An Efficient QoS-driven Service Composition Approach for Large-scale Service Oriented Systems

Bin Xu, Yixin Yan
Department of Computer Science and Technology, Tsinghua University
Tsinghua National Laboratory for Information Science and Technology
Beijing, 100084, China
{xubin, yanyx}@keg.cs.tsinghua.edu.cn

Abstract—**QoS (Quality-of-Service) has become a critical issue to guarantee the performance of service oriented systems (SOS). However, efficient ways to build SOS with required QoS are still being developed. In most cases of SOS, developers are more concerned on the service functionalities than QoS. In this paper, we propose a QoS-driven service composition approach to efficiently build SOS with optimal QoS. We show that, under certain conditions, the problem of QoS optimization can be solved by dynamic programming. The experiment results show that our approach can be used to solve large-scale service composition problem effectively and efficiently with QoS guaranteed.**

Keywords - service composition; quality of service; large-scale services; dynamic programming;

I. INTRODUCTION

As SOS becomes more widely adopted, a large number of services are composed from distributed software systems to handle complex processes in many business fields. At the same time, QoS has become a critical issue to guarantee the performance of SOS. Users usually prefer to use a system with better QoS.

While developing SOS, developers may have multiple choices for one desired service. For example, to know the weather, they can choose a weather forecast service from either Yahoo or Google. For an individual service, its QoS is often determined by its network conditions or the capability of servers, and thus may be difficult to improve. On the other hand, since an SOS consists of many services each with multiple candidates, the QoS of SOS can be improved by selecting services.

So far, efficient service composition algorithms that build SOS with required QoS are still difficult to find. In this paper, we study 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. The number of services on Internet grows very fast in recent years. Meanwhile, service oriented e-business systems have become more popular and more complex. So an approach that can efficiently handle a large-scale service composition for SOS is urgently needed.
2. Trade-off 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 have to suffer a long network delay. In comparison, people who use local online community market may have a limited selection on the range of goods but do not need to worry about network congestions. So, the problem becomes more complex after considering the QoS in SOS.
3. Traditional service composition algorithms usually focus only on the functionalities, such as I/O parameters. Non-functional QoS, e.g. response time and throughput, is often ignored, which makes the performance of SOS quite poor.
4. Semantic information complexity grows exponentially for computer to have a precise understanding about the data on the Web. But there is still no effective integration between Web services and semantic information which makes service composition with semantics hard to do in SOS.

To meet the challenges in building SOS, we propose a QoS-driven service composition algorithm that can efficiently build SOS with guaranteed QoS. The advantages of our algorithm 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 gives some preliminary definitions and a formal description of the problem. Section 3 presents the composition algorithm. Section 4 gives the experiment strategy, data set and result analysis. Section 5 reviews and compares the related works. Section 6 concludes the paper and proposes future work.

II. PROBLEM DEFINITION

This section presents some basic definitions as well as the description of the composition problem. They include the definitions of service and SOS (functionalities and non-functionalities), the rule 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:

Definition 1:

Service S = { D_{in}, D_{out}, R, T} where

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

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

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

T : throughput - the number of requests a service can support.

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. QoS can represent any kind of non-functional property. For simplicity, we consider only response time and throughput in this paper. Any service that has these features can be used in building SOS, defined as:

Definition 2:

Service-Oriented System (SOS) = { D_{in}, D_{out}, P, R, T} where

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

D_{out} = { d_i | d_i is an output type of the SOS, defined by some 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 from SOS;

T : throughput - the number of requests a SOS can support.

From these two definitions, we can see that response time and throughput for service and SOS are similarly defined. Take response time as an example, the response time of a single service means the time from sending a request to the service to receiving the response. As mentioned in the first section, it 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 is difficult to improve. But the response time for a SOS is a different story.

There are three basic flow structures in SOS including *sequence*, *parallel* and *switch* (Figure 1). A sequence consists of services that are invoked in order. A parallel consists of services that are invoked together at the same time. And 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 the three flow structures, we adopt some widely used QoS composition rules. Suppose sequence A consists of A1, A2, …, An; parallel B consists of B1, B2, …, Bn; switch C consists of C1, C2, …, Cn. Definition 3 shows how R(x) (the response time of x) and T(x) (the throughput of x) can be calculated. These calculating rules are used in the Web Service Challenge competition [1] which will be further discussed in the experiment section.

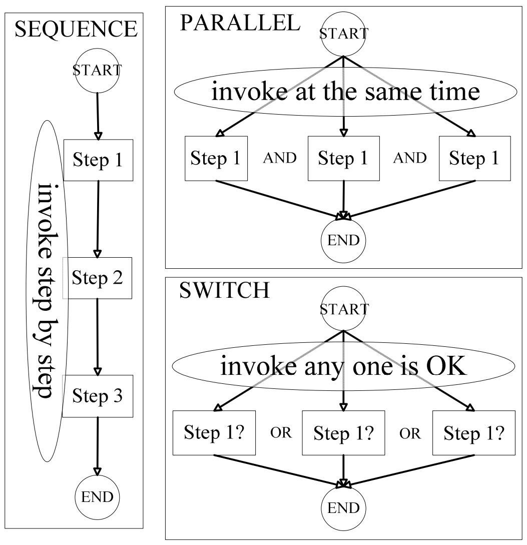

Figure 1. Three Basic Flow Structures in SOS

Definition 3:

$$R(A) = \sum_{i=0}^{n} R(A_i) \qquad Eq\,1$$

$$R(B) = \max\{R(B_1), R(B_2), ..., R(B_n)\} \qquad Eq2$$

$$R(C) = \min\{R(C_1), R(C_2), ..., R(C_n)\} \qquad Eq3$$

$$T(A) = \min\{T(A_1), T(A_2), ..., T(A_n)\} \qquad Eq4$$

$$T(B) = \min\{T(B_1), T(B_2), ..., T(B_n)\} \qquad Eq5$$

$$T(C) = \max\{T(C_1), T(C_2), ..., T(C_n)\} \qquad Eq6$$

Since SOS can be defined as a combination of the three structures, based on the above QoS definition and composition rules, the QoS of SOS can be determined. For example, in Figure 2, services C and D are invoked sequentially to form a sequence (Sequence 4). So 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 solution 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 to each other, so they can be invoked in parallel. Obviously, the response time of a parallel should the biggest response times (150) of all the branches (100 vs. 150) which can be regarded as the bottleneck of the parallel. After the calculation, the response time of SOS in this example is 1150ms and the throughput is 7000 invocation/min.

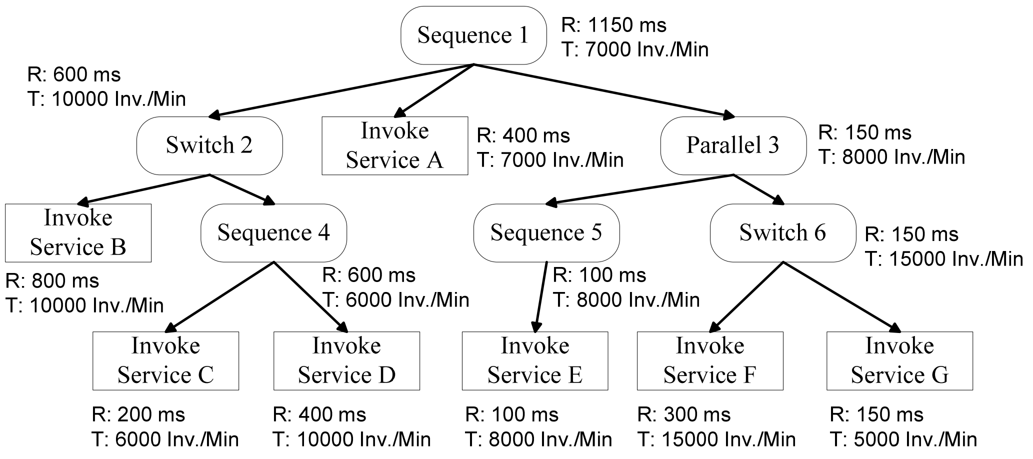

Figure 2. QoS Calculation for three structures in SOS

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 through selecting proper services. This is the goal of our study to find the optimized QoS.

Given a set of available services, a user request includes I/O parameters (D_{in} , D_{out} ,in Definition 2). The input data types of the SOS are provided by the user; the output data types and the QoS requirements of the SOS is what the users really want. So the problem is, given a set of services and a user request, how to find a service execution plan (P in Definition 2) that takes the input data types and outputs the requested data types with guaranteed QoS.

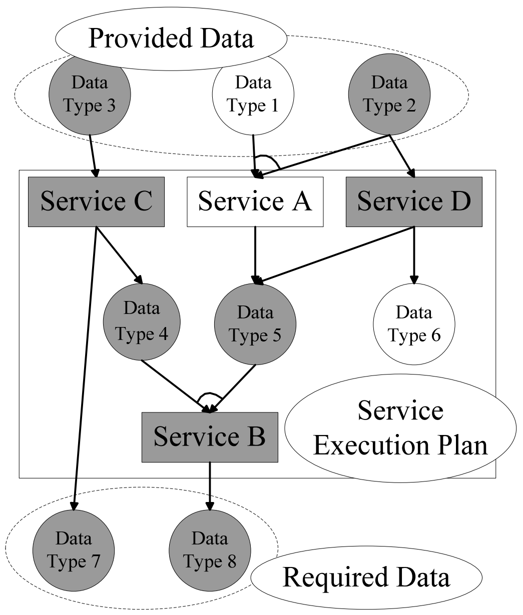

Figure 3. Example of Service Composition

Figure 3 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 the rectangle 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. And not all services are adopted in the plan (only grey ones). The execution plan is a service composition which we want to find out.

III. QoS-Driven And Large-Scale Service Composition Algorighm

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 the two properties, we propose a dynamic programming algorithm to solve the problem of service composition for SOS. The key ideas of the algorithm 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 the optimal composition.
2. We define a variable for every data type that maintains the best known QoS value for the 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 shortest 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 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. So, if every sub-problem is guaranteed to be local 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 use breadth-first search to find the solution. The QoS of services and data types are calculated and updated in this phase. When all required output data types are satisfied, we will produce the composition (P in Definition 2) using a depth-first trace back.

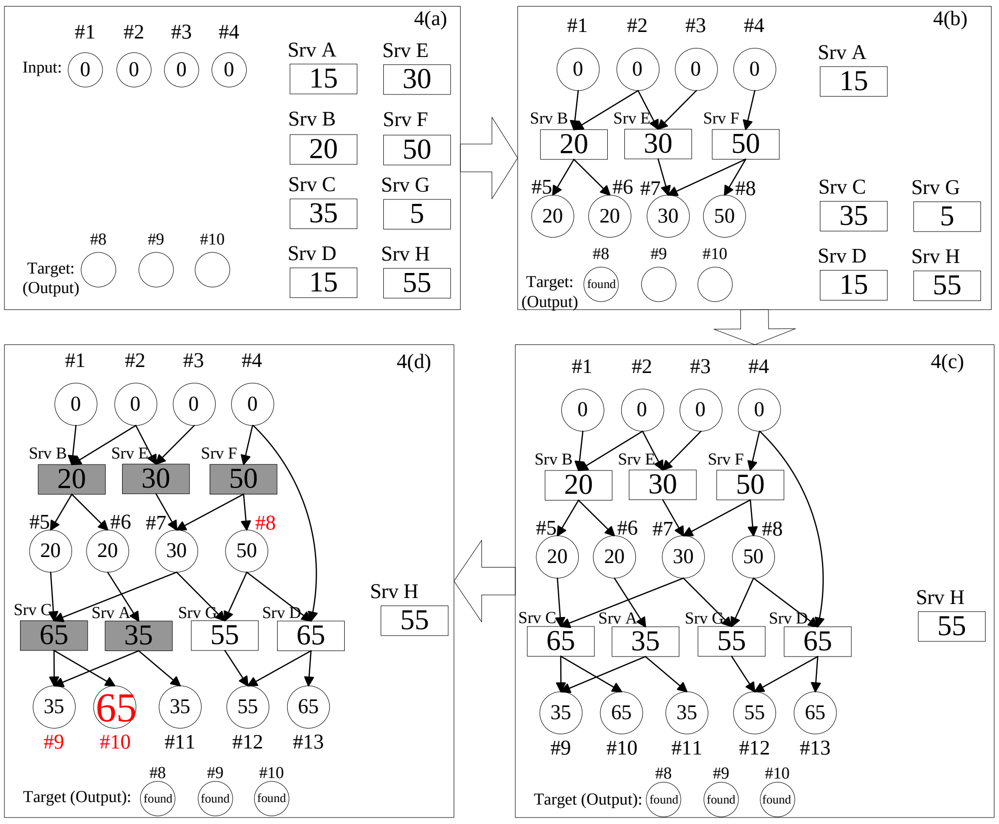

Figure 4. Example of How to Get the Composition Result Step by Step

Figure 4 shows an example of the search process to find the SOS with an optimal (minimum) 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}.

The algorithm first extends services B, E and F since they can be directly invoked using the input data types. It then updates the QoS of these services and the extended data types as shown in Figure 4(b). Based on the produced data types (data type #5, 6, 7 and 8) and all previous available data types, more services (C, A, G and D) can be extended. The algorithm updates these services' QoS using Definition 3. For example, the response time of C itself is 35 (in Figure 4(a)). When it is extended in the composition, the optimal response time from the beginning of the composition to where the service is will be based on the response time of all its input and itself. Since the processes that make data type #5 and #7 available can be executed in parallel, until all its input are available service C can be invoked, so the response time of itself is added at last, which equals to the maximum response time of data type #5 and #7 plus 35, resulting in 65.

The algorithm continues to extend the services and update the QoS until all required data types are covered and no more services can be added into the composition. In Figure 4(d), 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 time we can get all of them from the SOS.

When there is more than one services with output matches a data type, the algorithm selects the service with the best QoS performance. So, every extended service is based on the optimal path which makes the whole SOS have the best QoS performance.

The algorithm is shown on the next page.

```
main ()
{
    Define a set “available_data”, record the data type if the service, which can output it,
      is satisfied, initialize the set as blank.
    Initialize the response time: zero for provided data types, infinity for others.
    Initialize the throughput: infinity for provided data types, zero for others.
    Define an ontology tree to record the ontology concepts and their relationship.

    while solution is not found or still have available services
    {
      List<Service> current_layer;
      foreach unused service s
      {
        if s can be invoked/satisfied
        {
          current_layer.add(s);
          updateServiceQoS(s);
          available_data.add(s.output);
        }
      }
      solution.addlayer(currentlayer);
    }
    //generate the solution with the shortest response time
    //similar to generate the solution with best throughput
    print("<parallel>");
    foreach required datatype d
         traceback(d);
    print("</parallel>");
)

void updateServiceQoS(Service s)
{  //update the QoS to ensure the sub-structure is local optimal
    s.response_time = maxResponseTime(s.input_list) + s.self_response_time;
    s.throughput = min(minThroughput(s.input_list), s.self_throughput);
    foreach output datatype d of s
    {
      if d.response_time > s.response_time
      {
        d.response_time = s.response_time;
        d.ptr_response_time_generator = s;
      }
      if d.throughput < s.throughput
      {
        d.throughput = s.throughput;
        d.ptr_throughput_generator = s;
      }
    }
}

void traceback(DataType d)
{  //do a trace back search to output the bpel formatted solution
    if d belongs to the provided data set
      return;
    else {
      print("<sequence>\n<parallel>");
      foreach d.ptr_response_time_generator.input_list di  //input_list means the input data types of some service.
      {
        traceback(di);
      }
      print("</parallel>");
      print("invoke " + d.ptr_response_time_generator.name);
      print(</sequence>);
    }
}
```

Do a breadth-firth search until all required data types are covered and no more services can be used, and save the solution. When checking whether a service is satisfied, we use ontology mapping to check if all its input are satisfied.

Core part of the algorithm: update and save the QoS to ensure the sub-structure is local optimal. The QoS calculation follows the rules in Definition 3 for parallel.

Use a depth-first trace back search to generate the solution of BPEL formats.

Service is defined in terms of Definition 1

```
//a data type represents an input or an output of services. It is
//assigned a variable to record its best known QoS, which has been
//explained in the above paragraph.
struct DataType
{
   //indicate the concept which the data type belongs to.
   //all concepts are defined in the provided OWL file.
   Concept concept_of_datatype;

   float response_time; //indicate its current best response time

   //point to the service which generate it with best response time.
   Service ptr_response_time_generator;

   float throughput; //indicate its current best throughput.

   //point to the service which generate it with best throughput.
   Service ptr_throughput_generator;
}
```

IV. EXPERIMENTS

A. Experiment Process

The experiments are carried out based on four input files: 1) `Services.wsdl` which records all available Web services; 2) `Taxonomy.owl` which records all concepts in an ontology format [2]; 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 [3]; 4) `Query.wsdl` which records one user query including the provided data types and required data types.

The experiment follows the requirements of the Web Service Challenge (WS-Challenge) which is an annual service composition competition held in the IEEE e-Commerce conference (CEC) since 2006 [1]. It focuses on the semantic composition of Web services and uses OWL ontology to define services and their relationships. QoS is also introduced into the competition in 2009. 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 real. They are virtual since there are no actual service implementations that can be invoked on the Internet. But they are real 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 promised to have the optimal QoS (the shortest response time and the largest throughput). We can evaluate our experiment result by comparing with the benchmark.

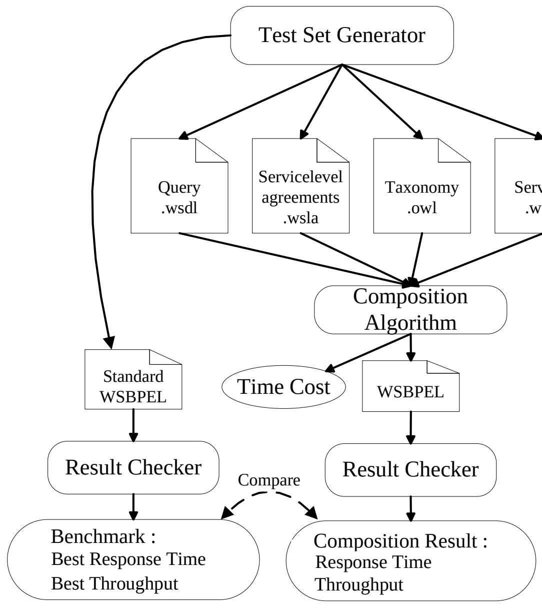

Figure 5. Experiment Process

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. Our algorithm can guarantee that the composition result has the optimal QoS, and it has been proved by comparing the benchmark with our results.

Figure 5 shows the experiment process. First, we use test set generator to generate four input files for each test set. Then our composition algorithm takes these four files as the input and output 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.

B. Experiment Settings and Results

For the experiment setting, we concern about the scale of Web services and ontology concepts. After investigating on the Internet, we found that the number of available Web services is about 2000, and the Cyc ontology [4] having about 150,000 concepts. The Cyc ontology is almost the largest ontology, and having a set of concepts which tries to describe universal subjects. Along with the development of semantic Web, more and more data on Web will be formalized and mapped to concepts of ontology. So we integrate concepts and I/O data types together. In the following experiment, we set the number of Web services and concepts at the scale of investigation, or even larger.

The test set of experiments 1 is conducted by changing the number of concepts and Web services while keeping the ratio between them. Table I shows the test sets of experiment 1. In test set 4, the number of concepts and Web services are according to our investigation on the Internet.

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

TABLE I. TEST SETS IN EXPERIMENT 1

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

For the composition result of each test set, we concern about the composition time cost and QoS values. Figure 6 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 second. Compared to Zeng's work [5] which is shown as the red line in the figure, our algorithm has a much better performance on the 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), which means under the same number of services, our experiment handles a more complex problem.

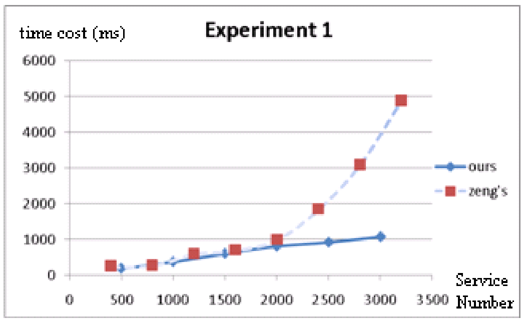

Figure 6. Efficiency Analysis of Experiment 1

The second study is, if the number of Web services remains the same while the number of ontology concepts increases, how the algorithm performs. This experiment is closer to the real world situation, because the semantic Web is expanding rapidly and the number of Web services becomes stable in recent years. Though there are about 2000 available Web services on the Internet, we enlarge the number of Web services and keep it in 20000. Table II shows the test sets for the study.

TABLE II. TEST SETS IN EXPERIMENT 2

Test Sets	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 and all test sets perform efficiently (less than 1.7 seconds).

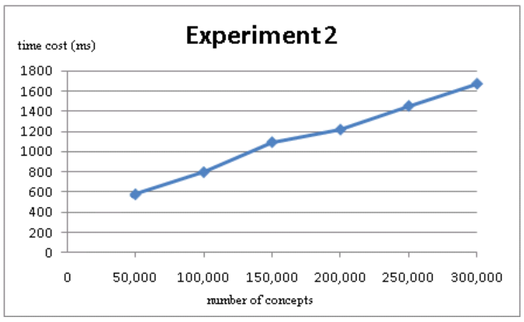

Figure 7. Efficiency Analysis of Experiment 2

The third experiment is that the number of the concepts remains the same while the number of Web services increases. This assumption is for the future development of Web. When most concepts are well described by ontology, the number of Web services may increase because of new businesses. Table III shows the test sets for this experiment.

TABLE III. TEST SETS IN EXPERIMENT 3

Test Sets	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

Figure 8 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 our algorithm performs stable and efficient (less than 1 seconds) under this assumption.

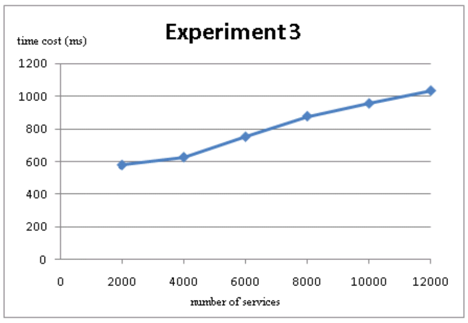

Figure 8. Efficiency Analysis of Experiment 3

Table IV shows the QoS values of composition result in the above three experiments. As we have explained above, there is a standard result of each data set which has the optimal QoS. And our algorithm can always guarantee that the composition result have the optimal QoS. The values in the table are the same with the values of the standard results. RT means the response time measured in ms; TP means throughput measured in the invocations per-seconds.

TABLE IV. QOS VALUES OF COMPOSITION RESULTS

Data Set	Experiment 1		Experiment 2		Experiment 3	
	RT	*TP*	*RT*	*TP*	*RT*	*TP*
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

C. *Discussion*

The experiments are based on the scale of a real Web. The numbers of services and concepts in the test sets have the same or larger scale than that on the Internet. Besides, we propose two trends for future Web and related series of tests. So the experiments well reflect and simulate the actual state of the Web. And the experiment results show that our algorithm has a very high efficiency. In most cases, a service composition can be done in less than 1 second.

V. Related Work

In the domain of automatic service composition, many works have been done based on the input and output parameters of services. Liang [6] proposed a semi-automated method for service composition. The main idea is to construct an AND/OR graph form a service dependency graph (SDG), applying a bottom-up search algorithm REV* to find a sub-graph for the solution. It is an effective method to find executions on parallel, but it doesn't consider the scale of the services and ignore the quality of services.

Concerned with QoS of software systems, Zeng [5, 7] proposed a global planning approach for service composition to optimize multiple criteria of QoS. Through integer programming, it can handle multiple execution paths. But the disadvantage is that it can only handle small scale service composition problem. All its experiments are based on only dozens of candidate services. Through comparison, our approach performs 3~5 times faster than Zeng's method in the large-scale service composition. Previous work like Cardoso [8] also considers QoS in service composition but this work doesn't focus on dynamic service composition.

Tao [9] studied the problem of service composition with multiple end-to-end QoS constrains. They proposed a broker-based architecture and several efficient heuristic algorithms to maximize the QoS. Xiao [10] also studies the QoS in end-to-end environment and presents a MCOP method in domain composition and adaptation problem. Tao improves Xiao's work to handle multiple workflows such as parallel, conditional and loops. However, their work doesn't address the performance of large-scale service composition.

Alrifai and Risse [11] 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 the local constraints. Second, they use distributed local selection to find the best web services that satisfy these local constraints. The disadvantages of their work include: 1. can not find the optimal QoS in all the experiments; 2. the composition time in random data set is poor.

Jaeger and Ladner [14] studies how already identified candidates, which a selection process originally has separated out, 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 [12, 13] also study the QoS in Web service composition.

VI. Conclusion and Future Work

Under the hypothesis that the QoS of SOS can be quantitatively calculated according to the included services and their relationships, QoS based service composition problem can be efficiently solved by our algorithm using a dynamic programming solution. If new services are deployed, we can also conveniently use them to refine the composition result without redo the whole composition. The biggest advantage is that we can handle a large-scale service composition within a very short time.

There are still some constraints of the approach which we want to extend on in the future. It 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. But as semantic Web develops and QoS becomes more important in e-business, these issues may be addressed in the near future.

Acknowledgment

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

References

[1] Web Service Challenge. http://www.ws-challenge.org

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

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

[4] Cyc ontology: http://www.cyc.com/cyc/technology/whatiscyc_dir/ma ptest

[5] L Zeng, B Benatallah. QoS-Aware Middleware for Web Service Co mposition. IEEE Transactions on Software Engineering, Vol. 30, No. 5, May 2004.

[6] Q. A. Liang and S. Y.W. Su. AND/OR graph and search algorithm fo r discovering composite web services. International Journal of Web S ervices Research, 2(4):48 – 67, 2005.

[7] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality Driven Web Services Composition. In Proceedings of the 12t h international conference on World Wide Web (WWW), Budapest, Hungary. ACM Press, May 2003.

[8] J. Cardoso. Quality of service and semantic composition of workflow s. Ph.D Thesis, University of Georgia, 2002.

[9] TAO YU, YUE ZHANG, and KWEI-JAY LIN. Efficient Algorithms for Web Services Selection with End-to-End QoS Constraints. ACM Transactions on the Web, Vol. 1, No. 1, Article 6, May 2007.

[10] XIAO,J. AND BOUTABA, R. 2005. QoS-aware service composition and adaptation in autonomic communication. IEEE J. Select. Areas Comm. 23, 12, 2344–2360.

[11] Alrifai, M., Risse, T. Combining Global Optimization with Local Sel ection for Efficient QoS-aware Service Composition. WWW 2009, A pril 20–24, 2009.

[12] Lecue, F., Mehandjiev, N. Towards Scalability of Quality Driven Se mantic Web Service Composition. IEEE International Conference on Web Services, July 2009.

[13] Stein, S., Payne, T.R., Jennings, N.R., "Flexible Provisioning of Web Service Workflows" ACM Transactions on Internet Technology, Feb. 2009.

[14] JAEGER,M.C. AND LADNER, H. 2005. Improving the QoS of WS compositions based on redundant services. In Proceedings of the Inte rnationalConference onNextGenerationWeb Services Practices (NWe SP 2005). 189–194.