TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 08/14 pp678-686
Volume 15, Number 6, December 2010

Efficient Composition of Semantic Web Services with End-to-End QoS Optimization[*]

XU Bin (许 斌)[**], LUO Sen (罗 森), YAN Yixin (闫奕歆)

Tsinghua National Laboratory for Information Science and Technology, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

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. This paper presents a model combining semantic data-links and QoS, 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 analyses 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 increases linearly in time when using an index to search services in the repository. Tests with a repository with 20 000 services and an ontology with 300 000 concepts show that the algorithm significantly outperforms current existing algorithms in terms of composition efficiency while achieving optimal end-to-end QoS.

Key words: semantic web service; web service composition; QoS; optimization

Introduction

In the service-oriented computing paradigm, single web services can be combined to create value-added services for business applications. Service compositions have been 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[1-4]. Given an abstract composition request (execution plan), which can be stated in a workflow-like language (e.g., BPEL[5]), 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. Thus, the problem can be mapped to a multi-choice multidimensional knapsack problem, which is known to be NP-hard in the strong sense[6]. Consequently, an optimal solution may not be expected to be found in a reasonable amount of time[7]; so many approximation algorithms have been proposed. Zeng et al.[2,4] used global planning to optimize multiple QoS criteria. Yu et al.[3] proposed a broker-based architecture as well as a heuristic algorithm to optimize the end-to-end QoS with multiple QoS constrains. Alrifai

Received: 2010-09-16

* Supported by the National High-Tech Research and Development (863) Program of China (Nos. 2009AA01Z120 and 2007AA010306)

** To whom correspondence should be addressed.
E-mail: xubin@tsinghua.edu.cn; Tel: 86-10-62781461

and Risse[1] gave a method to handle the global QoS requirements through combining global optimization with local selection.

The QoS-aware service composition problem is more concerned with I/O than the abstract composition. For example, a traveler with a GPS device wants to find a service to know the weather of his location as quickly as possible. Thus, the system is to find a composite service with minimal response time, whose input data is a location (latitude and longitude) and output data is a weather report.

The example composite service shown in Fig. 1 includes one Google web service (WS) and one Yahoo web service. The input data for the Google service is latitude and longitude from a GPS device with the output data being the city name, which acts as input data to the Yahoo service. The output data of the Yahoo service is the weather report, which is the data desired by the traveler. The total response time for both services is the composite services' response times.

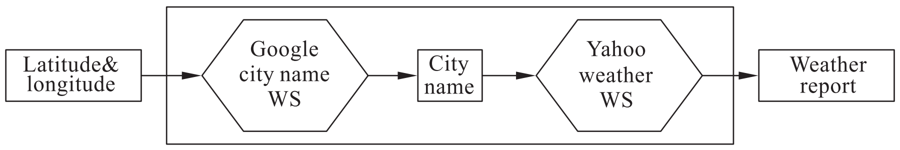

Fig. 1 Example of QoS-aware service composition with data-links

The QoS-aware service composition model with the data-links is shown in Fig. 2. 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. For example in Fig. 1, the execution plan is the services and data in the dotted box. The QoS is the response time.

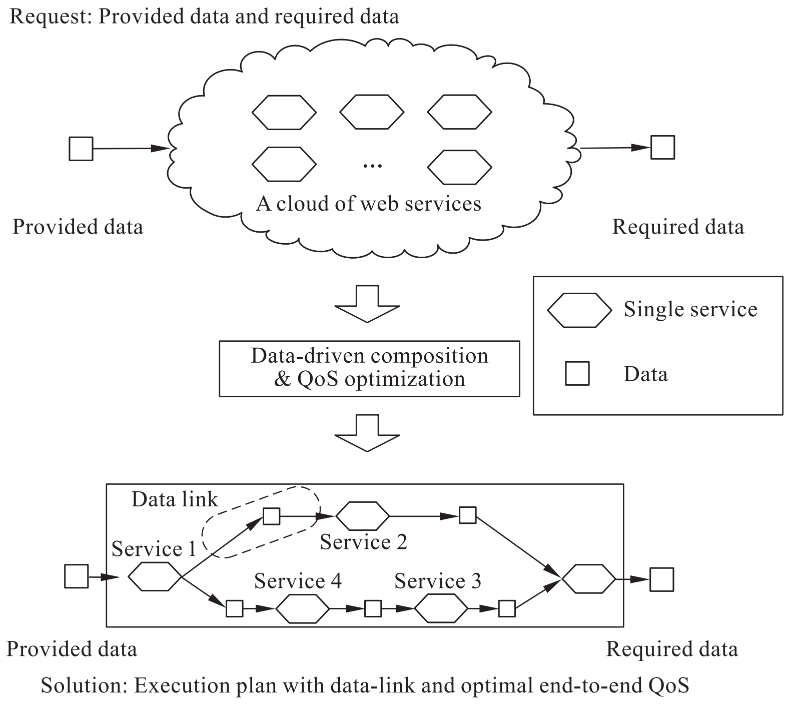
Request: Provided data and required data
A cloud of web services
Provided data
Required data
Single service
Data
Data-driven composition & QoS optimization
Data link
Service 1
Service 2
Service 4
Service 3
Provided data
Required data
Solution: Execution plan with data-link and optimal end-to-end QoS

Fig. 2 Conceptual overview

The QoS-aware service composition with data-links is not based on an abstract composition. That is, users do not need to provide a pre-defined execution plan as a request. 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 the former service's output can satisfy successive services' 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, and reliability.

Lecue and Mehandjiev[8] proposed semantic links between services to evaluate the matching degree of data between services. Their work is also based on the input of an abstract composition with semantic links regarded as a type of non-functional qualities evaluated by the calculating matching scores according to the I/O of the services. However, semantic links can not replace restricted data dependencies (data-links) because a service cannot be invoked until all its inputs are fully satisfied.

The contributions of this paper are:

● 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. The user giving service composition request can more easily give provided and required data than 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 an 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 datasets from the Web Service Challenge[9] are used to compare with other algorithms[10,11]. Tests show that the algorithm significantly outperforms existing algorithms in terms of composition efficiency while achieving optimal end-to-end QoS.

1 Problem Definitions

1.1 Service vs. composite service

Definition 1 Service $S=\{D_{\text{in}},D_{\text{out}},Q_{\text{s}}\}$ where $D_{\text{in}}(S)=\{d_i(S)\,|\,d_i(S)$ is an input data type of service *S*, defined by one specific concept in an ontology$\}$, $D_{\text{out}}(S)=\{d_j(S)|d_j(S)$ is an output data type of service *S*, defined by one specific concept in an ontology$\}$, $Q_{\text{s}}(S)=\{q_{\text{r}}(S)\,|\,q_{\text{r}}(S)$ 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.

Definition 2 Composite service $\text{CS}=\{D_{\text{in}},D_{\text{out}},P,Q_{\text{cs}}\}$ where $D_{\text{in}}(\text{CS})=\{d_i(\text{CS})\,|\,d_i(\text{CS})$ is an input data type of CS, defined by one specific concept in an ontology$\}$. $D_{\text{out}}(\text{CS})=\{d_j(\text{CS})\,|\,d_j(\text{CS})$ 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 $D_{\text{out}}(\text{CS})=\{D_{\text{out}}(S_1),D_{\text{out}}(S_2),\cdots,D_{\text{out}}(S_n)\}$. *P* is 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. $Q_{\text{cs}}(\text{CS})=\{q_r(\text{CS})\,|\,q_r(\text{CS})$ is an end-to-end QoS attribute of CS, such as total cost or total response time$\}$.

There are many QoS attributes which can be used to evaluate the services. Some of them can be considered at the design time, such as availability, extensibility, adaptability, testability, operability, deployability, and modifiability, while some of them are used at runtime, such as the response time and throughput. However, this analysis is not concerned about how the QoS of a single service is measured, but how to study and optimize the end-to-end QoS of a composite service.

1.2 Composite service execution plan

The execution plan is composed of sequences, parallels, and switches in structures. A sequence consists of services which are invoked in order. A parallel consists of services which are invoked at the same time. A switch consists of services which can be selectively invoked. The basic structures can be nested to form a complex structure. Adjacent services have data-links between them. The whole execution plan can be expressed using BPEL.

In the sample execution plan in Fig. 3, a user request includes the provided data (d1, d2, and d3) and the required data (d5 and d6). Services A and B are in one sequence. Service A's inputs are d1 and d2, while its output d4 acts as an input to service B. B's output is d5. Services C and D are in a switch, so C or D can generate d6. The sequence and switch can be invoked in parallel to get both d5 and d6. The execution plan is shown in Fig. 4 with the QoS (response time) but without the data.

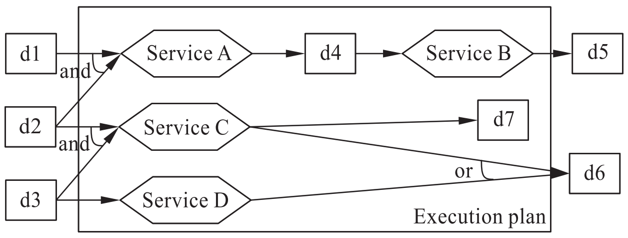

Fig. 3 Example of execution plan

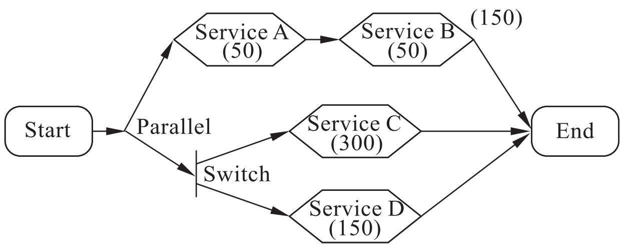

Fig. 4 Execution plan with QoS (response time)

1.3 End-to-end QoS calculation

The end-to-end QoS is calculated based on the execution plan. Table 1 lists the calculation at formulas for three basic composite structures. The two typical, most commonly used QoS (response time and throughput) are used as examples. The response time evaluates the time from sending a request to a composite service to receiving the response message. When the message passes through two services in sequence, the response time should be the sum of the two services' times.

Table 1 QoS calculation for various composite structures

QoS attribute	Composition structure	Calculation
Response time	Parallel	$R = \max\{R_1\}$
	Sequence	$R = \sum_{i=1}^{n} R_1$
	Switch	$R = \min\{R_1\}$
Throughput	Parallel	$T = \min\{T_1\}$
	Sequence	$T = \min\{T_1\}$
	Switch	$T = \max\{T_1\}$
Availability / Reliability	Parallel	$A = \prod_{i=1}^{n} A_1$
	Sequence	$A = \prod_{i=1}^{n} A_1$
	Switch	$A = \max\{A_1\}$
Cost (Price)	Parallel	$C = \prod_{i=1}^{n} C_1$
	Sequence	$C = \prod_{i=1}^{n} C_1$
	Switch	$C = \min\{C_1\}$

When the two services are invoked in parallel, the larger response time of the two is used as the composite service's response time. While two services are in a switch, the best one with minimal response time is used.

Throughput is the maximum amount of information passing through a composite service. Thus, the throughput is the minimum value in a sequence of two services. When two services are in a parallel, the throughput is the minimum value which is the bottleneck of the composite service. When two services are in a switch, the throughput is the one which has maximum throughput.

Referring to the response time calculation of the execution plan in Fig. 4, the switch uses service D whose response time is 150 ms (better than service D's 300 ms). In the sequence of services A and B, the total response time is 100 ms (50 ms plus 50 ms). In the parallel structure, the response time is the maximum one (150 ms). So the total response time of this execution plan is 150 ms.

1.4 Problem statement

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

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

(1) $D_{\text{in}}(R) \supseteq D_{\text{in}}(\text{CS})$, $D_{\text{out}}(R) \subseteq D_{\text{out}}(\text{CS})$;

(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 of weight different QoS attributes is not discussed in this paper.

2 Greedy Algorithm for QoS-Aware Service Composition

2.1 Algorithm description

A greedy algorithm (GA) is used to solve this problem. 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 GA has two values for each QoS attribute, self value and the accumulated value. For the example in Fig. 5a, the self value of the response time for service A (Srv A) is 15; while in Fig. 5c, 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 1.

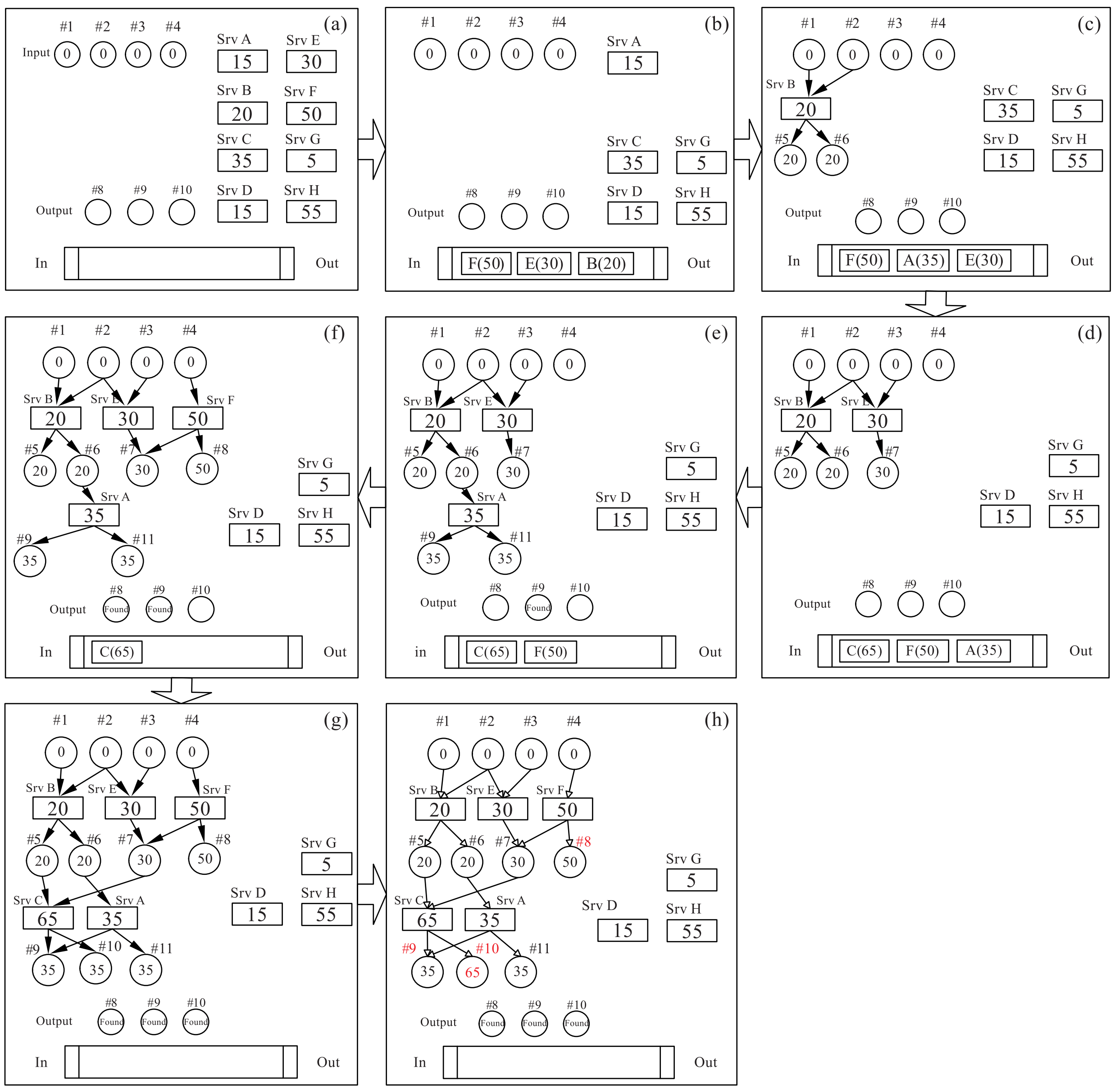

Fig. 5 GA search process

An intuitive example is given to describe the GA to find the composition with the minimum response time in Fig. 5. The given I/O data shown in Fig. 5a consists of data #1, #2, #3, and #4, while the required output data type is data #8, #9, and #10. In Fig. 5b, the satisfied services (B, E, and F) are pushed into the queue sorted by the accumulated response times. In Fig. 5c, 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. 5d, the first service in the queue, service E, is popped. The procedure is repeated in Figs.

5e and 5f until all the required output data types are found in Fig. 5g. Then, a trace back procedure helps to find the solution shown in Fig. 5h.

This example shows that new services will be added to the solution until all the required data is found. Each popped service from the queue always has the best accumulated QoS, so every time new data is found, it also has the best accumulated QoS. Thus the new added data gives a list of services whose inputs are satisfied which are then put into the queue.

A search procedure is used to find a solution with the minimum response time. The first step is to find the services which are satisfied by the provided data and put them into the priority queue.

The second step is to add services into the solution which is popped from the queue. The added service must have the minimum accumulated response time among all the services in the priority queue.

The third step of the search procedure is to push new services with the accumulated QoS into the priority queue. The second step popped a new service into the solution, so its output data could be used by other services. The service response time is always the accumulated value. For example, the response time of service A itself is 15 ms, but service A is invoked after service B whose response time is 20 ms. Thus, the accumulated response time of service A is 15 ms plus 20 ms (35 ms).

The second and third steps are repeated until all the required output data are found.

When determining whether a service is satisfied, the ontology is used in the composition to define the parameter types of the service 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. A service is satisfied once all its input data types are connected. The DataType in the algorithm is defined as below.

Definition of DataType

```
struct DataType
{
//the concept the datatype belongs to
    Concept concept_of_datatype;
//the accumulated response time for producing it
    float response_time;
//point to the service which generates it
    Service ptr_response_time_generator;
}
```

The main GA process is shown below.

Main GA Process

```
foreach Service $S_i$
    $S_i$.response_time = $S_i$.self_response_time
    foreach DataType $D_j$
        $D_j$.response_time = infinity
    foreach DataType $D_k$ in the provided DataTypes
        $D_k$.response_time = 0
        available_data.add($D_k$)
        available_service = getAvailableService(available_data)
    foreach 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 = getAvailableService(available_data)
        foreach 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 {
            foreach required DataType $D_r$
                traceback($D_r$);
        }
```

A traceback function is used at the end of the main process to generate the BEPL format solution, which is described in the following.

Tracebace Search

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

2.2 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 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.

2.3 Algorithm complexity 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 $O(\log n)$ time to put the service at the right position. Thus all the PUSH operations take $(O(m)+O(\log n))n = O(mn+n\log n)$ 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 $O(cn+n\log n)$ for PUSH and $O(c^2n)$ for SCAN.

In summary, the time complexity is $O(n)+O(mn+n\log n)+O(mn^2) = O(mn^2)$. If an index is used, the time complexity becomes $O(n)+O(cn+n\log n)+O(c^2n) = O(n\log n)$. 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 that 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 $O(cn_0)+O(cn_0+n_0\log n_0)+O(c^2n_0)$, which is constant.).

3 Performance Evaluation

The algorithm was validated by showing that it achieves the correct composite service with the optimal QoS in much lower computation time than for other algorithms.

3.1 Test process

The algorithm performance was tested based on the requirements of the annual Web Service Challenge (WS-Challenge)[12] 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[13] 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[14] 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 use the generator to generate 18 test sets. Then the composition algorithm and other algorithm are used to evaluate 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 the QoS-driven algorithm (QDA)[11] which placed second[9,10] in the WS-Challenge2009 performance evaluation[9].

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

3.2 Evaluations

The 18 test sets generated by the generator each have a different scale of web services and ontology concepts. There are about 20 000 web services available on the Internet[15], with the most widely used "OpenCyc" ontology[16] 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 2 lists the settings for these six test sets and the time cost for the QDA and GA algorithms.

Table 2 Concepts and services increase with the same ratio

Test set	Test set properties		Time cost (ms)	
	Number of concepts	Number of web services	QDA	GA
1	37 500	500	125	78
2	75 000	1000	300	78
3	112 500	1500	600	93
4	150 000	2000	800	109
5	187 500	2500	950	109
6	225 000	3000	1040	78

The results in Table 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 3000 services. In test set 6, the GA is more than 10 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 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 3 lists the results for these six test sets, and the time costs.

Table 3 Concepts increase while the services remain constant

Test set	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	1100	140
10	200 000	20 000	1220	188
11	250 000	20 000	1440	219
12	300 000	20 000	1680	210

The results in Table 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 8 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 4 lists the results for these six test sets, and the time costs.

Table 4 Services increase while the concepts remain constant

Test set	Test set properties		Time cost (ms)	
	Number of concepts	Number of web services	QDA	GA
13	150 000	2000	580	78
14	150 000	4000	620	78
15	150 000	6000	750	94
16	150 000	8000	880	109
17	150 000	10 000	960	125
18	150 000	12 000	1040	125

The results in Table 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 8 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.

3.3 Discussion

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 was 2 to 10 times faster than the QDA with the GA performance very stable even with large number of services and concepts. The GA always selected the service with the best accumulated QoS, so the services in the solution did not need to be updated in the following search process which improved the efficiency. The QDA was an iterative search process with new services added to the solution layer by layer until all the output data was found, so many redundant services were included. The GA search space was also much smaller than that of the QDA.

4 Conclusions and Future Work

This paper 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 based 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] Alrifai M, Risse T. Combining global optimization with local selection for efficient QoS-aware service composition. In: Proceedings of WWW 2009. Spain: ACM, 2009: 881-890.

[2] Zeng L, Benatallah B. QoS-aware middleware for web service composition. *IEEE Transactions on Software Engineering*, 2004, **30**(5): 311-327.

[3] Yu T, Zhang Y, Lin K J. Efficient algorithms for web services selection with end-to-end QoS constraints. *ACM Transactions on the Web*, 2007, **1**(1).

[4] Zeng L, Benatallah B, Dumas M, et al. Quality driven web services composition. In: Proceedings of WWW 2003. Hungary: ACM, 2003: 411-421.

[5] OASIS. Web services business process execution language. http://docs. oasis-open. org/wsbpel/2. 0/wsbpel-v2. 0. pdf, 2007.

[6] Pisinger D. Algorithms for Knapsack problems [Dissertation]. Copenhagen: University of Copenhagen, 1995.

[7] Maros I. Computational Techniques of the Simplex Method. Springer, 2003.

[8] Lecue F, Mehandjiev N. Towards scalability of quality driven semantic web service composition. In: Proceedings of IEEE International Conference on Web Services. USA: IEEE, 2009.

[9] Web Service Challenge 2009. http://www.ws-challenge.org/wsc09, 2009.

[10] Yan Y, Xu B, Gu Z, et al. A QoS-driven approach for semantic service composition. In: Proceedings of 2009 IEEE Conference on on Commerce and Enterprise Computing. Austria: IEEE, 2009: 523-526.

[11] Xu B, Yan Y. An efficient QoS-driven service composition approach for large-scale service oriented systems. In: Proceedings of IEEE International Conference on Service-Oriented Computing and Applications (SOCA09). 2009: 25-32.

[12] Web Service Challenge. http://www.wschallenge.org/. 2010.

[13] Web Ontology Language. http://www.w3.org/TR/owl-features/. 2004.

[14] Web Service Level Agreements. http://www.research.ibm.com/wsla/. 2003.

[15] Al-Masri E, Mahmoud Q H. Investigating web services on the world wide web. In: Proceedings of WWW 2008. Beijing, 2008: 795-804.

[16] Cycontology. http://www.cyc.com/cyc/technology/whatiscyc_dir/maptest. 2010.