

Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 576463, 15 pages
<http://dx.doi.org/10.1155/2015/576463>



Research Article

Prefiltering Strategy to Improve Performance of Semantic Web Service Discovery

Samira Ghayekhloo and Zeki Bayram

Department of Computer Engineering, Eastern Mediterranean University, Famagusta, Northern Cyprus, Mersin 10, Turkey

Correspondence should be addressed to Samira Ghayekhloo; sghayekhloo@gmail.com

Received 25 May 2015; Revised 12 August 2015; Accepted 14 September 2015

Academic Editor: Wan Fokkink

Copyright © 2015 S. Ghayekhloo and Z. Bayram. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Discovery of semantic Web services is a heavyweight task when the number of Web services or the complexity of ontologies increases. In this paper, we present a new logical discovery framework based on semantic description of the capability of Web services and user goals using F-logic. Our framework tackles the scalability problem and improves discovery performance by adding two prefiltering stages to the discovery engine. The first stage is based on ontology comparison of user request and Web service categories. In the second stage, yet more Web services are eliminated based upon a decomposition and analysis of concept and instance attributes used in Web service capabilities and the requested capabilities of the client, resulting in a much smaller pool of Web services that need to be matched against the client request. Our prefiltering approach is evaluated using a new Web service repository, called WSMO-FL test collection. The recall rate of the filtering process is 100% by design, since no relevant Web services are ever eliminated by the two prefiltering stages, and experimental results show that the precision rate is more than 53%.

1. Introduction

Semantic Web has been a popular topic of research since its introduction by Berners-Lee et al. in 2001 [1]. Based on this idea, automation of many tasks on the Internet is facilitated through the addition of machine understandable semantic information to Web resources. For instance, automatic discovery of Web services based on their functionality or composition of Web services which cannot fulfil the user requests individually becomes possible [2].

In recent years, complexity of conceptual models (e.g., WSMO [3] and OWL-S [4]) for semantic description of Web services as well as the increasing number of advertised services in repositories made the discovery processes of semantic Web services a heavyweight task [5]. In order to deal with the problem of scalability, researchers proposed various methods, such as indexing and caching mechanism [6], preprocessing strategies before actual matching [7, 8], and hybrid matchmakers that combine logic-based and non-logic-based reasoning [9, 10].

This paper presents a new logical framework and two prefiltering strategies to improve the speed and accuracy of

automated Web service discovery. Our discovery framework is based on the WSMO conceptual model for semantically describing user requests (goals), Web services, and domain ontologies. During the discovery process, goal capability descriptions such as inputs, outputs, preconditions, and postconditions (effects) are compared with advertised Web service capability descriptions in order to determine whether they match or not. Logical inference is utilized for matching, which guarantees that the capability requested by the goal is indeed satisfied by the capability of the Web service and also that the Web service has all it needs before it starts execution. Capability reasoning of goal and advertised services relies on ontologies which are used both to describe the services and goals and also to describe the common vocabulary needed by the services and goals.

Our two prefiltering stages are used to eliminate Web services that cannot possibly be successfully matched and reduce the number of Web services which go through the logic-based matching stage. Our first prefiltering stage uses a categorization scheme of Web services. Our second prefiltering stage uses a new technique of extracting attributes and concepts of objects utilized in the goal and the Web

service pre- and postconditions. Our technique can deal with objects and predicates that occur in a logical formula with usage of the conjunction (and) logical operator. We also make use of ontology-based mediation between concepts and attributes, so that two syntactically different symbols may be declared to denote the same thing semantically.

The remainder of this paper is organized as follows. Section 2 presents background information on the WSMO model, F-Logic, FLORA-2, and our logical discovery framework. In Section 3, F-Logic formalization of goal and Web services is described. Section 4 describes our preprocessing algorithms and shows how they work to reduce the run-time processing requirements in the matching phase. Section 5 shows experimental results of utilizing our proposed algorithms, as well as the evaluation of these results. In Section 6 we briefly describe related works on this field, and finally in Section 7 we have the conclusions and future works.

2. Background

Our semantic Web service discovery framework focuses on Web services, goals, ontologies, and mediators that are semantically described based on the WSMO [11] model and using the F-Logic [12] language as implemented in the FLORA-2 [13] logic system. The following subsections briefly introduce WSMO and its core elements, F-Logic, along with its reasoner FLORA-2 and our logical semantic Web service discovery framework.

2.1. WSMO Definition. Web services are semantically described by providing a high level declarative specification of Web service functionality and nonfunctional properties in order to facilitate automatic discovery, composition, and invocation of Web Services. Two prominent models in semantic Web service descriptions are Web Service Modelling Ontology (WSMO) [3] and Web Ontology Language for Services (OWL-S) [4]. There also exist other special purpose languages for the semantic description of Web services, such as DIANE Service Description (DSD) language [14] and Semantic Annotation for WSDL and XML schema (SAWSDL) [15].

WSMO comprises four core elements, namely, ontology, goal, Web service, and mediator. *Ontology* is defined as a formal, explicit specification of a shared conceptualization [16]. In the context of semantic Web services, ontology provides a common vocabulary to denote the types in the form of classes or concepts, properties, and interrelationships of concepts in a domain. *Goal* describes what the requester can provide and what it expects from a Web service. *Web service* description represents different functional and nonfunctional features of a deployed Web service. Finally, *Mediator* handles heterogeneity problems that possibly arise between goals, Web services, and ontologies.

2.2. F-Logic and FLORA-2. F-Logic (frame logic) is a powerful logic language with object modelling capabilities. It is used as a foundation for object-oriented logic programming and knowledge representation. Two popular reasoners of

F-Logic are FLORA-2 and OntoBroker [17]. Our proposed intelligent agent for semantic Web service discovery uses the FLORA-2 reasoning engine. FLORA-2 is considered as a comprehensive object-based knowledge representation and reasoning platform. The implementation of FLORA-2 is based on a set of run-time libraries and a compiler to translate a unified language of F-Logic [12], HiLog [18], and Transaction Logic [19, 20] into tabled Prolog code [13]. Basically, FLORA-2 supports a programming language that is a dialect of F-Logic including numerous extensions that involves a natural way to do meta-programming in the style of HiLog, logical updates in the style of Transaction Logic, and a form of defeasible reasoning described in [21].

2.3. Our Logical Semantic Web Service Discovery Framework.

In general, Web service discovery is the process of finding appropriate Web services with respect to the user request and ranking of discovered services based on user preference. Our discovery framework receives WSMO goal descriptions and WSMO Web service descriptions, all coded in F-Logic, along with related mediators and ontologies as input entities and for each goal returns an ordered list of Web services that can satisfy the needs of the goal.

Figure 1 depicts the architecture of our discovery framework. The framework consists of four stages: (i) the creation and maintenance of goals and Web services along with related domain ontologies and mediators, (ii) prefiltering stages, (iii) matchmaker, and (iv) ranking stage. In the *creation and maintenance* stage, Web service and goal descriptions which are specified based on our modified WSMO model, along with domain ontologies and mediators, are stored in different repositories. In the *prefiltering* stages, for a given goal, advertised Web services are filtered in two steps in order to narrow down the list of Web services that can be possible matches for the goal, the rest of the Web services being eliminated from consideration. In the *matchmaker* stage, the logical matchmaker checks whether each filtered Web service can really execute in a way such that the user goal is achieved. Finally, the *ranking* stage returns lists of matched Web services based on user preference regarding the minimization of some numeric result (e.g., the cost of a flight between two cities).

Focus of this paper is on our novel filtering strategies that we explain in detail in Section 4. Here we briefly describe our logical matchmaker mechanism.

Our logical matchmaker algorithm makes use of preconditions and postconditions of goals and Web services, as well as related domain ontologies and mediators which are imported in service descriptions. The proof commitments (i.e., what must be proven before a match can succeed) required for our logical inference based matching are given as follows:

- (1) $Onts \wedge Mediator \wedge Goal.Pre \models Ws.pre$: the precondition of the Web service ($Ws.Pre$) should be logically entailed by imported ontologies, mediators, and what is provided/guaranteed by the goal precondition ($Goal.Pre$).

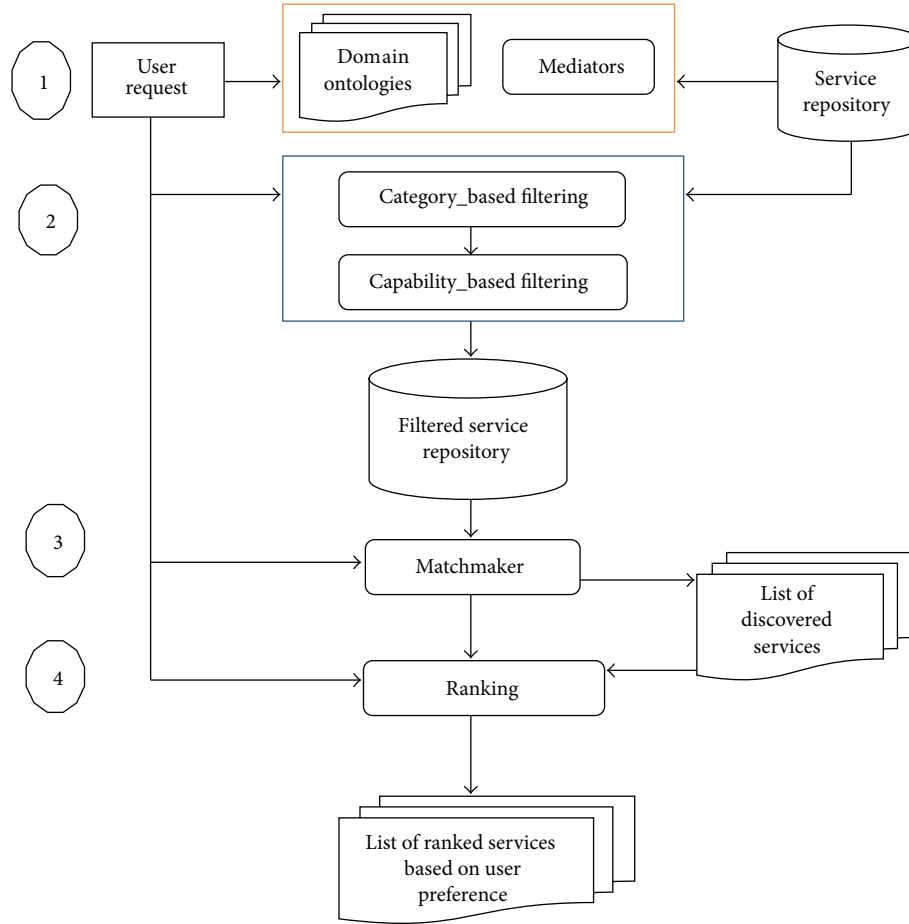


FIGURE 1: Proposed semantic Web service discovery framework including two prefiltering stages.

(2) $Onts \wedge Mediator \wedge Goal.Pre \wedge (Ws.pre \Rightarrow Ws.post) \models Goal.post$: if the postconditions for the Web service were satisfied, then the requirements of the goal should be satisfied. Note how we assume that the execution of the Web service guarantees the validity of the implication in $Ws.pre \Rightarrow Ws.post$.

3. Web Service and Goal Specification in F-Logic

In Listings 1, 2, and 3 we have the meta-level concept definitions of WSMO [11], with several enhancements. Listing 1 contains the Goal concept, instances of which are used to specify a user’s request. It has attributes for nonfunctional properties (such as quality of service, response time, and security), category information (such as transportation, education, and food), ontologies that need to be consulted that contain specific information about a domain (e.g., flight information ontology and geographical information ontology), mediator information (ontologies that deal with discrepancies in terms of defining equivalence classes of terms and synonymous relationship between them), capability needed from the Web service, and the interface demanded from the Web service (i.e., orchestration and choreography).

The *hasCategory* attribute has been newly introduced in our framework in order to allow for filtering based on categories.

The *Service* concept given in Listing 2 is almost identical to the *Goal* concept. Its two differences are as follows: (i) it specifies the *provided* capability instead of the *requested* capability, and (ii) it has an extra attribute called *otherSource* (not in the original WSMO specification) which lists the concepts that should be excluded from consideration in the filtering phase, since objects that are instances of the listed concepts should come from other sources, such as imported ontologies, and are not in the goal.

Listing 3 is the definition of the *Capability* concept. It has attributes for nonfunctional properties, imported ontologies, mediators used, precondition, assumption, postcondition, effect, and optimization. The *optimization* attribute allows the user to specify that the Web service returned by the discovery engine should be optimized with respect to some measure (e.g., price of a flight) and is an enhancement of the original WSMO specification.

When a capability object is part of a goal, precondition is a conjunction of embedded objects in the form of F-logic molecules which specify the information provided by the request to the Web service, and postcondition is a logical expression possibly containing embedded objects, predicates,

```

Goal []
  hasNonFunctionalProperty ⇒ NonFunctionalProperty,
  hasCategory ⇒ Category,
  importsOntology ⇒ Ontology,
  usesMediator ⇒ Mediator,
  requestsCapability ⇒ Capability,
  requestsInterface ⇒ Interface
[]].

```

LISTING 1: Goal concept in our extended version of WSMO.

```

Service []
  hasNonFunctionalProperty ⇒ NonFunctionalProperty,
  hasCategory ⇒ Category,
  importsOntology ⇒ Ontology,
  usesMediator ⇒ Mediator,
  hasCapability ⇒ Capability,
  hasInterface ⇒ Interface,
  otherSource ⇒ OntologyConcept
[]].

```

LISTING 2: Web service concept in our extended version of WSMO.

```

Capability []
  hasNonFunctionalProperty ⇒ NonFunctionalProperty,
  importsOntology ⇒ Ontology,
  usesMediator ⇒ Mediator,
  hasPrecondition ⇒ Axiom,
  hasAssumption ⇒ Axiom,
  hasPostcondition ⇒ Axiom,
  hasEffect ⇒ Axiom,
  optimization ⇒ OptSpecification
[]].

```

LISTING 3: Capability concept in our extended version of WSMO.

conjunction, disjunction, and negation operators. All logic variables in a goal postcondition are implicitly existentially quantified.

However, inside a Web service specification, precondition is a logical expression possibly containing embedded objects in the form of F-Logic molecules, predicates, conjunction, disjunction, and negation operators, where all logic variables are existentially quantified, and postcondition is a conjunction of embedded objects which specify the information provided by the Web service to the requester that is the result of the Web service execution. Note the similarities between the goal postcondition and Web service precondition, as well as the goal precondition and Web service postcondition.

Listings 4 and 5 show the main parts of a goal and a Web service specifications, respectively, among various available types of goals and Web services in our repository.

Listing 4 depicts capability descriptions of a goal instance, which belongs to *AirTransportation* category and describes a request for a flight ticket from Berlin to Istanbul and specifies that the user wants *Sabiha_Gokcen* as a destination airport. The requester also demands flights that have a total cost less than 500\$ for 2 people, and each returned flight must have the minimum cost among all other relevant flights. Note that logic variables start with the “?” symbol.

Listing 5 depicts part of the capability and category descriptions of a Web service instance in our Web service repository. The Web service instance belongs to the *PlaneTransportation* category and provides flight reservation for users who request a flight from one place to another. This Web service asks for source and destination cities, desired departure and arrival date, and number of people who would like to reserve this flight, consults two ontologies containing

```

Goal Instance
“Book a flight from Berlin to Istanbul”
hasCategory → AirTransportation,
requestsCapability → ${goal.I[
  hasPrecondition →
    ${reqFlight[
      originateCity → berlin,
      terminalCity → istanbul,
      departureDate → ?DDate,
      returnDate → ?RDate,
      numberPeople → 2
    ]:RequestTicket
    },
  hasPostcondition →
    (${?BookTicket[
      departureDate → ?DDate,
      returnDate → ?RDate
    ]:Response
    },
    is_equal(?ToAirport, Sabiha.Gokcen),
    less(?TotalCost, 500)
  ),
  optimization →
    ${optObj[optCost → ?TotalCost]}
  ]
}

```

LISTING 4: Part of goal instance specification dealing with the capability desired and the category of the desired service.

flight information (*FlightInfo_ont*) and geographical information (*Geographical_ont*), and returns the list of matching flights ordered according to minimum cost. The precondition needs two objects, one coming from the goal (instance of *RequestFlightTicket*) and one coming from an imported ontology (instance of *Flight*). The predicate *mult* multiplies its first and second parameters and binds its third parameter to the result. It is user-defined.

4. Proposed Two-Phase Prefiltering Mechanism

We propose a solution to tackle the scalability problem by adding two prefiltering stages before the logical matchmaker stage of our discovery framework. We call these two pre-processing algorithms, which offer different filtering levels, *Category-based Filtering* (*Cat_Filt*) and *Capability-based Filtering* (*Cap_Filt*).

Our algorithms that perform preprocessing reduce the input data of service matchmaking, so that the matching process is more streamlined; only logical reasoning about Web services that really matter with respect to the goal is carried out.

These preprocessing steps are performed through the main predicate which is called *%filterMain* in Listing 6.

```

Web service
“Reserve a flight”
hasCategory → PlaneTransportation,
importsOntology →
  {'../FlightInfo_ont.flr',
   '../Geographical_ont.flr'},
hasCapability → ${ws.x[
  hasPrecondition →
    (${?ReqFlight[
      startCity → ?FromCity,
      endCity → ?ToCity,
      departureDate → ?DDate,
      returnDate → ?RDate,
      numberPeople → ?HNumber
    ]:RequestAirplainTicket
    },
    ${?SomeFlight[
      departureDate → ?DDate,
      returnDate → ?RDate,
      fromAirport → ?FromAirport,
      toAirport → ?ToAirport,
      cost → ?Cost
    ]:Flight
    },
    mult(?Cost, ?HNumber, ?TotalCost)),
  hasPostcondition →
    ${response[
      departureDate → ?DDate,
      returnDate → ?RDate,
      fromAirport → ?FromAirport,
      toAirport → ?ToAirport,
      cost → ?TotalCost
    ]:Response}
  ]
}

```

LISTING 5: Part of Web service instance specification dealing with the capability and the category of the provided service.

Output of this predicate is list of goals and their related Web services which are inserted into the knowledge base called *RelatedGoalWsModule* for the subsequent logical matchmaker phase. Listing 6 shows two filtering stages of this predicate.

To facilitate understanding of the code, let us give a brief introduction to object-oriented notation used in FLORA-2. Suppose that *O* and *C* are two objects. *O : C* means that *O* is an instance of *C* (in FLORA-2, an object can simultaneously be a class). *C :: D* means that *C* is a subclass of *D*. Also for user-defined equality, suppose that *O1* and *O2* are different names (called id-terms in FLORA-2 terminology) that are supposed to denote the same object. This fact is stated in FLORA-2 with the notation *O1 := O2*. This facility enables the user to state that two syntactically different (and typically nonunifiable) terms represent the same object and can be used to define synonymy between such terms.

For each goal-Web service pair, the first stage, *Cat_Filt*, uses the *Global_Cat_Ont* to check semantic similarity of the goal category (*Cat_g*) against the Web service category (*Cat_w*).


```

(1) %FilterMain:- ?_Inserted = setof{ ?Ins |
           //-----First stage of filtering- Cat_Filt-----
(2)         ?GoalName[hasCategory → ?GoalCat]@?_GoalModule,
(3)         ?WsName[hasCategory → ?WsCat]@?_WsModule,
(4)         ((?WsCat :=: ?GoalCat); (?WsCat :: ?GoalCat); (?GoalCat :: ?WsCat)),
           //-----Second stage of filtering- Cap_Filt-----
(5)         %Filter_Cap (?GoalName, ?WsName),
(6)         alreadySelected(?WsName, GOAL)@FilteredWsModule,
(7)         alreadySelected(?WsName, WEBSERVICE)@FilteredWsModule,
(8)
(9)         insert{related(?GoalName, ?WsName)}@RelatedGoalWsModule,
(10)        ?Ins=related(?GoalName, ?WsName)
(11)        }.

```

LISTING 6: Prefiltering process containing two filtering stages (lines 2 to 4: *Cat_Filt*; lines 5 to 10: *Cap_Filt*).

```

(1)  $Cat\_Filt(g) = \{w \mid g \text{ has a category specified,}$ 
(2)    $w \in W,$ 
(3)    $Cat_w \in Global\_Cat\_Ont,$ 
(4)    $Cat_g \in Global\_Cat\_Ont,$ 
(5)    $(Cat_w :: Cat_g \text{ or } Cat_g :: Cat_w \text{ or } Cat_w :=: Cat_g)\} \cup$ 
(6)    $\{w \mid w \in W, w \text{ does not have a category specified}\} \cup$ 
(7)    $\{w \mid g \text{ does not have a category specified, } w \in W\}.$ 

```

FORMULA 1: Definition of *Cat_Filt* as a function.

According to Listing 6 line (4), if Cat_g and Cat_w are equal, synonymous, or in an inheritance relationship with one another, the Web service is kept for the next stage; otherwise it is discarded. In the second stage, *Cap_Filt*, first attributes and concepts of objects utilized in the goal and the Web service pre- and postconditions are extracted by our new algorithm (described in Section 4.2). Then, extracted concepts and attributes as well as our ontology-based mediation are used to select Web services which satisfy the following conditions.

(i) Their precondition concepts and attributes are a subset of, equal to or synonymous with the goal precondition concepts and attributes, and (ii) their postconditions' concepts and attributes are a superset of, equal to or synonymous with the goal postcondition concepts and attributes. Each goal is then logically tested for an exact match with only the Web services that survive the two-phase filtering process.

Our current scheme of *Cap_Filt* deals with logical expressions involving only the conjunction operator, positive molecules, and predicates. We shall consider extension of the scheme to deal with any logical expression involving the negation and disjunction operators as well as the conjunction operator in future works.

In the following sections, we describe the two filtering stages in more detail.

4.1. Filtering according to Categories (*Cat_Filt*). The *Cat_Filt* stage filters the original Web services repository according to both specified categories and synonyms defined in the

Global_Cat_Ont ontology. Figure 2 illustrates part of hierarchical structure of our specified domains in *Global_Cat_Ont*, which currently contains the three major categories for *transportation*, *food*, and *education*.

Global_Cat_Ont contains both structural knowledge (i.e., it defines subclass and superclass relationships between concepts of three specified domains) and a dictionary of synonymous concepts.

Formula 1 shows the abstract definition of *Cat_Filt* in the form of a function that takes a goal as a parameter. Here, g and w stand for goal instance and Web service instance, respectively, and W is the Web service repository. The result of the function is the union of three sets: (i) if the goal specifies a category (Cat_g), advertised Web services in the registry which have categories matching the goal's category, (ii) Web services that have no category specified, and (iii) all Web services in case no category is specified for the goal. This definition guarantees that if there is any possibility of a Web service matching the goal, it is never eliminated from consideration in the next phase.

In order to better illustrate previous definitions, consider a scenario where a user is searching for a flight lookup service among the existing Web services described in the repository. Suppose that goal category is *PlaneTransportation* and categories of advertised Web services are *AeroplaneTransportation*, *RailwayTransportation*, and *AirTransportation* consecutively. Result of *Cat_Filt* based on *Global_Cat_Ont* ontology on the described scenario is illustrated in Table 1.

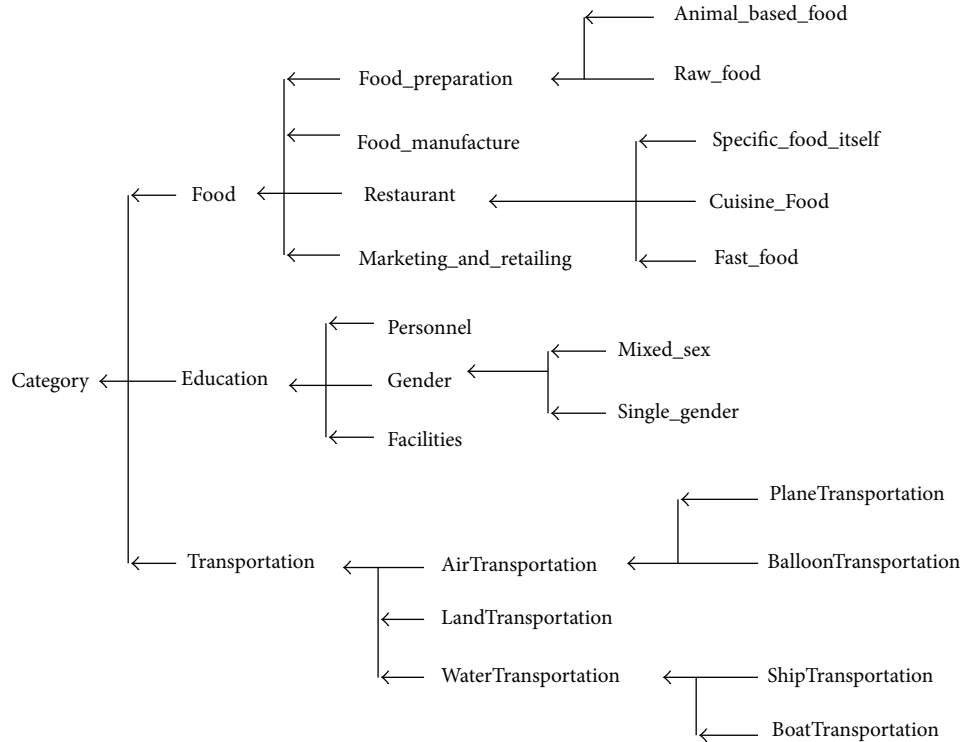


FIGURE 2: Part of the hierarchical structure of our specified domains in the category ontology *Global_Cat_Ont*.

TABLE 1: Results of *Cat_Filt* algorithm on described scenario.

Cat_G	Cat_{WA} AeroplaneTransportation	Cat_{WB} RailwayTransportation	Cat_{WC} AirTransportation
PlaneTransportation	√	×	√

According to the concept relationships definition in *Global_Cat_Ont* ontology, Cat_{WA} , *AeroplaneTransportation*, is a synonym of *PlaneTransportation* and Cat_{WC} , *AirTransportation*, is a subclass of *PlaneTransportation*. Thus, Web services A and C remain as inputs of *Cap_Filt*, and Web service (B) is discarded as irrelevant.

The result of the *Cat_Filt* stage is fed into the second stage *Cap_Filt* in order to eliminate even more Web services that cannot possibly be a match for the given goal.

4.2. Filtering according to Capability Decomposition (*Cap_Filt*). Unlike other proposals mentioned in related works, our *Cap_Filt* algorithm eliminates irrelevant Web services based on checking the attributes and concepts of objects employed in the goal and the Web service pre- and postconditions. This algorithm first extracts attributes and concepts of objects in goal and Web service specifications (it can deal with predicates and objects that occur in a logical formula possibly including the conjunction logical connective) and then analyzes semantic equivalency between extracted attributes and concepts in order to filter out unrelated Web services.

The level of similarity between such parameters is obtained based on their hierarchical relationships inside this

ontology. In this work levels of semantic similarity between parameters are defined as *exact*, *plug-in*, *subsume*, and *fail*. *Exact* means two concepts or two attributes are exactly identical in the same domain ontology. Similarity degree of two concepts or two attributes is *plug-in* only if concept or attribute of goal request is superclass of concept or attribute of the Web service. Degree of two concepts or two attributes is *subsume* only if concept or attribute of goal request is subclass of concept or attribute of the Web service. Finally *fail* degree expresses that there is no semantic-based relationship between two concepts or two attributes.

Also, our work, in order to gain more precise results and tackle the problem that two concepts or two attributes which are going to be investigated may not be equal syntactically, uses WordNet [28], a dictionary of synonymous words. Thus, synonym similarity between the goal and Web service parameters in the *Cap_Filt* algorithm is calculated by making use of the WordNet [29] online synonym dictionary.

As an illustration of the above definition, consider the instances of a matched goal and Web service defined in Section 3. List of concepts and attributes of our goal and Web service preconditions are shown in Table 2.

```

(1) %Filter_Cap (?GoalName, ?WsName):-
           //-----Pre-Condition-----
(2)     ?GoalName[requestsCapability → ?GCap]@?GoalModule,
(3)     ?GCap ~ ${?_GCapability[
(4)     hasPrecondition → ?GoalPre, hasPostcondition → ?GoalPost]}@?GoalModule,
(5)     ?WsName[hasCapability → ?Wcap]@?WsModule,
(6)     ?Wcap ~ ${?_WSCapability[
(7)     hasPrecondition → ?WsPre, hasPostcondition → ?WsPost]}@?WsModule,
(8)     %FindGoalOrWsAtt (?GoalPre, GoalWsAttModule),
(9)     %DC (?WsPre, ?Ws_Pre_Att_Cnp),
(10)    %Check_Att_Cnp (?WsName, ?Ws_Pre_Att_Cnp, WEBSERVICE),
           //-----Post-Condition-----
(11)    deleteall{?_A[?_B → ?_V]:?_C @GoalWsAttModule},
(12)    %FindGoalOrWsAtt (?WsPost, GoalWsAttModule),
(13)    %DC (?GoalPost, ?Goal_Post_Att_Cnp),
(14)    %Check_Att_Cnp (?WsName, ?Goal_Post_Att_Cnp, GOAL).

```

LISTING 7: Critical parts of the %filter_Cap predicate.

TABLE 2: List of concepts and attributes obtained as a result of %DC/2 predicate.

Name	[(ConceptName, [List of attributes])]
Goal.Pre	[[RequestTicket, [originateCity, terminalCity, departureDate, returnDate, numberPeople]]].
Ws.Pre	[[RequestAirplainTicket, [startCity, endCity, departureDate, returnDate, numberPeople], (Flight, [departureDate, returnDate, fromAirport, toAirport, cost])]].

As it is shown in Table 2, *originateCity* and *startCity* are the first attribute of goal precondition and Web service precondition, respectively. Although the spelling of these two attributes is different and they may not have any relation in domain ontology, they have the identical meaning. Our approach tackles this problem and considers the attributes similar to each other through the dictionary of synonymous words. We employ both semantic and synonymous equivalency of pre- and postconditions.

Listing 7 depicts the critical parts of the %Filter_Cap predicate. The filtering, which is based on concepts and attributes of objects in the capability specification of the Web service and goal, is carried out in the following manner:

- (1) Lines (2) to (7) read goal and Web service pre- and postconditions from their individual's modules.
- (2) As the process of checking semantic and synonymous similarity of goal and Web service specifications are done in knowledge base module (*GoalWsAttModule*), in Listing 7 line (8), attributes and concepts of goal preconditions are inserted into *GoalWsAttModule* through %FindGoalOrWsAtt/2 predicate.
- (3) Attributes and concepts of Web service preconditions are extracted via the %DC/2 predicate.

This transactional predicate decomposes a Web service precondition, and then extracted concepts and attributes are stored in different lists. Listing 7 line (9) depicts the calling of this predicate with parameters *?WsPre* (bound to a precondition) and *?Ws_Pre_Att_Cnp* (a free variable). As a result of the call, *?Ws_Pre_Att_Cnp* gets bound to the list of concepts and their corresponding attributes in the Web service precondition.

- (4) Finally, line (10) depicts %Check_Att_Cnp predicate that implements Algorithm 1. It compares concepts and attributes related to goal preconditions with concepts and attributes associated with Web service preconditions based on semantic equivalency between them. Output is the name of related Web services whose concepts and attributes exist in requested goal, as explained below. Web service names that pass through this level of filtering are stored in the knowledge base called *FilteredWsModule*.

Comparison of goal and Web service postconditions is similar to that of the preconditions, except for some changes in predicates' parameters.

- (5) As it is shown in line (11) of Listing 7, contents of knowledge base *GoalWsAttModule* which already consisted of goal precondition's attributes and concepts are erased in order to be replaced with the new data.
- (6) Attributes and concepts of Web service postconditions are moved into *GoalWsAttModule* by %FindGoalOrWsAtt predicate in line (12) of Listing 7.
- (7) Attributes and concepts of goal postconditions are extracted via %DC/2 predicate, and the results are stored in *?Goal_Post_Att_Cnp* variable as it is shown in line (13).


```

Input: List1 of the form [(Concept, [ListOfAttributes]), ...]
        (extracted from either Goal.Pre or Ws.Post)
        List2 of the form [(Concept, [ListOfAttributes]), ...]
        (extracted from either Ws.Pre or Goal.Post)
        Tag (either GOAL or WEBSERVICE)
Output: (WsName, Tag) (as insertion into module FilteredWsModule)
(1)   Let equiv(A, B) = (A :: B) or (B :: A) or (A :=: B)
(2)   PossibleMatch ← True
(3)   for all (Concept2, [ListOfAttributes2]) ∈ List2 do
(4)       if not ∃ (Concept1, [ListOfAttributes1]) ∈ List1 (
(5)           equiv(Concept1, Concept2) and
(6)           ∀ (attribute2 ∈ [ListOfAttributes2]) (
(7)               ∃ (attribute1 ∈ [ListOfAttributes1]) (
(8)                   equiv(attribute1, attribute2)
(9)               )
(10)            )
(11)        )
(12)   then
(13)       PossibleMatch ← False
(14)   end if
(15) end for
(16) if
(17)     PossibleMatch = True
(18) then
(19)     Insert (WsName, Tag) into FilteredWsModule
(20) end if

```

ALGORITHM 1: Filtering by comparing concepts and attributes.

(8) In line (14), similar to line (10), `%Check_Att_Cnp` predicate implements Algorithm 1. But this time it checks concepts and attributes related to Web service postconditions with concepts and attributes associated with goal postconditions based on semantic equivalency between them.

If all these checks succeed, then the pair of goal and its related Web services is inserted into the knowledge base so that full checking of the proof commitments can be carried out in the next stage.

5. Experimental Results and Discussions

Proper test collection is needed in order to evaluate the suitability and performance of service discovery frameworks. Currently, two de facto test collections are OWLS-TC [30] and SAWSDL-TC [31]. OWLS-TC, which mainly considers input and output parameters, is applicable for approaches that deal with OWL-S Web services descriptions, and approaches which employ SAWSDL Web service descriptions use the SAWSDL-TC test collection.

The latest version of OWLS-TC at this time is version 4 [32]; it consists of 1083 Web services and 42 queries which are written in the OWL-S language. Unfortunately, the majority of Web services in OWLS-TC are only partially described, being based on input and output types. Only in the last version (version 4), 160 Web services contain preconditions and postconditions (effects) which are described in different

languages such as SWRL (SWRL: A Semantic Web Rule Language, [33]) and PDDL (International Planning Competition, [34]).

The SAWSDL-TC test collection is established to support the performance appraisal of SAWSDL matchmakers. The latest version of SAWSDL-TC, at this time, is version 3; it consists of 1080 semantic Web services and 42 queries which are described in the SAWSDL language. However, descriptions of Web services and queries are only based on input and output parameters [35].

The majority of approaches (such as [7, 9, 23–26]) that work in our field and are mentioned in related works evaluated efficiency and accuracy of their works based on OWLS-TC version 3 test collection. Among all related works, authors of [36] evaluated their proposal based on last version of OWLS-TC test collection, but only input and output parameters are considered for evaluation of their work.

Therefore, due to unavailability of an appropriate test collection that covers main functional descriptions of Web services such as preconditions and postconditions, as well as a categorization scheme of Web services, we generated our own test collection of Web service/goal specifications and used this test collection to measure the gains in efficiency obtained by employing our proposed prefiltering strategy. We called our test collection *WSMO-FL* [37].

WSMO-FL contains three different domains, namely, transportation, food, and education, with 250 different F-Logic Web services descriptions, 6 different F-Logic goals descriptions, 22 concepts, and 1225 instances.

In this section, in order to validate our proposal, we performed experimental evaluations described and the results of that experimental study. For analysis, each test has been run 20 times performed on a machine with Windows 7, a 2.93 GHz Intel processor, and 4.00 GB of RAM.

In order to determine the actual improvements of our proposed prefiltering stages, we measured several indicators: (i) the average response time of our semantic Web service matchmaker with filtering (*Filt_Disc*) and without filtering (*Naive_Disc*), (ii) the number of Web services that have been effectively eliminated from the initial pool of available Web services at each prefiltering stage, (iii) precision, and (iv) recall. Due to the fact that our filtering stages never eliminate any Web service from consideration unless they are guaranteed to fail at the logical matching stage, it is no surprise that recall rate is always 100%.

The results of the performed tests for the goal are given in Table 3, showing the mean and median of the time it took to match the goal against varying number of Web services. The statistical measures (mean, median) were computed over 20 runs which yielded the raw data. Timing data was recorded for the two cases of matchmaker using the prefiltering phases *Filt_Disc* and matchmaker using no filtering at all *Naive_Disc*.

Figure 3 graphically depicts the same information as a line chart. It can be seen that when using *Filt_Disc*, the average response time is in range of 0.08 to 0.062 seconds, while for the same goal and Web services in *Naive_Disc* it dramatically increases and is in range of 0.08 to 17.5 seconds.

Figure 4 depicts the dramatic number of reductions in the number of Web services that remain after each prefiltering phase. The data has been collected by matching six different goals and varying number of Web services for each goal. The chart indicates that *Cap_Filt* through the semantic equivalency of goal and Web service concepts and attributes does a very good job of eliminating irrelevant Web services, given that most of the remaining Web services after its application pass the *Cat_Filt* stage.

To analyse the accuracy of our prefiltering stages, Table 4 gives the precision and recall values of the combined prefiltering stages for the same set of data obtained by running 6 requested goals against 250 Web services in the repository.

Precision is percentage of the retrieved Web services that are actually relevant. In our context, “retrieved Web services” means the Web services that survived the two-stage elimination process, and a Web service is “relevant” to a goal if the logical matchmaker says so. With these definitions, precision is formalized as follows [38]:

$$\text{Precision} = \frac{\text{Number of Relevant Web services in the retrieved set}}{\text{Number of Retrieved Web services}}. \quad (1)$$

Recall is the portion of the relevant Web services that are successfully retrieved. It is formalized as follows [38]:

$$\text{Recall} = \frac{\text{Number of relevant Web services in the retrieved set}}{\text{Number of all relevant Web services in the repository}}. \quad (2)$$

TABLE 3: Statistical comparison of *Filt_Disc* and *Naive_Disc*.

Number of WSs	Engine	Mean time (sec)	Median time (sec)
5	<i>Filt_Disc</i>	0.08	0.09
	<i>Naive_Disc</i>	0.08	0.08
50	<i>Filt_Disc</i>	0.27	0.27
	<i>Naive_Disc</i>	3.93	4.02
150	<i>Filt_Disc</i>	0.45	0.51
	<i>Naive_Disc</i>	12.39	12.39
250	<i>Filt_Disc</i>	0.62	0.60
	<i>Naive_Disc</i>	17.43	17.33

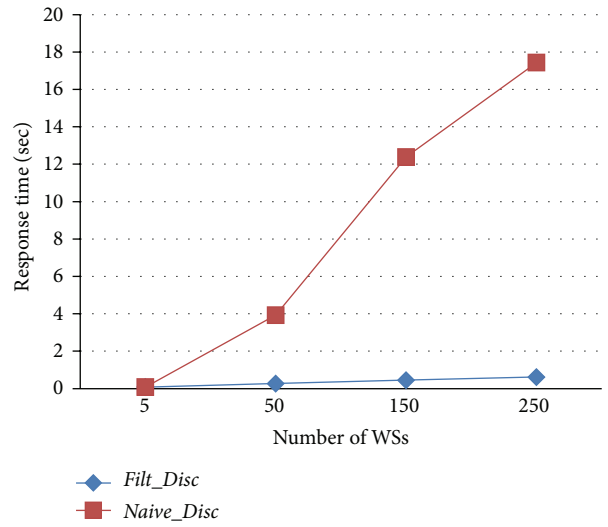


FIGURE 3: Comparison of *Filt_Disc* and *Naive_Disc*.

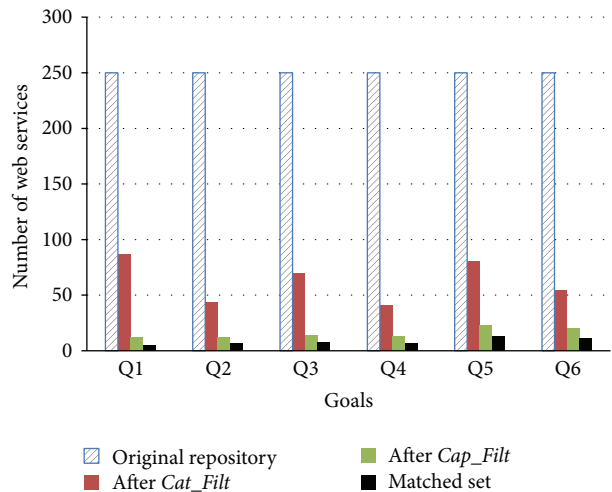


FIGURE 4: Effectiveness of the two prefiltering stages in eliminating irrelevant Web services.

As shown in Table 4, average precision for all request queries is 53.72% which to some extent can be considered a good precision rate. It means that around 55% of retrieved Web services are exactly matched with the requested goal and the others, around 45%, are irrelevant. However, the average recall of queries has the highest possible rate, 100%. With this

TABLE 4: Precision and recall of combined prefiltering stages in each requested goal.

Goal	Q1	Q2	Q3	Q4	Q5	Q6	Average
Precision	41.6%	58.3%	57.1%	53.8%	56.5%	55%	53.72%
Recall	100%	100%	100%	100%	100%	100%	100%

100% recall rate, all the relevant Web services in Web service repository are retrieved through the proposed prefiltering stages, an important feature that sets out filtering strategy apart from all the other proposals.

Figure 5 graphically shows the precision and recall rate of each requested goal together with the average precision line. The chart illustrates that precision rate of all requests except the first one (Q1) is higher than the average. Low precision rate of Q1 indicates that there exist many Web services in the repository whose attributes and concepts are semantically similar to the concepts' name and attributes' name of requested goal; however, the value of Web service attributes defined in ontologies does not match the requested value of goal attributes. Such Web services fail in the actual logical matching procedure.

As we explained before, the reason of top average recall rate is that all relevant Web services are retrieved by *Cat_Filt* and *Cap_Filt* algorithms, which is another way of saying that in the prefiltering stages, we only eliminate Web services that the matcher would definitely reject. The reason for the not-so-high average precision rate in prefiltering stages is that although the retrieved Web services are similar to requested goal due to semantic and synonymous equivalency of their concepts and attributes in domain ontology, maybe some Web services that will eventually be rejected pass through the filters. However, in a real world situation where thousands of categories exist and Web services, as well as goals, are annotated by categories, precision would be expected to rise significantly, since the majority of Web services under consideration would be eliminated by the first stage of filtering (*Cat_Filt*).

Since our framework evaluation is based on our newly generated test collection, WSMO-FL, a comparison between the accuracy and performance of our work and the other available works in the literature would not be very informative. However, an average of 100% for recall and 53.72% for precision indicates a satisfactory accuracy of this work. It should be pointed out that this accuracy was observed in a more complex condition of goals and Web services due to pre- and postcondition parameters. The other studies mentioned in related works did not consider these many complexities in their goals and Web services.

6. Overview of Related Works

Recently, although a wealth of insightful efforts have proposed different solutions to improve the semantic Web discovery process and their related scalability issues, we could not find any work that addresses the performance challenge of discovery process in a similar way to our work. In this section, we discuss proposals related to this field and analyse

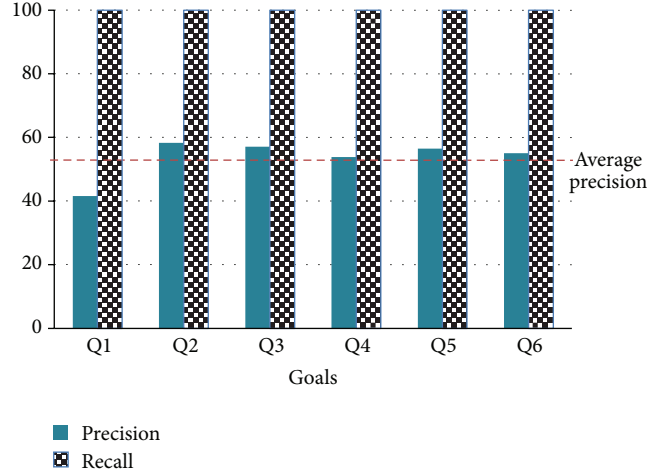


FIGURE 5: Precision and recall of each requested goal along with the average precision line.

their relationship with our solution, and their advantages are compared to our approach.

Table 5 compares our work with the related works based on several dimensions with respect to semantic Web service discovery improvement. These dimensions are preprocessing, discovery method, parameters, and frameworks. First three dimensions are further subdivided into subdimensions.

Preprocessing is subdivided into nonfunctional properties (NFPs) and functional properties (FPs). *NFP* here stands for methods of adding some NFP elements to the Web service and goal descriptions (e.g., categorization of each advertised/requested Web service at design time). *FP* stands for methods to compare functional parameters of goal and Web services. One more level of subdivision used in preprocessing factor is taxonomy (TX) (i.e., relationship between two concepts/attributes is described by using a hierarchical diagram), synonymity (SY) (i.e., syntactically two concepts/attributes are different but they have the same or identical meaning), and syntax (ST) (i.e., no synonymous or hierarchical relationships exist between two concepts/attributes and they are compared based on similarity of their string) similarity method measurements for each mentioned NFP and FP.

Discovery method represents which kinds of service matchmakers are used in the approaches: logic (LOG), non-logic (NLOG), or hybrid method (HY) which is combination of both logic and nonlogic methods.

Parameters demonstrates degree of completeness of a research (whether it uses the major functional parameters of goal and Web services or not). Major functional parameters of goal and Web services in OWL-S and WSMO models are input (I), output (O), precondition (PRE), and Postcondition or effect (POS/EFF).

We summarise the result of the comparative study of Web service discovery approaches in Table 5, where each row represents an approach and the columns stand for main dimensions in Web service discovery improvement. The symbol “√” is used to denote that the specified approach supports the corresponding dimension, and “—” means that it does not.

TABLE 5: Comparison of this work with related works.

App.	Preprocessing				Discovery method				Parameters				Framework
	NFP		FP		ST	LOG	NLOG	HY	I	O	PRE	POS/EFF	
TX	SY	TX	SY										
[9]	—	—	—	—	—	—	—	√	√	√	—	—	OWL-S
[10]	—	—	—	—	—	—	—	√	√	√	—	—	WSMO
[6, 22]	—	—	—	—	—	√	—	—	√	√	√	√	WSMO
[23]	—	—	—	—	—	√	—	—	√	√	√	√	OWL-S
[24]	—	—	—	—	—	—	—	—	√	√	√	—	OWL-S
[25]	√	—	—	—	—	√	—	—	√	√	—	—	OWL-S
[26]	—	√	—	—	—	—	—	—	√	√	—	—	OWL-S
[7]	—	—	√	—	—	—	—	—	√	√	—	—	OWL-S
[8]	—	—	√	—	—	—	—	—	√	√	—	—	WSMO
[27]	—	—	—	—	√	√	—	—	√	√	√	√	WSMO
Our work	√	√	√	√	—	√	—	—	√	√	√	√	WSMO

Approach name (APP). Hybrid method (HY). Taxonomy (TX). Synonymity (SY). Syntax (ST). Logic (LOG). Functional properties (FPs). Output (O). Input (I). Nonlogic (NLOG). Nonfunctional properties (NFPs). Precondition (PRE). Postcondition or effect (POS/EFF).

In order to highlight the advantages of our work with respect to the prior researches, we classified the related works into two groups: approaches that optimize semantic Web service discovery through (a) improvement of matchmakers and (b) application of prefiltering mechanism before actual matchmakers. The former discusses the related works where the only focus is to improve the performance of their matchmaker engines by employing various methods. The latter tries to reduce the size of original repository and the filtered repository is used as input of actual matchmaker.

(a) *Approaches That Improve the Matchmaker Engine.* Regarding the need to improve the discovery process and make it more scalable, some approaches attempt to improve the performance of the matchmaker engine without introducing any extra preprocessing stages.

Klusck et al. [9] implemented a hybrid matchmaker consisting of both approximated Information Retrieval (IR) matching, such as syntactic similarity technique, and OWL-DL logical reasoner to discover semantic Web services. Authors used four variants to calculate the text similarity of parameters, called *cosine*, *loss-of-information*, *extended Jacquard*, and *Jensen-Shannon*. In *OWLS-MX*, the logical reasoner only considers degree of semantic similarity between input and output parameters of OWL-S advertised/requested services and available concepts in the specified domain ontology. Later they developed their system to support WSMO services, called *WSMO-MX* [10]. Their comprehensive evaluations demonstrate that both approaches presented high precision in the S3 contest [39]. However, shortcomings of their solution are as follows. (i) They are time consuming because of high calculation costs related with both logic-based matching and text-based similarity matching; (ii) they retrieve Web services which are not related to the request.

The Klusck et al. approach can be improved if they utilize our preprocessing strategies on top of their actual matchmakers. For instance, by applying our prefiltering

stages before the hybrid matchmakers, especially on the logic-based matchmaker, they can potentially decrease the size of the initial Web service repository and consequently improve the overall performance of matchmaker.

Stollberg et al. in [6, 22] improved the matching process by implementing a caching mechanism that decreases the size of search space and reduces the matchmaker operations. The presented cache uses a *Semantic Discovery Caching* (SDC) graph that stores connections between client requests described as WSMO goal templates and their relevant Web services. Thus, when a goal instance is received, first, the system compares the goal instance with cached templates with respect to semantic similarity and if there is a match, merely the relevant Web services that are stored in the SDC graph are used for subsequent discovery.

Authors of [6, 22] claim that they presented a standard approach where both advertised and requested functionalities are formally expressed in terms of preconditions and effects (postcondition). Also they used first-order logic as the specification language for formal description of these terms. Since our proposal also has been established in the spirit of WSMO framework and developed to work on goals and Web services capability which consist of inputs, outputs, and pre- and postcondition, proposed caching approach can be completed when our prefiltering mechanisms are implemented before creating the caching graph. Thus, the number of relevant Web services which are stored in graph can be possibly decreased.

Authors of [23] introduced SPARQL as a language to describe the preconditions and postconditions of OWL-S Web services as well as user requests. They implemented a matchmaker that works through agents called *SPARQLent* (SPARQL agent). In this approach, a complete discovery solution of their algorithm is discussed and shows how SPARQL queries are used to modify and query the agent's knowledge base. Finally, they evaluated their proposal against *OWLS-MX* via *SME₂* test tool [40].

Although the method offered in [23] is based on pre- and postconditions of Web services and goals, their evaluation is performed based on OWLS-TC V3, while presented Web services descriptions in this test collection are without pre- and postconditions. In addition, our prefiltering stages could be also useful in avoiding SPARQL agent to load all available Web services on the repository and as a result cause further improvement to their agent performance.

Amorim et al. discuss a hybrid matchmaker called *OWL-S Discovery*. It is a combination of semantic filters based on input and output parameters of requested/advertised services and analysing each neighbour relationship in domain ontology [24]. Authors employed five levels of semantic similarity between input and output parameters, namely, exact, plug-in, subsume, fail, and sibling. Also, in order to analyse each neighbour relationship in the concepts, they use a dictionary to classify the concepts. Based on this dictionary, concepts are either identical or synonymous or neither synonymous nor identical, as in our work. At the end they compare their work with Paolucci's approach [41] and the hybrid algorithm OWLS-MX through OWLS-TC v3 test collection. Our proposal also can be applied to the top of *OWL-S Discovery* to further improve discovery processes. But our work uses a more expressive model to describe user requests and Web services descriptions as they contain pre- and postconditions.

(b) *Approaches Using Preprocessing Mechanisms*. These approaches make use of preprocessing mechanisms that help optimization of automated Web service discovery by narrowing down the set of existing Web services in the repository that will be considered by the service matchmaker. Preprocessing mechanisms are further subdivided into two categories: (1) preprocessing mechanisms based on categorization schemes of NFPs and (2) preprocessing mechanisms based on semantic similarity of FPs.

(1) *Prefiltering Based on Categorization Schemes of NFPs*. Most of the efforts related to prefiltering techniques follow the classification methods: either exploit hierarchical categorization schemes of Web services on the basis of domain ontologies [25] or use dictionary of synonymous words [26]. The filtering process is separate from matchmaker, so the results of this prefiltering stage are then inspected through any actual process of service matchmaking. The majority of the mentioned proposals adapted OWLS-TC v3 test collection by adding one element to the request and Web service NFPs that refer to service application domain.

Authors of [25, 26] implemented their categorization proposals on OWL-S Web services and verified them with respect to the OWLS-TC v3 data set. However, OWL-S service description in this test collection does not contain any information about service's application domain. Thus, in order to overcome the limitation of current OWL-S service profile elements both approaches added one NFP to the OWL-S service profile. Although both used the same idea, their solution is different. In [25] the defined category concept of the service request is compared with the defined category concept of advertised Web services via hierarchical

categorization scheme in global category ontology. A Web service is eliminated if it has no category relationship with the request category. However, in [26] equivalency of requested and advertised Web services category concepts are computed via their relationship in the WordNet [28] dictionary of synonyms words. This approach is lacking in its own matchmaker (i.e., evaluation is done via OWLS-MX matchmaker).

Although the idea of our first filtering stage is similar to the mentioned proposals, it has the following novelties. (i) Our proposed *Cat_filt* stage enriches the WSMO framework by adding an attribute called *hasCategory* to both goal and Web service descriptions. (ii) In order to increase the accuracy and performance of our categorization schemes, this work takes into account semantic similarity relationship between goal category and Web service category (i.e., if two categories mean the same thing or inherit the same class).

(2) *Prefiltering Based on Semantic Similarity of FPs*. Authors of [7, 8] also used preprocessing strategies before the actual matching process. Their prefiltering is based on only FPs of Web services. They present two different SPARQL queries to facilitate the search process on a semantic Web service registry. They automatically create SPARQL queries (called Q_{all} , Q_{some}) by analysing the user request, and by using these two filtering queries they are able to perform two levels of filtering on the initial Web service repository. Based on these two queries, only Web services containing all (in the case of Q_{all}) or some (in the case of Q_{some}) concepts referred by a user request are returned.

Our second filtering stage (*Cap_Filt*) is similar to the method proposed in [8]. Four major differences between our work and theirs are that

- (i) since in our prefiltering stage service descriptions consist of all information about inputs, outputs, and pre- and postconditions, we can obtain more accurate results than their strategies;
- (ii) our algorithm not only considers the hierarchical relationship of concepts and attributes but also takes into account the similarity of requested/advertised Web service concepts and attributes based on their synonyms;
- (iii) we employ an initial filtering phase based upon a categorization scheme, which could actually improve their performance as well if they used it before Q_{all} or Q_{some} algorithm;
- (iv) their approach consists only of a preprocessing stage to filter the preliminary Web service repository and they did not implement any service matchmaking, so they cannot be evaluated on their own.

Among all the mentioned approaches, [27] is the closest to our work. The INFRAWEBs project implements a discovery framework which consists of two components, prefiltering and discovery. In the prefiltering stage it uses traditional Information Retrieval techniques, and a logic-based matching implemented in Prolog is utilized as a service matchmaker.

Although the INFRAWEBs project has similarities with our work, some differences do stand out. Our prefiltering stage considers semantic equivalency of both NFP and FP of the requested/advertised services, analyzing objects, attributes, and concepts. Our discovery engine works with much richer descriptions of Web services and requests, encoded in frame logic. Our implementation uses the FLORA-2 language and execution environment, a much more powerful alternative to plain Prolog. It is conceivable that a combination of our approach and theirs can yield a discovery framework that is more effective at eliminating useless Web services than either approach alone.

7. Conclusions and Future Works

We have shown that the overall performance and accuracy of semantic Web service discovery frameworks can be improved significantly through the introduction of prefiltering stages that eliminate most of the irrelevant Web services from consideration at the computationally expensive matching stage. Specifically, in this paper, we proposed *Category-based* and *Capability-based* prefiltering mechanisms for narrowing down the number of Web service descriptions that need to be considered in the matching phase to determine their relevance to the current goal.

We evaluated the effectiveness of our proposal in a novel test collection, *WSMO-FL*, which consists of 250 Web service specifications of varying complexities and 6 goals. Our filtering stages stand out due to their 100% recall rate that is a consequence of their design, their ability to deal with complex specifications of goals and Web services written in an enhanced version of WSMO, and a reasonably high precision rate, as demonstrated experimentally, which is bound to increase considerably in the presence of a large number of categories and goals/Web services that make use of those categories. Our results also indicate that when the prefiltering stages are employed in the system, as expected, the search space is considerably reduced, and consequently response time of the system is improved dramatically.

Our work has several further advantages, summed up in the following:

- (i) Unlike the majority of semantic Web service discovery approaches which are only performed on input and output concepts, our semantic Web service discovery framework deals with concepts and attributes of Web service and goal pre- and postconditions.
- (ii) Our prefiltering stages are generic, so that they can be applied (after necessary adaptations) to improve the performance of other available service matchmakers.
- (iii) 100% recall rate of our framework implies that our method does not result in false negatives (FN) (i.e., Web services which are relevant but are classified as irrelevant): all relevant Web services are retrieved through the prefiltering algorithms.
- (iv) Due to incomplete service descriptions in OWLS-TC test collections (i.e., Web services are partially described only based on input/output concepts), for

the first time we created a new test collection called *WSMO-FL*, which contains fully defined Web services and goals capabilities (i.e., Web services and goals are described based on pre- and postconditions).

- (v) To the best of our knowledge *WSMO-FL* is the first larger test collection which is established based on the *WSMO* conceptual model. It uses frame logic (F-Logic) as a fully adequate expression language for specifying pre- and postconditions which is missing in currently available test collections.

For future work, we are planning to improve our scheme in the following ways:

- (i) extending the second stage so that it can work on *any* logical expression containing the logical connectives *conjunction (and)*, *disjunction (or)*, and *negation (not)* to any nesting depth;
- (ii) extending our new *WSMO-FL* test collection to (a) have a much larger number of Web services and goals, as well as categories, (b) increase complexity of Web service and goal pre- and postconditions, and (c) expand the dictionary of synonymous words in the existing domain ontologies.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

- [1] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American*, vol. 284, no. 5, pp. 34–43, 2001.
- [2] S. A. McIlraith, T. C. Son, and H. Zeng, "Semantic web services," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46–53, 2001.
- [3] D. Roman, U. Keller, H. Lausen et al., "The Web service modelling ontology," *Applied Ontology*, vol. 1, no. 1, pp. 77–106, 2005.
- [4] D. Martin, M. Burstein, J. Hobbs et al., "OWL-S: Semantic Markup for Web Services," W3C member submission 22, 2004.
- [5] L. D. Ngan and R. Kanagasabai, "Semantic Web service discovery: State-of-the-art and research challenges," *Personal and Ubiquitous Computing*, vol. 17, no. 8, pp. 1741–1752, 2013.
- [6] M. Stollberg, M. Hepp, and J. Hoffman, "A caching mechanism for semantic web service discovery," in *The Semantic Web: 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11–15, 2007. Proceedings*, vol. 4825 of *Lecture Notes in Computer Science*, pp. 480–493, Springer, Berlin, Germany, 2007.
- [7] J. M. García, D. Ruiz, and A. Ruiz-Cortés, "Improving semantic web services discovery using SPARQL-based repository filtering," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 17, pp. 12–24, 2012.
- [8] J. Mara García, D. Ruiz, and A. Ruiz-Corts, "A lightweight prototype implementation of SPARQL filters for WSMO-based discovery," Tech. Rep., Applied Software Engineering Research Group-University of Seville., ISA-11-TR-01, 2011.

- [9] M. Klusch, B. Fries, and K. Sycara, "OWLS-MX: a hybrid Semantic Web service matchmaker for OWL-S services," *Web Semantics*, vol. 7, no. 2, pp. 121–133, 2009.
- [10] M. Klusch and F. Kaufer, "WSMO-MX: a hybrid Semantic Web service matchmaker," *Web Intelligence and Agent Systems*, vol. 7, no. 1, pp. 23–42, 2009.
- [11] D. Fensel, H. Lausen, A. Polleres et al., "The concepts of WSMO," in *Enabling Semantic Web Services: The Web Service Modeling Ontology*, pp. 63–81, Springer Science+Business Media, 2007.
- [12] M. Kifer, G. Lausen, and J. Wu, "Logical foundations of object-oriented and frame-based languages," *Journal of the ACM*, vol. 42, no. 4, pp. 741–843, 1995.
- [13] M. Kifer, G. Yang, H. Wan, and C. Zhao, *FLORA-2: User's Manual, Version 1.0*, Stony Brook University, Stony Brook, NY, USA, 2014.
- [14] U. Küster, B. König-Ries, M. Klein, and M. Stern, "DIANE: a matchmaking-centered framework for automated service discovery, composition, binding, and invocation on the web," *International Journal of Electronic Commerce*, vol. 12, no. 2, pp. 41–68, 2007.
- [15] H. Lausen and J. Farrell, "Semantic annotations for WSDL and XML schema," W3C recommendation, 2007.
- [16] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199–220, 1993.
- [17] J. Angele, "OntoBroker: mature and approved semantic middleware," *Semantic Web*, vol. 5, no. 3, pp. 221–235, 2014.
- [18] W. Chen, M. Kifer, and D. S. Warren, "HiLog: a foundation for higher-order logic programming," *Journal of Logic Programming*, vol. 15, no. 3, pp. 187–230, 1993.
- [19] A. Bonner and M. Kifer, "A logic for programming database transactions," in *Logics for Databases and Information Systems*, pp. 117–166, 1998.
- [20] A. J. Bonner and M. Kifer, "Overview of transaction logic," *Theoretical Computer Science*, vol. 133, no. 2, pp. 205–265, 1994.
- [21] H. Wan, B. Grosz, M. Kifer, P. Fodor, and S. Liang, "Logic programming with defaults and argumentation theories," in *Logic Programming*, vol. 5649 of *Lecture Notes in Computer Science*, pp. 432–448, Springer, Berlin, Germany, 2009.
- [22] M. Stollberg, J. Hoffmann, and D. Fensel, "A caching technique for optimizing automated service discovery," *International Journal of Semantic Computing (World Scientific)*, vol. 5, no. 1, pp. 1–31, 2011.
- [23] M. L. Sbodio, D. Martin, and C. Moulin, "Discovering Semantic Web services using SPARQL and intelligent agents," *Journal of Web Semantics*, vol. 8, no. 4, pp. 310–328, 2010.
- [24] R. Amorim, D. B. Claro, D. Lopes, P. Albers, and A. Andrade, "Improving web service discovery by a functional and structural approach," in *Proceedings of the IEEE 9th International Conference on Web Services (ICWS '11)*, pp. 411–418, IEEE, Washington, DC, USA, July 2011.
- [25] T. Khmour, "Towards semantically filtering web services repository," in *Digital Information and Communication Technology and Its Applications*, vol. 167 of *Communications in Computer and Information Science*, pp. 322–336, Springer, Berlin, Germany, 2011.
- [26] K. Mohebbi, S. Ibrahim, and M. Zamani, "A pre-matching filter to improve the query response time of semantic web service discovery," *Journal of Next Generation Information Technology*, vol. 4, no. 6, 2013.
- [27] L. Kovács, A. Micsik, and P. Pallinge, "Two-phase semantic web service discovery method for finding intersection matches using logic programming," in *Proceedings of the Workshop on Semantics for Web Services*, Zurich, Switzerland, December 2006.
- [28] C. Fellbaum, *WordNet: An Electronic Lexical Database*, Blackwell Publishing, Oxford, UK, 1998.
- [29] <https://wordnet.princeton.edu/>.
- [30] <http://projects.semwebcentral.org/projects/owls-tc/>.
- [31] <http://projects.semwebcentral.org/projects/sawsdl-tc/>.
- [32] M. Klusch, M. A. Khalid, P. Kapahnke, B. Fries, and M. V. Saarbrücken, *OWLS-TC -OWL-S Service Retrieval Test Collection, User Manual*, 2010.
- [33] <http://www.w3.org/Submission/SWRL/>.
- [34] C. Aeronautiques, A. Howe, C. Knoblock et al., *The Planning Domain Definition Language (PDDL)*, 1998.
- [35] M. A. Khalid, B. Fries, M. Vasileski, P. Kapahnke, and M. Klusch, *SAWSDL-TC Service Retrieval Test Collection, User Manual, Version 3.0*, SAWSDL, Saarbrücken, Germany, 2010.
- [36] Z. Cong, A. Fernandez, H. Billhardt, and M. Lujak, "Service discovery acceleration with hierarchical clustering," *Information Systems Frontiers*, vol. 17, no. 4, pp. 799–808, 2014.
- [37] <http://cmpe.emu.edu.tr/samira/WSMO-FL.htm>.
- [38] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, vol. 463, ACM Press, New York, NY, USA, 1999.
- [39] M. Klusch, "Overview of the S3 contest: performance evaluation of semantic service matchmakers," in *Semantic Web Services: Advancement through Evaluation*, pp. 17–34, Springer, Berlin, Germany, 2012.
- [40] <http://projects.semwebcentral.org/projects/sme2/>.
- [41] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara, "Semantic matching of web services capabilities," in *The Semantic Web—ISWC 2002: First International Semantic Web Conference Sardinia, Italy, June 9–12, 2002 Proceedings*, vol. 2342 of *Lecture Notes in Computer Science*, pp. 333–347, Springer, Berlin, Germany, 2002.



Hindawi

Submit your manuscripts at
<http://www.hindawi.com>

