Chapter 22
Efficient Composition of Semantic Web Services with End-to-End QoS Optimization

Bin Xu and Sen Luo

Abstract The efficiency of QoS-aware service composition is important since most service composition problems are known to be NP-hard. With the growing number of Web services, service composition is like a decision problem on selecting services or/and execution plans to satisfy the users end-to-end QoS requirements (e.g. response time, throughput). Composite services with the same functionality may have different execution plans, which may cause different end-to-end QoS. A model combining semantic data-links and QoS is proposed, which leads to an efficient approach to automatic construction of a composite service with optimal end-to-end QoS. The approach is based on a greedy algorithm to select both services and execution plans for composite services. Empirical and theoretical analysis of the approach show that its time complexity is $O(mn^2)$ for a repository with n services and an ontology with m concepts. Moreover, the approach takes linear time in practice when using an index to search services in the repository.

22.1 Introduction

In the service-oriented computing paradigm, single web services can be combined to create value-added services for business applications. Service compositions have put into industrial practice in many areas like e-commerce, supply chain management, finance, and travel. With the growing number of web servers with different quality parameters (QoS), the service composition problem becomes a QoS-aware service composition problem, which is to find a composite service with the optimal end-to-end QoS.

The QoS-aware service composition problem has been discussed in many studies [1, 11–13]. Given an abstract composition request (execution plan), which can be

B. Xu (✉) · S. Luo
Department of Computer Science and Technology, Tsinghua University, Beijing, China
e-mail: xubin@keg.cs.tsinghua.edu.cn; luos@keg.cs.tsinghua.edu.cn

M.B. Blake et al. (eds.), *Semantic Web Services*, DOI 10.1007/978-3-642-28735-0_22,

stated in a work flow language (e.g. BPEL [8]), each abstract service (task node) in the execution plan has a candidate service list. The goal is to select one concrete service for each abstract service such that the aggregated QoS satisfies the user's end-to-end QoS requirement.

However, the problem we discuss about, The QoS-aware service composition with data-links, is not based on an abstract composition. The first goal of the service composition is to find a proper execution plan. The execution plan may contain many kinds of relationships between services such as sequence, parallel and switch. The services in the composite service should be linked by data, which requires that former services' output can satisfy successive service's input by semantics. In practice, several composite services may satisfy the request, but with different QoS. Thus, the second goal is to find the composite service with the best end-to-end QoS. The QoS attributes include response time, throughput, cost, availability, reliability and etc.

22.2 Approach: QoS-Aware Service Composition

The abstract composition based QoS-aware service composition problem can be mapped to a Multi-Choice Multidimensional Knapsack problem , which is known to be NP-hard in the strong sense [5]. Consequently, an optimal solution may not be expected to be found in a reasonable amount of time [4]; so many approximation algorithms have been proposed. Zeng et al. [12,13] used global planning to optimize multiple QoS criteria. Yu et al. [11] proposed a broker-based architecture as well as a heuristic algorithm to optimize the end-to-end QoS with multiple QoS constrains. Alrifai and Risse [1] gave a method to handle the global QoS requirements through combining global optimization with local selection.

The QoS-aware service composition model with the data-links is shown in Fig. 22.1. The composition request includes a provided data list and a required data list. The goal is to find an execution plan with data-links from the service repository and to get the optimal end-to-end QoS. Followings are some basic definitions:

Definition 22.1. Service S $= \{D_{in}, D_{out}, Q_s\}$ where

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

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

$Q_s(S) = \{q_r(S) | q_r(S)$ is a QoS attribute of service S, such as cost or response time$\}$

The most important features of a service include the 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. Thus, the QoS can represent any kind of non-functional property.

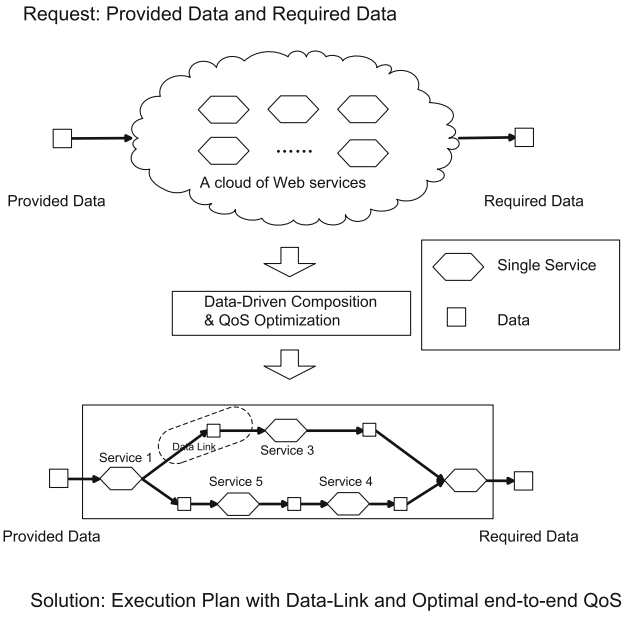

Fig. 22.1 Conceptual overview

Definition 22.2. Composite Service $CS = \{D_{in}, D_{out}, P, Q_{cs}\}$ where

$D_{in}(CS)$= $\{d_i(CS)|d_i(CS)$ is an input data type of CS, defined by one specific concept in an ontology$\}$

$D_{out}(CS) = \{d_j(CS)|d_j(CS)$ is an output data type of CS, defined by one specific concept in an ontology$\}$. If CS is composed by S_1 to S_n, then $D_{out}(CS) = \{D_{out}(S_1), D_{out}(S_2), \ldots\ldots, D_{out}(S_n)\}$

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

$Q_{cs}(CS)$= $\{q_r(CS)|q_r(CS)$ is end-to-end QoS attribute of CS, such as total cost or total response time$\}$

The problem of QoS-aware service composition with data-links can be stated as follows:

For a given composition request $R = \{D_{in}(R), D_{out}(R), q_r\}$ and a given service repository $SS = S_1, S_2, \ldots\ldots, S_n$, find a composite service CS such that:

1. $D_{in}(R) \supseteq D_{in}(CS), D_{out}(R) \subseteq D_{out}(CS)$;
2. The end-to-end QoS attribute $q_r(CS)$ is optimal.

This problem is concerned with a single QoS attribute of a composite service, not the integration of all the QoS attributes. The system can find the composite service with the minimum response time or maximum throughput. A utility function to weight different QoS attributes is not discussed in this paper.

22.3 Solution: Greedy Algorithm for QoS-Aware Service Composition

A greedy algorithm (GA) is used to solve this problem. An example of GA search process is shown in Fig. 22.2.

The GA has two values for each QoS attribute, self value and the accumulated value. For the example in Fig. 22.2(1), the self value of the response time for service A (Srv A) is 15; while in Fig. 22.2(3), the accumulated value of the response time for service A is 35. Since service A is the successor of service B, the response time of service B (20) plus service A (15) is the accumulated value (35) of service A. Thus, the accumulated value of each service is the end-to-end QoS from the beginning of the execution plan to this service, calculated according to Table 22.1

The given I/O data shown in Fig. 22.2(1) consists of data #1, #2, #3 *and* #4, while the required output data type is data #8, #9 *and* #10. In Fig. 22.2(2), the

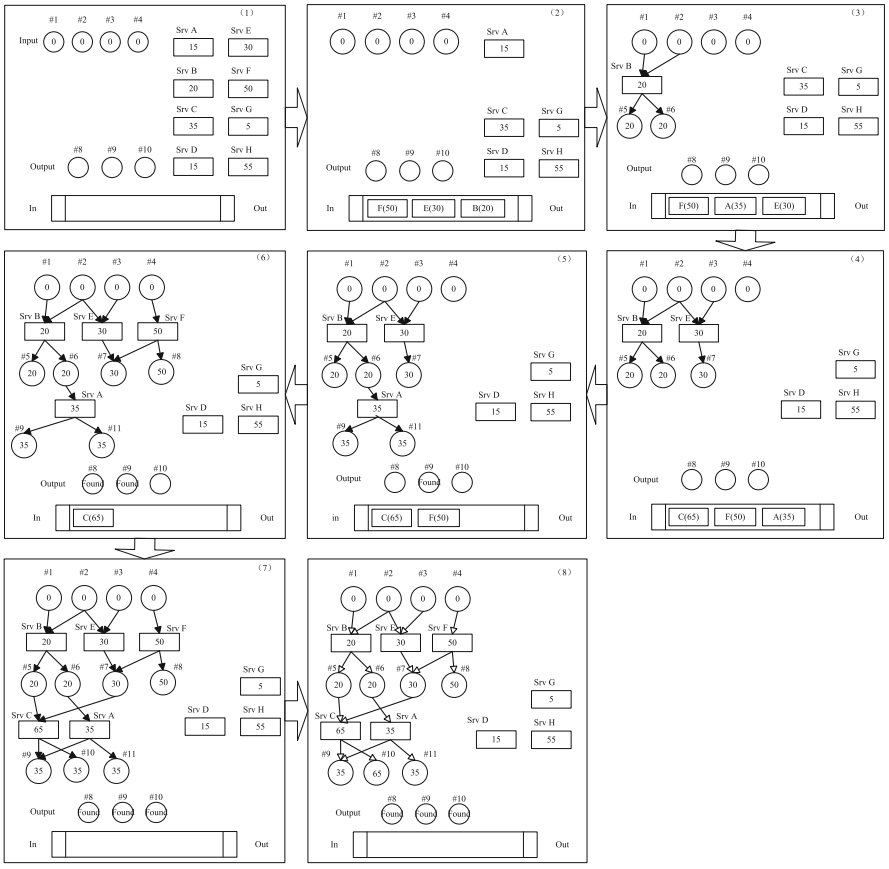

Fig. 22.2 GA search process

Table 22.1 QoS calculation for various composite structures

QoS attribute	Composition structure	Calculation
Response time	Parallel	$R = max\{R_i\}$
	Sequence	$R = \sum_{i=1}^{n}\{R_i\}$
	Switch	$R = min\{R_i\}$
Throughput	Parallel	$T = min\{T_i\}$
	Sequence	$T = min\{T_i\}$
	Switch	$T = max\{T_i\}$

satisfied services (B, E, F) are pushed into the queue sorted by the accumulated response times. In Fig. 22.2(3), the service with the minimum accumulated response time is popped and added into the solution (execution plan), with newly satisfied service A pushed into the queue. Then in Fig. 22.2(4), the first service in the queue, service E, is popped. The procedure is repeated in Fig. 22.2(5, 6) until all the required output data types are found in Fig. 22.2(7). Then, a trace back procedure helps to find the solution shown in Fig. 22.2(8).

22.4 Lessons Learned

22.4.1 Algorithm Description

The key idea is to select the service with the best accumulated QoS. A priority queue is defined to record all the satisfied services. A service becomes satisfied only when all of its inputs are satisfied. The priority of a service is determined by its accumulated QoS. A smaller accumulated response time gives a higher priority. A larger accumulated throughput also gives a higher priority.

The DataType in the algorithm is defined in Listing 22.1.

The main procedure is shown in Listing 22.2.

A trace back function is used at the end of the main process to generate BEPL format solution, which is described in Listing 22.3.

22.4.2 Algorithm Correctness Analysis

If the algorithm has proposed a solution, the solution must have the minimum accumulated response time. The solution is denoted as $Solution_1$ (with accumulated response time R_1). Suppose there is another solution with a smaller accumulated response time which will be denoted as $Solution_2$ (with a smaller accumulated response time R_2). If service S_{last} is the last popped service of $Solution_1$, then when S_{last} is pushed into the queue, its accumulated response time

```
struct DataType
{
    //the concept the data type belongs to
    Concept concept_of_data type;
    //the accumulated response time for producing it
    float response_time;
    //point to the service which generates it
    Service ptr_response_time_generator;
}
struct Service
{
    String name;
    //the least response time for invoking it
    float response_time;
    //time interval between start and complete
    float self_response_time;
    List<DataType> input;
    List<DataType> output;
}
```

Listing 22.1 Definition of DataType

```
for each Service S_i
  S_i.response_time = S_i.self_response_time
for each DataType D_j
    D_j.response_time = infinity
for each DataType D_k in the provided DataTypes
    D_k.response_time = 0
    available_data.add(D_k)
available_service = getAvalblSrv(available_data)
for each Service S_m in available_service
    priority_queue.push(S_m)
while(priority_queue is not empty and required data are not covered){
  Service s = priority_queue.pop()
    for each DataType D_o in s.output
      if(D_o.response_time > s.response_time)
          D_o.response_time = s.response_time
        available_data.add(D_o)
    available_service =getAvalblSrv(available_data)
    for each Service S_n in available_service
        S_n.response_time+=maxResponseTime(S_n.input)
        priority_queue.push(S_n)
    if all required DataType are found
        for each required DataType D_r
            traceback(D_r);
}
```

Listing 22.2 Main GA process

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

Listing 22.3 Trace back Search

is R_1. When S_{last} is popped, at least one of the services in $Solution_2$, denoted as S_0, must not have been popped from the queue; otherwise all the services in $Solution_2$ would have been popped to get $Solution_2$ rather than $Solution_1$. Then at least one of

services that produce the inputs of S_0 is not popped, otherwise S_0 would be popped. A trace back (a service is not popped because at least one of services that produce the inputs of it is not popped) will show a service that is satisfied by the provided data but not popped. However this is impossible, since this service was pushed into the queue when the queue was initialized and its accumulated response time was smaller than R_1 (otherwise R_2 will not be smaller than R_1). There is a contradiction so $Solution_1$ must be the optimal solution.

22.4.3 Algorithm Correctness Analysis

Suppose n is the number of the overall services and m is the number of the overall concepts in the ontology.

The algorithm has several main operations, denoted as SCAN (scan unused services to find satisfied services), POP (pop out a service from the priority queue) and PUSH (push a service into the priority queue).

For the POP operation, every service is popped from the queue at most once, so there are at most n POP operations. Every POP operation takes $O(1)$ time, so POP operations take $O(1)n = O(n)$ time.

The PUSH operation also has at most n PUSH operations. Each PUSH operation takes $O(m)$ time to update the corresponding response times and takes $O(logn)$ time to put the service at the right position. Thus all the Push operations take $(O(m) + O(logn))n = O(mn + nlogn)$ time.

The SCAN operation takes place at most n times regardless of the original SCAN, since there are n services overall and the SCAN operation is executed right after a POP operation. In each SCAN operation, the time to determine whether a service is satisfied is $O(m)$ and there are at most n unused services. So each SCAN operation takes $O(m)n = O(mn)$ time. All the SCAN operations take $O(mn) * O(n) = O(mn^2)$ time.

An index is used to record all the relationships between the I/O data types and the services. Assume a data type has a constant number of services, c, on average. Then $O(c)$ time is needed to update the response time in PUSH and to determine whether a service is satisfied in SCAN. The SCAN operation needs only to test at most c services to determine the satisfied services. The time complexity is then $O(cn + nlogn)$ for PUSH and $O(c^2n)$ for SCAN.

In summary, the time complexity is $O(n) + O(mn + nlogn) + O(mn^2) = O(mn^2)$. If an index is used, the time complexity becomes $O(n) + O(cn + nlogn) + O(c^2n) = O(nlogn)$. This is a very loose upper bound, which only happens in the worst case. In the best case, all the popped services are services in the optimal solution and the algorithm takes only a constant amount of time (assume the number of service in the optimal solution is n_0 and there are at most cn_0 PUSH operations, n_0 POP operations and n_0 SCAN operations. The total time is then $O(cn_0) + O(cn_0 + n_0logn_0) + O(c^2n_0)$, which is constant).

Table 22.2 Concepts and services increase with the same ratio

TestSet	Test set properties		Time cost (ms)	
	Number of concepts	Number of web services	QDA	GA
1	37,500	500	125	78
2	75,000	1,000	300	78
3	112,500	1,500	600	93
4	150,000	2,000	800	109
5	187,500	2,500	950	109
6	225,000	3,000	1,040	78

22.4.4 Performance Evaluation

The algorithm performance was tested based on the requirements of the annual Web Service Challenge(WS-Challenge) [9] which focuses on the semantic composition of Web services with QoS. WS-Challenge provides a set of standard testing tools and data sets.

WS-Challenge uses a generator to generate a test set. Each test set includes four input files: (1) Services.wsdl provides the available Web services; (2) Taxonomy.owl [6] provides all the concepts in the ontology with every input/output data type of the Web services defined as an instance of a concept; (3) Servicelevelagreements.wsla [7] provides the self QoS values (response time and throughput) of the Web services; and (4) Query.wsdl gives the user requests including the provided and required data types. The generator also gives a standard result for each test set.

The tests first used the generator to generate 18 test sets. Then the composition algorithm and other algorithm are used to evaluate on the 18 test sets with the time cost recorded during the composition procedure. The results are checked against the results provided by WS-Challenge.

The GA performance is compared to that of QDA (QoS-driven algorithm) [10] which placed second in the WS-Challenge 2009 performance evaluation.

The tests are on a machine with Intel Core 2 CPU 1.83 GHz, 1 GB RAM, running Windows XP.

The 18 test sets generated by the generator each has different scale of Web services and ontology concepts. There are about 20,000 Web services available on the Internet [2], with the most widely used "OpenCyc" ontology [3] having about 150,000 concepts.

The 1–6 test sets are designed with different numbers of concepts and Web services but a constant ratio between them. Table 22.2 lists the settings for these six test sets and the time cost for the QDA and GA algorithms.

The results in Table 22.2 show that the QDA time cost increases with the scale of the test sets, while the GA time cost is constant at about 100 ms, even for 225,000 concepts and 3,000 services. In test set 6, the GA is more than ten times faster than the QDA.

The 7–12 test sets fixed the number of Web services at 20,000 and changed the ontology concepts from 50,000 to 300,000 with the interval of 50,000. These sets

Table 22.3 Concepts increase while the services remain constant

TestSet	Test set properties		Time cost (ms)	
	Number of concepts	Number of web services	QDA	GA
7	50,000	20,000	600	234
8	100,000	20,000	800	141
9	150,000	20,000	1,100	140
10	200,000	20,000	1,220	188
11	250,000	20,000	1,440	219
12	300,000	20,000	1,680	210

Table 22.4 Services increase while the concepts remain constant

TestSet	Test set properties		Time cost (ms)	
	Number of concepts	Number of web services	QDA	GA
13	150,000	2,000	580	78
14	150,000	4,000	620	78
15	150,000	6,000	750	94
16	150,000	8,000	880	109
17	150,000	10,000	960	125
18	150,000	12,000	1,040	125

are closer to development trends on the real Web, with the semantic Web expanding rapidly and more ontologies appearing, the number of Web services has remained stable in recent years. Table 22.3 lists the results for these six test sets, and the time costs.

The results in Table 22.3 show that the QDA time cost increases linearly with the increasing number of concepts. The GA is more efficient with the time cost constant at about 200 ms. With test set 12, the GA was about eight times faster than the QDA.

In the 13–18 test sets, the number of concepts was held constant while the number of Web services increased. In the future, when most concepts are well described by ontologies, the number of Web services may increase because of new businesses. Table 22.4 lists the results for these six test sets, and the time costs.

The results in Table 22.4 show that when the number of concepts is constant, the QDA time cost changes linearly with the increasing number of services. The GA is again stable and efficient at about 100 ms. In test set 18, the GA is about eight times faster than the QDA.

For all 18 test sets, the QoS values in the composition result are the same as the standard results with the optimal QoS.

In all 18 test sets, the GA algorithm was very efficient, performing the composition in no more than 219 ms. Test sets 1–6 and 13–18 had time less than 125 ms. The GA is two to ten times faster than the QDA with the GA performance very stable even with large number of services and concepts. The GA always selects the service with the best accumulated QoS, so the services in the solution need not be updated in the following search process which improves the efficiency. The QDA

Table 22.5 Trace back search overhead

TestSet	Test set properties		Time cost (ms)	
	Number of concepts	Number of web services	GA	Trim-GA
1	5,000	500	55	47
2	40,000	4,000	85	78
3	60,000	8,000	95	93
4	60,000	8,000	165	156
5	100,000	15,000	550	159

is an iterative search process with new services added to the solution layer by layer until all the output data is found, so many redundant services are included. The GA search space is also much smaller than that of the QDA.

22.4.5 Advantages and Disadvantages

In context of service composition, we have found the following advantages in our algorithm:

- A QoS-aware service composition model with semantic data-links is proposed. Unlike the QoS-aware service composition problem having an abstract composition as a request, this model uses provided data and required data as request. Service composition request can be given out more easier by provided and required data than by an abstract composition. The QoS optimization in this model finds the execution plan as well as selects the services for the optimal end-to-end QoS.
- An efficient data-driven, QoS-optimized service composition algorithm is given. A greedy algorithm is used to select services from a repository of services and to construct execution plan to ensure the optimal end-to-end QoS. A theoretical analysis of the algorithm shows its time complexity is $O(mn^2)$, with linear time dependence in practice by using indexing. Test data sets from the Web Service Challenge [9] are used to compare with other algorithms [10]. Tests show that the algorithm significantly outperforms existing algorithms in terms of composition efficiency while achieving optimal end-to-end QoS.

22.4.6 Conclusions and Future Work

Trace back search is indeed a heavy overhead as shown in Table 22.5 (Trim-GA denotes GA algorithm without trace back search part). Test sets used here are the five test sets used in the WS-Challenge 2009.

To further improve the efficiency, an optimized trace back search is necessary to reduce the quantity of the services appeared in the solution. So our future work will be trace back search optimization.

22.5 Summary

This chapter presents a QoS-aware service composition model with data-links. The end-to-end QoS optimization in this model finds the execution plan and selects services with the optimal accumulated QoS. A greedy algorithm is used to select services for a given composition request. Tests show that the algorithm significantly outperforms existing algorithms in terms of composition time cost while still achieving the optimal end-to-end QoS. The algorithm was on problems having the scale of a real Web (20,000 services and 300,000 concepts). The algorithm will be applied to Web services with dynamic QoS in future work to satisfy real-time composition requests. A run-time composition engine will be developed to integrate real services.

References

1. M. Alrifai, T. Risse, Combining global optimization with local selection for efficient qos-aware service composition, in *WWW*, Madrid, Spain, 2009
2. E. Al-Masri, Q.H. Mahmoud, Investigating web services on the world wide web, in *WWW*, 2008
3. Cyc Ontology (2010), http://www.cyc.com/cyc/
4. I. Maros, *Computational Techniques of the Simplex Method*. International Series in Operations Research & Management Science, Vol. 61 (Kluwer, Boston, 2003)
5. D. Pisinger, *Algorithms for Knapsack Problems*, PhD thesis, University of Copenhagen, 1995
6. Web Ontology Language (2004), http://www.w3.org/TR/owl-features/
7. Web Service Level Agreements (2003), http://www.research.ibm.com/wsla/
8. Web Services Business Process Execution Language (2007), http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf
9. Web Service Challenge (2010), http://ws-challenge.georgetown.edu/wsc10/
10. B. Xu, Y. Yan, An efficient qos-driven service composition approach for large-scale service oriented systems, in *SOCA*, Taipei, 2009
11. T. Yu, Y. Zhang, K.J. Lin, Efficient algorithms for web services selection with end-to-end qos constraints. ACM Trans. Web **1**(1), 6-es (2007)
12. L. Zeng, B. Benatallah, Qos-aware middleware for web service composition. IEEE Trans. Softw. Eng. **30**(5), 311–327 (2004)
13. L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, Q.Z. Sheng, Quality driven web services composition, in *WWW*, Budapest, Hungary, 2003