.

3.1.1 Mediated Ontology

In the proposed approach, users formulate their queries against an MO. Such ontology is defined by domain experts and specified in RDF/RDFS. Formally, MO is defined by the 6-tuple (C, D, TP, OP, SC, SP). C is a set of classes. D is a set of data types. TP is a set of data-type properties; each TP property has a domain in C and a range in D. OP is the set of object properties; each OP property has its domain and range in C. SC is a relation over $C \times C$, representing the subclass relationship between classes. SP is a relation over $(OP \times OP) \cup (TP \times TP)$; it represents the subproperty relationship between properties in OP or properties in TP. Fig. 3 depicts a portion of the MO ontology in the healthcare domain—the employed representation is very similar to ER diagrams; the sole difference is that subclassing relationships SC and SP (denoted by dashed arrows) are distinguished from other ordinary relations. Examples of concepts include *Patient* and *Drug*. This ontology specifies that patients take drugs. Drugs have different characteristics like name, a universal code, and a URL reference to detailed information about it. Drugs may interact with each other and have specializations (e.g., Medications).[1] Concepts are interlinked by object properties; they are related to data types via data-type properties.

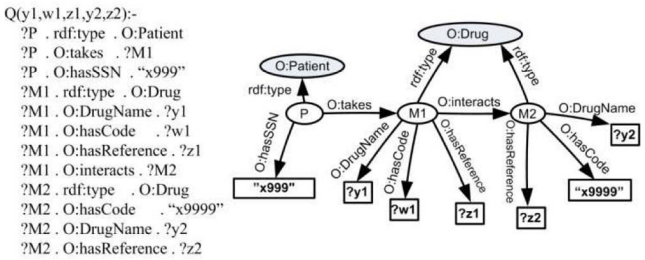

Fig. 4. SPARQL specification and graph 1.

3.1.2 Queries

We consider conjunctive queries over MO. Queries are expressed using SPARQL, the de facto query language for the Semantic Web.[2] Formally, a query Q has the form $Q(\overline{X}) :- G(\overline{X},\overline{Y})$. $Q(\overline{X})$ 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). The body may also contain constraints on the body variables of the form: $x\Theta$ *Constant* where $\Theta \in \{>, <, \geq, \leq\}$. This type of constraints is added to represent potential *equality and order constraints* that may be placed on the query's variables (e.g., age >50, name = "John Smith," etc.). Fig. 4 gives the SPARQL and graphical representations of the query Q_1 described in our e-prescription scenario. A query can be seen as a graph with two types of nodes: class and literal nodes. Class nodes refer to classes in the MO ontology (e.g., M_1 and M_2). They are linked via object properties and represent existential variables in the query. Literal nodes represent data types (e.g., y_1, w_1, z_1). They are linked with class nodes via data-type properties. Literal nodes may correspond to both existential and distinguished variables in a query.

3.1.3 Query Containment

Query containment enables the comparison between queries [17]. A query Q_1 is said to be *contained* in Q_2, denoted by $Q_1 \subseteq Q_2$, if and only if the answer to Q_1 is a subset of the answer to Q_2 for any *knowledge base D*. A knowledge base [4] is the union of a set of axioms defining the ontology at the class-level (a.k.a. *TBox*) and a set of facts defining the instances of the ontology's classes (a.k.a.

1. Medication refers to all medicines that are legally obtained for a therapeutic reason. Drug refers to all medicines used legally or illegally.

2. http://www.w3.org/blog/SW/2008/01/15/sparql_is_a_recommendation.

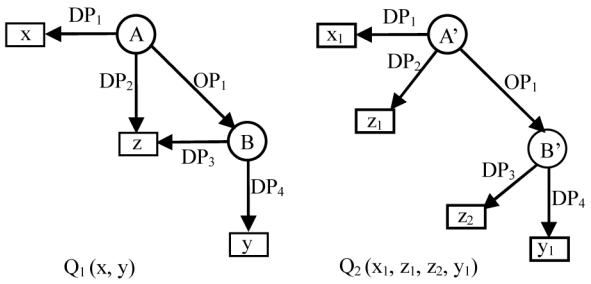

Fig. 5. Example of query containment.

ABox). Q_1 and Q_2 are *equivalent* if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$. *Containment mapping* has been used in data integration as the necessary and sufficient condition for testing query containment [8]. A query Q_1 contains Q_2 if and only if there is a *containment mapping* from Q_1 to Q_2. A mapping β from the variables of Q_1 to the variables of Q_2 is a containment mapping if 1) β maps every class node in Q_1 to a class node in Q_2, 2) all object properties in Q_1 hold between the mapped class nodes in Q_2, and 3) β maps the head of Q_1 to the head of Q_2. In Fig. 5, $Q_2(x_1, z_1, z_2, y_1)$ can be made contained in $Q_1(x,y)$ by the following containment mapping β: $A \rightarrow A'$, $B \rightarrow B'$, $x \rightarrow x_1$, $z \rightarrow z_1$, $z \rightarrow z_2$, $y \rightarrow y_1$. Indeed, all class nodes in Q_1 were mapped to the corresponding class nodes in Q_2. The object properties in Q_1 hold between the corresponding classes in Q_2. The join between the class nodes A and B over the variable z was enforced in Q_2 since Q_2 projects out the data-type properties DP_2 and DP_3; these two data-type properties are bound to the distinguished variables z_1 and z_2 in Q_2. Applying the mapping containment β to Q_2 results in $Q_2(x,z,z,y)$.

3.2 DP Service = RDF Parameterized View

We model DP services as RDF Parameterized Views (RPVs) over the MO ontology. As mentioned in Section 2.2, RPVs use concepts and relations from the MO ontology to capture the semantic relationships between input and output sets of a DP service. 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_3 from above without specifying a medication for which it is needed to learn the interacting medications. Therefore, a parameterized view must indicate which parameters are inputs, and which parameters are outputs. Parameterized views have been used to describe content and access methods in Global-as-View (GaV) integration architectures [17]. However, to the best of our knowledge, this work is the first to use parameterized views to model DP services.

A parameterized view can be seen as a parameterized SPARQL query. Formally, an RPV of a DP service S_i over an MO ontology is a predicate $S_i(\overline{\$X_i},\overline{?Y_i}) :- < \Phi(\overline{X_i},\overline{Y_i},\overline{Z_i}), Ct_i >$ 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*.
- $\Phi(\overline{X_i},\overline{Y_i},\overline{Z_i})$ represents the semantic relationship between input and output variables. $\overline{Z_i}$ is the set of existential variables relating $\overline{X_i}$ and $\overline{Y_i}$. Φ has the form of RDF triples where each triple is of the form (subject.property.object).
- Ct_i are the constraints imposed on $\overline{X_i}$, $\overline{Y_i}$, or $\overline{Z_i}$ variables. A constraint has the form: $x\Theta$ *Constant* where $\Theta \in \{>, <, \geq, \leq\}$.

Fig. 6 gives the RPVs of the DP services depicted in Table 1. Each RPV view is characterized by an access pattern; an access pattern specifies whether a parameter is input or output. The input and output variables are prefixed with the symbols "$"and "?," respectively. For example, the access pattern of the S_1 is ($a,?b). Other services may have the same body as S_1 but with different access patterns such as (?a,?b), ($a,$b), and (?a,$b).

RPVs support semantic reasoning while matching a service description with a query. Consider, for example, the case of a DP service S with an input parameter of type *Geographic-Region* and output parameter of type *Drug*; S returns the list of drugs that are produced in a given region. The RPV of S will express that "*the returned drug d is produced by a pharmaceutical company c located in the region r*." Let us assume that the user is looking for a service that takes as input the name of a *French-Geographic-Region* and returns as

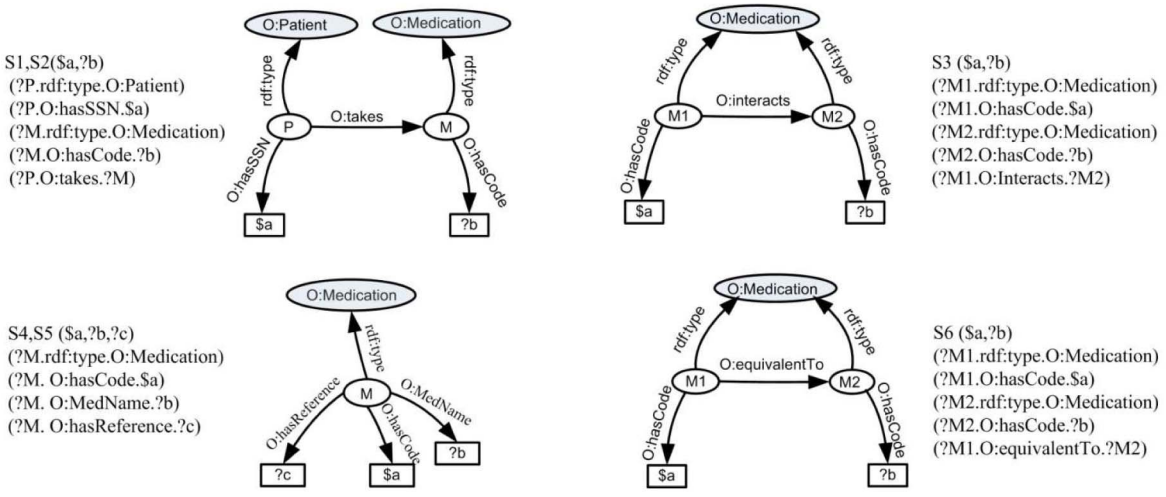

Fig. 6. RDF parameterized views—examples.

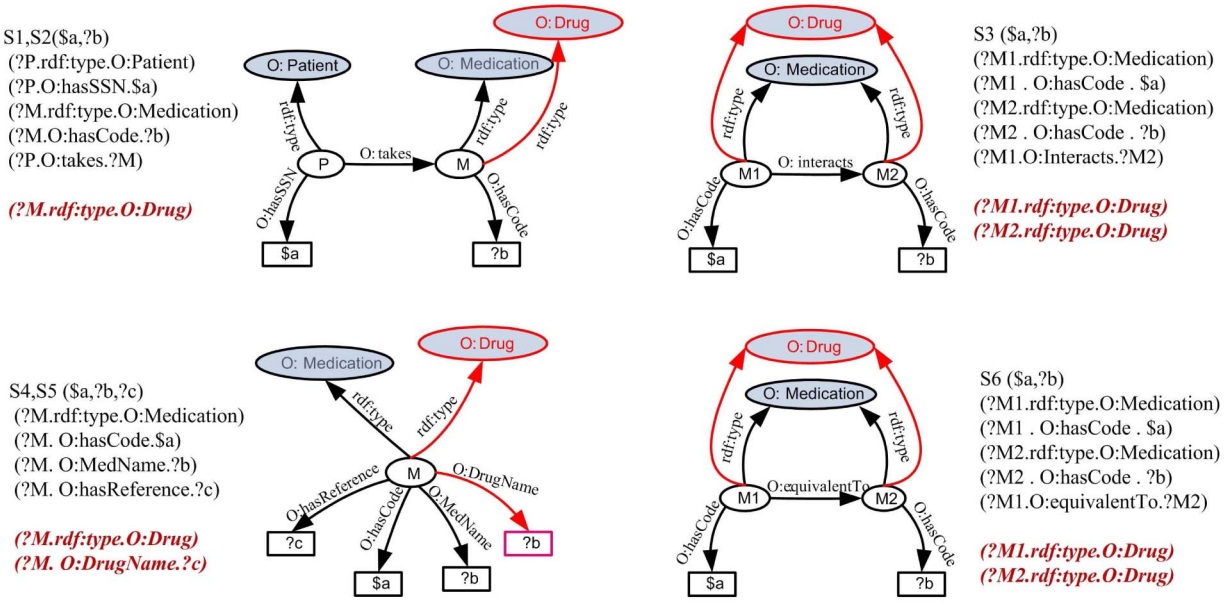

Fig. 7. Applying RDFS constraints to RPVs—example.

output the list of *French-Drug* produced in the region. Assume also that the MO ontology defines the concepts of *French-Drug* as "*a drug produced by a pharmaceutical company located in France*" and the concept of *French-Region* as "*a region that is part of France*." Based on the previous ontological definitions and on the RPV of S, a reasoner may infer that the results returned by invoking S with a *French-Region* parameter are subsumed by the concept *French-Drug* since a pharmaceutical company located a geographic region that is part of France is located in France, thus satisfying the definition of *French-Drug*.

3.3 Preprocessing RDF Parameterized Views

The RPVs associated to DP services are preprocessed (i.e., performed offline) prior to query resolution. Preprocessing enables the query mediator to return more results for a given query; for example, a service like S_4 that returns the details about a given *Medication* will not be returned as a matching service to a query that uses the concept *Drug* to request the same details. The preprocessing phase overcomes the problem based on the subclassing semantic constraints that are defined in the ontology. We identify two preprocessing steps: 1) applying RDFS semantic constraints and 2) skolemizing class nodes.

3.3.1 Applying RDFS Semantic Constraints

In this step, RPVs are extended to take into account the RDFS semantic constraints of the MO ontology. RDFS semantic constraints include: *rdfs:subClassOf*, *rdfs:subPropertyOf*, *rdfs:domain*, and *rdfs:range*. These semantic constraints are defined in the mediated ontology; for instance, the class *Medication* is a subclass of the class *Drug*. Fig. 7 extends the RPVs of Fig. 6 to state that a variable of type *Medication* is also of the type *Drug*. It also states that *MedName* is a subproperty of *DrugName*.

3.3.2 Skolemizing Class Nodes

Variables denoting classes nodes in RDF parameterized views are *skolemized*. By "skolemized," we mean that each variable denoting a class node is conceptually associated with a *skolem function* [27]; such function is helpful to merge class instances stemming from different services. In the RDF data model, a skolem function associated with an RDF class generates a new "*unique value*"—used as an identifier to the class instance (a.k.a. blank nodes)—when invoked with some new values of its arguments; it always returns the "*same value*" whenever it is invoked with the same values of arguments. We used skolem functions to specify the datatype properties that can be used to uniquely identify a class instance. For instance, the variable "?M" (of type *Medication*) is associated with a function F_2(hasCode) to specify that if two instances have the same code then they denote the same medication and thus can be merged. This is somewhat similar to the primary key constraint. The properties of a skolem function for a particular class are chosen by the domain experts. In our scenario, the skolem function associated with variables of type *Patient* is F_1(hasSSN); variables of types *Medication* and *Drug* are associated with the skolem function F_2(hasCode).

4 Processing Queries over DP Services

In this section, we describe our query rewriting algorithms. Given a query Q and a set of DP services represented by their corresponding RPVs $V = v_1, v_2 \ldots, v_i$, the query mediator rewrites Q as a composition of services whose union of RDF graphs (denoted to by G_V) covers the RDF graph of Q (denoted to by G_Q). The term "covers" means that: 1) all class nodes in G_Q are in G_V, 2) all object properties that hold between class nodes in G_Q hold between the corresponding class nodes in G_V, and 3) there is a containment mapping β from $G_Q \rightarrow G_V$ so that:

- β maps class nodes in G_Q to those in G_V, i.e., they have the same class types.
- β maps each literal node in G_Q to a literal node in G_V.
- Distinguished variables in Q are provided by the composition.

The RPV of an individual service may cover only a subgraph of G_Q. Therefore, the answer to Q may only be obtained by examining the various compositions of the different RPVs. The RDF graph of a valid combination has to satisfy the above-mentioned conditions. The construction of the compositions comprises two phases: 1) finding the subgraphs of G_Q that are covered by each RPV and 2) generating the composite service.

4.1 Finding Relevant Subgraphs

In the first phase, the query mediator compares G_Q with every RPV v_i in V and determines the class nodes and object properties in G_Q that are covered by v_i. The mediator 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 part of G_Q. The query mediator considers the following two cases for populating the mapping table:

Case 1 (Covering Class Nodes). Let us assume that v_i has a class node C_v whose type is similar to the type of a class node C_Q in Q. The RPV v_i covers C_Q if the mapped class node C_v verifies the following conditions:

1. If C_Q has a distinguished variable x in Q (i.e., a data-type property of C_Q is bound to a distinguished variable x in Q), then either the same data-type property of C_v is projected[3] in v_i, or it can be recovered because all data-type properties used in the skolem function of C_Q are projected in v_i and thus can be used to recover the missing distinguished variable of C_v.
2. If C_Q has an existential variable x in Q (i.e., a data-type property of C_Q binds to an existential variable x in Q), then one of the following conditions must be true:
   a. The variable x maps to a distinguished variable in v_i, or,
   b. The data-type property that is bound to x can be recovered because the data-type properties used in the skolem function of C_Q are projected in v_i, and thus it is possible to recover the data-type property of C_v that corresponds to x.
   c. All class nodes in Q that have x in their triples are covered in v_i and the join between these class nodes over x is enforced in v_i.
3. If C_Q has a constant in its triples, then either v_i has to project the data-type property of C_v that corresponds to the constant, or such data-type property can be recovered.
4. If C_Q is involved in an object property p in Q, then v_i has either to project the attributes of the skolem function of C_v to enforce the join implied by p, or it has to cover p.

Case 2 (Covering Object Properties). Let us assume that v_i includes an object property p of Q in its definition such that the class nodes linked by p can be mapped to the corresponding class nodes of p in Q (i.e., they have the same types). The RPV v_i covers p if it 1) projects the data-type properties used in the skolem function of each of the class nodes linked by p or 2) covers the class nodes for which it does not project the data-type properties used in its skolem function.

3. The data-type property is bound to a distinguished variable in v_i.

Example. Let us now illustrate the above-mentioned cases using the services and query Q_1 described in our e-prescription scenario. We consider the following candidate services:

- Services S_1 and S_2—The same discussion applies to S_1 and S_2; however, S_2 is eliminated because *John*'s SSN ("x999") is out of the range accepted by S_2. S_1 has a matching object property *takes*. The class nodes $S_1.P$[4] and $S_1.M$ linked by this property map to the corresponding class nodes in Q_1 (i.e., to $Q_1.P$ and $Q_1.M_1$) as required in Case 2. $S_1.M$ is mapped to $Q_1.M_1$ owing to the added triple (in the preprocessing phase) to state that a *Medication* is also a *Drug*. The data-type properties used in the skolem functions of *Patient* and *Drug* (medications can be uniquely identified by their codes based on their data-type property *hasCode*) are projected by S_1 (i.e., they correspond to distinguished variables in S_1). Note that there is a join over the existential variable $Q_1.M_1$ (i.e., $Q_1.M_1$ is the subject of the object property *interacts* in Q_1). Even though S_1 does not contain *interacts* property and does not provide all of the requested data-type properties of $Q_1.M_1$, it provides the codes of $Q_1.M_1$'s instances which can be used to enforce the join over $Q_1.M_1$ and recover the missing data-type properties of G_{Q1} through other services. Therefore, S_1 is considered as covering the object property *takes* (Case 2). The covered property $takes(Q_1.M_1, Q_1.P)$ along with class nodes that are necessary for this partial covering (i.e., $Q_1.P(x)Q_1.M_1(w_1)$) are inserted in the mapping table (Table 2). The corresponding partial containment mapping $\beta : Q_1.P \rightarrow S_1.P, Q_1.M_1 \rightarrow S_1.M, x(\text{"}x999\text{"}) \rightarrow a, w_1 \rightarrow b$ along with the service inputs and outputs are also inserted in Table 2.
- *Service* S_3—S_3 has a matching object property *interacts*. The class nodes linked by this property $S_3.M_1$ and $S_3.M_2$ map to the corresponding classes in Q_1 (i.e., to $Q_1.M_1$ and $Q_1.M_2$). Both class nodes were mapped to $Q_1.M_1$ and $Q_1.M_2$, respectively, owing to the added triple to state that a *Medication* is also a *Drug*. Furthermore, S_3 has the medication codes as a distinguished variable for each of the class nodes $Q_1.M_1$ and $Q_1.M_2$ (Case 2). These codes can be used to enforce the joins over $Q_1.M_1$ and to recover missing data-type properties for $Q_1.M_1$ and $Q_1.M_2$. Therefore, S_3 is considered as covering the object property interacts. The covered RDF subgraph will include the covered property along with class nodes that are necessary for this partial covering (i.e., $interacts(Q_1.M_1, Q_1.M_2)Q_1.M_1(w_1)Q_1.M_2(x9999)$). The associated partial mapping is: $\beta : Q_1.M_1 \rightarrow S_3.M_1, Q_1.M_2 \rightarrow S_3.M_2, w_1 \rightarrow a, x9999 \rightarrow b$ (Table 2).
- *Services* S_4 *and* S_5—The same discussion applies to S_4 and S_5; however, S_5 only covers $Q_1.M_1$ as the value "x9999" specified in Q_1 is out of the range accepted by $S_5.S_4$ has a class node $S_4.M$ that can be matched with $Q_1.M_1$ and $Q_1.M_2$. All

4. We use S.C_i to denote the class node C_i in the RDF graph of S. We also use Q.C_i to denote the class node C_i in the RDF graph of Q.

TABLE 2
Mapping Table for the E-Prescription Scenario

	Service	Variables Mapping	Input	Output	Covered nodes and Object properties
1	S_1("x999",?w_1)	Q_1.P → S_1.P, Q_1.M_1 → S_1.M, "x999" → a, w_1 → b	x	w_1	takes(Q_1.M_1, Q_1.P)Q_1.P("x999")Q_1.M_1(w_1)
2	S_3(w_1, "x9999")	Q_1.M_1 → S_3.M_1, Q_1.M_2 → S_3.M_2, W_1 → a, "x9999" → b	w_1	"x9999"	interacts(Q_1.M_1, Q_1.M_2) Q_1.M_1(w_1) Q_1.M_2 ("x9999")
3	S_4(w_1,?y_1,?z_1)	Q_1.M_1 → S_4.M, w_1 → a, y_1 → b, z_1 → c	w_1	y_1, z_1	Q_1.M_1 (w_1,y_1,z_1)
4	S_4(w_2,?y_2,?z_2)	Q_1.M_2 → S_4.M, "x9999" → a, y_2 → b, z_2 → c	"x9999"	y_2, z_2	Q_1.M_2 ("x9999",y_2,z_2)
5	S_5(w_1,?y_1,?z_1)	Q_1.M_1 → S_5.M, w_1 → a, y_1 → b, z_1 → c	w_1	y_1, z_1	Q_1.M_1 (w_1,y_1,z_1)

the data-type properties of $Q_1.M_1$ and $Q_1.M_2$ that bound to distinguished variables in Q_1 also bound to distinguished variables in S_4, thus satisfying the condition 1 of Case 1. Furthermore, $Q_1.M_1$ and $Q_1.M_2$ are involved in object properties in Q_1. However, S_4 has the data-type property *hasCode* of medication bound to a distinguished variable in its RDF view, thus satisfying the condition 4 of Case 1. Therefore, S_4 can be used to cover $Q_1.M_1$ with the partial mapping β: $Q_1.M_1 \rightarrow S_4.M$, $w_1 \rightarrow a, y_1 \rightarrow b, z_1 \rightarrow c$ and $Q_1.M_2$ with the mapping $\beta : Q_1.M_2 \rightarrow S_4.M, ``x9999'' \rightarrow a, y_2 \rightarrow b, z_2 \rightarrow c$ (Table 2).

- *Service* S_6—Even though S_6 has the same input and output types as S_1, it does not match Q_1 since it does not cover the object property *takes*. Note that S_6 covers $Q_1.P$ and $Q_1.M_1$; however, it does not project any data-type property of $Q_1.P$ and $Q_1.M_1$ that can be used for answering Q_1. Hence, S_6 is not inserted in the mapping table.

Algorithm 1 describes the statements executed for handling cases 1 and 2 (covering class nodes and object properties) described above. The query mediator compares each class node C_i in Q to each class node C_j in the views and, if classes match, it adds C_i as a covered class node to t (a candidate tuple of T) and calls Subalgorithm 1 with t as a parameter for testing the conditions 1-4 of Case 1 (Algorithm 1, lines 1-9). If Subalgorithm 1 returns true, then t (augmented, if necessary, with additional covered class nodes/object properties) is added to T. The query mediator then compares object properties in Q with object properties in views and, in case of match, it calls Subalgorithm 2 to check Case 2 (Algorithm 1, lines 12-22).

Algorithm 1. Finding relevant RDF subgraphs:
Inputs: An RDF query $\boldsymbol{Q}$, a set of RDF views $\boldsymbol{V}$.
Outputs: Containment and connectivity table $\boldsymbol{T}$.
/* the function pbdv(c) returns the data-type properties of the concept c that are bound to distinguished variables. */
/* the function rdp(c) returns recoverable data-type properties of the concept c. */
/* the function pbev(c) returns the data-type properties of the concept c that are bound to existential variables. */
/* the function pbc(c) returns the data-type properties of the concept c that are bound to constants. */
/* const_i the constant value of a data-type property p_i, $pred_i$ a predicate placed on the property p_i. */
Begin
01: for each class node C_i in Q do
02: for each view v in V do
03: for each class node C_j in v do
04: if C_i and C_j have the same class type then
05: $C_i \rightarrow t$
06: if ($ClassNodeCovering(t, C_i, C_j)$) then
07: add t to the mapping table T
08: t = ø
09: end for
10: end for
11: end for
12: for each object property p_i in Q do
13: for each v in V do
14: for each object property p_j in v do
15: if p_i and p_j are the same then
16: $p_i \rightarrow t$
17: if ($ObjectPropertyCovering(t, p_i, p_j)$) then
18: add t to T
19: $t = \oslash$
20: end for
21: end for
22: end for
End

Subalgorithm 1. ClassNodeCover (t, C_i, C_j)
Inputs: t a candidate line in $\boldsymbol{T}$; $\boldsymbol{C_i}, \boldsymbol{C_j}$ two class nodes
Outputs: Boolean value

Begin
01: Test = true
02: if $pbdv(C_i) \not\subset pbdv(C_j) \cup rdp(C_j)$ then
03: return false
04: if $pbc(C_i) \not\subset pbdv(C_j) \cup rdp(C_j)$ then

TABLE 3
Combinations in the E-Prescription Scenario

Combination	Lines numbers	Involved services
C_1	1,2,3,4	$S_1(\$x,?w_1)$, $S_3(\$w_1$, "x9999"), $S_4(\$w_1,?y_1,?z_1)$, S_4("x9999",$?y_2,?z_2$)
C_2	1,2,4,5	$S_1(\$x,?w_1)$, $S_3(\$w_1$, "x9999"), $S_5(\$w_1,?y_1,?z_1)$, S_4("x9999",$?y_2,?z_2$)

TABLE 4
Checking that C_1 is Executable

Bound variables	Invoked Services
	S_1("x999",$?w_1$)
w_1	S_1("x999",$?w_1$), $S_3(\$w_1,?w_2)$, $S_4(\$w_1,?y_1,?z_1)$,
w_1, y_1, z_1, w_2	S_1("x999",$?w_1$), $S_3(\$w_1,?w_2)$, $S_4(\$w_1,?y_1,?z_1)$, $S_4(\$w_2,?y_2,?z_2)$
$w_1, y_1, z_1, w_2, y_2, z_2$	All services

05: return false
06: for each const_i in $pbc(C_i)$
07: if $const_i \notin pred_i$ then
08: return false
09: for each data-type property d in $pbev(C_i)$
10: if $d \notin pbdv(C_j) \cup rdp(C_j)$ then
11: pick up a class node C_k having d from Q
12: if C_k does not map to a class node C_m in v such that C_j and C_m join over d then
13: return false
14: if C_m is not in t then
15: $C_m \rightarrow t$
16: Test = Test and $ClassNodeCover(t, C_k, C_m)$
17: if d does not appear in other class nodes in Q then
18: discard d from $pbev(C_i)$
19: end for
20: if C_i is involved in an object property p_i in Q
21: if C_j does project out the associated skolem data-type properties then
22: if p_i does not map to a matching p_j in Q then
23: Return false
24: $p_i \rightarrow t$
25: test = test and $ObjectPropertyCover(t, p_i, p_j)$
26: return test
End

Subalgorithm 2. ObjectPropertyCovering (t, p_i, p_j))
Inputs: t, p_i, p_j
Outputs: Boolean value
Begin
01: test = true
02: fetch the nodes $< C_i, C_k >$ linked by p_i in Q
03: fetch the nodes $< C_j, C_m >$ linked by p_j in v
04: if $< C_i, C_k >$ or $< C_j, C_m >$ have not the same type then
05: Return false
06: C_i and $C_k \rightarrow t$
07: if C_j does not project skolem properties then
08: test = test and $ClassNodeCover(t, C_i, C_j)$
09: if C_j does not project skolem properties then
10: test = test and $ClassNodeCover(t, C_k, C_m)$
11: return test
End

In lines 1 and 2 of Subalgorithm 1, the query mediator verifies Case 1.1; in lines 4-8, it checks Case 1.3 and ensures that constants in C_i satisfy the specified values constraints (if any) in the view; in lines 9-18, it verifies Case 1.2 and, if C_i has an existential variable, it ensures that the view covers all class nodes (in Q) whose triples contain the existential variable; in lines 19-23, it verifies Case 1.4 by calling Subalgorithm 2. In lines 1-5 of Subalgorithm 2, the query mediator fetches the class nodes linked by the tested object properties from Q and v. Then, it tests whether these class nodes do not project the data-type properties used in their skolem functions, in which case the view must also cover these class nodes (Subalgorithm 2, lines 6-10).

4.2 Generating Composite Services

After generating the mapping table in the previous phase, the query mediator explores the different combinations from that table to cover the query graph [3], [17]. It considers the combination of disjoint sets of covered object properties and class nodes, we consider disjoint sets of covered object properties and class nodes for the following reason: each line in the mapping table contains a class node CN_i or an object property OP_i along with the minimum set of class nodes/object properties (CNs/OPs) that are linked with that class node/object property (CN_i/OP_i) via some joins that cannot be enforced if other class nodes/object properties (CNs/OPs) from a different view were used in the combination (this happens when the joins are made over existential variables in the view). This assumption speeds up the second step of the rewriting algorithm because it prunes the combinations with joins that cannot be enforced.

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.

Example. Continuing with the e-prescription scenario, there are two possible combinations (Table 3): the combination of the first, second, third, and fourth rows from Table 2 (referred to as C_1 in Table 3) and the combination of the first, second, fourth, and fifth rows from Table 2 (referred to as C_2 in Table 3).

Let us now consider combination C_1; only $S_1(\text{``}x999,\text{''}?w_1)$ can be invoked at the beginning as its input parameter is bound. After the invocation of $S_1(\text{``}x999,\text{''}?w_1)$, the variable w_1 become available as shown in Table 4; hence, $S_3(\$w_1, \text{``}x9999\text{''})$ and $S_4(\$w_1, ?y_1, ?z_1)$ can be invoked. After the invocations of these services, all variables become bound, and the whole set of services can be invoked. Consequently, C_1 is executable and is considered as a valid composition.

Now assume that S_1 has a different access pattern, say $S_1(?a, \$b)$. In this case, C_1 would become unexecutable as its component services cannot be invoked; none of their input parameters is bound: the variable a (which is bound in Q_1 to a constant) of the service $S_1(?a, \$b)$ is an output variable now.

In the running example, we had only two valid rewritings (combinations), but in the general case we may have multiple possible rewritings. In Section 6.2, we show that in some cases we may have a sheer number of possible rewritings. Depending on the adopted hypothesis behind the completeness of the DP services (whether the composition system operates in the closed word or the open world settings) one may need to choose only one or all of the possible rewritings.

Algorithm 2 allows the composite service generation as discussed above. The query mediator first combines atomic class nodes with missing data-type properties to have complete class nodes (line 1). Then, for each individual class node and object property in Q, it sets up a subset to include the covering services (lines 2 and 3). The query mediator inserts each service (each tuple in T corresponds to a service) in all the subsets corresponding to its covered class nodes and object properties (lines 4 and 5). Then, it combines elements from the different constructed subsets and returns the combinations that cover all class nodes and object properties in Q and for which the subgraphs covered by the services involved in the combination are disjoint (lines 6-10).

Algorithm 2. Combining RDF subgraphs:
Inputs: Mapping and connectivity table.
Outputs: set of candidate compositions.
/* the function nop(s) returns the class nodes and object properties covered by s */

Begin:
01: add missing data-type properties to each class node that is missing properties.
02: for each *class node* and *object property* in Q
03: set up a subset set_i
04: for each service S in T
05: put S in each of the covered subsets
06: from every subset $\text{set}_i \ldots \text{set}_n$ combine services
07: if a combination C satisfies
08: $\ldots \cup nop(\text{S}_j) \cup nop(\text{S}_i) \ldots = nop(Q)$ and
09: nop(S_i) and nop(S_j) $= \phi, (\text{i} \neq \text{j})$ then
10: add C to the candidate compositions
End

Algorithm 3 checks whether a combination is executable. For that purpose, the query mediator initializes the invokable services list A with ø and the bound variables list B with Q's constants (lines 1 and 2). Then, it adds the services that are invokable with the current bound variables to A (lines 3-10). Whenever a new service is added to A, its outputs variables are added to B; the query mediator checks again whether the remaining services become invokable with the updated list B. If no bound variable is added to B, the query mediator checks whether A contains all the services in composition (lines 11 and 12), in which case it returns true (i.e., the composition is executable).

Algorithm 3. Executability Validation
Inputs: A combination C_i of services with access patterns.
Outputs: A Boolean value denoting whether or not the combination is a valid composition.

Begin
01: A = ø /*initialize the list of invocable services */
02: B = {Variables bound to constants in Q}
03: repeat
04: done = true
05: for $i = 1$ to k (the total number of services in C_i) do
06: if $\text{S}_i \notin$ A and inputVariables(S_i) $\subseteq$ B then
07: A = A and S_i
08: B = B $\cup$ outputVariables (S_i)
09: done = false
10: until done
11: if A $\equiv \text{C}_i$ return True
12: else return False
End

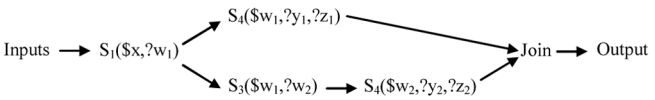

Fig. 8. Example of dependency graph.

Component services must be executed in a particular order depending on their access patterns. 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 execution 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. Fig. 8 shows the dependency graph for C_1. There is a dependency between $S_1(\$x, ?w_1)$ and $S_3(\$w_1, ?w_2)$, and between $S_1(\$x, ?w_1)$ and $S_4(\$w_1, ?y_1, ?z_1)$. Therefore, $S_1(\$x, ?w_1)$ must be executed first, then $S_3(\$w_1, ?w_2)$ and $S_4(\$w_1, ?y_1, ?z_1)$ can be executed in parallel. $S_1(\$x, ?w_1)$ is called the parent of the services $S_3(\$w_1, ?w_2)$ and $S_4(\$w_1, ?y_1, ?z_1)$. There is also a dependency between $S_3(\$w_1, ?w_2)$ and $S_4(\$w_2, ?y_2, ?z_2)$, hence $S_3(\$w_1, ?w_2)$ executes before $S_4(\$w_2, ?y_2, ?z_2)$.

Algorithm 4 shows the way composite services are executed by the query mediator. The query mediator creates a thread T_i for each service S_i in composition C. The thread T_i takes its input tuples from a separate join thread J_i that joins the outputs of S_i's parents. If S_i has no parents in C, then T_i takes its inputs from the input relation I that contains all specific values in the query. The thread T_i invokes S_i for each input tuple, filters the returned tuples, joins them with the input tuple, and writes them to its output. The query result is obtained from the output of the join thread J_{out} which joins the outputs of all services that are leaves in the dependency graph of C. $\triangleleft \triangleright \triangleleft \infty$

Algorithm 4. Executing Compositions:
Input: 1) - An executable composition $\boldsymbol{C}$
2) - An input relation $\boldsymbol{I}$.
Output: The results of the query Q

Begin:
1: for each web service S_i in $\boldsymbol{C}$
2: launch a thread T_i
3: if S_i has no parents in $\boldsymbol{C}$
4: set up T_i to take input from $\boldsymbol{I}$
5: else if S_i has a single parent S_p in $\boldsymbol{C}$
6: set up T_i to take input from T_p's output

TABLE 5
Mapping Table for the Parameterized Query Q_2

	Service	Variables Mapping	In	Out	Constraints	Covered nodes/properties
1	$S_1(\$x,?w_1)$	$Q_2.P \rightarrow S_1.P$, $Q_2.M_1 \rightarrow S_1.M$, $x \rightarrow a$, $w_1 \rightarrow b$	x	w_1	x≥x888	takes($Q_2.M_1$, $Q_2.P$) $Q_2.P(x)Q_2.M_1(w_1)$
2	$S_2(\$x,?w_1)$	$Q_2.P \rightarrow S_2.P$, $Q_2.M_1 \rightarrow S_2.M$, $x \rightarrow a$, $w_1 \rightarrow b$	x	w_1	x≤x888	takes($Q_2.M_1$, $Q_2.P$) $Q_2.P(x)$ $Q_2.M_1(w_1)$
3	$S_3(\$w_1, ?w_2)$	$Q_2.M_1 \rightarrow S_3.M_1$, $Q_2.M_2 \rightarrow S_3.M_2$, $w_1 \rightarrow a$, $w_2 \rightarrow b$	w_1	w_2		interacts($Q_2.M_1$, $Q_2.M_2$) $Q_2.M_1(w_1)$ $Q_2.M_2$ (w_2)
4	$S_4(\$w_1,?y_1,?z_1)$	$Q_2.M_1 \rightarrow S_4.M$, $w_1 \rightarrow a$, $y_1 \rightarrow b$, $z_1 \rightarrow c$	w_1	y_1, z_1	w_1≥p660	$Q_2.M_1$ (w_1,y_1,z_1)
5	$S_5(\$w_1,?y_1,?z_1)$	$Q_2.M_1 \rightarrow S_5.M$, $w_1 \rightarrow a$, $y_1 \rightarrow b$, $z_1 \rightarrow c$	w_1	y_1, z_1	w_1≤8999	$Q_2.M_1$ (w_1,y_1,z_1)
6	$S_4(\$w_2,?y_2,?z_2)$	$Q_2.M_2 \rightarrow S_4.M$, $w_2 \rightarrow a$, $y_2 \rightarrow b$, $z_2 \rightarrow c$	w_2	y_2, z_2	w_2≥p660	$Q_2.M_2$ (w_2,y_2,z_2)
7	$S_5(\$w_2,?y_2,?z_2)$	$Q_2.M_2 \rightarrow S_5.M$, $w_2 \rightarrow a$ $y_2 \rightarrow b$, $z_2 \rightarrow c$	w_2	y_2, z_2	w_2≤8999	$Q_2.M_2$ (w_2,y_2,z_2)

7: else
8: launch a join thread J_i
9: set up T_i to take input from J_i's output
10: launch join thread J_{out} as query results

Thread T_i:
1: while (tuples available on T_i's input)
2: read a tuple t_1 from T_i's input
3: invoke S_i with values t_1
4: for each returned tuple t_2
5: apply all query predicates that pertain to the attributes in t_2
6: if t_2 satisfies all predicates
7: write $t_1 \bowtie t_2$ to T_i's output

Thread J_i
1: perform the join of the outputs of S_i's parents in C

Thread J_{out}
1: perform the join of the outputs of web services that are leaves in C.

End

5 Handling Parameterized Queries

In the previous section, we addressed the problem of answering queries with specific input values (e.g., a patient and prescribed medication identified by "x999" and "x9999," respectively). The resulting composite service cannot necessarily be reused to answer the same query for a different patient and/or different medication. In this section, we generalize our query answering approach to answer *parameterized query*; in these queries, ranges of values or no specific values are given for the input parameters of the query. For instance, let us assume that our physician *Alice* would like to answer the following query Q_2 "*For any given a social security number x of a patient and a drug code* w_2 *representing the medication to be prescribed, verify whether the medications taken by the patient may interact with* $\$w_2$." Q_2 is parameterized over the data-type properties *hasSSN* of *Patient* and *hasCode* of *Drug*.

One implication of considering parameterized queries is that component services cannot be chosen at the composition time. The selection of those services depends on the actual values of input parameters, which are provided at the execution time. For instance, S_2 was not considered in the case of Q_1 because the SSN value was out of the range of SSNs accepted by S_2. S_2 becomes usable in the case of the parameterized query Q_2, as Q_2 does not specify any specific value for the SSN. Table 5 shows the mapping table for the parameterized query Q_2.

As some DP services cover partially the required ranges of input/output values, multiple similar services (with the same binding patterns) are combined together to cover completely the required ranges of values. For example, S_1 and S_2 are used together to cover the complete range of patients' SSNs. The same applies to S_4 and S_5 in covering the concepts $Q_2.M_1$ and $Q_2.M_2$.

5.1 Eliminating False Invocations

DP services may be called with input values that violate their specified constraints on accepted input values. Invoking a service with a data value that violates its constraints is called *false invocation*. False invocations either return an empty result set or throw an error message. Eliminating false invocations has a significant impact on the execution time of the whole composition. We exploit the constraints placed on the accepted values of input parameters to filter out false invocations. For example, filters are inserted before calling S_1 and S_2 to verify whether the patient's SSN is greater than "x888" in the case of S_1, and lower than "x888" in the case of S_2.

5.2 Orchestrating the Generated Composite Service

The composite service generated for a parameterized query is deployed as a new DP service to be invoked for different

TABLE 6
Execution Plan Constructs

Operation	Description
Invoke (S, I)	A thread-based process invoking the service S with each single data tuple in the input relation I.
Join(I_1, …, I_n)	A thread-based process joining the tuples in a set of relations I_1, …, I_n.
Union (I_1, …, I_n)	A thread-based process unifying the tuples in a set of relations I_1, …, I_n.
Select(I, Condition)	A process selecting the tuples that satisfy a given condition in a given relation

values of the input parameters. Similarly to traditional web services composition (e.g., BPEL [2]), the generated composite service needs to be translated into an execution plan (called orchestration in BPEL) describing the data and control flows. The generated composite service is translated into an execution plan using the operations depicted in Table 6.

Each service occurrence in the generated composite service is translated into an "*invoke*" operation. An *Invoke (S, I)* operation invokes the service S with each data tuple in the input relation I. The outputs of DP services that cover the same portion in the query (e.g., S_1 and S_2) are grouped by a "*union*" operation. This operation is responsible for removing redundant tuples. The "*Join*" operation is used to feed a service with data tuples coming from its parents. The "*Select*" operation is used to filter out tuples that do not satisfy the service invocation constraint, hence avoiding false invocations.

Fig. 9 shows a graphical representation of the execution plan for the composite service generated for Q_2. The plan starts with filtering out the tuples in the input relation before invoking S_1 and S_2. The results are aggregated in the subsequent union operation (*union(3)*). It is important to note that the query mediator does not wait until an operation produces all its output tuples to proceed with the execution of subsequent operations; an operation starts executing as soon as its preceding operations begin to produce data tuples on their outputs. For example, once the union operation *union(3)* starts to receive tuples from its parents operations it relays them to the operations *select(4a)*, *select(4b)*, and *invoke(4)*. The operations *select(4a)* and *select(4b)* are used to verify the constraints placed on S_4 and S_5 before invoking those services. The same applies for the operations *select(4a)* and select(4b). The final result is obtained from the *join(7)* operation.

6 Implementation and Evaluation

In this section, we first describe an implementation of our approach. Then, we provide an analytical and experimental evaluation.

6.1 Implementation

The architecture of our system for querying and composing DP services is shown in Fig. 10. Data sources (e.g., relational databases, legacy data sources, silos of data-centric home-grown or packaged applications) with different proprietary interfaces are all exposed as DP services. As in WSDL-S, we annotate DP services with their RDF views; we exploit the *extensibility feature* of WSDL to hook *operations elements* in a WSDL file to their corresponding RDF views. WSDL-S annotates the different elements of a WSDL file (including *inputs*, *outputs*, and functional aspects like *operations*, their *preconditions*, and *effects*) using the *XML extensibility feature* of WSDL files. In our framework, we exploit the same

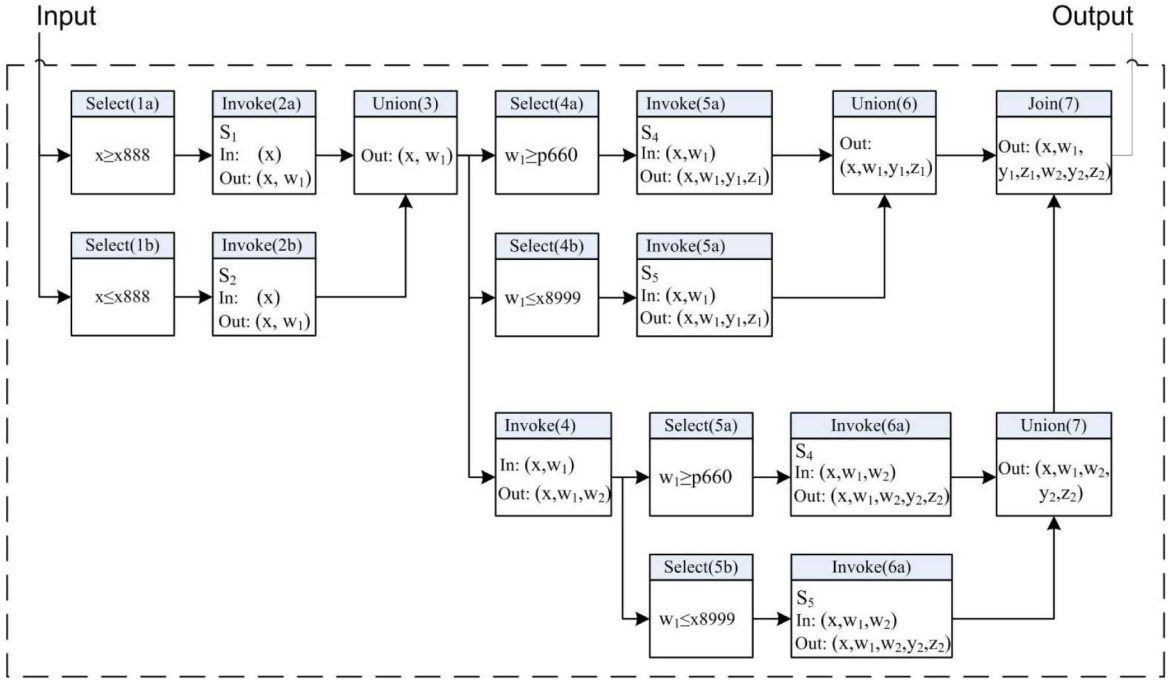

Fig. 9. Execution plan for Q2.

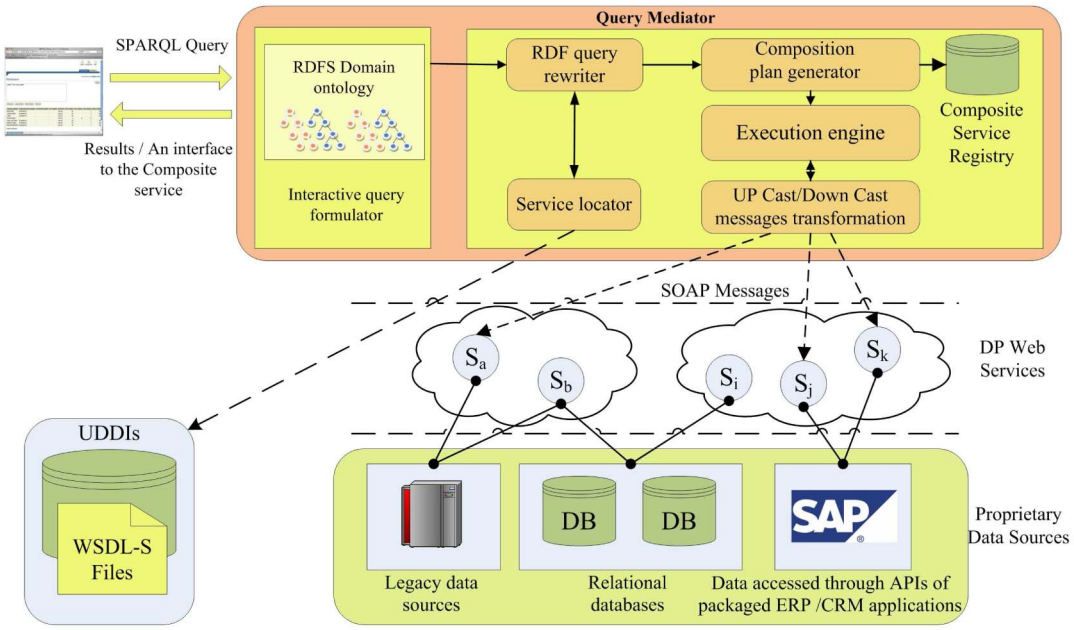

Fig. 10. Architecture.

extensibility feature of WSDL files to annotate each *operation element* with its associated RDF view. Specifically, we added to each *operation* element an extensible element "***rdfQuery***" containing the RDF view. Upon publishing a DP service within a service registry, the service provider has to define (manually or semiautomatically) the RDF views capturing the semantics of each published operation in the advertised service. These views are incorporated inside the corresponding ***rdfQuery*** elements.

The query mediator implementing our service query model includes five components (Fig. 10). The *interactive query formulator* is a web-based query interface that helps users specify their RDF queries (SPARQL queries) over the mediated ontology. The *service locator* discovers WSDL-S descriptions by accessing the service registries (UDDI). At the services registry side, services are associated with the ontology nodes used in expressing their RDF views in accordance with the approach [12]. The mechanism to relate semantics (i.e., ontology concepts) with services advertised in the UDDI registries are the *tModel keys* and the *category bags* of registry entries. Services are matched based on whether or not they are annotated with one or more of the ontological concepts used in formulating the posed query. The *RDF query rewriter* implements our RDF query rewriting algorithms; it determines whether the services returned from the *service locator* can be used for answering the posed query. The *composition plan generator* generates plans for the posed query. The generated plans are either sent for immediate execution or are deployed as new web services; the choice depends on whether the posed query is specific or parameterized. The *execution engine* implements the different operators used in the generated plans. In addition, it transforms the messages exchanged with the invoked services. Services may have schemas for their input/output messages that are different from the schemas of input and output parameters of their corresponding views (which are dictated by the ontology structure). Service providers need to specify for each published operation the following mappings: 1) the mapping between the input message and xml schema obtained from the serialization of input parameters of the associated RDF view (which is called as the *down cast mapping)* and 2) the mapping between the output message and the schema obtained from the serialization of output parameters of the associated RDF view, which is called as the *up cast mapping*. These mappings can be expressed using *XQuery* and *XSLT*.

6.2 Evaluation

The RDF query rewriting algorithm has two major phases. In the first phase, the algorithm goes through each class node and each object property in the query and compares them to each class node and each object property in each of the considered views. Assume N_1 is the number of class nodes in Q, N_2 is the number of object properties in Q, and L is the number of the considered views, the complexity of the first phase is $O(\mathrm{N}_1 \times \mathrm{M}_1 \times \mathrm{L} + \mathrm{N}_2 \times \mathrm{M}_2 \times \mathrm{L})$. In the second phase, the algorithm constructs individually for each of the class node and object property in Q a list L_i containing their various covering services—the number of constructed lists is $(\mathrm{N}_1 + \mathrm{N}_2)$. Then, it tests each possible combination obtained from the Cartesian product of various lists for covering the query. Assume I_i is the number of elements in a constructed list L_i, the complexity of the second step is:

$$O\Big(\prod_{i=1}^{N1+N2} I_i\Big).$$

The worst case happens when each of the class nodes in the views map to all class nodes in Q and each object property in the views map to all object properties in Q, in which case each constructed list will have a sheer number of elements to be tested against elements from other lists. Assume N_1 is the number of class nodes in Q, M_{1Max} is the maximal number of class nodes in a view, and L is the number of

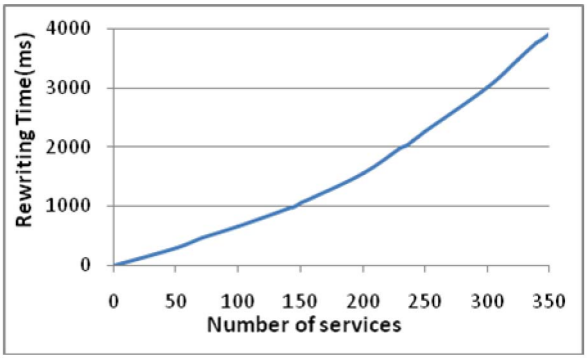

Fig. 11. Chain queries with 10 object properties linking 11 class nodes.

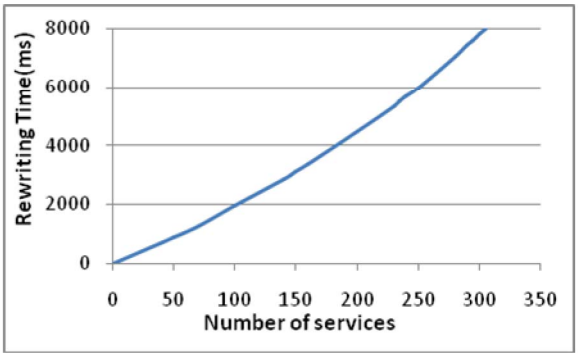

Fig. 12. Chain queries with 12 object properties and views with three and four object properties.

views, the worst case running time will be of the order: $O(\mathrm{N}_1 \times \mathrm{M}_{1\mathrm{Max}} \times L)^{N1+N2}$ – note that N_1 is always bigger than N_2 by /1/ in any RDF view, hence the assumption that the list with the maximal number of elements will be a one corresponding to a class node ($\mathrm{I}_{\mathrm{Max}} = \mathrm{N}_1 \times \mathrm{M}_{1\mathrm{Max}}$). We note that all query rewriting algorithms are limited because the general problem of answering conjunctive queries using conjunctive views is NP-Complete [18] as it may involve searching through an exponential number or rewritings.

While the complexity of the RDF query rewriting algorithm is high, its performance is, in general, very good for realistic problem sets. We conducted experiments to evaluate the performance of our RDF-oriented query rewriting and examine its behavior and scalability as the number of services increases. We wanted to test the following hypothesis: *query rewriting techniques can be used to generate composition plans for new web services from large number of existing web services*. We considered two general classes of queries, chain, and star queries [40]. In all experiments, the queries and views were generated randomly by an RDF query generator implemented in Java. The query rewriting algorithm is implemented in Java and runs on a Pentium (4) 3.06 GHz and 512 MB RAM running Windows XP (SP2). The results are average of 10 runs.

6.2.1 Chain Queries

The graph of a chain query/view includes a line of class nodes linked by a chain of object properties. We have considered RDF queries and views in which only the first and the last class nodes project out the data-type properties used in their skolem functions as distinguished variables. Experiments were conducted for two particular cases: 1) queries and views with a length of 10 object properties (linking 11 class nodes). In this case, an RDF view is usable only if it is identical to the query. The corresponding results in Fig. 11 show that the algorithm can handle up to 350 views in less than 4 seconds and 2) queries with a length of 12 object properties and views with a length of three and four object properties. As shown in Fig. 12, the algorithm was able to handle 311 views in less than 8 seconds. As the number of views becomes larger, the number of those intersecting with the query increases substantially as does the number of rewritings to be tested. However, for a set of 350 views (the maximum number of RDF views considered in the experiments—a number considered enough for most of realistic applications) the algorithm performed fairly well.

6.2.2 Star Queries

In star queries, only one class node is linked with every other class node via an object property. No links exist between the other class nodes. We have omitted the case where class nodes in every RDF views project the data-type properties that are used in their skolem functions because this mirrors the performance of chain queries. Fig. 13 shows the performance in the star scenario. When both the posed query and views have a central class node linked to five other class nodes via five object properties the algorithm scales up to 350 views in less than 1 second.

6.2.3 Summary

Experiment results showed that the RDF rewriting algorithm can handle hundreds of DP services in a reasonable time. In the context of parameterized queries, the resulting composition is meant to be deployed as a new web service that is used for subsequent invocations: query rewriting is performed once and the generated composite service is used throughout the lifetime of that service. In most application domains, while the number of concrete DP services that are available on the web might be very large, the logical views that are implemented by concrete services within the application domain are limited in number (the same view may be implemented by multiple DP services). Therefore, one would use a scheme similar to that in [26] whereby a query is rewritten in terms of a set of logical RDF views; the views used in the rewriting are then matched against the concrete DP services. In such setting, query

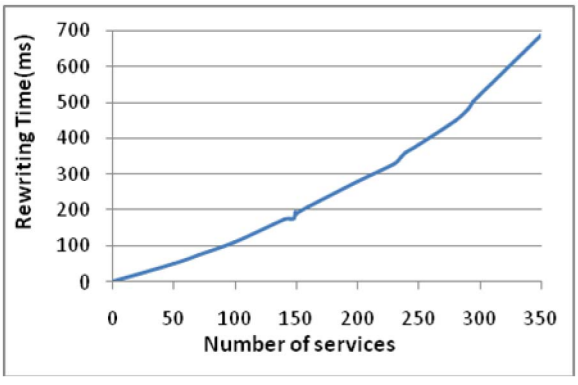

Fig. 13. Star queries/views with five object properties.

rewriting is generally performed in a reasonable time because 1) the number of logical views is limited and 2) one needs to pick up only one rewriting out of the possible ones and match it to concrete DP services—the rewriting algorithm can stop as soon as the first rewriting is available.

7 Related Work

In this section, we compare our approach to related work. First, we review some industrial platforms and projects that relate to DP services. Then, we review previous work in the following areas: 1) *Web Service Composition*, 2) *Querying DP Services*, and 3) *Data Integration*.

7.1 Industrial Platforms for DP Services

Many products are currently offered or being developed to make the creation of DP services easier than ever, to cite a few, AquaLogic by BEA Systems [6], Astoria by Microsoft [25], MetaMatrix by RedHat [32], Composite Software [10], Xcalia [44], and IBM [43]. In these platforms, the semantics of a DP service is known as long as the SOA application developer stays within the platform (e.g., AquaLogic) [6]. Once outside, for example, when DP services are published in a service registry outside the enterprise boundaries, it becomes hard to differentiate between services as their semantics are not defined. Our work complements these industrial efforts by providing an integrated framework to declaratively describe the semantics of a DP service (offered by the mentioned products) and a model to query and compose DP services.

7.2 Web Service Composition

Previous approaches in the area of web service composition (e.g., [14], [41], [42], [46], [23], [5]) have focused only on EP web services. In these approaches, the exploited composition algorithms (which are largely inspired by AI planning techniques) assume that the capability of a web service can be captured based on the *business function* implemented by the service and its *inputs*, *outputs*, *preconditions*, and *effects (IOPEs)*. The work reported in [38] is a representative example; the described SHOP2-based system in that work composes web services automatically based on their implemented functionalities, input and output constraints, preconditions, and effects. This assumption makes these composition algorithms [37] inapplicable to DP services that all share the same business function (i.e., data retrieval) and have no preconditions or effects. In order to represent the capability of a DP service (and capture its semantics) one must define a declarative view over a mediated ontology to capture the semantic relationship between its inputs and outputs. For this reason, we employ a data integration approach for the purpose of automating the composition of DP services and use the query rewriting techniques as the composition algorithm.

7.3 Web Services Query Models

Recent approaches addressed the issues of querying web services [39], [34], [45], [26], [20]. The query model proposed in [26] consists of three levels: *query level*, *virtual level*, and *concrete level*. At the query level, users express their declarative queries over a set of domain relations. Domain relations are mapped to a set of web service-like operations at the virtual level. These operations represent the different functionalities services may offer in a particular domain. Concrete services at the concrete level use these virtual operations to represent their functionalities. Our approach follows a similar methodology to query DP services, i.e., users pose their declarative queries over mediated ontologies without being concerned of how services are selected and composed to answer their queries. However, as the semantics of a DP service cannot be captured by a *virtual operation*, we use RDF parameterized queries over RDF/S mediated ontologies. A formal *service query model* is presented in [45]. The model captures the three key features of web services: *functionality*, *behavior*, and *quality*, along with a *service query algebra* that is used to formulate service query expressions. It also provides measures to evaluate the quality of the different Services execution plans resulting from handling a posed query. The proposed query model does not consider the semantic relationships that may exist between inputs and outputs of a DP service. The Web Service Management System (WSMS) proposed in [39] addresses query optimization over web services. Services are modeled as *relations* used directly in expressing users' queries; i.e., users are implicitly assumed to have an understanding of the underlying semantics for each of the services that are available to them, and they directly express their queries by projecting/joining input/output parameters of available services. In our work, users are not required to have an implicit knowledge of the semantics of available DP services. They express their queries over domain ontologies and then queries are rewritten in terms of services based on their RDFS views. In addition to specific queries, we are also able to answer parameterized queries. Further, our data model is richer, i.e., our queries are formulated on mediated ontologies rather than on services' relational predicates and not restricted to "*pipeline queries*."

7.4 Data Integration Systems

Another area of related research is that of data integration systems (e.g., InfoMaster [15] and Information Manifold [19]). Our work differs from these works in many ways. First, the key focus in these systems is shifted toward resolving specific queries given a set of incomplete data sources, whereas in our work the focus is on constructing a composition of services that is independent of a particular input value. That is, we resolve parameterized queries. This necessitates the introduction of an optimization mechanism to filter out "at the execution time" irrelevant services when the composition is invoked with a specific input value. Second, compared to previous query rewriting algorithms [30], [13], [16] that were proposed for the relational data model; our proposed query rewriting algorithm is compliant with the RDF/RDFS data models. Our proposed algorithm takes into account the RDFS semantic constraints of "*subClassOf*," "*subPropertyOf*," and "*domain*" and "*range*" that are defined in domain ontologies. As far as we know, our algorithm is the first to address the problem of composing DP services based on query rewriting techniques.

Some works have addressed the problem of RDF query answering [31], [9]; however, these works are limited only to answering specific queries whereas our work addresses also

parameterized queries and handles queries and views with limited access constraints. In addition, we take into account the RDFS hierarchy constraints while resolving RDF queries.

The problem of RDF query rewriting is related to graph covering [21]; however, existing algorithms in that area focus on testing whether a graph H is covered by another graph G (i.e., G can be reduced to H) whereas in our work we focus on building a new graph (the composition's graph) out of set of atomic graphs (the services' graphs). Also, the graphs in our work are different from those in that area as we must distinguish between edges that are data-type properties and edges that are object properties, the vertexes that have types from those that are blank nodes. Such distinctions make existing algorithms in graph covering inapplicable to our purposes.

Active XML (AXML) embeds web service calls in XML documents [1]. In contrast to our work, AXML does not address query resolution by web service composition. In addition, the web services inside a document remain unchanged all along the life time of the document and are selected at the creation time of the document. Hence, AXML framework is suitable when there is a need to manage a fixed predefined set of data sources. Senellart et al. [35] address the issue of accessing the "hidden web" by means of web services; it presents some techniques to transform the hidden data sources on the web (e.g., web forms) to web services that have explicit semantics for their inputs and outputs. However, this work does not address the automatic composition of the created DP services.

8 Conclusion

In this paper, we presented a novel approach for querying and automatically composing DP services. DP services are described as RDF views over a mediated ontology. These views are enriched with RDFS semantic constraints (e.g., *subClassOf* and *subPropertyOf*) and used to annotate WSDL descriptions. We proposed RDF query rewriting algorithms to compose DP services. We also conducted a performance analysis on a wide data set of chain and star queries to assess our proposed. The results show that the algorithm scales up very well to a large number of services, covering thus most realistic applications. As a future work, we plan to address data privacy concerns when composing DP services. We also plan to consider Quality of Service (QoS) while processing queries and composing DP services.

References

[1] S. Abiteboul, O. Benjelloun, and T. Milo, "Web Services and Data Integration," *Proc. Int'l Conf. Web Information Systems Engineering (WISE)*, pp. 3-6, 2002.

[2] G. Alonso, F. Casati, H.A. Kuno, and V. Machiraju, *Web Services—Concepts, Architectures and Applications*. Springer, 2004.

[3] A. Argyriou, M. Herbster, and M. Pontil, "Combining Graph Laplacians for Semi-Supervised Learning," *Proc. Advances in Neural Information Processing Systems*, vol. 18, pp. 5-8, 2005.

[4] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge Univ. Press, 2003.

[5] D. Calvanese, G.D. Giacomo, M. Lenzerini, M. Mecella, and F. Patrizi, "Automatic Service Composition and Synthesis: the Roman Model," *IEEE Data Eng. Bull.*, vol. 31, no. 3, pp. 18-22, 2008.

[6] M.J. Carey, "Data Delivery in a Service-Oriented World: The BEA aquaLogic Data Services Platform," *Proc. SIGMOD Conf.*, pp. 695-705, 2006.

[7] M.J. Carey, "Declarative Data Services: This Is Your Data on SOA," *Proc. IEEE Int'l Conf. Service-Oriented Computing and Applications*, p. 4, 2007.

[8] A.K. Chandra and P.M. Merlin, "Optimal Implementation of Conjunctive Queries in Relational Data Bases," *Proc. Conf. Record of the Ninth Ann. ACM Symp. Theory of Computing*, pp. 77-90, 1977.

[9] H. Chen, Z. Wu, and Y. Mao, "Rewriting Queries Using Views for RDF-Based Relational Integration," *Proc. IEEE Int'l Conf. Tools with Artificial Intelligence (ICTAI)*, pp. 260-264, 2005.

[10] Composite Software, Inc., SOA Data Services, http://compositesoftware.com/solutions/soa.shtml, 2010.

[11] Y. Ding, D. Fensel, M. Klein, and B. Omelayenko, "The Semantic Web: Yet Another Hip?" *Data and Knowledge Eng.*, vol. 41, nos. 2/3, pp. 205-227, 2002.

[12] A. Dogac, G. Laleci, Y. Kabak, and I. Cingil, "Exploiting Web Service Semantics: Taxonomies vs. Ontologies," *IEEE Data Eng. Bull.*, vol. 25, no. 5, pp. 10-16, Dec. 2002.

[13] O.M. Duschka, M.R. Genesereth, and A.Y. Halevy, "Recursive Query Plans for Data Integration," *J. Logic Programming*, vol. 43, pp. 49-73, 2000.

[14] M.A. Eid, A. Alamri, and A. El-Saddik, "A Reference Model for Dynamic Web Service Composition Systems," *Int'l J. Web and Grid Services*, vol. 4, no. 2, pp. 149-168, 2008.

[15] M.R. Genesereth, A.M. Keller, and O.M. Duschka, "Infomaster: An Information Integration System," *ACM SIGMOD Record*, vol. 26, pp. 539-542, 1997.

[16] G. Grahne and A.O. Mendelzon, "Tableau Techniques for Querying Information Sources through Global Schemas," *Proc. Int'l Conf. Database Theory (ICDT)*, pp. 332-347, 1999.

[17] A.Y. Halevy, "Answering Queries Using Views: A Survey," *Very Large Data Bases J.*, vol. 10, pp. 270-294, 2001.

[18] A.Y. Halevy, A.O. Mendelzon, Y. Sagiv, and D. Srivastava, "Answering Queries Using Views," *Proc. Principles of Database Systems (PODS)*, pp. 95-104, 1995.

[19] A.Y. Halevy, A. Rajaraman, and J.J. Ordille, "The World Wide Web as a Collection of Views: Query Processing in the Information Manifold," *Proc. Workshop Materialized Views: Techniques and Applications (VIEWS)*, pp. 43-55, 1996.

[20] D. Hull, "Semantic Matching of Bioinformatic Web Services," PhD thesis, The Univ. Manchester, School of Computer Science, 2008.

[21] J. Kratochvil, A. Proskurowski, and J.A. Telle, "Complexity of Graph Covering Problems," *Nordic J. Computing*, vol. 5, pp. 93-105, 1998.

[22] D. Martin et al., "Bringing Semantics to Web Services: The OWL-S Approach," *Proc. First Int'l Workshop Semantic Web Services and Web Process Composition (SWSWPC '04)*, pp. 26-42, July 2004.

[23] B. Medjahed and A. Bouguettaya, "A Multilevel Composability Model for Semantic Web Services," *IEEE Trans. Knowledge Data Eng.*, vol. 17, no. 7, pp. 954-968, July 2005.

[24] B. Medjahed, A. Bouguettaya, and A.K. Elmagarmid, "Composing Web Services on the Semantic Web," *Int'l J. Very Large Data Bases*, vol. 12, no. 4, pp. 333-351, Nov. 2003.

[25] Microsoft Corporation: ADO.NET Data Services (also known as Project Astoria), http://astoria.mslivelabs.com, 2007.

[26] M. Ouzzani and A. Bouguettaya, "Efficient Access to Web Services," *IEEE Internet Computing*, vol. 8, no. 2, pp. 34-44, Mar. 2004.

[27] T. Pankowski, "XML Schema Mappings Using Schema Constraints and Skolem Functions," *Knowledge-Driven Computing*, vol. 102/2008. Springer, 2008.

[28] M.P. Papazoglou and B. Kratz, "Web Services Technology in Support of Business Transactions," *Service Oriented Computing and Applications*, vol. 1, no. 1, pp. 51-63, 2001.

[29] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: A Research Roadmap," *Int'l J. Cooperative Information Systems*, vol. 17, no. 2, pp. 223-255, 2008.

[30] R. Pottinger and A.Y. Halevy, "MiniCon: A Scalable Algorithm for Answering Queries Using Views," *Very Large Scale Data Base J.*, vol. 10, nos. 2/3, pp. 182-198, 2001.

[31] B. Quilitz and U. Leser, "Querying Distributed RDF Data Sources with SPARQL," *Proc. Fifth European Semantic Web Conf. (ESWC '08), The Semantic Web: Research and Applications*, vol. 5021/2008, pp. 524-538, 2008.

[32] Red Hat, Inc.: MetaMatrix Enterprise Data Services Platform, http://www.redhat.com/jboss/platforms/dataservices, 2007.

[33] D. Roman et al., "WWW: WSMO, WSML, and WSMX in a Nutshell," *Proc. First Asian Semantic Web Conf. (ASWC '06), The Semantic Web,* pp. 516-522, Sept. 2006.
[34] M. Sabesan and T. Risch, "Adaptive Parallelization of Queries over Dependent Web Service Calls," *Proc. First IEEE Workshop Information and Software as Services (WISS '09),* 2009.
[35] P. Senellart, S. Abiteboul, and R. Gilleron, "Understanding the Hidden Web," *European Research Consortium for Informatics and Math. News,* vol. 72, pp. 32-33, Jan. 2008.
[36] A.P. Sheth, K. Gomadam, and A. Ranabahu, "Semantics Enhanced Services: METEOR-S, SAWSDL and SA-REST," *IEEE Data Eng. Bull.,* vol. 31, no. 3, pp. 8-12, 2008.
[37] E. Sirin, B. Parsia, and J.A. Hendler, "Filtering and Selecting Semantic Web Services with Interactive Composition Techniques," *IEEE Intelligent Systems,* vol. 19, no. 4, pp. 42-49, July/Aug. 2004.
[38] E. Sirin, B. Parsia, D. Wu, J.A. Hendler, and D.S. Nau, "HTN Planning for Web Service Composition Using SHOP2," *J. Web Semantics,* vol. 1, no. 4, pp. 377-396, 2004.
[39] U. Srivastava, K. Munagala, J. Widom, and R. Motwani, "Query Optimization over Web Services," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 355-366, 2006.
[40] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and Randomized Optimization for the Join Ordering Problem," *Very Large Data Bases J.,* vol. 6, no. 3, pp. 191-208, 1997.
[41] S.G.H. Tabatabaei, W.M.N. Wan-Kadir, and S. Ibrahim, "A Comparative Evaluation of State-of-the-Art Approaches for Web Service Composition," *Proc. Int'l Conf. Software Eng. Advances (ICSEA),* vol. 4, pp. 488-493, 2008.
[42] T. Weise, S. Bleul, D. Comes, and K. Geihs, "Different Approaches to Semantic Web Service Composition," *Proc. Third Int'l Conf. Internet and Web Applications and Services,* pp. 90-96, 2008.
[43] K. Williams and B. Daniel, "SOA Web Services—Data Access Service," *Java Developer's J.,* 2006.
[44] Xcalia, Inc., Xcalia Data Access Services, http://www.xcalia.com/products/xcalia-xdas-data-access-service-SDO-DAS-data-integration-through-web-services.jsp, 2010.
[45] Q. Yu and A. Bouguettaya, "Framework for Web Service Query Algebra and Optimization," *ACM Trans. Web,* vol. 2, no. 1, 2008.
[46] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed, "Deploying and Managing Web Services: Issues, Solutions, and Directions," *Very Large Data Bases J.,* vol. 17, pp. 537-572, Mar. 2008.

Mahmoud Barhamgi received the BCS degree in computer science from Damascus University (Syria) in 2002 and the MSc degree from the INSA Institute (Lyon, France) in computer science in 2005. He is a PhD candidate at the University Claude Bernard Lyon1 (France). His research interests include web services and web data integration. The focus of his PhD dissertation is on the automatic selection and composition of web services on the Semantic Web. He has published several papers on web services in international conferences including the PVLDB, the WWW, and the ICDE.

Djamal Benslimane received the PhD degree in computer science from Blaise Pascal University. He is a full professor of computer science at Lyon University, France, and a member of the Laboratoire d'InfoRmatique en Image et Système d'information (LIRIS) research laboratory. He has coorganized different scientific events (e.g., local chair of the international conference Notere 2008, FQAS 2004, ICDE-InterDB 2006 Workshop database interoperability, and VLDB-InterDB 2007, etc.), and special issues on context-aware web services in the *Distributed and Parallel Databases International Journal* (Springer, 2007), and service mashups in the *IEEE Internet Computing* magazine (2008). His research interests lie in the areas of distributed information systems, ontologies, and web services.

Brahim Medjahed received the PhD degree in computer science from the Virginia Polytechnic Institute and State University (Virginia Tech) in May 2004. He is an assistant professor in the Department of Computer and Information Science at the University of Michigan-Dearborn. He received *Computer*'s best paper award (Wilkes Award) in 2008 and the 2004 "Outstanding Graduate Research Award" from Virginia Tech's Department of Computer Science. He guest edited a special issue of the *ACM Transactions on Internet Technology* on Semantic Web services. He is on the editorial board of the *International Journal of Next-Generation Computing*. He has served on numerous conference program committees. He is the author of more than 60 publications. His research interests include information and data management, service-oriented computing, distributed computing, Semantic Web, and data integration. He is a member of the IEEE, the IEEE Computer Society, and the ACM.