Modeling and Discovery of Data Providing Services

Roman Vaculín¹, Huajun Chen², Roman Neruda¹, Katia Sycara³

¹ Institute of Computer Science, Academy of Sciences of the Czech Republic
(vaculin,roman}@cs.cas.cz)
² College of Computer Science, Zhejiang University, Hangzhou, 310027, China
(huajunsir@zju.edu.cn)
³ The Robotics Institute, Carnegie Mellon University (katia@cs.cmu.edu)

Abstract

Web Services providing access to datasources with structured data have an important place in the SOA. In this paper we focus on modeling and discovery of generic data providing services (DPS), with the goal of making data providing services available for interactions with service requesters in contexts such as service composition and mediation. In our model RDF Views are used to represent the content provided by the DPS. A characterization of match between description of DPS as RDF Views and the OWL-S service request is specified, based on which we developed a flexible matchmaking algorithm for discovery of data providing services. Finally, we propose a realization of the DPS using a SOAP version of the SPARQL protocol and a dynamic configuration interface allowing easy interactions of service requesters with data providing services.

1 Introduction

Integration of conventional data sources with web services composition and discovery techniques is a rather neglected topic. This is a little bit surprising, especially if we consider that only some part of data and functionalities is available in the form of WS interfaces, and the remaining potentially useful data, residing in the databases, remain invisible to the composition and discovery components. Existing composition and discovery approaches assume that a Web Service call (operation) has a form of a simple transformation that consumes some specific set of inputs and produces a specific set of outputs. A data service, on the contrary, represents a set of potential transformations depending on the database schema and data it stores, and communicates with requesters by means of queries and result datasets. Consider, for example, the difference between a nameLookup service that returns a student’s name given its student’s ID (SID), and the database containing information about students, including their name, SID, age and gender. The nameLookup service represents a mapping from SID to name (SID → name), while in the case of the students database, many possible mappings can be obtained by formulating an appropriate query (e.g., SID → name, name → gender, SID → age, etc.). Of course, the data service can be seen as a transformation query → dataset, but this is of no use for discovery and composition purposes since it provides no information about the stored data.

In this paper, we focus on the problem of representation and discovery of a generic data providing service (DPS), with the goal of making it useful in the context of service composition and interactions with service requesters. We assume that a data providing service provides an access to one or more possibly distributed and heterogeneous databases, or to an RDF store. In order to deal with schemas and data heterogeneity, we suggest to describe data sources as views over a shared mediated schema [11]. Specifically, the local schema of each data source is mapped to concepts of a shared OWL ontology, and its terms are used to define RDF Views [5] describing the data sources in the mediated schema. In essence, we model a data source as a set of RDF Views, with each RDF View describing relations and constraints related to some fragment of the data source content (e.g., one table in a relational database). We use the term DPS Profile to describe capabilities of the DPS which the DPS wants to publish to the outside world, mainly for the discovery purposes. We demonstrate the use of RDF Views in the semantic web services discovery component. We describe how the degrees of match between a service request...
and the DPS Profile can be defined and we devise a matchmaking algorithm for discovery of data providing services.

Using RDF Views expressed in terms of a shared ontology has several advantages. (1) The use of views over a mediated schema known from traditional database integration area [11] allows us to overcome the problem of data and schemas heterogeneity. (2) RDF Views can be used to figure out what possible functional mappings can be computed by a given DPS. (3) Expressing the mediated schema in terms of the OWL ontology, which is independent of the underlying database technology, allows us to treat various categories of data sources in the same fashion (e.g., relational databases, RDF stores, etc.). (4) Describing the DPS in the DPS Profile as a set of RDF Views which are independent from the real physical schema allows to make only some parts of the data available for the public use via the DPS.

The rest of the paper is organized as follows. In Section 2, we introduce the problem and the necessary terminology. Section 3 defines RDF Views as means for modeling DPS. In Section 4 we give the characterization of matching conditions for DPS. Section 5 describes the matchmaking algorithm. In Section 6 we discuss briefly the DPS realization. Section 7 summarizes related work, and Section 8 contains conclusions and future work.

2 The Overall Problem Setting

Figure 1 shows our overall discovery problem setting. We assume that the service requester needs a service with three inputs of types \( A, B, C \) and requires two outputs of types \( D \) and \( E \) to be returned. We assume that the DPS has registered with the Matchmaker by publishing its advertisement (message 1) which describes the data provided by the DPS. Assuming that the DPS contains data that allow outputs of type \( D \) and \( E \) to be retrieved based on inputs of type \( A, B, C \), we want to achieve the following:

1. Given the search request \( R \) of the form \( A, B, C \rightarrow D, E \) is sent to the Matchmaker (message 2) the DPS needs to be identified as a possible match for this request.

2. After discovering the DPS (message 3), the service requester must be able to communicate with the DPS without having to formulate the appropriate SQL or SPARQL query for retrieving the required data. Ideally, to the service requester the DPS should be presented as an ordinary service realizing the mapping \( A, B, C \rightarrow D, E \), not as a generic data providing service. In terms of OWL-S, the DPS should be presented as a process with three inputs of type \( A, B, C \) and two outputs of type \( D, E \). Thus, the service requester would simply execute the process which would lead to sending appropriate SOAP messages (messages 4 and 6).

3. Given the search request \( R \), the DPS must be able to identify the appropriate query \( Q \) which will allow it to retrieve requested information based on provided inputs.

In the rest of this paper we focus mainly on problem 1, however, we briefly discuss problems 2 an 3 in Section 6.

Before describing the model for DPS representation we review some important OWL-S terminology and introduce new relevant terms. OWL-S [16] is a semantic web services description language. OWL-S covers three areas: the Service Profile describes what the service does in terms of its capabilities and it is used for discovery purposes; the Process Model specifies ways of how clients can interact with the service; the Grounding links the process model to the specific execution infrastructure (e.g., maps processes to WSDL operations and allows for sending messages in SOAP). The elementary unit of the Process Model is an atomic process, which represents one indivisible operation that the client can perform by sending a particular message to the service and receiving a corresponding response. Processes are specified by means of their inputs, outputs, preconditions, and effects (IOPEs).

The Service Profile supports description of several perspectives of the service which are relevant for the discovery purposes. Most importantly, it describes the service from the functional perspective by means of its IOPEs in a similar fashion as processes are defined. The Service Profile is used as an advertisement when publishing the service capabilities to the discovery service. On the other hand, a service request describes required capabilities of a searched service.

For purposes of this paper a basic service request \( R \) is a tuple \( R = (I, O) \), where \( I \) and \( O \) are sets of typed input and output parameters of the requested service, i.e., \( I = \{ (?\nu, T) \mid ?\nu \in \text{Var}, T \in \text{Types} \} \), \( O = \{ (?\nu, T) \mid ?\nu \in \text{Var}, T \in \text{Types} \} \).
Types}, \( Var \) is a set of input and output names (we assume each input and output name to be unique) and \( Types \) is a set of types (either primitive XSD types or OWL classes defined in some ontology). We use \( Var_I = \{ ?v \mid (?v, _) \in I \} \) and \( Var_O = \{ ?v \mid (?v, _) \in O \} \) to denote the set of input names and output names respectively. \( Var_{Free} \) is used to denote free variables which are not used in \( Var_I \) or \( Var_O \), i.e. \( Var_{Free} = (Var_I \cup Var_O) = \emptyset \). For information gathering services, oftentimes additional constraints for their inputs and outputs need to be satisfied. Therefore we define the extended service request as a tuple \( R = (I, O, C_I, C_O) \) where \( C_I \) and \( C_O \) are sets of additional constraints that must hold for inputs and outputs respectively. We define \( C_I \) as a set of RDF triples (RDF graph) in which some nodes can be replaced by variables from \( Var_I \cup Var_{Free} \), and \( C_O \) as a set of RDF triples in which some nodes can be replaced by variables from \( Var_I \cup Var_O \cup Var_{Free} \). Variables from \( Var_{Free} \) are treated as existentially quantified.

**Example:** The following example demonstrates an extended service request for a service that consumes an identifier of a student as an input and returns the name of the school that the student attends:

\[
R = (I = \{ (?\text{studentID}, \text{xsd:string}) \}, O = \{ (?\text{schoolName}, \text{xsd:string}) \}, C_I = \{ (?\text{studentID}, \text{rdf:type}, \text{Student}) \}, C_O = \{ (?\text{s}, \text{rdf:type}, \text{School}), (?\text{s}, \text{foaf:name}, \text{schoolName}), (?\text{studentID}, \text{attends}, ?\text{s}) \})
\]

The example demonstrates that extended requests can capture more precisely the constraints imposed on inputs and outputs and also the relations between inputs and outputs. Typically, OWL-S based matchmakers work with basic service requests. However, extended requests seem to be more suitable for information gathering services and for the purposes of matching degrees characterization.

A **data source** is any component that provides access to structured data by means of some query interface. A **data providing service (DPS)** is a component which encapsulates one or more data sources and makes it available as a WS interface (e.g., as a set of WSDL operations). A **DPS Profile** is a description of capabilities of some data providing service for the discovery purposes. In general, the content of the DPS Profile is derived from the OWL-S Service Profile. The DPS Profile differs from the generic Service Profile in the description of functional properties of the service: the IOPEs are replaced by RDF Views which are more suitable for describing the schema of the DPS.

### 3 Modeling the Data Providing Service

We model the DPS as a set of RDF Views. This idea was inspired by work in the database semantic integration area [6], where RDF Views are used for purposes of heterogeneous databases integration.

**Definition 1 (RDF View):** Let \( Var \) be a set of variable names, \( X \subseteq Var, \ Y \subseteq Var, \ X, \ Y \) finite. An **RDF View** is an expression of the form \( R(\vec{X}) : -G(\vec{X}, \vec{Y}) \).

\( R(\vec{X}) \) is called the **head** of the view with \( R \) being a relational predicate. \( G(\vec{X}, \vec{Y}) \) is called the **body** of the view. \( G \) is an RDF graph with some nodes replaced by variables from \( \vec{X} \) and \( \vec{Y} \). Variables in \( \vec{X} \) are called **distinguished variables**, and variables in \( \vec{Y} \) are called **existential variables**.

The basic idea of RDF Views is the following. In the context of relational databases, the head of an RDF View corresponds to one relational table: the relational predicate of the head corresponds to the table name, and distinguished variables correspond to column names. The body of each view specifies how the data from the corresponding table relate to relations and concepts in the shared ontology.

Figure 2 demonstrates the use of RDF Views for a data source containing data about a simplified school domain. The lower part of the figure (Data Source Layer) displays relational tables containing data which the DPS uses to answer possible queries. The figure shows only table names and names of table columns. The middle part of the figure shows three RDF Views which establish mappings between relational tables and the shared ontology. Finally, the top part of the figure shows the RDF Views represented as an RDF graph. In the top right corner relevant OWL classes are defined in the UML notation. For example, the View1, maps each row in the **Person** class. The variable \(?y1\) is an existential variable, which in this case corresponds to the URI (**rdf:id**) of the **Person** class. Notice, that the View2 uses existential variables \(?y2\) and \(?y3\) to establish a relation between a **Student** instance \(?y2\) and a **School** instance \(?y3\), expressing that a student \(?y2\) attends a school \(?y3\) \((?y2, \text{attends}, ?y3))\). Also, notice that in some cases the database provides all data which define the corresponding OWL class. This is the case for the **Person** class, for which all its properties (namely **foaf:name**, **foaf:age** and **foaf:gender**) are provided by the database and RDF Views. On the contrary, in the case of the **School** class, the coverage is not complete, since information about its address (**property address**) is missing in the database. Distinguishing different degrees of coverage of OWL classes by data provided by the DPS will play a significant role in the matchmaking.

Although we introduced RDF Views in the close relationship with some existing database schema, it is important to realize the usefulness of RDF Views as a standalone description of the schema of the DPS. RDF Views are independent from the physical schema and from the particular technology used in the datasource. We can simply see each RDF View as a specification of some data schema fragment and its mapping to the shared ontology. The RDF View tells us which variables are directly provided by the data source (distinguished variables) and how do these pieces of data relate to OWL instances in the mediated schema. RDF Views provide enough information about the structure of
Figure 2. RDF Views for a DPS providing data about a school domain

data stored in the DPS for the discovery purposes as well as for the query identification and answering.

4 Discovery of DPS

Our approach to discovery of DPS builds on the work of Paolucci [14] describing the matchmaking of OWL-S Web Services. We extend the work by allowing advertisements in the form of DPS Profiles in addition to OWL-S Profiles.

4.1 Matchmaking problem

The matchmaking problem can be formulated as follows: given the service request \( R \) and a set of published service advertisements \( \text{Advertisements} \), find the set \( \text{Match} \), \( \text{Match} \subseteq \text{Advertisements} \) such that each \( A \in \text{Match} \) represents a service capable of satisfying the request \( R \). The matching algorithm should be flexible, efficient and should minimize false positives and false negatives. In [14], the ability of service to satisfy a request is defined by means of relations between input and output types of advertisements and requests. A service can satisfy a request, if it produces at least all requested outputs and does not need any other inputs than those provided by the requester. The following definition gives a precise characterization of match for a basic service request and the OWL-S service advertisement.

**Definition 2 (Matching conditions for conventional services):** For a basic request \( R = (I_R, O_R) \) and a service advertisement \( A = (I_A, O_A) \) (we consider only inputs and outputs and abstract from any other features available in the Service Profile) the following conditions need to hold for \( A \) to satisfy \( R \):

1. \( \forall (i_o, T_{i_o}) \in I_A \ \exists (o_i, T_{o_i}) \in O_A \) such that \( T_{i_o} \approx T_{o_i} \)

2. \( \forall (i_o, T_{i_o}) \in I_R \ \exists (o_i, T_{i_o}) \in O_R \) such that \( T_{i_o} \approx T_{o_i} \)

Relation \( \approx \) defines a match between two types. We say that there is a match between two types \( T_{o_i} \) and \( T_{i_o} \) if one of the following conditions hold: (1) \( T_{o_i} = T_{i_o} \) or \( T_{o_i} \text{ subclassOf } T_{i_o} \) (exact match); (2) \( T_{o_i} \text{ subsumes } T_{i_o} \) (plug in match); (3) \( T_{o_i} \text{ subsumes } T_{i_o} \) (subsume match). We say that there is no match (or match failed) if none of the conditions holds for two types.

The three possible conditions of the \( \approx \) relation can be seen as degrees of match with the decreasing quality. An exact match is the most preferable, followed by the plug in match and subsume match. In [14] the characterization of match from Definition 2 is used for computing the overall degree of match between the request \( R \) and an advertisement \( A \) and also for computing the set \( \text{Match} \) ordered according to the degree of match.

4.2 Matching Characterization for DPS

For DPS Profiles the characterization from Definition 2 cannot be used directly. First of all, the DPS Profile of the form \( P_{DPS} = \{ V | \text{an RDF View} \} \) does not distinguish service inputs and outputs. It only describes the schema of available data. What will be an input and what an output of some service call depends on the particular query. Second, although RDF Views describe the schema in terms of types and properties defined in a shared ontology, the information about one class can be spread among several views. Third, the DPS might provide an incomplete coverage of some classes by which we mean that some information about the class is missing in the DPS (such as the address of the School class in Figure 2). To deal with these problems we introduce the concept of Class Views and types.
(a) Basic class views
Person(?name, ?age, ?gender) ->
(\(?Y_1, \text{foaf}\text{\_name}, ?\name\))
(\(?Y_1, \text{foaf}\text{\_age}, ?\age\))
(\(?Y_1, \text{foaf}\text{\_gender}, ?\gender\))
Student(?studentID, ?schoolName) ->
(\(?Y_2, \text{foaf}\text{\_identifier}, \text{\_studentID}\))
(\(?Y_2, \text{foaf}\text{\_age}, ?\age\))
School(?schoolName, ?schoolURL) ->
(\(?Y_3, \text{foaf}\text{\_name}, ?\name\))
(\(?Y_3, \text{foaf}\text{\_homepage}, \text{\_schoolURL}\))

(c) Function dist represented as a matrix

<table>
<thead>
<tr>
<th></th>
<th>Person</th>
<th>Student</th>
<th>School</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Student</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>School</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) Extended class views
Complete coverage:
Person(?name, ?age, ?gender) ->
(\(?Y_1, \text{foaf}\text{\_name}, ?\name\))
(\(?Y_1, \text{foaf}\text{\_age}, ?\age\))
(\(?Y_1, \text{foaf}\text{\_gender}, ?\gender\))
Inherited coverage:
Student(?studentID, ?schoolName) ->
(\(?Y_2, \text{foaf}\text{\_identifier}, \text{\_studentID}\))
(\(?Y_2, \text{foaf}\text{\_age}, ?\age\))
Partial coverage:
School(?schoolName, ?schoolURL) ->
(\(?Y_3, \text{foaf}\text{\_name}, ?\name\))
(\(?Y_3, \text{foaf}\text{\_homepage}, \text{\_schoolURL}\))

Figure 3. Class Views corresponding to RDF Views in Fig. 2

coverage for the DPS Profile.

**Definition 3 (Class View):** Let \(\text{Var}\) be a set of variable names, \(\mathbb{X} \subset \text{Var}\), \(\mathbb{Y} \subset \text{Var}\), \(\mathbb{X}, \mathbb{Y}\) finite. A **Class View** is an expression of the form \(C(\mathbb{X}) = G(\mathbb{X}, \mathbb{Y})\), where \(C\) is the class name defined in the ontology (\(C\) is called the type of the Class View), \(\mathbb{X}\) is the set of distinguished variables, \(\mathbb{Y}\) is the set of free existentially quantified variables, and \(G\) is an RDF graph with some nodes replaced by variables from \(\mathbb{X}\) and \(\mathbb{Y}\). All triples in \(G\) are of the form \(\{?, ?, \text{foaf}\text{-}\text{type}, C\}\), where \(?x \in \mathbb{Y}\) and \(\{?, \text{foaf}\text{-}\text{rdf}\text{-}\text{type}, C\}\). For a Class View \(CV\) we use \(\text{Class}(CV)\) to denote the type of \(CV\).

**Definition 4 (Covered Types):** Let \(\text{DPS} = \{V | V\text{ an RDF View}\}\) be a DPS Profile. **Types covered by \(\text{DPS}\)** is a set \(\text{Types}_{\text{DPS}} = \{CV | CV\text{ a classview}\}\), such that

1. for each class \(C\) used in any \(V \in \text{DPS}\) there is exactly one Class View \(CV \in \text{Types}_{\text{DPS}}\)
2. each \(CV \in \text{Types}_{\text{DPS}}\) contains in its body only triples that were explicitly mentioned in some \(V \in \text{DPS}\)
3. each \(CV \in \text{Types}_{\text{DPS}}\) contains in its body all triples that were used in some \(V \in \text{DPS}\) and that describe the type of the \(CV\).

Class Views and types covered by the DPS Profile offer a different perspective on information presented in RDF Views. For a given \(\text{DPS}\) profile, we gain its associated covered types set \(\text{Types}_{\text{DPS}}\) by reorganizing triples in RDF Views of \(\text{DPS}\) according to what class they belong to. If necessary, variable names might need to be unified to match. Class Views contained in \(\text{Types}_{\text{DPS}}\) represents classes provided by the DPS. Also we can immediately see for what properties the data are available.

Figure 3(a) shows the Class Views gained from RDF Views in Figure 2. When we compare the Class Views in Figure 3(a) with definitions of their corresponding classes (Fig. 2) we can easily see, that all properties of the **Person** class are provided by the DPS, while for the classes **Student** and **School** some data are missing. The case of the **School** class is simple since no data are provided for its **address** property. The situation for the **Student** class is a little bit tricky.

Given only the triples that are directly describing the class **Student** we can see that some properties are missing (namely \(\text{foaf}\text{\_age}, \text{foaf}\text{\_name}\) and \(\text{foaf}\text{\_gender}\)). However, the missing values of these properties can be possibly retrieved from the **Person** Class View, because the class **Student** is defined as a subclass of the **Person** class. It is important to notice, that retrieving data by using the Class View of some superclass does not work always. The reason for this is the fact, that there is no guarantee that for every instance of some class stored in a data providing service the same service will also store information about its superclasses. For example, in our particular database, we cannot be sure that for every student there will also be a corresponding record in the **People** table. In a similar fashion, it is possible to retrieve some missing pieces of data for a given class by using views of its subclasses. However, in this case, the chance of succeeding is even lower than in the case of superclasses, because it is even less likely that given an instance of some class, there will be a corresponding record in the view of its subclass (since subclass covers some subset of the superclass). We conclude this discussion by defining degrees of coverage which describe possible relations between Class Views and corresponding OWL classes.

**Definition 5 (Degrees of coverage):** Let \(CV\) be a Class View of a type \(C\). \(P\) be a set of all properties defined in the ontology, such that \(\forall p \in \text{domain}(p) \subseteq \text{subsumes } C\).\(^1\) We say that a property \(p\) is covered by the \(CV\) if there exists a triple in the \(CV\) body which has \(p\) as its predicate. We define the following degrees of coverage:

1. **complete coverage:** every property in \(P\) is covered by \(CV\)
2. **inherited coverage:** every property in \(P\) is covered by \(CV\) or by some \(CV'\) with the type \(C'\), such that \(C' \text{ superclassOf } C\)
3. **mixed coverage:** every property in \(P\) is covered by \(CV\) or by some \(CV'\) with the type \(C'\), such that \(C' \text{ superclassOf } C\), or \(C \text{ superclassOf } C'\)
4. **partial coverage:** some property in \(P\) is not covered

Figure 3(b) extends Class Views from the Fig. 3(a) with annotations indicating their degree of coverage. The uncovered properties (the stroke through text) and properties inherited from super classes (italicized text) are showed as well.

\(^1\)We assume a local close world assumption to be able to retrieve all properties of \(C\). Also we consider only domain specific properties ad exclude generic properties such as \(\text{rdf}\text{-}\text{id}\).
Degrees of coverage have an impact on the definition of matching relation which is used for assessing the degree of match between types in the service request and in the advertisement. We introduce a match relation \( \sim \) which combines matching degrees of the relation \( \approx \) (exact, plug in, subsume) with degrees of coverage. The \( \sim \) evaluates the match between a class \( C \) and a Class View \( CV \). We use the degree of match \( \approx \) as the main criteria for the overall match assessment and the degree of coverage as the secondary criteria. To simplify the notation we use \( E \) for exact match, \( P \) for plug in, \( S \) for subsume, and we use indexes for indicating the level of coverage. For example, for \( C \sim CV \) we say that the overall degree of match is \( E_c \) (stands for an exact match with complete coverage), if \( C \approx Class(CV) \) is exact and \( CV \) has the complete coverage. We use the operator \( > \) to define goodness of the overall match degree, e.g. \( E_i > E_m \) expresses that an exact match with an inherited coverage is better than an exact match with mixed coverage. The overall matching degrees hierarchy is defined as follows:

\[
E_i > E_0 > E_m > P_i > P_0 > S_i > S_m > E_p > P_p > S_p > fail
\]

The partial coverage is ranked worse than anything else since we want to penalize a situation when some information is returned incomplete.

Finally, we need to talk about the concept of connectivity of Class Views before giving the definition of matching conditions for DPS services. Let us return to the nameLookup service example mentioned in Section 1. The fact that the service realizes the mapping \( SIS \rightarrow name \) means, that given the input of type \( SIS \) the service somehow calculates the output of type \( name \). Now, if we want to realize a service like this by using some DPS, first of all, the DPS has to contain data of both types, namely \( SIS \) and \( name \), and second, there has to exist some relation in the schema of the DPS between these two types, that allows a \( name \) to be retrieved given the \( SIS \). Speaking in terms of Class Views represented as RDF graphs, there must exist a path in the RDF graph between Class Views of type \( SIS \) and \( name \). The following definition formalizes the notion of path between Class Views.

**Definition 6 (Connectivity of Class Views):** Let \( Types_{DPS} \) be a set of Class Views of some DPS, \( G \) the RDF graph representation of \( P_{DPS} \), \( CV_1, CV_2 \in Types_{DPS} \). We say that \( CV_1 \) is connected with \( CV_2 \) in \( G \) (and write \( CV_1 \sim CV_2 \)) if there exists a path \( t \) that connects \( CV_1 \) and \( CV_2 \). Path \( t \) can use only edges defined in \( G \), which includes properties in Class Views and possible IS-A (superclass) relations between instances. The length of path \( t \) is defined as a number of all edges included in \( t \). The distance between \( CV_1 \) and \( CV_2 \) (we write \( dist(CV_1,CV_2) \)) is defined as the length of the shortest path connecting \( CV_1 \) and \( CV_2 \).

The \( dist \) function can be represented as a matrix indexed by Class Views from \( Types_{DPS} \). The matrix contains values of distances in each cell for each combination of two Class Views. Figure 3(c) shows the \( dist \) function represented as a matrix for classes from Figure 2.

**Definition 7 (Matching conditions for data providing services):** For an extended request \( R = \{IR, O_R, C_{IR}, C_{OR}\} \) and a data providing service advertisement \( P_{DPS} \) and its associated set of covered types (Class Views) \( Types_{DPS} \) the following conditions need to hold for \( P_{DPS} \) to satisfy \( R \):

1. \( \forall (\alpha_r, T_o) \in O_R \exists CV_O \in Types_{DPS} \) such that \( T_o \sim CV_O \) and
   
   \[ \text{(a) } \exists CV_I \in Types_{DPS} \exists (\alpha_I, T_i) \in I_R \text{ such that } CV_I \sim T_i \text{ and } CV_I \sim CV_O \]

2. \( C_{IR} \) and \( C_{OR} \) must hold in the RDF graph of the \( P_{DPS} \), i.e., RDF graphs defined by \( C_{IR} \) and \( C_{OR} \) must form sub-graphs of the RDF graph of the \( P_{DPS} \) after possible variables unification.

The condition 1 expresses that for each output in the request there has to exist some Class View \( CV_O \) which matches the requested output type. This means that the DPS must be able to produce every output. In addition to that, the condition 1a requires that the Class View \( CV_I \) must be connected with some Class View \( CV_J \) (\( CV_I \sim CV_O \)) which corresponds to some input provided by the requester. This condition guarantees that every output is somehow related to at least one input. Also it guarantees that it will be possible to generate the query for retrieving the outputs.

The condition 2 makes sure that constraints \( C_{IR} \) and \( C_{OR} \) are satisfied by the DPS. If we replace the extended request \( R \) with the basic service request and omit the condition 2, we get a matching condition for the basic service request.

## 5 Matchmaking algorithm for DPS

Definition 7 gives minimal necessary conditions for the matchmaking between the extended request and the DPS Profile. We use this definition in the matchmaking algorithm. The algorithm works in two phases. In the first phase (Algorithm 1), the new advertisement \( P_{DPS} \) is registered, preprocessed and indexed. The second phase takes care of finding the set \( Match \) for a given service request \( R \) (Algorithm 2).

Both algorithms share the main data structure, \( Advertisements \), which is basically a look-up table (inverted index) in which advertisements together with some auxiliary structures are stored. Stored advertisements are indexed by classes (types) to support a fast retrieval of advertisements which produce or consume a given class. Advertisement are stored together with precomputed degree of match for each relevant class.

Since the main computation burden is carried out during the registration phase, the retrieval of matching advertisements can be done relatively efficiently. In the Step 1 the
Algorithm 1 registerAdvertisement($P_{DPS}$)

1. transform RDF Views in $P_{DPS}$ into set $Types_{P_{DPS}}$ containing Class Views
   1.1 reorganize triples in RDF Views of $P_{DPS}$ according to what class they belong to
   1.2 calculate degrees of coverage for each Class View
   1.3 propagate triples to Class Views along the ISA hierarchy
2. compute the distance matrix $D$ representing the dist function
3. store <$P_{DPS}, Types_{P_{DPS}}, D$> in the inverted index $Advertisements$

Algorithm 2 matchRequest($R = \langle I_R, O_R, C_{I_R}, C_{O_R} \rangle$)

1. $Match \leftarrow$ retrieve all <$P_{DPS}, Types_{P_{DPS}}, D$> from $Advertisements$ that are able to produce all requested output types $T_{O_R}$, <$\gamma_{I_R}, \gamma_{O_R}$> $\in O_R$
2. from $Match$ remove advertisements which have no link to inputs for some output (cond. 1a of Def. 7): use $I_R$ and the distance matrix $D$
3. from $Match$ remove advertisements that do not match constraints $C_{I_R}, C_{O_R}$
4. for each member of $Match$ calculate the total degree of match
5. return sort($Match$)

6 DPS Realization

Our prototype DPS implementation is using the Dartgrid infrastructure [6] as an underlying mechanism for mapping relational database schemas to the mediated schema. Dartgrid supports creation of mappings and RDF Views. It also supports evaluation of SPARQL queries over heterogeneous databases by using query rewriting techniques.

The DPS is implemented as a read only SPARQL [1] endpoint which communicates according to the SOAP SPARQL protocol [7]. We use the Joseki² library to implement the SPARQL endpoint. This makes the DPS accessible to generic SPARQL aware clients. In addition to that we are working on a dynamic configuration interface which allows the runtime interface customization based on the specific client request. Specifically, the configuration interface supports an operation $createDynamicProcess$. This operation has one input parameter, which accepts the extended service request $R$. For the request $R$ the following needs to be done:

1. The SPARQL query $q$ is identified which is able to retrieve the requested output given the provided input. If the matchmaking algorithm was used to find the DPS for the request $R$ the query is guaranteed to exist.
2. A new OWL-S atomic process file $A_R$ is generated with inputs and outputs corresponding to the request $R$. Process $A_R$ is returned to the service requester as the result of the operation $createDynamicProcess$.
3. A grounding file $G_R$ is generated which maps the process $A_R$ to the SOAP SPARQL endpoint. This is achieved by using XSLT transformations that map each input of $A_R$ to corresponding variable of the SPARQL query. The grounding also maps data from the dataset produced by the SPARQL endpoint to outputs of the process $A_R$.

The main advantage of the described approach is that it allows to generate custom interfaces to specific service requests which can be all served by using one generic SPARQL endpoint. This is achieved by having the additional mapping layer in the form of OWL-S grounding and process.

7 Related Work

There are several areas related to our work. We start with the semantic web services frameworks. Existing semantic web services frameworks such as WSMO [15], SAWSDL [8] or OWL-S [16] do not consider data source services directly. To our knowledge this is the first attempt to deal with such a problem. However, a slightly similar approach to describing services by using RDF graph patterns can be found in [13]. Another area related is the web services discovery field. We extend the work of Paolucci et. al. [14], which was recently extended in other directions as well. For example, Klusch et. al. [12] address the problem of flexible matchmaking by using a hybrid matchmaker which combines the semantic approach with approaches from the information retrieval field. Benatallah et. al. in [4] propose an approach that allows a combination of several services to satisfy the service request. Most recently, [3] analyzes the correctness of [14] and suggests an improved algorithm based on bipartite graph matching. Another field closely

²http://www.joseki.org/
related to our work is database discovery. Gravano et. al. [9], for example, are identifying the relevant databases for a given query as the text database discovery problem. The problem of database discovery in the Grid context is addressed by the OGSA-DAI framework described in [2]. In this approach web services are used to access databases in a common middleware allowing uniform access to data resources. The system also allows consumers to discover the properties of structured data stores. Another approach to accessing databases via web services is presented in [10] with the main focus on efficient data transfer for data centric applications. Finally, our work is closely related to query rewriting area in the database domain. The paper [11] gives a very good overview of the topic. Chen et. al. in [5] and [6] propose the Dartgrid as system for semantic integration of heterogeneous databases.

8 Conclusions and Further Work

In this paper we described mechanisms for specification of generic data providing services using RDF Views. Our goal was to provide flexible means of modeling structured data sources such as relational databases or RDF stores and to allow their easy integration within an existing semantic web services infrastructure. We provided characterization of matching conditions for data providing services and based on that we developed an algorithm for flexible matching of service requests with advertisements describing data providing services. Finally, we proposed the protocol for interaction with the data providing service which allows an easy integration with service requesters.

The matchmaking of RDF Views opens several questions. Specifically, the issue of match degrees needs to be addressed in more detail. Our approach to calculation matching degree completely ignores relations between matched outputs and inputs in the RDF graph of the \( \mathcal{P}_{DPS} \). For example, it seems that profiles in which inputs and outputs are closer to each other in terms of the distance as defined in Definition 6 are more likely to be a better match than those with a bigger distance. The same seems to be true for profiles in which inputs of the request are more connected to outputs. A candidate metric which would capture such an intuition can be a weighted average distance between inputs and outputs of the request in the RDF graph of the provider’s profile. However, it is not quite clear how such an information can be combined with degrees of match based solely on matching degrees between inputs and outputs.

On the technical level, the configuration interface of the DPS relies on the OWL-S grounding layer end employs XSLT transformations. An interesting question is if a similar flexibility of the interface can be achieved without adding the semantic layer on top of the standard WS stack.

References