Improving Semantic Web Services Discovery Using SPARQL-Based Repository Filtering

José María García*, David Ruiz, Antonio Ruiz-Cortés

University of Seville, ETSI Informática, Av. Reina Mercedes, s/n, 41012 Sevilla, Spain

Abstract

Semantic Web Services discovery is commonly a heavyweight task, which has scalability issues when the number of services or the ontology complexity increase, because most approaches are based on Description Logics reasoning. As a higher number of services becomes available, there is a need for solutions that improve discovery performance. Our proposal tackles this scalability problem by adding a preprocessing stage based on two SPARQL queries that filter service repositories, discarding service descriptions that do not refer to any functionality or non-functional aspect requested by the user before the actual discovery takes place. This approach fairly reduces the search space for discovery mechanisms, consequently improving the overall performance of this task. Furthermore, this particular solution does not provide yet another discovery mechanism, but it is easily applicable to any of the existing ones, as our prototype evaluation shows. Moreover, proposed queries are automatically generated from service requests, transparently to the user. In order to validate our proposal, this article showcases an application to the OWL-S ontology, in addition to a comprehensive performance analysis that we carried out in order to test and compare the results obtained from proposed filters and current discovery approaches, discussing the benefits of our proposal.

Keywords: Semantic Web Services, Service Discovery, Scalability, Service Repositories, Semantic Web Query Languages

1. Introduction

Current Semantic Web Services (SWS) discovery solutions often suffer from scalability issues, so large and complex service repositories cannot be properly handled by them. Although the research community is putting effort into improving discovery mechanisms, the underlying reasoning facilities do not scale well in general [1]. The approach taken in this paper does not consist on yet another discovery mechanism, but on the inclusion of a preprocessing stage that filters service repositories using two different queries, so that the search space for discovery processes is reduced in our experiments, on average, from 12.5% of the original repository size up to 1.1%, depending on the concrete query used and the nature of the repository and user request. Consequently, service discovery execution time is greatly improved, performing the whole process, when using our proposed filters, at least 9.1 times faster and up to 44.7 times faster, with a contained penalty on precision, depending on each corresponding query and the underlying discovery mechanism chosen.

The number of currently available services in public repositories[1] is expected to explode in the future, so that billions of services will be able to be consumed in the Web [2]. Furthermore, currently available semantic descriptions, in terms of SWS classical ontologies such as OWL-S or WSMO, present a high complexity for defining and processing them. Both issues lead to a scenario where discovery mechanisms based on different logic formalisms have scalability issues. Consequently, current research efforts focus on providing improvements and optimizations of those mechanisms, using lightweight semantic technologies, in order to enhance the usability of SWS [3, 4].

In order to alleviate the scalability problem on semantic discovery mechanisms, there are some proposals that provide different techniques to improve the discovery performance, such as indexing or caching descriptions [5], using several matchmaking stages [6], and hybrid approaches that include non-semantic techniques [7]. Our proposal takes a novel approach of reducing the input for discovery mechanisms, so that the resulting process is more streamlined, only reasoning about services which actually matter with respect to the user request. Thus, our solution filters services that can be discarded *a priori*, because they are

*Corresponding author. Tel.: +34 9545 59814. Fax: +34 9545 57139

Email addresses: josemgarcia@us.es (José María García), druiz@us.es (David Ruiz), aruiz@us.es (Antonio Ruiz-Cortés)

URL: http://www.isa.us.es/josemaria.garcia (José María García)

[1]At the moment of writing, *seekda!* service crawler has indexed 28,606 services, *ProgrammableWeb* has registered 3,287 web APIs, and *iServe* repository contains 2,193 SWS descriptions.

not related at all with requirements and preferences stated by the user, considerably reducing the search space before actual discovery.

For example, consider the following scenario: a semantic service repository contains thousands of services from several travel-related domains, such as hotel bookings, plane tickets, car rentals, and travel insurances. If a user looks for a service that returns hotels given a particular city and a country, it is not necessary to process the whole repository to discover candidate services for the user request, but only consider the portion of services that are specifically related to the hotel lookup domain concepts that appear on the request, in this case. Thus, using lightweight technologies to preprocess the repository, the search space can be reduced in order to save computational resources and improve discovery performance.

For the proposed preprocessing, our proposal analyzes the user request in order to extract the concepts that are being used in its semantic definition (in the above example, some of them could be *City*, *Country* or *Hotel*, for instance). Then, the repository is filtered so that only services that use those concepts or related ones are selected to become the input for the subsequent discovery process (*e.g.* services whose definitions refer to *City*, *Country* and/or *Hotel* concepts, in the latter case).

Two different SPARQL [8] queries perform the filtering in our approach, namely $\mathcal{Q}_{all}$ and $\mathcal{Q}_{some}$. The former returns only those services whose definitions contain *all* the concepts referred by a user request, assuming that services have to fulfill every term of the request in order to be useful for the user. In turn, the latter query selects service definitions that refer to *some* (at least one) of the concepts referred by a user request, assuming that those services may satisfy its requirements and/or preferences to some extent, despite the missing information.

Our solution does not pretend to provide yet another discovery mechanism, but to introduce a preprocessing filtering stage, based on an accepted standard, that yields a notable improvement on heavyweight semantic processes, such as matchmaking of services. Furthermore, our proposed filtering does not add a noticeable amount of execution time with respect to matchmaking, because SPARQL queries used present a linear complexity on the size of the dataset and graph patterns included [9].

To the best of our knowledge, there are no proposals on filtering semantically-enhanced service repositories, but it is acknowledged that some sort of preprocessing can alleviate discovery and ranking tasks performed on those repositories [6]. To sum up, the main contributions of the proposal presented in this article are the following:

1. We propose a technique to improve semantic service discovery performance, based on a preprocessing stage that filters repositories in order to reduce the search space of subsequent discovery processes.
2. Our proposal is applicable to any discovery mechanism because it is performed before actual discovery occurs, and it allows interoperability with existing service repositories. In this work, we use OWLS-MX hybrid matchmaker [7] to illustrate this point, though our proposal has been also applied to other discovery mechanisms [10].
3. Filtering is performed automatically from user requests, analyzing them and obtaining standard SPARQL queries without user interaction. Two different queries are presented, enabling two filtering levels, depending on the user needs and the characteristics of service repositories. We analyze and thoroughly discuss each query throughout the article.
4. In order to assess the actual impact of our proposal, we carried out a comprehensive, experimental study. Using a widely-used test collection (OWLS-TC), we applied our proposed filters to several discovery mechanisms, evaluating and discussing performance improvements using the Semantic Web Service Matchmaker Evaluation Environment (SME^2).

The rest of the article is structured as follows. Firstly, Section 2 presents some background information to contextualize and motivate the proposal. In Section 3 we show how to use SPARQL-based filtering within a discovery scenario, presenting both restrictive and relaxed filters that can be applied in different cases. Section 4 discuss the integration and implementation of our proposal applied to SWS frameworks, specifically OWL-S. Then, in Section 5 the performed experimental study is explained, analyzing the results and discussing the advantages of our proposal. Section 6 outlines the related work on this field. Finally, in Section 7 we discuss the conclusions.

2. Background

Using a Semantic Web query language is a natural fit for performing SWS discovery and ranking processes in terms of user requests, because, essentially, these processes search for elements in some sort of persistent storage using selection and ordering criteria. However, current query languages present shortcomings with respect to the level of inference and computation needed for SWS discovery and ranking. In the following we introduce the background elements of our proposal in order to contextualize and further motivate our work.

2.1. Querying the Semantic Web

There are three main approaches for Semantic Web query languages: graph-based, rule-based, and DL-based query languages [11, 12, 13]. Firstly, graph-based query languages allow to fetch RDF [14] triples based on matching triple patterns with RDF graphs. Secondly, rule-based query languages propose logic rules to define queries, supporting RDF reasoning systems. Finally, DL-based query languages allow to query Description Logics (DL) ontologies described in OWL-DL [15], being able to search for concepts, properties, and individuals. In general, rule- and

DL-based query languages provide more reasoning mechanisms than graph-based ones, though it depends on the entailment regime applied to the concrete triple store and querying system. However, the former are not mature enough and they are in early stages of development [11], so the latter are more widely used, especially SPARQL [8], which is the current W3C Recommendation.

There are several graph-based query languages with different features [11], but SPARQL is the only language that is a W3C Recommendation [8]. In fact, it is fully supported in several implementations[2]. As a consequence, SPARQL (and its extensions) is the most widely used query language for the Semantic Web. There are several SPARQL implementations, such as Virtuoso, Sesame and ARQ,[3] which is included in the Jena Semantic Web Framework for Java. The latter is the chosen one for our evaluation tests presented in Section 5.

SPARQL, as a graph-based query language, explicitly accounts for the definition of labelled directed graphs by RDF triples, which conforms the very foundations of a Semantic Web ontology. Its main approach to query semantic repositories is to define graph patterns involving triple patterns, matching RDF triples, which are usually denoted (s, p, o), where s is the subject, p the predicate, and o the object. In order to work with said repositories, SPARQL has four different types of queries: `SELECT`, `CONSTRUCT`, `DESCRIBE` and `ASK`. Each type serves for a different purpose: `SELECT` queries return variables and their bindings with respect to the stored RDF triples; `CONSTRUCT` queries build an RDF graph based on a template defined in the query; `ASK` queries test whether a pattern has any solution or not; and `DESCRIBE` queries return a graph made up of triples relating to a nominated resource, in some preconfigured way, according to the querying system implementation.

Essentially, each SPARQL query is defined by a graph pattern expression, based on Turtle notation [16], that is matched against an RDF dataset in order to bind the variables used in triple patterns involved in that expression. Variables may substitute the subject, predicate, and / or object of any triple pattern, that ranges over matching RDF triples, binding variables accordingly. A basic graph pattern containing a set of triple patterns can be filtered using built-in conditions that restrict variable bindings, or combined with other patterns that may be matched optionally (not rejecting solutions if variables included in an optional graph pattern cannot be bound) or alternatively (one or more of several alternative graph patterns may match). Finally, a list of modifiers can be applied to SPARQL queries solutions, so that they can be returned ordered by a defined criteria, ensuring that solutions are unique, or limiting the number of them, for instance.

[2]`http://www.w3.org/2001/sw/DataAccess/tests/implementations`

[3]Virtuoso: `http://www.openlinksw.com/virtuoso/`; Sesame: `http://www.openrdf.org/`; ARQ: `http://jena.sourceforge.net/ARQ/`

Current SPARQL recommendation offers a very simple entailment that does not support proper reasoning. For instance, if a graph pattern looks for an RDF resource of a given class A, it does not match with resources whose classes are subclasses of A, though intuitively they should match. To solve this issue, SPARQL querying systems implement various entailment regimes, some of them being formalized by W3C for the next version of SPARQL recommendation[4]. However, if simple entailment, as defined in [8], is the only available regime in the SPARQL implementation being used, two approaches can be taken: (1) the implicit knowledge, such as subclassing, can be made explicit by adding corresponding RDF triples to the dataset prior to SPARQL query execution; or (2) subclasses can be made explicit directly in the graph patterns of the query. The latter solution is chosen for our proposal evaluation as discussed in Section 4.3.

Currently, the SPARQL recommendation is being revised to apply some other extensions already identified, such as insert/update/delete queries, access to collection members, or aggregate functions (`COUNT`, `SUM`, `GROUP BY`, etc). Furthermore, different authors propose extensions to further improve reasoning features [17], expressiveness of queries [18], or even approaches that add DL-based languages features [13]. However, in this work we stick to SPARQL 1.0 (the recommendation at the time of writing) to improve discovery processes, though some of the extensions discussed can be also applied (see Section 6).

2.2. Semantic Web Services

SWS are often defined using specific semantic frameworks, which add extensions to non-semantic service descriptions as in SAWSDL [19], or provide ontologies, such as OWL-S [20], WSMO [21], or WSMO-Lite [22], that serve as the foundations for tools to discover and rank services in terms of user requests described using their provided facilities. Furthermore, these ontologies can be extended to improve those tasks using other ontologies [23, 24, 25], so that service descriptions may include information about quality-of-service and preferences, for instance.

Essentially, a service description, whether it is defined using OWL-S, WSMO, SAWSDL, or WSMO-Lite, is composed of several statements or terms that define service features, which can describe functionality (such as input and output parameters) or non-functional aspects. These terms refer to several related domain concepts. Similarly, user requests are composed of a number of terms that describe the requirements the requested service has to meet. Each requirement is also related to one or more particular concepts.

For instance, OWL-S service descriptions feature the service functionality within *service profiles*, where different types of terms describe inputs, outputs, preconditions

[4]`http://www.w3.org/TR/sparql11-entailment/`

Listing 1: OWL-S service profile example.

```
@prefix profile:
   <http://www.daml.org/services/owl-s/1.1/
    Profile.owl#>.
@prefix process:
   <http://www.daml.org/services/owl-s/1.1/
    Process.owl#>.
@prefix portal:
   <http://purl.org/iserve/ontology/owlstc/
    portal.owl#>.
@prefix travel:
   <http://purl.org/iserve/ontology/owlstc/
    travel.owl#>.

:CityCountryHotelProfile a profile:Profile;
    profile:hasInput   :CityInput;
    profile:hasInput   :CountryInput;
    profile:hasOutput :HotelOutput.

:CityInput a process:Input;
    process:parameterType portal:City.
:CountryInput a process:Input;
    process:parameterType portal:Country.
:HotelOutput a process:Output;
    process:parameterType travel:Hotel.
```

and results, correspondingly. In turn, WSMO service descriptions are defined by a *capability* and *interfaces*, that contains a number of terms describing preconditions, assumptions, postconditions, effects; and inputs, outputs and transition rules, respectively.

In order to develop an abstract and interoperable solution, decoupled from concrete SWS ontologies, our filters are defined in Section 3 using an abstract vocabulary of terms and their referred domain concepts. Concrete applications to existing SWS frameworks can be consequently developed by identifying correspondences between this vocabulary and the corresponding SWS ontology so that proposed filters can be implemented using SPARQL queries over SWS ontologies (see Section 4).

Consequently, we assume that service descriptions and user requests are defined in terms of concepts from some domain ontologies, which depend on the concrete scenario. As an example, consider the scenario described in Section 1, where a repository contains several services related to travel domains. Service descriptions may feature several statements or *terms* defining their provided functionality and non-functional properties using *concepts* from travel domain ontologies. Thus, a hotel lookup service description (showcased as an OWL-S profile in Listing 1, using Turtle notation) will contain terms that refer to concepts like *City*, *Country* or *Hotel*, for instance.

2.3. Discovering and Ranking

The common use case for discovery and ranking of SWS is depicted in Figure 1: Starting from a service repository ($\mathcal{S}$) containing definitions either using OWL-S, WSMO, SAWSDL, or WSMO-Lite, for instance, that conforms the search space, the *discovery* process searches for these available service definitions, which are described in terms of domain ontologies ($\mathcal{O}$), that match with a user request ($\mathcal{U}$).

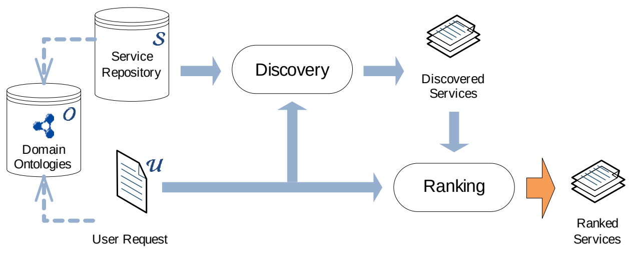


Figure 1: Semantic discovery and ranking processes.

This matchmaking is usually performed using logic reasoning techniques, such as DL reasoners [26, 27, 28], logic programming [25, 29], or hybrid approaches [7, 30, 31]. The resulting discovered services are a subset of the initial repository, where each instance of this subset is considered to be compliant with the user request, to some extent.

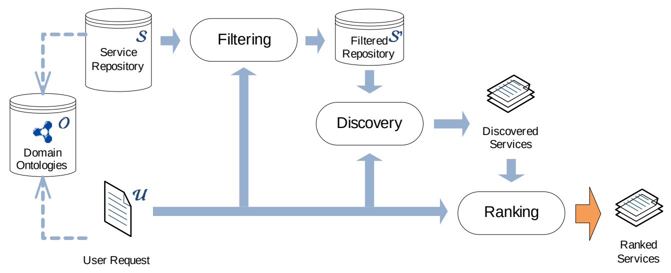
Concerning user requests for SWS discovery and ranking, there are several approaches on how to define them. Thus, in standard WSMO they are described as goals, where the functionality requested by a user is defined by means of capabilities and interfaces. They can be used to match corresponding services in the discovery stage taking into account preconditions, effects, inputs, and outputs, among other description elements pertaining to capabilities and interfaces.

Furthermore, some authors extend WSMO goals to refine non-functional properties descriptions, so that they can be used to rank previously discovered services [25, 31]. Therefore, using both discovered services and preferences described in the user request [23], the *ranking* process returns an ordered list of those services in terms of stated preferences. Altough user requests used in our evaluation only contains information about inputs and outputs, more complex user requests can also take benefit of our proposal [10].

SWS discovery techniques particularly suffer from performance and scalability issues in this context. The underlying logic formalisms are not sufficiently scalable for the current Web [1], so there is a need for lightweight approaches to SWS discovery or optimizations over currently available solutions. The main motivation of this work is to come out with a solution that effectively improves discovery and ranking, turning them into more lightweight processes, while making the most of currently available matchmakers.

3. Preprocessing Service Repositories using SPARQL

As discussed before, current SWS discovery and ranking tend to be complex, heavyweight processes. In the following we present our abstract filtering proposal and how it can be implemented using standard, automatically generated SPARQL 1.0 queries.



$\mathcal{S}$ Service Repository
Filtering
$\mathcal{S}'$ Filtered Repository
$\mathcal{O}$ Domain Ontologies
Discovery
Discovered Services
$\mathcal{U}$ User Request
Ranking
Ranked Services

Figure 2: Service procurement architecture including a SPARQL filtering stage.

3.1. Filtering a Service Repository

Figure 2 showcases our proposed architecture as an alternative to the one described in Figure 1. Our solution adds a new preprocessing stage, previous to the discovery process, that filters the service repository, using SPARQL queries as described in Section 3.2. The aim of the filtering stage is to obtain $\mathcal{S}'$ services from the original repository $\mathcal{S}$ that may be possibly matched with the user request $\mathcal{U}$ in the discovery process, discarding those ones that cannot fulfill that request at all.

In a general scenario, our proposed filtering stage discriminates service descriptions depending on whether concepts referenced within their terms are present in the user request or not. To this extent, two different filters can be applied, offering different filtering levels. On the one hand, one of the filters ($\mathcal{Q}_{all}$) only returns service descriptions that refer to the whole set of related concepts described in the user request. On the other hand, a more relaxed filter ($\mathcal{Q}_{some}$) returns those service descriptions that refer to some (at least one) of the concepts that are also referred by the user request. In turn, both filters discard services whose terms do not refer to any of the related concepts referred in the requirements of the user request, because in that case it can be inferred that they are not related to the service the user is searching for.

Considering that a service description is defined using a series of terms that refer to several domain concepts, we can describe our generic proposal as follows. Let $\mathcal{D} = (\mathcal{O}, \mathcal{S}, \mathcal{U})$ be a 3-tuple that represent a discovery scenario as outlined in Figure 2, where each element of the tuple is defined in the following.

Domain ontologies ($\mathcal{O}$). Let $\mathcal{O}_i$ be a certain domain ontology whose concepts can be referred by the user request and service descriptions from a certain discovery scenario $\mathcal{D}$. The set of *domain ontologies* $\mathcal{O}$ is defined as the set of ontologies that can be used to define the rest of the elements from that scenario, *i.e.* the user request and service descriptions.

$$\mathcal{O} = \mathcal{O}_1 \cup \cdots \cup \mathcal{O}_n$$

Service repository ($\mathcal{S}$). Let $\mathcal{O}_{\mathcal{S}_i}$ be a subset of $\mathcal{O}$. A *service repository* $\mathcal{S}$ is a set of service descriptions $\mathcal{S}_i$ that are defined by several terms t_{ij}. Each term refer to a set of concepts $\mathcal{C}_{ij}$ defined in the ontology $\mathcal{O}_{\mathcal{S}_i}$. Therefore, each $\mathcal{S}_i$ is represented as a set of tuples that relate terms with their corresponding set of referred concepts:

$$\mathcal{S}_i = \{(t_{i1}, \mathcal{C}_{i1}), \ldots, (t_{in}, \mathcal{C}_{in}) \colon \mathcal{C}_{i1} \cup \cdots \cup \mathcal{C}_{in} \subseteq \mathcal{O}_{\mathcal{S}_i}\}$$

User request ($\mathcal{U}$). Similarly, a *user request* $\mathcal{U}$ contains requirements in the form of terms that refer to some subset of concepts from a domain ontology $\mathcal{O}_{\mathcal{U}} \subseteq \mathcal{O}$:

$$\mathcal{U} = \{(t_1, \mathcal{C}_1), \ldots, (t_n, \mathcal{C}_n) \colon \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n \subseteq \mathcal{O}_{\mathcal{U}}\}$$

In order to better illustrate previous definitions, consider an scenario where a user is searching for a hotel lookup service like the described in Listing 1. The corresponding user request $\mathcal{U}$ is defined as follows:

$$\begin{aligned}\mathcal{U} = \{&(inputTerm_{u1}, \{\text{portal:City}\}),\\ &(inputTerm_{u2}, \{\text{portal:Country}\}),\\ &(outputTerm_{u1}, \{\text{travel:Hotel}\})\}\end{aligned}$$

This user is going to search for services described in a repository $\mathcal{S}$ that contains three services related to travel domains, such that:

$$\begin{aligned}\mathcal{S}_1 = \{&(inputTerm_{11}, \{\text{portal:City}\}),\\ &(outputTerm_{11}, \{\text{travel:LuxuryHotel}\})\}\\ \mathcal{S}_2 = \{&(inputTerm_{21}, \{\text{portal:City}\}),\\ &(inputTerm_{22}, \{\text{portal:Country}\}),\\ &(outputTerm_{21}, \{\text{travel:Hotel}\})\}\\ \mathcal{S}_3 = \{&(inputTerm_{31}, \{\text{travel:Surfing}\}),\\ &(outputTerm_{31}, \{\text{travel:Beach}\})\}\end{aligned}$$

Finally, the global domain ontology in this example could be simply considered as the set of concepts involved in previous descriptions: $\mathcal{O} = \{\text{portal:City}, \text{portal:Country}, \text{travel:LuxuryHotel}, \text{travel:Beach}, \text{travel:Hotel}, \text{travel:Surfing}\}$.

Once the elements that conform the discovery scenario $\mathcal{D} = (\mathcal{O}, \mathcal{S}, \mathcal{U})$ are properly defined, the two previously introduced filters can be used alternatively to obtain an $\mathcal{S}' \subseteq \mathcal{S}$ so that the subsequent discovery process defined by $\mathcal{D}' = (\mathcal{O}, \mathcal{S}', \mathcal{U})$ performs better.

In order to simplify both filters definitions, we denote with $\mathcal{C}_{\mathcal{S}_i}$ the subset of concepts from $\mathcal{O}_{\mathcal{S}_i}$ that are actually referred in the terms featured in $\mathcal{S}_i$. Equivalently, $\mathcal{C}_{\mathcal{U}}$ is the subset of referred concepts in $\mathcal{U}$.

$$\begin{aligned}\mathcal{C}_{\mathcal{S}_i} &= \{c \in \mathcal{O}_{\mathcal{S}_i} \colon \exists (t_{ij}, \mathcal{C}_{ij}) \in \mathcal{S}_i | c \in \mathcal{C}_{ij}\}\\ \mathcal{C}_{\mathcal{U}} &= \{c \in \mathcal{O}_{\mathcal{U}} \colon \exists (t_j, \mathcal{C}_j) \in \mathcal{U} | c \in \mathcal{C}_j\}\end{aligned}$$

Consequently, in the example described before, the corresponding concepts subsets of $\mathcal{O}$ for the service descriptions in $\mathcal{S}$ and the user request $\mathcal{U}$ are the following:

$$\begin{aligned}
\mathcal{C}_{\mathcal{S}_1} &= \{\text{portal:City}, \text{travel:LuxuryHotel}\}\\
\mathcal{C}_{\mathcal{S}_2} &= \{\text{portal:City}, \text{portal:Country}, \text{travel:Hotel}\}\\
\mathcal{C}_{\mathcal{S}_3} &= \{\text{travel:Surfing}, \text{travel:Beach}\}\\
\mathcal{C}_{\mathcal{U}} &= \{\text{portal:City}, \text{portal:Country}, \text{travel:Hotel}\}
\end{aligned}$$

The application of both filters to a service repository $\mathcal{S}$ return a subset $\mathcal{S}'$ depending on the corresponding filter applied. In the case that $\mathcal{S}' = \mathcal{Q}_{all}(\mathcal{S},\mathcal{U})$, the application of the filter returns a subset of $\mathcal{S}$ only containing services whose referred concepts are a superset of those referred by a user request $\mathcal{U}$, *i.e.* all concepts referred by the user request are referred by returned service descriptions.

$$\mathcal{Q}_{all}(\mathcal{S},\mathcal{U}) = \{\mathcal{S}_i \in \mathcal{S} \colon \mathcal{C}_{\mathcal{U}} \subseteq \mathcal{C}_{\mathcal{S}_i}\}$$

In turn, if we identify $\mathcal{S}' = \mathcal{Q}_{some}(\mathcal{S},\mathcal{U})$, the filter selects those services from $\mathcal{S}$ that share at least one referred concept with the user request $\mathcal{U}$, so the intersection of corresponding referred concepts sets cannot be empty.

$$\mathcal{Q}_{some}(\mathcal{S},\mathcal{U}) = \{\mathcal{S}_i \in \mathcal{S} \colon \mathcal{C}_{\mathcal{U}} \cap \mathcal{C}_{\mathcal{S}_i} \neq \emptyset\}$$

Results of applying both filters to the described example are, in the first proposed filter case: $\mathcal{Q}_{all}(\mathcal{S},\mathcal{U}) = \{\mathcal{S}_2\}$, and in the second case: $\mathcal{Q}_{some}(\mathcal{S},\mathcal{U}) = \{\mathcal{S}_1, \mathcal{S}_2\}$.

Although $\mathcal{Q}_{all}$ effectively reduces the discovery search space ($\mathcal{Q}_{all}(\mathcal{S},\mathcal{U}) \subseteq \mathcal{S}$) and, consequently, processing time, it may excessively restrict the candidate services to be considered for the subsequent discovery process, whose resultant precision and/or recall may be affected, as we corroborate in our experiments in Section 5. Thus, the proposed $\mathcal{Q}_{some}$ filter relaxes the former one by considering each concept referenced in the user request as a matching alternative within the set of concepts referred by service description terms. In this case, service descriptions that do not refer to any concept used in the user request are discarded for the following discovery stage. In general, $\mathcal{Q}_{all}(\mathcal{S},\mathcal{U}) \subseteq \mathcal{Q}_{some}(\mathcal{S},\mathcal{U}) \subseteq \mathcal{S}$, so filtering repositories using $\mathcal{Q}_{some}$, the amount of services that are considered for discovery (and ranking) is reduced less than in the $\mathcal{Q}_{all}$ scenario. However, the overall performance improvement is also high, while it slightly affects the process precision/recall relation, as analyzed in Section 5.

3.2. A SPARQL Implementation for Filters

The abstract description of our proposed filters $\mathcal{Q}_{all}$ and $\mathcal{Q}_{some}$ introduced previously can be implemented in any existing SWS discovery scenario using SPARQL SELECT queries. Given a concrete user request defined using an existing SWS framework, both filters can be instantiated as SPARQL queries that select corresponding services from an RDF-based repository, which contains descriptions based on the same SWS framework. In this case, generated queries have to be also based on graph patterns ranging over that SWS framework RDF representation. Nevertheless, to better account for interoperability some proposals that integrate SWS framework definitions [32, 33, 34] can also apply our proposed filters (see Section 6).

Queries need to be instantiated for each user request $\mathcal{U}$, because they depend on the structure of that request. In order to compose $\mathcal{Q}_{all}$ and $\mathcal{Q}_{some}$ filters, the implementation has to analyze which concrete concepts referred by the user request are going to be included in the corresponding SPARQL query. Specifically, query generation depends not only on the structure of the ontology our proposal is being applied to, but also on the concrete instance $\mathcal{U}$ of the user request itself, especially on the concepts referred by its terms ($\mathcal{C}_{\mathcal{U}}$). As a consequence, queries have to be tailored depending on the corresponding instances managed by each discovery process. However, the generation of $\mathcal{Q}_{all}$ and $\mathcal{Q}_{some}$ SPARQL queries can be done automatically, maintaining the transparency for the user of our proposed filtering stage within the discovery process.

On the one hand, $\mathcal{Q}_{all}$ filter is implemented as a query that searches for services whose featured terms refer to every concept referred in the user request. Thus, for each term and its corresponding concepts, $\mathcal{Q}_{all}$ query contains a triple pattern that matches service definition triples that contains those concepts, depending on the structure of the underlying SWS ontology. On the other hand, $\mathcal{Q}_{some}$ query is generated similarly, but each triple pattern matching a user request referred concept is grouped with the rest as alternative patterns, *i.e.* using the UNION keyword, because $\mathcal{Q}_{some}$ searches for services whose terms refer to at least one concept referred by the user request. The following section presents an application of both queries to OWL-S.

4. Application to Existing SWS frameworks

Our proposed preprocessing stage can be easily adapted to any SWS framework, such as WSMO, OWL-S, SAWSDL or WSMO-Lite, so that it can be virtually included within any discovery process. Application to these frameworks can be performed by identifying correspondences between elements from the filter definition discussed in Section 3.1 and the facilities that each framework provides to describe user requests, service terms and their referred concepts. Therefore, the SPARQL implementation of both filters contains triple patterns, using the target SWS framework ontology, that refer to services ($\mathcal{S}$), requests ($\mathcal{U}$), terms and domain concepts ($\mathcal{C}_{\mathcal{S}_i}$ and $\mathcal{C}_{\mathcal{U}}$). In the following, we present a concrete OWL-S implementation of filters, but another early implementation to WSMO services can be found in [10], further proving our proposal applicability.

Listing 2: $\mathcal{Q}_{all}$ SPARQL query applied to OWL-S.

```
SELECT DISTINCT ?service
WHERE {
  ?service a service:Service;
           service:presents ?profile.
  # ?profile has at least two inputs and an output...
  ?profile profile:hasInput ?inputTerm1.
  ?profile profile:hasInput ?inputTerm2.
  ?profile profile:hasOutput ?outputTerm1.
  # ...and referred input concepts are City...
  {?inputTerm1 process:parameterType portal:City}
  # ...Country...
  {?inputTerm2 process:parameterType portal:Country}
  # ...and the output concept is Hotel
  {?outputTerm1 process:parameterType travel:Hotel}
}
```

4.1. An OWL-S Implementation

In order to implement an application of our proposed filtering stage that relies on OWL-S descriptions, they have to be published in a triple store and queries have to be defined in terms of OWL-S constructs. Basically, both service descriptions ($\mathcal{S}$) and user requests ($\mathcal{U}$) are modeled as `Service Profiles`. A service profile may contain several terms that further define features of an OWL-S service functionality, such as `Inputs`, `Outputs`, `Preconditions`, and `Results`. Already presented Listing 1 shows an OWL-S service profile RDF description that is used as an example user request in the following.

That service profile example has been taken from the OWLS-TC test collection used to evaluate our proposal in Section 5. In this collection, service descriptions merely contain information about inputs and outputs, and their parameter types, but no preconditions and results. Although both inputs and outputs can be related to the corresponding service profile by using the abstract `hasParameter` OWL-S property, in OWLS-TC profiles are explicitly related to inputs and outputs with `hasInput` and `hasOutput` properties. In consequence, our filters are refined to take into account the stated difference between inputs and outputs terms in OWL-S descriptions, so that they can obtain more accurate results.

Listings 2 and 3 presents our proposed $\mathcal{Q}_{all}$ and $\mathcal{Q}_{some}$ filter queries, respectively[5]. The identified correspondences between the elements of our abstract filtering proposal and OWL-S constructs are introduced for both SPARQL queries, as described in Section 3.2, so that they can be directly used to filter an OWL-S repository. In this example, both queries have been generated from the sample user request $\mathcal{U}$ defined in Section 3.1, whose referred concepts to be matched against service descriptions are $\mathcal{C}_{\mathcal{U}} = \{\text{portal:City}, \text{portal:Country}, \text{travel:Hotel}\}$.

Note that the presented OWL-S application refines the filters proposed in Section 3, taking into account that each type of term in $\mathcal{U}$ should be matched with the corresponding terms from service descriptions $\mathcal{S}_i$. In consequence, $\mathcal{C}_{\mathcal{U}}$ and $\mathcal{C}_{\mathcal{S}_i}$ sets of concepts are split in two subsets each, depending on the type of term (input or output), and compared with the corresponding one to obtain both filters results.

Listing 3: $\mathcal{Q}_{some}$ SPARQL query applied to OWL-S.

```
 1 SELECT DISTINCT ?service
 2 WHERE {
 3   ?service a service:Service;
 4            service:presents ?profile.
 5   # match all inputs and outputs of the profile...
 6   ?profile profile:hasInput ?inputTerms.
 7   ?profile profile:hasOutput ?outputTerms.
 8   # ...that refer to some concepts of user request
 9   {?inputTerms process:parameterType portal:City}
10   UNION
11   {?inputTerms process:parameterType portal:Country}
12   UNION
13   {?outputTerms process:parameterType travel:Hotel}
14 }
```

In principle, if RDF(S) entailment regime were applied to the RDF dataset of the service repository, making the inferred knowledge explicit, $\mathcal{Q}_{some}$ could have been written using a more concise and general approach that does not need to process the user request instance in order to explicitly reflect its referred concepts. Thus, lines 9 to 13 in Listing 3 could be substituted by the following excerpt, with :reqProfile being the concrete `ServiceProfile` instance that is used to look for requested services. However, if we have to account for inference as considered in Section 4.3 because a basic entailment is the only available in our querying system, then $\mathcal{Q}_{some}$ as defined in Listing 3 is more convenient.

```
?inputTerms process:parameterType ?inputConcepts.
?outputTerms process:parameterType ?outputConcepts.
:reqProfile rdf:type service:Service.
:reqProfile profile:hasInput ?reqInputTerms.
:reqProfile profile:hasOutput ?reqOutputTerms.
?reqInputTerms process:parameterType ?inputConcepts.
?reqOutputTerms process:parameterType ?outputConcepts.
```

4.2. Automatic Generation of Filter Queries

Right before the filtering is executed, corresponding SPARQL queries have to be generated using OWL-S user requests. Consequently, generation algorithms need to be applied to the OWL-S ontology, as discussed in Section 3.2. Essentially, the user request $\mathcal{U}$ (defined as a service profile as in Listing 1) has to be analyzed to obtain the concepts that are referred by each description term ($\mathcal{C}_{\mathcal{U}}$). The automatic generation of queries can also differentiate terms in order to get better results with basic entailment regimes.

For the evaluation discussed in Section 5, our filtering queries generated from OWLS-TC user requests only take inputs and outputs into account, though service profiles may contain more information terms that could be also analyzed to obtain more referred concepts from the corresponding domain ontology [10]. Therefore, for each OWLS-TC user request, its service profile is traversed

[5]Prefixes are omitted for the sake of clarity, but they correspond to those shown in Listing 1, in addition to `service` that refers to `http://www.daml.org/services/owl-s/1.1/Service.owl`

identifying each input and output, and adding a triple pattern to the corresponding query to match services with the same referred parameter types.

4.3. Dealing with SPARQL Entailment

If the RDF dataset does not contain subclassing knowledge as explicit triples, there are two different approaches to deal with the SPARQL basic entailment regime issues as described in Section 2.1. On the one hand, the implicit knowledge concerning subclasses can be retrieved using a DL reasoner [35, 36], so that corresponding RDF triples can be added to the RDF dataset, providing RDFS entailment. As this inferencing process is time-consuming, it may be executed periodically on the whole repository to properly update the dataset, in order to minimize its impact on query execution. However, this approach does not account for the fact that, at the moment a query is executed, the RDF dataset may not contain all the corresponding inferred triples.

On the other hand, queries can be rewritten, explicitly including subclasses of the concepts referenced in user requests. Thus, a DL reasoner is executed when generating SPARQL queries for both $\mathcal{Q}_{all}$ and $\mathcal{Q}_{some}$ filters to obtain the related subclasses for each concept referred in the user request. As a consequence, service descriptions whose referred concepts are subclasses of user request concepts can also be returned by our filtering stage, improving the accuracy of the results.

For instance, the chosen reasoner (Pellet [36] in our experiments) may infer that `LuxuryHotel` instances are also `Hotel` instances, because there is a subclass relationship between these classes. Then both of them can be considered as valid alternatives for a referred concept in a service description, if the user is looking for a service that features a `Hotel` concept as its input. Thus, an additional pattern alternative where `?inputTerms` refers to a `LuxuryHotel` concept have to be included in line 13 of Listing 3. Similarly, $\mathcal{Q}_{all}$ queries can also be modified to take concept subclasses into account. In this case, line 14 of Listing 2 have to be modified to the same patterns used in the $\mathcal{Q}_{some}$ case for `Hotel` concept, *i.e.*:

```
{?outputTerms process:parameterType travel:Hotel}
UNION
{?outputTerms process:parameterType travel:LuxuryHotel}
```

5. Analysis and Evaluation

Our proposed filters have to be thoroughly analyzed, using experimental results, in order to corroborate their soundness and expected benefits. Each filter has been tested in different situations, measuring several indicators to determine the actual improvements of our proposed preprocessing stage. In this section we describe the performed experimental evaluation, along with an interpretation and discussion of the results for that experimental study, which validates our proposal.

5.1. Experimental Scenario

In order to experimentally test the suitability and performance of our proposal, a proper test collection has to be used. There are some publicly available collections to evaluate service discovery algorithms for OWL-S and SAWSDL services. Particularly, we evaluate our proposal with respect to the OWL-S Services Retrieval Test Collection (OWLS-TC v3[6]). This collection contains 1007 OWL-S service descriptions from different domains, in addition to 29 user requests (referred as *queries*) and their corresponding sets of relevant services, so that, for each OWL-S query, the performance and effectiveness of matchmakers can be evaluated by checking whether returned services are relevant to the corresponding query or not.

In our experimental prototype, SPARQL query execution was implemented in Java using the Jena Semantic Web Framework. Therefore, our implementation reads OWL-S service descriptions from OWLS-TC, which are parsed and processed by Jena, enabling the execution of SPARQL queries over them. Then, the results from the query execution are used to filter the list of services that take part in the subsequent discovery process, improving its performance.

Nevertheless, our proposal cannot be evaluated on its own, because it does not perform service discovery, but includes a preprocessing stage to filter repositories before service matchmaking. Thus, in order to evaluate the actual impact of proposed filters using OWLS-TC, they have to be tested on top of an OWL-S service matchmaker, so that the differences between using filters or directly performing the discovery process can be analyzed.

The actual evaluation of our prefiltering proposal has been done using the Semantic Web Service Matchmaker Evaluation Environment (SME^2 v2.1[7]). SME^2 is an open source tool that can be used to test and compare several SWS matchmakers using the same test collection (OWLS-TC v3 in our case) as the input for each matchmaker. The variables measured by SME^2 that we use to compare matchmakers are the following:

- **Precision.** The proportion of returned services that are actually relevant for the corresponding query. The more precision a query execution presents, the more accurate the answer is.
- **Recall.** The proportion of the relevance set that is returned by a query. The more recall a query answer has, the more relevant services are returned by the corresponding query.
- **Fallout.** The proportion of non-relevant services retrieved by a query. In other words, it measures the amount of false positives returned by the corresponding query with respect to the complete answer set.

[6]http://projects.semwebcentral.org/projects/owls-tc/
[7]http://projects.semwebcentral.org/projects/sme2/

Table 1: Average query response times and precision.

Matchmaker	Filter	Avg query response time		Avg query precision	
OWLS-M0	$\mathcal{Q}_{all}$	1283	(±57) ms	31.62	(±6.20) %
	$\mathcal{Q}_{some}$	6333	(±3023) ms	68.13	(±7.49) %
	None	57332	(±1592) ms	49.55	(±6.70) %
OWLS-MX3 (M3)	$\mathcal{Q}_{all}$	1321	(±61) ms	31.45	(±6.15) %
	$\mathcal{Q}_{some}$	5500	(±2810) ms	72.02	(±6.28) %
	None	58456	(±214) ms	82.96	(±4.50) %

- **Query response time.** For each query, it measures the time a concrete matchmaker spends on evaluating that query and returning the corresponding results, without the initialization time needed for registering service descriptions.
- **Memory usage.** Measured samples of the amount of memory a matchmaker uses during its whole execution time.

Precision, recall and fallout are standard, well-known measures for evaluating information retrieval techniques [37]. Particularly, SME^2 computes precision and fallout using a macro-averaged approach that sums up the results from all query executions. Thus, for each query, SME^2 measures precision and fallout at equidistant standard recall values, and then it obtains the mean value for these measures at each recall level. Nevertheless, SME^2 also computes the well-known *average precision* measure for each single query, enabling performance evaluation regardless of the number of services returned by the matchmaker. Section 5.2 discusses the mean average precision, along with the others measures.

Our prototype implements the `IMatchmakerPlugin` interface so that it can be plugged into SME^2. However, it has to be associated with another matchmaker that is called using the same interface to actually perform SWS discovery after prefiltering the input. For evaluation purposes we have chosen some variants of OWLS-MX, which is a hybrid SWS matchmaker that combines both logic-based approaches and information retrieval techniques for a high performance discovery [7]. Each chosen variant is firstly executed as is, and then with $\mathcal{Q}_{all}$ and $\mathcal{Q}_{some}$ filters on top of it. Thus, the different combinations of a OWLS-MX variant and (possibly) a corresponding filter are compared against each other in order to evaluate the performance of our proposal in different situations.

For the sake of brevity, in the following we only compare the performance results of two different OWLS-MX variants, namely *OWLS-M0* and *OWLS-MX3 (M3)*, because the other variants present similar results to the latter. *OWLS-M0* is a simple, logic-based matchmaker that only uses reasoning techniques, while *OWLS-MX3 (M3)* adds text similarity matchings to avoid false positives and improve the precision of the results. Evaluation results of the rest of the variants are available upon request to the authors.

5.2. Analyzing Tests Results

Firstly, we analyze the performance improvement obtained by using our proposed filters before service discovery. Table 1 presents a summary of the evaluation performed where both OWLS-M0 and OWLS-MX3 (M3) variants are compared in terms of their average execution time and mean average precision for all OWL-S queries of the test collection, along with confidence intervals calculated using a confidence level of 95%. Most query response times are highly improved when using any of the filters, though $\mathcal{Q}_{some}$ filter impact is lower because it returns more results as shown in Figure 3. Noteworthy, actual filtering time does not affect the overall OWL-S query response time, because our proposed SPARQL queries can be executed in polynomial time by SPARQL implementations [9].

Experimental results show that, on average, response time of OWLS-M0 is 44.7 times faster if applying $\mathcal{Q}_{all}$ filter, and about 9 times faster if $\mathcal{Q}_{some}$ filter is the applied one. OWLS-MX3 (M3) performance is similarly improved (44.3 times faster with $\mathcal{Q}_{all}$ and 10.6 times faster with $\mathcal{Q}_{some}$). Even though the confidence interval in $\mathcal{Q}_{some}$ cases is large, in the worst case scenario, the execution is at least 6.1 times faster when using OWLS-M0 matchmaker, and 7 times faster for OWLS-MX3 (M3).

Despite its high time performance, $\mathcal{Q}_{all}$ filtering shows worse performance in terms of average precision than the rest of the evaluated alternatives, providing an average value of about 31%. In turn, $\mathcal{Q}_{some}$ shows a better average precision on all the evaluation tests than $\mathcal{Q}_{all}$. Thus, for logic-based OWLS-M0 variant, $\mathcal{Q}_{some}$ filtering presents an improvement of about 19% on precision with respect to the execution of OWLS-M0 with no preprocessing. For OWLS-MX3 hybrid variant, average precision only drops by 11%, though response time is considerably faster. Note that average precision measures have a strong dependency on the concrete query and services registered in the repository.

Response time improvements are correlated to the degree of filtering each filter is able to provide. Figure 3 presents a logarithmically-scaled box plot that analyses the proportion of services returned for the 29 queries from OWLS-TC with respect to the initial repository of 1007

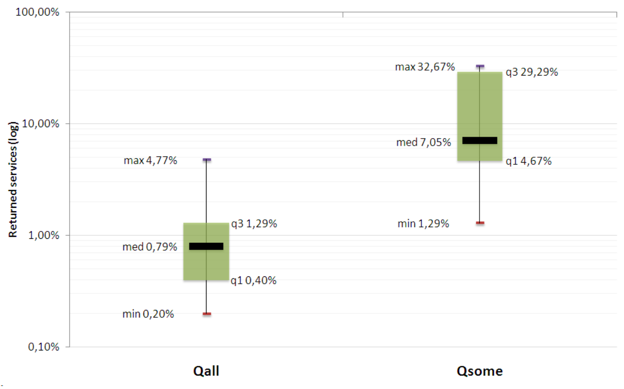


Figure 3: Returned results with respect to the original repository size.

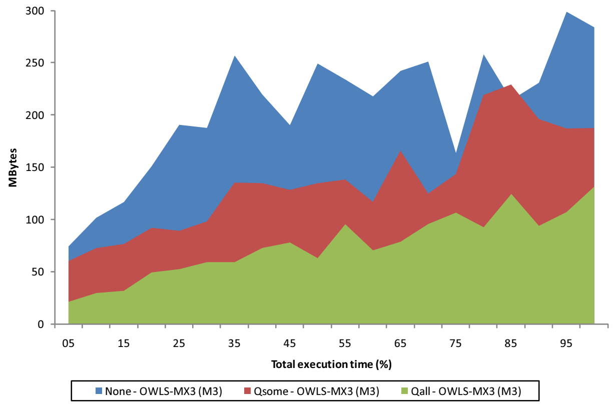


Figure 4: Memory consumption statistics when filtering OWLS-MX3 (M3).

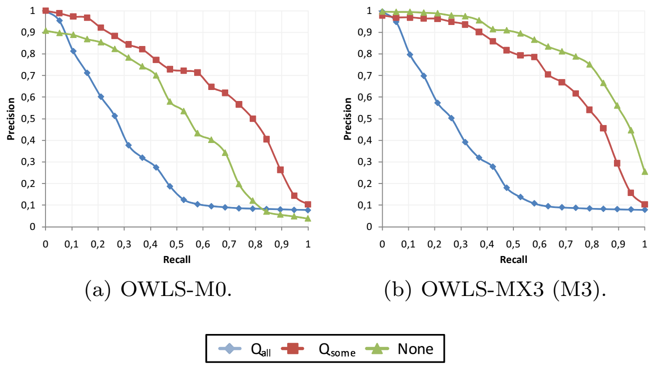


Figure 5: Recall-Precision effect when filtering OWLS-MX variants.

services. In general, $\mathcal{Q}_{all}$ filter returns a very low number of services (most queries returning between 0.4 and 1.29% of the original repository), greatly improving query response time as discussed before. On the other hand, $\mathcal{Q}_{some}$ filter results vary between a bigger range, with a median value of 7.05 % of the original repository, so the corresponding query response time for each matchmaker is slightly slower when using $\mathcal{Q}_{some}$ filter than when using $\mathcal{Q}_{all}$. In particular, some OWLS-TC queries present a lower filtering degree when using $\mathcal{Q}_{some}$, causing a noticeable variation on the response time that explains the larger $\mathcal{Q}_{some}$ confidence interval shown in Table 1. Additionally, the discovery process presents less initialization time because the number of services to be loaded by matchmakers is significantly low, especially when $\mathcal{Q}_{all}$ filter is applied.

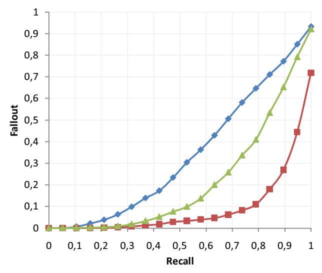
Furthermore, Figure 4 presents the performance gain in terms of memory consumption, only showcasing samples from the execution of OWLS-MX3 (M3) variant for the sake of clarity. Results show that filtering the repository leads to a lower memory usage, because the matchmaker needs to access less resources. On average, OWLS-MX3 (M3) needs 1.5 times less memory if $\mathcal{Q}_{some}$ filter is applied, and 2.8 times less if filtering with $\mathcal{Q}_{all}$. In conclusion, the use of our proposed filters substantially improves the overall performance of OWLS-MX matchmaker hybrid variants, both in terms of response time and memory consumption, though the impact on precision, recall and fallout has to be evaluated.

In order to analyze the penalty on precision and recall, Figure 5 compares the macro-averaged precision of the two discussed OWLS-MX variants when different filters are applied (*i.e.* using $\mathcal{Q}_{all}$, $\mathcal{Q}_{some}$, or no filter, respectively). It shows that when prefiltering the repository using $\mathcal{Q}_{all}$, both OWLS-MX variants behave similarly. Precision in this case drops at a high pace as the recall level increases, performing much worse than the rest of the combinations, though at the highest recall levels $\mathcal{Q}_{all}$ filtering slightly improves precision over OWLS-M0 (Figure 5a) without filtering. The low number of results obtained when filtering repositories using $\mathcal{Q}_{all}$ query is the cause for this low precision.

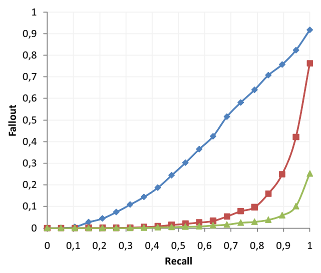
However, $\mathcal{Q}_{some}$ filtering performs reasonably well, with a loss in precision of at most 29% with respect to the precision obtained with OWLS-MX3 (M3) variant at high recall levels, as shown in Figure 5b. Interestingly, the evaluation shows that applying $\mathcal{Q}_{some}$ filtering to OWLS-M0 variant improves the precision of the answered set (up to 38% of difference), especially with recall levels over 50%. Thus, the more accurate results obtained by $\mathcal{Q}_{some}$ filtering help purely logic-based formalisms to find more relevant services, while avoiding more false positives.

In turn, Figure 6 represents false positives returned by each compared variant as their fallout. $\mathcal{Q}_{some}$ filtering applied to OWLS-M0 again improves the results when compared to the results of OWLS-M0 without applying any filter, as shown in Figure 6a. In the case of OWLS-MX3 (M3) (Figure 6b) fallout difference when applying $\mathcal{Q}_{some}$ filtering turns higher as recall level increases, especially from 70% on. As with precision, prefiltering repositories using $\mathcal{Q}_{all}$ query leads to much higher fallout levels, no matter the OWLS-MX variant used.

Obtained fallout performance results are a consequence of the prototype implementation used to evaluate our pro-



(a) OWLS-M0.



(b) OWLS-MX3 (M3).

Figure 6: Recall-Fallout effect when filtering OWLS-MX variants.

posal performance using SME[2], that requires each query result to be a ranked list of all the services that were registered in the system. Thus, our prototype also includes those services that do not pass the corresponding filter at the end of the ranked list. Analyzing filtering results of both queries, if only filtered services are taken into account when evaluating the fallout for each case, fallout will drop to less than 7% for $\mathcal{Q}_{some}$, and 0.02% for $\mathcal{Q}_{all}$ filter. Thus, the amount of false positives in a generic discovery scenario is reduced by using our prefiltering proposal, in general.

5.3. Discussion

As a general conclusion from the performed evaluation, though the more restrictive $\mathcal{Q}_{all}$ filter may be better suited to filter because it reduces the size of the service repository to a greater extent, $\mathcal{Q}_{some}$ filter turns to be more suitable in general because the precision penalty is negligible while execution time is fairly improved, outperforming service matchmaking without applying any filter. In turn, $\mathcal{Q}_{all}$ filter scales well in every situation, though the greater loss of precision have to be considered, so it may only be applied in scenarios with really large repositories.

Both filters clearly improve the subsequent discovery stage by reducing the search space for matchmaking algorithms. However, there is a trade-off between precision, recall, and execution time that should be evaluated, depending on the concrete scenario, in order to choose the filter to use. Actually, the current trend in the literature and real-world applications is to achieve better performance and usability, by sacrificing precision, recall, or both [4], so our proposal provides a feasible and efficient solution in this direction.

The main feature of using our proposed filters is that not only total execution time is very low, but actual filtering is efficiently executed, providing a high scalability. Furthermore, our solution also reduces the time needed for registering services for the matchmaking process, because the filter execution minimizes the number of candidate services. Consequently, a hybrid architecture can be applied, where $\mathcal{Q}_{all}$ filter is executed in the first place. If after performing service matchmaking, the obtained results did not present sufficient quality, $\mathcal{Q}_{some}$ filter could be used in place, executing again the matchmaking process. Note that even in the worst case, *i.e.* applying both filters and the corresponding matchmaking for each filtered repository, the total query execution time is 7.5 times faster than the OWLS-M0 matchmaking process for the whole service repository, and 8.6 times faster than OWLS-MX3 (M3). This approach is similar to the Best-Matches-Only solution proposed in [38], where if the most accurate results are found (*i.e.* $\mathcal{Q}_{all}$ returns good enough results), they are used, but in other case fairly appropriate results (*i.e.* results from $\mathcal{Q}_{some}$) may also be useful.

Additionally, another mixed approach may be taken, where both filters are jointly used before discovery and ranking processes take part. Thus, $\mathcal{Q}_{all}$ may be used to filter services that refers to concepts from the hard requirements of the user request, *i.e.* terms that have to be fulfilled in order to consider the corresponding service as a candidate. Then, $\mathcal{Q}_{some}$ filter can be applied to obtain services that refers to some of the concepts used in preferences, *i.e.* terms that state how candidate services should be ranked after discovery. Consequently, both filters can be integrated into one that take into consideration the differences between requirements and preferences [23].

Concerning the user requests applied in our evaluation, OWLS-TC v3 only provides information about inputs and outputs. However, an OWL-S user request may also contain preconditions, results, functional classification, and non-functional properties, in general. Our proposal can be seamlessly applied to these different terms of an OWL-S profile description, or in general to any SWS user request, because they also refer to concepts from domain ontologies. For instance, conditional expressions can be simply analyzed in order to obtain which concepts appear inside them. An early prototype on filtering WSMO services described in [10] is able to obtain those referred concepts from conditions and rules described within a WSMO capability. The evaluation of that approach presents similar results as the ones presented in this article, with respect to precision and improved performance of discovery when applying our proposed filters.

Finally, although the evaluation of our proposal has been carried out using OWLS-MX variants as the underlying service matchmaker, the prototype implementation can be easily adapted to any matchmaker that implements SME[2] interfaces. In the 4th International Semantic Service Selection (S3) Contest in 2010 we presented an evolution of the prototype implementation that allows to change the underlying service matchmaker[8]. EMMA – an Enhanced MatchMaking Add-on – was implemented as a configurable OWL-S matchmaking plugin compatible with SME[2] 2.1.1. Although EMMA offers a similar precision as

[8]http://www-ags.dfki.uni-sb.de/~klusch/s3/s3c-2010-summary-report-v2.pdf

Table 2: Related work analysis.

Proposal	AUT	APP	INT
Lamparter *et al.* [39]	∼	✓	×
Iqbal *et al.* [40]	∼	✓	×
Sbodio *et al.* [41]	∼	✓	×
Siberski *et al.* [42]	×	✓	×
Pedrinaci *et al.* [32]	×	✓	✓
Chabeb *et al.* [33]	×	✓	✓
Norton *et al.* [34]	×	✓	✓
Agarwal *et al.* [6]	∼	×	×
Stollberg *et al.* [5]	✓	×	✓
Klusch *et al.* [7]	✓	∼	×
Kiefer *et al.* [43]	✓	✓	×
Carenini *et al.* [44]	∼	×	✓
Our proposal	✓	✓	✓

the prototype evaluated in this article, average query response time is worse than the prototype, because of the way SME^2 plugins register the available services. This issue is identified in the S3 Contest 2010 report, so next version of SME^2 application will allow the use of pre-filtering techniques, such as our proposed solution.

6. Related Work

In the following, we discuss proposals related with our work, analyzing their relationship with our solution. Firstly, we describe approaches that use Semantic Web query languages to perform discovery and ranking. Then, we discuss some SWS ontology integration proposals that our proposal can be applied to, providing a higher interoperability. Finally, we analyze proposals that offer solutions to improve discovery processes.

We have focused our analysis on three key aspects to compare related work with our proposed filtering solution, namely: (1) if discovery can be automatically optimized by using the analyzed approach (AUTomation), (2) if it can be applied to any SWS framework (APPlicability), and (3) to what extent each proposal can be integrated with other discovery approaches in order to further optimize their performance (INTegrability). Table 2 sums up our comparison results, showing wether the discussed proposals provide full (✓), partial (∼), or no (×) support to the analyzed aspects.

There exist several proposals that use a Semantic Web query language to perform discovery and ranking of services [45], though they do not use queries explicitly to filter repositories. They choose SPARQL as their base language, enabling their applicability to any SWS framework, though they add some extensions in order to fully support these tasks. The analyzed proposals provide some optimizations to their algorithms, though they are not fully automatized. Additionally, their optimized discovery approaches cannot be integrated with other solutions. Thus, Lamparter *et al.* [39] provide an ontology to represent service offers and requests that conforms the foundations for a discovery and selection process, which is performed using rules in SWRL[46] and SPARQL queries. These queries includes predicates that have to be evaluated at run-time, so they include an extension to SPARQL that is implemented using different proposed algorithms. Thus, a query for a user request is provided, though this query depends on rules that change the matchmaking policy, allowing some *ad hoc* optimizations.

Another discovery approach that uses SPARQL to actually perform semantic service discovery is proposed by Iqbal *et al.* in [40]. In this case, authors embed semantic information about services using SAWSDL. Thus, they define pre and post-conditions of services using SPARQL `CONSTRUCT` queries so that depending on each service functionality, they add corresponding RDF tuples representing that functionality to the knowledge base. Then, their discovery algorithm use an `ASK` query to check whether a service fulfills a user request or not, returning the results. In this case, authors use standard-only SPARQL queries to perform discovery, and their service discovery algorithm can progressively relax the conditions from the user request in case no results are returned in the first place, as in our hybrid approach discussed in Section 5.3.

Sbodio *et al.* also introduce SPARQL queries to describe OWL-S service pre and post-conditions, and user requests, providing a matchmaker implementation based on agents called SPARQLent [41]. They discuss a complete discovery solution that uses SPARQL queries to modify and ask the agent's knowledge base, evaluating their proposal against OWLS-MX using SME^2, as in our work. Although they provide some optimizations to their discovery algorithm, our proposal could be also applied to further improve their agent performance, by preventing it to load the complete set of available services on the repository.

Finally, there is another approach, more related to ranking, presented in [42], where Siberski *et al.* propose an extension to SPARQL so that preferences are described directly using the query language, without basing on existing preferences and non-functional properties ontologies, as in other semantic ranking approaches [23, 25, 31]. They provide a `PREFERRING` clause that states preferences among values of variables, similar to `FILTER` expressions. However, this approach does not have the flexibility and reasoning facilities that provides a solution based on an external ontology, and it uses non-standard SPARQL extensions without providing an implementation.

Concerning the integration of SWS frameworks, there are a number of proposals in the literature that address this issue to tackle applicability and integrability of different discovery solutions. However, they do not provide facilities to automatically optimize discovery mechanisms. Pedrinaci *et al.* present a service repository called iServe that exposes service descriptions as linked data in terms of a Minimal Service Model (MSM) [32]. This model serves the purpose of an ontology of integration that simplifies SWS frameworks, integrating not only OWL-S, WSMO, SAWSDL and WSMO-Lite services, but also MicroWSMO [47] or SA-REST [48] descriptions of Web APIs. Our proposal can be also applied to MSM so that our filters can be applied to services registered in iServe, that provides a SPARQL endpoint that can be used to retrieve descriptions for a subsequent discovery.

Chabeb *et al.* describe another ontology of integration in [33]. They discuss a systematic approach to generate mappings between OWL-S, WSMO and plain WSDL services, matching concepts from the different SWS ontologies using similarity techniques that validate the inferred correspondences. Their resulting ontology merges concepts from different SWS frameworks, as opposite to the MSM, that only capture part of those SWS ontologies, offering a more concise approach. Our proposed filters can also be applied to this merged ontology, providing a global-as-view approach to query OWL-S, WSMO and WSDL services [49].

Norton *et al.* present a similar proposal in [34], where the authors also take a 'union' approach to integrate OWL-S, WSMO, and WSMO-Lite descriptions. They present several SPARQL CONSTRUCT queries that transform SWS descriptions to and from the Semantic SOA Reference Ontology, a standard proposed by OASIS. Applicability of our proposal can be also achieved by implementing our filters using this reference ontology, and then using Norton *et al.* approach to project SWS descriptions into the model, allowing our proposed queries to be applied to them.

Concerning the need for an improved discovery process which tackles scalability issues, Agarwal *et al.* discuss a hybrid approach that use different discovery mechanisms together, in order to improve discovery performance [6]. They also propose a simple filtering stage based on an efficient classification-based discovery. However, this filter rely on a less expressive user request. Our proposal may be also applied to the authors hybrid approach in order to further improve discovery but using a more expressive model to describe user requests [23].

In order to improve discovery engines, Stollberg *et al.* provides a caching mechanism that reduces the search space and minimizes matchmaking operations [5]. The proposed cache uses a graph that stores relationships between user requests described as WSMO goal templates, and their related services. Thus, goal instances are compared with cached templates in terms of semantic similarity, and if there is a match, only the related services stored in the graph are used for the subsequent discovery. This proposal can be complemented by using our filters when creating the cache graph.

Klusch *et al.* take a different approach in OWLS-MX [7], where they present a hybrid matchmaker that combines information retrieval techniques, such as syntactic similarity, with classical DL-based discovery, in order to improve OWL-S service matchmaking. Their comprehensive evaluation proves that hybrid approaches present a better performance than classical ones. Our proposal has been applied to their proposed OWLS-MX variants, further improving performance results, especially on logic-based ones as discussed in Section 5. Moreover, similar solutions have been also proposed by the authors for WSMO [50] and SAWSDL [51] service matchmaking.

Kiefer *et al.* present in [43] another hybrid matchmaker called iMatcher, which uses information retrieval techniques to improve the discovery process. In this proposal, authors use a SPARQL extension (iSPARQL [17]) that enables the introduction of similarity operators into query elements. Thus, different similarity strategies are combined with logic-based discovery in order to improve precision and recall of the matchmaking process. Additionally, machine-learning can also be applied to automatically choose the most appropriate strategy to be included in the hybrid matchmaking, for each case.

Finally, Carenini *et al.* propose a customizable hybrid architecture for SWS discovery and ranking named GLUE2 [44]. GLUE2 offers a set of specialized discovery components, such as functional discovery, dynamic discovery, non-functional discovery, and ranking, among other additional components [52]. Based on WSMO, GLUE2 enables the configuration of the discovery workflow on a case by case basis. Thus, as with other hybrid approaches such as [30], our proposed filtering stage can be integrated with GLUE2 as an additional component so that it can be included in any hybrid discovery workflow.

Note that most proposed discovery optimization proposals are coupled with a concrete SWS framework and a corresponding discovery mechanism, as shown in Table 2. However, we designed our filtering proposal to allow its application to any SWS definition framework and available discovery mechanisms, so it can even be applied on top of any of the discussed proposals that already improve service discovery process, as our experimental evaluation shows in Section 5.

7. Conclusions

Although Semantic Web query languages are not widely used for SWS discovery and ranking, they can certainly play a role in these scenarios. As discussed in this paper, some authors extend SPARQL query language to directly support these stages, but our proposal sticks to the recommendation at the time of writing (1.0), providing two different filter queries that may be used before actual discovery process in order to reduce the set of available services from the initial repository. Consequently, the reduced search space further improves scalability and performance in discovery and ranking stages, decreasing the total execution time and memory consumption of these processes, with a contained penalty on precision, recall and fallout.

In this work, we have run comprehensive evaluation tests, analyzing the actual improvement. The conclusions obtained are mainly that our proposal effectively reduces the search space, while it conforms a generic solution, adaptable to any SWS framework that a potential user may want to use. Particularly, we discuss an application to OWL-S-based services in order to test the implemented prototype using the OWLS-TC test collection, though we also introduce additional applications to other frameworks.

Our proposal of including a (possibly multiple) filtering stage before the discovery and ranking processes has several additional benefits, summarized in the following:

- Proposed filters are generic, so they can be used no matter what kind of user request and service descriptions are defined for each concrete scenario. Corresponding SPARQL queries can be generated automatically from a given user request.

- Our proposal does not distinguish between types of concepts, *i.e.* both functional and non-functional concepts can be used to filter the repository. In consequence, concepts being used for both discovery and ranking stages can be considered.

- Filters can be applied to any SWS framework because they are based only on domain concepts referred by service descriptions and user requests.

- Our filtering stage can be applied to improve any currently available matchmaking implementation. The actual improvement on the overall discovery performance depends on the nature of the matchmaker, providing a high impact on performance with DL-based matchmakers, and a relative impact on hybrid approaches.

- Our solution is based on the current standard query language for the Semantic Web, *i.e.* SPARQL 1.0. Nevertheless, our proposed queries do not use any extension to the standard, so they are compatible with most SPARQL implementations.

In conclusion, our proposal follows the current research trend on developing lightweight, scalable applications and extensions that effectively enable the adoption of Semantic Web technologies, by improving current discovery mechanisms in terms of scalability and performance, while offering a contained penalty on precision with respect to classical, heavyweight approaches to SWS matchmaking.

Acknowledgments

This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project SETI (TIN2009-07366), by the Andalusian Government under projects ISABEL (TIC-2533) and THEOS (TIC-5906), by the EU FP7 IST project 27867 SOA4All, and by the EC FP7 Network of Excellence 215483 S-CUBE.

Authors would like to thank the reviewers for their invaluable opinion and recommendations that improved this work substantially; Carlos Pedrinaci and Marco Luca Sbodio for their insightful comments and suggestions on early versions; the S3 Contest organization for their evaluation tools and support; and Adela del-Río-Ortega and Carlos Rivero for their useful technical support and reviews.

References

[1] V. Haarslev, R. Möller, On the Scalability of Description Logic Instance Retrieval, Journal of Automated Reasoning 41 (2) (2008) 99–142.

[2] J. Davies, J. Domingue, C. Pedrinaci, D. Fensel, R. González-Cabero, M. Potter, M. Richardson, Towars the open service web, BT Technology Journal 26 (2).

[3] J. Domingue, D. Fensel, R. González-Cabero, SOA4All, Enabling the SOA Revolution on a World Wide Scale, in: ICSC, IEEE Computer Society, 2008, pp. 530–537.

[4] D. Fensel, The Potential and Limitations of Semantics Applied to the Future Internet, in: J. Filipe, J. Cordeiro (Eds.), WEBIST, INSTICC Press, 2009, pp. 15–15.

[5] M. Stollberg, M. Hepp, J. Hoffman, A Caching Mechanism for Semantic Web Service Discovery, in: K. Aberer, et al. (Eds.), ISWC/ASWC, Vol. 4825 of LNCS, Springer, 2007, pp. 480–493.

[6] S. Agarwal, M. Junghans, O. Fabre, I. Toma, J. P. Lorre, D5.3.1 First Service Discovery Prototype, Tech. rep., SOA4All (2009).

[7] M. Klusch, B. Fries, K. Sycara, OWLS-MX: A hybrid Semantic Web service matchmaker for OWL-S services, Web Semantics: Science, Services and Agents on the World Wide Web 7 (2) (2009) 121–133.

[8] E. Prud'hommeaux, A. Seaborne, SPARQL Query Language for RDF, Recommendation, W3C (2008).

[9] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, ACM Transactions on Database Systems 34 (3) (2009) 1–45.

[10] J. M. García, D. Ruiz, A. Ruiz-Cortés, A lightweight prototype implementation of SPARQL filters for WSMO-based discovery, Tech. Rep. ISA-11-TR-01, Applied Software Engineering Research Group - University of Seville (2011).

[11] J. Bailey, F. Bry, T. Furche, S. Schaffert, Web and Semantic Web Query Languages: A Survey, in: N. Eisinger, J. Maluszynski (Eds.), Reasoning Web, Vol. 3564 of LNCS, Springer, 2005, pp. 35–133.

[12] M. Sintek, S. Decker, Triple - a query, inference, and transformation language for the semantic web, in: I. Horrocks, J. Hendler (Eds.), The Semantic Web - ISWC 2002, Vol. 2342 of LNCS, Springer, 2002, pp. 364–378.

[13] E. Sirin, B. Parsia, SPARQL-DL: SPARQL Query for OWL-DL, in: C. Golbreich, A. Kalyanpur, B. Parsia (Eds.), OWLED, Vol. 258 of CEUR Workshop Proceedings, 2007.

[14] F. Manola, E. Miller, RDF Primer, Recommendation, W3C (2004).

[15] D. L. McGuinness, F. van Harmelen, OWL Web Ontology Language Overview, Recommendation, W3C (Feb. 2004).

[16] D. Beckett, T. Berners-Lee, Turtle - terse rdf triple language, Team submission, W3C (2011).

[17] C. Kiefer, A. Bernstein, M. Stocker, The Fundamentals of iSPARQL: A Virtual Triple Approach for Similarity-Based Semantic Web Tasks, in: K. Aberer, et al. (Eds.), ISWC/ASWC, Vol. 4825 of LNCS, Springer, 2007, pp. 295–309.

[18] F. Alkhateeb, J. F. Baget, J. Euzenat, Constrained Regular Expressions in SPARQL, in: SWWS, CSREA Press, 2008, pp. 91–99.

[19] J. Farrell, H. Lausen, Semantic Annotations for WSDL and XML Schema, Recommendation, W3C (Aug. 2007).

[20] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, Others, OWL-S: Semantic Markup for Web Services, Tech. Rep. 1.2, DAML (2006).

[21] D. Fensel, H. Lausen, A. Polleres, J. de Bruijn, M. Stollberg, D. Roman, J. Domingue, Enabling Semantic Web Services: The Web Service Modeling Ontology, Springer, 2007.

[22] T. Vitvar, J. Kopecky, J. Viskova, D. Fensel, WSMO-Lite Annotations for Web Services, in: S. Bechhofer, M. Hauswirth, J. Hoffmann, M. Koubarakis (Eds.), ESWC, Vol. 5021 of LNCS, Springer, 2008, pp. 674–689.

[23] J. M. García, D. Ruiz, A. Ruiz-Cortés, A Model of User Preferences for Semantic Services Discovery and Ranking, in: L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, T. Tudorache (Eds.), ESWC (2), Vol. 6089 of LNCS, Springer, 2010, pp. 1–14.

[24] E. Maximilien, M. Singh, A framework and ontology for dynamic Web services selection, IEEE Internet Computing 8 (5) (2004) 84–93.

[25] X. Wang, T. Vitvar, M. Kerrigan, I. Toma, A QoS-Aware Selection Model for Semantic Web Services, in: A. Dan, W. Lamersdorf (Eds.), ICSOC, Vol. 4294 of LNCS, Springer, 2006, pp. 390–401.

[26] L. Li, I. Horrocks, A software framework for matchmaking based on semantic web technology, in: WWW, ACM Press, 2003, pp. 331–339.

[27] C. Lutz, U. Sattler, A Proposal for Describing Services with DLs, in: Int. Workshop on Description Logics, 2002.

[28] K. Sycara, M. Paolucci, A. Ankolekar, N. Srinivasan, Automated discovery, interaction and composition of Semantic Web services, Web Semantics: Science, Services and Agents on the World Wide Web 1 (1) (2003) 27–46.

[29] I. Toma, D. Roman, D. Fensel, B. Sapkota, J. Gomez, A multi-criteria service ranking approach based on non-functional properties rules evaluation, in: ICSOC, Vol. 4749 of LNCS, Springer, 2007, pp. 435–441.

[30] J. M. García, D. Ruiz, A. Ruiz-Cortés, O. Martín-Díaz, M. Resinas, An hybrid, QoS-aware discovery of semantic web services using constraint programming, in: B. Krämer, K.-J. Lin, P. Narasimhan (Eds.), ICSOC, Vol. 4749 of LNCS, Springer, 2007, pp. 69–80.

[31] J. M. García, I. Toma, D. Ruiz, A. Ruiz-cortés, A service ranker based on logic rules evaluation and constraint programming, in: 2nd ECOWS Non-Functional Properties and Service Level Agreements in Service Oriented Computing Workshop, Vol. 411 of CEUR Workshop Proceedings, 2008.

[32] C. Pedrinaci, D. Liu, M. Maleshkova, D. Lambert, J. Kopecky, J. Domingue, iServe: a Linked Services Publishing Platform, in: Ontology Repositories and Editors for the Semantic Web Workshop at ESWC 2010, Vol. 596 of CEUR Workshop Proceedings, 2010.

[33] Y. Chabeb, S. Tata, D. Belaïd, Toward an Integrated Ontology for Web Services, in: M. Perry, H. Sasaki, M. Ehmann, G. Ortiz Bellot, O. Dini (Eds.), ICIW, IEEE Computer Society, 2009, pp. 462–467.

[34] B. Norton, M. Kerrigan, A. Marte, On the Use of Transformation and Linked Data Principles in a Generic Repository for Semantic Web Services, in: Ontology Repositories and Editors for the Semantic Web Workshop at ESWC 2010, Vol. 596 of CEUR Workshop Proceedings, 2010.

[35] V. Haarslev, R. Möller, RACER System Description., in: IJCAR, 2001, pp. 701–706.

[36] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL-DL reasoner, Web Semantics: Science, Services and Agents on the World Wide Web 5 (2) (2007) 51–53.

[37] R. Baeza-Yates, B. Ribeiro-Neto, Modern information retrieval, Addison-Wesley, 1999.

[38] W. Kießling, Foundations of preferences in database systems, in: VLDB, Morgan Kaufmann, 2002, pp. 311–322.

[39] S. Lamparter, A. Ankolekar, R. Studer, S. Grimm, Preference-based selection of highly configurable web services, in: WWW, ACM Press, 2007, pp. 1013–1022.

[40] K. Iqbal, M. L. Sbodio, V. Peristeras, G. Giuliani, Semantic Service Discovery using SAWSDL and SPARQL, in: SKG, IEEE Computer Society, 2008, pp. 205–212.

[41] M. L. Sbodio, D. Martin, C. Moulin, Discovering Semantic Web Services using SPARQL and Intelligent Agents, Web Semantics: Science, Services and Agents on the World Wide Web (2010) –.

[42] W. Siberski, J. Z. Pan, U. Thaden, Querying the Semantic Web with Preferences, in: ISWC, Vol. 4273 of LNCS, Springer, 2006, pp. 612–624.

[43] C. Kiefer, A. Bernstein, The Creation and Evaluation of iSPARQL Strategies for Matchmaking, in: S. Bechhofer, M. Hauswirth, J. Hoffmann, M. Koubarakis (Eds.), ESWC, Vol. 5021 of LNCS, Springer, 2008, pp. 463–477.

[44] A. Carenini, D. Cerizza, M. Comerio, E. D. Valle, F. de Paoli, A. Maurino, M. Palmonari, A. Turati, GLUE2: A Web Service Discovery Engine with Non-Functional Properties, in: ECOWS, IEEE Computer Society, 2008, pp. 21–30.

[45] J. M. García, C. Rivero, D. Ruiz, A. Ruiz-Cortés, On Using Semantic Web Query Languages for Semantic Web Services Provisioning, in: SWWS, CSREA Press, 2009, pp. 67–71.

[46] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, SWRL: A Semantic Web Rule Language Combining OWL and RuleML, Technical report, W3C Member Submission (2004).

[47] M. Maleshkova, J. Kopecky, C. Pedrinaci, Adapting SAWSDL for Semantic Annotations of RESTful Services, in: R. Meersman, P. Herrero, T. S. Dillon (Eds.), OTM Workshops, Vol. 5872 of LNCS, Springer, 2009, pp. 917–926.

[48] A. P. Sheth, K. Gomadam, J. Lathem, SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups, IEEE Internet Computing 11 (6) (2007) 91–94.

[49] D. Calvanese, G. de Giacomo, M. Lenzerini, A Framework for Ontology Integration, in: I. F. Cruz, S. Decker, J. Euzenat, D. L. McGuinness (Eds.), The Emerging Semantic Web, Frontiers in Artificial Intelligence and Applications, IOS press, 2001.

[50] M. Klusch, F. Kaufer, WSMO-MX: A hybrid Semantic Web service matchmaker, Web Intelligence and Agent Systems 7 (1) (2009) 23–42.

[51] M. Klusch, P. Kapahnke, I. Zinnikus, SAWSDL-MX2: A Machine-Learning Approach for Integrating Semantic Web Service Matchmaking Variants, in: ICWS, IEEE Computer Society, 2009, pp. 335–342.

[52] M. Palmonari, M. Comerio, F. de Paoli, Effective and Flexible NFP-Based Ranking of Web Services, in: L. Baresi, C.-H. Chi, J. Suzuki (Eds.), ICSOC-ServiceWave, Vol. 5900 of LNCS, Springer, 2009, pp. 546–560.