

An Accumulated-QoS-First Search Approach for Semantic Web Service Composition

Sen Luo, Bin Xu, Yixin Yan

Tsinghua National Laboratory for Information Science and Technology

Department of Computer Science and Technology

Tsinghua University, Beijing, 100084, China

{luos, xubin, yanyx}@keg.cs.tsinghua.edu.cn

Abstract

One of the thrilling uses of Web services lies in Web service composition, which integrates existing services into composite ones with higher usage. With increasing complexity of service composition, semantic information, which is well-regulated and easy to be retrieved, has been applied in Web service composition. And QoS has become a critical issue to evaluate the performance of Web applications nowadays. In this paper, a service composition approach combining semantic data-link and QoS together is proposed. The approach aims at finding the composite service that satisfies the requirement with optimal QoS. Accumulated QoS is used as criteria for searching and pruning the search space, thus making the approach efficient. The correctness and efficiency of the approach are proved both in theory and experiment.

1. Introduction

Web service has become a popular technology in system design and implementation along with the development of SOA. And Web service composition, the process of constructing a composite service from existing ones to achieve a specific task, has become a hot research area. Web service composition, especially automatic Web service composition, is a complex and difficult task, so semantic information is used to facilitate the composition. Semantic information helps to construct a more understandable Internet environment as well as an easier way for human-computer interaction. One of the most important targets is to turn the information from implicit to explicit, e.g. in the form of ontology. In this paper, a powerful conceptual model which combining semantic data-link with Web service composition is used.

In the mean while, as more and more Web services are available, QoS has become an important issue in service composition. Using existing Web services, various composite services that satisfy the requirement but have different QoS can be found. To find the QoS-optimal composite service efficiently is a practical problem, and this problem is difficult in the following aspects:

- The number of Web services is getting larger and may reach thirty thousand available. This makes

the search space large for service composition, and brings challenge to the efficiency.

- The scale of semantic information is getting larger too. There may be up to one hundred thousand concepts in ontology, thus the handling of semantic information is very time consuming. And the relations among semantic information make the handling process harder.
- The aggregation of QoS is difficult. Although QoS for individual Web service is given out directly, the QoS of the composite service, which is the target to optimize, must be calculated through aggregation rules.

The problem in this paper is different from previous Web service composition. QoS is not taken into account in WSC (ws-challenge competitions) until 2009 [7]. Unlike other QoS-driven composition approaches, there is not a pre-defined execution plan. The goal is to construct the execution plan as well as make sure the QoS of the composite service is optimal.

Powerful repositories for Web services and semantic information are created to meet the challenge brought by the large scale Web services and semantic information. A breadth-first search approach was proposed in WSC'09 [3]. Although the approach can find the QoS-optimal composite service, all the Web services have to be searched. To be more efficient, this paper focuses on the pruning of search space. After taking a deeper look at the aggregation rules for QoS, each Web service is assigned an accumulated QoS which is used as criteria for searching. In this way there is no need to search all the services. The efficiency and effectiveness of the approach have been proved both in theory and in experiment.

The rest of this paper is organized as follows. Section 2 describes the problem proposed in WSC'10. Section 3 presents the algorithm and the correctness proof is given. Section 4 gives some experimental results. Section 5 discusses the related works. Section 6 gives the conclusion and future works.

2. Overview

In WSC'08, a Web service is defined by its input and output data. Additionally, QoS (only throughput and response time in WSC) is introduced into the Web service

model since WSC'09. A Web service can be represented as:

Service $S = \{D_{in}(S), D_{out}(S), R, T\}$ where

$D_{in}(S) = \{d_i(S) \mid d_i(S) \text{ is an input data type of service } S, \text{ defined by one specific concept in ontology}\};$

$D_{out}(S) = \{d_j(S) \mid d_j(S) \text{ is an output data type of service } S, \text{ defined by one specific concept in ontology}\};$

R = Response time measured in milliseconds;

T = Throughput measured in invocations per second.

The most important features of a service include I/O parameters and QoS. Each I/O parameter of a service can be mapped to one concept in an ontology to express semantic information about the service. And it is important to notice that the service defined above can represent not only a single Web service, but also a composite service. By using the relations between concepts, $\{D_{in}(S), D_{out}(S)\}$ of service S can be extended to $\{D'_{in}(S), D'_{out}(S)\}$ as follows:

$D'_{in}(S) = \{d_i(S) \mid d_i(S) \text{ is an input data type of service } S \text{ or a sub-class of the input data type}\};$

$D'_{out}(S) = \{d_j(S) \mid d_j(S) \text{ is an output data type of service } S \text{ or a super-class of the output data type}\}.$

A request R can be defined with $\{D_{in}(R), D_{out}(R)\}$. For a service with extended input/output $\{D'_{in}, D'_{out}\}$, if $D'_{in}(S) \subseteq D_{in}(R)$, $D'_{out}(S) \supseteq D_{out}(R)$, then the service satisfies the request.

In order to evaluate the QoS of service composition, response time (R) and throughput (T) is introduced into the service model. The QoS of a single Web service is indicated in a SLA file. And the QoS of a composite service is decided by the aggregation rules for different composition structures, such as sequence, flow and case. Let $A_1, \dots, A_n$ denote the single services.

The aggregation rules of response time are:

$$R(sequence) = \sum_{i=1}^n R(A_i)$$

$$R(case) = \min\{R(A_1), \dots, R(A_n)\}$$

$$R(flow) = \max\{R(A_1), \dots, R(A_n)\}$$

The aggregation rules of throughput are:

$$T(sequence) = \min\{R(A_1), \dots, R(A_n)\}$$

$$T(case) = \max\{R(A_1), \dots, R(A_n)\}$$

$$T(flow) = \min\{R(A_1), \dots, R(A_n)\}$$

Given all the available Web services and their QoS, the goal of the service composition problem is to find a composite service to satisfy the request with the optimal QoS (minimum response time or maximum throughput).

3. Accumulated-QoS-First Search Approach

3.1 Algorithm description

This paper proposes an Accumulated-QoS-First search approach to solve the problem. The key idea is to search the services according to the best accumulated QoS. The accumulated QoS of each service is the end-to-end QoS from the beginning of the execution plan to the service, calculated according to the aggregation rule. A priority queue is defined to record all the satisfied services. A service becomes a satisfied service when all of its inputs are satisfied. The priority of a service is determined by its accumulated QoS. For example, the smaller the response time is, the higher the priority will be; the bigger the throughput is, the higher the priority will be. And without loss of generality, in the algorithm description only response time is considered. Data structures in the algorithm are defined as below. There are two main data structures, *DataType* which represents ontology concept and *Service* which represents Web service.

Data Structure

```

1  struct DataType
2  {
3      //the concept the datatype belongs to
4      Concept concept_of_datatype;
5      //the accumulated response time for producing it
6      float response_time;
7      //point to the service which generates it
8      Service ptr_response_time_generator;
9  }
10 struct Service
11 {
12     String name;
13     //the least response time for invoking it
14     float response_time;
15     //time interval between start and complete
16     float self_response_time;
17     List<DataType> input;
18     List<DataType> output;
19 }
```

There are three main steps in the search procedure. The first step is to find the services which are satisfied by the provided data of request and put them into the priority queue, sorted by the accumulated response time.

The second step is to pop a service from the queue. The selected service has two important properties: 1. it is satisfied; 2. it has the minimum accumulated response time among all the services in the priority queue. Then modify accumulated QoS of DataType and Service related to this service.

The third step of the search procedure is pushing new services into the priority queue. In the second step, a new service is popped and put into the solution, so its output data makes some other services satisfied.

The search procedure stops if the queue is empty, which means the request is unable to satisfy; or all the required data of request are satisfied, which means a solution is found.

The main procedure is shown below.

Main Search Procedure

```

1  foreach Service  $S_i$ 
2       $S_i.response\_time = S_i.self\_response\_time$ 
3  foreach DataType  $D_j$ 
4       $D_j.response\_time = \infty$ 
5  foreach DataType  $D_k$  in the provided DataTypes
6       $D_k.response\_time = 0$ 
7      available_data.add( $D_k$ )
8  available_service = getAvalblSrv(available_data)
9  foreach Service  $S_m$  in available_service
10     priority_queue.push( $S_m$ )
11  while(priority_queue is not empty and required
    data are not covered)
12  {
13     Service  $s = priority\_queue.pop()$ 
14     for each DataType  $D_o$  in s.output
15         if( $D_o.response\_time > s.response\_time$ )
16              $D_o.response\_time = s.response\_time$ 
17         available_data.add( $D_o$ )
18     available_service = getAvalblSrv(available_data)
19     foreach Service  $S_n$  in available_service
20          $S_n.response\_time += maxrspnstime(S_n.input)$ 
21         priority_queue.push( $S_n$ )
22     if all required DataType are found {
23         foreach required DataType  $D_r$ 
24             traceback( $D_r$ );
25     }
```

At the last line of the main search procedure shown in the above pseudo code, a trace back search is performed to generate required BEPL format solution.

Trace back Search

```

1  traceback( $D_r$ ){
2      if  $D_r$  belongs to the provided DataTypes
3          return
4      print("<sequence>\n<parallel>")
5      foreach DataType  $D_m$  in
         $D_r.ptr\_response\_time\_generator.input$ 
6          traceback( $D_m$ );
7      print("</parallel>")
8      print("invoke " +
         $D_n.ptr\_response\_time\_generator.name$ )
9      print("</sequence>")
10 }
```

The efficiency of the algorithm is affected both by the main procedure and the trace back search.

3.2 Algorithm Correctness Analysis

This sub-section discusses why the algorithm is able to find the composite service with optimal QoS.

First if there is a composite service that can satisfy the request, the algorithm will not assert there is no solution. And if the algorithm proposes a composite service, the composite service must have the optimal QoS.

Suppose that the algorithm has proposed a solution, and then the solution must be the one with minimum response time. Otherwise there must be another solution with smaller response time. The solution found is denoted as Solution₁ (with response time R_1) and the other one is denoted as Solution₂ (with response time R_2). Suppose service S_{last} is the service that has the greatest accumulated response time in Solution₁. When S_{last} is pushed into the queue, its accumulated response time is R_1 . When S_{last} is popped, at least one of services in Solution₂ is not popped out from the queue; otherwise Solution₂ will be got other than Solution₁. Then at least one service that produces the inputs of the not popped service is not popped, otherwise the service should be popped. Trace it back like this and a service that is satisfied by provided data concept and it is not popped out will be got. This is impossible, as this service is push into the queue when we initial the queue and its response time is smaller than R_1 (otherwise R_2 will not be smaller than R_1). There is a contradiction so the solution found must be the optimal solution.

4. Experiment

To evaluate the performance of the algorithm, the QoS-driven algorithm [3] is chosen to be compared with. The QoS-driven algorithm is the approach which won the second place in WSC'09 [3].

WSC'09 competition testSet [7] is used as test sets. And the configuration of the test machine is: AMD Core 2 CPU 2.10GHz with 2GB RAM, running with Windows 7. The QoS-driven algorithm proposed in [3] is denoted as QDA, and the algorithm presented in this paper is denoted as AQF. To eliminate the effect of trace back search on efficiency, the two algorithms trim the trace back search part, denoted as Trim-QDA and Trim-AQF, thus the comparison can be focused on how the main search procedure affects the efficiency.

TABLE I. EXPERIMENTAL RESULTS

Test Set	Test Sets Properties		Time Cost(ms)			
	Number of Concepts	Number of Web services	QDA	AQF	Trim-QDA	Trim-AQF
1	5,000	500	110	55	94	47
2	40,000	4,000	162	85	109	78
3	60,000	8,000	245	95	188	93
4	60,000	8,000	350	165	265	156
5	100,000	15,000	700	550	354	159

As shown in the experimental results, trace back search is indeed a heavy overhead especially in test set 5, since the search might make service appears multiple times in the solution. For the composition efficiency, AQF is almost two times faster than QDA in most test sets. This is mainly because QDA has to search all the services to get the optimal solution while AQF can do a lot of pruning using Accumulated-QoS-First manner.

5. Related Work

Service composition can be divided into two categories, one is data-driven service composition and the other is QoS-driven service composition.

In data-driven service composition, not only the involved service should be determined, but also the work flow that organizes these services should be carried out. Liang [6], Gu [1] and Yan [2]'s works can be categorized to the data-driven service composition.

With the growing number of alternative Web services that provide the same functionality, QoS-driven Web service composition has been extensively studied in the past few years. Zeng [4] and Yu T[5]'s works fall in QoS-driven service composition.

The problem discussed in this paper combine characteristic of both data-driven and QoS-driven composition, which makes it different from mentioned previous works. The most similar one is the work done by Yan [3].

6. Conclusion and Future Work

An Accumulated-QoS-First search approach is proposed in this paper to find the composite service with optimal QoS. This approach is very efficient and effective, which have been shown both theoretically and experimentally. To further improve the efficiency, an optimized trace back search is necessary to reduce the quantity of the services appeared in the solution.

7. Acknowledge

This work is supported by China National High-Tech Project (863) under grant No. 2009AA01Z120 and 2007AA010306.

References

- [1] Zhifeng Gu, Bin Xu and Juanzi Li. Inheritance-Aware Document-Driven Service Composition. CEC/EEE'07, Tokyo, Japan, 2007. IEEE Computer Society.
- [2] Y Yan, B Xu and Z Gu. Automatic Service Composition using AND/OR Graph. CEC/EEE'08, Washington D.C. 2008. IEEE Computer Society.
- [3] Y Yan, B Xu, Z Gu and S Luo. A QoS-Driven Approach for Semantic Service Composition. CEC/EEE'09, Vienna, Austria, 2009. IEEE Computer Society.
- [4] L Zeng, B Benatallah. QoS-Aware Middleware for Web Service Composition. IEEE Transactions on Software Engineering, Vol. 30, No. 5, May 2004.
- [5] Yu T, Zhang Y, and Lin KJ. Efficient Algorithms for Web Services Selection with End-to-End QoS Constraints. In ACM Transactions on the Web, Vol. 1, No. 1, Article 6, May 2007.
- [6] Q. A. Liang and S. Y. W. Su. AND/OR graph and search algorithm for discovering composite web service. In International Journal of Web Service Research, vol. 2, no.4, pages 48–67, 2005.
- [7] Web Service Challenge 2009. <http://ws-challenge.georgetown.edu/wsc09>