DISCOVERY AND QUERY: TWO SEMANTIC PROCESSES FOR WEB SERVICES

Po Zhang[1], Juanzi Li[1] and Kehong Wang[1]
[1] *Department of Computer Science of Tsinghua University, 100084, Beijing, China*

Abstract: In this paper, we focuses on two process phases of web services, namely the web services discovery and web service query. Web services discovery is to locate the appropriate service, and web service query is to search the service data during execution of web service. When an end-user wants to book a ticket, he will discover the web service first, and then query this service to determine whether it can provide the satisfiable ticket. So the two process phases should work together, not only to discover possible satisfiable service, but to find the real satisfiable service data. It's a promising idea to adopt Semantic Web technology to implement the two processes. This paper first proposes the whole architecture for discovery and query, then gives four algorithms to discover web services and three algorithms to query web service based on the service data instance concept and similarity calculation. Accordingly, two frameworks separately for discovery and query are implemented. The result proves the approach of combination of two processes can really meet the personal requirements, so has certain application value.

Key words: Web Services Discovery; Query; Ontology; Similarity Computing.

1. INTRODUCTION

Web services [1] are self-contained, self-describing, modular applications that can be published, located and invoked across the web. Once a web service is deployed, other applications or other web services can discover and invoke it. UDDI, WSDL and SOAP, which are three most important technologies of web services, provide limited support in mechanizing service recognition and discovery, service configuration and combination, service comparison and automated negotiation. Service discovery is currently done by name/key/category of the information model in UDDI which roughly defines attributes that describe the service provider, the relationships with other providers and how to access the service instance. The fixed set of attributes in UDDI limits the way queries can be composed. Although UDDI can find more service information in WSDL, the WSDL only describes the

service in a low-level form of interface signature and communication protocol and can't provide enough semantic description for locating the service intelligently.

The semantic web [2] will make data on the web defined and linked in a way, that it can be used by machines - not just for display purposes, but also for using it in various applications. Bringing web services applications to their full potential requires their combination with semantic web technology.

Nowadays, lots of approaches for semantic web service discovery have been proposed. However, it's not enough for an end user only to find a web service interface. For example, which is from OpenTravel Alliance Message Users Guide [8] , "Bob is planning a trip for his wife and child to fly from London to Los Angeles. He would like to depart on August 13 and prefers a non-stop flight, but if he has to make a stopover, he prefers that there be a maximum of one stopover. He also would like to fly on a 757 and have his tickets mailed to him. He wants to fly economy class. Bob requests availability on flights from London to Los Angeles on August 13. And he wants to get the roundtrip tickets information." If Bob only find a web service interface which can provide a ticket, he still can not know whether the ticket can satisfy his concrete requirements such as the exact date or airplane type, even the logic expression of personal preferences.

So the approach proposed in this paper not only focused on semantic web service discovery, but adopts semantic web technology to support web service query. The combination with discovery and query can solve the above example case.

Section 2 proposes the whole architecture for combination of discovery and query. Upon this model and architecture, four algorithms for web service discovery and three algorithms for web service query is separately introduced in Section 3 and Section 4. Two frameworks for discovery and query are introduced in Section 5. Section 6 illustrates an example. Related works are discussed in section 7. Finally gives the conclusion and future work in section 8.

2. THE WHOLE ARCHITECTURE

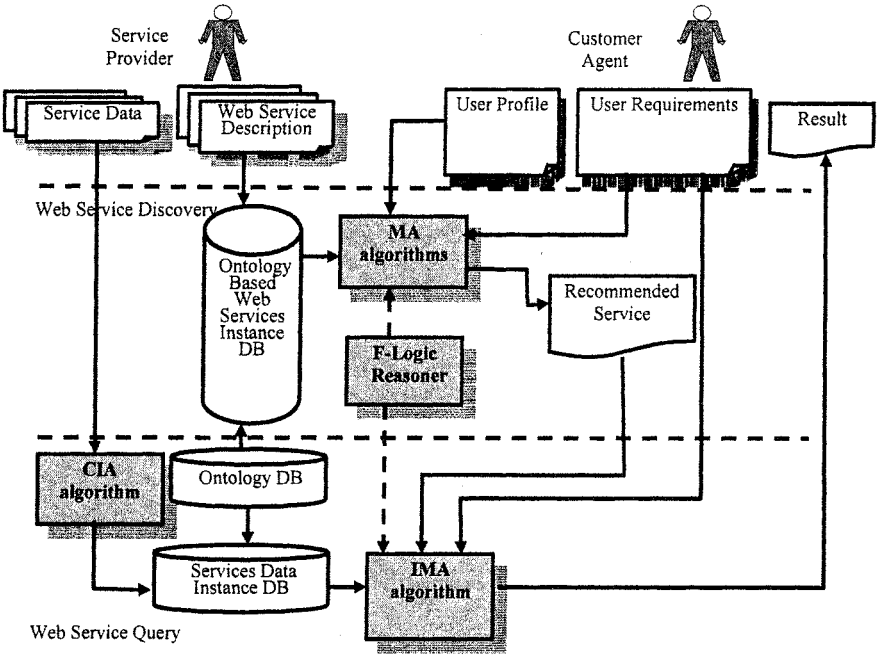

Figure 1. The whole architecture for web service discovery and query.

As illustrated in figure 1, Service Provider and Customer Agent are two important roles in this architecture. Service Provider provides web service description and service data for service discovery and query separately. Web service description will be stored in web services DB, and service data will be stored in service data instances DB through CIA (Create Instances Algorithm). MA is a general service match algorithm, which includes output/input match algorithm, precondition match algorithm and effect contentdegree algorithm. MA is used to fulfill the web services discovery and IMA (Instance Match Algorithm) is used to fulfill the web service query. F-Logic[7] Reasoner is to do the reasoning job during this two processes. Customer Agent provides the user profile which describes the user basic information and the user requirements which represents the service interface information and concrete service data requirements such as a specific ticket.

The ontologies in this architecture can be separated into two main categories, namely the domain ontology (DO) and service upper ontology

(SUO). The domain ontology used here is the travel domain ontology, which can be SchemaWeb[10], or from http://keg.cs.tsinghua.edu.cn/persons/zp/travel_onto_is.owl. And the service upper ontology adopts the OWL-S[11] Specification in order to align with current Web services Standards. OWL-S Specification is a OWL-based Web service ontology, and supplies Web service providers with a core set of markup language constructs for describing the properties and capabilities of their Web services in unambiguous, computer-interpretable form.

3. DISCOVERY ALGORITHMS

3.1 General Match Algorithm

algorithm MA(S, cusProfile, reqSO, reqEff)
/* Input: candidate Services S, custom Profile cusProfile, required Service Operation reqSO, required Effect reqEff*/
/* Output: Services which satisfy user requirement Result*/

1)Result ← S

2)ServiceOperationF ← getCommonOper(S)

3)ServiceOperationS ← getSpecOper(S)

4)For each service[i] in Result

5)isMatched ← false

6)SOper ← accessSerivceOper(service[i])

7)SEff ← accessServiceEff(service[i])

8)If (reqEff ∈ SEff) ∧ PrecondtionMatch(service[i], cusProfile) goto 9)
Else goto 19)

9)For each serviceoperation[i] in ServiceOperationF ∩ SOper

10)isMatched ← Output/InputMatch(serviceoperation[i], reqSO, Output|Input)
11)if (isMatched!=true) goto 19)

12)contentDegree ←queryOperEff(serviceoperation[i], reqEff)

13)remove serviceoperation[i] from ServiceOperationF
14)remove serviceoperation[i] from SOper

15)For each serviceoperation[i] in ServiceOperationS ∩ SOper Repeat 10)-12)
16)remove serviceoperation[i] from ServiceOperationS
17)remove serviceoperation[i] from SOper
18)If (SOper!=NULL) For each serviceoperation[i] in SOper Repeat 10)-12)
19)If (isMatched!=true) remove service[i] from Result

20)Result ← Result.sort(contentDegree)

21)Return Result

This algorithm is the general match algorithm for web services discovery. Two operation sets for different purposes are defined. ServiceOperationF is the set of common operations in different services; ServiceOperationS is the set of specific operations in different services, complementing to ServiceOperationF. Because ServiceOperationF is more general than ServiceOperationS, ServiceOperationF will be matched before ServiceOperationS. If the common operation can not meet user requirements, then the match algorithm will remove the candidate service from result immediately. This optimized method will have better performance to discovery web services. accessServiceOper() is the method to get all the operations in a service instance; accessServiceEff() is the method to get all the effects in a service instance. Other algorithms such as Output/Input Match algorithm and queryOperEff algorithm will be introduced in the following sections.

3.2 Output/Input Match Algorithm

algorithm Output/InputMatch(SO, reqSO, direction)

```
/* Input: Service Operation SO, required Service Operation reqSO, Output
or Input indicator direction*/
/* Output: true or false*/

1)isMatched ← false

2)If (direction==Output)  goto 3)  Else goto 9)

3)sCustomer ← accessOutput(reqSO)

4)If (sCustomer.size==0)  isMatched ← true, goto 14)

5)sOutput ← accessOutput(SO)

6)If (sOutput.size==0)  isMatched ← false, goto 14)

7)For each output[i] in sOutput

 isMatched ← isOntologySubsumeOrEqual(sCustomer, output[i])

8)goto 14)

9)sInput ← accessInput(SO)

10)If (sInput.size==0)  isMatched ← true, goto 14)

11)sCustomer ← accessInput(reqSO)

12)If (sCustomer.size==0)  isMatched ← false, goto 14)

13)For each input[i] in sInput

 isMatched ← isOntologySubsumeOrEqual(input[i], sCustomer)

14)Return isMatched
```

This algorithm is to match the output and input of an operation with user requirements. The main idea is that the outputs of service operation should meet the outputs of user requirements, while the inputs of user requirements should meet the inputs of service operation. For optimizing purpose, when the output of user requirements is empty or the input of service operation is

empty, the match succeeds. accessOutput() is the method to get all the outputs of an operation; accessInput() is the method to get all the inputs of an operation; isOntologySubsumeOrEqual() is the method to use ontology relations to determine the subsume or equal relation between two output/input class.

3.3 Precondtion Match Algorithm

```
algorithm PreconditionMatch(Serv, cusProfile)
/* Input: a Service instance Serv, customer profile cusProfile*/
/* Output: true or false*/
1)isMatched ← false
2)SPre ← accessServicePre(Serv)
3)If (SPre.size==0)   isMacthed ← true, goto 14)
4)tempPre ← ϕ
5)For each precondition[i] in SPre
6)flogicFact ← cusProfile
7)flogicRule ← precondition[i]
8)If(flogic_engine(flogicFact+flogicRule)==true)
      tempPre.append(precondition[i])
9)If isEqual(tempPre, SPre) isMatched← true, goto 14)
10)SOper ← accessSerivceOper(Serv)
11)For each serviceoperation[i] in SOper
12)OPre ← accessOperPrecondtion(serviceoperation[i])
13)If (tempPre.contains(OPre)) isMatched← true, goto 14)
14)Return isMatched
```

This algorithm is to match the preconditions of a service instance with user profile. From the general match algorithm introduced in section 4.1, it can be inferred that precondition match algorithm should be done earlier than output/input match algorithm. It's also for optimizing purpose. This algorithm adopts flogic reasoning engine, which takes the user profile as flogicfact and the preconditions as flogicrule, to determine whether the user profile can match preconditions. It's obvious that if all the preconditions of a service instance are matched, the match algorithm will succeed. However, if not all the preconditions are matched, then to find one operation whose

preconditions are all matched, if this operation exists, then the algorithm will also succeed. Otherwise, the algorithm fails. accessServicePre() is the method to get all the preconditions of a service instance; accessServiceOper() is the method to get all the operations in a service instance; accessOperPrecondtion() is the method to get all the preconditions of an operation.

3.4 Effect ContentDegree calculation Algorithm

algorithm queryOperEff(SO, reqEff)
/* Input: Service Operation SO, customer profile cusProfile*/
/* Output: contentDegree*/
1)If (reqEff.size==0) contentDegree ← 1, goto 7)
2) contentDegree ← 0
3)OEff ← accessOperEffect(SO)
4)If (OEff.size==0) goto 7)
5)contentDegree ← contentDegree + (reqEff∩OEff).size
6)contentDegree ← contentDegree/reqEff.size
7)Return contentDegree

This algorithm is to calculate the contentdegree to represent the effects match degree to user requirements. The contentdegree can be defined as the ratio of the cardinality of effects intersection set to the cardinality of requirement effects set. It's an important value for MA algorithm to sort the candidate services result.

4. QUERY CALCULATIONS

4.1 Service Data Instance and Similarity calculation

Definition 4.1 Service Data Instance. The Ontology Instance constructed from service data based domain ontology is called Service Data Instance.

Definition 4.2 Category of Similarity. Assume a Service Data Instance I has N properties, in which p properties are plain text type, q properties are enumerated type, r properties are numerical type and k properties are boolean type, N = p + q + r + k. I can be represented as

$P(I) = \{(PT_1, PT_2,\ldots, PT_p), (PE_1, PE_2,\ldots, PE_q), (PN_1, PN_2,\ldots, PN_r), (PB_1, PB_2,\ldots, PB_k)\}$

PT stands for text property, PE stands for enumerated property, PN stands for numerical property, PB stands for boolean property.

Definition 4.3 Boolean Similarity. It's the simplest similarity, which is to compute the similarity of boolean type properties. The similarity function of PBs is as follows:

$$\mathrm{SimB}(PB_i, PB_j) = \overline{XOR(value_1, value_2)}$$

Definition 4.4 Numerical Similarity. It is used to compute the similarity of numerical type properties. The similarity function of PNs is as follows:

$$\mathrm{SimN}(PN_i, PN_j) = 1 - \frac{|value_i - value_j|}{MAX(value_i, value_j)}$$

Definition 4.5 Enumeration Similarity. It is used to compute the similarity between enumerated type properties. Set operation is used to define the similarity.

$$\mathrm{SimE}(PE_i, PE_j) = \frac{|\mathrm{Set}(PE_i) \cap \mathrm{Set}(PE_j)|}{|\mathrm{Set}(PE_i) \cup \mathrm{Set}(PE_j)|}$$

Set(PE) represents the value set of PE, the calculation result is the ratio of the cardinality of intersection set to the cardinality of union set.

Definition 4.6 Text Similarity. It is used to compute the similarity of plain text type properties. With the cosine measure, the similarity function of FTs is as follows:

$$\mathrm{SimT}(PT_i, PT_j) = \frac{\sum_{k=1}^{S} w_i^k w_j^k}{\sqrt{(\sum_{k=1}^{S} w_i^{k\,2})(\sum_{k=1}^{S} w_j^{k\,2})}}$$

w_i^k is the weight for kth value of feature vector of plain text, can be computed by frequency of features.

Definition 4.7 Single Layer Similarity. It's to compute different type of properties between two Ontology Instances, and then get the similarity

between them. When encounter an objectproperty, the URI of this property referring to should be considered as plain text type and processed using Text Similarity.

Definition 4.8 Multi Layer Similarity. Different from Single Layer Similarity, when encounter an objectproperty, the URI of this property referring to should be considered as another Instance whose depth increases one, then assign the deeper layer instance similarity as this property similarity. This recursive calculation won't stop until all the properties are datatypeproperty or reach stop condition. The deepest layer similarity is the same as Single Layer Similarity. The calculated similarity is called Multi Layer Similarity.

4.2 Create Instances Algorithm

algorithm CIA(S, Onto)
/* Input: Web Service S, Ontology Model Onto*/
/* Output: The all service data instance InstanceList */

1) InstanceList ← ϕ

2) while (S has new output)
3) O1 ← the next output of S
4) Map O1 to Class C1 in Onto
5) for each property p of C1
7) if (p is DataTypeProperty) then goto 9)
8) else If (p is ObjectProperty) then goto 13)
9) Get the data value v of p from S, and erase v from S
10) if (the data value v is null) goto 5)
11) Store the data value v to p
12) goto 5)
13) if reach the stop condition goto 5)
14) for each range classes C of p
15) push C1
16) C1 ← C
17) goto 5)
18) pop C1
19) goto 14)

20) goto 5)
21) generate a new instance I1
22) append I1 to InstanceList
23) goto 2)
24) return InstanceList

This algorithm is to create instance from service data. Assuming service S has N outputs mapping to N classes in ontology DB, and one class averagely has M datatype properties and I object properties. When recursing k layers, the time complexity to create a service instance is equal to the time complexity to iterate all the properties and assign them: $N \times (M + I \times M + I^2 \times M + \cdots + I^{k-1} M + I^k)$, approximate $O(2NM^k)$ when I≈ M. This time complexity is exponential complexity to the number of properties of one class. Assuming the average length of one property value is L, the space complexity is $O(2NLM^k)$.

4.3 Multi Layer Similarity Calculation Algorithm

algorithm ComputeSimilarity(I1, I2, depth)
/* Input: ontology instance I1, ontology instance I2*/
/* Output: the similarity between I1 and I2*/
1) similarity ← 0, count ← 0
2) for each property of I1
3) if (p is DataTypeProperty) then goto 5)
4) else If (p is ObjectProperty) then goto 8)
5) similarity ← similarity + ComputeSimilarity(data value v1 of p in I1, data value v2 of p in I2)
6) count ← count + 1
7) goto 2)
8) if reach the stop condition goto 14)
9) for each range instances I1' of p in I1 and I2' of p in I2
10) similarity ← similarity + ComputeSimilarity(I1', I2', depth+1)
11) count ← count + 1
12) goto 9)
13) goto 2)
14) similarity ← similarity + ComputeSimilarity(uri1 of range instance of p in I1, uri2 of range instance of p in I2)

15) count ← count + 1
16) goto 2)
17) similarity ← similarity/count
18) return similarity

This algorithm is to calculate the multiple layer similarity between two ontology instances. Assuming one class averagely has M properties. When recursing k layers, the time complexity to compute the two ontology instances is as twice as the time complexity to iterate all the properties and read them. From section 4.2, this time complexity is $O(4M^k)$, and the space complexity is $O(4LM^k)$.

4.4 Instance Match Algorithm

algorithm IMA(Req, S, Onto, Rules)
/* Input: User Reqirements Req, Web Service S, Ontology Model Onto, Logic Rules */
/* Output: The all satisfiable service data instances ResultList */

1) ResultList ← ϕ

2) InstanceList ← CIA(S, Onto)
3) for each instance I in InstanceList
4) for each rule R in Rules
5) flogicFact ← I
6) flogicRule ← R
7)If (flogic_engine(flogicFact+flogicRule)==true)
then goto 9)
8) else goto 3)
9) goto 4)
10) similarity ← ComputeSimilarity(I, Req, 1)
11) assign similarity to instance I
12) append (I, similarity) to ResultList
13) goto 3)
14) ResultList.sort
15) return ResultList

This algorithm is to control CIA algorithm and similarity calculation algorithm. It also adopts flogic reasoning engine to do the logic comparison

between the user preferences and service data instances, which takes the service data instances as flogic fact and the user preferences logic as flogic rule.

5. TWO FRAMEWORKS

5.1 The Web Services Discovery Framework

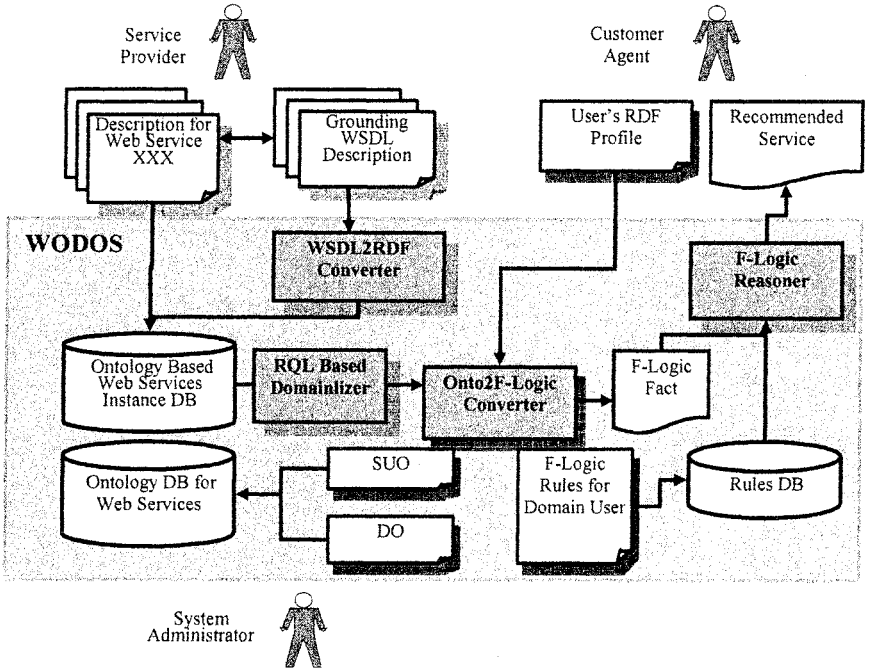

Figure 2. Web Services Intelligent Discovery System.

WSIDS (Web Services Intelligent Discovery System) is an intelligent web services discovery framework and consists of several components that cooperate in semantic Web Service discovery.

We use WODOS (Web Oriented Distributed Ontology System) to build the framework. The WODOS is a semantic web infrastructure system developed by us, something like Jena. The WSIDS will use these characteristics of WODOS:

1. Store the ontology in the format of RDF/OWL-Lite into the relationship data base.
2. Import/export the OWL-Lite/RDF file.
3. Query ontology of base using RQL
4. An embedded F-logic inference engine witch can import OWL-Lite/RDFS ontology as F-Logic's fact using a converter.

The key of the WODOS is that the platform is a "total solution" for the semantic web applications built on an expansible, flexible, scalable and open architecture. In this system, varied operations of ontology are added, updated and deleted dynamically. So we bring forward a WSDL2RDF converter and deploy it into WODOS. The WODOS is extended for the WSIDS.

As illustrated in figure 2, it is system administrator to create DO and SUO by the standard of OWL-Lite/RDFS. The system administrator can use some tools to do this, such as OntoEdit. He also need create F-Logic rules for domain users' references and constraints description. After the system administrator create the ontology and rules, WODOS can import the OWL-Lite files and F-Logic files into the database.

To discover web services, we need set up a services database based on the ontology system created by the administrator. The database will include semantic description of many services. The information need be provided by service provider. If the service provider need to insert a new service into the database, the WSDL file of this service and a service description OWL file grounded to the WSDL should be created. Then the WSDL2RDF converter will converter WSDL file to RDF file and import result file into database of WODOS together with the OWL file grounded to it.

The discovery procedure will begin if a customer agent gives a request. This request includes user's profile which includes the information about the customer such as country, sex, business trip or personal trip and so on. Then this profile will be translated to the F-Logic's fact by Onto2F-Logic converter. The Onto2F-Logic converter also translates all ontology of four layers into the F-Logic's fact. Before that, the RQL Based Domainlizer will "cut out" the ontology in the Web Services Instance DB into the same domain as the request of the customer agent. This is because of there are perhaps many domains of services ontology in the Instance DB and they need not be translated into F-Logic's fact. As getting the fact, the F-Logic

reasoner also get the preferences and constraints rules from the rules DB, as the result of F-Logic reasoner, a list of services, which are recommended to customer and are not reject by the preferences and constraints rules, are brought forward to customer agent.

5.2 The Web Service Query Framework

WSDIM(Web Service Data Instances Matcher) is a framework to implement the Web Service query computation and algorithms. It not only supports the First-Order Logic such as F-Logic, but supports the Description Logic such as Racer [13]. The graphical user interface of this framework is illustrated in figure 3. This similarity result is the result of the example introduced in section 6.

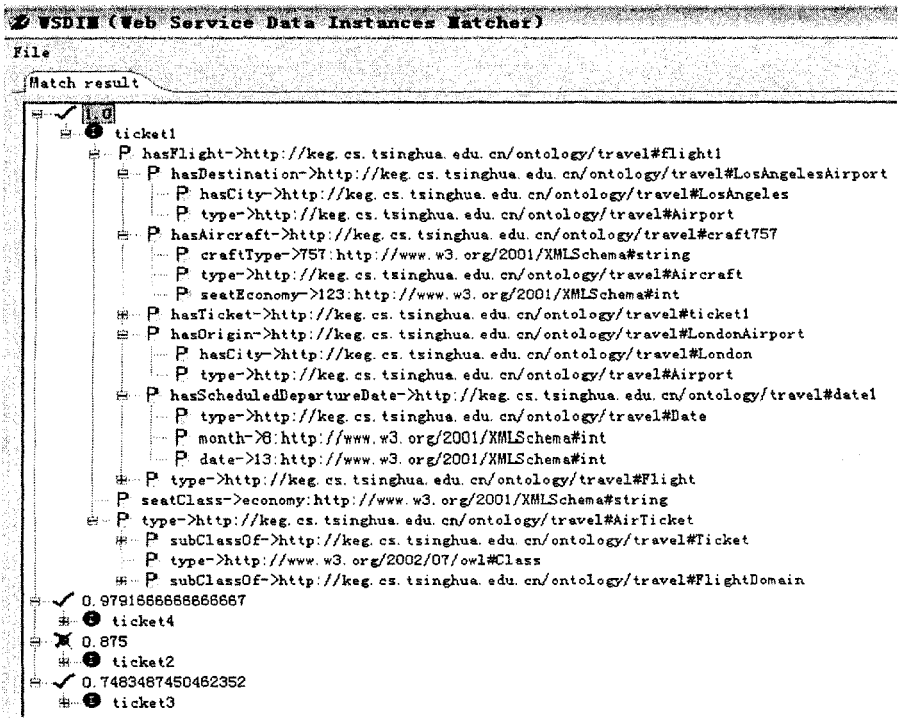

Figure 3. The similarity result computed by WSDIM.

6. AN EXAMPLE

As the use case described in OpenTravel Alliance Message Users Guide [8], "Bob is planning a trip for his wife and child to fly from London to Los Angeles. He would like to depart on August 13 and prefers a non-stop flight, but if he has to make a stopover, he prefers that there be a maximum of one stopover. He also would like to fly on a 757 and have his tickets mailed to him. He wants to fly economy class. Bob requests availability on flights from London to Los Angeles on August 13. And he wants to get the roundtrip tickets information."

Assuming there are four different web services: AirChina_Service, CathayPacific_Service, ShanghaiAir_Service and NorthernAir_Service. AirChina_Service will provide an airchina ticket as output, while using ticket information and credit card information as inputs. The other three services also have ticket as output and the same inputs. According to Bob's requirements, all these four services can go through Output/Input Algorithm. However, the precondition of AirChina_Service requires the membership of airchia_zhiyin club which is not true for Bob whose profile gives the membership of POP and CSSS club. And the effects of ShanghaiAir_Service and CathayPacific_Service don't include the "ticketmailed" effect in user requirement, so the contentdegree of these two services are both 0 according to Effect ContentDegree calculation Algorithm, while NorthernAir_Service can provide this effect, gets contentdegree 1. Then through the general MA algorithm, NorthernAir_Service which has the highest contentdegree will be recommended to Bob. Up to now, the discovery of web services has been finished.

Then Assuming NorthernAir_Service has some different service data, and we can pick four of them to delegate all the service data. The four service data (tickets) are: ticket1, ticket2, ticket3, ticket4. In which, ticket1, ticket2 and ticket4 is from London to LosAngeles, while ticket3 is from LosAngeles to London; the departure date of ticket1, ticket2 and ticket3 are all August 13, while ticket4 is October 15; ticket1 doesn't have a stopover, while ticket3 and ticket4 have one stopover NewYork airport, and ticket2 have two stopover NewYork airport and Washington airport; the airplane type of

ticket1 and ticket4 is 757, while ticket3 is 747 and ticket2 is unknown; all these tickets provide economic class.

NorthernAir_Service description can be obtained from http://keg.cs.tsinghua.edu.cn/persons/zp/NorthernAir.owl, while the four service data instances can be obtained from http://keg.cs.tsinghua.edu.cn/persons/zp/NorthernAirInstances.owl. The similarity calculation algorithm is to get the similarity between two ontology instances. Now, the service data instances have been constructed, so the user requirements should also be constructed as ontology instance as illustrated in figure 4.

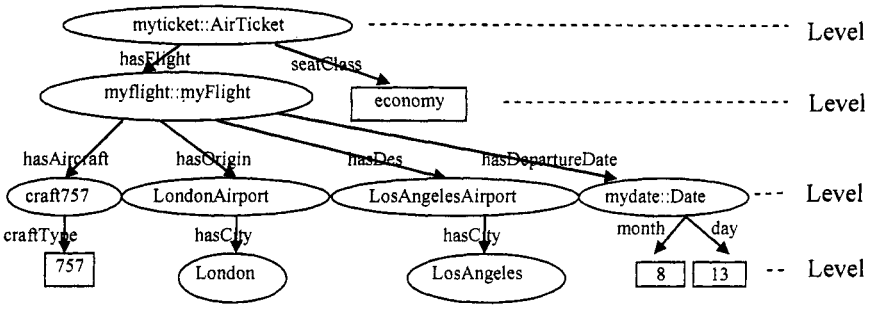

Figure 4. The ontology instance for end user requirements.

And the logic expression for user requirements can be written in f-logic format:

```
FORALL X,Y,X1,Y1,XO,XD isReturnTicket(X,Y) <-
  X["hasFlight"->>X1]
  AND Y["hasFlight"->>Y1]
  AND X1["hasOrigin"->>XO]
  AND X1["hasDestination"->>XD]
  AND Y1["hasOrigin"->>XD]
  AND Y1["hasDestination"->>XO].
```

This means the ticket X and ticket Y is roundtrip tickets whose origin airport is the other's destination airport and destination airport is the other's origin airport. This flogic result through F-Logic Engine can be obtained at http://keg.cs.tsinghua.edu.cn/persons/zp/result.txt.

Then through IMA algorithm, the sort of the four tickets is: tikcet1 1.0, ticket4 0.979, ticket2 0.875, ticket3 0.748. So the NorthernAir_Service not only provides the service interface to meet user requirements, but really provides the specific satisfiable ticket – ticket1.

After finishing the two processes, Bob eventually get the satisfiable ticket from NorthernAir_Service.

The meaning of myticket:AirTicket in figure 4 is that myticket is the instance of ontology class AirTicket, in order to represent the required ticket information. myflight:myFlight defines a new class myFlight, which is the subclass of class Flight, in order to extend the constraints for class Flight. Class myFlight adds owl:minCardinality and owl:maxCardinality constraints to the property hasStopover of class Flight, in which owl:minCardninality is 0 and owl:maxCardinality is 1, represents the user requirement for stopover constraints. The user request can be obtained from http://keg.cs.tsinghua.edu.cn/persons/zp/request.owl

7. RELATED WORKS

Now semantic web service has become a hot research topic, which uses ontology concept to enhance service discovery. The work in [4] annotates the operation, input, and output description of a Web Service, described in WSDL format, with DAML+OIL-based ontological concepts. Precondition and effect of the service are also added to WSDL as additional information such as [9] [5], but they are not used for queries as only the matching of the operation, input, and output is considered. The work in [6] and [3] both consider behavioral aspects in their service models but those aspects are not fully considered or used as query constraints for service matching. The work in [3] enhances WSDL with DAML+OIL based ontological information and considers a web service by its behavioral aspects all rounds. It allows the operation, input, output, precondition, and effect to be used as query constraints, and additionally consider the case when output or effect of the service has some conditions placed on them - the case when we provide a rule-based reasoning to determine the output and effect for query matching. And METEOR-S project [12] also proposed a infrastructure for semantic

publication and discovery of web services, which is called METEOR-S WSDI.

It seems that the research on web services discovery is more active than web service query. The technology used in web service query is still based on keyword search. Adopting semantic web technology to assist web service query, and then to combine web services discovery and query has more application value for meeting end user requirements.

8. CONCLUSION AND THE FUTURE WORK

The new features in this paper comparing to other approaches are as follows:

The first is this paper use two kinds of ontologies for web services discovery and query. The domain ontology is a travel domain ontology, which constructed from Open Travel Alliance Guide which is a famous travel alliance including more than 150 travel industry companies. This ontology can be obtained from SchemaWeb[10], or from http://keg.cs.tsinghua.edu.cn/persons/zp/travel_onto_is.owl. And the service upper ontology adopts the OWL-S Specification in order to align with current Web services Standards.

The second is to propose a whole architecture for web services discovery and query, which is based on the service model above, and to represent end user requirements with flogic based user preferences and constraints.

Then various algorithms is given for the whole architecture proposed above, including the more detailed algorithms for web services discovery and the similarity calculation based algorithms for web service query. And two frameworks are implemented accordingly.

Finally the use case illustrated in this paper can be represented and solved easily in this architecture, and shows the application value for end user.

How to reduce the exponential complexity similarity algorithm to lower complexity is an important future study issue. And the automatic generation of preferences and constraints rules form learning of user choices is an important and interesting issue that will also be studied further.

9. REFERENCES

1. Hugo Haas. Web Services activity statement. W3C, http://www.w3.org/2002/ws/Activity, 2001.
2. Tim Berners-Lee, James Hendler, Ora Lassila. The Semantic Web. Scientific American, 2001, 284(5):34-43.
3. Natenapa Sriharee and Twittie Senivongse, Department of Computer Engineering Chulalongkorn University, "Discovering Web Services Using Behavioral Constraints and Ontology", Distributed Applications and Interoperable Systems, 4th IFIP WG6.1 International Conference, DAIS 2003, Proceedings, Springer, Paris, France, November 17-21, 2003, pp.248-259.
4. Kaarthik Sivashanmugam, Kunal Verma, Amit P. Sheth, John A. Miller, "Adding Semantics to Web Services Standards", Proceedings of the International Conference on Web Services, ICWS '03, Las Vegas, Nevada, USA. CSREA Press 2003, pp.395-401
5. Peer, J, "Bringing Together Semantic Web and Web Services", Proceedings of the 1st International Semantic Web Conference (ISWC 2002), Lecture Notes in Computer Science Vol. 2342. Springer Verlag, Sardinia (Italy), 2002, pp.279-291.
6. Paolucci, M. et al., "Semantic Matching of Web Services Capabilities", Proceedings of the 1st International Semantic Web Conference (ISWC 2002), Sardinia (Italy), Lecture Notes in Computer Science, Vol. 2342. Springer Verlag (2002).
7. Michael Kifer, Georg Lausen, and James Wu. Logical foundations of objectoriented and frame-based languages. Journal of the ACM, 42(4), 1995, pp. 741-843.
8. http://www.opentravel.org/
9. Uche Ogbuji. Supercharging WSDL with RDF - Managing structured Web Service metadata. IBM developerWorks article, 2000.
10. Po Zhang. Travel Ontology. http://www.schemaweb.info/schema/SchemaDetails.aspx?id=236, 2005.2.
11. http://www.daml.org/services/owl-s/
12. METEOR Project on Workflow and Semantic Web Process, http://lsdis.cs.uga.edu/proj/meteor/meteor.html
13. http://www.sts.tu-harburg.de/~r.f.moeller/racer/