QSynth: A Tool for QoS-Aware Automatic Service Composition

Wei Jiang, Charles Zhang, Zhenqiu Huang, Mingwen Chen, Songlin Hu, Zhiyong Liu

Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
Graduate University of the Chinese Academy of Sciences, Beijing, China
Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China

Abstract

With the proliferation of Web services, service engineers demand good automatic service composition algorithms that not only synthesize the correct work plans from thousands of services but also satisfy the quality requirements of the users. Our observation is that conventional approaches suffer from serious limitations in scalability and accuracy when addressing both requirements simultaneously. We have designed and implemented a tool QSynth to use QoS objectives of service requests as the search directives. This approach effectively prunes the search space and significantly improves the accuracy of the search results. Evaluations show that, compared to the state of the art, QSynth achieves superior scalability and accuracy with respect to a large variety of composition scenarios. Our design of QSynth won the performance championship of Web Services Challenge 2009.

1 Introduction

Today’s Web services with rich functionalities are built by composing many existing and more specialized Web services from third party providers. Such service compositions first need to be correct, i.e., satisfying user requests despite the compositional scale and the complexity of their internal work plans. They also need to have good quality of service (QoS) in terms of speed, cost, reliability, and many other measures. As the number of third-party providers grows rapidly, how to automate the process of service composition has drawn the significant research attention [19, 18, 5, 11, 15, 20, 7].

Most conventional approaches consider the search of correct work plans (automatic composition problem) and of the ones that give the optimal QoS (service selection problem) as two separate sub-problems of the general QoS-aware automatic service composition problem. Our observation is that this view often leads to sub-optimal solutions in terms of scalability and accuracy. QoS objectives can be used to drive the process of work plan synthesis, leading to very effective pruning of the search space and the better planning of the search steps. As a result, both the scalability and the precision of QoS-aware automatic service composition can be dramatically improved. Let us further elaborate these observations through a simple example.

Our hypothetical service composition scenario involves eight Web services listed in Table 1, where we only consider the response time as the QoS parameter of these services, for the simplicity of illustration. Our sample request \( R \) is to compute the driving direction to a restaurant in a tourist city, where the restaurant satisfies her dining preferences, such as Chinese food, and the destination satisfies her travel preferences, such as not more than 80 miles away. The task of the service engineer is to come up with a work plan among services afore-listed, which satisfies our example request in the shortest time. For example, \( \{ W_2, W_3, W_7 \} \) can be a candidate work plan for request \( R \).

Research in [19, 18] solves the QoS-aware service selection problem. They assume the work plan is pre-defined and each task in the plan is not a concrete service but representing a service class with multiple candidates of different QoS measures. This approach requires enumerating all viable solutions, i.e., enumerating \( \{ W_1 \}, \{ W_2, W_3, W_7 \}, \{ W_2, W_4, W_8, W_7 \} \) in our example (Figure 1), for they don’t belong to the same pre-defined plan. Enumerating such solutions is an exponential process, not feasible when the number of services is large.

On the other hand, many automatic composition approaches do not take the QoS attributes into account. Some approaches [20], search all the results, while others [6] give...
priority to the results with fewer services or simpler work plans. In our example, the solution $\{W_1\}$ could be chosen because it has the fewest number of services and the smallest depths. But in terms of the overall QoS, it is the worst of the three solutions by having the maximum response time.

Research in [13, 10, 16] tries to combine QoS computations into the process of work plan generation. The work in [13] uses the AI Planner to address the problem of QoS-aware automatic service composition. But it assumes that the overall QoS can be expressed as the sum of the individual QoS values. Many QoS measures, such as throughput, do not belong to this type. Research in [10] finds all the service composition results and then chooses the optimal one, which is too costly. The work in [16] tries to find the top $k$ optimal results, but the correctness is not guaranteed due to the heuristic rule it adopts.

<table>
<thead>
<tr>
<th>Web Service</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Function Description</th>
<th>QoS Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>W_1</td>
<td>$I_A, I_B, I_C$</td>
<td>$O_D$</td>
<td>Get the driving direction of the restaurant.</td>
<td>900</td>
</tr>
<tr>
<td>W_2</td>
<td>$I_A, I_B$</td>
<td>$O_E, O_F$</td>
<td>Get the city and its zipcode.</td>
<td>100</td>
</tr>
<tr>
<td>W_3</td>
<td>$I_C, I_E$</td>
<td>$O_H$</td>
<td>Get the addresses of restaurants.</td>
<td>200</td>
</tr>
<tr>
<td>W_4</td>
<td>$I_C, I_F$</td>
<td>$O_G$</td>
<td>Get the zipcodes of restaurants.</td>
<td>500</td>
</tr>
<tr>
<td>W_5</td>
<td>$I_K$</td>
<td>$O_H$</td>
<td>Get the addresses of restaurants.</td>
<td>600</td>
</tr>
<tr>
<td>W_6</td>
<td>$I_L, I_J$</td>
<td>$O_D$</td>
<td>Get the driving direction of the restaurant.</td>
<td>500</td>
</tr>
<tr>
<td>W_7</td>
<td>$I_H$</td>
<td>$O_D$</td>
<td>Get the driving direction of the restaurant.</td>
<td>200</td>
</tr>
<tr>
<td>W_8</td>
<td>$I_G$</td>
<td>$O_H$</td>
<td>Get the address of the restaurant.</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 1. A motivating example

To overcome the afore-mentioned limitations, we have designed QSynth, a tool to perform QoS-aware service composition automatically in milliseconds for thousands of services. It is 100% precise and capable of handling different QoS types, and different composition patterns. QSynth explores the solution space by exploring all possible subgraphs in the service dependency graph that are qualified, i.e., satisfying the user request. However, instead of simply enumerating sub-graphs, we perform QoS calculations concurrently with the graph exploration. The QoS computation history is recorded and used as a guidance for effectively pruning unnecessary path explorations. Our algorithm outperforms related algorithms in precision and time as confirmed by experiments and time complexity. The techniques reported in this work won the performance championship of the Web Service Challenge 2009 [1], which shows the effectiveness of our approach compared to the state of the art. In essence, our algorithm is based on graph search. But it is different from the classical graph search algorithm in some respects, such as the goal and some properties of graph.

The main contributions of this paper are as follows:

1. Two effective mechanisms are adopted in our algorithm. One is inverted index table which is used to represent the graph. The other is counting mechanism which is used to judge whether a service is enabled.

2. We tackle the problem of automatic service composition and QoS-aware service selection problem together with an algorithm that both efficient and 100% precise.

3. We implement a tool QSynth to facilitate service composition based on the above algorithm and common standards.

The remainder of the paper is organized as follows. Related works on QoS-aware automatic service composition are introduced in Section 2. Section 3 presents the definition of QoS-aware automatic service composition problem and QoS model. Section 4 illustrates the architecture of QSynth and the concrete algorithm of QSynth. Then, Section 5 shows some experiments that have been carried out to evaluate our approach. Finally, Section 6 presents the conclusions.

2 Related Work

In SOC (Service Oriented Computing) paradigm, since it is always impossible to find composition results manually from huge amount of services, automatic service composition is proposed to enable automatic search of work plans for a given request. Some approaches, such as [11, 15], treat the problem as an AI Planning problem. The goal through the states generated by a set of actions. While some other research, such as [7, 12], considers it as a graph search problem, solving it with technologies like shortest path, A*, etc. In addition to centralized algorithms, a distributed approach is put forward in [8], which in turn improves system performance by utilizing distributed computing resources.

Meanwhile, to guarantee local or global QoS requirements of service composition, the QoS-aware service composition has attracted a lot of attention from different fields as well. Compared to automatic service composition, it assumes the existence of a predefined work plan with a set of "abstract" tasks, while the objective is to select service for each task from its candidate services to meet local or global QoS constraints. The Zeng04 [19] uses two service selection approaches based on Multiple Criteria Decision Making: local optimization and global planning based on Integer Programming. In Yu07 [18], the authors model the problem as a multiobjective 0-1 knapsack problem or a multi-constraint optimal path problem, and then compute optimal result according to the objective function. Alternatively, the [5] proposes genetic algorithms to address the problem. Al-

...
though it is slower than Integer Programming, it can deal with non-linear constraints.

In general, all these approaches for automatic service composition or QoS-aware service composition only consider one single aspect of QoS-aware automatic service composition, which not only has to search work plans for a given request, but also needs to guarantee global QoS requirements at the same time.

Some pioneers have proposed approaches to address the QoS-aware automatic service composition problem, such as Liu’05 [10], Wang’06 [16] and Naseri’07 [13]. Recently, from the WSC 09 [1], we know more about the work of others who are also interested in QoS-aware automatic service composition. Peter’09 [4] proposes a fast approach by some index tables stored in database to fit the competition. Yan’09 [17] augments the approach in [6] with QoS. All of them lack the capability of providing 100% precision or their efficiency are not good enough when the scale of services is big.

3 QoS-Aware Service Composition

3.1 General Terms

We first define a few terms we use.

QoS: The QoS is the non-functional properties of Web service, such as response time and throughput. A comprehensive category of QoS is defined in [14].

Request: A request \( R \) has input parameters \( I_R \) and desired output parameters \( O_R \). The \( I_R \) specifies the information the user can provide in terms of type definition and the \( O_R \) declares what the user needs.

Web Service: A Web service \( W_i \) \((1 \leq i \leq N)\) is considered as a tuple. \( W_i = \{I_{W_i}, O_{W_i}, Q_i\} \). \( I_{W_i} \) represents the input parameters of \( W_i \), and \( O_{W_i} \) represents the output parameters of \( W_i \). Let \( Q_{i,j} \) \((1 \leq j \leq m)\) be the \( j \)th dimension QoS criterion value of \( W_i \), so \( Q_i = \{Q_{i,j}|_{j=1}^m\} \). Specially, the request \( R \) can also be considered as a virtual service whose QoS values are the overall QoS values of the composed service.

ParameterMatch: Strictly speaking, two Web service parameters (the inputs or outputs of services), \( P_a \) and \( P_b \) are declared as matched not only by their type definitions but also by the ontological relationships. Let \( \text{Concept}(P) \) be the concept which the parameter \( P \) belongs to. Here, \( P_a \) and \( P_b \) can be matched if \( \text{Concept}(P_a) \) is subClassOf or the same as \( \text{Concept}(P_b) \). The subClassOf is transitive.

Web Service Match: Let \( W_a \) and \( W_b \) be two Web services, \( W_a \) can match \( W_b \) if there exist some outputs of \( W_a \) can match some inputs of \( W_b \). It is labeled as: \( O_{W_a} \bigcap I_{W_b} \neq \emptyset \).

We distinguish Web service match between full match and partial match. \( W_a \) fully matches \( W_b \), if \( O_{W_a} \geq I_{W_b} \).

For example, \( W_3 \) fully matches \( W_7 \) in Figure 1, all the inputs of \( W_7 \) can be provided by \( W_3 \). \( W_a \) partially matches \( W_b \) if \( (O_{W_a} \bigcap I_{W_b} \neq \emptyset) \land (O_{W_a} \nsubseteq I_{W_b}) \). For example, in Figure 1, \( W_2 \) partially matches \( W_3 \), for one input of \( W_3 \), \( C \), can’t be provided by \( W_2 \). We use partial match in this way because it gives better composition flexibility.

3.2 The Problem

Definition 2.1. QoS-aware automatic service composition.

Given a set of services and a request \( R \), find a work plan \( T \) that defines an invoking order over a set of Web services \((W_1, W_2, \ldots, W_N)\) by satisfying the following conditions:

I. \( \{I_R \cup O_{W_1} \cup \ldots \cup O_{W_i}\} \supseteq I_{W_{i+1}} \) (1 \( \leq i \leq N - 1 \));
II. \( \{O_{W_1} \cup O_{W_2} \cup \ldots \cup O_{W_N}\} \supseteq O_R \);
III. The overall QoS of \( T \) is optimal;

This process is called QoS-Aware Automatic Service Composition. When the above formulas hold, we say \( R \) is satisfied, and we say the \( W_i \) in \( T \) is enabled. Our assumptions are: (1) the QoS measures considered here are quantitative. (2) the QoS values are static, e.g., the mean value of each QoS measure. Dynamic QoS is beyond the scope of this paper.

Our approach models QoS-aware automatic service composition as a graph search problem. Conceptually, synthesizing a work plan starts by building a dependency graph, as illustrated in Figure 1, where we connect two services, \( W_A \) and \( W_B \), from \( W_A \) to \( W_B \) if one of \( W_A \)'s output parameters matches one of \( W_B \)'s input parameters. A candidate solution is essentially a connected sub-graph in this dependency graph satisfying the basic condition that the union of input parameters of the direct successors of the start node \( (I_R) \) is a subset of \( R \)'s input parameters and the union of the output parameters is a superset of \( R \)'s output parameters \( (O_R) \). The final solution is simply to find such sub-graphs that give the global optimal QoS as defined by the service level agreement (SLA).

3.3 QoS Types and Computing Rules

We define four types of QoS measures. They are sum type (such as response time), min type (such as throughput), multiplication type (such as reputation) and max type. In addition, all the QoS measures can be categorized into two classes [18], [19]. One is negative, the higher the value, the lower the quality, such as response time and price. The other is positive, the higher the value, the higher the quality.
such as throughput and reputation. Since the composition result is represented as a DAG, we use three composition patterns: Sequence, Joint and Split to sufficiently represent the atomic structure of the service composition result. We can transform DAG to other formats, such as the BPEL format in the experiment. More patterns for QoS model can be found in [9].

We present the QoS computing rules with different types of QoS measures and different composition patterns. Table 2 shows the rules and the notations for them are shown below.

\[ L(W_N, j) : \text{Suppose that there is a DAG which ends with } W_N, \text{ and there are } N - 1 \text{ Web service } W_i, (1 \leq i \leq N - 1) \text{ before } W_N, \text{ } L(W_N, j) \text{ is the overall QoS value of the } j\text{th dimension QoS measures from } W_1 \text{ to } W_N. \]

\[ F : \text{One function in } \sum, \min, \prod, \max \text{ according different QoS types. The parameters of this function are } Q_{i,j}. \]

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Computing rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_{Sequence}(W_{N,j}) )</td>
<td>( F_1(Q_{i,j})_{i=1}^N ) (1 \leq i \leq N)</td>
</tr>
<tr>
<td>( L_{Joint}(W_{N,j}) )</td>
<td>( F_1(F_2(Q_{i,j})<em>{i=1}^{N-1}), Q</em>{N,j}) )</td>
</tr>
<tr>
<td>( L_{Split}(W_{k,j}) )</td>
<td>( F_1(L(W_j), Q_{k,j}) ) (1 \leq k \leq N)</td>
</tr>
</tbody>
</table>

**Table 2. QoS computing rules.** \( F_1, F_2 \in F \)

Take \( L_{Sequence}(W_{N,j}) \) for example. If we want to compute the overall value of the \( j\text{th dimension QoS criterion (assuming that it is response time) of a sequence which contains } W_i \) (1 \( \leq i \leq N \). The computing rule is:

\[ L_{Sequence}(W_{N,j}) = \sum (Q_{i,j})_{i=1}^N. \] Our algorithm focuses on single QoS measurement. If there are multiple QoS measurements, one possible but not perfect approach is using weighted sum approaches to transform all the QoS values into an aggregate QoS score.

4 QoS-Aware Automatic Composition with QSynth

This section discusses the core algorithm and the architecture of QSynth. We also provide time complexity and guarantees of our algorithm.

4.1 The Overview of QSynth

The architecture of QSynth is illustrated in Figure 2. It consists of five components.

**QoS Data Handler.** It collects QoS information for Web services from three sources. They are "Real Dynamic QoS Monitor" (monitoring the real Web services and getting their QoS information real time); "Real Static QoS Data" (getting the QoS values of Web services by history statistic) and "Simulation QoS Data" (generating QoS data by program). Note that "Real Dynamic QoS Monitor" is not yet implemented in this version of QSynth.

**Initialization Module.** It extracts Web services information from WSDL, OWL-S files, and QoS Data Handler, then stores these services in the Service Database.

**The Knowledge Base.** The Knowledge Base contains service database and two inverted indices tables. The service database contains all the Web services that can be used to generate service compositions. Based on the service database, two inverted indices tables: output inverted index and input inverted index are created. Each entry in the index tables is a tuple, (parameter, service list). We can look up all the services by their inputs or outputs.

**The Compositor.** The Compositor finds the optimal service composition result that can fulfill the request. It uses a worklist algorithm to find the enabled services and filters out the disabled services. It also records the optimal QoS values and the corresponding services through a two-level optimization approach.

**Result Generator.** Based on the optimal QoS providers for the parameters, the Result Generator uses a backward search to find the related services in the optimal composition result and the invoking sequences among them.

4.2 Request Processing

The process of handling a request consists of two steps. The first step is conducting forward breadth first search by worklist algorithm. The second step is executing backward search to generate the optimal composition result from information recorded in step one.

4.2.1 The algorithm for searching optimal QoS

The general approach of our algorithm is executing breadth first search from the services that can be triggered by \( I_R \).
At the same time, we decrease the count parameter together with the current computed optimal QoS. We then take the following two alternative steps.

1. **Known (Line 4-8 in Algorithm 1, Line 1, 2 in Procedure 2).** If optimal QoS is not yet known, we record the output parameter in the second queue that contains unprocessed enabled services. The second queue contains the services that need to be reprocessed. For each item in the second queue, we process it in the same way as the Step2 presented above (Line 12-17 in Algorithm 1). Usually, if the optimal QoS values of service update y times, this service will be processed in the second queue for each output parameter, all previously processed services which need this output parameter should be reprocessed by inserting them into another queue that contains the reprocessed services no duplicately (Line 2-7 in Procedure 2).

**Algorithm 1 Worklist Algorithm To Fulfill The Request**

- **Require:** Service Repository
  - for all Service $W_i \in Service Database$ do
  - $W_i.count \leftarrow |I_{W_i}|$
  - end for
  - enabledservices.add($I_R.trigger()$)
  - while enabledservices $\neq \emptyset$ do
  - $W \leftarrow enabledservices.remove()$
  - ForwardEnable($W$)
  - end while
  - if $R$ is not enabled then
  - return no result.
  - end if
  - while reProServices $\neq \emptyset$ do
  - Service $W_{update} \leftarrow reProServices.remove()$
  - if $W_{update}.calculateQoS()$ is better than $W_{update}.allQoS$ then
  - updateQoS($W_{update}.successors$)
  - end if
  - end while

After the queue that includes the unprocessed enabled services is empty, we process the second queue that includes the reprocessed services. For each item in the second queue, we process it in the same way as the Step2 presented above (Line 12-17 in Algorithm 1). Usually, if the optimal QoS values of $x$ inputs of a service update $y$ times, this service will be inserted into the second queue and reprocessed $x \times y$ times. Since this is very costly, we reduce multiple appearances of this service in the second queue to one all the time. This optimization will significantly reduce the time of handling the queue a lot as confirmed by our experiments in Section 5.2. Both queues will eventually become empty.

**Optimization:** We apply three kinds of filtering optimization in our algorithm: 1) Only the enabled services are considered, so the disabled services are excluded; 2) Although many enabled services can provide specific parameters, we only care about the service which can offer opti-

---

**Figure 2. QSynth Architecture**

Performing and tracking QoS computations at the same time. This leads to very effective pruning of search spaces and accurate rendering of final results, for only the enabled services are traversed. Note that this algorithm distinguishes itself from other classical graph search algorithms, such as shortest paths algorithms. Two obvious differences are: (1) our goal is finding DAGs, not only the paths in shortest paths problem; (2) we can traverse the nodes in dependency graph only when their preconditions are enabled.

**Data Structure:** Every service ($W_i$) is represented by a data structure of five members: selfQoS records the QoS value (or the weighted sum if its QoS is multidimensional) of $W_i$, allQoS records the overall optimal QoS value computed so far after $W_i$ is considered, $I_{W_i}$ and $O_{W_i}$ are stored in lists inputs and outputs, and the count is initialized by the number of parameters in $I_{W_i}$. We say the service $W_i$ is enabled if its count goes to zero. We use a hash table, Offered Parameters (OPars), as a registry to store the best QoS (optQoS) computed for a particular parameter and its corresponding provider (optProvider). The OPars updates constantly during the search time. The worklist algorithm has two queues, one is the enabledservices which contains the new enabled services, another is the reProServices which contains the services that need to be reprocessed.

**Algorithm:** Our algorithm starts by populating the queue that contains unprocessed enabled services. The queue is initialized with the Web services that can be directly enabled by $I_R$. We then examine the output parameters of each queue item ($W_i$) in turn. For each output parameter in $O_{W_i}$, we first check if its optimal QoS is already known (Line 4-8 in Algorithm 1, Line 1, 2 in Procedure 2). We then take the following two alternative steps.

Step 1: If optimal QoS is not yet known, we record the parameter together with the current computed optimal QoS. At the same time, we decrease the count of all services based on the number of parameter-matches between each of them and $O_{W_i}$. If the count of a service $W$ becomes zero, we say $W$ is an enabled service and insert it into the queue that contains the unprocessed enabled services (Line 9-20 in Procedure 2).

Step 2: If we have already recorded an optimal value for $O_{W_i}$, we compare the currently computed QoS value with the recorded one. If we have a better result, we update the corresponding entry to store the new optimal value and its provider. Note that since we have a new optimal value for an output parameter, all previously processed services which need this output parameter should be reprocessed by inserting them into another queue that contains the reprocessed services no duplicately (Line 2-7 in Procedure 2).
mal QoS value. 3) To make our algorithm more efficient, the enabled services whose allQoS are worse than current $O_R$.allQoS value are pruned (line 14,15 in Algorithm 1 and line 15,16 in Procedure 2). Because the parameters and services always record the optimal QoS value in the execution process and these optimal QoS values are updated whenever there are better QoS values, this approach is called two-level optimization mechanism.

Procedure 2 ForwardEnable(W)

Require: Service: W, OPars, reProServices, Input inverted index table: inTable
1: for all parameter out $\in W$.outputs do
2: if $\exists$ item $\in$ OPars where item = out then
3: if item.optQoS is worse than $W$.allQoS then
4: item.optQoS $\leftarrow$ W.allQoS
5: item.optProvider $\leftarrow$ W
6: reProServices.addUnique(inTable.lookup(out))
7: end if
8: else
9: OPars.add(out); out.optProvider $\leftarrow$ W; out.allQoS $\leftarrow$ W.allQoS
10: A services set (setNeed) $\leftarrow$ inTable.lookup(out)
11: for all Service $W' \in$ setNeed do
12: $W'$.count--
13: if $W'$.count = 0 then
14: $W'$.allQoS $\leftarrow$ W'.calculateQoS()
15: if $W' \neq O_R \land W'$.allQoS better than $O_R$.allQoS then
16: enabledServices.add($W'$)
17: end if
18: end if
19: end for
20: end if
21: end for

4.2.2 Generate the optimal result

After the above operations, the optimal composition result can be generated by a backward search from the end node $O_R$ to the start node $O_R$. For each service, we find the optimal predecessors by its inputs and their corresponding optProviders. Thus, a DAG is generated to represent the optimal QoS service composition. We also take Figure 1 as an example, and the process of the backward search is shown by double arrow edge.

4.3 Time Complexity and Soundness

This section presents the time complexity and some properties of our algorithm. At first, we focus on computing the time complexity of $O(worklist algorithm) + O(generating the result)$. Two time complexity are presented. The first one is used to guide the design and improvement of our algorithm, while the second one shows the upper bound of our algorithm in terms of nodes and edges number.

The first time complexity: Let $n$ be the total number of all Web services or the nodes in the graph, and $c$ be the average number of inputs of them or the average number of incoming edges. Let $a$ be the average reprocessed times and $n'$ be the reprocessed services ($n' \leq |W_{enabled}| \leq n$). The total upper bound time is $O(un'c + nc)$. That’s why we try to use no multiple appearances optimization to reduce the reprocessed times $u$, and use filtering to reduce the $n'$.

The second time complexity: Let $n$ be the total number of all Web services or the nodes in the graph, and $m$ be the total number of all edges. ($m \leq n \times c$), for some inputs may be not enabled. The upper bound of our algorithm is $O(mn)$.

Next, we present three Lemmas of our algorithm: (1) it will stop; (2) it can find a solution as long as there are solutions for the request; (3) our solution has the global optimal QoS value.

5 Evaluation

Our evaluation consists of three parts: we compare Yan’09 [17] and Peter’09 [4] with QSynth by the precision, efficiency and memory usage; we analyze the optimization effect of no multiple appearances of services in reProServices of QSynth. We also carry out some experiments on real QoS data sets to evaluate the three systems furthermore. Our workload is generated by the WSC 2009 Testset Generator. This generator has three input parameters: concept number, service number and solution-depth. In order to prove the scalability and efficiency of our algorithm in different and even extreme conditions, we generate three groups of data sets and each group contains six different test sets by varying these three parameters. We conduct experiments on each test set several times to get the average values.

5.1 Compare With Other Systems

5.1.1 Accuracy and Query Time

The accuracy and query time are the evaluation criterions of WS-challenge 09. The result can be found in [1]. From our experiments with different service number (1000-10000), concept number (1000-25000) and solution depth (4-30), we get the same conclusion to the WS-challenge 09 result.

If you are interested in time complexity analysis and proofs and other details, you can refer to the appendix which is available at http://debs.ict.ac.cn/QSynth.pdf.

We remove the appendix for the page limitation.
The experiment results are shown in Figure 3 and Figure 4. Generally, the accuracy order: QSynth (100%) and Yan’09 > Peter’09. The query time order: QSynth < Peter’09 < Yan’09. All the three system get the same optimal throughput and the results are not presented. Besides this, we also use the real QoS values of real Web service to do experiments and obtain the same conclusion in Figure 3 [4]. In [2] and [3], the authors collect QoS values of 2,507 real Web services. We extract the the response time and throughput from them. Then we assign the scores to the Web services generated by WSC 2009 Testset Generator. Because the Web services in real data set do not always abide by the standards and many of them are too simple. Also the requests are not available in real data set.

5.1.2 Memory Usage

These experiments aim at evaluating the memory usage of different systems. Figure 4(4) shows that memory usage of the three system with different number of services. The result is that the memory usage order: Yan’09 < QSynth < Peter’09. In the last test set, Peter’09 is out of memory. The memory usage of QSynth grows linearly with the increasing number of services. Other test sets show the similar trend, we omit them here.

5.2 Optimization Effect

All the above experiments are performed on QSynth that does not contain multiple appearances of services in the reProServices. Now we study the difference if QSynth does not adopt this optimization and re-processes these services whenever one of their inputs updates to a better optQoS. This experiment is performed on the data sets of different number of concepts and the result is presented in Table 3. In the best case of it, query time is reduced from 958.5ms to 333.3ms and the updated times of allQoS of services are reduced by 94%.

<table>
<thead>
<tr>
<th>Concept Number</th>
<th>Optimization</th>
<th>No optimization</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Time-Update Times</td>
<td>Update Times</td>
<td>Query Time</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>115-130</td>
<td>146-730</td>
<td>600</td>
</tr>
<tr>
<td>5000</td>
<td>333.3-4761</td>
<td>958.5-77412</td>
<td>72651</td>
</tr>
<tr>
<td>10000</td>
<td>109-32</td>
<td>109.7-132</td>
<td>100</td>
</tr>
<tr>
<td>15000</td>
<td>78.5-2</td>
<td>78-2</td>
<td>0</td>
</tr>
<tr>
<td>20000</td>
<td>78-16</td>
<td>93.7-132</td>
<td>116</td>
</tr>
<tr>
<td>25000</td>
<td>88.3-2</td>
<td>88.7-2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3. The optimization of reProServices

From the above experiments, we can come to a conclusion that our algorithm exhibits stable performance with all variable factors including service number, concept number, the solution depth and memory usage. It is a practical algorithm that fits large scale service composition in reality.

6 Conclusions

To handle QoS-aware service composition automatically and efficiently, it is not a good strategy to split the problem into two sub problems: automatic service composition without the considering of QoS, QoS-aware service selection with the assumption that we know the pre-defined abstract processes and all the candidate services for each abstract task. To address the scalability, precision and performance problems of QoS-aware automatic service composition, an efficient tool QSynth is proposed and implemented in this paper. QSynth adopts a worklist algorithm which combines three kinds of filtering and two-level optimization approach in the forward search, also a backward search is executed to obtain the optimal service composition result. By these mechanisms, the search space is narrowed down and the precision is guaranteed. Experiments on simulative and real QoS data sets show that QSynth has good scalabi-
ity, efficiency, 100% precision with smaller memory usage than other available systems.

Our future plans are enhancing QSynth to handle dynamic QoS data and designing better service composition algorithm. In order to do service match precisely, the semantic information is very important, we will care about the new advances in semantic Web service and integrate them in the next version of QSynth.

7 Acknowledgments

For the completion of this research, we would like to thank the National Basic Research Program of China under Grant No.2007CB310805, the Beijing science and technology plan projects under Grant No.Z09000100960907, the National Natural Science Foundation of China under Grant No.60752001, the Beijing Municipal Natural Science Foundation project under Grant No.4092043.

References