Web Service Aggregation and Selection Based on Join Operation in RDB

Jianxiao Liu[1][2], Xiaoxia Li[2], Zhihua Xia[1]

[1] School of Computer & Software, Nanjing University of Information Science & Technology, China
[2] College of Informatics, Huazhong Agricultural University, China
liujianxiao321@163.com, lixiaoxiahn@163.com, xia_zhihua@163.com

Abstract

How to realize Web service organization and management quickly and accurately, and build an effective service selection mechanism to choose services with correlations to meet users' functional and non-functional requests, and thus to meet the individual and dynamic changing requirements is a key problem in the Service-Oriented Software Engineering (*SOSE*). The Web service and ontology information are stored into RDB (relational database) in our method, it realizes Web service aggregation and selection in term of service interface (*Input* and *Output*) and execution capability (*Precondition* and *Effect*). Firstly, the Web service clustering method based on self-join operation in RDB is proposed to cluster services efficiency. Then it uses the abstract service extraction method to get abstract services, and uses Web service aggregation approach based on join operation to organize the clustered services. Finally, the Web service selection method is proposed to select the atomic service and a set of services with correlations to meet users' functional and *QoS* (quality of service) requirements. In addition, the case study and experiments are used to explain and verify the effectiveness of the proposed methods.

Keywords: Web Service Clustering, Aggregation, Join operation, Selection

1 Introduction

There are all kinds of services on the internet in the era of service computing. Users will choose services to meet their individual and dynamic changing requirements, which include the functional and *QoS* (Quality of Service) requests. It needs to organize and manage Web service efficiently, and thus to supply the services with high *QoE* (Quality of Experience) for users and enhance the quality of on-demand service [1].

As we all know, users' individual requests mainly include functional and non-functional (*QoS*) requirements. We should consider the two kinds of requirements when to realize service organization. Service organization refers to organize all kinds of Web services in the service registration center based on users' requirements. Service clustering is a service organization method according to users' functional requirement. While there are various types of services which realize different functions on the internet, and users usually need a set of services with different functions. Therefore, it is necessary to organize service clusters in further based on clustering Web services with similar functions. And this approach is called as service aggregation. Service aggregation refers to organize service clusters according to service execution relationships. Then users can find a set of services with correlations to meet their requirements quickly and accurately in the organized services. In addition, there are several Web service description languages [2], such as WSDL, OWL-S, WADL, SWSO/SWSL, WSMO/WSML, etc. We can use ontology to annotate these heterogeneous services that are described using different languages from the semantic level. How to annotate services (interface and capability) in the service registration center using ontology and do matching calculation from the semantic level, and thus help to organize and select services for users are key technologies to be solved. The main work of this paper includes the following aspects.

(1) It proposes a Web service clustering approach based on self-join operation in RDB. This approach calculates the matching degree between services from aspects of interface and capability, and uses the self-join operation in RDB to join the related tables. The semantic reasoning relationships and status path of concepts are used to do the calculation. It can enhance Web service clustering efficiency and accuracy.

(2) On the basis of service clustering, it proposes a Web service aggregation method using join operation in RDB. The abstract service of specific type is extracted from the service clusters firstly. Then it gets the execution dependency relationships between abstract services in term of service interface and capability. The join operation in RDB is used to organize service clusters and thus to realize Web

*Corresponding Author: Zhihua Xia; E-mail: xia_zhihua@163.com
DOI: 10.3966/160792642018011901005

service aggregation.

(3) A Web service selection method is proposed to meet users' individual requirements. Services are selected to form service execution path dynamically according to different requests. Then it selects proper services furtherly to meet users' *QoS* request. This method can help users to get the atomic service and a set of services with correlations, and thus to meet users' functional and *QoS* requirements.

(4) The case study and experiments are used to illustrate and verify the proposed methods.

The related work will be described in Section 2. The overall architecture is introduced in Section 3. The algorithms of realizing Web service aggregation and selection are elaborated in Section 4. The case study is described in Section 5. In Section 6, the experiments are used to validate the proposed approaches. The conclusion and next step work are given finally.

2 Related Work

This section elaborates the related wok about Web service clustering, service aggregation and selection.

2.1 Web Service Clustering

Service clustering refers to organize candidate Web services by unsupervised classification approach. Skoutas et al. have proposed a ranking and clustering Web services method by defining the multi-criteria dominance relationships between services [3]. Li *et al.* have proposed a topic-oriented clustering approach for domain services is [4]. This approach can cluster services described in WSDL, owls and text, which can effectively solve the problem of single service document type. Dasgupta et al. have proposed a self-organizing clustering algorithm called Taxonomic clustering for organizing semantic Web Services taxonomically [5]. Yu et al. have classified semantic services according to different topics, functionality and other aspects [6]. Liu *et al.* have proposed a service clustering method using service ontology [7]. The service ontology is got through the modeling of specific service type, and it uses service ontology to enhance service clustering efficiency. But the accuracy of clustering will be determined by the accuracy of service ontology. Kumara et al. mainly used the ontology learning technology and information-retrieval-based term similarity method to realize service clustering [8]. Wu et al. have proposed a service clustering method called WTCluster [9]. This method mainly uses WSDL document and tags to cluster services and two kinds of labels evaluation and recommendation strategies are given. Kumara *et al.* have proposed a context aware post-filtering for Web service clustering method [10].

The other aspect research work concentrates on using the traditional clustering methods [11]. Elgazzar et al. have proposed a novel technique to mine Web Service Description Language (WSDL) documents and cluster them into Web service groups with similar functions [12]. The Quality Threshold (QT) clustering algorithm is used. Yu et al. use the k-means algorithm to cluster service through computing the similarity of the operation and other services properties [13], and the service community can be formed. Liu et al. have clustered services to form service community through clustering the basic terms about services [14]. The k-means clustering algorithm is used in [15] to cluster the users' requests and candidate services. Sellami et al. have proposed a functionality-driven clustering approach for distributed Web service registries [16]. This approach uses fuzzy clustering technique.

Some of the above Web services clustering approaches use the traditional algorithms to cluster Web services. But these methods do not use the ontology technology to do the matching computation from the semantic level, and it will influence the Web service clustering accuracy. In addition, the time of some methods is too much when to calculate the service similarity, and this leads to service clustering efficiency is too low. Our method in the paper uses the semantic reasoning relationship between concepts, and we use the self-join operation in RDB to do the calculation on the tables of service interface and capability. The parameter matching calculation times can be reduced and the service clustering efficiency and accuracy will be enhanced. In addition, all the service clustering methods only organize the services which realize similar function, but the services with different function are not considered. This paper proposes a Web service aggregation method based on join operation in RDB to organize the formed service clusters furtherly.

2.2 Web Service Aggregation

Service aggregation refers to organize service clusters according to service dependency relationships. There exist two kinds of research work about service organization. The first one refers to organize services in real-time according to users' individual requirements in the service registration center. The implementation of this method is difficult and the efficiency is relatively low [17]. The second one refers to use the relationship between services and organize them based on users' common requests. Services are organized in the orientation of specific topic in this method. When users' individual requirements have been proposed, it will select service directly based on the service execution dependency relationships. This approach can make full use of the service relationships, and thus to help find the atomic service and a set of services with correlations quickly and accurately. We mainly concentrate on the second aspect.

The second aspect mainly includes the following research work. Wu et al. have used a logical petri net-based approach to compose service clusters in virtual layer [18]. The basic composition models of service cluster nets (SCNs) are presented. Zhou et al. have concentrated on the data providing services discovery [19]. On the basis of clustering data providing services, they have mentioned organizing different service clusters into cluster network. But they have not elaborated the detail process of how to organize service clusters. On the basis of Web service clustering, we have organized the service clusters from aspects of semantic interoperability [20] and users' requirement features (role, goal, process) [21]. Hu *et al.* have proposed a user-oriented service workflow constructing method [22]. The services are clustered and the spanning tree approach is used to represent the services in the same cluster firstly. Then the service clusters are organized through the workflow business logic method. This method mainly uses the hybrid particle swarm optimization algorithm to select services with the best *QoS*. Aznag et al. have used the Formal Concept Analysis (FCA) formalism to organize the constructed hierarchical clusters into concept lattices according to their topics [23]. Liu *et al.* have organized Web services using the method of service group and service node [24]. Service group is similar to the service clusters that are formed through clustering, and service node is similar to the abstract service of specific service cluster in our method. Sellami et al. have used the community to organize and manage Web services [25]. The fuzzy clustering algorithm is used to cluster services to form service community, and the service communities are organized from the point of functionality.

The above research approaches use the service execution relationship to organize services. But some of them are not doing the matching calculation from the semantic level, and it can influence the service organization accuracy. Some of them only consider the business logic execution relationship, but the service interface, execution capability and *QoS* information are not considered. Based on the Web service clustering and extracting abstract services, this paper uses the join operation in RDB to determine the abstract service execution dependency relationships quickly. Then it aggregates and organizes services in further. This method can lay the foundation of service selection to meet users' individual requirements.

2.3 Web Service Selection

Web service selection refers to select services to meet users' requirements in the registration center. There are a lot of research work about Web service selection. The *QoS*-based service selection method mainly uses certain mechanism to select services with proper *QoS* values for users' non-functional requests [26]. Wang *et al.* have proposed a composite service selection method [27]. They use fuzzy linear programming technology to select composite services, and it is a feasible and supplementary manner to select services. There also exists some other service selection approaches, such as the method of resolving the conflicting requests [28], service selection for dynamic service binding at runtime [29], ant colony method [30], agent-based technology, cooperative evolution based method [31], adaptive learning mechanism [32], etc.. The above methods use different mechanisms to select services from different aspects. On the basis of realizing service aggregation, this paper selects the proper atomic service and a set of services with correlations directly according to users' requests. It can help to enhance service selection efficiency and accuracy.

3 Overall Architecture and Definition

3.1 Overall Architecture

On the basis of storing the information of Web service and ontology, we realize Web service aggregation and selection based on the join operation in RDB. The overall architecture is shown in Figure 1.

In Figure 1, the architecture mainly includes the following parts.

Web service and ontology information storage. It uses tables in RDB to store Web service basic information, such as *Input*, *Output*, *precondition*, etc. The ontology is also stored for the calculation from the semantic level. Services are annotated by the ontology concepts, and it can help to enhance the accuracy of Web service aggregation and selection.

Web service clustering. The tables of service interface and capability are joined using self-join operation in RDB. This can reduce the times of calculating parameter matching degree, and enhance service clustering efficiency and accuracy. Services can be clustered to form different service clusters, and thus to make foundation for service aggregation.

Web service aggregation. On the basis of service clustering, it uses abstract service extraction algorithm to extract abstract services in different service clusters. The abstract service execution relationship can be got in the view of service interface and capability. The join operation is used to organize different service clusters, and thus to realize service aggregation.

Service selection. According to users' individual requests, it uses Web service selection method to get services that can meet users' function and *QoS* requirements. The atomic service and a set of services with correlations can be selected for users.

3.2 Web Service Definition and Storage

Based on the definition of environment ontology [33], we use the concept status transformation to describe the capability of Web services.

Definition 1. Web Service (*ws*): *ws*={*WSName*, *Interface*, *Capability*, *QoS*}

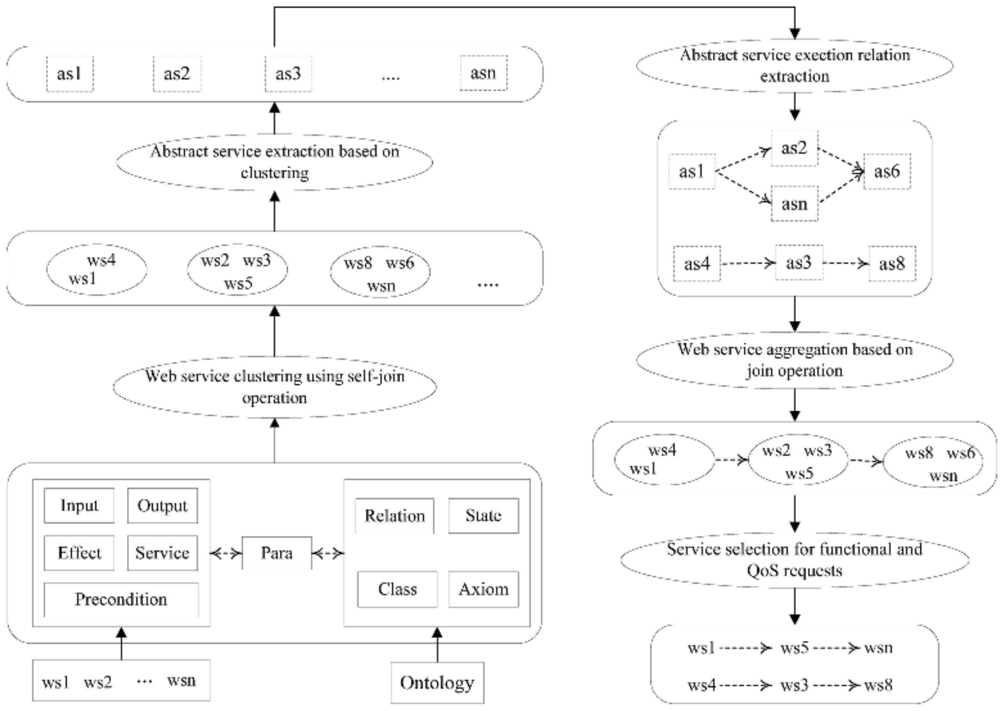

as1
as2
as3
....
asn
Abstract service extraction based on clustering
ws4 ws1
ws2 ws3 ws5
ws8 ws6 wsn
....
Web service clustering using self-join operation
Input
Output
Effect
Service
Precondition
Para
Relation
State
Class
Axiom
ws1 ws2 ··· wsn
Ontology
Abstract service exection relation extraction
as1
as2
asn
as6
as4
as3
as8
Web service aggregation based on join operation
Service selection for functional and QoS requests
ws1 ws5 wsn
ws4 ws3 ws8

Figure 1. Overall architecture of our approach

• *WSName* represents the name of *ws*.

• *Interface*={*Input*, *Output*}, and it denotes the input and output set of *ws*.

$Input=\{In_i, In_i \in Class, i=0,1,\ldots, i\text{num}\}$

$\text{Output}=\{\text{Out}_o, \text{Out}_o \in \text{Class}, o=0,1,\ldots, \text{onum}\}$

• *Capability*={*Precondition*, *Effect*}, it indicates the prerequisite for service execution and the effect resulting from *ws* execution.

$\text{Precondition}=\{\text{Prec}_p, p=0,1,\ldots, \text{pnum}\}$, $\text{Prec}_p =\{c_p: s_p, c_p \in \text{Class}, s_p \in \text{hsm}(c_p)\}$

$Effect=\{Eff_e, e=0,1,\ldots,e\text{num}\}$, $Eff_e=\{c_e:rt_e \to ot_e, c_e \in Class, rt_e, ot_e \in \text{hsm}(c_e), e=0,1,\ldots,e\text{num}\}$

• $QoS=\{\{QosName_q, Value_q\}, QosName_q \in Class, q=1,2\ldots q\text{num}\}$. $QosName_q$ can be time, cost, reliability, availability of *ws*. $Value_q$ represents the specific value of *QoS*.

In the above definition, *Class* denotes the concept set of ontology and $hsm(c_p)$ denotes the status set of c_p. In_i in *Input* denotes each input element of *ws*, and *i* denotes the input number in *Input*. Similarly, we can get the meaning of Out_o, $Prec_p$ and Eff_e. The $c_p{:}s_p$ in $Prec_p$ means the concept c_p is in the status of s_p, where *p* denotes the precondition number in *Precondition*. The $c_e{:}rt_e \to ot_e$ means the status change of concept c_e (from rt_e to ot_e). *Input*, *Output*, *Precondition* and *Effect* are called as *IOPE*.

We design service table to store the service basic information. *Service*(*ws_id*, *wsname*, *price*, *time*, *availability*, *reliability*). The *ws_id* represents the service ID, the properties of *price*, *time*, *availability* and *reliability* represent the *QoS* information of services. The *IOPE* information of Web services are designed in the following tables: *Input*(*i_id*, *pid*, *ws_id*), *Output*(*o_id*, *pid*, *ws_id*), *Precondition*(*pr_id*, *pid*, *tid*, *ws_id*), *Effect*(*e_id*, *pid*, *tid_s*, *tid_e, ws_id*). *Para*(*pid*, *pname*, *cid*). *Para* stores the parameter information, and the parameters in *IOPE* are correlated with *Para* through *pid*.

In order to express Web service *IOPE* information from the semantic level, the ontology information is also stored. And we correlate it with the *IOPE* properties of services. We design the following tables: *Class*(*cid*, *cname*), *Relation*(*rid*, *rname*), *Axiom*(*aid*, *cid_1*, *cid_2*, *rid*), *State*(*tid*, *tname*), *ClassState*(*cs_id*, *cid*, *sid*), *StateTrans*(*sp_id*, *tid_1*, *tid_2*). *Class* stores the concepts in ontology. *Relation* stores the relation type of concepts. *Axiom* stores the concept relationships. *State* stores the status information. *ClassState* store the concept status. Each tuple in *Para* is correlated with ontology concept through *cid*, and *IOPE* will be described using ontology concepts. We mainly consider the following semantic relationships: Exact/Plugin/ Subsume/Intersect/Fail. They refer to the relationships of *equivelantOf*, *subClassOf*, *superClassOf*, *intersection* and *fail* respectively. These relationships can be denoted as $c_1 \equiv c_2$、$c_1 \supset c_2$、$c_1 \subset c_2$、$c_1 \cap c_2$、$c_1 \psi c_2$ respectively. The c_1 and c_2 are two concepts. The semantic reasoning relationship between concepts can be got using reasoning machine directly, such as Jena, Pellet, OWL-API, etc..

4 Service Aggregation and Selection

4.1 Web Service Clustering

In this section, we mainly discuss the basic Web service clustering algorithm, and how to realize Web service clustering from the level of interface and capability.

4.1.1 Web Service Clustering Algorithm

On the basis of storing Web services information using RDB, a Web service clustering approach using self-join operation in RDB is elaborated in Algorithm 1. This method does the matching calculation from the semantic level, and it can enhance service clustering efficiency and accuracy.

Algorithm 1. Web service clustering algorithm (*RDBWSClustering*)

```
Input: WS={$ws_1, ws_2, \ldots ws_n$}, Tables{ws, IOPE,
        Para, Relation, Axiom, StateTrans, StatePath}
Output: cluster[]
1: $iope_1, iope_2 \leftarrow IOPE, Cluster \leftarrow \varnothing, vin[n][n],$
   $vout[n][n], vprec[n][n], veff[n][n], mid_{iope}, temp_{iope}$
2: $mid_{iope} \leftarrow iope_1 \bowtie_{(iope1.pid@iope2.pid) \wedge (iope1.ws\_id!=iope2.ws\_id)} iope_2$
3: foreach $i \in mid_{iope}.ws\_id\_1$
4:   foreach $j \in mid_{iope}.ws\_id\_2$
5:     $temp_{iope} \leftarrow \sigma_{ws\_id\_1=i \&\& w\_sid\_2=j}(mid_{iope})$
6: $vin[i][j]/vout[i][j] \leftarrow matchIO(temp_{iope}$, Table{Para,
   Relation, Axiom})
7:     $vprec[i][j] \leftarrow matchPREC(temp_{iope}$, Table{Para,
   Relation, Axiom, StateTrans, StatePath})
8: $veff[i][j] \leftarrow matchEFCT(temp_{iope}$,
   Table{Para, Relation, Axiom, StateTrans,
   StatePath})
9:   endfch
10: endfch
11: assign each service into different clusters,
     $cluster[i] \leftarrow ws_i$
12: $cluster \leftarrow clusterws(cluster, vin, vout, vprec, veff)$
13: return cluster
```

In the above algorithm, *ws_id_1* and *ws_id_2* denote the ID of two Web services. *vin*[*n*][*n*], *vout*[*n*][*n*], *vprec*[*n*][*n*] and *veff*[*n*][*n*] represent the *IOPE* matching matrix of services in *WS*. $temp_{iope}$ and mid_{iope} are the intermediate tables. *StatePath* in the input of Algorithm 1 is the status path table that is formed through *StateTrans*, and it stores the concept status path information. In step 2, $iope_1.pid@iope_2.pid$ means there exist semantic relationships in *Relation* between the concept of $iope_1.pid$ and $iope_2.pid$. In Algorithm 1, it firstly does self-join operation on *IOPE* tables to get mid_{iope}, as shown in step 2. Then it gets *IOPE* matching matrix, as shown in step 3-10. Step 11 is used to allocate services into *n* service clusters. Based on the *IOPE* matching matrix, services are clustered using step 12.

4.1.2 Service Clustering from Interface Level

In the step 6 of Algorithm 1, *matchIO* is used to get matching degree (*vin*[*i*][*j*]) between the interface (*Input* and *Output*) of ws_i and ws_j, as shown in Algorithm 2.

Algorithm 2. Interface matching algorithm (*matchIO*)

```
Input: $temp_{IO}$, Table{Para, Relation, Axiom}
Output: $val_{io}$
1: $val_{sum} \leftarrow 0, num \leftarrow temp_{IO}.count$
2: foreach $i$ 1$\leftarrow num$
3:  $val_{sum} += match(\pi_{pid\_1}(temp_{IO}), \pi_{pid\_2}(temp_{IO}),$
    Table{Para, Relation, Axiom})
4:endfch
5: $val_{io} \leftarrow val_{sum}/num$
6: return $val_{io}$
```

In Algorithm 2, *pid_1*, *pid_2* are the *pid* number of $temp_{IO}$. This algorithm gets the average matching degree of parameters in different tuples of $temp_{IO}$, and it computes the *IO* matching degree between specific services. The *num* in step 2 can be calculated using the aggregation function Count(*) of SQL. In step 3, *match*() is used to compute the matching value between the specific *pid*, as shown in Algorithm 3.

Algorithm 3. Parameter matching algorithm (*match*)

```
Input: pid_1, pid_2, Table{Para, Relation, Axiom}
Output: val
1: Table{AR}, $tcid1 \leftarrow 0, tcid2 \leftarrow 0, trname \leftarrow \varnothing$
2: $AR \leftarrow Axiom \bowtie Relation$
3: $tcid1 \leftarrow \pi_{cid}(\sigma_{pid=pid\_1}(Para)),$
   $tcid2 \leftarrow \pi_{cid}(\sigma_{pid=pid\_2}(Para))$
4: $trname \leftarrow \pi_{rname}(\sigma_{cid\_1=tcid1 \&\& cid\_2=tcid2}(AR))$
5: if(trname==equivalentOf) then $val \leftarrow 1.0$
6: else if(trname==subClassOf) then $val \leftarrow 0.8$
7: else if(trname==superClassOf) then $val \leftarrow 0.6$
8: else if(trname==intersection) then $val \leftarrow 0.4$
9: else $val \leftarrow 0$
10: return val
```

In the above algorithm, *AR* is the intermediate table. The *tcid1* and *tcid2* are two numbers of *cids* in *Para*. Algorithm 3 firstly does the natural join of *Axiom* and *Relation* to get *AR*, as shown in step 2. Then it finds *cid* (*tcid1* and *tcid2*) whose *pid* are equal to *pid_1* and *pid_2* in *Para* respectively. The corresponding *trname* of *cid* (*tcid1* and *tcid2*) in *AR* will be got. Finally, the matching value of *pid_1* and *pid_2* will be got according to *trname*.

4.1.3 Service Clustering from Capability Level

Using the step 2 in Algorithm 1, we can get

Precondition self-join table of Web services. In the step 7 of Algorithm 1, *vprec*[*i*][*j*] can be calculated using Algorithm 4.

Algorithm 4. Precondition matching algorithm (*matchPREC*)

```
Input: $temp_{prec}$, $Table\{Para, Relation, Axiom, StateTrans, StatePath\}$
Output: $val_{pre}$
1: $val_{sum} \leftarrow 0$, $tval_{enty} \leftarrow 0$, $tval_{state} \leftarrow 0$, $tval_{mid} \leftarrow 0$,
   $um \leftarrow temp_{prec}.count$
2: foreach $i$ $1 \leftarrow num$
3:   $tval_{enty} \leftarrow match(\pi_{pid\_1}(temp_{prec}), \pi_{pid\_2}(temp_{prec}),$
     $able\{Para, Relation, Axiom\})$
4:   if($tval_{enty} > 0.4$) then
5:     $tval_{state} \leftarrow matchstate(\pi_{tid\_1}(temp_{prec}),$
       $\pi_{tid\_2}(temp_{prec}), Table\{StateTrans, StatePath\})$
6:     $tval_{mid} \leftarrow tval_{enty} + tval_{state}$
7:   endif
8:   $val_{sum} \leftarrow val_{sum} + tval_{mid}$
9: endfch
10: $val_{pre} \leftarrow val_{sum}/num$
11: return $val_{pre}$
```

The implementation of *match* in step 3 (Algorithm 4) is shown in Algorithm 3. In Algorithm 3, the matching degree $tval_{enty}$ between c_p in $Prec_p=c_p:state_p$ is calculated firstly. When the value is larger than the threshold, *matchstate* (Algorithm 5) will be used to calculate the matching degree $tval_{state}$ between $state_p$. Then it gets the average value of $tval_{enty}$ and $tval_{state}$.

Algorithm 5. Status matching algorithm (*matchstate*)

```
Input: $tid\_1$, $tid\_2$, $Table\{StateTrans, StatePath\}$
Output: $val$
1: Table$\{temp_{sp}\}$, $snum$, $enum$, $palength \leftarrow 0$,
   $pathnum \leftarrow StatePath.num$
2: if($tid\_1 == tid\_2$) return 1
3: foreach $i$:$1 \leftarrow pathnum$
4:   $temp_{sp} \leftarrow \sigma_{sp\_id=i}(StatePath)$
5:   $snum \leftarrow \pi_{num}(\sigma_{tid=tid\_1}(temp_{sp}))$
6:   $enum \leftarrow \pi_{num}(\sigma_{tid=tid\_2}(temp_{sp}))$
7:   if($snum$!=0 && $enum$!=0) then
8:     $palength \leftarrow enum - snum$; break
9:   endif
10: endfch
11: $val \leftarrow 1/(|palength|+1)$
12: return $val$
```

Algorithm 5 gives the calculation process of getting the matching degree of two specific *tid* according to *StateTrans* and *StatePath*. In Algorithm 5, if *tid_1* and *tid_2* are equal, then return 1. Otherwise, we judge each path in *StatePath* to get the status path table of specific path. Then we get the status number *snum* and *enum* of *tid_1* and *tid_2* respectively, as shown in step 5-6. According to *snum* and *enum*, *val* will be got, and return.

A concept in ontology has different status, and the different status transformation path can be constructed correspondently. There exist different semantic reasoning relationships between concepts, and they are denoted as $c_1 \equiv c_2$、$c_1 \supset c_2$、$c_1 \subset c_2$、$c_1 \cap c_2$、$c_1 \psi c_2$. Using the above concept reasoning relationships, we can get the following concept status transformation rules.

$(c_i \equiv c_j) \wedge (c_i:t_m) \wedge (c_i:t_n) \wedge (c_i:t_m \rightarrow t_n) \Rightarrow (c_j:t_m) \wedge (c_j:t_n) \wedge (c_j:t_m \rightarrow t_n)$

$(c_i \supset c_j) \wedge (c_i:t_m) \wedge (c_i:t_n) \wedge (c_i:t_m \rightarrow t_n) \Rightarrow (c_j:t_m) \wedge (c_j:t_n) \wedge (c_j:t_m \rightarrow t_n)$

As shown in Figure 2, c_5 has the following status: t_1、t_2、t_3 and t_4. The corresponding status path includes $c_5:t_1 \rightarrow t_3$ and $c_5:t_1 \rightarrow t_2 \rightarrow t_4$. We can also get $c_6:t_1 \rightarrow t_3$, $c_6:t_1 \rightarrow t_2 \rightarrow t_4$, and $c_{13}:t_1 \rightarrow t_3$, $c_{13}:t_1 \rightarrow t_2 \rightarrow t_4$ through status transformation rules.

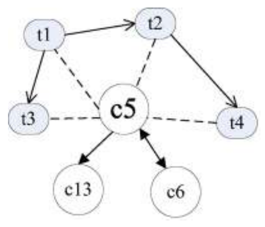


Figure 2. Ontology concept status

The concrete process of getting *Effect* of *ws* is similar to *matchPREC*. Due to limited spaces, we will not elaborate it in detail.

Using above approach, the *CA* matching matrix of Web services can be got. Thus the services in *WS* will be clustered. A field named *as_id* will be added into *Service* table to denote the cluster that the service belongs to, and it is defined as *ASService*(*ws_id*, *wsname*, *as_id*). Services whose *as_id* are same realize same function and have different *QoS* values. When to add a new service, the matching calculation between the new service and each tuple in the *Service* table will be done firstly. When the matching value is larger than the threshold, the service will be inserted into the corresponding service cluster.

4.2 Service Aggregation

This section mainly discusses the concrete process of how to realize service aggregation.

4.2.1 Abstract Service Extraction

On the basis of service clustering, this section mainly introduces how to extract the abstract service (*AS*) from different service clusters, and thus to make foundation for service aggregation. Abstract service can be seen as the representative of particular service cluster, and it does not refer to the actual service. The definition of abstract service is similar to *ws* in Definition 1. But the *QoS* of *AS* refers to the scope of *QoS* property. We define it as $QoS=\{\{QosName_q, Unit_q, Min_q, Max_q\}, QosName_q, Unit_q \in Class, q=1,2...qnum\}$. The parameters express the corresponding name, unit, minimum and maximum.

Algorithm 6. Abstract service extraction algorithm (*ExtractAS*)

```
Input: Cluster[], Table{ws, IOPE, Para, Relation, Axiom}
Output: as
1: $IWS \leftarrow \emptyset$, $vinum \leftarrow 0$, $PidRela \leftarrow \emptyset$
2: Create table TASInput(pid, inum, as_id)
3: foreach i: 1←Cluster.num
4:   $IWS=\pi_{ws\_id}(\sigma_{as\_id='i'}(ws))$
5:   foreach j: 1←IWS.ws_id
6:     foreach $k \in \pi_{pid}(\sigma_{ws\_id='j'}(Input))$
7:       if(tuple<pid=k, as_id =i>∈TASInput)
8:         Update TASInput set inum=inum+1 where pid=k
9:       else TASInput.insert(k, 1, i)
10:    endfch
11:  endfch
12: endfch
13: PidRela←GenePidRela(TASInput, Cluster[], Table{Para, Relation,Axiom})
14: foreach i: 1←Cluster.num
15:  foreach tuple $tu \in \sigma_{as\_id='i'}(PidRela)$
16:    if(tu.rid= =1 || tu.rid= =2) then
17:      $tiopenum\_1 \leftarrow \pi_{tnum}(\sigma_{tpid='tu.pid1' \& as\_id='i'}(TASInput))$
18:      $tiopenum\_2 \leftarrow \pi_{tnum}(\sigma_{tpid='tu.pid2' \& as\_id='i'}(TASInput))$
19:        if((tiopenum_1<=tiopenum_2) && (tiopenum_2/ Cluster[i].length>0.3)) then
20:          Update TASInput Set inum=inum+tiopenum_1 where pid='tu.pid2' and as_id='i'
21:          Delete from TASInput where tpid='tu.pid1' and as_id='i'
22:      else Update tu.pid1 of TASInput
23:    endif
24:  endfch
25: endfch
26: $ASInput \leftarrow \pi_{pid,as\_id}(TASInput)$
27: Similar to step 2-26 to get ASOutput/ ASPrecondition/ ASEffect
28: foreach i: 1←Cluster.num
29:   $TQOS \leftarrow \sigma_{sc\_id='i'}(ws)$
30:   foreach q: 1←qnum
31:     $as_i.QosName_q \leftarrow \{TQOS.\mathrm{Min}(QosName_q), TQOS.\mathrm{Max}(QosName_q)\}$
32:   endfch
33:   $as_i \leftarrow ASInput/ASOutput/ASPrecondition/ASEffect$
34: endfch
35: return as
```

Algorithm 6 gives the process of how to extract the abstract service from each service cluster named *Cluster*[]. For example, the step 2-12 realizes how to generate *TASInput* according to *ws.Input*. The tuples belongs to certain service cluster in *ws* will be found firstly, then they will be inserted it into *IWS*, as seen in step 4. The tuples in *IWS* will be judged, and it will find the tuples which have same *ws_id* in *Input*. When *pid* is in *TASInput*, its *num* will plus 1. Otherwise, the tuple will be inserted into *TASInput* and assigned *num* to 1. The step 13 uses Algorithm 7 (*GenePidRela*) to generate concept semantic relationship table named *PidRela*. Step 14-25 gives the process of updating *TASInput* according to the semantic relationships and the number of two *pid* in *PidRela*. When there exists the relationship of *subClassof* or *superClassof* (*rid*=1 or *rid*=2) between *pid* of concepts, then it compares the number of two concepts. The concept number with fewer occurrences will be added onto the concept number with more occurrences. Then it deletes the tuple with fewer occurrences, as seen in step 16-26. The properties of *pid* and *as_id* in *TASInput* are extracted to form *ASInput*. The tables of *ASOutput*, *ASPrecondition* and *ASEffect* are got using the same approach. Step 29-32 is used to assign *QoS* scope of services in the same service cluster to the *QoS* property of the corresponding abstract service. The *QoS* information of *AS* will be got and the *AS* table is designed: *AS*(*as_id*, *name*, *price_min*, *price_max*, *time_min*, *time_max*, …).

Algorithm 7. Getting concept semantic relationship algorithm (*GenePidRela*)

```
Input: TIOPE, Cluster[], Table{Para, Relation, Axiom}
Output: PidRela
1: cid1, cid2, trid←0
2: Create Table PidRela(pid1, pid2, rid, as_id)
3: foreach i: 1←Cluster.num
4:   $STAB \leftarrow \sigma_{as\_id='i'}(TIOPE)$
5:   if(STAB.count(*)<2) then goto step 3
6:   foreach $pid1, pid2 \in \pi_{pid}(STAB)$
7:     $cid1 \leftarrow \pi_{cid}(\sigma_{pid='pid1'}(Para))$
8:     $cid2 \leftarrow \pi_{cid}(\sigma_{pid='pid2'}(Para))$
9:     $rid \leftarrow \pi_{rid}(\sigma_{cid1='cid1' \& cid2='cid2'}(Axiom))$
10:    PidRela.insert(pid1,pid2,rid,i)
11:  endfch
12: endfch
13: return PidRela
```

Algorithm 7 gives the process of getting the semantic relationships between the parameters of specific abstract service. All the service clusters will be judged in turn, the tuples will be extracted and inserted into *STAB* according to its ID, as seen in step 4. If the number of tuples in *STAB* is less than 2, it means there is no parameter to be merged. Otherwise, it will get two *pid* randomly, and find the corresponding *cid* in *Para* according to *pid*. The *rid* between concepts in *Axiom* will be found according to *cid*, and then inserted into *PidRela*, as seen in step 6-10.

4.2.2 Abstract Service Execution Relationship Extraction

Based on the abstract services from different service clusters, this section introduces how to extract the execution relationship between abstract services in terms of interface and capability.

Algorithm 8. AS service execution relationship extraction algorithm (*GetASRelation*)

```
Input: $AS=\{as_1, as_2, \ldots as_n\}$, Tables{ASIOPE, Para, Relation, Axiom, StateTrans, StatePath}
Output: ASRelation
1: $table_{IO}$, $table_{CA}$, $temp_{IO} \leftarrow \varnothing$, inum, tnum, iasid,
   oasid, $val_c$, onum ← 0
2: Create Table ASRelation(as_s, as_e)
3: $table_{IO} \leftarrow ASOutput \bowtie_{(ASOutput.pid@ASInput.pid) \&\& (ASOutput.as\_id!=ASInput.as\_id)} ASInput$
4: foreach tuple $tu \in table_{IO}$
5:   oasid ← tu.as_id_o, iasid ← tu.as_id_i
6:   $onum \leftarrow (\pi_{as\_id=oasid} ASOutput)$.count(*)
7:   $inum \leftarrow (\pi_{as\_id=iasid} ASInput)$.count(*)
8:   $temp_{IO} \leftarrow \sigma_{as\_id\_o=oasid \&\& as\_id\_i=iasid}(table_{IO})$
9:   $tunum \leftarrow temp_{IO}$.count(*)
10:  if(tunum= =onum && tunum= =inum) then
11:     ASRelation.insert('as'+oasid, 'as'+iasid)
12:  endif
13: endfch
14: $table_{CA} \leftarrow ASEffect \bowtie_{(ASEffect.pid@ASPrecondition.pid) \&\& (ASEffect.tid\_e\# ASPrecondition.tid) \&\& (ASEffect.as\_id!=ASPrecondition.as\_id)} ASPrecondition$
15: foreach tuple $tu \in table_{CA}$
16:   $val_c \leftarrow$ match(tu.pid_e, tu. pid_p, Table{Para,
      Relation, Axiom})
17:   if($val_c$>0.4) then
18:     $val_s \leftarrow$ matchstate(tu.tid_e, tu.tid_p,
        Table{StateTrans, StatePath})
19:     if(($val_c$+$val_s$)/2>0.5) then
20:       ASRelation.insert('as'+tu.as_id_e, 'as'+
          tu.as_id_p)
21:     endif
22:   endif
23: endfch
24: return ASRelation
```

In step 6, *onum* can be got through the aggregation function Count(*) in SQL. The step 2 in Algorithm 8 is used to create the service execution dependency table (*ASRelation*). The service execution matching pair about service interface will be extracted firstly using step 3-13. Then it gets the service matching pair from the aspect of capability using step 14-23. *ASOutput* and *ASInput* will be joined to get $table_{IO}$ about service interface. The tuples in $table_{IO}$ will be judged in turn and the corresponding service execution dependency matching pair will be inserted into *ASRelation*. Similarly, it will do join operation from the aspect of service capability. Finally, return *ASRelation*.

Supposing S_1 and S_2 are two Web services, SO_{1m} expresses the parameter numbers in the *Output* of S_1, and SI_{2n} expresses the parameter numbers in the *Input* of S_2. The *num* is the tuple number of *as_id_o* and *as_id_i* in $table_{IO}$. (1) $SO_{1m}=SI_{2n}=num$; (2) $SO_{1m}=num$ and $SI_{2n}\neq num$; (3) $SO_{1m}\neq num$ and $SI_{2n}=num$. When one of the above three conditions is met, it is included that there exist *IO* execution relationship between S_1 and S_2. As shown in Figure 3, p_1-p_n denotes the *IO* elements between S_1 and S_2.

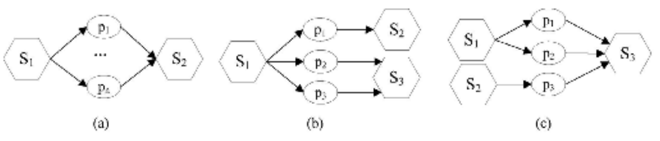


Figure 3. Service IO relationship

4.2.3 Web Service Aggregation Using Join Operation

Based on the abstract services (as_1-as_n) execution dependency relationship between services (ws_1-ws_m), this section mainly introduces how to organize service clusters and thus to realize service aggregation.

Algorithm 9. Generating *ASExePath* (*GeneServiceExePath*)

```
Input: ASRelation
Output: ASExePath
1: i, j, pnum ← 1, fnum ← 2
2: $ASPA_1 \leftarrow$ ASRelation, pnum ← 1
3: repeat
4:   $ASPA_i \leftarrow ASPA_{i-1} \bowtie_{ASPAi-1.as\_e=ASRelation.as\_s} ASRelation$
5:   foreach tuple $tu \in ASPA_i$
6:     id ← 'p'+j;
7:     foreach k:1 ← $TB_i$.column
8:       ASExePath.insert(<id,tu.tid_'k', pnum>)
9:       pnum ← pnum+1
10:    endfch
11:    j ← j+1, pnum ← 1
12:  endfch
13:  pnum ← 1, fnum ← fnum+1
14: until $ASPA_i$ has no tuples
15: return ASExePath
```

Algorithm 9 gives the process of how to generate status path table (*ASExePath*) based on *ASRelation*. It does the self-join operation of *ASRelation* in turn. Each tuple will be inserted into *ASExePath*. When i=1, $ASPA_1$=*ASRelation*. It will be stored into *ASExePath* through *ASPathNum*. When i=2, *ASRelation* will be done self-join operation. Then we store each path into *ASExePath*.

4.3 Web Service Selection

On the basis of aggregating services, this section mainly introduces how to select services to meet users' functional and *QoS* requirements. We define the users' request in **Definition 2**.

Definition 2. Request(*RE*): *RE*={*ReInput*, *ReOutput*, *ReCapa*, *ReQoS*}

$ReInput=\{rinput_i, rinput_i \in Class, i=1,2,\ldots, rinum\}$.

$ReOutput=\{routput_o, routput_o \in Class, o=1,2,\ldots, ronum\}$.

$ReCapa=\{reff_c, c=1, 2,\ldots, efnum\}$, $reff_c=\{rentity_c: rprestate_c \rightarrow rpoststate_c, rentity_c \in Class, rprestate_c, rpoststate_c \in \text{hsm}(rentity_c)\}$.

$ReQoS=\{\{rqosname_q, rvalue_q, runit_q\}, rqosname_q, reunit_q \in Class, q=1,2, \ldots, rqnum\}$.

4.3.1 Service Selection for Functional Requirements

On the basis of aggregating services, this section mainly introduces how to construct service execution path according to users' functional requests. It selects abstract services from aspects of interface and capability, and thus to select a set of services with correlations to meet users' requirements in further.

In Algorithm 10, the elements in *RE.ReInput* are judged firstly, and the corresponding *as_id* will be found in *ASInput* according to *RE.ReInput*. The *as_id* will be inserted into as_{st}, and it deletes the duplicate services in as_{st}, as shown in step 2-11. Then it judges as_i in as_{st} to get whether its output includes the elements in *ReOutput* or not. If it includes the elements, we add as_i into *cas* and update *ReOutput*, as seen in step 13-16. When *ReOutput* is not null, the tuple whose *as_s* is equal to as_i in *ASPathNum* will be inserted into $table_{ASPath}$. Then each tuple *tu* in $table_{ASPath}$ will be judged. If the output of *tu.as_e* includes the elements of *ReOutput*, the tuples of specific *pa_id* in *ASExePath* will be inserted into $table_{PathIF}$. The elements of column *as* in $table_{PathIF}$ will be taken out in turn to construct service execution path. Then it adds the path into *cas* and update *RE.ReOutput*, as seen in step 17-27. Step 30-33 is used to add the service execution path into *cas*, and the path includes the services which has correlation with the service in as_{ca}. Finally, return *cas*. The *cas* stores the abstract service execution path, and it is denoted as <*as_s*, *as_e*>. When *as_s*=*as_e*, it refers to atomic service. When *as_s*≠*as_e*, <*as_s*, *as_e*> expresses a set of services with correlations.

Algorithm 10. Service selection algorithm for functional requirements (*RWSFunSelect*)

```
Input: RE, Table{ASIOPE, ASRelation, ASPathNum,
        ASExePath, Para, Relation, Axiom}
Output: cas
1: cas, rpath, $as_{io}$, $table_{ASPath}$, $as_{ca}$←∅, pidnum,
   $val_{in}$←0, i←1, $list_{as}$, $table_{PathIF}$
2: foreach $Rin_i \in RE.ReInput$
3:   pidnum←getpid($Rin_i$, Table{Class, Para})
4:   foreach tuple $tu \in ASInput$
5:     $val_{in}$←match(pidnum, tu.pid, Table{Para,
       Relation, Axiom})
6:     if($val_{in}$>0.5) then
7:       $as_{io}$.add('as'+tu.as_id)
8:     endif
9:   endfch
10: endfch
11: $as_{io}$←DelDuplicate($as_{io}$)
12: Similar to step 2-11, and get $as_{ca}$ whicn can realize
    RE.ReCapa
13: foreach $as_i \in as_{io}$
14:   if($as_i$.Output⊆RE.ReOutput) then
15:     cas←cas∪ <$as_i$, $as_i$>
16:     RE.ReOutput←RE.ReOutput-$as_{id}$.Output
17:   if(RE.ReOutput!=null) then
18:     $table_{ASPath} \leftarrow \sigma_{as\_s=\ asi}(ASPathNum)$
19:     foreach tuple $tu \in table_{ASPath}$
20:       if(tu.as_e.Output⊆RE.ReOutput) then
21：        $table_{PathIF} \leftarrow \sigma_{pa\_id=tu.pa\_id}(ASExePath)$
22:         foreach tuple $tupath \in table_{PathIF}$
23:           $list_{as}$←$list_{as}$∪ tupath.as
24:         endfch
25:         cas←cas∪<$list_{as}$>
26:         RE.ReOutput←RE.ReOutput-tu.as_e.Output
27:     endfch
28:   endif
29: endfch
30: foreach $as_c \in as_{ca}$
31:   find the tuple tu with as_s=$as_c$ || as_e=$as_c$
32:   cas←cas∪<tu.as_s, tu.as_e>
33: endfch
34: return cas
```

4.3.2 Service Selection for QoS Requirements

Based on the abstract service execution paths, this section mainly introduces how to select services to meet users' *QoS* requirements in further.

On the basis of the abstract service execution paths, the above algorithm selects services to meet *QoS* requirements in further. It judges each abstract service execution path in *cas*, and the atomic service which includes one service will be handled firstly. Then it deals with the service execution path which includes multiple services. When there is one service in abstract

Algorithm 11. Service selection algorithm for QoS (*RWSQoS*)

```
Input: $RE$, $cas$, $ASIOPE$, $ASPathNum$, $ASExePath$, $ASService$, $IOPE$
Output: $RWS$
1: $RWS \leftarrow \varnothing$, $table_{ws}$, $table_{in}$, $table_{out}$, $list_{aws}$, $list_{cws} \leftarrow \varnothing$, $table_{ASPath}$
2: foreach $cas_i = <as\_s, \ldots, as\_e> \in cas$
3:   if($cas_i.length$=2 && $as\_s$= =$as\_e$) then
4:     $table_{ws} \leftarrow \sigma_{\text{'as'}+as\_id=as\_s}(ASService)$
5:     foreach tuple $tu \in table_{ws}$
6:       $table_{in} \leftarrow \sigma_{ws\_id=tu.ws\_id}(Input)$
7:       $table_{out} \leftarrow \sigma_{ws\_id=tu.ws\_id}(Output)$
8:     if($table_{in}.pid \subseteq RE.ReInput$&&$table_{out}.pid \subseteq RE.ReOutput$)
9:         $list_{aws}$.add($tu.ws\_id$)
10:      endif
11:    endfch
12:    if($list_{aws}$.length= =0) then
13:      $RWS$.add($GetWSIDSeman$($as\_s$, $ASService$, $Input$, $Output$))
14:    else if($list_{aws}$.length= =1) then $RWS$.add ($list_{aws}.ws\_id$)
15:    else foreach $ws\_id \in list_{aws}$
16:        if($matchqos$($RE.ReQoS$, $ws\_id$, $Service$)>0.5) then
17:        $RWS$.add($tu.ws\_id$)
18:      endfch
19:  else
20:    get $pa\_id$ of $<as\_s, \ldots, as\_e>$ in $ASPathNum$
21:    $list_{cws} \leftarrow matchCompoWS(pa\_id, ASExePath)$
22:    using step 15-18 to get service with proper $QoS$ in $list_{cws}$, and store into $RWS$
23: endfch
24: return $RWS$
```

service execution path, it will find service with proper *QoS* in service cluster directly, as seen in step 3-18. The services whose *Input* and *Output* are matched from the grammar level will be selected firstly, as seen in step 4-11. When there is no this kind of services, it will select services from the semantic level and add it into *RWS*. Step 19-22 gives the process of dealing with the situation of there are multiple abstract service execution paths. Through Algorithm 14 (*matchCompoWS*), it can find services which have *Exact* relationship with abstract services. These services can construct the service execution path named $list_{cws}$, and the service with proper *QoS* values will be added into *RWS*. Finally, return *RWS*.

The *GetWSIDSeman* in step 13 of Algorithm 11 is used to calculate concept matching degree from the semantic level. The realization of finding service *id* of specific abstract service *as_s* in particular service cluster is shown in Algorithm 12.

Algorithm 12. Service selection algorithm from semantic level (*GetWSIDSema*)

```
Input: $as$, $ASService$, $Input$, $Output$
Output: $ws\_id$
1: $ws\_id \leftarrow \varnothing$, $table_{ws}$, $table_{in}$, $table_{out}$, $list_{ws\_id}$
2: $table_{ws} \leftarrow \sigma_{\text{'as'}+as\_id=as}(ASService)$
3: foreach tuple $tu \in table_{ws}$
4:   $table_{in} \leftarrow \sigma_{ws\_id=tu.ws\_id}(Input)$
5:   $table_{out} \leftarrow \sigma_{ws\_id=tu.ws\_id}(Output)$
6:   foreach tuple $tuin \in table_{in}$
7:     foreach element $ele_{in} \subseteq RE.ReInput$
8:       $val_{in} \leftarrow match(tuin.p\_id, getpid(ele_{in}))$
9:       use bipartite graph algorithm to get input matching value $TRE_{in}$ between $table_{in}$ and $RE.ReInput$
10:  endfch
11:  Same to step 6-10 to get output matching value $TRE_{out}$ between $table_{out}$ and $RE.ReOutput$
12:  if(($TRE_{in}$+ $TRE_{out}$)/2>0.6) then
13:    $list_{ws\_id}$.add($tu.ws\_id$, ($TRE_{in}$+ $TRE_{out}$)/2)
14:  endif
15: endfch
16: find the $ws\_id$ whose matching value is the biggest in $list_{ws\_id}$
17: return $ws\_id$
```

The above algorithm is used to find services which have semantic relationships with specific abstract service *as*. It firstly extracts services in certain service cluster of abstract service in *ASService* to construct $table_{ws}$, as seen in step 2. Then it judges each tuple *tu* in $table_{ws}$, and gets the Input and Output information of *tu.ws_id* in *Input* and *Output* to construct $table_{in}$ and $table_{out}$, as seen in step 3-5. The bipartite graph algorithm is used to calculate the matching degree TRE_{in} between $table_{in}$ and *RE.ReInput*, TRE_{out} between $table_{out}$ and *RE.ReOutput*, as seen in step 6-11. The services with *ws_id* will be added into $list_{ws\_id}$ when the threshold is larger than the average value of TRE_{in} and TRE_{out}, as shown in step 12-15. Finally, it finds the service with *ws_id* whose matching value is the largest in $list_{ws\_id}$.

The *matchqos* in step 16 of Algorithm 11 is used to calculate the matching degree between *RE.ReQoS* and the *QoS* of specific service with *ws_id*. The concrete realization process is shown in Algorithm 13.

Algorithm 13 is used to calculate the matching degree between users' *QoS* request and the *QoS* information of specific service. It finds the specific service in *Service* according to *ws_id* firstly, as seen in step 2-3. Then it compares the service *QoS* with the users' request in turn to see whether it meets users' request or not. If it meets the condition, the corresponding matching value is assigned to 1. Otherwise, it is assigned to 0, as seen in step 4-11. Finally, it gets the average matching value of all the *QoS* properties.

Algorithm 13. Service QoS matching algorithm (*matchqos*)

```
Input: ReQoS, ws_id, Service
Output: $val_{qos}$
1: $val_{qos}$←0, num←ReQoS.num, revalq[num],
   $sum_{qos}$←0
2: foreach tuple tu∈Service
3:   if(tu.ws_id==ws_id) then
4:     foreach q∈{1,2, …, num}
5:       if(ReQoS.rqosname$_q$= =availability ||
         reliability)
6:         if(tu.availability>=rvalue$_q$ || tu.reliability>=
           rvalue$_q$)
7:           revalq[q] ←1.0
8:         if(tu.Price<=rvalue$_q$ || time<=rvalue$_q$) then
9:           revalq[q] ←1.0
10:      endif
11:    endfch
12:  endif
13: endfch
14: foreach q∈{1,2, …, num}
15:   $sum_{qos}$←$sum_{qos}$+ revalq[q]
16: endfch
17: $val_{qos}$←$sum_{qos}$/num
18: return $val_{qos}$
```

Algorithm 14. Composite service selection algorithm (*matchCompoWS*)

```
Input: pa_id, ASExePath, ASService
Output: $list_{cws}$
1: $list_{cws}$←∅, $table_{ASPath}$, $as_{first}$, $list_{first}$, asnum←2,
   astotalnum
2: $table_{ASPath}$←$\sigma_{pa\_id=pa\_id}$(ASExePath)
3: astotalnum←$table_{ASPath}$.count(*)
4: $as_{first}$←$\pi_{as}$ ($\sigma_{num=1}$($table_{ASPath}$))
5: Same to step 4-18 in Algorithm 11 to get $list_{first}$ of
   the first service $as_{first}$
6: foreach ws ws_id∈$list_{first}$
7:   $list_{cws}$.add(ws_id)
8:   $table_{out}$←$\sigma_{ws\_id=ws\_id}$(Output)
9:   while(asnum<astotalnum)
10:    $as_{next}$←$\pi_{as}$ ($\sigma_{num=asnum}$($table_{ASPath}$))
11:    $table_{ws}$←$\sigma_{\text{'as'}+as\_id=as\_next}$(ASService)
12:    foreach tuple tu∈$table_{ws}$
13:      $table_{in}$←$\sigma_{ws\_id=tu.ws\_id}$(Input)
14:      find the tu.ws_id which is exactly matched
         between $table_{out}$ and $table_{in}$
15:    endfch
16:    $list_{cws}$.get(ws_id).add(tu.ws_id)
17:    asnum++
18:  end while
19: endfch
20: return $list_{cws}$
```

The above algorithm gives the process of finding the services with proper *QoS* values in service clusters of specific abstract services in the service execution path. It finds the abstract service execution path of specific *pa_id* in *ASExePath* firstly. Services in the path will be stored into $table_{ASPath}$, as shown in step 2. Then the first service as_{first} in $table_{ASPath}$ will be found. Step 4-18 in Algorithm 11 will be used to find $list_{first}$ with proper *QoS* of as_{first}, as seen in step 3-4. Each service *ws_id* in $list_{first}$ is judged in turn to get the $table_{out}$ (output information) of *ws_id*. Then it gets as_{next} which is after as_{first} in $table_{ASPath}$. In the service cluster of as_{next}, services whose *Input* are matched with $table_{out}$ will be found and added into the service execution path $list_{cws}$ of *ws_id*, as seen in step 6-19. Through the above method, it finds the service whose interface is exactly matched with the services in the abstract service execution path of *ws_id*. Finally, return $list_{cws}$.

5 Case Study

Table 1 shows the *IOPE* information of Web services (*ws1-ws15*).

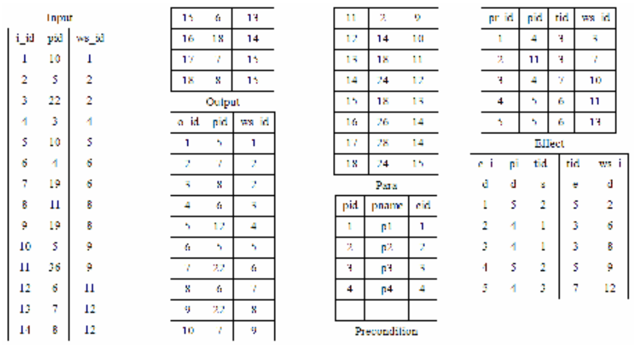
Table 1. Web service examples

WSName	Input	Output	Precondition	Effect
ws1	p10	p5	-	-
ws2	p5,p22	p7, p8	-	p5: t2→t5
ws3	-	p6	p4:t3	-
ws4	p3	p12	-	-
ws5	p10	p5	-	-
ws6	p4,p19	p22	-	p4:t1→t3
ws7	-	p6	p11:t3	-
ws8	p11,p19	p22	-	p4:t1→t3
ws9	p5,p36	p7, p2	-	p5: t2→t5
ws10	-	p14	p4:t7	-
ws11	p6	p18	p5:t6	
ws12	p7,p8	p24	-	p4:t3→t7
ws13	p6	p18	p5:t6	
ws14	p18	p26,p28	-	-
ws15	p7,p8	p24	-	-

In Table 1, *p1~p28* represent the parameters in *Para*. *t1~t7* represent the states in *State*. *p11*: *t3* means the concept *p11* is in the status of *t3*, and *p4*: *t1→t3* denotes the status change of *p4* (from *t1* to *t3*). In practice, the name and *IOPE* of services can be got easily from service description documents, such as .wsdl, .owls, etc.

The *IOPE* of *ws1*-*ws15* in Table 1 is shown in Figure 4. *Para* store the parameter information. The parameters in *IOPE* are correlated with *Para* through *pid*.

The tables in Figure 5 store the ontology information of *ws1-ws15* in Table 1.



Input

i_id	pid	ws_id
1	10	1
2	5	2
3	22	2
4	3	4
5	10	5
6	4	6
7	19	6
8	11	8
9	19	8
10	5	9
11	36	9
12	6	11
13	7	12
14	8	12
15	6	13
16	18	14
17	7	15
18	8	15

Output

o_id	pid	ws_id
1	5	1
2	7	2
3	8	2
4	6	3
5	12	4
6	5	5
7	22	6
8	6	7
9	22	8
10	7	9
11	2	9
12	14	10
13	18	11
14	24	12
15	18	13
16	26	14
17	28	14
18	24	15

Para

pid	pname	cid
1	p1	1
2	p2	2
3	p3	3
4	p4	4

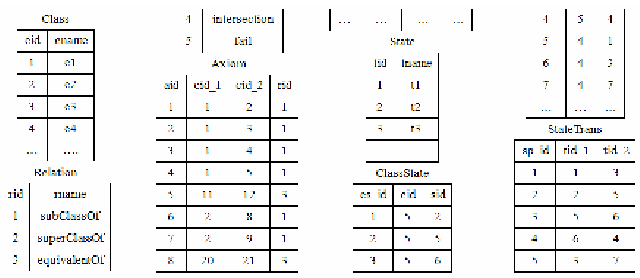
Precondition

pr_id	pid	rid	ws_id
1	4	3	3
2	11	3	7
3	4	7	10
4	5	6	11
5	5	6	13

Effect

e_id	pid	tids	rid	ws_id
1	5	2	5	2
2	4	1	3	6
3	4	1	3	8
4	5	2	5	9
5	4	3	7	12

Figure 4. IOPE information of Web services



Class

cid	cname
1	c1
2	c2
3	c3
4	c4
…	….

Relation

rid	rname
1	subClassOf
2	superClassOf
3	equivalentOf
4	intersection
5	tail

Axiom

aid	cid_1	cid_2	rid
1	1	2	1
2	1	3	1
3	1	4	1
4	1	5	1
5	11	12	3
6	2	8	1
7	2	9	1
8	20	21	3
…	…	…	…

State

tid	tname
1	t1
2	t2
3	t3

ClassState

cs_id	cid	sid
1	5	2
2	5	5
3	5	6
4	5	4
5	4	1
6	4	3
7	4	7
…	…	…

StateTrans

sp_id	tid_1	tid_2
1	1	3
2	2	5
3	5	6
4	6	4
5	3	7

Figure 5. Ontology information

As shown in **Definition 1**, we use concepts and concept status to express the interface and capability of Web services. The ontology information of web services in Table 1 is shown in Figure 6.

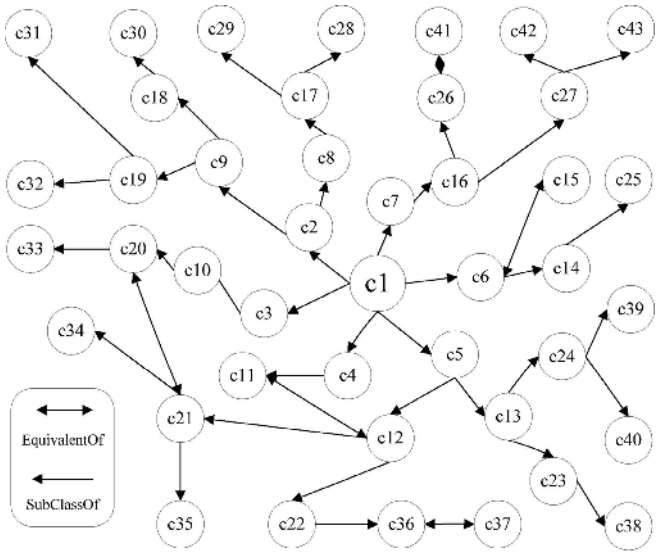


Figure 6. Ontology concept relationships

Through the above method, *WS*={ws_1,ws_2,….ws_{15}}in Table 1 will be clustered into different service clusters in terms of *IO* and *CA*. We can get *cluster*[1]={ws_1, ws_4, ws_5}, *cluster*[2]={ws_2, ws_9}, *cluster*[3]={ws_{12}, ws_{15}}, *cluster*[4]={ws_6, ws_8}, *cluster*[5] ={ws_3, ws_7, ws_{10}}, *cluster*[6]={ws_{11}, ws_{13}}, *cluster*[7]={ws_{14}}. A field named *as_id* will be added to denote the cluster that the service belongs to, as shown in Table 2.

Table 2. ASService

ws_id	wsname	as_id
1	ws1	1
2	ws2	2
…	…	
15	ws3	3

Services whose *as_id* is same realize same function and have different *QoS* values. When to add a new service, the matching calculation between the new service and each tuple in the table of *Service* will be done firstly. When the matching value is larger than the threshold, the service will be inserted into the corresponding service cluster.

5.1 Abstract Service Extraction

The *TASInput* will be generated using step 3-12 in Algorithm 6. When *Cluster.num*=1, *IWS* will be got through step 4, as seen in Table 3.

Table 3. IWS

ws_id
1
4
5

When *IWS.ws_id*=1, the tuple with *ws_id*=1 in *Input*: *tuple<pid*=10, *as_id=1>*$\notin$*TASInput*, EXE(*TASInput*. insert(10, 1, 1)).

When *IWS.ws_id*=4, the tuple with *ws_id*=4 in *Input*: *tuple<pid*=3, *as_id =1>*$\notin$*TASInput*, EXE(*TASInput*. insert(3, 1, 1)).

When *IWS.ws_id*=5, the tuple with *ws_id*=5 in *Input*: *tuple<pid*=10, *as_id =1>* $\in$ *TASInput*, EXE(Update *TASInput* set *inum*=2 where *pid*=10).

TASInput can be generated using the above method and it is shown in Table 4.

Table 4. TASInput

pid	inum	as_id
10	2	1
3	1	1
5	2	2
22	1	2
36	1	2
7	2	3
8	2	3
4	1	4
11	1	4
19	2	4
6	2	6
18	1	7

When *Cluster.num*=1 in Algorithm 6, we can get *STAB* through step 4 and it is shown in Table 5.

Table 5. STAB

pid	inum	as_id
10	2	1
3	1	1

We can get *cid1*=10, *cid2*=3 and *rid*=2 (*superClassOf*), then the tuple (10, 3, 2, 1) will be inserted into *PidRela*. *PidRela* can be generated based on *TASInput* using above method, and it is shown in Table 6.

Table 6. PidRela

pi_id	pid1	pid2	rid	as_id
1	10	3	2	1
2	5	22	5	2
3	5	36	5	2
4	22	36	1	2
5	7	8	5	3
6	4	11	1	4
7	4	19	5	4
8	11	19	5	4

Through the step 14-25 in Algorithm 6, *TASInput* can be updated using *PidRela*.

When *Cluster.num*=1, it gets *tu*(1, 10, 3, 2, 1)$\in$ $\sigma_{as\_id=1}$(*PidRela*). Then (*tu.rid*=2) $\wedge$ (*tiopenum_1*=2) $\wedge$ (*tiopenum_2*=1) $\wedge$ (*tiopenum_1*>*tiopenum_2*)$\Rightarrow$EXE(Update *TASInput* Set *inum*=3 where *pid*=10 *&& as_id*=1, Delete from *TASInput* where *pid*=3 *& as_id*=1). Then *TASInput* will be updated.

We can get *TASInput* using the above method, as shown in Table 7.

Table 7. TASInput

pid	inum	as_id
10	3	1
5	2	2
22	2	2
7	2	3
8	2	3
4	2	4
19	2	4
6	2	6
18	1	7

It gets the property of *pid* and *as_id* in *TASInput*, and generates *ASInput*. Similarly, it generates *ASOutput*, *ASPrecondition* and *ASEffect* using the same method. The *IOPE* of abstract service in different service clusters of *ws1*-*ws15* in Table 1 are shown in Table 8 to Table 11.

Through step 29-32 in Algorithm 6, the *QoS* information of *AS* will be got and it is shown in Table 12.

Table 8. ASInput

pid	as_id
10	1
5	2
22	2
7	3
8	3
4	4
19	4
6	6
18	7

Table 9. ASOutput

pid	as_id
5	1
7	2
8	2
24	3
22	4
6	5
18	6
26	7
28	7

Table 10. ASPrecondition

pid	tid	as_id
4	3	5
5	6	6

Table 11. ASEffect

pid	tid_s	tid_e	as_id
5	2	5	2
4	1	3	4

Table 12. AS

as_id	name	price_min	price_max	time_min	time_max	…
1	as1	0.7	1.0	0.4	0.8	…
2	as2	0.8	1.0	0.3	0.6	…
3	as3	0.5	0.5	0.1	0.3	…

5.2 Abstract service execution relationship extraction

Using step 3 in Algorithm 8, *table$_{IO}$* can be generated through *ASOutput* and *ASInput*, as shown in Table 13.

Table 13. Table$_{IO}$

pid_o	as_id_o	pid_i	as_id_i
5	1	5	2
7	2	7	3
8	2	8	3
22	4	22	2
6	5	6	6
18	6	18	7

For example, for the tuple *tu*(5, 1, 5, 2) in $table_{IO}$, we get *oasid*=1, *iasid*=2, *onum*=1 and *inum*=2. The $temp_{IO}$ will be got through step 8, as shown in Table 14.

Table 14. Temp_{IO}

pid_o	as_id_o	pid_i	as_id_i
5	1	5	2

Then it gets *tunum*=1⇒*tunum*=*onum*⇒*ASRelation*.insert(*as1*, *as2*). The service execution dependency table (*ASRelation*) will be generated using the above method, as shown in Table 15.

Table 15. ASRelation(IO)

as_s	as_e
as1	as2
as2	as3
as4	as2
as5	as6
as6	as7

Using step 14 in Algorithm 8, $table_{CA}$ can be generated through *ASPrecondition* and *ASEffect*. It is shown in Table 16.

Table 16. Table_{CA}

pid_e	tid_s	tid_e	as_id_e	pid_p	tid_p	as_id_p
4	1	3	4	4	3	5
5	2	5	2	5	6	6

For the tuple *tu*(5, 2, 5, 2, 5, 6, 6) in $Table_{CA}$, val_c=*match*(5, 5, *Table*{*Para*, *Relation*, *Axiom*})=1.0, val_s=*matchstate*(5, 6, *Table*{*StateTrans, StatePath*})=0.5, and *ASRelation*.insert(*as2*, *as6*). Similarly, *ASRelation*.insert(*as4*, *as5*). The service execution dependency table (*ASRelation*) will be got from the *CA* level, as shown in Table 17.

Table 17. ASRelation(CA)

as_s	as_e
as2	as6
as4	as5

We can get the execution relationship between *as1*-*as7* through *ASRelation(IO)* and *ASRelation(CA)*, as shown in Table 18.

Table 18. ASRelation

as_s	as_e
as1	as2
as2	as3
as4	as2
as5	as6
as6	as7
as2	as6
as4	as5

5.3 Service Selection

For example, for the users' request *RE.ReInput*={*p4*, *p19*}, *RE.ReOutput*={*p24*}, the abstract service with proper function will be selected to form service execution path based on the service aggregation set (*ws1*-*ws15*). The concrete process is shown as follows.

<1>Rin_1=*p4*⇒*pidnum*=4

<2>In *match*() of Algorithm 3: (*tcid1*=4) ∧ (*tcid2* =4) ∧ (*trname*=*equivalentOf*)⇒val_{in}=1.0

<3>val_{in}>0.5⇒as_{io}={*as4*}, and Rin_1=*p19*⇒as_{io}.add(*as4*)⇒as_{io}={*as4*, *as4*}

<4>*DelDuplicate*(as_{io}) ⇒as_{io}={*as4*}

<5>*as4.Output*={*p22*}⇒*as4.Output*⊄*RE.ReOutput*, *RE.ReOutput*={*p24*}⇒*RE.ReOutput*!=*null*. Using step 18 in Algorithm 10, $table_{ASPath}$ can be got and it is shown in Table 19.

Table 19. $\text{Table}_{\text{ASPath}}$

pa_id	as_s	as_e
p3	as4	as2
p7	as4	as5
p11	as4	as3
p12	as4	as6
p13	as4	as6
p16	as4	as7
p17	as4	as7

<6>When *tu*=<*p11*, *as4*, *as3*>, *tu.as_e*=*as3*⇒*tu.as_e.Output*⊆*RE.ReOutput*. The $table_{PathIF}$ can be got through step 21, and it is shown in Table 20.

Table 20. $\text{Table}_{\text{PathIF}}$

pa_id	as	num
p11	as4	1
p11	as2	2
P11	as3	3

<7>$list_{as}$={*as4*, *as2*, *as3*}, *RE.ReOutput*←*RE.ReOutput*-*as3.Output*⇒ *RE.ReOutput*=*NULL*.

<8>*cas*=$list_{as}$={*as4*, *as2*, *as3*}.

6 Experiment

6.1 Experiment Environment

Software Environment: Windows XP, MyEclipse 8.5 M2, Mindswap OWL-S API(http://www.mindswap.org/2004/owls/api/), xampp(http://www.apachefriends.org/en/xampp.html).

Hardware Environment: CPU: double Intel (R) Core (TM)2 Duo CPU P8400@ 2.26GHz, memory: 2G.

Dataset: OWLS-TC(http://projects.semwebcentral.org/projects/owls-tc/). This dataset includes 5 subdirectories: *services*, *queries*, *ontology*, *domains* and *wsdl*. Services in different areas are in the directory of *services*, and ontology set are in the directory of *ontology*. In order to do the validation, we take a number of concepts with certain semantic relationships between them. We generate Web service randomly for experiment from the aspect of interface.

6.2 Experiment Analysis and Comparison

This section discusses the concrete experiments we have done about service clustering, service aggregation and service selection.

6.2.1 Service Clustering Experiment

Experiment 1. Comparison of Web service clustering efficiency, accuracy and recall rate

We use three criteria to evaluate the performance of our approach, namely Time, Accuracy and Recall [4].

The following methods use the semantic relationships among concepts to calculate service similarity, and we compare the time, accuracy and recall rate of these methods.

RDBJO: it uses the self-join method of RDB to realize Web service clustering in our approach.

AGENES [34]: it uses the traditional agglomerative nesting algorithm to cluster Web services.

QT [9]: it uses Quality Threshold (QT) algorithm to cluster the similar Web services.

K-medoids [35]: the partition clustering algorithm of K-medoids is used to realize Web service clustering.

This experiment is mainly to compare the time, accuracy and recall rate of RDBJO, AGENES, QT and K-medoids. The service interface is mainly considered. The experiment is taken in the following different services numbers: 20, 40, 60, 80, 100, 120, 140, 160, 180 and 200. The service clustering time, accuracy and recall rate of the above four approaches are shown in Figure 7, 8 and 9 respectively.

For the specific clustering method in Figure 7, Web service clustering time increases dramatically as the service number becomes larger. For the certain number of services, we can see the service clustering time is largely different using different methods. The clustering time of RDBJO is the least of all, the AGENES is the most, and the K-medoids is followed by RDBJO. This is because AGENES method needs to calculate the similarity of every two services, and it leads to the clustering time of this method is the maximum. The RDBJO method proposed in this paper uses the self-join operation in RDB. It helps to reduce the time of calculating concept matching degree, and the services can be clustered quickly. The QT method needs to do *IO* matching calculation between different services in turn, and it uses the semantic reasoning relationships to calculate service similarity. The time used is more than the RDBJO method.

In Figure 8 and 9, the service clustering accuracy and recall rate of RDBJO, AGENES and QT are same in the case of particular number of Web services. And K-medoids method is the lowest. This is because the service cluster centers are determined randomly when using K-medoids method to cluster services. The correctness of these service cluster center will influence the service clustering accuracy and recall rate directly.

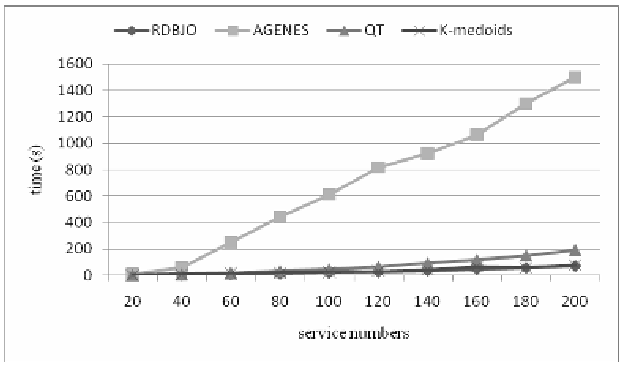


Figure 7. Comparison of Web service clustering time

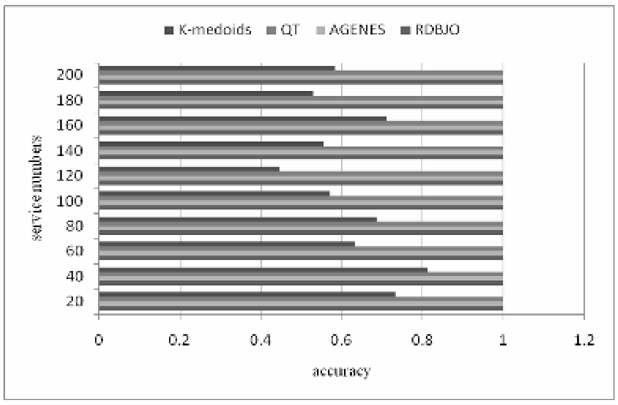


Figure 8. Comparison of Web service clustering accuracy

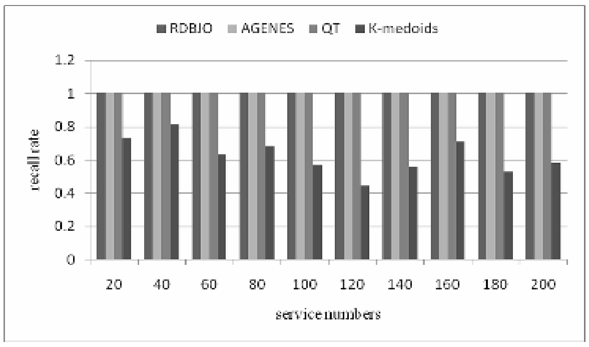


Figure 9. Comparison of Web service clustering recall

We can conclude that the service clustering time of RDBJO is the least of all, but its Web service clustering accuracy and recall rate are influenced. The time of K-medoids is slightly higher than RDBJO, but its accuracy and recall rate is the lowest of the four methods. The clustering accuracy and recall rate of AGENES is same to RDBJO, but it needs to specify the number of service clusters in advance.

Experiment 2. Comparison of efficiency and accuracy using different semantic relationships between concepts.

Exact method: it only considers the Exact relationship between concepts to computer service similarity, and it uses the self-join operation in RDB to cluster services. This experiment compares the service finding efficiency and accuracy of Exact and RDBJO methods. We evaluate service clustering accuracy of RDBJO and Exact. The experiment is taken in the following services numbers: 100, 200, 300, 400, 500, 600, 700, 800, 900 and 1000 separately. The service clustering time and accuracy of Exact and RDBJO are shown in Figure 10 and Figure 11.

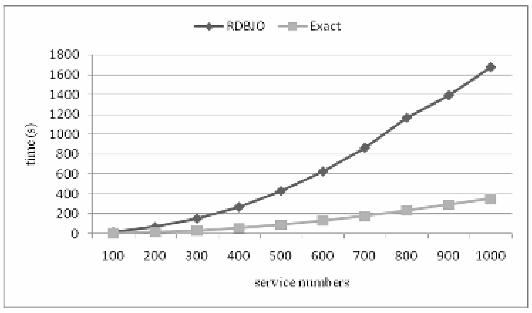


Figure 10. Comparison of Web service clustering time

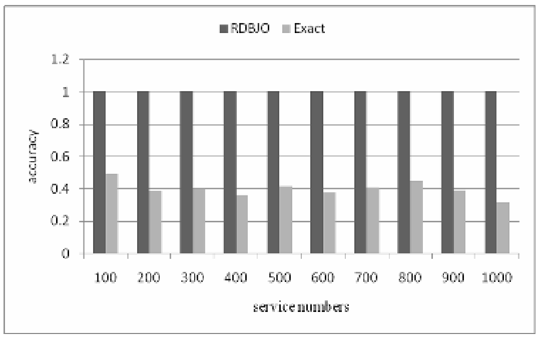


Figure 11. Comparison of Web service clustering accuracy

From Figure 10 and Figure 11, we can see Web service clustering time increases as the service number becomes larger of the two methods. For the certain number of services, the time of Exact method is less than RDBJO method. This is because Exact method only compares whether two concepts are equivalent or not when to calculate the matching degree between ontology concepts. It doesn't consider the concept reasoning relationships, such as superClassof, subClassof, etc. Therefore the clustering efficiency of Exact method is the largest. But its service clustering accuracy is far less than RDBJO. This is because it only considers the concept equivalence relationship, and it does not consider other reasoning relationships between concepts. Its concept matching degree is not accurate and the service similarity calculation accuracy will be influenced. Thus its service clustering accuracy will be reduced.

6.2.2 Service Aggregation Experiment

On the basis of the 36 service clusters that are formed through RDBJO clustering method of 1000 services, we do experiment about the abstract service extraction and getting abstract service execution relationships.

Experiment 3. Comparison of extracting abstract service time in different service clusters

The abstract services can be extracted from the 36 service clusters using Algorithm 6. This experiment compares the time that is used to extract abstract services from different service clusters. The experiment result is shown in Table 21.

Table 21. Getting abstract service time of different service clusters

Cluster ID	1	2	3	4	5	6	7	8	9	10	11	12
Time(s)	12.809	6.634	10.445	6.976	11.921	9.632	11.706	9.367	12.622	9.315	5.621	10.477
Cluster ID	13	14	15	16	17	18	19	20	21	22	23	24
Time(s)	5.520	9.460	10.905	6.252	8.688	7.030	7.448	9.252	7.446	9.001	6.010	6.461
Cluster ID	25	26	27	28	29	30	31	32	33	34	35	36
Time(s)	7.208	6.999	6.458	9.871	4.229	6.300	6.847	5.158	4.092	5.022	4.667	3.418

In Table 21 we can see the abstract service extraction time is different in different service clusters. This is mainly related to the different number of services in clusters. We can see the time of extracting abstract service is becoming more as the service number increases. For example, there are 52 services in the service cluster of Cluster ID=1 and 24 services in the service cluster of Cluster ID=2. The time of the former (12.809s) is about twice than the latter (6.634s).

Experiment 4. Comparison of time and numbers of extracting execution dependency relationships

On the basis of 36 abstract services being formed, this experiment compares the time of extracting abstract service execution relationships and the corresponding numbers. The result is shown in Table 22.

Table 22. Comparison of getting extract service execution relationship time and number

Different cases	Parameters	
	Time(s)	Number
Exact/super/sub/interaction	1.4966228	200
Exact/super/sub	1.0899271	134
Exact/super	0.8873854	92
Exact	0.6838185	53

In the case of different semantic relationships between concepts, the time of getting abstract service execution relationships using Algorithm 8 and the number of service execution relations is largely different. The time is the least of all when it only considers the *Exact* relationship, and the corresponding number of abstract service execution dependency relationship is the least of all. When it uses the relationships of Exact/super/sub/interaction, the time and the number of abstract service execution relations is the most of all. It means the more comprehensive of the concept semantic relationships, the more time and number of finding service execution relationships. This is because the more semantic relationships that is used, the more number of services which can meet this kind of relationship.

Experiment 5. Comparison of service aggregation time

The abstract service execution relationship table is joined in turn to realize service aggregation. The 36 abstract services will be organized and the different service execution paths will be formed. In the case of including 2~9 services in execution path, this experiment compares the time and path numbers of generating *ASExePath*. The result is shown in Table 23.

Table 23. Comparison of service aggregation time and path number

Service numbers in path	2	3	4	5	6	7	8	9
Time(s)	21.489496	31.600504	52.229053	96.29843	154.96674	243.13222	398.29663	593.4134
Path Number	53	80	114	163	230	322	465	703

In the case of including different number of services in service execution paths, the number of service execution path is largely different. As the service number in certain service path increases, the time of realizing service aggregation and number of service execution path are becoming larger.

6.2.3 Service Selection Experiment

Experiment 6. Comparison of service finding time and average path number under different request numbers

On the basis of service aggregation, this experiment generates different number of service requests. According to the service requests, it uses Algorithm 11 to find services to meet users' functional requests based on service aggregation. In the case of different thresholds (0.1-0.9), we compare the service finding time and the average service finding numbers. The result is shown in Figure 12 and Figure 13.

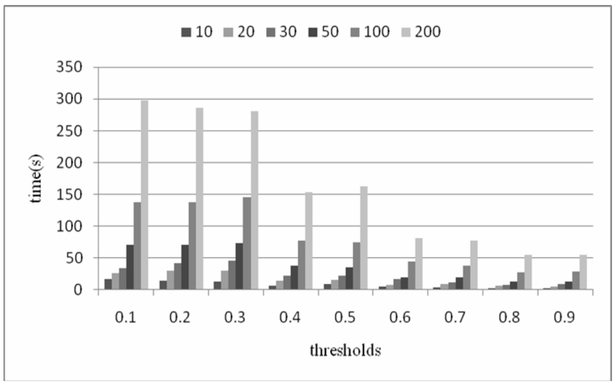


Figure 12. Service finding time of different request numbers

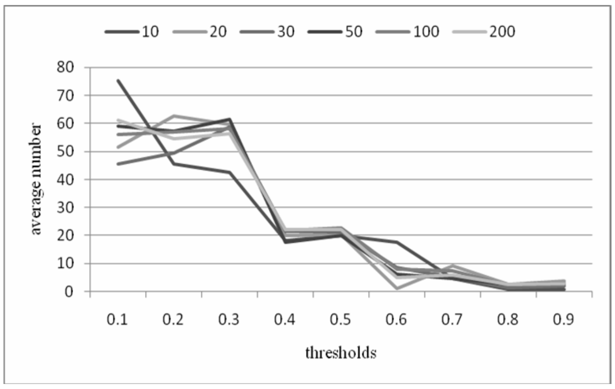


Figure 13. Service finding average numbers of different request numbers

For the certain threshold in Figure 9 and Figure 10, the service finding time and the average service numbers are becoming more as the number of users' request increases. For the certain number of users' requests, the service finding time and the average service numbers are becoming less as the threshold increases. This is because as the threshold increases, the number of services that can meet the condition is becoming less. The number of services that can be composed will be reduced, and it leads to the service finding time to be reduced.

There are different numbers (1~9) of services in the service execution path. In the condition of setting different thresholds, we can find there are different service numbers in execution path. The result is shown in Table 24.

Table 24. Average path numbers with different service numbers

Thresholds	Average path numbers								
	1	2	3	4	5	6	7	8	9
0.1	1.0	1.6	2.7	3.3	4.7	6.4	9.0	12.9	17.8
0.2	1.0	1.6	2.1	3.1	5.3	6.6	9.7	13.9	20.2
0.3	1.0	1.4	1.9	2.5	4.0	5.2	7.6	10.5	15.2
0.4	0.2	0.6	0.8	0.9	1.7	2.3	3.0	4.7	6.3
0.5	0.5	0.5	0.7	1.4	1.8	2.5	3.4	5.1	6.5
0.6	0.2	0.1	0.3	0.4	0.7	0.7	1.3	1.4	2.6
0.7	0.2	0.2	0.3	0.3	0.5	0.8	1.0	1.4	2.1
0.7	0.0	0.1	0.1	0.1	0.1	0.3	0.3	0.4	0.7
0.9	0.0	0.0	0.1	0.0	0.2	0.3	0.2	0.5	0.6

For the certain threshold in Table 24, as the service numbers which is included in service execution path increases, the average path number of services that meet request is becoming more and more. For the certain service numbers in service execution path, the average path number is becoming less as the threshold increases.

For example, for the specific service request *RE*, its interface is *Input*=http://127.0.0.1/ontology/books.owl #F, and *Output*=http://127.0.0.1/ontology/books.owl# Once.

<1> Services whose Input is matched with *RE.Input* are *cas*=<5.owls, 10.owls, 17.owls, 18.owls, 22.owls, 30.owls>.

<2> The Input of 5.owls is Input=http://127.0.0.1 /ontology/books.owl#F, and match(*RE.Input*, 5.owls. *Input*)=1.0. The Output of 5.owls is Output=http:// 127.0.0.1/ontology/books.owl#Weekly, and match(*RE. Output*, 5.owls.*Output*)=0.4. Then we can get (1.0+0.4)/2=0.7 and 5.owls can be matched with *RE*. The output of service in <10.owls, 17.owls, 18.owls, 22.owls, 30.owls> is not matched with *RE.Output*.

<3> Services in *cas* are seen as the first service, and we get the service execution path to meet *RE*, as shown in the following.

10.owls→17.owls
30.owls→31.owls
18.owls→29.owls→31.owls
5.owls→3.owls→10.owls→17.owls
10.owls→17.owls→9.owls→17.owls
17.owls→27.owls→36.owls→33.owls→29.owls→31.owls, etc.

We get the total service execution path number that can meet *RE* is 56. The number of atomic service is 1. The number of service execution path which includes 2 services is 2. The number of service execution path which includes 3 services is 4. The number of service execution path which includes 9 services is 21, etc.

7 Conclusion

In the service-oriented software engineering, how to organize and manage Web services, and thus to select the atomic service and a set of composite services with correlations to meet users' functional and *QoS* requirements efficiently is a key problem to be solved. On the basis of storing Web service and ontology information, a Web service clustering approach based on self-join operation in RDB is firstly proposed to cluster services in term of service interface and capability. Then the abstract service extraction method is used to get abstract service from specific service clusters. The dependency relationship between abstract services are determined in the view of interface and capability, and thus to realize service aggregation and organization. Then it selects the atomic service and a set of service to meet the functional and *QoS* requirements for users' individual requirements. The experiments are used to verify the proposed methods.

The next step research work includes the following aspects: considering the service operation, the relationship between operation and *IO* to cluster services; considering users' features to realize service aggregation and selection; building evolution mechanism to update the service aggregation.

Acknowledgments

This research is supported by the National Basic Research Program of China (2014CB340401), the Project Funded by the Priority Academic Program Development of Jiangsu Higer Education Institutions (PAPD), the Jiangsu Collaborative Innovation Center on Atmospheric Environment and Equipment Technology (CICAEET), the NSFC (61173141), the Fund of Jiangsu Engineering Center of Network Monitoring (KJR1308, KJR1402), the Fund of MOE Internet Innovation Platform (KJRP1403).

References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-Oriented Computing: A Research Roadmap, *International Journal of Cooperative Information Systems*, Vol. 17, No. 2, pp. 223-255, June, 2008.

[2] Z. J. Fu, X. M. Sun, Q. Liu, L. Zhou, J. G. Shu, Achieving Efficient Cloud Search Services: Multi-keyword Ranked Search over Encrypted Cloud Data Supporting Parallel Computing, *IEICE Transactions on Communications*, Vol. E98-B, No. 1, pp. 190-200, January, 2015.

[3] D. Skoutas, D. Sacharidis, A. Simitsis, T. Sellis, Ranking and Clustering Web Services Using Multicriteria Dominance Relationships, *IEEE Transactions on Services Computing*, Vol. 3, No. 3, pp. 163-177, July-September, 2010.

[4] Z. Li, K. Q. He, J. Wang, N. Zhang, An On-Demand Services Discovery Approach Based on Topic Clustering, *Journal of Internet Technology*, Vol. 15, No. 4, pp. 543-555, July, 2014.

[5] S. Dasgupta, S. Bhat, Y. Lee, Taxonomic Clustering and Query Matching for Efficient Service Discovery, *Proc. of 2011 IEEE International Conference on Web Services*,

Washington, DC, 2011, pp. 363-370.

[6] L. Yu, Z. L. Wang, L. M. Meng, X. S. Qiu, Clustering and Recommendation for Semantic Web Service in Time Series, *KSII Transactions on Internet and Information Systems*, Vol. 8, No. 8, pp. 2743-2762, August, 2014.

[7] J. X. Liu, K. Q. He, J. Wang, D. Ning, A Clustering Method for Web Service Discovery, *Proc. of 2011 IEEE International Conference on Services Computing*, Washington, DC, 2011, pp. 729-730.

[8] B. T. G. S. Kumara, I. Paik, W. H. Chen, Web-service Clustering with a Hybrid of Ontology Learning and Information-retrieval-based Term Similarity, *Proc. of IEEE 20th International Conference on Web Services*, Santa Clara, CA, 2013, pp. 340-347.

[9] L. Chen, L. K. Hu, Z. B. Zheng, J. Wu, J. W. Yin, Y. Li, S. G. Deng, WTCluster: Utilizing Tags for Web Services Clustering, *Proc. of 9th International Conference on Service-Oriented Computing (ICSOC 2011)*, Paphos, Cyprus, 2011, pp. 204-218.

[10] B. T. G. S. Kumara, I. Paik, H. Ohashi, W. H. Chen, K. R. C. Koswatte, Context Aware Post-Filtering for Web Service Clustering, *Proc. of 2014 IEEE International Conference on Services Computing*, Anchorage, AK, 2014, pp. 440-447.

[11] Y. H. Zheng, B. Jeon, D. H. Xu, Q. M. J. Wu, H. Zhang, Image Segmentation by Generalized Hierarchical Fuzzy C-means Algorithm, *Journal of Intelligent and Fuzzy Systems : Applications in Engineering and Technology*, Vol. 28, No. 2, pp. 961-973, March, 2015.

[12] K. Elgazzar, A. E. Hassan, P. Martin, Clustering WSDL Documents to Bootstrap the Discovery of Web Services, *Proc. of 2010 IEEE International Conference on Web Services*, Miami, FL, 2010, pp. 147-154.

[13] Q. Yu, M. Rege, On Service Community Learning: A Co-clustering Approach, *Proc. of 2010 IEEE International Conference on Web Services*, Miami, FL, 2010, pp. 283-290.

[14] X. Z. Liu, Q. Zhao, G. Huang, H. Mei, T. Teng, Composing Data-Driven Service Mashups with Tag-based Semantic Annotations, *Proc. of 2011 IEEE International Conference on Web Services*, Washington, DC, 2011, pp. 243-250.

[15] X. Z. Wang, Z. J. Wang, X. F. Xu, Semi-Empirical Service Composition: A Clustering Based Approach, *Proc. of 2011 IEEE International Conference on Web Services*, Washington, DC, 2011, pp. 219-226.

[16] M. Sellami, W. Gaaloul, S. Tata, Functionality-Driven Clustering of Web Service Registries, *Proc. of 2010 IEEE International Conference on Services Computing*, Miami, FL, 2010, pp. 631-634.

[17] Y. J. Ren, J. Shen, J. Wang, J. Han, S. Y. Lee, Mutual Verifiable Provable Data Auditing in Public Cloud Storage, *Journal of Internet Technology*, Vol. 16, No. 2, pp. 317-323, March, 2015.

[18] H. Y. Wu, Y. Y. Du, A Logical Petri Net-Based Approach for Web Service Cluster Composition, *Chinese Journal of Computers*, Vol. 38, No. 1, pp. 204-218, January, 2015.

[19] Z. B. Zhou, M. Sellami, W. Gaaloul, M. Barhamgi, B. Defude, Data Providing Services Clustering and Management for Facilitating Service Discovery and Replacement, *IEEE Transactions on Automation Science and Engineering*, Vol. 10, No. 4, pp. 1131-1146, October, 2013.

[20] J. X. Liu, K. Q. He, D. Ning, Web Service Aggregation Using Semantic Interoperability Oriented Method, *Journal of Information Science and Engineering*, Vol. 28, No. 3, pp. 437-452, May, 2012.

[21] J. X. Liu, K. Q. He, J. Wang, F. Liu, X. X. Li, Service Organization and Recommendation Using Multi-granularity Approach, *Knowledge-Based Systems*, Vol. 73, pp. 181-198, January, 2015.

[22] C. H. Hu, M. Wu, G. P. Liu, D. Z. Xu, An Approach to Constructing Web Service Workflow Based on Business Spanning Graph, *Chinese Journal of Software*, Vol. 18, No. 8, pp. 1870-1882, August, 2007.

[23] M. Aznag, M. Quafafou, Z. Jarir, Leveraging Formal Concept Analysis with Topic Correlation for Service Clustering and Discovery, *Proc. of 2014 IEEE International Conference on Web Services*, Anchorage, AK, 2014, pp. 153-160.

[24] S. L. Liu, Y. X. Liu, F. Zhang, G. F. Tang, N. Jing, A Dynamic Web Services Selection Algorithm with QoS Global Optimal in Web Services Composition, *Chinese Journal of Software*, Vol. 18, No. 3, pp. 646-656, March, 2007.

[25] M. Sellami, O. Bouchaala, W. Gaaloul, S. Tata, Communities of Web Service Registries: Construction and Management, *Journal of Systems and Software*, Vol. 86, No. 3, pp. 835-853, March, 2013.

[26] D. H. Lin, C. Q. Shi, T. Ishida, Dynamic Service Selection Based on Context-Aware QoS, *Proc. of 2012 IEEE Ninth International Conference on Services Computing*, Honolulu, HI, 2012, pp. 641-648.

[27] P. Wang, K. M. Chao, C. C. Lo, On Optimal Decision for QoS-aware Composite Service Selection, *Expert Systems with Applications*, Vol. 37, No. 1, pp. 440-449, January, 2010.

[28] G. S. Kang, J. X. Liu, M. D. Tang, X. Q. Liu, K. K. Fletcher, Web Service Selection for Resolving Conflicting Service Requests, *Proc. of 2011 IEEE International Conference on Web Services*, Washington, DC, 2011, pp. 387-394.

[29] K. Vukojevic-Haupt, F. Haupt, D. Karastoyanova, F. Leymann, Service Selection for On-demand Provisioned Services, *Proc. of 2014 IEEE 18th International Enterprise Distributed Object Computing Conference*, Ulm, Germany, 2014, pp. 120-127.

[30] R. X. Wang, L. Ma, Y. P. Chen, The Research of Web Service Selection Based on the Ant Colony Algorithm, *Proc. of 2010 International Conference on Artificial Intelligence and Computational Intelligence*, Sanya, China, 2010, pp. 551-555.

[31] X. Q. Fan, X. W. Fang, C. J. Jiang, Research on Web Service Selection Based on Cooperative Evolution, *Expert Systems with Applications*, Vol. 38, No. 8, pp. 9736-9743, August, 2011.

[32] X. G. Wang, J. Cao, Y. Xiang, Dynamic Cloud Service Selection Using an Adaptive Learning Mechanism in Multi-cloud Computing, *The Journal of Systems and Software*, Vol. 100, pp. 195-210, February, 2015.

[33] P. W. Wang, Z. Jin, L. Liu, G. J. Cai, Building Toward Capability Specifications of Web Services Based on an Environment Ontology, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 20, No. 4, pp. 547-561, April, 2008.

[34] P. Sun, C. J. Jiang, Using Service Clustering to Facilitate Process-oriented Semantic Web Service Discovery, *Chinese Journal of Computers*, Vol. 31, No. 8, pp. 1340-1353, August, 2008.

[35] S. Ram, Y. Hwang, H. M. Zhao, A Clustering Based Approach for Facilitating Semantic Web Service Discovery, *Proc. of the 15th Annual Workshop on Information Technolgies & Systems (WITS)*, Milwaukee, WI, 2006, pp. 1-6.

Biographies

Jianxiao Liu is a lecture in college of informatics of Huazhong Agricultural University. He received the Ph.D. degree in software engineering from Wuhan University of China in 2012. Until now, he has published over 20 papers. His current research interest is service computing and machine learning.

Xiaoxia Li is a lecture in college of informatics of Huazhong Agricultural University. She received the Ph.D. degree in computer application technology from Wuhan University of China in 2012. Her current research interest is collaborative computing.

Zhihua Xia received the Ph.D. degree in computer science and technology from Hunan University of China in 2011. He works as an associate professor in School of Computer & Software, Nanjing University of Information Science & Technology. His research interests include digital forensic and encrypted image processing.