SPARQL-based OWL-S Service Matchmaking

Said Mirza Pahlevi, Akiyoshi Matono, and Isao Kojima

National Institute of Advanced Industrial Science and Technology (AIST)
Grid Technology Research Center, Tsukuba, Ibaraki 305-8568, Japan

Abstract. The increasing availability of Web services has made the accurate and efficient discovery and selection of target services an important issue. OWL-S is a language that semantically describes Web services to facilitate automated service discovery and selection. This paper proposes an OWL-S service matchmaking mechanism that utilizes the emerging standard SPARQL. In this mechanism, a service requestor queries service repositories via a matchmaker by using the SPARQL query language. The matchmaker performs service matching and ordering with the help of a SPARQL query engine. Using SPARQL as the query language offers several advantages that are not provided by existing matchmaking systems, allowing complex service matching and reducing the difficulty of query formulation. In addition, it makes the architecture of the matchmaking system loosely-coupled and that of the matchmaker lightweight.

1 Introduction

One of the primary goals of semantic Web services is to enable automatic discovery and selection of the services. Automatic discovery is an automated process for locating a Web service that provides a particular class of service capabilities while adhering to client-specific constraints. To discover a service, a software agent needs a computer-interpretable description of the service and a means by which to access it. OWL-S [1], WSMO [2] and WSDL-S [3] are examples of languages and models that semantically describe Web services.

In this paper, we focus on using OWL-S for describing Web services because we adopt a matching method used by many service matchmakers that is based on the OWL-S service description. OWL-S describes the Web services in terms of the capabilities offered based on the Web Ontology Language (OWL) [4].

In a typical scenario, a service provider describes advertised services using an OWL-S compliant ontology and submits the advertisements to a Web service repository. A service requester queries the repository by creating a service request using the ontology. The repository matches registered advertisements to the request by performing inferences on the concept hierarchy and orders the results based on the degree of match between the request and advertisements. Since queries are formulated using an ontology rather than a high-level query language, queries are quite difficult to compose and the repository usually only supports simple queries. Furthermore, the system needs to implement a proprietary query engine because each system uses a different matching method.

The recently approved standard WS-Resource Framework (WSRF) [5] defines conventions related to managing Web services, which improves several aspects of these services to make them more adequate for grid applications. It defines a *WS-Resource* that describes the relationship between a service and a resource. In this environment, resources are discovered by identifying the Web services that interact with them. Resource discovery in the grid is currently based on text matching [6]. Text matching-based resource discovery, however, is not a feasible solution for grids having a large number of resources with different capabilities distributed across different organizations.

This paper proposes a new OWL-S service matchmaking mechanism based on the SPARQL query language [7] that is able to support semantic resource matching within a grid. We envision service repositories with a SPARQL query processing capability[1] and a matchmaker that serves requester queries. A service provider uses the OWL-S ontology to describe a service, but a service requester formulates a query using SPARQL sent to a matchmaker.

On receiving a requester query, the matchmaker forwards it to a repository and receives query results. The matchmaker needs to order results because either they are inherently unordered or they are ordered in a way that is not based on class subsumption relationships. To this end, the matchmaker rewrites the requester query before sending it to a repository so that query results will contain all information necessary for result ordering. This paper proposes two mechanisms: *query rewriting* and *result ordering* that enable the matchmaker to order query results based on class subsumption relationships. The matchmaker also allows a requester to set an ordering preference based on service characteristics.

Using SPARQL to formulate a service request provides the following advantages.

- *Imposing no restrictions on the repository system.* Since SPARQL is a standard query language, the repository does not need to identify whether a query comes from a matchmaker. Furthermore, the repository can use existing SPARQL query engines [8].
- *Loosely-coupled architecture.* A matchmaker can easily switch from one repository to another or send a simultaneous query to multiple repositories.
- *Lightweight matchmaker.* The matchmaker must only perform lightweight tasks such as query rewriting and result ordering, while the heavyweight task of service matching is provided by a "standard" SPARQL query engine in a repository system.
- *Complex service matching.* A requester can use the rich SPARQL constructs (e.g., optional, filtering, and union) during service requests. Furthermore, SPARQL inherently supports some sorts of grid resource matching.
- Ability to use standard *SPARQL query results in XML format* [9] and *protocols* [10] in communications between a requester and the matchmaker and between the matchmaker and the repository.

[1] The SPARQL query engine could be attached to the matchmaker or a stand-alone module. However, we adopted this approach because we envision a future in which many RDF repositories are located behind a SPARQL query engine.

The rest of the paper is organized as follows. Section 2 briefly describes the semantic markup for Web services, OWL-S, and the SPARQL query language. Section 3 reviews related work. Section 4 describes the proposed matchmaking mechanism. Section 5 describes semantic resource matching in the Grid. The final section discusses advanced use of SPARQL for service and resource matching, and outlines future activities.

2 Background

2.1 OWL-S

OWL-S [1] is an ontology of service concepts for describing the properties and capabilities of web services in a machine interpretable form. It consists of three main parts/classes: the `ServiceProfile`, `ProcessModel`, and `ServiceGrounding`. The `ServiceProfile` is used for advertising and discovering services. It includes three basic types of information: what organization provides the service (e.g., contact information), what function the service computes (e.g., inputs and outputs) and what characteristics the service has (e.g., service category). The `ProcessModel` gives a detailed description of a service's operation and the `ServiceGrounding` specifies the details of how to access the service.

A class `Service` simply binds the three parts together into a unit via three object properties: `presents` (to the `ServiceProfile`), `describedby` (to the `ProcessModel`) and `supports` (to the `ServiceGrounding`).

Since `ServiceProfile` is used for service discovery, this paper mainly deals with the class or classes that extend the `ServiceProfile`.

2.2 SPARQL Query Language

SPARQL consists of a query language [7], a means of conveying a query to a query processor [10], and the XML format in which query results will be returned [9]. The specification is currently under discussion as a W3C Working Draft but the availability of several SPARQL query engines [8] means that the specification is getting stable for the practical use.

The SPARQL query language is based on matching graph patterns. The simplest pattern is the *triple pattern* consisting of subject, predicate, and object. Any or all of subject, predicate, and object values may be replaced by a variable prefixed with either "?" or "$". Combining triple gives a *basic graph pattern*, where an exact match to a graph is needed to fulfill a pattern.

The query below finds a service (bound to variable $?s$) that has a `serviceProfile` (bound to $?o$) whose types/classes of its two `pr:hasInput` values (bound to $?in_1$ and $?in_2$) are `:Input1` and `:Input2`, respectively.

```
PREFIX sr: <http://www.daml.org/services/owl-s/1.1/Service.owl#>
PREFIX pr: <http://www.daml.org/services/owl-s/1.1/Profile.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
BASE <http://www.owl-ontologies.com/unnamed.owl#>
```

```
SELECT ?s WHERE ?s sr:presents ?o.
?o pr:hasInput ?in1. ?in1 rdf:type :Input1.
?o pr:hasInput ?in2. ?in2 rdf:type :Input2.
```

The `PREFIX` keyword associates a prefix label with a URI and `BASE` keyword defines the based URI to resolve relative URIs. The `SELECT` clause indentifies the variables to appear in the query results and the `WHERE` clause contains a graph pattern.

SPARQL provides other graph patterns for complex matching such as *optional graph pattern* for matching semi-structured RDF graphs, *alternative graph pattern* for combining graph patterns so that one of several alternative graph patterns may match, and *patterns on named graphs* where patterns are matched against named graphs. It also provides value constraints which restrict RDF terms in a solution. SPARQL includes a number of syntax shortcuts that simplify the writing of patterns. For example, triple patterns sharing the same subject and predicate can be written using the ",“ notation (e.g., triples with predicate `pr:hasInput` in the query above can be written as $?o$ `pr:hasInput` $?in_1$`,` $?in_2$`.`) and `rdf:type` can be replaced with keyword "a“.

3 Related Work

3.1 OWL-S Semantic Matchmaking

Over the last few years, researchers have proposed several semantic matching methods. [11] is the first method that provides an algorithm for matching service advertisements and requests based on service inputs and outputs. An advertisement matches a request when *all request outputs* are matched by advertisement outputs, and *all advertisement inputs* are matched by request inputs. This characteristic guarantees that the matched service provides all outputs requested by the requester, and that the requester provides all input required for correct operation to the matched service.

More specifically, an advertisement output outA exactly matches a request output outR when it is identical with outR or when outR is an immediate subclass of outA. If outA subsumes outR then this is considered a Plug-in match; conversely, if outR subsumes outA, then this is considered a Subsume match. If no subsumption relationship appears between outA and outR, then this is considered a Fail match. Matching of inputs involves a similar computation method but reverses the order of request and advertisement.

The following matching methods are based on [11]. [12] adds a new similarity metrics intersection to detect when an advertisement satisfies only some features of a request, whereas [13] further examines relationships among classes and their property classes. [14] and [15] add syntactic similarity and [16] adds service category matching and user-defined matching to semantic matching. [17] proposes a rather different matching approach. It computes the best combinations of Web services to provide the most satisfactory request outputs while requiring the least possible inputs that are not provided in the request.

All matchmaking systems, however, use ontology or description logic in service requests instead of a high-level query language such as SPARQL. Furthermore, because the systems have different matching methods, they use proprietary matching engines.

3.2 Resource Discovery in the Grid

The grid middleware, Globus Toolkit (GT) 4.0 [18], provides a service, called the Index Service [6], that facilitates grid resource monitoring and discovery. This service collects data from various sources and publishes them as XML documents. This service provides structured (XML) data but without semantic constraints. S-MDS [19] is a framework to support automatic service discovery and monitoring in the grid. It describes the metadata of WS-Resources using OWL-S ontology and provides an efficient mechanism to aggregate and maintain the ontology instances. S-MDS, however, is a kind of an index service rather than a matchmaking system. Our matchmaking system can work with S-MDS to provide semantic resource discovery and selection in the grid.

Classad [20] is a matchmaking framework to resource management in distributed environment with decentralized ownership of resources. In this framework, a service advertisement and request are formulated using a semi-structured data model called classified advertisements (classad) which consists of attribute-value pairs. A matchmaker matches the advertisement and the request based on constraints specified by attributes in the classads. This framework, however, only allows a *bilateral match*, that is, matching a single request with a single resource. [21] extends the work so that a single request can be matched with multiple (types of) resources (*gang match*).

Redline [22] is a grid matchmaking system that reinterprets matching as a constraint problem and exploits constraint-solving technologies to implement matching operations. It provides two new matching methods which are not supported by classad: *set match* and *congruous match*. The former matches between a request and a resource set with particular aggregated properties whereas the latter matches between multiple requests and a resource.

Ontology-based Matchmaker (OMM) [23] is an ontology-based resource selector for solving resource matching in the grid. Resources and requests are described by (different) ontologies and they are matched using matching rules. The matchmaker can be easily extended by adding vocabularies and inference rules to include new concepts about resources. Like classadd, OMM supports bilateral match and gang match.

Classad and Redline, however, are based on text matching rather than semantic matching. Furthermore, they do not use ontology for service advertisement and request. On the other hand, OMM supports semantic matching for resource discovery. It, however, uses proprietary ontology to describe resources. Similar to the OWL-S matchmakers, a resource request is formulated using ontology and a special matching engine is needed to process the request.

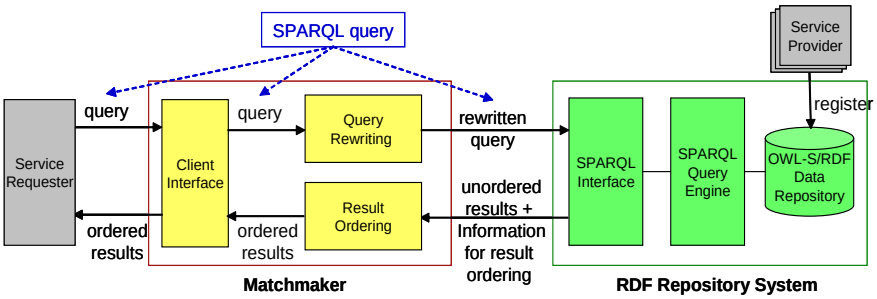

Fig. 1. System architecture

4 SPARQL-based OWL-S Service Matchmaking

Fig. 1 presents the basic architecture of the proposed matchmaking system. A requester sends a SPARQL query as a service request to a matchmaker through a client interface. The matchmaker rewrites the query and sends the rewritten query to a repository system. The query is rewritten such that matching results returned by the repository will contain the information necessary for result ordering. Matching results are ordered based on the degree of match between the query and its resulting services and the ordered results are returned to the requester.

A SPARQL query treats an RDF graph purely as data, i.e. it does not interpret RDF schema information or entailments. The query, however, can be issued to an inferred graph, thus allowing query over the entailments. In this paper, we assume that in addition to storing OWL-S service instances with their schema, an RDF repository provides an inferred graph of the instances. The former is called *uninferred Graph* (*uGraph*), whereas the latter is called *inferred Graph* (*iGraph*)[2]. *iGraph* is a query's default graph, and *uGraph* is accessed through the SPARQL named graph mechanism.

The rest of this paper will apply the following definitions. A graph pattern in the definitions is matched against *iGraph*. For simplicity, SPARQL prefix keywords and labels are always omitted from a query and graph pattern and syntax shortcuts are always used.

Definition 1. Type-matching query graph pattern (type-matching pattern) $pt =$ $\{?o\ p\ ?var_1, \ldots, ?var_k .\ ?var_1\ \mathtt{a}\ c_1 .\ \ldots\ ?var_k\ \mathtt{a}\ c_k .\}$ *is defined as a graph pattern that matches* k *values of predicate* p *while restricting values to those from types/classes* $c_1, \ldots, c_k$. p *is called the* target property/predicate *and* $c_1, \ldots, c_k$ *is called the* target property value (tpv) classes[3].

Definition 2. Relative classes *of class* c *include all its superclasses and subclasses (including* c*).*

[2] When a resource is defined as an instance from class c in *uGraph*, it is inferred as instances from c's subclasses in *iGraph*.

[3] Henceforth, we will use *type* and *class* and *property* and *predicate* interchangeably.

Definition 3. *Given type-matching pattern pt: Instance s is said to be a* subclass instance *of target property p if s matches pt. In this case, values of p (of s) val_1, ..., val_k (which are bound to var_1, ..., var_k of pt) are from a subclass of (tpv classes) c_1, ..., c_k, respectively.*

Definition 4. *Given type-matching pattern pt: Instance s is said to be a* superclass instance *of target property p if s does not match pt and values of p (of s) val_1, ..., val_k are from a relative class of (tpv classes) c_1, ..., c_k, respectively. In this case, one (or more) of the relative classes is a superclass of a tpv class (excluding the tpv class itself).*

Definition 5. *Given type-matching pattern pt: Property p of instance s is said to be* an inclusive property *(with respect to pt) if s is a subclass or superclass instance of target property p and p has (exacly) k values. Otherwise p is said to be* a non-inclusive property*.*

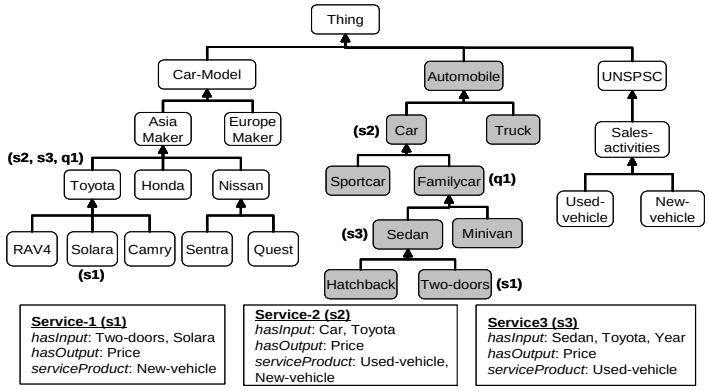

Fig. 2. A fragment of ontologies and service instances

Example 1 Fig. 2 presents a fragment of some domain-specific ontologies and service instances. The figure also presents properties and their value types of the instances. For example, "`hasInput=Two-doors, Solara`" of s_1 denotes that s_1 has a property `hasInput` with two values whose types are `Two-doors` and `Solara`, respectively.

q_1 = `SELECT` $?s$ `WHERE` $?s$ `presents` $?o$.
$?o$ `hasInput` $?in_1$, $?in_2$. $?in_1$ `a Familycar.` $?in_2$ `a Toyota.`

Query q_1 above contains type-matching pattern {$?o$ `hasInput` $?in_1$, $?in_2$. $?in_1$ `a Familycar.` $?in_2$ `a Toyota.`}. s_1 is a subclass instance of `hasInput` because it matches the query. Since it has two `hasInput` values, the `hasInput`

is an inclusive property. Similarly, s_3 is a subclass instance of `hasInput` because it matches the query. Its `hasInput` property, however, is a non-inclusive one because the property has three values of which one type is `Year`. However, s_2 is a superclass instance of `hasInput` because it does not match q_1 and its `hasInput` value types are `Car` and `Toyota`, which are from the relative classes of `Familycar` and `Toyota`, respectively. The `hasInput` property is an inclusive one.

4.1 Query Rewriting

Upon receiving query results from a repository, the matchmaker orders them based on *the degree of match* between concepts included in the query and in matched services. This paper uses the degree of match defined in [11] (see Section 3.1). The results, however, do not contain the data or information necessary for the ordering task. For example, applying q_1 to *iGraph* (of ontologies shown in Fig 2) returns s_1 and s_3, but the results lack the following information:

- *Superclass instances of property* `hasInput`. q_1 leaves s_2 as an unmatched service because s_2 is a superclass instance of `hasInput`. Lack of this data will prevent application of the Subsume match calculation.
- *Immediate subclass/superclass information.* For example, `Two-doors` (which is the `hasInput` value type of s_1) is not an immediate subclass of `Familycar`, unlike `Sedan` (which is the `hasInput` value type of s_3). Lack of this information will prevent application of Exact and Plug-in match calculations.
- *Property inclusiveness information.* For example, `hasInput` of s_1 is an inclusive property, whereas that of s_3 is a non-inclusive one. Lack of this information will prevent application of the Fail match calculation.

This paper examines the following requester query[4].

`SELECT` $?s$
`WHERE otherGP.` $?s$ `presents` $?o$`.`
$?o$ p_1 $?var_{1,1}$`,` ..., $?var_{1,k_1}$`.` $?var_{1,1}$ `a` $c_{1,1}$`.` ... $?var_{1,k_1}$ `a` c_{1,k_1}`.`
...
$?o$ p_m $?var_{m,1}$`,` ..., $?var_{m,k_m}$`.` $?var_{m,1}$ `a` $c_{m,1}$`.` ... $?var_{m,k_m}$ `a` c_{m,k_m}`.`

The query contains a set of type-matching patterns and other graph patterns (`otherGP`). `otherGP` can be optional, alternative, and/or value constraint graph patterns [7].

The query rewriting procedure involves rewriting each type-matching pattern such that pattern solutions will contain superclass and subclass instances of the target property, class subsumption information, and target property inclusiveness information. More formally, given the following query, where $1 \leq i \leq m$,

`SELECT` $?s$
`WHERE otherGP.` $?s$ `presents` $?o$`.`
$?o$ p_i $?var_{i,1}$`,` ..., $?var_{i,k_i}$`.` $?var_{i,1}$ `a` $c_{i,1}$`.` ... $?var_{i,k_i}$ `a` c_{i,k_i}`.`

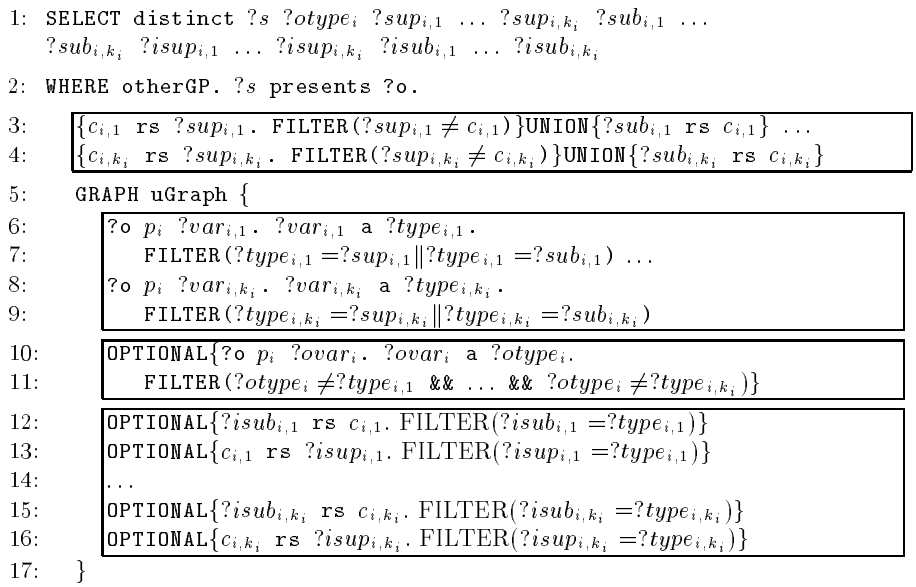

Fig. 3. Query rewriting procedure

The query is rewritten as shown in Fig 3.

The upper part (lines 2–4) is matched against the default *iGraph* while the lower part (lines 6–16) is matched against *uGraph*. Lines 3 and 4 of the upper part extract the relative classes of tpv classes $c_{i,1}, \ldots, c_{i,k_i}$ by using the `rdfs:subClassOf` property (abbreviated as *rs*). Value constraints (`FILTER`) are used to reduce the result size because a class is a subclass and superclass of itself. The lower part can be further divided into three subparts:

1. The first part (lines 6–9) matches instances in which target property value types are (relative) classes matched by the upper part. $?var_{i,j}$ is bound to a target property value and $?type_{i,j}$ is bound to an *uninferred class* (i.e., a class defined in *uGraph*) of the value ($1 \leq j \leq k$). This part corresponds to the type-matching patterns in the original query.
2. The second part (lines 10 and 11) retrieves property inclusiveness information. The optional graph pattern verifies the existence of other p_i values of types other than relative classes. This is done by defining a new variable $?ovar_i$ for the value and excluding relative classes during type matching for the value. Information about p_i property inclusiveness can easily be obtained from the pattern solution by examining the binding value of $?otype_i$. If this is bound to some value, then p_i is a non-inclusive property; otherwise it is an inclusive one (see Example 2 below).

[4] For simplicity, the path length from {?*s* `presents` ?*o*} to type-matching patterns is set here at zero, but the path could be set at any arbitrary length.

3. The third part (lines 12–16) retrieves immediate subclass and superclass information. This part uses the `rdfs:subClassOf` property combined with an optional graph pattern to bind the immediate subclass and superclass of tpv class $c_{i,j}$ to variable $?isub_{i,j}$ and $?isup_{i,j}$, respectively $(1 \leq j \leq k_i)$.

Let s_o and s_r be a set of services contained in the solutions to a requester query and the corresponding rewritten query, respectively. Then, $s_o \subseteq s_r$ and $s_r - s_o = s_{sup}$, where s_{sup} is a set of superclass instances of the target properties. The proof is that the lower part of the rewritten query does not add any restrictions to the services bound to variable $?s$, except that the type of p_i value (which is bound to $?var_{i,j}$) *must* be $c_{i,j}$ (or its subclass) or a superclass of $c_{i,j}$. The former restriction is identical to that in the original query while the latter *relaxes* the query to include superclass instances. Note that the optional graph patterns in the lower part do not give any matching restrictions while `otherGP` is matched against the default graph as in the original query.

Example 2 q_1 shown above is rewritten as follow and the binding solutions are shown in Table 1.

```
SELECT distinct ?s ?otype1 ?sup1 ?sup2 ?sub1 ?sub2 ?isup1 ?isup2
?isub1 ?isub2
WHERE ?s presents ?o.
   {Familycar rs ?sup1. FILTER(?sup1 ≠ Familycar)}
     UNION{?sub1 rs Familycar}
   {Toyota rs ?sup2. FILTER(?sup2 ≠ Toyota)}
     UNION{?sub2 rs Toyota}
   GRAPH uGraph {
     ?o hasInput ?var1. ?var1 a ?type1.
       FILTER(?type1 =?sup1||?type1 =?sub1).
     ?o hasInput ?var2. var2 a ?type2.
       FILTER(?type2 =?sup2||?type2 =?sub2).
     OPTIONAL{?o hasInput ?ovar1. ?ovar1 a ?otype1.
       FILTER(?otype1 ≠?type1 && ?otype1 ≠?type2)}
     OPTIONAL{?isub1 rs Familycar. FILTER(?isub1 =?type1)}
     OPTIONAL{Familycar rs ?isup1. FILTER(?isup1 =?type1)}
     OPTIONAL{?isub2 rs Toyota. FILTER(?isub2 =?type2)}
     OPTIONAL{Toyota rs ?isup2. FILTER(?isup2 =?type2)}
   }
```

Table 1. Binding solutions

	$?s$	$?otype_1$	$?sup_1$	$?sup_2$	$?sub_1$	$?sub_2$	$?isup_1$	$?isup_2$	$?isub_1$	$?isub_2$
b_1:	s_1				Two-doors	Solara				Solara
b_2:	s_2		Car			Toyota	Car			
b_3:	s_3	Year			Sedan	Toyota			Sedan	

There are three binding solutions b_1, b_2, and b_3 which bind values in instances s_1, s_2, and s_3, respectively, to some variables. From b_1, the `hasInput` of s_1 is an inclusive property because $?otype_1$ is unbound. $?sub_1$ binding value indicates that the first `hasInput` value type of s_1 (i.e., `Two-doors`) is a subclass of `Familycar` while $?sub_2$ and $?isub_2$ values indicate that the second value type of s_1 (i.e., `Solara`) is an immediate subclass of `Toyota`. Similarly, from b_2, the `hasInput` of s_2 is an inclusive property, and the first `hasInput` value type (i.e., `Car`) is an immediate superclass of `Familycar` while the second value type is equal to `Toyota`. The last solution b_3 indicates that the `hasInput` of s_3 is a non-inclusive property because $?otype_1$ is bound to some value (i.e., `Year`). The first `hasInput` value type of s_3 is an immediate subclass of `Familycar` while the second value type is equal to `Toyota`.

It is important to note that subclass and superclass variables are only bound to classes that are "actually used" in service instances. For example, $?sup_2$ is unbound because no `hasInput` value types of matched services are superclasses of `Toyota`. This will greatly reduce the size of return results [5]

4.2 Service Ordering

Generally, a requester query consists of one *service function matching* (based on service inputs and outputs) and optionally one or more *service characteristics matching* (based on service characterization according to some domain-specific ontologies). The next sections describe the scoring mechanisms used in matching.

Service Function Scoring Given requester query q containing a service function type-matching pattern

$\{?o$ `hasInput` $?in_1, \ldots, ?in_{k_{in}}.\ ?in_1$ `a` $cin_1.\ \ldots\ ?in_{k_{in}}$ `a` $cin_{k_{in}}.$
$?o$ `hasOutput` $?out_1, \ldots, ?out_{k_{out}}.\ ?out_1$ `a` $cout_1.\ \ldots\ ?out_{k_{out}}$ `a` $cout_{k_{out}}.\}$

and binding solution b from matching results of q's rewritten query: The degree of match between q and b with respect to target property `hasInput` is given by Eq. 1.

$$d_{in}(q, b) = \text{DIN}(cin_1, b.c_1) + \ldots + \text{DIN}(cin_{k_{in}}, b.c_{k_{in}}) \quad (1)$$

$\text{DIN}(cin_i, b.c_i)$ is the degree of match between class cin_i and the corresponding class c_i which is bound to subclass/superclass variables in b. The degree of match could be Exact, Plug-in, or Subsume, where Exact>Plug-in>Subsume. The scoring formula indicates that the larger $d_{in}(q, b)$ the more similar inputs of the matched service (which is bound to variable $?s$ in b) to inputs specified in q.

Fig. 4 shows computation of DIN which is similar to [11], but without the Fail match. b is considered to be a Fail match when the `hasInput` (of a service bound to variable $?s$ in b) is a non-inclusive property. This results from the fact

[5] This is one reason we use the *uGraph*. Another reason is to obtain the uninferred class and immediate subclass information of target property values, which is impossible to do when matching the pattern against *iGraph*.

that the requester is not able to provide the bound service in b all the input required for correct operation.

The degree of match for b with respect to target property `hasOutput` is given by Eq. 2. It uses DOUT function shown in Fig. 5. Different from DIN, DOUT uses subclass relationships to determine Exact and Plug-in matches.

$$d_{out}(q, b) = \text{DOUT}(cout_1, b.c_1) + \ldots + \text{DOUT}(cout_{k_{out}}, b.c_{k_{out}}) \tag{2}$$

```
DIN(inR, inA)
 if inR=inA return Exact
 if inR immediateSuperclassOf inA
   return Exact
 if inR superclassOf inA
   return Plug-in
 otherwise return Subsume
```

Fig. 4. DIN calculation

```
DOUT(outR, outA)
 if outR=outA return Exact
 if outR immediateSubclassOf outA
   return Exact
 if outR subclassOf outA
   return Plug-in
 otherwise return Subsume
```

Fig. 5. DOUT calculation

Finally, summing up the degree of match of inputs and outputs and normalizing the value between 0 and 1 gives the degree of match with respect to the service function (Eq. 3).

$$d_f(q, b) = \frac{k_{out} \times d_{in}(q, b) + k_{in} \times d_{out}(q, b)}{2 \times Exact \times k_{in} \times k_{out}} \tag{3}$$

Note that when $d_{in}(q, b)$ fails, $d_f(q, b)$ also fails.

Example 3 Let Exact=3, Plug-in=2, and Subsume=1. The rewritten query of q_2 below has three binding solutions b_1, b_2, and b_3 that bind s_1, s_2, and s_3, respectively. Matchmaker assigns $d_f(q_2, b_1) > d_f(q_2, b_2)$ and $d_f(q_2, b_3)$ =Fail.

q_2 = `SELECT` ?s `WHERE` ?s `presents` ?o,
?o `hasInput` ?in_1, in_2, ?in_1 `a Familycar.` in_2 `a Toyota,`
?o `hasOutput` ?out. ?out `a Price.`

$d_{in}(q_2, b_1)$ = DIN(Familycar, Two-doors) + DIN(Toyota, Solara) = 5,
$d_{in}(q_2, b_2)$ = DIN(Familycar, Car) + DIN(Toyota, Toyota) = 4,
$d_{out}(q_2, b_1) = d_{out}(q_2, b_2)$ = DOUT(Price, Price) = 3,
$d_f(q_2, b_1) = 0.92$, and $d_f(q_2, b_2) = 0.83$.

Service Characteristics Scoring Service characteristics describe the categories, classifications, and other features owned by a service. A domain-specific ontology may be used to describe these characteristics. For example, the OWL

ontologies of NAICS [24] and UNSPSC [25] can be used to describe service classification and products, respectively. In many cases, it is desirable to let a service requester specify a service-ordering preference based on domain-specific ontology matching. For example, during the car selling service discovery, a requester may give a higher ordering preference to services that sell only new cars or that sell only Toyota cars. In a grid environment, a requester may give a higher ordering preference to grid resources that are managed only by a specific organization.

Given requester query q containing a service characteristic type-matching pattern, where target property p_i specifies service characteristics,

$\{?o\ p_i\ ?var_1, \ldots, ?var_k.\ ?var_1\ \texttt{a}\ cs_1.\ \ldots\ ?var_k\ \texttt{a}\ cs_k.\}$

and binding solution b from matching results of q's rewritten query: The degree of match between q and b with respect to p_i is given by Eq. 4.

$$d_{p_i}(q,b) = \begin{cases} 1 + \frac{\text{DOUT}(cs_1,b.c_1)+\ldots+\text{DOUT}(cs_k,b.c_k)}{Exact \times k} & b \text{ is given a higher preference} \\ \frac{\text{DOUT}(cs_1,b.c_1)+\ldots+\text{DOUT}(cs_k,b.c_k)}{Exact \times k} & \text{otherwise} \end{cases} \tag{4}$$

The value of the dividend that sums up the degree of match of the service characteristic is normalized between 0 and 1 by dividing it with $Exact \times k$. DOUT is used for degree of match calculation because service characteristics are a kind of requested output. By default, b is given a higher ordering preference if the p_i (of the bound service) is an inclusive property[6]. To specify a higher ordering preference to the non-inclusive property, p_i' `ORDER BY (!`p_i'`)` is used[7].

Finally, the total degree of match between q and b with respect to service characteristics specified by target properties $p_1, \ldots, p_m$ is given by Eq. 5. The divisor $2 \times m$ normalizes the match degree between 0 and 1.

$$d_c(q,b) = \frac{\sum_{i=1}^{m} d_{p_i}(q,b)}{2 \times m} \tag{5}$$

Example 4 The rewritten query of q_3 below has two binding solutions $b_1$and b_2 that bind s_1 and s_2, respectively. The matchmaker gives a higher ordering preference to b_1 because s_1 sells only new vehicles (i.e., the `serviceProduct` property of s_1 is inclusive). Putting `ORDER BY (!serviceProduct)` in the query will reverse the ordering preference.

q_3 `= SELECT` $?s$ `WHERE` $?s$ `presents` $?o$`,`
$?o$ `hasInput` $?in_1$`,` in_2`,` $?in_1$ `a Familycar.` in_2 `a Toyota,`
$?o$ `hasOutput` $?out$`.` $?out$ `a Price.`
$?o$ `serviceProduct` $?d$`.` $?d$ `a New-vehicle.`

[6] `ORDER BY` (p_i) can be used to explicitly specify the ordering preference.

[7] Since this construct is only used for the purposes of result ordering, the matchmaker removes it before sending the rewritten query to a repository system.

Overall Scoring and Service Ordering Given requester query q with target properties $p_1, \ldots, p_k$ specifying service characteristics and a set of binding solutions bs from matching results of q's rewritten query. For each binding solution $b \in bs$, the matchmaker calculates the overall degree of match between q and b, $d(q,b)$ $(0 \leq d(q,b) \leq 1)$ using Eq. 6 shown below and orders the solutions in descending order of their degrees of match.

$$d(q,b) = \alpha \times d_f(q,b) + (1-\alpha) \times d_c(q,b), \text{where } 0 \leq \alpha \leq 1 \tag{6}$$

α is a weight that specifies the portion of $d_f(q,b)$ and $d_c(q,b)$ that includes in the total scoring. By appropriately setting the weight value, the ordering scheme not only can meet a user specific ordering requirement, but also can be used for other ordering purposes. For example, by setting $\alpha = 0$ the ordering scheme can be used to order general SPARQL query results. It is important to note that $d(q,b)$ fails when $d_f(q,b)$ fails.

5 Semantic Resource Matching in the Grid

As described earlier, resources in the Grid are accessed via Web services and resource discovery is done by identifying Web services that provide access to the resources. In this sense, if the Web services are described using the OWL-S language, such as proposed in [19], our scheme can be applied for basic resource matching in the Grid, i.e., bilateral match.

In addition, use of SPARQL as the request language enables congruous and gang matches, which are important during grid resource discovery. These matches are made possible because SPARQL allows several graph patterns to be used in a query and allows them to refer to the same common variables. The following example from [21] demonstrates a gang match using SPARQL. The purpose is to match a job (resource request) with two types of resources: workstations and software package licenses. A job that uses the packages needs to allocate both a machine and a license before it can run. Licensing terms may entail that some licenses are valid only on some machines, while others may be valid in certain subnets. Assuming the two resources are described by different RDF graphs and the workstation RDF graph is the default graph, the following query performs a gang match against the two resources. Note that the two graph patterns refer to the same variable *?address*.

```
SELECT ?machine ?license
WHERE ?machine hasArchitecture ?arch. ?arch a Pentium.
   ?machine hasMemory ?mem. FILTER(?mem > 1000).
   ?machine hasAddress ?address. FILTER(?address ="foo.go.jp").
   GRAPH PackageLicense{
     ?license hasValidHost ?address.
     ?license hasApplication ?app. FILTER(?app ="sim_app").
   }
```

It is important to note that bilateral, congruous, and gang matches using our matchmaking scheme are done based on the OWL-S service description that allows semantic matching. This is different from the conventional grid resource discovery systems [20, 21, 6, 22] that use text matching to perform the matching tasks.

6 Discussion and Future Work

The optional graph pattern is one of the interesting features provided by the SPARQL query language. It defines additional graph patterns that do not cause solutions to be rejected if the solutions cannot be matched, but do bind the solutions when they can be matched. This feature is not supported by existing matchmaker systems, but can be used by a service requester for complex input/output and service characteristics matching. For example, it can match a service that provides additional outputs or characteristics to the required ones.

The other interesting feature provided by the SPARQL query language is its value constraints. It restricts solutions by imposing constraints on values that can be bound to variables. This feature is useful, for example, to filter services based on the `serviceName` and `textDescription` of the `serviceProfile` using regular expressions. The language also allows application-specific constraints on values in a solution.

The other feature, alternative graph patterns, can be useful when matching several graph patterns. A pattern can be nested and used together with other graph patterns such as optional ones. In this paper, we do not allow these patterns to be used with type-matching patterns (e.g., to connect two type-matching patterns). A requester should be able to send multiple queries and obtain the same results.

The above graph patterns and value constraints do not affect service ordering. Optional graph patterns may contain type-matching patterns but an ordering preference cannot be assigned to the patterns[8].

We have implemented the matchmaker using Java and use Jena SPARQL query parser [26] for query rewriting and result ordering. The matchmaker also incorporates Jena SPARQL query and inference engines so that it can process RDF data stored in a file system or in a repository that does not have SPARQL query and inference engines.

We are currently expanding work in certain directions. The first is finding ways to support the use of alternative graph patterns for type-matching patterns. To enable this, we must extend the query rewriting and scoring mechanisms. The second is finding ways to support the set match for grid resource discovery. Since SPARQL does not currently provide aggregate functions, the matchmaker is not able to use a SPARQL query engine to perform the set match. A matchmaker can, however, realize a set match by processing query results from a repository.

[8] These patterns are ignored by the query rewriting module.

References

1. http://www.daml.org/services/owl-s/1.1/.
2. http://www.wsmo.org/2004/d2/v0.2/20040306/.
3. http://www.w3.org/Submission/WSDL-S/.
4. http://www.w3.org/TR/owl-features/.
5. http://www.oasis-open.org/committees/wsrf.
6. http://www.globus.org/toolkit/docs/4.0/info/index/.
7. http://www.w3.org/TR/rdf-sparql-query/.
8. http://esw.w3.org/topic/SparqlImplementations.
9. http://www.w3.org/TR/2006/CR-rdf-sparql-XMLres-20060406/.
10. http://www.w3.org/TR/2006/CR-rdf-sparql-protocol-20060406/.
11. Paolucci, M., et al.: Semantic matching of web services capabilities. In: Proc. of the First International Semantic Web Conference on The Semantic Web. (2002) 333–347
12. Li, L., Horrocks, I.: A software framework for matchmaking based on semantic web technology. In: Proc. of the 12th international conference on World Wide Web. (2003) 331–339
13. Kuang, L., et al.: Exploring semantic technologies in service matchmaking. In: Proc. of the Third IEEE European Conference on Web Services. (2005) 226–234
14. Cardoso, J., Sheth, A.: Semantic e-workflow composition. Journal of Intelligent Information System **21** (2003) 191–225
15. Klusch, M., et al.: Owls-mx: Hybrid semantic web service retrieval. In: Proc. of International AAAI Fall Symposium on Agents and the Semantic Web. (2005)
16. Jaeger, M.C., et al.: Ranked matching for service descriptions using owl-s. In: Kommunikation in verteilten Systemen (KiVS 2005). (2005) 91–102
17. Benatallah, B., et al.: Semantic reasoning for web services discovery. In: WWW2003 Workshop on E-Services and the Semantic Web. (2003)
18. http://www.globus.org/toolkit/.
19. Said, M.P., Kojima, I.: Towards Automatic Service Discovery and Monitoring in WS-Resource Framework. In: Proc. of the First International Conference on Semantics, Knowledge and Grid. (2005) 932–938
20. Raman, R., Livny, M., Solomon, M.: Matchmaking: Distributed resource management for high throughput computing. In: Proc. of the Seventh IEEE International Symposium on High Performance Distributed Computing. (1998) 140–146
21. Raman, R., Livny, M., Solomon, M.: Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. In: Proc. of the 12th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-12). (2003) 80–89
22. Liu, C., Foster, I.: A Constraint Language Approach to Matchmaking. In: Proc. of the 14th International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications (RIDE'04). (2004) 7–14
23. Tangmunarunkit, H., et al.: Ontology-based Resource Matching in the Grid—The Grid meets the Semantic Web. In: Proc. of SemPG03. Volume 2870. (2003) 706–721
24. http://www.naics.com/.
25. http://www.unspsc.org/.
26. http://jena.sourceforge.net/.