A Lightweight Prototype Implementation of SPARQL Filters for WSMO-based Discovery

José María García, David Ruiz and Antonio Ruiz-Cortés
{josemgarcia,druiz,aruiz}@us.es

Applied Software Engineering Research Group
University of Seville, Spain
May 2011

Technical Report ISA-11-TR-01

This report was prepared by the

Applied Software Engineering Research Group (ISA)
Department of computer languages and systems
Av/ Reina Mercedes S/N, 41012 Seville, Spain
http://www.isa.us.es/

Copyright©2011 by ISA Research Group.

Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and 'No Warranty' statements are included with all reproductions and derivative works.

NO WARRANTY
THIS ISA RESEARCH GROUP MATERIAL IS FURNISHED ON AN 'AS-IS' BASIS. ISA RESEARCH GROUP MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTIBILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Support: 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.

List of changes

Version	Date	Description
1.0	May 2011	First release

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 more complex services become 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 property requested by the user before the actual discovery. By using this approach, the search space for discovery mechanisms is fairly reduced, consequently improving the overall performance of this task. Furthermore, this particular solution do not provide yet another discovery mechanism, but it is easily applicable to any of the existing ones. Moreover, proposed queries are automatically generated from service requests, transparently to the user. In order to validate our proposal, a concrete application to a WSMO discovery scenario is showcased in this paper. A comprehensive performance analysis is also presented in order to test and compare the results obtained from proposed queries and classical discovery approaches, discussing the benefits of our proposal.

Contents

1 Introduction **1**

2 Background **3**
2.1 Querying the Semantic Web 3
2.2 Semantic Web Services 4
2.3 Discovering and Ranking 6

3 Filtering Service Repositories using SPARQL **7**
3.1 Querying a Service Repository 7
3.2 Automatic Generation of Queries 10

4 A WSMO Implementation **11**
4.1 Ontology Mapping 11
4.2 Query Rewriting 13
4.3 Query Execution 15

5 Analysis and Evaluation **16**
5.1 Defining the Experiments 16
5.2 Analyzing Tests Results 19
5.2.1 Execution Time 19
5.2.2 Q_{some} Results 21
5.2.3 Q_{all} Results 22
5.2.4 Discovery Improvement 23
5.3 Statistical Analysis 24
5.4 Discussion 25

6 Related Work **26**

7 Conclusions **28**

List of Figures

1 Integrated ontology for querying a service repository. 4
2 Semantic discovery and ranking processes. 6
3 Service procurement architecture including a SPARQL filtering stage. 7
4 Parameters and output variables of experiments. 17
5 Execution time results. 20
6 Filtering results for Q_{some}. 21
7 Filtering results for Q_{all}. 22
8 Performance evaluation of discovery mechanisms. 23

List of Tables

1 Mappings between WSMO and our integrated model 12
2 Mean and std. deviation, and IC of evaluated variables. 24
3 Pearson correlations between evaluated parameters. 25

1 Introduction

Current service discovery solutions that rely on Semantic Web technologies are not sufficiently scalable, so large and complex service repositories cannot be properly handled by them. Although there is a great research effort to improve discovery mechanisms, the underlying reasoning facilities do not scale well in general [14]. The approach taken in this paper does not consist on another discovery mechanism, but on the inclusion of a preprocessing stage where service repositories are filtered using two different queries, so that the search space for discovery processes is reduced in our experiments, on average, from 78.65% of the original repository size up to 5.83%, depending on the query used and the nature of the repository, consequently improving service discovery by reducing its execution time from a 63.93% to a 11.32% of the original time needed for each corresponding repository.

Service repositories are increasing the number of registered services at a high pace, as more services can be found publicly[1]. Furthermore, semantic definitions of these services are richer and more complex than ever before, integrating many heterogeneous concepts. That leads to an scenario where discovery mechanisms based on different logic formalisms have scalability issues. Consequently, improvements and optimizations on those mechanisms are needed in order to enhance the usability of Semantic Web Services (SWS) [7, 9].

In order to alleviate the scalability problem on semantic discovery mechanisms, our proposal takes the novel approach of reducing the input for those mechanisms, so that the resulting process is more streamlined, only reasoning about services which actually matter with respect to the user request. Thus, services that can be discarded *a priori*, because they are not related at all with requirements and preferences stated by the user, are filtered so that the search space is considerably reduced prior to actual discovery mechanisms.

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 for renting a car, 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 car rentals domain concepts, in this case. Thus, on an evenly distributed repository, where there were the same number of services related to each domain in the example (hotels, planes, car rentals, and insurances), only 25% of the services would be considered for the discovery process, considerably improving

[1] *seekda.com* service crawler has indexed at the moment of writing 28,529 services.

its performance.

For the proposed preprocessing, the user request is analyzed in order to extract the concepts that are being used in its semantic definition (in the above example, some of them could be *Car* or *DrivingLicence*, for instance). Then, the repository is filtered so that only services that uses those same concepts are selected to become the input for the subsequent discovery process (*e.g.* services whose definitions refer to *Car* and/or *DrivingLicence* concepts, in the latter case). The filter is performed in our approach by two different SPARQL[27] queries, namely Q_{some} and Q_{all}. The former selects service definitions that refers to some (at least one) of the concepts referred by a user request, assuming that those services may satisfy its requirements and/or preferences despite the missing information. In turn, the latter query 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.

Our solution does not pretend to provide yet another discovery mechanism, but to introduce a preprocessing stage, based on an accepted standard, that yields a notable improvement on heavyweight semantic processes, such as matchmaking of services. To the best of our knowledge, there is no proposals on filtering semantically-enhanced service repositories, but it is known that some sort of preprocessing can alleviate discovery and ranking tasks performed on those repositories [2]. To sum up, the main contributions of the proposal presented in this paper are the following:

1. A technique to improve semantic service discovery performance is proposed, based on a preprocessing stage where repositories are filtered 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. In this work, a third-party lightweight Description Logics discovery mechanism is used to illustrate this point, namely WSMX DL discovery.

3. Preprocessing 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. Each one is analyzed and thoroughly discussed throughout the article.

4. A comprehensive, experimental study is carried out in order to assess the actual impact of our proposal. Several theoretical scenarios were

generated, from repositories whose services refer to a uniformly distributed set of concepts, to scenarios where there are some concepts that are more predominant than others in service descriptions.

The rest of the paper is structured as follows. Firstly, Sec. 2 presents some background information to contextualize and motivate the proposal. In Sec. 3 we show how to use SPARQL queries within a discovery scenario, presenting both generic and more specific queries that can be applied in different cases. Section 4 discuss the integration and implementation of our proposal applied to WSMO service discovery. Then, in Sec. 5 the experimental study done is explained, analyzing the results and discussing the advantages of our proposal. Section 6 outlines the related work on this field. Finally, in Sec. 7 we discuss the conclusions.

2 Background

Using a Semantic Web query language becomes a straightforward option to perform 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. In the following we introduce these background elements of our proposal in order to contextualize and further motivate our work.

2.1 Querying the Semantic Web

There exists two main approaches for Semantic Web query languages: RDF-based and DL-based query languages [4, 30]. On the one hand, RDF-based query languages allow to fetch RDF triples based on matching triple patterns with RDF graphs. On the other hand, DL-based query languages allow to query OWL-DL ontologies, being able to search concepts, properties, and individuals. Although DL-based query languages provide more reasoning mechanisms than RDF-based ones, the former are not mature enough and they are in early stages of development [4], so the latter are more widely used, especially SPARQL [27], which is the current W3C Recommendation.

There are several RDF-based query languages with different features [4], but SPARQL is the only language that is a W3C recommendation [27]. 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

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

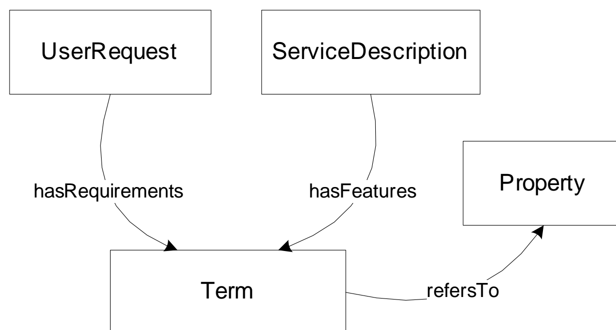

Figure 1: Integrated ontology for querying a service repository.

Virtuoso, Sesame, or ARQ,[3] which is included in the Jena Semantic Web Framework for Java. The latter is the chosen one for our evaluation tests (*cf.* Sec. 4).

SPARQL, as an RDF-based query language, can handle RDF triples, which conforms the very foundations of a Semantic Web ontology. Its main approach to query semantic repositories is to match RDF patterns, consisting of triples of subject, predicate, and 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 directly; `CONSTRUCT` queries build an RDF graph based on a template defined in the query; `ASK` queries test whether or not a pattern has any solution; and `DESCRIBE` queries return an RDF graph not based on a template in the query (as in `CONSTRUCT` queries) but on a pre-configured graph.

Currently, the SPARQL recommendation is being revised to apply some 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 [3], or even approaches that add DL-based languages features [30]. However, in this work we only use standard SPARQL to improve discovery processes, though some of the extensions discussed can be also applied (*cf.* Sec. 6).

2.2 Semantic Web Services

SWS are often defined using specific ontologies, such as OWL-S [24] or WSMO [28], which provide basic tools to discover and rank services in terms of user

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

requests described within the same ontology. However, these ontologies can be extended to improve those tasks using other ontologies [13, 25, 35], 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 or WSMO, is composed of several terms that define service features, which can be related to functional or non-functional properties. 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 with one or more particular property.

For instance, WSMO service descriptions feature the service functionality within a *capability* description. Capabilities define preconditions, assumptions, postconditions and effects about the provided functionality using several axioms or terms, and can be annotated to express non-functional properties. User requests are described as WSMO *goals*, that provide a similar structure to define requested services.

Figure 1 shows this abstraction as a generic, integrated ontology, where `UserRequest`, `ServiceDescription`, `Term`, and `Property` are the classes that model previously referred elements, and `hasRequirements`, `hasFeatures` and `refersTo` are object properties defining the relationships between instances of those classes. Although only requirements are taken into consideration in our proposal, preferences descriptions for SWS ranking may be also included easily, by using the extended ontology proposed in [13].

In order to decouple our proposal from concrete SWS ontologies, there is a need for this kind of ontology integration . Thus, our work is defined after the generic, integrated ontology presented in Fig. 1, that merges user requests and descriptions from different SWS ontologies, such as OWL-S or WSMO, by means of specified mappings.

Our proposal is also based on the assumption that service descriptions and user requests are defined in terms of concepts from a domain ontology, which depends on the concrete scenario. As an example, consider the scenario described in Sec. 1, where a repository contains several services related to travel domains. Service descriptions may *feature* several statements or *terms* defining their provided capabilities and non-functional properties using concepts from a travel domain ontology. These referred concepts are interpreted as `Property` instances in the proposed integrated ontology. Thus, a car rental service description will contain some terms that refers to concepts or properties like *Car* or *Price*, for instance. User requests are described in a similar way, though related terms are linked using the `hasRequirements` object property.

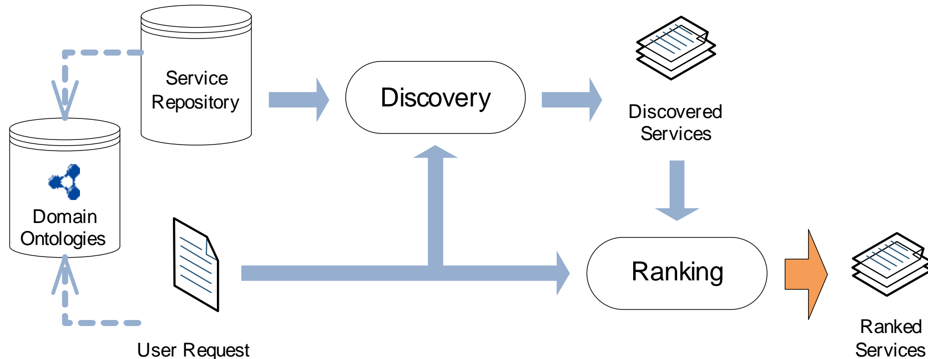

Figure 2: Semantic discovery and ranking processes.

2.3 Discovering and Ranking

The common use case for discovery and ranking of SWS is depicted in Fig. 2. Starting from a service repository containing definitions either using OWL-S [24], WSMO [28], SAWSDL [8], or WSMO-Lite [20], for instance, the *discovery* process tries to match user requirements with these available service definitions, which are described in terms of domain ontologies. This matchmaking is usually performed using logic reasoning techniques, such as DL reasoners [22, 23, 33], logic programming [34, 35], or hybrid approaches [11, 12, 19]. The resulting discovered services are a subset of the initial repository, where each instance of this subset are 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, so it can be used to match corresponding services in the discovery stage. Some authors extend WSMO goals to also include non-functional properties, which can be used to rank previously discovered services [35, 12]. Therefore, using both discovered services and preferences described in the user request [13], the *ranking* process returns an ordered list of those services in terms of stated preferences.

SWS discovery techniques, particularly, suffer from performance and scalability issues in this context. The main motivation of this work is to come out with a solution that effectively improves discovery and ranking, making them into more lightweight processes.

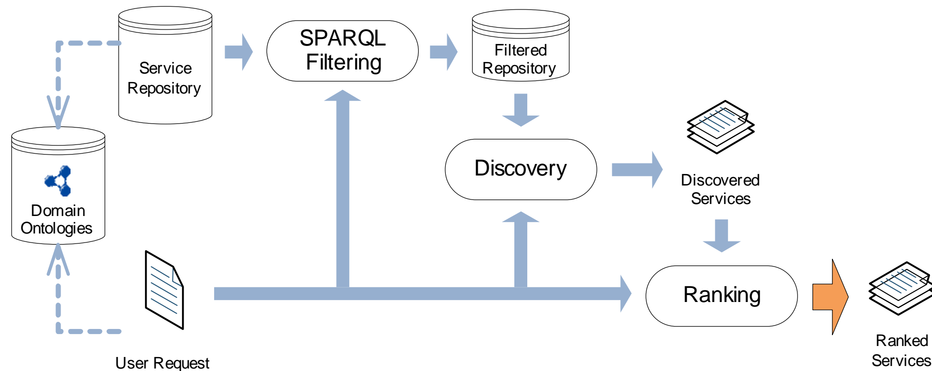

Figure 3: Service procurement architecture including a SPARQL filtering stage.

3 Filtering Service Repositories using SPARQL

Current SPARQL implementations do not provide sufficient reasoning facilities to adequately perform SWS discovery and ranking, which tend to be complex, heavyweight processes. Nevertheless, in this work we propose the introduction of standard, automatically generated SPARQL queries to improve the performance of those processes without changing the underlying mechanisms, by pre-selecting service candidates from a repository with respect to the user requirements.

3.1 Querying a Service Repository

In order to simplify and generalize our proposal, the queries presented in the following are based on the integrated ontology already shown in Fig. 1, so that they can be easily adapted to use more specific SWS ontologies, using mappings or mediation ontologies [5]. *Cf.* Sec. 4 for a materialization in WSMO.

Once services and user requests are well established, the stage where SPARQL can be used to improve discovery and ranking processes has to be defined. Our proposal adds a new stage previous to the discovery process, where the service repository is filtered using SPARQL queries. In Fig. 3 the proposed architecture is showcased. The aim of the filtering stage is to obtain services from the original repository that may be possibly matched with the user request in the discovery process, discarding those ones that cannot fulfill that request at all.

The key point in this scenario is that service repositories may contain thousands of services, but most of them may not be related to the service

requested by the user. Assuming both service descriptions and user requests are defined using the ontology from Fig. 1, our proposed SPARQL queries for the filtering stage discriminates service descriptions depending on whether properties referenced in their terms are present in the user request or not. To this extent, two different filters can be applied depending on how strict they are. On the one hand, one of the filters (Q_{some}) returns those service descriptions that refer to some (at least one) of the properties that are also referred by the user request. On the other hand, the stricter filter (Q_{all}) only returns service descriptions that refer to the same concepts as the user request. In turn, services whose features do not refer to any of the properties referred in the requirements of the user request are discarded by both filters, because in that case it can be inferred that they are not related to the service the user is searching for.

The firstly proposed filtering query Q_{some}, shown in Listing 1, returns the URIs set of service descriptions that match the graph patterns enumerated in the `WHERE` clause, *i.e.* those services that *some* of the properties referred by their features are also referred by the requirements of the user request. `URInstance` is the name of the concrete instance of `UserRequest` class used to look for requested services. Thus, service descriptions that do not refer to any property used in the user request are discarded for the following discovery stage. Consequently, the number of services that are considered for discovery (and ranking) may be reduced, improving the overall performance and scalability of SWS procurement. The actual degree of this improvement is analyzed in Sec. 5.

Listing 1: Q_{some} SPARQL query.

```
PREFIX ao: <http://www.isa.us.es/2009/IntegratedOntology.owl#>
SELECT DISTINCT ?service
WHERE { # match properties referred by services...
  ?service ao:hasFeatures ?featuredTerms.
  ?featuredTerms ao:refersTo ?properties.
  # ... with those referred by the user request
  :URInstance ao:hasRequirements ?requiredTerms.
  ?requiredTerms ao:refersTo ?properties.
}
```

Although this query effectively reduce the search space, the pattern matching performed by SPARQL implementations consists on a Cartesian product between properties from the user preference and services descriptions. Thus, even service descriptions that only share some, but not all properties with the user request are returned, though they may not be going to be matched in the discovery stage, because of the lack of enough information in their descriptions.

In order to obtain fewer but closer service descriptions with respect to the

user preference, another SPARQL query is proposed, which is more specific and precise. Q_{all} query selects services from the repository that exactly have terms about *all* the properties which are referenced by the user request. In order to achieve this goal, the query must contain matching patterns for each property, specifying the property instances that the user is looking for. Listing 2 shows an example of such a query.

Listing 2: The more specific query Q_{all}.

```
PREFIX ao: <http://www.isa.us.es/2009/IntegratedOntology.owl#>
SELECT DISTINCT ?service
WHERE { # ?service has at least three properties...
  ?service ao:hasFeatures ?featuredTerm1.
  ?featuredTerm1 ao:refersTo ?property1.
  ?service ao:hasFeatures ?featuredTerm2.
  ?featuredTerm2 ao:refersTo ?property2.
  ?service ao:hasFeatures ?featuredTerm3.
  ?featuredTerm3 ao:refersTo ?property3.
  FILTER ( # ...and these properties are Car, Price and Availability
    ?property1 =
      <http://www.example.org/2009/DomainOntology.owl#Car> &&
    ?property2 =
      <http://www.example.org/2009/DomainOntology.owl#Price> &&
    ?property3 =
      <http://www.example.org/2009/DomainOntology.owl#Availability>
  )
}
```

Concretely, this query looks for service descriptions in the repository that refer to at least three properties (*i.e.* that matches the six graph patterns of the `WHERE` clause), and afterwards it only returns instances whose matched properties are the specified in the `FILTER` clause (in the example, `Car`, `Price` and `Availability`. Note that each property may be defined in different ontologies, such as WSMO [28], or the proposed by Maximilien *et al.* [25], for instance. Nevertheless, this query can be derived from a user request automatically, so it can be adapted to match the specific properties referenced in each corresponding user request.

This kind of specific query is more restrictive than the former query, shown in Listing 1, returning an even more reduced number of candidate services for the discovery stage, as it is discussed in Sec. 5. However, in certain scenarios, especially when the user request has several optional terms (as with preferences for the ranking process), Q_{some} query may be more appropriate, so there is a trade-off between precision or flexibility that has to be considered in each case. Thus, if user requests contain a high number of mandatory terms (requirements) that have to be fulfilled completely, Q_{all} will return more accurate results than Q_{some}, though a hybrid approach may be also taken if both mandatory and optional terms are included in the user request.

3.2 Automatic Generation of Queries

Queries need to be defined for each user request, because they depend on the structure of that request. On the one hand, query Q_{some}, as shown in Listing 1, only varies on the actual instance of `UserRequest` being used at that precise time. In this case the generation of Q_{some} is straightforward, because its main structure does not vary.

On the other hand, in order to compose query Q_{all}, some analysis have to be done, because concrete properties referred by the user request are used in the `FILTER` clause of the query. Thus, Q_{all} generation depends not only on the structure of the ontology being used by our proposal, but on the concrete instance of `UserRequest` itself, especially on the properties referred by its terms. As a consequence, Q_{all} has to be tailored depending on the corresponding instances managed by each discovery process. However, the generation of query Q_{all} can be done automatically, maintaining the transparency for the user of our proposed SPARQL filtering stage within the discovery process. Algorithm 1 shows the procedure that compose Q_{all} from a given user request.

Algorithm 1: Generation of query Q_{all}.

Input: The user request instance u
Output: A generated SPARQL query Q_{all}

```
1  Q_all = "PREFIX ont:
   <http://www.isa.us.es/2009/AbstractOntology.owl#>"
2  Q_all += "SELECT DISTINCT ?service WHERE {"
3  i = 0
4  foreach Property p in u.hasRequirements.refersTo do
5  |  i++
6  |  Q_all += "?service ont:hasFeatures ?featuredTerm" + i + "."
7  |  Q_all += "?featuredTerm" + i + "ont:refersTo ?property" + i + "."
8  end
9  Q_all += "FILTER ("
10 foreach Property p in u.hasRequirements.refersTo do
11 |  Q_all += "?property" + i + "=" + p.IRI
12 |  i--
13 |  if i > 0 then Q_all += " && "
14 end
15 Q_all += ") }"
```

The presented algorithm is fairly simple, and presents a linear complexity on the number of properties referred by the user request. It iterates over

properties adding the corresponding matching patterns and filter statements to the query. Thus, it does not depend on the number of services registered in the repository, or the complexity of the domain ontologies used, so the overhead of the composition of queries is practically negligible.

4 A WSMO Implementation

Our proposed preprocessing stage can be easily adapted to any SWS framework, so that it can be virtually included within any discovery process. In order to do so, service descriptions and user requests, expressed using a specific SWS framework, need to be mapped to our integrated ontology, or the other way around, so that queries can be used directly as defined in Sec. 3.1, or mapped to the corresponding SWS framework concepts. This mapping can be defined using ontology mediators [5] or ontology mappings [26], for instance. Thus, the concrete work that has to be done for each mapping is summarized in the following steps:

1. Define mappings between concepts of the integrated ontology shown in Fig. 1 and equivalent concepts of the target framework.
2. Rewrite queries discussed in Sec. 3.1 using previously defined mappings.
3. Adapt query generation algorithms from Sec. 3.2 to iterate over the concepts of the target framework.

Our solution can be easily applied to any SWS frameworks, such as WSMO, OWL-S, SAWSDL or WSMO-Lite. In the following, the mappings and queries rewriting to a concrete WSMO implementation are presented.

4.1 Ontology Mapping

In order to to adapt our proposal to the WSMO ontological model, several mappings have to be defined. In Table 1, the correspondences between WSMO entities, and concepts from the integrated ontology used in our proposal (*cf.* Sec. 3) are summarized.

Thus, the `UserRequest` concept from our ontology could be interpreted as a WSMO `Goal`, which describes its *requirements* (`Terms` in our ontology) as a requested `Capability` and some `nonFunctionalProperties` values. Additionally, a `ServiceDescription` are modeled in WSMO as a `webService`, which also *features* or provides some `Capability`, as well as `nonFunctional-Properties`. Both *capabilities* and *non functional properties* instances refer

Table 1: Mappings between WSMO and our integrated model

WSMO ontology	Integrated ontology
`Goal`	`UserRequest`
`webService`	`ServiceDescription`
`Capability` axioms	`Terms`
`nonFunctionalProperties`	`Terms`
`Concept`	`Property`

to several concepts that are interpreted as instances of the `Property` concept of our ontology [13].

Using these mappings, our proposed queries can be expressed or rewrote using WSMO elements, so our preprocessing stage can be executed before any WSMO discovery implementation, provided that the WSML [32] repository to be used allows to access its semantic definitions in RDF. In the following, we discuss how to express and generate both queries so that they can be applied to WSMO services and goals.

In order to implement an application of our proposed filtering stage that relies on WSMO descriptions, queries should be defined in terms of WSMO constructs. Basically, `Terms` in our integrated ontology corresponds to *Capabilities* in WSMO, and user requests are modeled as *Goals*, as described before. However, there is no direct equivalent to the *refersTo* relation in WSMO. Therefore, this relation must be inferred or extracted from descriptions transparently.

Our approach consists on iterating capability descriptions in order to obtain concepts referred by them, and adding these concepts to service descriptions and goals using non-functional properties in WSML. Additionally, other properties, such as quality-of-service properties, that may be already present in the non-functional section of the WSML description, are also included in the values of the newly added *refersTo* property, which should contain all the concepts referred by service descriptions and goals. Algorithm 2 sums up the procedure to add the *refersTo* property to services and goals descriptions.

Esentially, Alg. 2 iterate over the WSML axioms that describe service capabilities for *webServices* and *goals*, and look for instantiations of shared variables that are linked with their concept classes by the `memberOf` construct, which are interpreted as referred properties that have to be added to the *refersTo* set. In addition, it also looks at the non-functional properties and consider their classes as properties referred, consequently including them in the *refersTo* set, too.

Although this process could be costly in terms of execution time if there

Algorithm 2: Obtain concepts referred by WSML descriptions.

```
   Input: A WSML webService description or goal desc
   Output: The webService or goal desc with a refersTo
           nonFunctionalProperty
 1 if desc.nonFunctionalProperties.refersTo = ∅ then
 2 |   refersToSet = ∅
 3 |   foreach Axiom a in desc.capability do
 4 |   |   foreach Variable var in a do
 5 |   |   |   if var is memberOf concept p then
 6 |   |   |   |   refersToSet = refersToSet ∪ p
 7 |   |   |   end
 8 |   |   end
 9 |   end
10 |   foreach nonFunctionalProperty nfp in desc do
11 |   |   refersToSet = refersToSet ∪ nfp
12 |   end
13 |   desc.nonFunctionalProperties.refersTo = refersToSet
14 end
```

were a high number of services, it can be done *off-line* before actual discovery and ranking are performed, at least for service descriptions. Thus, the `refersTo` non-functional property is only obtained and stored in the repository each time a new service description is added or modified, considerably reducing the overhead when discovering and ranking these descriptions, because the algorithm only has to be executed for the user goal at execution time.

4.2 Query Rewriting

Elements based on the WSMO model are described using some variant of WSML language and are commonly serialized in plain text files. Although this serialization is performed using the standard WSML syntax, it is possible to represent WSML constructs using an RDF syntax, so that they can be queried using SPARQL. In order to translate original WSML files to WSML/RDF [6] representation WSMO4RDF library can be used, which is a part of the Ontology Representation and Data Integration Framework (ORDI SG)[4].

[4] `http://www.ontotext.com/ordi/ORDI_SG/Wsmo4rdf.html`

Listing 3: Q_{some} query applied to WSMO.

```
PREFIX wsml: <http://www.wsmo.org/wsml/wsml-syntax#>
PREFIX ont:  <http://www.isa.us.es/2009/AbstractOntology.owl#>
PREFIX part: <http://www.w3.org/.../SimplePartWhole/part.owl#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT DISTINCT ?service
WHERE {
  ?service     rdf:type                wsml:webService.
  ?service     part:hasPart_directly   ?wsCap.
  ?wsCap       ont:refersTo            ?properties.
  :Goal        rdf:type                wsml:goal.
  ?goal        part:hasPart_directly   ?cap.
  ?cap         ont:refersTo            ?properties.
}
```

Thus, the generic query Q_{some} is applied to WSML/RDF representation as shown in Listing 3. In this case, service descriptions and goals have to be differentiated by type checking, because both are related to capabilities using the same `hasPart_directly` relation. Nevertheless, both services and goals are matched as in Listing 1 by properties referred by corresponding capabilities using the *refersTo* relation previously obtained. Similarly, Listing 4 presents the WSMO equivalent query for the case of a more specific query (*cf.* Listing 2 for the version defined after the integrated ontology presented before).

Listing 4: Q_{all} query applied to WSMO.

```
PREFIX wsml: <http://www.wsmo.org/wsml/wsml-syntax#>
PREFIX ao:  <http://www.isa.us.es/2009/AbstractOntology.owl#>
PREFIX part: <http://www.w3.org/.../SimplePartWhole/part.owl#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT DISTINCT ?service
WHERE {
  ?service     rdf:type                wsml:webService.
  ?service     part:hasPart_directly   ?wsCap.
  ?wsCap       ont:refersTo            ?property1.
  ?wsCap       ont:refersTo            ?property2.
  ?wsCap       ont:refersTo            ?property3.

  FILTER (
    ?property1 =
      <http://www.example.org/2009/DomainOntology.owl#Car> &&
    ?property2 =
      <http://www.example.org/2009/DomainOntology.owl#Price> &&
    ?property3 =
      <http://www.example.org/2009/DomainOntology.owl#Availability>
  )

}
```

4.3 Query Execution

In our experimental tool, query execution was implemented in Java using the Jena Semantic Web Framework. First of all, relevant WSML files are parsed and translated to WSML/RDF representation using WSMO4RDF, so that Jena is able to process them and execute our proposed SPARQL queries, conveniently adapted to the WSML/RDF representation. Then, the results from the query execution are used to filter the list of services that take part in the discovery process. The discovery implementation used for the experiments is the lightweight-DL discovery provided by WSMO reference implementation WSMX, which uses Pellet [31] as the DL reasoner.

Algorithm 3: Generation of query Q_{all} for WSMO.

```
    Input: The goal instance g
    Output: A generated SPARQL query Q_all
 1  Q_all = "PREFIX wsml:
    <http://www.wsmo.org/wsml/wsml-syntax#>"
 2  Q_all += "PREFIX ont:
    <http://www.isa.us.es/2009/AbstractOntology.owl#>"
 3  Q_all += "PREFIX part:
    <http://www.w3.org/.../SimplePartWhole/part.owl#>"
 4  Q_all += "PREFIX rdf:
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>"
 5  Q_all += "SELECT DISTINCT ?service WHERE {"
 6  Q_all += "?service rdf:type wsml:webService."
 7  Q_all += "?service part:hasPart_directly ?wsCap."
 8  i = 0
 9  foreach Property p in g.nonFunctionalProperties.refersTo do
10  |   i++
11  |   Q_all += "?wsCap ont:refersTo ?property" + i + "."
12  end
13  Q_all += "FILTER ("
14  foreach Property p in g.nonFunctionalProperties.refersTo do
15  |   Q_all += "?property" + i + "=" + p.IRI
16  |   i--
17  |   if i > 0 then Q_all += " && "
18  end
19  Q_all += ") }"
```

Finally, the automatic generation of queries, specifically Q_{all}, has to be slightly modified to iterate over the values of the `refersTo` non-functional

property of the corresponding goal, which contains the classes of the properties referred by that goal. Algorithm 3 shows how this generation is performed. Note that, as a previous step, `refersTo` property of the goal has to be inferred, as discussed in Alg. 2.

5 Analysis and Evaluation

Our proposed queries have to be thoroughly analyzed, using experimental results, in order to obtain conclusions about their soundness and benefits. Each query have been tested in different situations, measuring both time and size of the resulting repository. In this section, the performed experiments are described, along with an interpretation and discussion of the results for that experimental study, which validates our proposal.

5.1 Defining the Experiments

In order to test the suitability and performance of our proposal, there is a need for a test collection that can be used with the developed tools. Experiments were conducted within a WSMO discovery scenario, as discussed in Sec. 4, so services and user requests have to be described using WSML. However, a suitable, complex enough test collection of WSML descriptions is not available, so we developed a method to generate parametrized test collections, which could be used for performance tests. Thus, several service repositories and related ontologies were created for the experiments. In order to populate these repositories, each service description is generated using concepts from a DL ontology which contains a simple hierarchy of disjoint properties. Then, a goal is similarly generated.

Figure 4 presents the already discussed discovery scenario that our experiments are contextualized in. The identified parameters, shown using boxes, allow to test different situations varying their values. Each parameter have an influence in one of the input artifacts for the studied scenario, namely the service repository, the domain ontology and the user request, represented in Fig. 4 by the dashed arrows. Brief definitions and parameter ranges are enumerated in the following:

- **Repository size (R).** The number of services stored in the repository is a parameter that ranged from 100 to 1,000, with a step of 100, for a total of 10 different values in the conducted experiments.
- **Domain ontology concepts (O).** The number of the domain ontology concepts, which double as instances of `Property` class from Fig.

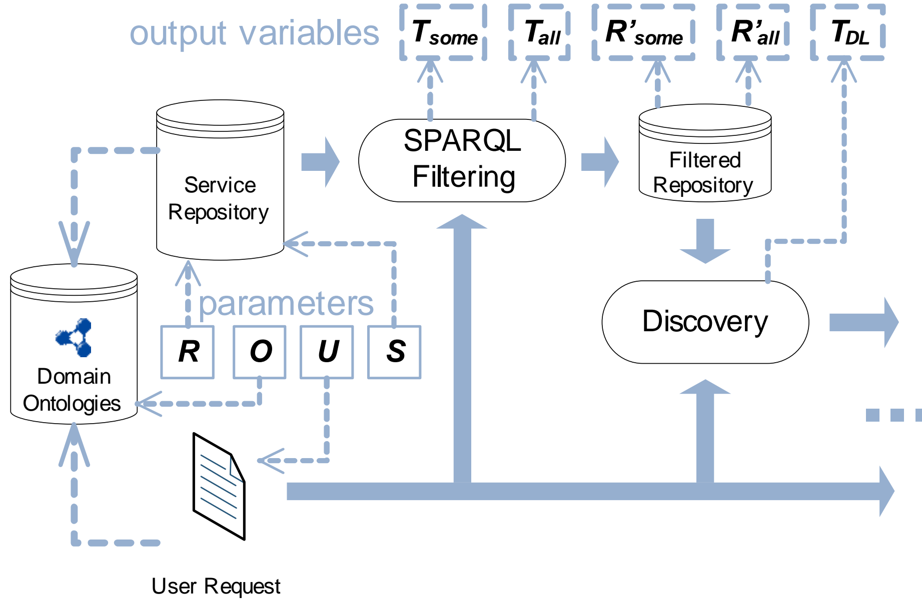

Figure 4: Parameters and output variables of experiments.

1, can be also parametrized. In our experiments, this number ranged from 20 to 100 concepts, incrementing by 20 for each step.

- **User request properties (U).** Another parameter that varies in our tests is the proportion of properties referred by the user request (*i.e.* goals), with respect to the previous parameter, *i.e.* the number of available properties defined in the domain ontology. Five different values were selected for this parameter, ranging from 5 percent to 25 percent. Higher values were not tested because it is unlikely that users define their requests using a high number of concepts, especially as the domain ontology size increases.

- **Service properties (S).** Similarly, the proportion of properties referred by service descriptions is also parametrized, ranging as user request properties from 5% to 25%. As in the previous case, it is unlikely that services manage a lot of concepts, so it is not necessary to test higher values for this parameter.

The ontology representing the domain managed by services is populated with a concrete number of simple concepts (O), depending on each generated repository (R). These concepts are mutually disjoint, with the exception of a simple hierarchy that is randomly created within the ontology: a super-concept is chosen among all the concepts, and then a random number of

concepts are declared as sub-concepts of the former. A scenario with a larger ontology represents a repository which could contain more heterogeneous services, *i.e.* described services offer many different functionalities. Moreover, a larger ontology also means a lower discovery performance, because its complexity increases.

Goals and services are created after the ontology, by selecting one concept that is going to be part of a simple postcondition of the corresponding capability, and, with some additional concepts (up to U and S, respectively), they are directly included in a `refersTo` non-functional property of the element described. Note that concepts from the domain ontology are treated as functional or non-functional property depending on the case, because our proposal does not make any distinction between the nature of referred properties.

Additionally, properties referred by each service and the goal are selected using two different distributions, each one representing a different scenario. On the one hand, in the case of an uniform distribution of service properties, each property defined within the domain ontology has the same probability to appear in service descriptions. Thus, this kind of repository reflects a situation where services may offer many different functionalities. Potentially, each concept from the domain ontology will be referred by the same number of services, *i.e.* the number of different functionalities among services in the repository will be approximately the number of concepts of the ontology.

On the other hand, a power-law distribution is also used to select which properties are referred by service definitions, so most of these definitions refers to a few common properties. Concretely, a Zipf distribution is used because it can be applied to our tested scenarios[1]. This distribution is based on the Zipf's law [36], which interprets that the frequency of any concept is inversely proportional to its rank in the frequency table, *i.e.* the most referred concept will occur twice as often as the second most referred one, which occurs twice as often as the following most frequent concept, and so on. In this case, repositories are fairly homogeneous, *i.e.* they contain many services with the same functionality, and there are few different functionalities. This scenario may be closer to real-world repositories than uniformly-distributed repositories, because in general service repositories are focused on a particular domain. However, larger and more general repositories may fall in between a uniform and a Zipf distribution of properties referred by their service descriptions, so it is worth to test both extremes.

For each experiment, several output variables have been measured. In Fig. 4, the following variables are showcased using dashed boxes, which are connected with dashed arrows to the measured artifacts and processes:

- **Filtering execution times (T_{some} and T_{all}).** The SPARQL filtering stage execution time is measured for each corresponding query, so T_{some} contains the execution time in milliseconds of Q_{some}, and T_{all} measures the same for Q_{all}. These times actually includes both the repository serialization to WSML/RDF files and the query execution itself.
- **Filtered repository size (R'_{some} and R'_{all}).** The filtered repository size is also measured for each query, correspondingly stored in R'_{some} and R'_{all} variables. These variables can be compared to R to analyze to what extent the queries have filtered the original repository.
- **Discovery execution time (T_{DL}).** After filtering, the discovery process is performed and its execution time is stored in the T_{DL} variable. In our experiments, this variable is measured in three different situations: (1) without filtering, (2) filtering with Q_{some}, and (3) filtering with Q_{all}, so that time improvement can be analyzed for each kind of filter.

5.2 Analyzing Tests Results

The implemented testing environment is able to generate several test collections and perform corresponding benchmarking tests at once. These tests were executed in a machine with Windows XP Professional SP3, Java 6, 2.4 GHz CPU and 2 GB of RAM. Furthermore, in order to thoroughly study the benefits of both queries in different situations, tests were conducted varying the four different parameters as described before.

Each combination of parameter values were used to generate two test repositories: one using a uniform distribution to pick up service properties, and the other using a Zipf distribution with 1.0 as its exponent. The whole generation, filtering and discovery process were executed 10 times for each parameter combination and distribution, for a total of 25,000 conducted experiments, measuring each output variable discussed in Sec. 5.1. Experimental results are detailed, analyzed, and discussed in the following.

5.2.1 Execution Time

Figure 5 shows T_{some} and T_{all} varying R, O, and S parameters. [5] As R grows, total execution time of queries linearly increases, while it shows a greater slope as more concepts are present in service descriptions (higher values for

[5] For the sake of clearness, intermediate values for some parameters are omitted in figures throughout this section.

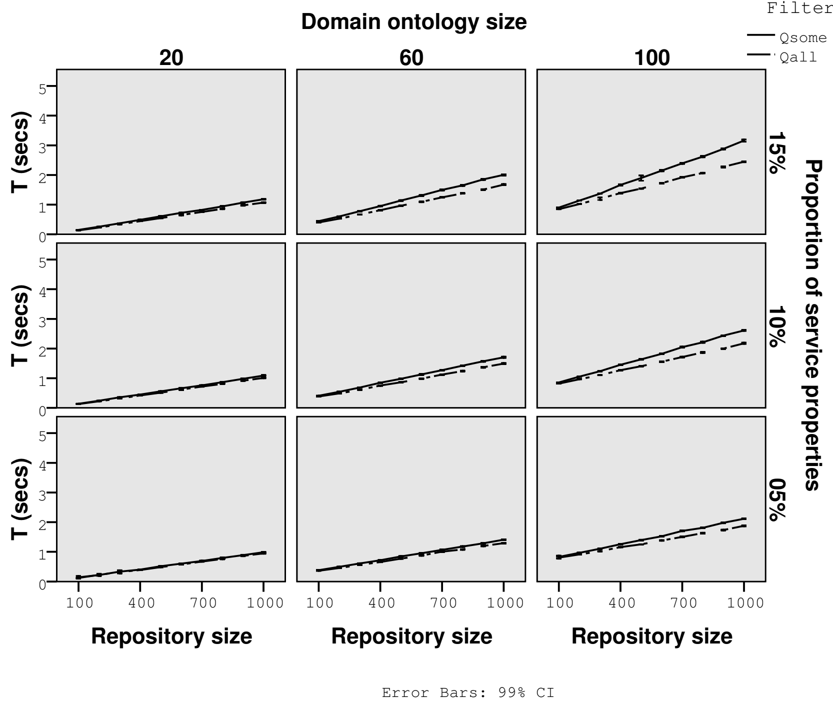

Figure 5: Execution time results.

S also confirm that behavior). O also affects the slope, in addition to a general increase in execution time as the number of concepts in the ontology rises. The more complex relations that can arise by using larger ontologies are the main reason for that behavior. Furthermore, query execution time significantly rises with higher values for both parameters.

In every test case, T_{some} (represented in Fig. 5 with a continuous line) is longer than T_{all} (the dash/dot line in the figure). Furthermore, the difference become larger with higher values for all the three parameters. However, with lower values, both queries tend to have a similar execution time. Note that for execution time, U does not affect at all, because they are not directly referred on any query, so they do not contribute to the complexity of each query execution (*cf.* Sec. 5.3 correlations discussion). Moreover, the distribution used to pick up properties for service descriptions does not affect query neither T_{some} nor T_{all}.

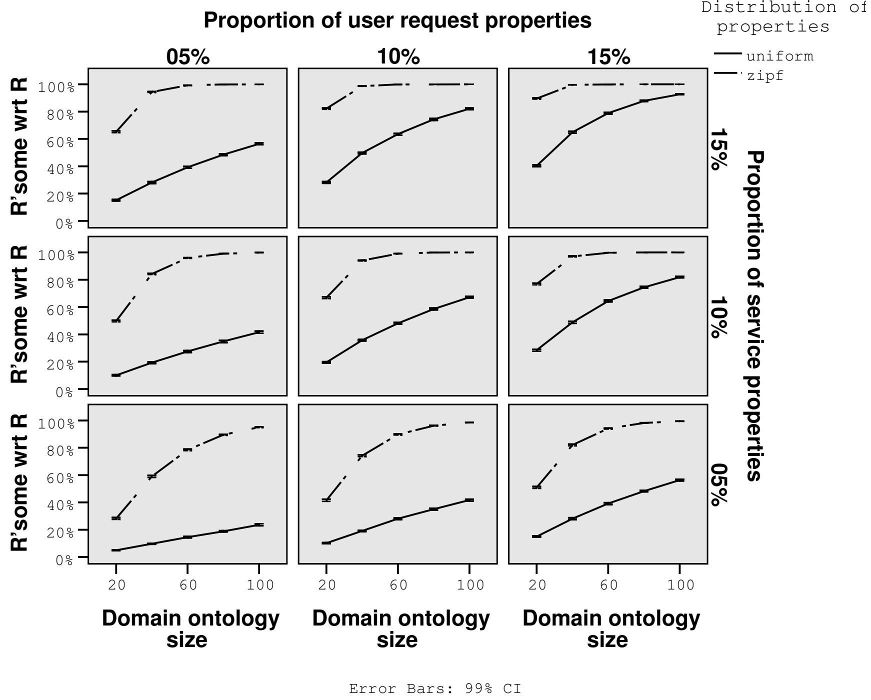

Figure 6: Filtering results for Q_{some}.

5.2.2 Q_{some} Results

In order to evaluate how Q_{some} performs, we measured the proportion of services returned by that query execution with respect to the R value for each experiment. Figure 6 shows how R'_{some} behaves depending on S, U, $\emptyset$, and the distribution of properties used to create each repository. As we are showing resulting repository proportions instead of number of services returned, R does not affect to Q_{some} performance evaluation, as expected.

In general, Q_{some} filters at a higher degree when the repository follows a uniform distribution of properties. In contrast, Zipf-distributed repositories are only filtered to some extent when the proportion of service properties has a low value (5%). As U and S increases, R'_{some} approaches the number of services in the original repository (100%). Furthermore, a higher number of U affects more to the performance decrease of Q_{some}, because the more properties are referred by the user, the more services are likely to use one of them in their descriptions.

Increasing O mainly produces a higher R'_{some}, which means that very large domain ontologies reduce filter performance. In conclusion, best scenarios for

using Q_{some} are those where U is low, and there are not many concepts in the domain ontology (a low O) but they are uniformly distributed among service descriptions.

5.2.3 Q_{all} Results

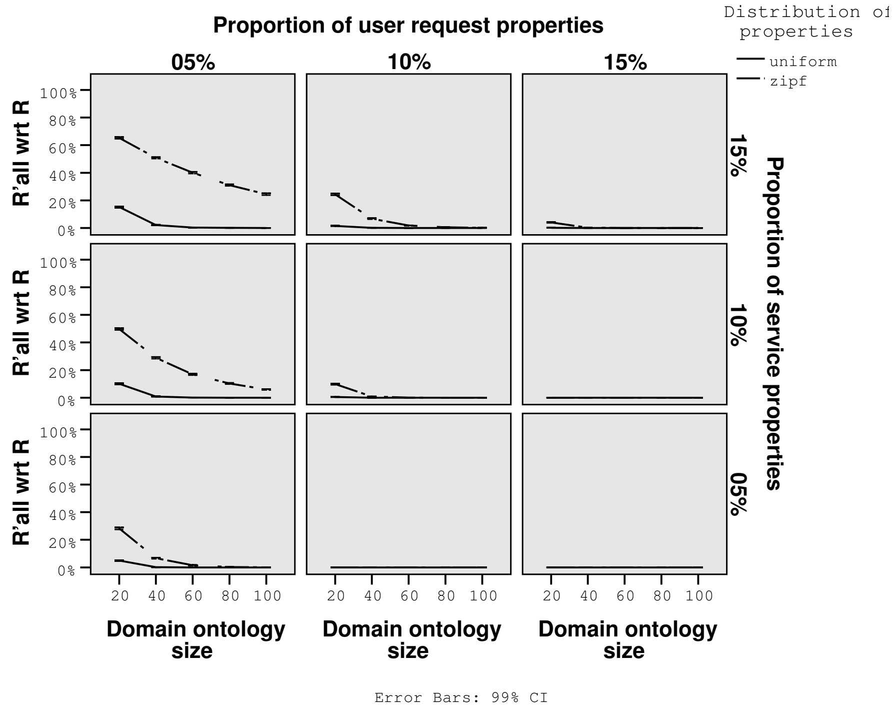

Figure 7: Filtering results for Q_{all}.

As in the previous case, R'_{all} with respect to R was measured for each test collection generated. Results are shown in Fig. 7, where a completely different behavior from Q_{some} is depicted. In this case, as U increases, Q_{all} filters more services (*i.e.* R'_{all} decreases). However, a higher S shows a less strict filter. As the nature of Q_{all} is inclusive, *i.e.* it looks for services that refer to *all* the properties of the user request, more concepts referred by service definitions cause that it is more likely that a service refers to all the properties of the user request.

Although Q_{all} filters many more services than Q_{some}, higher values of O or U actually make Q_{all} to return no results. This is even more noticeable with uniformly-distributed repositories. In this case, there might be some services in the original repository that would fulfill user requests to some

extent. However, these service descriptions are not likely to contain every property of the user request, so R'_{all} tends to 0. Thus, the best situations for filtering repositories using Q_{all} query are those where both U and O have low values.

5.2.4 Discovery Improvement

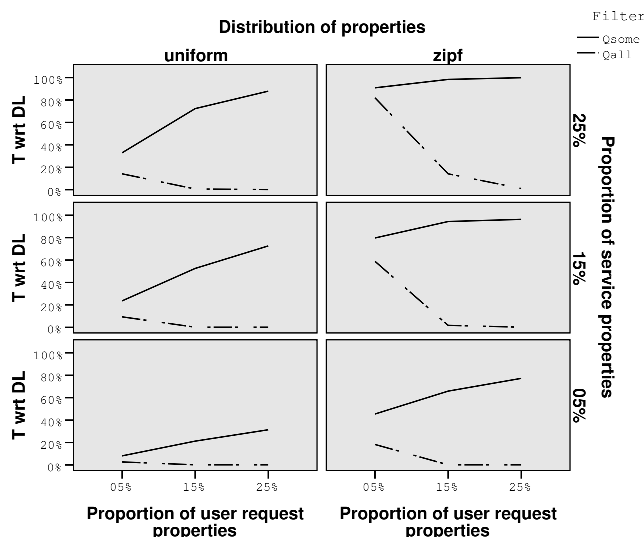

Figure 8: Performance evaluation of discovery mechanisms.

The actual benefits of using the proposed filters in a discovery scenario is shown in Fig. 8. In this figure, T_{DL} values obtained when performing discovery after Q_{some} and Q_{all} filtering are compared with the case where no filtering stage is performed. Due to the DL discovery implementation used in experiments (WSMX lightweight DL discovery), some parameters (R and O) were fixed in order to get results in a reasonable time: a repository size of 600 service descriptions, and 40 ontology concepts were chosen. Furthermore, T_{DL} without any filter applied is constant, so the improvement can be measured depending on the rest of the parameters (namely U and S). Actually, T_{DL} increases linearly by R, but exponentially by O [14].

When the service procurement scenario includes Q_{some} as the choice for filtering the repository (continuous line), the T_{DL} improvement is noticeable, especially with lower values for U and S, though in Zipf-distributed repositories the improve is not so accused. An increase on both U and S produces a higher T_{DL} in this case.

Table 2: Mean and std. deviation, and IC of evaluated variables.

	Uniform			Zipf		
	μ	σ	CI	μ	σ	CI
R'_{some}	63.31%	27.90%	±0.64%	93.99%	12.95%	±0.30%
R'_{all}	0.82%	3.37%	±0.08%	10.85%	21.40%	±0.49%
T_{some}	1.27 s	0.77 s	±0.02 s	1.29 s	0.77 s	±0.02 s
T_{all}	1.06 s	0.57 s	±0.01 s	1.08 s	0.58 s	±0.01 s

Finally, a filtering stage that uses Q_{all} before the DL discovery (dash/-point line) shows a great improvement with respect to plain discovery. However, as shown in Fig. 7, with uniformly-distributed repositories, a low execution time may appear because Q_{all} returns no results, so the afterwards discovery does. However, with lower proportion of service and user request properties, the T_{DL} for this discovery mechanism is only, on average, a 14% of the plain discovery mechanism applied on a uniformly-distributed repository, and a 56% on a Zipf-distributed one.

5.3 Statistical Analysis

A complete statistical analysis have been performed on test runs in order to corroborate our expected results, and to further support the conclusions obtained from figures shown in this section. A summary of this analysis is presented in the following.

Main statistical descriptives are shown in Table 2, for each measured variable in our experiments, with the exception made for T_{DL} because it has not been comprehensively tested.The analysis of these values shows that Q_{some} returns 63.31% of the services on average if their properties are distributed uniformly, though its high standard deviation is caused because of Q_{some} performance depends a lot on the repository parameters. The higher mean value of a Zipf-distributed repository shows that Q_{some} does not filter so much in that case. On the other hand, Q_{all} performs better, meaning that it filters on average more than Q_{some}. Additionally, Q_{all} also filters more when service properties are uniformly distributed (it returns 0.82% of the original repository), in contrast to the case of a Zipf distribution (10.85%), though it could still be considered a good enough result.

Concerning execution time, values are very similar among the cases presented in Table 2, with Q_{some} lasting about 0.21 seconds more than Q_{all}. In this case, we can conclude with a high confidence that, on average, Q_{some}

Table 3: Pearson correlations between evaluated parameters.

	Uniform				Zipf			
	R	O	U	S	R	O	U	S
R'_{some}	0.000	**0.524**	**0.560**	**0.561**	0.000	**0.511**	**0.241**	**0.415**
R'_{all}	-0.003	**-0.313**	**-0.321**	**0.141**	0.000	**-0.202**	**-0.618**	**0.345**
T_{some}	**0.661**	**0.620**	0.019	**0.295**	**0.670**	**0.616**	0.019	**0.291**
T_{all}	**0.692**	**0.642**	0.002	**0.229**	**0.700**	**0.630**	0.006	**0.231**

execution has a penalty time of at most 2.04 seconds ($\mu + \sigma$). However, note that this time includes the RDF serialization needed to use SPARQL with our test repository, so if the repository used allowed to be directly accessed in RDF, that penalty time could be significantly shorter.

Very narrow confidence intervals (CI), computed using a 99% confidence level, shows that, for every variable, mean values can be considered to be robust enough, so they can be used to summarize our experimental results. Thus, if our proposed filters were applied to real scenarios modeled like our test collections, the performance could be predicted by our presented results.

Table 3 shows the two-tailed Pearson-coefficient values calculated between the evaluated variables and the parameters of each test scenario, as described in Sec. 5.1. Values written in bold face mark those *p-values* that give a correlation with a 99% significance. Thus, in the first two rows, R'_{some} and R'_{all} are correlated with O, U and S. However, U is more important (*i.e.* has a higher correlation) for Q_{some} performance in a uniformly-distributed repository than for Q_{all}, though it is the other way around in a Zipf-distributed repository. Execution times for both queries (T_{some} and T_{all}) depends on R, O and S, according to the *p-values* shown in Table 3.

5.4 Discussion

As a general conclusion from the performed tests, the more specific query (Q_{all}) is better suited to filter and reduce the size of the service repository, so it clearly improve the subsequent discovery stage by reducing the search space for matchmaking algorithms. Furthermore, it scales well in every situation, providing even better precision in proportion when the service repository contains a higher number of services.

However, in certain scenarios, where flexibility and soft matching are a concern, the more generic query (Q_{some}) may be more suitable. The higher time penalty must be taken into consideration, both in the filtering and

the discovery stage, because of the less reduced search space, though it still improves the performance of discovery and ranking processes. Thus, there is a trade-off between precision or recall that should be evaluated depending on the concrete scenario. Actually, the current trend in the literature and real-world applications is to achieve better performance and usability, by sacrificing precision, recall, or both[9], so our proposal provides a feasible and efficient solution in this direction.

The main feature of using our proposed queries is that the time penalty is very low, so a hybrid approach may be taken, where both queries are used successively before discovery and ranking processes take part. Firstly, Q_{all} may be executed, and if some results are returned, they are directly injected into the discovery and ranking process. However, if no results are returned by Q_{all}, the more generic Q_{some} query is executed and its results are used in the subsequent discovery and ranking process. This approach is similar to the Best-Matches-Only solution proposed in [18], where if the most accurate results are found (*i.e.* Q_{all} returns results), they are used, but in other case fairly appropriate results (*i.e.* results from Q_{some}) can be useful.

Finally, if the execution time of the filtering stage is analyzed, actual query execution time is significantly lower than the WSML/RDF serialization (approximately 1 millisecond on average), an it is not so affected by the variation of the parameters defined in the experiment. Consequently, our proposed filtering stage could be further optimized by serializing repositories to RDF before performing that filtering.

6 Related Work

There are some proposals that use a Semantic Web query language to perform discovery and ranking of services, though they do not use them to filter repositories. They choose SPARQL as their base language, though some extensions have to be added to fully support these tasks. Thus, Lamparter *et al.* [21] provide an ontology to represent service offers and requests that conforms the foundations for a discovery and selection process performed using rules in SWRL[15] 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, e.g. allowing matching degrees as in [33]. Although this proposal effectively use extended SPARQL to perform discovery, it cannot be applied to generic scenarios because different rules have to be defined for each case.

Another discovery approach that uses SPARQL to actually perform semantic service discovery is proposed by Iqbal *et al.* in [16]. In this case, the authors embed semantic information about services using SAWSDL, which is an extension to add semantics to WSDL descriptions [8]. 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, though authors do not take non-functional properties into account.

Finally, there is another approach, more related to ranking, presented in [29], 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 [13, 12, 35]. 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.

A study of these approaches and an analysis of the suitability of different query languages to perform discovery and ranking is presented in [10]. The conclusions of that study are that the main limitations of current approaches are, on the one hand, their lack of mechanisms to perform complex matchings and reasoning tasks, and on the other hand, their high coupling between description formalisms and algorithms used to evaluate the queries. The generic queries showcased in Sec. 3.1 conform a slightly different approach, because they are used at a previous stage of the discovery process, filtering the candidates and reducing the search space.

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 [2]. 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 [13].

7 Conclusions

Although Semantic Web query languages are not widely used for SWS discovery and selection, 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 standard, providing two different 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.

In this work, comprehensive simulation tests have been run, analyzing the actual reduction of the search space depending on several variables. The conclusions obtained are mainly that our proposal effectively reduce the search space, while it conforms a generic solution, adaptable to any SWS framework that a potential user may want to use. Particularly, an application to WSMO-based services is discussed. Thus, a mixed approach is proposed, where both queries may be executed until reasonable results are returned, reducing the time consumption of matchmaking.

Our proposal of including a (possibly multiple) filtering stage before the discovery and ranking processes has several benefits in addition to the already discussed optimization of discovery and selection processes by reducing the search space. These additional benefits are enumerated 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. Q_{all} query is more specific, but can be generated automatically from a corresponding user request.
- Our proposal do not distinguish between types of properties, *i.e.* both functional and non-functional properties can be used to filter the repository. In consequence, properties being used for both discovery and ranking stages are considered.
- Filters can be applied to any SWS framework because they are based only in domain concepts referred by service descriptions and user requests. An application to the WSMO framework is presented in Sec. 4
- Our proposal is based on the current standard query language for the Semantic Web, *i.e.* SPARQL. Nevertheless, our proposed queries do not use any extension to the standard, so they are compatible with most SPARQL implementations.

References

[1] L.A. Adamic and B.A. Huberman. Zipf's law and the Internet. *Glottometrics*, 3(1):143–150, 2002.

[2] S. Agarwal, M. Junghans, O. Fabre, I. Toma, and J. P. Lorre. D5.3.1 First Service Discovery Prototype. Technical report, SOA4All, 2009.

[3] F. Alkhateeb, J. F. Baget, and J. Euzenat. Constrained Regular Expressions in SPARQL. In *SWWS*, pages 91–99. CSREA Press, 2008.

[4] J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and Semantic Web Query Languages: A Survey. In N. Eisinger and J. Maluszynski, editors, *Reasoning Web*, volume 3564 of *LNCS*, pages 35–133. Springer, 2005.

[5] J. de Bruijn, M. Ehrig, C. Feier, F. Martin-Recuerda, and F. Scharffe. *Ontology Mediation, Merging, and Aligning*. Wiley, 2006.

[6] J. de Bruijn and J. Kopecky. WSML/RDF. Technical report, WSML, 2008.

[7] John Domingue, Dieter Fensel, and Rafael González-Cabero. SOA4All, Enabling the SOA Revolution on a World Wide Scale. In *ICSC*, pages 530–537. IEEE Computer Society, August 2008.

[8] Joel Farrell and Holger Lausen. Semantic Annotations for WSDL and XML Schema. Recommendation, W3C, August 2007.

[9] Dieter Fensel. The Potential and Limitations of Semantics Applied to the Future Internet. In J. Filipe and J. Cordeiro, editors, *WEBIST*, pages 15–15. INSTICC Press, 2009.

[10] J. M. García, C. Rivero, D. Ruiz, and A. Ruiz-Cortés. On Using Semantic Web Query Languages for Semantic Web Services Provisioning. In *SWWS*, pages 67–71. CSREA Press, 2009.

[11] J. M. García, D. Ruiz, A. Ruiz-Cortés, O. Martín-Díaz, and M. Resinas. An hybrid, QoS-aware discovery of semantic web services using constraint programming. In B. Krämer, K.-J. Lin, and P. Narasimhan, editors, *ICSOC*, volume 4749 of *LNCS*, pages 69–80. Springer, 2007.

[12] J. M. García, I. Toma, D. Ruiz, and 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, volume 411 of *CEUR Workshop Proceedings*, 2008.

[13] J.M. Garcia, D. Ruiz, and 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, and T. Tudorache, editors, *ESWC (2)*, volume 6089 of *LNCS*, pages 1–14. Springer, 2010.

[14] Volker Haarslev and Ralf Möller. On the Scalability of Description Logic Instance Retrieval. *Journal of Automated Reasoning*, 41(2):99–142, August 2008.

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

[16] K. Iqbal, M. L. Sbodio, V. Peristeras, and G. Giuliani. Semantic Service Discovery using SAWSDL and SPARQL. In *SKG*, pages 205–212. IEEE Computer Society, December 2008.

[17] C. Kiefer, A. Bernstein, and M. Stocker. The Fundamentals of iSPARQL: A Virtual Triple Approach for Similarity-Based Semantic Web Tasks. In K. Aberer et al., editors, *ISWC/ASWC*, volume 4825 of *LNCS*, pages 295–309. Springer, 2007.

[18] W. Kießling. Foundations of preferences in database systems. In *VLDB*, pages 311—-322. Morgan Kaufmann, 2002.

[19] M. Klusch, B. Fries, and K. Sycara. Automated semantic web service discovery with owls-mx. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 915–922, New York, NY, USA, 2006. ACM.

[20] J. Kopecký and T. Vitvar. WSMO-lite. Technical Report D11v0.2 Working Draft, WSMO, 2008.

[21] S. Lamparter, A. Ankolekar, R. Studer, and S. Grimm. Preference-based selection of highly configurable web services. In *WWW*, pages 1013–1022. ACM Press, 2007.

[22] L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *WWW*, pages 331–339. ACM Press, 2003.

[23] C. Lutz and U. Sattler. A Proposal for Describing Services with DLs. In *Int. Workshop on Description Logics*, 2002.

[24] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, and Others. OWL-S: Semantic Markup for Web Services. Technical Report 1.2, DAML, 2006.

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

[26] N. F. Noy. Semantic integration: a survey of ontology-based approaches. *ACM SIGMOD Record*, 33(4):65–70, December 2004.

[27] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Recommendation, W3C, 2008.

[28] D. Roman, H. Lausen, and U. Keller. Web service modeling ontology (WSMO). Technical Report D2 v1.3 Final Draft, WSMO, 2006.

[29] W. Siberski, J. Z. Pan, and U. Thaden. Querying the Semantic Web with Preferences. In *ISWC*, volume 4273 of *LNCS*, pages 612–624. Springer, 2006.

[30] E. Sirin and B. Parsia. SPARQL-DL: SPARQL Query for OWL-DL. In C. Golbreich, A. Kalyanpur, and B. Parsia, editors, *OWLED*, volume 258 of *CEUR Workshop Proceedings*, 2007.

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

[32] N. Steinmetz and I. Toma. The Web Service Modeling Language WSML. Technical report, WSML, 2008.

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

[34] I. Toma, D. Roman, D. Fensel, B. Sapkota, and J.M. Gomez. A multi-criteria service ranking approach based on non-functional properties rules evaluation. In *ICSOC*, volume 4749 of *LNCS*, pages 435–441. Springer, 2007.

[35] X. Wang, T. Vitvar, M. Kerrigan, and I. Toma. A QoS-Aware Selection Model for Semantic Web Services. In A. Dan and W. Lamersdorf, editors, *ICSOC*, volume 4294 of *LNCS*, pages 390–401. Springer, 2006.

[36] G.K. Zipf. *Selected studies of the principle of relative frequency in language.* Harvard University Press, 1932.