Web Service Aggregation and Selection Based on Join Operation in RDB

Jianxiao Liu\textsuperscript{1,2}, Xiaoxia Li\textsuperscript{2}, Zhihua Xia\textsuperscript{1}\textsuperscript{*}

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

\textsuperscript{*} Corresponding Author: Zhihua Xia; E-mail: xia_zhihua@163.com

DOI: 10.3966/160792642018011901005

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 (\textit{Input} and \textit{Output}) and execution capability (\textit{Precondition} and \textit{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 \cite{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 \cite{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
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 work 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 \}
• **WSName** represents the name of ws.
• **Interface**={**Input**, **Output**}, and it denotes the input and output set of ws.
  \[
  \text{Input} = \{I_{i_1}, I_{i_2} \in \text{Class} , i = 0,1,\ldots, \text{num}\}
  \]
  \[
  \text{Output} = \{O_{o_1}, O_{o_2} \in \text{Class} , o = 0,1,\ldots, \text{onum}\}
  \]
  • **Capability**={**Precondition**, **Effect**}, it indicates the prerequisite for service execution and the effect resulting from ws execution.
  \[
  \text{Precondition} = \{\text{Prec}_p , p = 0,1,\ldots, \text{num}\}, \text{Prec}_p = \{c_p : s_p, c_p \in \text{Class}, s_p \in \text{hsm}(c_p)\}
  \]
  \[
  \text{Effect} = \{\text{Eff}_p , e = 0,1,\ldots, \text{enum}\}, \text{Eff}_p = \{c_{e \cdot rt \rightarrow ot_p} , c_{e \cdot rt} \in \text{Class}, rt_e, ot_p \in \text{hsm}(c_e), e = 0,1,\ldots, \text{enum}\}
  \]
  • **QoS**\(\{\text{QosName}_q, \text{Value}_q\}, \quad \text{QosName}_q \in \text{Class}, q=1,2,\ldots, qnum\) 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 **Input** denotes each input element of ws, and i denotes the input number in **Input**. Similarly, we can get the meaning of **Output**, **Prec_p** and **Eff_p**. 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 \cdot rt \rightarrow ot_p}** means the status change of concept c_e (from rt to ot). **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 \(c_1 \equiv c_2\), \(c_1 \subset c_2\), \(c_1 < c_2\), \(c_1 \cap c_2\), \(c_1 \psi c_2\) 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={ws1, ws2, ..., wsn}, Tables{ws, IOPE, Para, Relation, Axiom, StateTrans, StatePath}

Output: cluster[]
1: iope1, iope2→IOPE, Cluster→∅, vin[n][n], vout[n][n], vprec[n][n], veff[n][n], midiope tempiope
2: midiope→iope1*iope2=π
3: foreach i∈midiope, ws_id_1
4: foreach j∈midiope, ws_id_2
5: tempiope←σ_{ws_id_1=1\&ws_id_2=2}(midiope)
6: vin[i][j]→matchIO(tempiope, Table{Para, Relation, Axiom})
7: vprec[i][j]→matchPREC(tempiope, Table{Para, Relation, Axiom, StateTrans, StatePath})
8: veff[i][j]→matchEFCT(tempiope, Table{Para, Relation, Axiom, StateTrans, StatePath})

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. tempiope and midiope 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, iope1.pid@iope2.pid means there exist semantic relationships in Relation between the concept of iope1.pid and iope2.pid. In Algorithm 1, it firstly does self-join operation on IOPE tables to get midiope, 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, and ws2, as shown in Algorithm 2.

Algorithm 2. Interface matching algorithm (matchIO)

Input: tempiope, Table{Para, Relation, Axiom}

Output: valio
1: valsum←0, num←tempiope.count
2: foreach i←num
3: valsum←match(π_{pid_1}(tempiope), π_{pid_2}(tempiope), Table{Para, Relation, Axiom})
4: endfcf
5: valio←valsum/num
6: return valio

In Algorithm 2, pid_1, pid_2 are the pid number of tempiope. This algorithm gets the average matching degree of parameters in different tuples of tempiope, 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←0, tcid2←0, trname←∅
2: AR←Axiom= Relation
3: tcid1←π_{cid}(π_{pid=pid_1}(Para)),
   tcid2←π_{cid}(π_{pid=pid_2}(Para))
4: trname←π_{trname}(π_{cid=1\&cid=2}(AR))
5: if(trname==equivalentOf) then val←1.0
6: else if(trname==subClassOf) then val←0.8
7: else if(trname==superClassOf) then val←0.6
8: else if(trname==intersection) then val←0.4
9: else val←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, $v_{prec}[i][j]$ can be calculated using Algorithm 4.

Algorithm 4. Precondition matching algorithm (matchPREC)

Input: $temp_{prec}$ Table \{Para, Relation, Axiom, StateTrans, StatePath\}
Output: val
1: $val_{num} ← 0$, $val_{en} ← 0$, $val_{stat} ← 0$, $val_{mid} ← 0$, $um ← temp_{prec}.count$
2: foreach $i \sim num$
3: $val_{en} ← match(\pi_{tid}.(temp_{prec}), \pi_{tid}.(temp_{prec}), able\{Para, Relation, Axiom\})$
4: if ($val_{en} > 0.4$) then
5: $val_{stat} ← matchstate(\pi_{tid}.(temp_{prec}), \pi_{tid}.(temp_{prec}), Table\{StateTrans, StatePath\})$
6: $val_{mid} ← val_{en}+val_{stat}$
7: endif
8: $val_{num} ← val_{num}+val_{mid}$
9: endif
10: $val_{pre} ← val_{num}/num$
11: return $val_{pre}$

The implementation of $match$ in step 3 (Algorithm 4) is shown in Algorithm 3. In Algorithm 3, the matching degree $val_{en}$ between $c_j$ in $Prec_j = c_j • state_p$ is calculated firstly. When the value is larger than the threshold, $matchstate$ (Algorithm 5) will be used to calculate the matching degree $val_{stat}$ between $state_p$. Then it gets the average value of $val_{en}$ and $val_{stat}$.

Algorithm 5. Status matching algorithm (matchstate)

Input: tid_1, tid_2, Table \{StateTrans, StatePath\}
Output: val
1: $Table\{temp_{sp}\}, sum_{num}, enum_{num}, palength ← 0$
2: $pathnum ← StatePath.num$
3: if (tid_1 = = tid_2) return 1
4: foreach $i \sim pathnum$
5: $temp_{sp} ≅ \sigma_{sp.id}(StatePath)$
6: $sum_{num} ← \sum_{num}(\sigma_{id=tid_1}(temp_{sp}))$
7: $enum_{num} ← \sum_{num}(\sigma_{id=tid_2}(temp_{sp}))$
8: if ($sum_{num} = 0$ & & $enum_{num} = 0$) then
9: $palength ← enum_{num} - sum_{num}$; break
10: endif
11: $val ← 1/(\lfloor palength \rfloor + 1)$
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 $sum_{num}$ and $enum_{num}$ of tid_1 and tid_2 respectively, as shown in step 5-6. According to $sum_{num}$ and $enum_{num}$, 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 $c_i \equiv c_2$, $c_i \supset c_2$, $c_i \subset c_2$, $c_i / c_2$, $c_i \vee c_2$. Using the above concept reasoning relationships, we can get the following concept status transformation rules.

$\begin{align*}
(c_i \equiv c_2) & \Rightarrow (c_i; t_m) \wedge (c_i; t_n) \Rightarrow (c_i; t_m) \wedge (c_i; t_n) \\
(c_i \supset c_2) & \Rightarrow (c_i; t_m) \wedge (c_i; t_n) \Rightarrow (c_i; t_m) \wedge (c_i; t_n) \\
(c_i \subset c_2) & \Rightarrow (c_i; t_m) \wedge (c_i; t_n) \Rightarrow (c_i; t_m) \wedge (c_i; t_n)
\end{align*}$

As shown in Figure 2, $c_5$ has the following status: $t_1 \cdot t_2 \cdot t_3$ 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 $c_5; t_1 \rightarrow t_3$, $c_5; t_2 \rightarrow t_4$ and $c_5; t_1 \rightarrow t_3$, $c_5; t_1 \rightarrow t_2 \rightarrow t_4$ 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
Algorithm 6. Abstract service extraction algorithm (ExtractAS)

Input: Cluster[], Table{ws, IOPE, Para, Relation, Axiom}

Output: as

1: IWS←∅, vinum←0, PidRela←∅
2: Create table TASInput(pid, inum, as_id)
3: foreach i: 1←Cluster.num
4:  IWS=π_pid,as_id=(σ_as_id=’i’(ws))
5:  foreach j: 1←IWS.ws_id
6:  foreach k∈π_pid(σ_as_id=’i’(Input))
7:  if(tuplex<pid=k, as_id =’i’∈TASInput)
8:  Update TASInput set inum=inum+1 where pid=k
9:  else TASInput.insert(k, 1, i)
10: endfich
11: endfich
12: endfich
13: PidRela←GenePidRela(TASInput, Cluster[], Table{Para, Relation, Axiom})
14: foreach i: 1←Cluster.num
15:  foreach tuple tu∈σ_as_id=’i’(PidRela)
16:  if(tu.pid=1 || tu.pid=2) then
17:  tiopenum 1←−π_num(σ_pid=’tu.pid’ & as_id=’i’)
18:  tiopenum 2←−π_num(σ_pid=’tu.pid’ & as_id=’i’)
19:  if(tiopenum_1<=tiopenum_2) &&
20:  (tiopenum_2/Cluster[i].length>0.3) then
21:  Update TASInput Set
22:  inum=inum+tiopenum_1 where
23:  pid=’tu.pid2’ and as_id=’i’
24:  Delete from TASInput where pid=’tu.pid1’ and as_id=’i’
25:  else Update tu.pid1 of TASInput
26:  endif
27:  endif
28:  endif
29:  ASInput←π_pid,as_id(TASInput)
30:  if ASInput=π_pid,as_id(TASInput)
31:  as,QosName_q←{TQOS.Min(QosName_q), TQOS, Max(QosName_q)}
32:  endwhile
33:  ASInput/ASOutput/ASPrecondition/ASEffect
34:  endfich
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 into 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←σ_as_id=’i’(TIOPE)
5:  if(STAB.count(*)<2) then goto step 3
6:  foreach pid1, pid2∈π_pid(STAB)
7:  cid1←π_cid(σ_cid=’pid1’(Para))
8:  cid2←π_cid(σ_cid=’pid2’(Para))
9:  rid←π_cid(σ_cid=’cid1’ & cid2=’cid2’(Axiom))
10:  PidRela.insert(pid1,pid2,rid,i)
11:  endfich
12:  endfich
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 \(\{ASInput, Para, Relation, Axiom, StateTrans, StatePath\}\)

**Output:** \(ASRelation\)

1. \(table_{IO}, table_{CA}, temp_{IO}\times{\emptyset}\rightarrow{\emptyset}, inum, num, iasid, oasid, val, _onum_{=0}\)
2. Create Table \(ASRelation( as_s, as_e)\)
3. \(table_{IO}\rightarrow{ASOutput}\)
   \[\text{\{(ASOutput.as_id\rightarrow{ASInput.as_id})\}}\]
4. foreach tuple \(tu\in{table_{IO}}\)
5. \(\text{onum}\rightarrow{\langle\text{π}_{as\text{.id}=\text{oasid}}\text{ASOutput}\rangle.\text{count(*)}\}}
6. \(\text{inum}\rightarrow{\langle\text{π}_{as\text{.id}=\text{iasid}}\text{ASInput}\rangle.\text{count(*)}\}}
7. \(\text{temp}_{IO}\rightarrow{\langle\text{π}_{inum}\text{.count(*)}\}}
8. \(\text{if}(\text{onum}==\text{inum}\&\&\text{inum}==\text{num} \text{then})\)
9. \(\text{ASRelation}.\text{insert}(\text{as}^+\text{iasid}, \text{as}^+\text{oasid})\)
10. endif
11. endfch

12. \(table_{CA}\rightarrow{\text{ASEffect}}\)

13. \(\text{ASPrecondition.\text{insert(ASPrecondition.pid\rightarrow{\langle\text{π}_{id\text{.tid}_p}\rangle}}\}}\)

14. foreach tuple \(tu\in{table_{CA}}\)
15. \(\text{val}\rightarrow{\langle\text{match}(tu._\text{pid}_e, tu._\text{pid}_p)\rangle}\)
16. \(\text{idr}\rightarrow{\langle\text{matchstate}(tu._\text{tid}_e, tu._\text{tid}_p)\rangle}\)

17. if(\(\text{val}>0.4\) then
18. \(\text{if}(\text{val}+\text{val}_r+\text{val}_l)/2>0.5\) then
19. \(\text{ASRelation}.\text{insert}(\text{as}^+\text{tu.as_id}_e, \text{as}^+\text{tu.as_id}_p)\)
20. endif
21. endif
22. endfch
23. endfch
24. return \(ASRelation\)

In step \(6\), \(onum\) can be got through the aggregation function \(\text{Count(*)}\) 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 \(as\), \(id\) and \(as\_id\) in \(table_{IO}\). (1) \(SO_{1m}=SI_{2n}=num\); (2) \(SO_{1m}=num\) and \(SI_{2n}=num\); (3) \(SO_{1m}=num\) 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, \(p_1\rightarrow{p_n}\) 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 \(\langle as_1, as_s, \ldots, as_n\rangle\) execution dependency relationship between services \(\langle WS_1, WS_m\rangle\), this section mainly introduces how to organize service clusters and thus to realize service aggregation.

**Algorithm 9.** Generating \(ASExePath\) (GeneService ExePath)

**Input:** \(ASRelation\)

**Output:** \(ASExePath\)

1. \(i, j, pnnum_{=1}, fnnum_{=2}\)
2. \(ASPA_{i}\rightarrow{ASRelation}, pnnum_{=1}\)
3. repeat
4. \(\text{ASPA}_{i}\rightarrow{ASPA}_{i+1}\)
5. foreach tuple \(tu\in{ASPA}_{i}\)
6. \(id_{=i't+j'}\)
7. \(\text{foreach k:1}\rightarrow{TB_k}_\text{column}\)
8. \(\text{ASExePath}.\text{insert}(i'd, tu._\text{tid}_k, \text{pnnum}_{=})\)
9. \(\text{pnnum}_{=\text{pnnum}+1}\)
10. endfch
11. \(j_{=j'+1}, \text{pnnum}_{=1}\)
12. endfch
13. \(\text{pnnum}_{=1}, \text{fnnum}_{=\text{fnnum}+1}\)
14. until \(\text{ASPA}_{i}\) 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 \( i=1 \), \( ASPath_1=ASRelation \). It will be stored into \( ASExePath \) through \( ASPATHNum \). When \( i=2 \), \( AS Relation \) 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(\( RE \)) : \( RE=\left\{ REInput , ReOutput , ReCapa , ReQoS \right\} \)

\( ReInput =\{ rinput , rinput_i \in Class , i=1,2,..., rinum \} \)

\( ReOutput =\{ routput , routput_o \in Class , o=1,2,..., ronum \} \)

\( ReCapa =\{ reff_c , c=1,2,..., efunm \} , reff_e =\{ rentity : rprestate_e , rpoststate_e \in Class , rprestate_e , rpoststate_e \in hsm \{ rentity \} \} \)

\( ReQoS =\{ \{ rgosname_p , rvalue_p , runit_p \} , rgosname_p , runit_p \in Class , q=1,2,..., rqnum \} \)

#### 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 \( as_{in} \), and it deletes the duplicate services in \( as_{in} \) as shown in step 2-11. Then it judges \( as_i \) in \( as_{in} \) to get whether its output includes the elements in \( ReOutput \) or not. If it includes the elements, it adds \( as_i \) 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 \( tu_{.e} \) 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 27-29. 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 \( \langle as_s , as_e \rangle \). When \( as_s = as_e \), it refers to atomic service. When \( as_s \neq as_e \), \( \langle as_s , as_e \rangle \) expresses a set of services with correlations.

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

**Input:** \( RE , Table \{ ASInput , ASRelation , ASPATHNum , ASExePath , Para , Relation , Axiom \} \)

**Output:** \( cas \)

1: \( cas , rpath , as_{in} \rightarrow table_{ASPath} , as_{ca} \rightarrow \emptyset , pidnum , val_{as}=0 , i=1 , list_{as} , table_{PathIF} \)

2: foreach \( Rin \in RE.ReInput \)

3: \( pidnum = getpid(Rin) , Table \{ Para , Relation , Axiom \} \)

4: foreach tuple \( tu \in ASInput \)

5: \( val_{as} = match(pidnum , tu.pid , Table \{ Para , Relation , Axiom \} ) \)

6: if \( \{ val_{as} > 0.5 \} \)

7: \( as_{as} = add( as_{i}+tu.as_id ) \)

8: endif

9: endif

10: endif

11: as_{w} = DelDuplicate \( ( as_{w} ) \)

12: Similar to step 2-11, and get \( as_{ca} \) which can realize \( RE.ReCapa \)

13: foreach \( as_{i} \in as_{in} \)

14: if \( ( as_{i} . Output \neq RE.ReOutput ) \)

15: \( cas \leftarrow cas \cup \langle as_{i} , as_{e} \rangle \)

16: endif

17: \( RE.ReOutput = RE.ReOutput-as_{id}.Output \)

18: if \( ( RE.ReOutput = null ) \)

19: \( table_{ASPath} = \sigma_{ as_{s} = as_{e} } ( ASPATHNum ) \)

20: foreach tuple \( tu \in table_{ASPath} \)

21: \( tu_{.as} . Output \neq RE.ReOutput ) \)

22: \( table_{PathIF} = \sigma_{ pa_{id} = tu_{.pa} .id , ( ASExePath ) } \)

23: foreach tuple \( tu_{path} \in table_{PathIF} \)

24: \( list_{as} = list_{as} \cup tu_{path}.as \)

25: endif

26: \( RE.ReOutput \leftarrow RE.ReOutput-tu_{.as} . Output \)

27: endif

28: endif

29: endif

30: foreach \( as_{c} \in as_{ca} \)

31: find the tuple \( tu \) with \( as_{s}=as_{c} \), \( as_{e}=as_{c} \)

32: \( cas \leftarrow cas \cup \langle tu_{.as} .s , tu_{.as} .e \rangle \)

33: endif

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, ASIOP, ASPPathNum, ASEExePath, ASService, IOPE

Output: RWS

1: RWS ← ∅, table_in, table_out, list_out, list_in ← ∅, table SPDX
2: foreach cas = <as_s, ..., as_e> ∈ cas
3: if(cas.length = 2 & as_s = as_e) then
4:   table_in ← σ_as_s.id = as_s(ASService)
5:   foreach tuple tu ∈ table_in
6:     table_in[ws_id] = tu.ws_id(Inner)
7:     table_out[ws_id] = tu.ws_id(Outer)
8:     if(table_in[ws_id] ∈ RE.Input & table_out[ws_id] ∈ RE.Output)
9:       list_in.add(tu[ws_id])
10:  endfch
11:  endfch
12:  if(list_in.length = 0) then
13:    RWS.add(GetWSIDSemana(ass, ASService, Input, Output))
14:  else if(list_in.length = 1) then
15:    RWS.add(list_in[ws_id])
16:  else foreach ws_id ∈ list_in
17:    if(matchqos(RE.ReQoS, ws_id, Service) > 0.5) then
18:      RWS.add(tu[ws_id])
19:  endfch
20:  else
21:    list_in[ws_id] ← matchCompoWS(pa_id, ASEExePath)
22:    using step 15-18 to get service with proper QoS in list_out, and store into RWS
23:  endfch
24: return RWS

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_in, as seen in step 2. Then it judges each tuple tu in table_in, 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 between tablein and RE.ReInput, TREout between tableout and RE.ReOutput, as seen in step 6-11. The services with ws_id will be added into listws_id when the threshold is larger than the average value of TREin and TREout as shown in step 12-15. Finally, it finds the service with ws_id whose matching value is the largest in listws_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: valqos
1: valqos ← 0, num ← ReQoS.num, revalq[num], sumqos ← 0
2: foreach tuple tu ∈ Service
3: if(tu.ws_id == ws_id) then
4: foreach q ∈ {1, 2, ..., num}
5: if(ReQoS.rsqosname_q == availability || reliability)
6: revalq[q] ← 1.0
7: if(tu.Price <= rvalue_q) then
8: revalq[q] ← 1.0
9: endif
10: endfch
11: endfch
12: endfch
13: endfch
14: foreach q ∈ {1, 2, ..., num}
15: sumqos ← sumqos+ revalq[q]
16: endfch
17: valqos ← sumqos/num
18: return valqos

Algorithm 14. Composite service selection algorithm (matchCompoWS)

Input: pa_id, ASExePath, AS_Service
Output: listws
1: listws ← ∅, tableASPath, asfirst, listfirst, asnum ← 2, astotalnum
2: tableASPath ← σ_pa_id=pa_id(ASExePath)
3: astotalnum ← tableASPath.count(*)
4: asfirst ← π_as_first σ_asnum=astotalnum(tableASPath)
5: Same to step 4-18 in Algorithm 11 to get listfirst of the first service asfirst
6: foreach ws ws_id ∈ listfirst
7: listws.add(ws_id)
8: tableout ← σ_ws_id=ws_id(Output)
9: while(asnum <= astotalnum)
10: asnext ← π_as_next σ_asnum=astotalnum(tableASPath))
11: tablews ← σ_ws_id=asnext(AS_Service)
12: foreach tuple tu ∈ tablews
13: tablein ← σ_ws_id=asnum-1(tablews)
14: find the tu.ws_id which is exactly matched between tableout and tablein
15: endfch
16: listws.get(ws_id).add(tu.ws_id)
17: asnum++
18: end while
19: endfch
20: return listws

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 tableASPath, as shown in step 2. Then the first service asfirst in tableASPath will be found. Step 4-18 in Algorithm 11 will be used to find listfirst with proper QoS of asfirst, as seen in step 3-4. Each service ws_id in listfirst is judged in turn to get the tableout (output information) of ws_id. Then it gets asnext which is after asfirst in tableASPath. In the service cluster of asnext, services whose Input are matched with tableout will be found and added into the service execution path listws 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 listws.

5 Case Study

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

Table 1. Web service examples

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

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.
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.

Table 2. ASService

<table>
<thead>
<tr>
<th>ws_id</th>
<th>wsname</th>
<th>as_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ws1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ws2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>ws3</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>ws3</td>
<td>3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ws_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

When IWS.ws_id=1, the tuple with ws_id=1 in Input: tuple=\langle pid=10, as_id=1 \rangle \notin TASInput, EXE(TASInput. insert(10, 1, 1)).

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

When IWS.ws_id=5, the tuple with ws_id=5 in Input: tuple=\langle pid=10, as_id =1 \rangle \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

<table>
<thead>
<tr>
<th>pid</th>
<th>inum</th>
<th>as_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

When Cluster.num=1 in Algorithm 6, we can get STAB through step 4 and it is shown in Table 5.
We can get $ cid_1 = 10, cid_2 = 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>
<thead>
<tr>
<th>Table 5. STAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

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 $ \sigma_{su\_id=2}(P_1) \land (tiopenum_1 = 2) \land (tiopenum_2 = 1) \land (tiopenum_1 > tiopenum_2) \Rightarrow EX $ E(Update $ TASInput $ Set inum=3 where $ pid=10 \&\& as\_id=1 $, Delete from $ TASInput $ where $ pid=2 \&\& as\_id=1 $). Then $ TASInput $ will be updated.

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

<table>
<thead>
<tr>
<th>Table 6. PidRela</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi_id</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 8. ASInput</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 9. ASOutput</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>28</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 10. ASPrecondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 11. ASEffect</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 12. AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>as_id</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

5.2 Abstract service execution relationship extraction

Using step 3 in Algorithm 8, the $ QoS $ information of $ AS $ will be got and it is shown in Table 12.

<table>
<thead>
<tr>
<th>Table 13. TableIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>18</td>
</tr>
</tbody>
</table>
For example, for the tuple \( tu(5, 1, 5, 2) \) in \( table_{IO} \), we get \( oasid = 1, isid = 2 \), \( onum = 1 \) and \( inum = 2 \). The \( temp_{IO} \) will be got through step 8, as shown in Table 14.

**Table 14. Temp_{IO}**

<table>
<thead>
<tr>
<th>pid</th>
<th>as</th>
<th>id</th>
<th>pid</th>
<th>as</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Then it gets \( tunum = 1 \Rightarrow tunum = onum \Rightarrow ASRelation \). 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)**

<table>
<thead>
<tr>
<th>as_s</th>
<th>as_e</th>
</tr>
</thead>
<tbody>
<tr>
<td>as1</td>
<td>as2</td>
</tr>
<tr>
<td>as2</td>
<td>as3</td>
</tr>
<tr>
<td>as4</td>
<td>as2</td>
</tr>
<tr>
<td>as5</td>
<td>as6</td>
</tr>
<tr>
<td>as6</td>
<td>as7</td>
</tr>
</tbody>
</table>

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}**

<table>
<thead>
<tr>
<th>pid</th>
<th>t</th>
<th>id</th>
<th>as</th>
<th>id</th>
<th>pid</th>
<th>t</th>
<th>id</th>
<th>as</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the tuple \( tu(5, 2, 5, 2, 5, 6, 6) \) in \( Table_{CA} \), \( val_1 = match(5, 5, Table\{Para, Relation, Axiom\}) = 1.0 \), \( val_2 = matchstate(5, 6, Table\{StateTrans, StatePath\}) = 0.5 \), 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)**

<table>
<thead>
<tr>
<th>as_s</th>
<th>as_e</th>
</tr>
</thead>
<tbody>
<tr>
<td>as2</td>
<td>as6</td>
</tr>
<tr>
<td>as4</td>
<td>as5</td>
</tr>
</tbody>
</table>

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

**Table 18. ASRelation**

<table>
<thead>
<tr>
<th>as_s</th>
<th>as_e</th>
</tr>
</thead>
<tbody>
<tr>
<td>as1</td>
<td>as2</td>
</tr>
<tr>
<td>as2</td>
<td>as3</td>
</tr>
<tr>
<td>as4</td>
<td>as2</td>
</tr>
<tr>
<td>as5</td>
<td>as6</td>
</tr>
<tr>
<td>as6</td>
<td>as7</td>
</tr>
<tr>
<td>as2</td>
<td>as6</td>
</tr>
<tr>
<td>as4</td>
<td>as5</td>
</tr>
</tbody>
</table>

**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.

\( <1> \) \( Rin_1 = p4 \Rightarrow pidnum = 4 \)

\( <2> \) In match() of Algorithm 3: \( (tcid1 = 4) \land (tcid2 = 4) \land (trname = equivalentOf) \Rightarrow val_{io} = 1.0 \)

\( <3> \) \( val_{io} = 0.5 \Rightarrow as_{io} = \{ as4 \} \), and \( Rin_1 = p19 \Rightarrow as_{io}. add(as4 = \{ as4, as4 \} \)

\( <4> \) DelDuplicate(as_{io}) \Rightarrow as_{io} = \{ as4 \}

\( <5> \) \( as4 \).Output = \{ p22 \} \Rightarrow as4. Outputz RE.ReOutput, \( RE.ReOutput = \{ p24 \} \Rightarrow RE.ReOutput = null \). Using step 18 in Algorithm 10, \( Table_{AP} \) can be got and it is shown in Table 19.

**Table 19. Table_{ASPath}**

<table>
<thead>
<tr>
<th>pa</th>
<th>id</th>
<th>as_s</th>
<th>as_e</th>
</tr>
</thead>
<tbody>
<tr>
<td>p3</td>
<td>ah</td>
<td>as4</td>
<td>as2</td>
</tr>
<tr>
<td>p7</td>
<td>ah</td>
<td>as4</td>
<td>as5</td>
</tr>
<tr>
<td>p11</td>
<td>ah</td>
<td>as4</td>
<td>as3</td>
</tr>
<tr>
<td>p12</td>
<td>ah</td>
<td>as4</td>
<td>as6</td>
</tr>
<tr>
<td>p13</td>
<td>ah</td>
<td>as4</td>
<td>as6</td>
</tr>
<tr>
<td>p16</td>
<td>ah</td>
<td>as4</td>
<td>as7</td>
</tr>
<tr>
<td>p17</td>
<td>ah</td>
<td>as4</td>
<td>as7</td>
</tr>
</tbody>
</table>

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

**Table 20. Table_{PathIF}**

<table>
<thead>
<tr>
<th>pa</th>
<th>id</th>
<th>as</th>
<th>num</th>
</tr>
</thead>
<tbody>
<tr>
<td>p11</td>
<td>ah</td>
<td>as4</td>
<td>1</td>
</tr>
<tr>
<td>p11</td>
<td>ah</td>
<td>as2</td>
<td>2</td>
</tr>
<tr>
<td>p11</td>
<td>ah</td>
<td>as3</td>
<td>3</td>
</tr>
</tbody>
</table>

\( <7> \) list_{io} = \{ as4, as2, as3 \}, \( RE.ReOutput = \{ as3, Output \Rightarrow RE.ReOutput = NULL \).

**6 Experiment**

**6.1 Experiment Environment**


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.

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.

---

**Figure 10.** Comparison of Web service clustering time

**Figure 11.** Comparison of Web service clustering accuracy

---

### Table 21. Getting abstract service time of different service clusters

<table>
<thead>
<tr>
<th>Cluster ID</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster ID</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>Cluster ID</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
</tr>
</tbody>
</table>

---

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

<table>
<thead>
<tr>
<th>Different cases</th>
<th>Parameters</th>
<th>Time(s)</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact/super/sub/interaction</td>
<td>1.4966228</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>Exact/super/sub</td>
<td>1.0899271</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>Exact/super</td>
<td>0.8873854</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>Exact</td>
<td>0.6838185</td>
<td>53</td>
<td></td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Service numbers in path</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time(s)</td>
<td>21.489496</td>
<td>31.600504</td>
<td>52.229053</td>
<td>96.29843</td>
<td>154.96674</td>
<td>243.13222</td>
<td>398.29663</td>
<td>593.4134</td>
</tr>
<tr>
<td>Path Number</td>
<td>53</td>
<td>80</td>
<td>114</td>
<td>163</td>
<td>230</td>
<td>322</td>
<td>465</td>
<td>703</td>
</tr>
</tbody>
</table>

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.

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

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

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#F 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 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


**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.