SOCA
DOI 10.1007/s11761-011-0085-8

ORIGINAL RESEARCH PAPER

Towards efficiency of QoS-driven semantic web service composition for large-scale service-oriented systems

Bin Xu · Sen Luo · Yixin Yan · Kewu Sun

Received: 26 May 2010 / Revised: 16 May 2011 / Accepted: 21 May 2011

Abstract Quality-of-Service (QoS) performance guarantee for service-oriented systems (SOS) has become a critical problem. With the increasing number of offered services comes the challenge of efficiently building large-scale SOS to meet the required QoS criteria. Optimization of QoS-driven semantic Web service composition is known to be NP-hard. We address the efficiency issue by developing a polynomial time algorithm (QDA) for shortest sequence composition. We use dynamic programming to find service candidates for each execution. When all the services are searched, we use a depth-first trace back to derive the execution plan. We have tested our approach under Web-scale demands 20,000 services and 150,000 semantic concepts. In comparison with existing approaches, our experimental results show that QDA can be used to solve large-scale service composition problem effectively and efficiently with QoS guarantee.

Keywords Service composition · Quality-of-service · Large-scale services · Dynamic programming

1 Introduction

By composing many services from distributed software systems, service-oriented system (SOS) can handle complex processes for many business applications. With the increase in scale and complexity of the Web, Quality-of-Service (QoS) has become a critical challenge for the performance of SOS. Users clearly prefer to choose a SOS with better QoS, such as minimal response time or maximal throughput.

Efficient service composition algorithms for building SOS to meet QoS requirements remain an open challenge. In this paper, we address the following problem: given a set of available services, how to efficiently build a QoS-guaranteed SOS. There are four reasons why this task may be difficult:

1. **Scalability:** The number of services on the Internet has grown rapidly in recent years. According to statistics from the world's largest service registry—Seekda [1], there are about 20,000 public Web services. At the same time, service-oriented e-business systems have become more popular and more complex. An approach that can efficiently handle large-scale service composition for SOS is urgently needed.
2. **Semantics:** SOS must have a precise understanding about the semantics of data on the Web. Yet, there is still no effective integration between Web services and semantic information, making service composition with semantics difficult in SOS.
3. **Narrow focus on functionalities:** Traditional service composition algorithms usually focus only on functionalities, such as I/O parameters. Non-functional QoS criteria, such as response time and throughput, are often ignored, making the performance of SOS quite poor.
4. **Complexity:** Trade-offs exists between functionalities and non-functionalities when selecting services for SOS. For example, when using e-Bay, people can find a wide range of goods but sometimes must suffer long network delays. In comparison, people who use a local online market may have access to a limited range of goods but do

B. Xu (✉) · S. Luo · Y. Yan · K. Sun
Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, People's Republic of China
e-mail: xubin@keg.cs.tsinghua.edu.cn

S. Luo
e-mail: luosen@keg.cs.tsinghua.edu.cn

Y. Yan
e-mail: yanyx@keg.cs.tsinghua.edu.cn

K. Sun
e-mail: kewusun@keg.cs.tsinghua.edu.cn

Published online: 09 June 2011

not need to worry about network congestion. Addressing these complex tradeoffs is a challenge when considering QoS in SOS.

To meet the challenges in building SOS, we propose a QoS-driven service composition algorithm (QDA) that can efficiently build SOS with guaranteed QoS. The advantages of QDA include the abilities to: (1) efficiently handle large-scale service pools; this is the most important contribution of this paper; (2) integrate semantic information ("concepts" in ontology) with services through I/O matching in the composition; (3) satisfy both functional and non-functional (QoS) requirements. In this paper, we focus on two common QoS attributes in SOS: response time and throughput.

The rest of this paper is organized as following. Section 2 reviews and compares the related work. Section 3 gives some preliminary definitions and a formal description of the problem. Section 4 presents QDA. Section 5 gives our experiment strategy, test data sets and analysis of the results. Section 6 introduces our investigation on the practical relevance of QDA. Section 7 concludes the paper and proposes future work.

2 Related work

The composition of Web services involves using an orchestration model (such as BEPL) to define the order in which abstract Web services are called at design time and to define the dynamic selection of concrete Web services to be invoked at run time. The former process, deciding which abstract Web services are appropriate for the user's needs and how to construct them, is mostly based on functional requirements; non-functional properties are not formally considered. The latter process, dynamic Web service selection, involves choosing concrete Web services to fulfill the functionality of the abstract Web services schema under the guidance of non-functional properties, such as QoS.

The former process has been treated as a classical AI planning or graph search problem. Shankar [2–4] was in favor of regarding it as an AI planning problem and applied classic AI planning algorithms to the problem. Liang [5] proposed a semi-automated method for service composition; the main idea of his work is to construct an AND/OR graph from the service dependency graph and to find a sub-graph for the solution by applying a bottom-up search algorithm. Several other researchers [6,7] regard it as a graph search problem. Web Service Challenge [8] has proposed several competitions for service composition to address this problem, attracting many researchers. Gu [9] proposed a document-driven approach, which won first place at Web Service Challenge 2007. Yan [10] proposed an algorithm to search for the service composition with the shortest sequence, winning first place at Web Service Challenge 2008.

With a growing number of alternative Web services that provide the same functionality, QoS has become an important issue and the subject of extensive research. Ran [11] proposed an approach to manage QoS based on certification. Singhera [12] included QoS monitoring in a framework to overcome the lack of support for dynamism. The Service Level Agreement (SLA) framework [13] proposes differentiated levels of Web services using automated management and service level agreements. The SLA approach has become widely accepted with many papers focused on SLA formulation as well as SLA monitoring and enforcement [14,15]. We adopt SLA in this paper.

As Web service selection increasingly becomes a practical problem, extensive work has been done to address this challenge. Zeng [16,17] proposed a global planning approach for service composition to optimize multiple QoS criteria. Through integer programming, his approach can handle multiple execution paths. The disadvantage is that it can only handle small-scale service composition problems: Zeng's experiments are based on only dozens of candidate services. In contrast, our approach performs 3∼5 times faster than Zeng's method for large-scale service composition problems. Previous work such as that of Cardoso [18] also considers QoS in service composition but this work does not consider dynamic service composition.

Tao [19] studied the problem of service composition with multiple end-to-end QoS constraints. They proposed a broker-based architecture and several efficient heuristic algorithms to maximize QoS. Xiao [20] also studied QoS in end-to-end environments and presented a MCOP method for domain composition and the adaptation problem. Tao improved Xiao's work to handle multiple workflows such as parallelism, conditionals, and loops. However, their work does not address the performance of large-scale service composition.

Alrifai and Risse [21] proposed a solution for optimizing QoS in dynamic service selection. First, they use mixed integer programming to find the optimal decomposition of global QoS constraints into local constraints. Second, they used distributed local selection to find the best web services that satisfy these local constraints. The disadvantages of their work include inability to find the optimal QoS in all their experiments and poor composition times for random data sets.

Jaeger and Ladner [22] studied how pre-identified candidates, who have separated out by a selection process, can improve a composition with respect to particular QoS categories. They propose a model which uses redundant arrangements which involve the alternative candidates so as to supplement the originally assigned service. Some other works [23,24] also studied QoS in Web service composition.

The problem discussed in this paper is different from all previous work in that at design time, both functional requirements and non-functional properties are taken into account. Previous work usually considers them separately. In other words, we do not construct the workflow with abstract services that meet the user's purpose and then select services compliant to the defined workflow. Our approach is to take QoS into account when constructing the workflow at design time. By combining the two processes of web service composition, we make web service composition more efficient and effective.

This problem is related to but differs from several other research efforts. Web mashups are web applications generated by combining content, presentation, or application functionality from disparate web sources. Building mashups is a nontrivial task which requires human intervention, even with tools [25–28]. In contrast, the approach in this paper composes services automatically. The goal of Web service discovery and recommendation is to find or recommend services under specific circumstance for users. In the approach of this paper, Web service discovery and recommendation can be regarded as a working premise.

3 Problem statement

This section presents some basic definitions as well as a description of the composition problem. They include the definitions of service and SOS (functionalities and non-functionalities), the rules regarding how QoS is defined and calculated in SOS, and the optimization targets.

In general, a service can be developed by different methods and deployed on different platforms. In this paper, we define a service as follows.

Definition 1 Service $S = \{D_{\text{in}}, D_{\text{out}}, R, T\}$ where

$D_{\text{in}} = \{d_i | d_i$ is an input type of service, defined by a specific concept in the ontology$\}$;

$D_{\text{out}} = \{d_i | d_i$ is an output type of service, defined by a specific concept in the ontology$\}$;

R: response time—the time from receiving a request to producing the response;

T: throughput—invocations per minute supported by the service.

We identify the most important features of a service including I/O parameters and QoS. Each I/O parameter of a service can be mapped to a concept of some ontology to express semantic information about the service. For example, the service in Fig. 1 has input data types $d1$ and $d2$, as well as output data types $d3$, $d4$, and $d5$. QoS can represent any kind of non-functional property. Without loss of generality, we consider only response time and throughput in this paper. The service in Fig. 1 has response time of 4,000 ms and throughput of 6,000 invocations per minute. Any service that has these features can be used in building SOS, defined as follows.

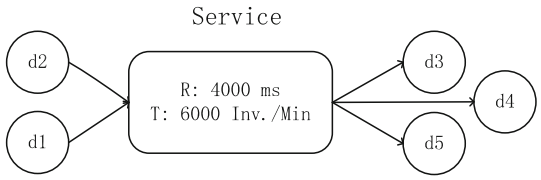

Fig. 1 Service example

Definition 2 Service-Oriented System (SOS) = $\{D_{\text{in}}, D_{\text{out}}, P, R, T\}$ where

$D_{\text{in}} = \{d_i | d_i$ is an input type of the SOS, defined by a specific concept in the ontology$\}$;

$D_{\text{out}} = \{d_i | d_i$ is an output type of the SOS, defined by a specific concept in the ontology$\}$;

P: the implementation of the SOS. It is an execution plan (such as BPEL) where services can be invoked following certain dependency rules to perform certain tasks;

R: response time—the time from receiving a request to producing the response by SOS;

T: throughput—invocations per minute supported by the SOS.

From these two definitions, we can see that I/O parameters and QoS for service and SOS are similarly defined. We view SOS as a compound service, which consists of services under some execution plan (P). How can we improve a compound service's QoS? Consider response time as an example. The response time of a single service is defined as the time from sending a request to the service to receiving the response. As mentioned in the first section, the response time could be affected by network conditions or the capability of servers where the service is deployed. Therefore, in most cases, the response time of a single service cannot be improved by the SOS developer. However, the response time for a SOS is a different story. The response time for a SOS is not only affected by the response time of services used in it, but also the execution plan through which the services are organized. We will discuss this in detail in the following.

For the execution plan, there are three basic flow structures in SOS including *sequence*, *parallel* and *switch* (Fig. 2).

A sequence consists of services (steps) that are invoked in order. A parallel flow structure consists of services that are invoked concurrently. A switch consists of services that can be selectively invoked. Simple processes can be nested inside of more complex processes. The whole execution plan can be expressed in a BPEL file.

Based on these three flow structures, we adopt several widely used QoS composition rules. Suppose sequence A consists of $A_1, A_2, \ldots, A_n$; parallel B consists of $B_1, B_2,$

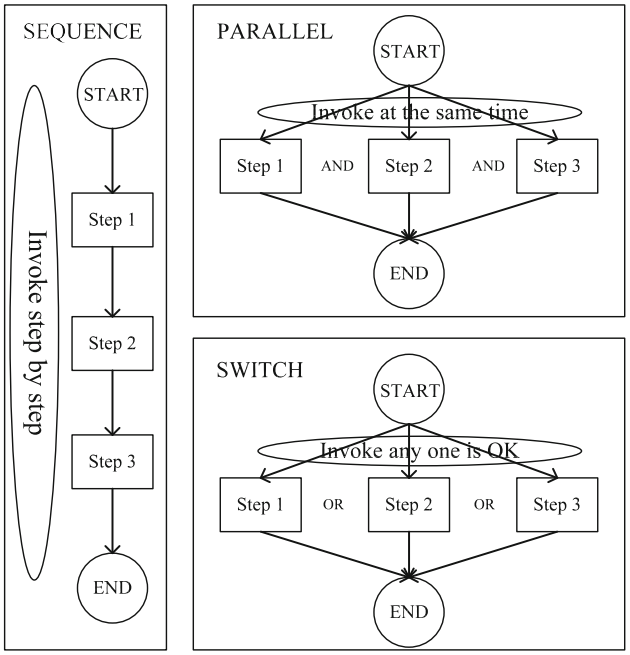

Fig. 2 Three basic flow structures in SOS

$\ldots, B_n$; switch C consists of $C_1, C_2, \ldots, C_n$. Definition 3 shows how $R(x)$ (the response time of x) and $T(x)$ (the throughput of x) can be calculated. These calculation rules are used in the Web Service Challenge competition [8] which will be further discussed in the experiments section.

Definition 3

$$R(A) = \{R(A_1) + R(A_2) + \cdots + R(A_n)\} \quad (1)$$
$$R(B) = \max\{R(B_1), R(B_2), \ldots, R(B_n)\} \quad (2)$$
$$R(C) = \min\{R(C_1), R(C_2), \ldots, R(C_n)\} \quad (3)$$
$$T(A) = \min\{T(A_1), T(A_2), \ldots, T(A_n)\} \quad (4)$$
$$T(B) = \min\{T(B_1), T(B_2), \ldots, T(B_n)\} \quad (5)$$
$$T(C) = \max\{T(C_1), T(C_2), \ldots, T(C_n)\} \quad (6)$$

These QoS composition rules are commonly used and intuitive. For the sequence flow structure, the steps are invoked one after another, and thus its response time is the sum of response time of its steps, and the minimal throughput of its steps determines its overall throughput. For the parallel flow structure, all the steps could be invoked at the same time; thus, its response time is the maximal response time of its steps, and as all its steps need to be invoked, its throughput is determined by the minimal throughput of all the parallel steps. Finally, for the switch flow structure, we can choose any one of its steps to invoke, and thus the response time is the minimal response time of its steps. The throughput for the switch flow structure is a little confusing. At first glance, it may appear that we should add up the throughputs of its steps to derive the overall throughput. However, we note that the steps may have different individual throughputs, and we should not simply add them up. We use maximal throughput of its steps to denote the throughput of the switch flow structure.

Since SOS can be defined as a combination of the three flow structures, based on the above QoS definition and composition rules, the QoS of SOS can be determined. For example, in Fig. 3, services C and D are invoked sequentially to form a sequence (Sequence 4). Thus, the response time of the sequence (600 ms) is the sum of the response time of service C (200 ms) and D (400 ms). If two or more services can be chosen in a switch-like services F and G, the response time should be the smallest one (150) of these services (300 and 150). The calculation can be extended recursively, e.g., sequence 5 and switch 6 are independent of each other, so they can be invoked in parallel.

Obviously, the response time of a parallel flow structure should be the biggest response times (150) of all the branches (100 vs. 150) which can be regarded as the bottleneck of the parallel flow structure. Completing the calculations, we find that the overall response time of SOS in this example is 1150 ms and the overall throughput is 7000 invocations/ min.

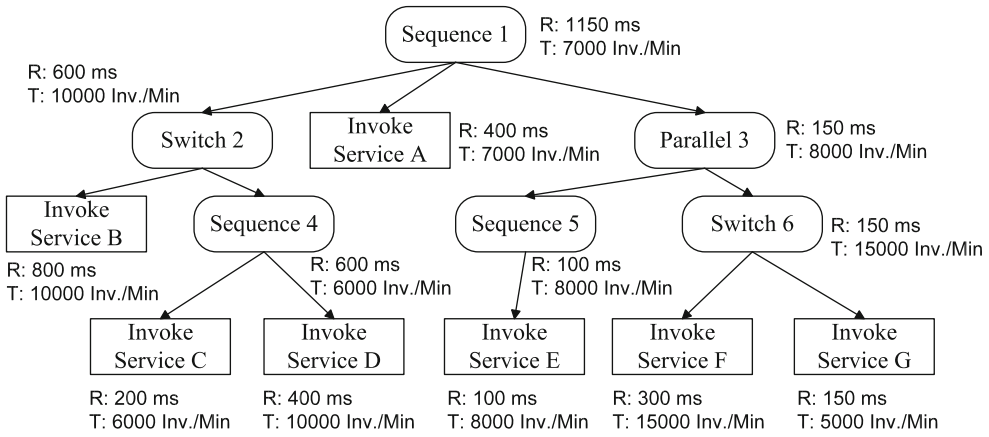

Fig. 3 QoS calculation for three structures in SOS

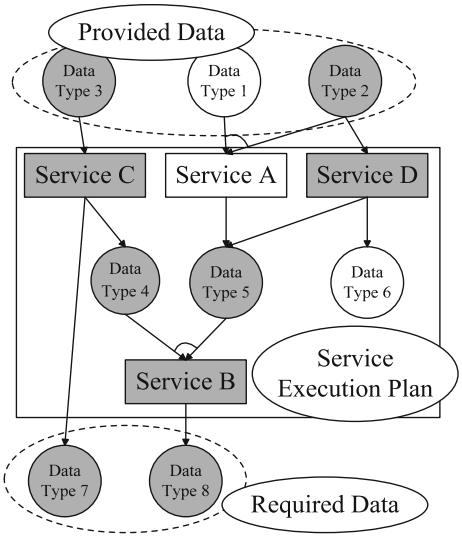

Fig. 4 Example of service composition

From the above discussion, we can see that the QoS of SOS is different from that of a single service and it is possible for developers to improve the QoS of SOS by selecting proper services and execution plan. Following is our definition of the composition problem.

Definition 4 Given a set of available services and a user request $\{D_{\text{in}}, D_{\text{out}}\}$ (D_{in}, D_{out} is defined in Definition 2), the composition problem is how to find a service execution plan (P in Definition 2) that takes the input data types and outputs the requested data types with optimized QoS.

Figure 4 shows an example. The user request includes the input (data types 1, 2, and 3) and the output (data types 7 and 8). The intermediate part which is enclosed by rectangular box is the execution plan of services. In the plan, the invocation of services follows certain rules. For example, service D outputs data type 5 which is the input of service B, so B cannot be invoked before D. Not all services are adopted in the plan (only gray ones). The execution plan is a service composition which is the goal of QDA.

4 QoS-Driven Algorithm (QDA) for shortest sequence composition

4.1 Algorithm description

Before introducing QDA, let us look at this problem from an empirical perspective. How do the structures and the number of services affect the QoS of SOS? We make the following observations.

(a) For the sequence flow structure and the parallel flow structure, the fewer the steps, the better the QoS.

(b) For the switch flow structure, if there are steps S_i and S_j and QoS of S_i is better than QoS of S_j. Then we can delete step S_j without affecting the correctness of the composition (the modified composition will still be a solution to the problem and the QoS remains unchanged).

(c) Generally speaking, the composition can be regarded as a sequence with a complex inner structure. Usually, the longer the sequence is, the larger the response time will be and the smaller the throughput will be.

From the above facts, we can deduce two trends. One trend is that the fewer services that are invoked, the better the QoS of SOS will be. The other trend is that the shorter the sequence, the better the QoS. QDA is based on these two trends. QDA seeks the composition with the fewest services or with the shortest sequence.

We first define and analyze the complexity of the Least Services Composition Problem and Shortest Sequence Composition Problem.

Definition 5 The Least Services Composition Problem (LSCP) is a decision problem defined as follows: Under the problem statement of Sect. 3, given a set of services S, a composition request R and a positive integer k, is there a composition that satisfies the request using S' (a subset of S) where the size of $S' <= k$?

Definition 6 The Shortest Sequence Composition Problem (SSCP) is a decision problem defined as follows: Under the problem statement of Sect. 3, given a set of services S, a composition request R and a positive integer k, is there a composition that satisfies the request with sequence length (the definition of sequence length of composition is given out in Definition 7) at most k?

Definition 7 Sequence length of type (SLT): for any provided data type, SLT is zero; for any un-provided type, SLT is the minimal value of sequence length of service (SLS) that output the type.

Sequence length of service (SLS): SLS is 1 plus the maximal value of sequence lengths of type (SLT) that the service requires as inputs.

Sequence length of composition (SLC): SLC is the maximal value of sequence length of service (SLS) in the composition.

We show that the Least Services Composition Problem is NP-complete.

First, we can see that LSCP falls in NP, as we can verify a solution in polynomial time (we verify whether every service in the workflow can be invoked, whether all the required

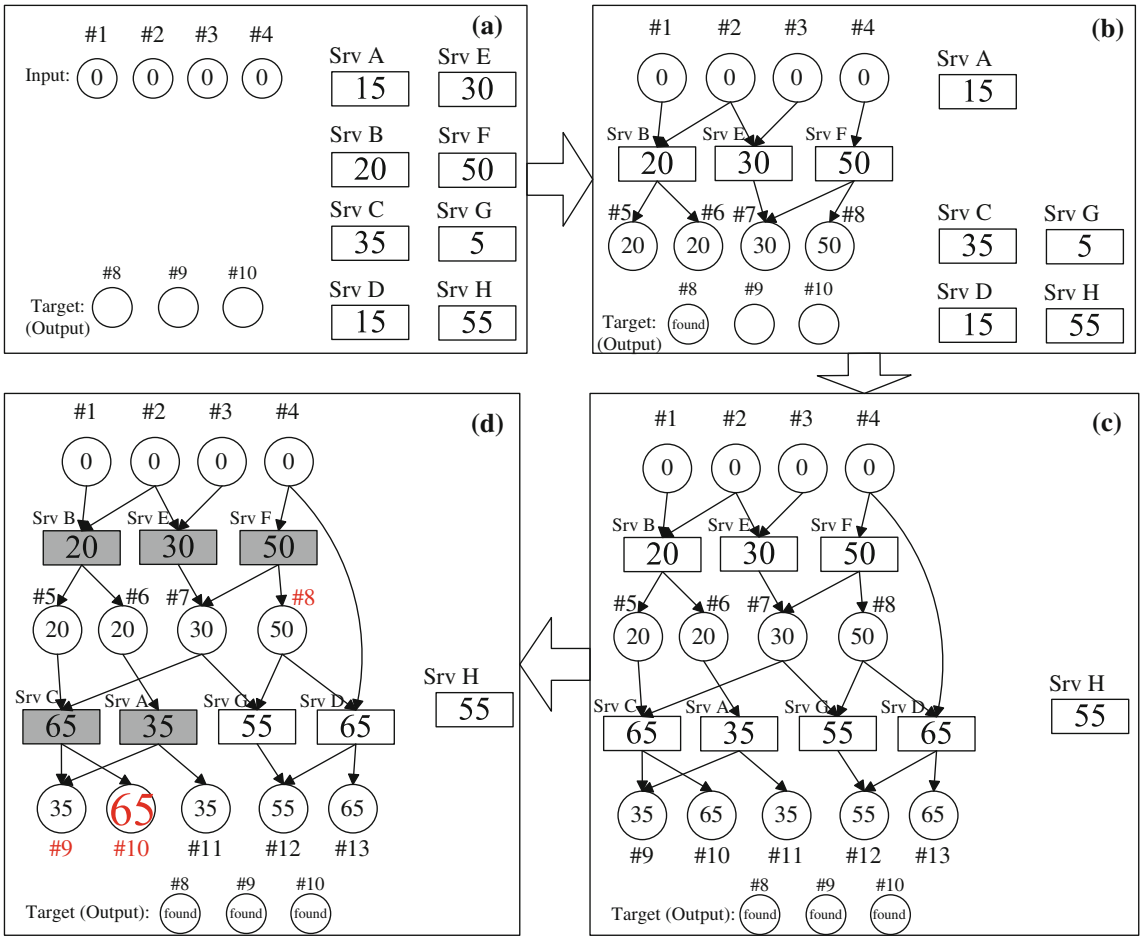

Fig. 5 Example of service composition

data is produced and whether the number of services in the workflow is not larger than k).

We use the vertex cover problem (VCP), which is a well-known NP-complete problem, for reduction. The vertex cover problem is defined as follows: Given a graph G and a positive integer k, does G contain a vertex cover of size at most k?

For a given graph G, we can use the following method to reduce an instance of the vertex cover problem to an instance of the Least Services Composition Problem. 1. We view edges of G as concepts in a composition problem (in this way the concepts will have no parent-child relations). 2. We view nodes of G as services (for an arbitrary node n, the service corresponding to it requires no inputs and produces the concepts represented by the edges incident to n). 3. The composition request R for LSCP includes all edges as required outputs and has no provided inputs. 4. The positive integer k remains the same for vertex cover problem and LSCP. By this way, we reduce vertex cover problem to LSCP in polynomial time.

If there is a solution to LSCP, there must be a solution to VCP cover problem. There is no way that a solution to VCP exists, but no any corresponding solution to LSCP exists. We can say that a solution to VCP exists if and only if there is a corresponding solution to LSCP.

As vertex cover problem is an NP-complete problem, so is the Least Services Composition Problem.

However, when we analyze the complexity of the Shortest Sequence Composition Problem, we found that it can be solved in polynomial time. We will prove this claim at the end of this section.

Inspired by the above insights, we design a stage-by-stage algorithm to achieve an optimal solution.

There are two important properties in the composition problem: *overlapping sub-problem* and *optimal sub-structure*. The overlapping sub-problem property means that a big problem can be divided into several small ones, and the solution of a small problem can be saved in order to be directly re-used in the following search. The optimal sub-structure property means that, to ensure the optimization of the whole

problem, every small sub-problem must be in its optimal state.

Based on these two properties, we propose a dynamic programming algorithm named QDA to solve the SSCP for SOS. The key ideas of QDA are:

1. We define a variable for every service that maintains the best known QoS value so far for the service. When executing the SOS, this value records the best QoS from the beginning to where the service locates. It will be assigned and updated while searching for the optimal composition.
2. We define a variable for every data type that maintains the best known QoS value for that data type. When executing the SOS, this value records the QoS from the beginning to where the data type is produced. It will be assigned and updated during the composition. For example, the response time of a data type is determined by the first service that can produce the data type. If more than one service can produce the data type as output, the variable records the minimal response time.
3. Concepts in ontology are used in the composition to define the parameter types of services I/O and their type hierarchy. Each data type of a service I/O can be mapped to a concept. If an output data type of service A can match an input data type of service B according to the ontology concept hierarchy, the two services can be connected. We say that a data type is *satisfied* when at least one service can output a matched data type. Similarly, we say a service is *satisfied* when all its input parameters are satisfied.
4. We use a stage-by-stage strategy in searching for the optimal solution. We model every stage of the search process as a "small part of the whole problem". The optimal results of all known sub-problems are saved and reused in the following search. Thus, if every sub-problem is guaranteed to be locally optimal, the whole solution can be guaranteed to be optimal because every service selection is based on the states and the optimal values of previous sub-structures.
5. We combine breadth-first search and depth-first trace back to find the solution. The QoS of services and data types are calculated and updated in the breadth-first search. When all required output data types are satisfied, we will produce the composition (P in Definition 2) using a depth-first trace back.

Figure 5 shows an example of the search process to find the SOS with an optimal (minimal) response time. The user request includes the provided input data types {data type #1, #2, #3, and #4}, required output data types {data type #8, #9, and #10}, and all available services {A, B, . . . , H}.

QDA first adds services B, E, and F to the composition since they can be directly invoked using the input data types. It then updates the QoS of these services and produces new data types (#5, 6, 7, and 8) as shown in Fig. 5b. Based on the produced data types and all previous available data types, more services (C, A, G, and D) can be added to the composition. QDA updates these services' QoS using Definition 3. For example, in Fig. 5a, the response time of C itself is 35. When it is added in the composition, the optimal response time from the beginning of the composition to service C is based on the response time of all its input (#5 and #7) and itself. In Fig. 5d, the maximum response time of data type #5 and #7 is 30, then the response time of service C is updated to 65 (30 plus 35).

QDA continues to add services and update QoS until all required data types are covered and no more services can be added into the composition. In Fig. 5d, from data type #8, #9, and #10, it conducts a trace back depth-first search to record the path of the SOS. At this point, we know that the minimum response time of data type #8, #9, and #10 are 50, 35, and 65, respectively. The maximum of the three will be the minimum response time we can get from the SOS.

QDA is detailed in the following. First we define data structure for services and data types which are used to record best known QoS.

Definition of DataType and Service

```
1  struct DataType
2  {
3    //the concepts that match to the datatype
4    Concept concept_of_datatype;
5    //the least response time for producing it
6    float response_time;
7    //the greatest throughput for producing it
8    float throughput;
9    //point to the service which generate it with least
     //response time
10     Service ptr_response_time_generator;
11     //point to the service which generate it with greatest
       //throughput
12     Service ptr_throughput_generator;
13   }
14   struct Service
15   {
16     String name;
17     float response_time;
18     float self_ response_time;
19     float throughput;
20     float self_ throughput;
21     List<DataType> input;
22     List<DataType> output;
23   }
```

The main part of QDA is presented in the following pseudo code.

Main Search Process

```
1  foreach Service S_i {
2    S_i.response_time = S_i.self_response_time;
3    S_i.throughput = S_i.self_throughput;
4  }
5  foreach DataType D_j {
6    D_j.response_time = infinity;
7    D_j.throughput = 0;
8  }
9  foreach DataType D_k in the provided DataTypes{
10   D_k.response_time = 0;
11   D_k. throughput = infinity;
12   available_data.add(D_k);
13 }
14 while((avalbl_srv =
     findAvailableServices(available_data)) is not empty) {
15   foreach Service S_n in avalbl_srv{
16     queue.push(S_n);
17   }
18   while (queue is not empty){
19     Service s = queue.pop();
20     UpdateServiceQoS(s);
21     for each DataType D_o in s.output{
22       available_data.add(D_o);
23     }
24   }
25 }
26 if all required DataType are found {
27   print("<parallel>");
28   foreach required DataType D_r {
29     traceback(D_r);
30   }
31   print("</parallel>");
32 }
```

As shown in the pseudo code, QDA performs some initializations which include initialization of response time and throughput for both for Service and DataType. We then execute a while loop which does a breadth-first search until there are no more services that can be used. In the while loop, we update QoS related to the newly used service which is intended to ensure the sub-structure is locally optimal (thus the whole solution is optimal). Finally, when there are no more services that can be used, we use a depth-first trace back search to generate the solution in BEPL format.

In the pseudo code above, findAvailableServices() is used to find currently available but unused services. The pseudo code for findAvailableServices() is given below.

Find Available Services

```
1  findAvailableServices(available_data)
2  {
3    available_service.empty;
4    for each Service S_i {
5      if S_i is unused and S_i.input is subset of
   available_data{
6        available_service.add(S_i);
7        mark S_i as used Service;
8      }
9    }
10   return available_service;
11 }
```

UpdateServiceQoS(S_m) is a very important function in QDA; its pseudo code is shown in the following.

Update Service QoS

```
UpdateServiceQoS(S_m)
{
  S_m.response_time = maxResponseTime(S_m.input)+
S_m. self_ response_time;
  S_m.throughput = min(minThroughput(S_m.input), S_m.
self_throughput);
  for each DataType D_i in S_m.output{
    if D_i.response_time > S_m.response_time{
      D_i.response_time = S_m.response_time;
      D_i.ptr_response_time_generator = S_m;
    }
    if D_i.throughput < S_m.throughput{
      D_i.throughput = S_m.throughput;
      D_i.ptr_throughput_generator = S_m
    }
}
```

Last but not least, traceback(D_n) is used to generate the solution in BEPL format. Its pseudo code is given below.

Trace Back Search

```
traceback(D_n)
{
  if D_n belongs to the provided DataTypes{
    Return;
  }
  print("<sequence> \n<parallel>");
  foreach DataType D_m in
D_n.ptr_response_time_generator.input{
    traceback(D_m);
  }
  print("</parallel>");
  print("invoke " +
D_n.ptr_response_time_generator.name);
  print(</sequence>);
}
```

4.2 Analysis of algorithm complexity

First, we want to point out that QDA seeks the solution with the shortest sequence. We use a stage-by-stage method and each stage corresponds to exploration of a certain sequence. For example, in the second stage, all the services falling in sequence are searched. Then the first solution we get will be the shortest sequence solution.

However, the solution with shortest sequence might not be the optimal result. For instance, the user request includes the provided input data types {data type #1}, required output data types {data type #2, #3}, and all available services {A, B, C}. The properties of the services are shown in the following table. In this example, composition with least services and composition with shortest sequence are both Service A. However, the optimal result is sequence Service B and Service C.

Service	Inputs	Outputs	Response time (ms)
A	#1	#2,#3	500
B	#1	#4	100
C	#4	#2,#3	200

To overcome this shortcoming, in QDA, we do not stop as soon as we get the shortest sequence solution. We continue the search until there are no available services, and in this way we can make sure that we get the optimal result. UpdateServiceQoS(S_m) makes sure every data type is pointed out to the optimal generator and through a trace back starting from the required data types, the generated result must have the optimal QoS.

Finally, we show that QDA executes in polynomial time. Suppose the number of services is n and the number of concepts is m. We note that at every stage, we explore at least one service. Thus, the number of unused services will decrease from stage to stage. Clearly, there are at most n stages, and at every stage, all the current unused services are scanned. If a service is not available currently, we continue to scan other unused services. If a service can be invoked, we add it to the current stage and update QoS related to it. This takes $O(m)$ time. There are at most n unused services, so a stage takes $O(mn)$ time. Finally, the overall searching process takes $O(mn) * n = O(mn^2)$ time. This a very loose upper bound.

5 Experiments

5.1 Experimental methodology

Our experiments follow the requirements of the Web Service Challenge (WS-Challenge), an annual service composition competition held at the IEEE e-Commerce conference (CEC) since 2006 [8]. It focuses on the semantic composition of Web services and uses OWL ontology to define services and their relationships. QoS criteria were introduced into the competition in 2009.

In WS-Challenge, the data formats and the contest data itself are based on the OWL, WSDL, WSLA, and WSBPEL schemas for ontologies, services, service qualities, and service orchestrations. The WSDL file contains a set of services along with annotations of their input and output parameters. The OWL file contains the taxonomy of concepts used in this challenge in OWL format. The quality-of-service for a service is specified using WSLA language. The challenge itself is also represented by a valid WSDL service description. Participants are required to give out composition solutions using WSBPEL. WS-Challenge awards the most efficient system. Evaluation of efficiency consists of two parts, composition evaluation and time measurement. WS-Challenge will give out several challenge sets with different scale and orchestration complexity. The score calculation for per challenge set is as following.

- +6 Points for finding the service composition with least response time that solves the challenge.
- +6 Points for finding the service composition with largest throughput that solves the challenge
- +6 Points for the composition system which finds the service composition with least response time or largest throughput that solves the challenge in fastest time.
- +4 Points for the composition system which finds the service composition with least response time or largest throughput that solves the challenge in the second fastest time.
- +2 Points for the composition system which finds the service composition with least response time or largest throughput that solves the challenge in the third fastest time.

WS-Challenge has provided a set of standard experimental tools including a test set generator and a service composition result checker. The test set generator is used to generate four input files and a benchmark. The generated Web services are virtual yet realistic. They are virtual since there are no actual service implementations that can be invoked on the Internet. But they are realistic in our experiment because they are consistent with Definition 1, including I/O and QoS values. The benchmark (a standard result) is also provided by the test set generator and is guaranteed to have the optimal QoS (least response time and largest throughput). We evaluate our experimental results by comparison with the benchmark.

We use the composition result checker provided by WS-Challenge to check whether our composition is correct for the request and calculate the QoS values of the composition.

Figure 6 shows the experimental process. First, we use the test set generator to generate four input files for each test set. (1) Services.wsdl which records all available Web services; (2) Taxonomy.owl which records all concepts in an ontology format [29]; every input/output data type of the Web services is defined as an instance of some concept; (3) Servicelevelagreements.wsla which records the QoS values (response time and throughput) of Web services [30]; (4) Query.wsdl which records one user query including the provided data types and required data types.

Then QDA takes these four files as input and produces a BPEL file as the composition result. At the same time, we record the time cost during the composition procedure. Finally, we use the checker to check whether the result is correct, record the QoS values, and compare it with the standard result.

5.2 Experimental settings and results

For the experimental setting, we are concerned about the scale of Web services and ontology concepts. After investi-

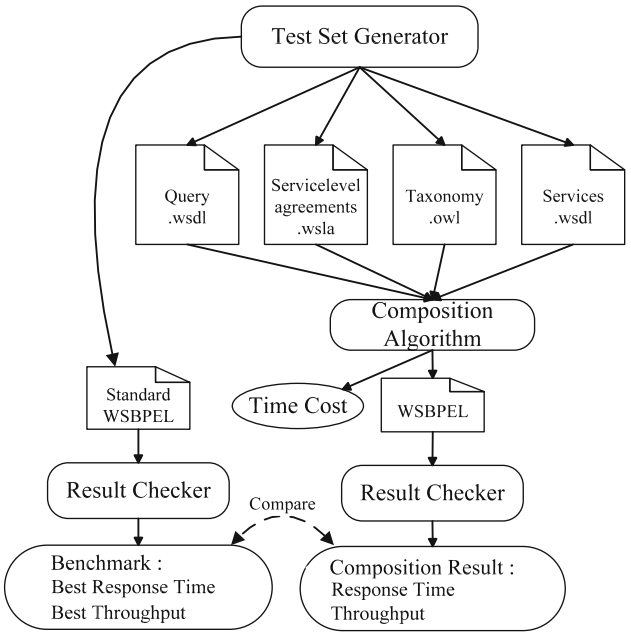

Fig. 6 Experimental process

Table 1 Test sets in Experiment1

Test set ID	Test sets properties	
	Number of concepts	Number of web services
1	37,500	500
2	75,000	1,000
3	112,500	1,500
4	150,000	2,000
5	187,500	2,500
6	225,000	3,000

gations on the Internet, we found that the number of available Web services is about 20,000, and the Cyc ontology [31] has about 150,000 concepts. The Cyc ontology is possibly the largest existing ontology, having a set of concepts which tries to describe universal subjects. Along with the development of the semantic Web, more and more data on the Web will be formalized and mapped to concepts of ontology. In our approach, we integrate concepts and I/O data types together. In the following experiments, we set the ratio between the number of Web services and the number of concepts based on our earlier investigation, to approximately 75:1.

The test sets in Experiment 1 were derived by changing the number of concepts and Web services while maintaining the above ratio between them. Table 1 shows the test sets of Experiment 1.

The configuration of our test machine is: Intel Core 2 CPU 1.83 GHz with 1 GB RAM, running Windows XP.

For the composition results of each test set, we are concerned about the composition time cost and QoS values.

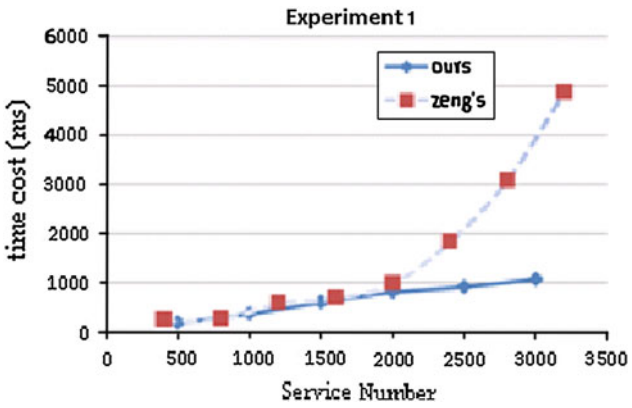

Fig. 7 Efficiency analysis of Experiment 1

Table 2 Test Sets in Experiment 2

Test set ID	Test sets properties	
	Number of concepts	Number of web services
1	50,000	20,000
2	100,000	20,000
3	150,000	20,000
4	200,000	20,000
5	250,000	20,000
6	300,000	20,000

Figure 7 shows the composition time cost for each test set. We can see that the time complexity is linear. In most cases, the time cost is less than 1 s. Compared to Zeng's work [16] which is shown as the red line in the figure, QDA has much better performance with respect to time cost. In addition, our experiments are based on a much larger concept scale (concepts in a universal domain) than that in Zeng's experiments (concepts in a local domain). This means that under the same number of services, our experiment handles more complex problems.

The second study investigates the performance of QDA when the number of Web services remains the same while the number of ontology concepts increases. This experiment is close to the real world, because the semantic Web is expanding rapidly while the number of Web services has become relatively stable in recent years. Since there are about 20,000 available Web services on the Internet, we fix the number of Web services at 20,000 and vary the number of semantic concepts. Table 2 shows the test sets for Experiment 2.

Figure 8 shows the composition time cost for each test set. We can see that the time complexity is linear and all test sets perform efficiently (less than 1.7 s).

In the third experiment, the number of concepts remains fixed while the number of Web services increases. This assumption models the future growth trends of the Web and Web applications. When most concepts are well described by

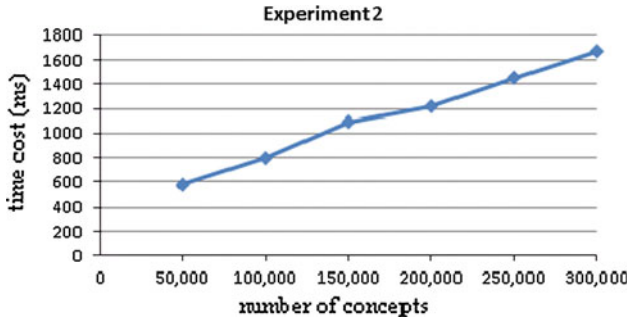

Fig. 8 Efficiency analysis of Experiment 2

Table 3 Test sets in Experiment 3

Test set ID	Test sets properties	
	Number of concepts	Number of web services
1	150,000	2,000
2	150,000	4,000
3	150,000	6,000
4	150,000	8,000
5	150,000	10,000
6	150,000	12,000

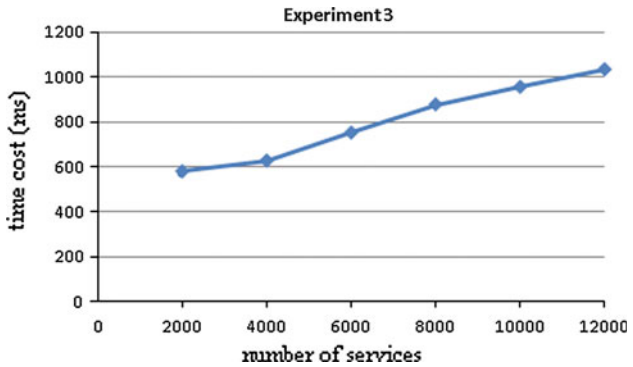

Fig. 9 Efficiency analysis of Experiment 3

Fig. 10 Processes of service cluster

ontology, the number of Web services may increase because of new business enterprises. Table 3 shows the test sets for Experiment 3.

Figure 9 shows the composition time cost for each test set. We can see that when the number of concepts remains the same, the time cost will not change significantly even with the increase of Web services. This means QDA performs stably and efficiently (less than 1 s) under this situation.

Table 4 shows the QoS values of composition results in the above three experiments. As we have explained above, there is a standard benchmark result for each data set which has the optimal QoS. QDA can always guarantee that the composition result achieves the optimal QoS. The values in the table match the values of the standard results. *R* is the response time measured in ms; *T* is throughput measured in invocations per minute.

Table 4 QoS values of composition results

Test set ID	Experiment 1		Experiment 2		Experiment 3	
	R	*T*	*R*	*T*	*R*	*T*
1	1,950	1,000	960	6,000	850	3,000
2	1,700	2,000	1,800	1,000	1,370	1,000
3	1,760	3,000	1,370	1,000	1,460	13,000
4	1,370	1,000	1,240	3,000	590	9,000
5	1,400	3,000	2,210	3,000	1,480	4,000
6	1,030	5,000	1,120	8000	890	5,000

Table 5 Test sets properties

Test set ID	Number of web services	Number of concepts	*R*	*T*
1	500	5,000	500	15,000
2	4,000	40,000	1,690	6,000
3	8,000	60,000	760	4,000
4	8,000	60,000	1,470	4,000
5	15,000	100,000	4,070	4,000

5.3 Discussion

Our experiments are based on the scale of a realistic Web environment. The numbers of services and concepts in the test sets are at the same or larger scale than that on the Internet today. In addition, we examine two trends for the future Web through Experiments 2 and 3. Overall, our experimental results show that QDA has a very high efficiency. In most cases, optimal service composition can be completed in less than 1 s.

5.4 Web service challenge 2009

We took part in Web Service Challenge 2009 using the QDA proposed in this paper and finished in second place among 9 teams. The teams that took part in Web Service Challenge 2009 competed for effectiveness and efficiency using 5 test sets. The test sets properties and the results of Web Service Challenge 2009 are given in Tables 5 and 6.

The number in Table 6 is the time cost for finding a functionally correct composition. The trace back search incurs heavy overhead in QDA, especially when the sequence length of composition becomes large (in test set 5, the sequence length of composition is greater than 30). The team in the first

Table 6 Web service challenge 2009 results

Test set ID	1. Place (ms)	2. Place (ms)	3. Place (ms)	4. Place (ms)
1	<300	<300	<300	531
2	<300	<300	<300	2,219
3	<300	<300	<300	21,438
4	<300	<300	<300	∞
5	<300	938	<300	∞

place was more efficient than our team because they did a better job on trace back search. Improvement of the efficiency of the trace back routine through some pruning will be our future work. We outperformed the third place team because they could not find the composition with best QoS, although their algorithm did find the functionally correct composition in less time than QDA.

6 Practical relevance

We experiment QDA on artificial test sets. In this Section, we will introduce our investigation of the nature of Web services in Seekda to validate that QDA has a promising application.

In Fig. 10, we use a crawler to collect public Web services from Seekda. After the crawler collects WSDL documents from the Internet, an information extractor extracts functionality information from the documents, such as types, messages, port types, etc. Next, a term generator is used to generate terms using the extracted information. The term generation process has four steps: lexical analysis, tag removal, stop word removal, and vector generator. In the end, the cluster uses a K-Means algorithm to group the services by functionality. We correct deviations manually to let the services in the same group have the same functionality.

After service grouping is done, although the services in the same group have the same functionality, they may have different input/output parameters. We encapsulate services in the same group using a uniform interface, to make use of these services. After encapsulation, services in the same group can be invoked in a uniform way, namely one can be substituted by another in the same group.

It is impossible to obtain all the Web services from Seekda through one query. We obtained all the providers through country enumeration, as the providers of Web Services in Seekda are organized by country, and then found all the Web services through provider query. Although Seekda claims to have more than 28,000 Web services, we were only able to extract 18,250 Web services using the aforementioned method. In addition, there are occasions in which Seekda claims that some provider has several Web services through country enumeration, but when we query Seekda with the provider it says there are no Web services. We call these Web services "lying services" and found there were 4,347 lying services. Moreover, there were 335 Web services with invalid WSDL. In total, we collected 13,568 available Web services from Seekda.

Using the process described at the beginning of this section, we clustered the collected Web services into 300 groups (Web services in the same group have the same or similar functionality). We made sure that Web services in the same group have the same functionality through manual check. We found hundreds of service groups containing services with the same functionality, ranging from several to one hundred. For example, the Airport service group contains 14 Web services which can complete flight checks; the E-mail service group includes 34 Web services which can send e-mail; and the SMS service group has 165 Web services that can send short messages. In the end, we used the encapsulation approach to make the services in the same group replaceable.

Through investigation we can see that there are a large number of Web services with various functionalities and there are indeed Web services which offering the same functionality. Services with the same functionality can be substituted by each other. QDA has a promising application as it organizes Web services to achieve higher functionality and selects Web services to offer better QoS.

7 Conclusion

With the increasing number of Web services, it is a challenge to efficiently build large-scale SOS to meet the required QoS criteria. Though most of the QoS-driven semantic Web service composition is known to be NP-hard, we solve the efficiency issue by developing a polynomial time algorithm (QDA) for shortest sequence composition, using a dynamic programming method. We prove this both by complexity analysis and experiments. Our work provides some advances in the following aspects. We can handle a large-scale service composition in a very short time. We believe this is the biggest contribution of our work. We integrate functional and non-functional consideration together to achieve the optimal solution that satisfies QoS requirements.

In the future, we will continue our research on the composition algorithm. In QDA, all the services are scanned. We will endeavor to work out some pruning criteria to further improve QDA. Also, there are still some constraints of our approach which are worth extending in the future. This depends on some preconditions in practice, e.g., finding the ontology concepts and the mapping between the concepts and service I/O; detecting the QoS of each available service before the composition starts. As the semantic Web develops and QoS becomes more and more important in e-business, these issues must be addressed in the near future.

Acknowledgments This work is supported by the China National High-Tech Project (863) under grants No. 2007AA010306.

References

1. http://webservices.seekda.com/
2. Ponnekanti SR, Fox A (2002) Sword: a developer toolkit for web service composition. In: Proceedings of the 11th international WWW conference (WWW2002)
3. Sirin E, Parsia B, Wu D, Hendler J, Nau D (2004) Htn planning for web service composition using shop2. Web Semant Sci Serv Agents World Wide Web 1(4):377–396
4. McIlraith S, Son T (2002) Adapting golog for composition of semantic web services. In: KR2002, Toulouse, France, pp 482–493, 22–25 April
5. Liang QA, Su SYW (2005) AND/OR graph and search algorithm for discovering composite web services. Int J Web Serv Res 2(4):48–67
6. Hashemian SV, Mavaddat F (2006) A graph-based framework for composition of stateless web services. In: ECOWS, pp 75–86
7. Milanovic N, Malek M (2006) Search strategies for automatic web service composition. Int J Web Serv Res 3(2):1–32
8. Web Service Challenge. http://www.ws-challenge.org
9. Gu Z, Xu B, Li J (2007) Inheritance-aware document-driven service composition CEC/EEE'07. IEEE Computer Society, Japan
10. Yan Y, Xu B, Gu Z (2008) Automatic service composition using AND/OR graph CEC/EEE'08. IEEE Computer Society, Washington D.C.
11. Ran S (2003) A framework for discovering web services with desired quality of services attributes. In: Proceedings of the international conference on web services. CSREA Press, Bogart, GA, pp 208–213
12. Singhera ZU (2004) Extended web services framework to meet non-functional requirements. In: Proceedings of the symposium on applications and the internet workshops. IEEE CS, Silver Spring, MD, pp 334–340
13. Ludwig H, Keller A, Dan A, King RP, Franck R (2003) Web Service Level Agreement (WSLA) language specification. http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf/
14. Dan A, Davis D, Kearney R, Keller A, King R, Kuebler D, Ludwig H, Polan M, Spreitzer M, Youssef A (2004) Web services on demand: WSLA-driven automated management. IBM Syst J 43(1):136–158
15. Skene J, Lamanna DD, Emmerich W (2004) Precise service level agreements. In: Proceedings of the 26th international conference on software engineering (ICSE'04)
16. Zeng L, Benatallah B (2004) QoS-aware middleware for web service composition. IEEE Trans Softw Eng 30(5):311–327
17. Zeng L, Benatallah B, Dumas M, Kalagnanam J, Sheng QZ (2003) Quality driven web services composition. In: Proceedings of the 12th international conference on World Wide Web (WWW). ACM Press, Budapest, Hungary, May 2003
18. Cardoso J (2002) Quality of service and semantic composition of worklows. Ph.D. Thesis, University of Georgia, Georgia
19. Yu T, Zhang Y, Lin K-J (2007) Efficient algorithms for web services selection with end-to-end QoS constraints. ACM Trans Web 1(1), Article 6, May 2007
20. Xiao J, Boutaba R (2005) QoS-aware service composition and adaptation in autonomic communication. IEEE J Select Areas Commun 23(12):2344–2360
21. Alrifai M, Risse T (2009) Combining global optimization with local selection for efficient QoS-aware service composition. WWW 2009, 20–24 April 2009
22. Jaeger MC, Ladner H (2005) Improving the QoS of WS compositions based on redundant services. In: Proceedings of the international conference on next generation web services practices (NWeSP 2005), pp 189–194
23. Lecue F, Mehandjiev N (2009) Towards scalability of quality driven semantic web service composition. In: Proceedings of th IEEE international conference on web services, July 2009
24. Stein S, Payne TR, Jennings NR (2009) Flexible provisioning of web service workflows. ACM Trans Internet Technol 9(1):1–45
25. Google Mashup Editor. http://code.google.com/gme/
26. Microsoft's Popfly. http://www.popfly.com/
27. IBMs QEDWiki. http://services.alphaworks.ibm.com/qedwiki/
28. Yahoo Pipes. http://pipes.yahoo.com/
29. Web Ontology Language. http://www.w3.org/TR/owl-features/
30. Web Service Level Agreements. http://www.research.ibm.com/wsla/
31. Cyc ontology. http://www.cyc.com/cyc/technology/whatiscyc_dir/maptest/