Continuous Query for QoS-Aware Automatic Service Composition

Wei Jiang[1,2], Songlin Hu[1], Dongwon Lee[3], Shuai Gong[1,2], Zhiyong Liu[1]
[1]Institute of Computing Technology, Chinese Academy of Sciences, China
[2]Graduate University of the Chinese Academy of Sciences, China
[3]The Pennsylvania State University, USA
{jiangwei,husonglin,gonfshuai,zyliu}@ict.ac.cn, dongwon@psu.edu

***Abstract*—Current QoS-aware automatic service composition queries over a network of Web services are often one-time in nature. After a network of Web services is built, such queries are issued once, and answers are found from the scratch. The underlying assumption is that the participating Web services are rather *static* so that their functional and non-functional parameters seldom change. However, such an assumption is often baseless. New services come and go, service APIs change gradually, and QoS values fluctuate. Therefore, a support for efficiently handling "continuous" service composition queries is desired. In this paper, we propose an event driven continuous query algorithm for QoS-aware automatic service composition problem to cope with different types of dynamic services. Moreover, we integrated this algorithm in our service composition system, QSynth. Finally, we evaluate our proposal using both real QoS data and synthetic Web service data and show the superior performance of ours, compared to the state-of-the-art solution which won the performance championship of Web Service Challenge in 2009 and 2010.**

I. Introduction

The number and diversity of Web-accessible services have been increasing rapidly, and are expected to keep growing in the near future [1]. Although so many services can provide more choices for SOA-based applications, they also bring about a heavy burden on modeling and verification of service composition that plays a very important role in SOA methodology. To meet this challenge, **Automatic Service Composition (ASC)** problem [2] has been proposed to automatically discover and compose multiple services to satisfy a given query with a pair of inputs and outputs. No predefined process template is needed in this problem. Furthermore, to guarantee the overall Quality-of-Service (QoS) of composite services, different composition techniques have been studied in [3], [4], [5], [6] to generate composition service with favorite QoS as well as correct function. By and large, all of the aforementioned approaches share a common assumption such that characteristics and behaviors of existing Web services seldom change.

However, in real settings, such a rigid assumption on the static characteristics of Web services often fails. New Web services are introduced and unpopular ones are discontinued daily. Working Web services can become invalid temporarily or indefinitely due to problems like unforseen network errors. Interfaces and QoS values of existing services may change at any time. In order to recognize the dynamic nature of Web services, we conduct eleven week-long survey of Web services on Internet, which are collected from seekda.com, webservicelist.com, and xmethods.net. Table I presents the different values of our collected Web service data for every two continue weeks by several criteria.

Therefore, we emphasize that Web services are not *static* but *dynamic*. In such a dynamic environment, a composition of services found to satisfy a request r at time t_1 can easily become an invalid answer if part of the composition becomes unresponsive or some QoS values change. To be able to cope with such a situation, in this paper, we claim that the continuous support for automatic service composition requests is greatly desired, regardless of the changes in the underlying service network and/or components therein.

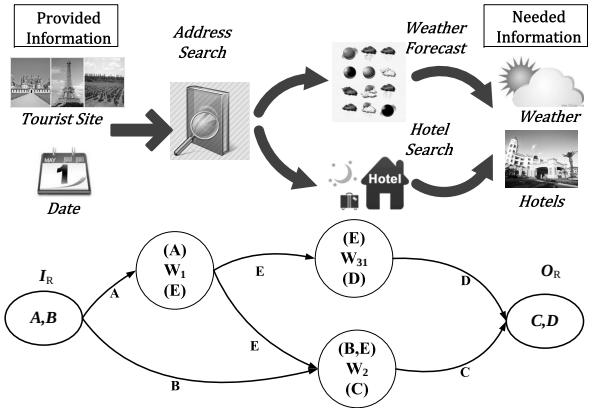
In general, we can divide the queries of service composition into two types: (1) the *one-time query* is handled only once over a snapshot of available services at time t; and (2) the *continuous query* is to run continuously over the service network and produce new service composition as the network changes dynamically.

Example 1. Let us illustrate the problem with Fig.1(a), where a company provides a one-stop service for travelers using three Web services (i.e., address, weather, and hotel services) together. Table II shows the details of these services. When some component services in Fig.1(a) become invalid (e.g., W_{31}), the service composition can be maintained via service replacement (top of Fig.1(b)). In other words, we keep the logic structure of this process and replace the unavailable services by other services whose functions are equivalent to the unavailable ones. However, in order to cope with service changes, we sometimes have to reschedule the composition logic rather than only replacing some nodes, as shown in the bottom of Fig.1(b). Furthermore, the new composition may even use a completely different set of services. Thus, n-m replacement is usually needed besides the simple 1-1 replacement [7]. □

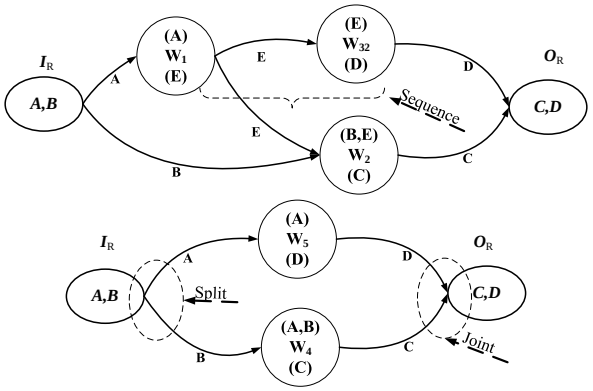
Although adaptive *service selection* for dynamic services is discussed in [8], [9], [10], [11], [12], few work has done on dynamic QoS-aware *automatic service composition* problem. Thus, we propose a novel event driven continuous query mechanism for it. Service selection usually requires composition template as predefined process, which contains abstract service classes and each of them need to be bound to a concrete service at runtime. While this requirement is not necessary in QoS-aware ASC problem, which can generate

Table I
THE CHANGE OF PUBLIC WEB SERVICES FROM INTERNET

Criteria	*Week1-2*	*2-3*	*3-4*	*4-5*	*5-6*	*6-7*	*7-8*	*8-9*	*9-10*	*10-11*
# of Common URL	11370	11823	11255	11306	10240	10355	11786	11697	11635	11518
# of unchanged WSDL	10888	11321	10404	10623	9675	9863	11239	11122	11104	10889
% of changed WSDL	4.24%	4.25%	7.56%	6.04%	5.52%	4.75%	4.64%	4.92%	4.56%	5.98%
# of delete operation	61	11	179	380	60	52	109	13	94	24
# of new operation	0	65	555	35	145	25	66	55	62	342
# of inputs/outputs changed operation	78	29	271	143	88	90	91	49	40	144



(a) A service composition example



(b) Two new service composition examples

Figure 1. Service Composition Example

Table II
THE SERVICES AND REQUEST IN FIGURE 1

Web Service & Request	*Inputs*	*Outputs*	*Description*
Request(R)	Tourist Site(A) Date(B)	Weather(C) Hotels(D)	The request provides tourist site, date and needs the weather and hotels of the city.
Address Search(W_1)	Tourist Site(A)	City(E)	It finds the city which the tourist site in.
Weather Forecast(W_2)	City(E) Date(B)	Weather(C)	It returns the weather of the city on the date.
HotelSearch (W_{31}, W_{32})	City(E)	Hotels(D)	It finds hotels in the city.
Weather Forecast(W_4)	Tourist Site(A) Date(B)	Weather(C)	It returns the weather of the tourist site on the date.
Hotel Locate(W_5)	Tourist Site(A)	Hotels(D)	It finds hotels near the tourist site.

the composition automatically. In fact, our approach can also be used to support the adaptive selection when certain branches of the process need to be rescheduled. In addition, our approach can be aware of both the changes of the "internal" services of a certain service composition and the "external" services that might affect it. Note that the changes of "external" services might result in a new composition result with better overall quality. Moreover, unlike most current service selection approaches that support adaptation of service compositions in a reactive way, we tackle dynamic service in an event driven way. Finally our continuous query can handle multiple dynamic services together and update all affected service composition results in a batch processing way.

The general idea of our continuous query supported QoS-aware ASC approach is: First, we build a dependency graph among available services and store the graph efficiently by two inverted index tables. Second, we start a forward search to find the enabled services whose inputs have been fully satisfied and obtain the optimal service composition result by a backward search then. Third, whenever underlying services change, we update the dependency graph and the status of affected services therein. Finally, we obtain the new optimal service composition result if the original one becomes invalid or sub-optimal. The first two steps are presented in our previous one time query algorithm [3]. This paper focuses on the last two steps which are used to support continuous query.

Because our approach avoids requery of service composition from scratch and only searches a small set of affected services, it yields better performance. Besides, the requery approach does not know directly which services and composition results will be affected after dynamic service occur, so it is impossible for the requery approach to decide which query should be re-executed once again. This shows additional advantages of our approach.

The main contributions of this paper are as follows:

(1) We first introduce continuous query mechanism into QoS-aware automatic service composition problem, which effectively handles dynamic services in an event driven way. It can not only support replacement of services or branches of the affected compositions, but also enables rescheduling of the composition logic if needed.

(2) We present an efficient algorithm to support contin-

uous queries. It is not only awareness of new and invalid services, but also sensitive to QoS and interface changes.

(3) We implement the continuous query algorithm in our system, QSynth, and validate it using both real and synthetic QoS data.

II. Background

This section describes QoS-aware automatic service composition problem and our one time query solution briefly.

A. Notations

Definition 1 (Optimal QoS-Aware ASC Problem) *Given a set of services and a query or request R, find a specific service composition SC that defines an invoking structure (not only sequence) over a set of Web services $(W_1, W_2, \ldots, W_N)$ by satisfying the following conditions (Let I, O be the inputs and outputs of request and Web services):*

$(1)\{I_R \cup O_{W_1} \cup \ldots \cup O_{W_i}\} \supseteq I_{W_{i+1}}$ $(1 \leq i \leq N-1)$;
$(2)\{O_{W_1} \cup O_{W_2} \cup \ldots \cup O_{W_N}\} \supseteq O_R$;
(3) *The overall QoS of SC is optimal among all the service compositions that can enable R;*

In brief, given a set of services and a query R, our one-time query aims at finding the optimal service composition result for R. To make it simple for us to illustrate our approach, we assume that each service has one dimension of QoS value in this paper, e.g., response time. If there exists multiple QoS measurements, one approach is using Multiple Attribute Decision Making approach, i.e., simple additive weighting, to transform all the QoS values into one aggregated QoS score before conducting our algorithm. Another approach is to use skyline [13] to handle multiple dimensions of QoS values as we have done in [14].

B. One-Time Query

In this part, we describe how to handle one-time query by QSynth. QSynth is our system for semantic optimal QoS-aware ASC problem [3], [15], which is also the championship of Web Service Challenge 2009, 2010 [16]. Given a query, it extracts Web services information from WSDL, OWL-S, WSLA files and returns the answers with optimal response time and throughput in the format of BPEL file.

Graph Construction: QSynth first collects the interfaces of Web services and builds a *dependency graph* for them like Fig.2, where we connect W_A to W_B if one of W_A's outputs *matches* one of W_B's inputs. Every node in the graph represent a service with an unique id. Inputs, outputs of the service are shown above and below its id respectively. Every edge has an attached *parameter* that indicates the *match parameter* between a service and its successor service. Meanwhile, every service as a node in the dependency graph has a QoS value, response time. The I_R and O_R of R are inserted into the dependency graph as two virtual nodes,

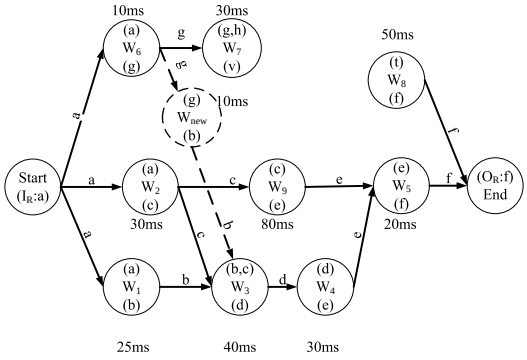


Figure 2. Dependency Graph

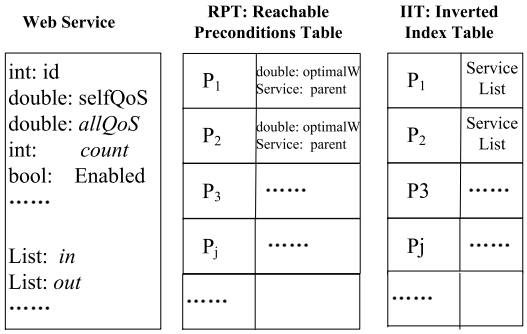


Figure 3. Data Structure

Start and End. A candidate service composition result is essentially a connected sub-graph in this dependency graph satisfying the basic condition that the union of inputs of the direct successors of Start is a subset of R's inputs (I_R) and the union of the outputs is a superset of R's outputs (O_R). The final solution of one time query is to find such a sub-graph that gives the optimal overall QoS value.

Data Structure: Our data structure is presented in Fig.3. We use two inverted index tables to represent the graph. Every entry in the *input* inverted index table is a tuple, $(parameter, services\ list)$, where $services\ list$ includes those services whose inputs contain $parameter$. Similarly, we build the *output* inverted index table.

Each service W_i in the dependency graph is represented as $W_i = \{I_{W_i}, O_{W_i}, selfQoS, allQoS, count, Enabled\}$. $selfQoS$ is its own QoS value of W_i. $count$ is assigned to the size of I_{W_i} initially. We say the service W_i is *enabled* if its $count$ goes to zero. Whenever a new enabled services (Start is the first enabled service) can match one input of W_i (that input is not matched/enabled before), $count$ of W_i decreases one. $Enabled$ value will be set to TRUE when $count$ is 0. $allQoS$ records the optimal[1] overall QoS from Start to current service. A hash table, Reachable Preconditions Table (RPT) is designed to store the optimal overall QoS ($optQoS$) for each *enabled input* and its optimal predecessor ($parent$).

[1]There may be several candidate DAGs from Start to current service, which results in different overall QoS values for the current service. Only the best one is assigned to $allQoS$.

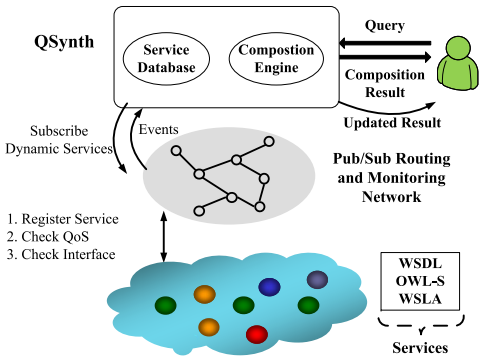


Figure 4. Architecture of QSynth that supports CQ

Solution: We exploit the algorithm in [3], [15] to find the optimal service composition result. It contains two steps. The first step is conducting a forward search from Start node and recording enabled services, enable inputs and their $parent$s. The second step is executing a backward search from End to generate the optimal composition result guided by the enabled inputs and their $parent$s, which are recorded in RPT. An example of one time query is described in the appendixes of this paper [17].

III. Dynamic Services and Their Affections

A. Dynamic Services

In this paper, *Dynamic services* refer to the following four categories of services : new services, unavailable services, services whose interfaces change, other services which are available but their $selfQoS$ change.

The current architecture of QSynth is shown in Fig.4. QSynth receives events about dynamic services from the pub/sub network by the approaches in [18]. For example, when response time of service W_i is bigger than a threshold (e.g., 3s) or a new operation is added, an event will be triggered and sent to QSynth by pub/sub network, containing information about this dynamic service.

B. Dynamic Services Affections

Dynamic services will influence on: dependency graph, affected services and composition results. Let's discuss them respectively.

First, dynamic services may change the structure of *dependency graph*. As we represent the graph by two inverted index tables, it's easy to update dependency graph by deleting the corresponding items or adding new items in the invert index tables. For example, if W_{new} with input g and output b is added into Fig.2, we can update the original graph by finding key g in input inverted index table and adding W_{new} to the corresponding list. In a similar way, we can update the output inverted index table as well.

Second, dynamic services may affect status values (values of $allQoS$ and $Enabled$) of some related services, e.g., some *enabled* services may become *unabled*. We call them "affected services". *Affected services* refer to these services whose *new* $allQoS$ are different from the original ones because of dynamic services and their propagative effects. To cope with affected services, we update the status of them by a forward traverse which starts from dynamic services until the current services in all branches are not affected. Note that this is not a trivial task. Take Fig.2 as an example, we assume that the $selfQoS$ of W_1 and W_2 are changed and they both are dynamic services. If we update W_1 first and update W_3, W_4, W_5 after that, we have to update W_3, W_4, W_5 again when we handle W_2. A more efficient way is: we update W_1, W_2 first and update W_3, W_4, W_5 after that. So it is important to avoid this kind of reductant updates and renew the status of affected services in an efficient way. This is an important problem for continuous query.

At last, the dynamic services may result in that the original optimal service composition becomes invalid or sub-optimal. In this case, we need to generate a new composition result. But it is too costly to generated new result whenever dynamic services occur like requery approach, we need to judge whether the new optimal composition result should be generated or not by an effective method. We will discuss how to address the above problems in the following section.

IV. Continuous Query Algorithm

We first introduce the categories of affected services and discuss their handling order then. Finally, we give the details of our continuous query algorithm.

A. Categories of Affected Services

Concretely, the influence of dynamic services are: Type 1: some services are changed from enabled to unabled; Type 2: some services are changed from unabled to enabled; Type 3: the overall QoS of some services become better; Type 4: the overall QoS of some services become worse. In fact, we can divide all the above ones into only two categories. In the following, the left of arrow is the old $allQoS$, the right is the new one.

First category: the affected services whose $allQoS$ become better. It contains two situations: $X \rightarrow X - L$(type 3) and $+\infty$[2] $\rightarrow X$ (type 2) $(0 < L < X < +\infty)$.

Second category: the affected services whose $allQoS$ become worse. It contains two situations: $X \rightarrow X + L$ (type 4) and $X \rightarrow +\infty$ (type 1) $(0 < L, X < +\infty)$.

B. Algorithm

General Idea: Given a continuous query, we firstly tackle it by our previous approach for one time query in [3], [15]. Meanwhile, enabled services and RPT are stored. When dynamic services are notified to QSynth, we first determine whether they will affect the status of other services. If not, we only need to update the graph structure and dynamic services' status. Otherwise, we conduct a forward search

[2]When an available service is unabled, its $allQoS$ is $+\infty$. When a service is unavailable, its $allQoS$ is $+\infty$ too.

starting from dynamic services and only update its successors that become affected services. This process continues until all current successors are unaffected. An update order that avoids re-processing of affected services is adopted. Finally, we judge if it is necessary to generate a new service composition result by a simple rule: whether the original service composition contain *final* affected service[3] .

Update Order: We give a simple intitule explanation for it. Inspired by Dijkstra algorithm [19], we make use of a similar order to update the status of affected services. The distance changes[4] in Dijkstra algorithm are $+\infty \rightarrow X$ $(0 < X < +\infty)$. In our problem, for the first category of affected services whose $allQoS$ got improved, we can handle it like Dijkstra algorithm; For the second category, we transform it into the first category by assigning the original $allQoS$ of these services to $+\infty$, while its new $allQoS$ is not reassigned. Note that the original $allQoS$ is not valid for the generation of new service composition. Only the new status of these services with new $allQoS$ will decide the new composition result. The proof is skipped because of page limitation.

The Details of Algorithm: Our algorithm is presented in Algorithm 1. Concretely, two new members, $newAllQoS$ and $pqQoS$, are added to our data structure first. $allQoS$ of W_i records the original overall QoS from Start to W_i in last service composition search. $newAllQoS$ is the new overall QoS of W_i after dynamic services are processed. If $newAllQoS \neq allQoS$, this service is an affected service. We will push it into a priority queue and handle it later. The $pqQoS$ ($pqQoS = \min(newAllQoS, allQoS)$) decides priorities of the items in the priority queue.

Step 1: Handle all dynamic services and put the affected services of them into the priority queue (line 1-14 in Algorithm 1): Concretely, for each service whose interface is changed, it is coped as a combination of service deletion and service adding. Besides, we handles the remaining dynamic services in step 1.1-1.3 respectively. All the affected services among them will be inserted into priority queue, whose $allQoS \neq newAllQoS$.

Step 2: Update the status of affected services (line 15-26 in Algorithm 1): We handle each item in the priority queue iteratively until it is empty. At each loop, the item with the smallest $pqQoS$ in current priority queue is popped and is handled by the following two ways:

(1) If the popped item belongs to the first category of affected services, we will update its direct successors. The successors whose $newAllQoS$ are different from $allQoS$ will be pushed into priority queue too (lines 19-22).

(2) If the popped item belongs to the second category of affected services, we will assign its $allQoS$ to $+\infty$. If its $newAllQoS$ is not equal to $+\infty$, we will push it into priority queue again (In fact, this item becomes the first category of affected services). Otherwise, if its $newAllQoS$ is $+\infty$ and it is an unavailable service, we remove it from graph (lines 24-26).

Step 3: After the priority queue is empty, we generate the new composition result if necessary (lines 27-30).

(1) If the request's overall QoS equals $+\infty$, it means that no service composition can satisfy the request.

(2) Otherwise, we judge whether the original optimal service composition result contains *final* affected service. If so, a backward search is conducted to obtain new result. If not, the original optimal composition result is kept.

A case study of continuous query algorithm in presented in the appendixes[17].

Algorithm 1: Continuous Query Algorithm

Input: Interface changed Services: $InterS$, New Services: $NewS$, Unavailable Services: UnS, Others Dynamic Services: $otherDS$, Priority Queue: PQ

1 For each Services $iS \in InterS$, put the original one to UnS and the new one to $NewS$;

`//Step1.1: unavailable services`
2 **foreach** *Service* $delS \in UnS$ **do**
3 $delS.newAllQoS \leftarrow +\infty$;
4 **if** *delS.allQoS* == $+\infty$ **then** remove it from graph;
5 **else** $PQ.add(delS)$;

`//Step1.2: new services`
6 **foreach** *Service* $addS \in NewS$ **do**
7 add $addS$ into graph;
8 $addS.allQoS \leftarrow +\infty$;
9 **if** *addS.newAllQoS*==$+\infty$ **then** $Continue$;
10 **else** $PQ.add(addS)$;

`//Step1.3: dynamic services whose QoS are changed`
11 **foreach** *Service* $othS \in OtherDS$ **do**
12 **if** *PQ.find(othS)== true* **then** $adapt(othS)$;
13 **else if** $othS.allQoS \neq +\infty$ **then** $PQ.add(addS)$;
14 **else** $Continue$;

`//Step2: handle Priority Queue`
15 **while** $PQ \neq \emptyset$ **do**
16 $first \leftarrow PQ.pop()$;
17 **if** *first*==O_R **then**
18 $first.allQoS \leftarrow first.newAllQoS$; $Continue$;
19 **if** *first.newAllQoS* < *first.allQoS* **then**
20 $first.allQoS \leftarrow first.newAllQoS$;
21 **foreach** *par* $\in O_{first}$ **do**
22 $adapt$(services whose $allQoS$ are changed because of par);
23 **else**
24 $first.allQoS \leftarrow +\infty$; update $first$'s direct successors;
25 **if** *first.newAllQoS*$\neq +\infty$ **then** $PQ.add(first)$;
26 **else** remove $first$ from graph if it is unavailable;

`//Step3: generate composition result`
27 **if** $O_R.allQoS == +\infty$ **then**
28 **return** no result;
29 **else if** *Stored service result contains final affected services* **then**
30 **return** new result by backward search;

[3]Our approach can handle several dynamic services together. Thus, a service may become affected service because of some dynamic services and then become unaffected service because of others. A *final* affected service means its $allQoS$ is different from the original value after all affections are processed.

[4]Let V_s be the start node, the distance of every node is the distance from V_s to it. Then, a forward search is conducted from V_s, all the reachable nodes are inserted into a priority queue with their distances. Every time, the node with the smallest distance in the current queue is removed first. The Dijkstra algorithm handles its successors and updates their distances until the queue is empty.

C. Discussion

We elaborate our discussion to the following questions.

$Q1$: How does our continuous query algorithm guarantee that it can get the optimal service composition result?

$Q2$: How does our algorithm avoid redundant renewal of affected services?

$Q3$: What's the cost of our continuous query algorithm? Is there any side effect?

$Q4$: What's the time complexity of continuous query algorithm?

Because of page limitation, the discussion about these four questions can be found in the appendixes [17] .

V. Evaluation

In this section, we present an experimental evaluation on three approaches: our continuous query algorithm (CQ-Order), unordered continuous query algorithm (CQ-No order) that update affected services in a random order, and re-query approach (Re-Query) that we re-execute one-time query whenever a dynamic service is reported. We will measure (a) **efficiency**, in terms of the cost time which covers from the time that QSynth receives the events of dynamic services to the time that current service composition is updated, and (b) **sensitivity**, which means whether these three approaches are sensitive to the degree of QoS value change, and (c) **accuracy**, which means whether the updated service composition result generated by our algorithms are still have the same overall QoS to the result returned by Re-Query approach. These approaches are compared on a 2.4GHz machine with 2 GB RAM running Windows 7. We conduct these experiments on both real and synthetic QoS data.

A. Data Set

Our synthetic data set is generated by public WS-Challenge Testset Generator [16]. We compare the three approaches under different service number and different dynamic service number. Dynamic services are generated randomly. For our real QoS data set, the response time of about 8000 real Web services at different timestamps are collected[5]. Then, we replace the synthetic response time with real response time for the generated Web services[6].

B. Different Number of Registered Services

This experiment aims to evaluate the efficiency of three approaches with respect to different number of registered services that ranges from 1000 to 8000. The number of dynamic service is 100. The total cost time of handling all the dynamic services one by one are presented in Figure 5(1). Note that all the figures use a logarithmic scale. The result shows that CQ-order yields the best performance. With the growing of services, re-query approach costs more time since it needs to search a bigger search space. While cq (cq-order, cq-no order) algorithms depend on the number of dynamic services and affected services, not the registered services, so they show a fairly stable performance. Meanwhile, cq-order is better than cq-no order by 4%$\sim$30%, this is because the redundant update of affected services in cq-no order algorithm. The result of real QoS data is shown in Fig.5(3) with a similar trend. Moreover, the composition results returned by cq-order, cq-no order have the same overall QoS as that of re-query. This convinces the accuracy of continuous query algorithms. Other experiments in the following show the same conclusion as well.

[5]It is available at http://debs.ict.ac.cn/realqosdata.rar.

[6]We do not use real Web services in the experiment. Because there is still not enough semantic information for their WSDL documents now. Different services may describe the same thing with various words, it is not trivial to do Web service match accurately without semantic information, so we have to use generated WSDL documents which contain semantic information.

C. Different Number of Dynamic Services

This experiment aims to evaluate the efficiency of three approaches under different number of dynamic services. The registered service number is 6000, while the dynamic services range from 50 to 250. As illustrated in Fig.5(2), the cost time of re-query approach grows nearly linearly with the increment of the number of dynamic services. That's because the re-query of each dynamic service costs almost the same time under the same test set. While the cost time of continuous query algorithms are growing not linearly since their cost time depend on both dynamic services and affected services. While the affected services are various with regard to different dynamic services. The result for real QoS data is shown in Fig.5(4) with a similar trend.

D. Different QoS Change

This experiment aims to evaluate the sensitivity of three approaches to the degree of QoS value change. Fig.6(1) presents the result of test set whose registered service number is 6000 and dynamic service number is 100. These dynamic services' $selfQoS$ are added 10ms, 50ms, 100ms, 200ms, and 300 ms respectively. But their original $selfQoS$ are different. The result shows that these three approaches are not sensitive to the degree of QoS change. That's because the time complexity of continuous query algorithms do not depend on the degree of QoS change, so does re-query approach.

E. Cost Time of Each Dynamic Service

This experiment aims to compare the cost time of handling each dynamic service. Fig.6(2) shows the result of test set with 6000 registered services and 100 dynamic services. The other cases are similar and skipped. The same to our expectation, the cost time of each dynamic service by re-query approach is greater than that of continuous query algorithms, because continuous query only processes dynamic services and affected services which are only a small part of the entire service space. The cost time of

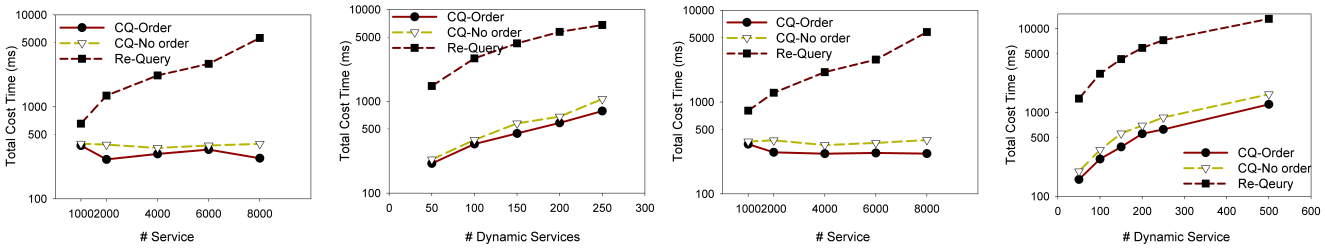


Figure 5. (1) Total cost time(# of services) (2) Total cost time(# of dynamic services) (3) Total cost time(# of services) (Real QoS) (4) Total cost time(# of dynamic service)(Real QoS)

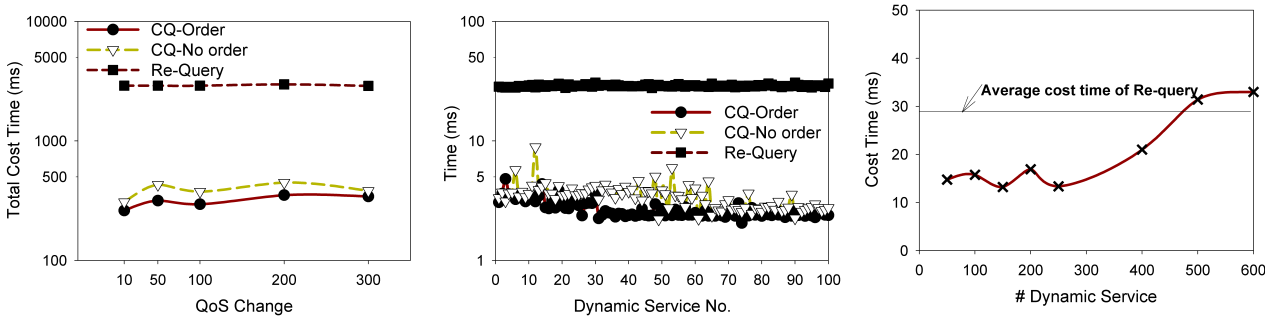


Figure 6. (1) Different QoS changes (2) Cost time for each dynamic service (3) Special Case

each dynamic service by re-query approach is stable because of the same test set. However, the cost time of continuous query algorithms are different, for the affected scope of each dynamic service is various.

F. Special Case

There may be special case where we need to handle huge dynamic services in one update progress. Although this is not very practical in real scenario, we may still handle many dynamic services together to reduce the cost time with the loss of real-time update (e.g., periodic update). Fig.6(3) shows the cost time of cq-order approach when the number of dynamic services ranges from 50 to 600 in one update progress. The cost time is between 16 and 32ms, which is better than 600*2.7 = 1620ms largely if we handle these dynamic services one by one (the average cost time of each dynamic service of CQ-Order is 2.7ms as shown in Fig.6(2)). However, the average cost time of re-query is 28.9ms. Thus, in this test set, when the number of dynamic service is over 500 in one update process, it is better to use re-query.

From the above analysis, we can come to a conclusion that our continuous query algorithms exhibit stable performance with the variable factors including service number, dynamic service number, and the degree of QoS change. Our continuous query algorithms are also optimal just like the re-query approach. It beats the re-query approach of current Web Service Challenge champion by 10 times except special case.

VI. Related Work

Research in [8], [10], [11] discusses how to replan or rebind in **service selection problem** when some services are different from the declared QoS or become invalid. Mixed integer programming approach [8], region-based service reconfiguration approach [9], IP approach [10] and Markov decision process solution [11] have been proposed. The work in [12] first selects multiple services for every abstract service class of a complex process template by a heuristic algorithm, which aims at determining a set of near-optimal service compositions. At runtime, if a specific service composition is no longer possible or its QoS decreases, an alternative composition will be executed. On the whole, these approaches always assume the existence of a predefined abstract process, a set of ”abstract” tasks or service classes, and the service instances with same functionality for each task. This assumption is not needed in our approach. Besides, these approaches take action at runtime in a reactive way and does not discuss much about rescheduling the composition logic. While our approach takes action as early as dynamic services are monitored in an event driven way. Our approach can also reschedule the composition logic if necessary.

With the consideration of non-functional preferences of users and mass services available nowadays, QoS-aware automatic service composition has attracted a lot of attraction [3], [4], [5], [6]. Web service research community has hosted Web Service Challenge [16] to solicit software for the problem. However, they merely focus on how to efficiently

retrieve the optimal or near-optimal service composition results on the assumption of static services. A naive approach to handle dynamic services is to reqeury from scratch whenever a change occurs. But it has several limitations as we declared in the section of introduction. Although research in [20], [7] show how to address dynamic services for automatic service composition problem in AI planning context, it does not consider QoS criteria, which simplifies the problem. Research in [21] shows how to address service composition efficiently with the consideration of QoS in dynamic environment by treating the problem as the shortest path problem. However, service composition tends to find DAG-like results containing sequence, split and join patterns and so on, rather than paths or chains.

In DBMS field, in order to support continuous query, a lot of database management systems [22], [23], [24] have been proposed. But they usually make tradeoff among efficiency, accuracy, and storage. Data stream are usually considered as append-only and approximate algorithms are adopted in most cases. The related technologies of it contains: sliding window, batch processing, sampling and synopsis structure. However, many events about dynamic services are about update which will modify the status of registered services. The sampling and synopsis structure can not adopted, because only the state of art status of services will influence new service composition and the history changes of Web services are not concerned. However, the history information is also necessary in DBMS. For example, a query in DBMS field may find how many times the price of a stock is above a certain price from two months ago.

VII. Conclusion

Existing QoS-aware automatic service composition approaches are usually based on the assumption of static Web services, but this is not true in real scenarios. In order to get rid of this strict assumption, we make use of continuous query mechanism and propose an event driven solution to handle different types of dynamic services. In this solution, only the region of affected services are updated instead of the whole Web service space. N-m substitution is supported instead of 1-1 substitution. Moreover, it avoids redundant updates of affected services. The evaluation also suggests that our continuous algorithm has good scalability and efficiency with respect to different test sets. It achieves much higher performance than re-query approach.

Acknowledgement

Dongwon Lee is in part supported by NSF DUE-0817376 and DUE-0937891 awards. Wei Jiang and Songlin Hu are supported by the National NaturalScience Foundation of China under Grant No.61070027, the Beijing NaturalScience Foundation project under Grant No.4092043, the Science and Technology Project of State Grid Information & Telecommunication Company Ltd. under Grant No.SGIT[2011]567 and the Planned Science and Technology Project of Guangdong Province, China under Grant No.2010B050100009.

References

[1] Eyhab Al-Masri and Qusay H. Mahmoud. Investigating web services on the world wide web. In *WWW '08*, pages 795–804, NY, USA, 2008. ACM.

[2] Annapaola Marconi and Marco Pistore. In *ICSOC*, pages 89–157. Springer-Verlag, Berlin, Heidelberg, 2009.

[3] Wei Jiang, Charles Zhang, Zhenqiu Huang, Mingwen Chen, Songlin Hu, and Zhiyong Liu. Qsynth: A tool for qos-aware automatic service composition. In *ICWS'10*, pages 42–49.

[4] Peter Bartalos and Maria Bielikova. Qos aware semantic web service composition approach considering pre/postconditions. *ICWS'10*, pages 345–352, 2010.

[5] F. Wagner, F. Ishikawa, and S. Honiden. Qos-aware automatic service composition by applying functional clustering. In *ICWS 2011*, pages 89 –96, july 2011.

[6] Ying Ma, Liang Chen, Jian Hui, and Jian Wu. Cbbcm: Clustering based automatic service composition. In *SCC 2011*, pages 354 –361, july 2011.

[7] Yuhong Yan, Pascal Poizat, and Ludeng Zhao. Repair vs. recomposition for broken service compositions. In *ICSOC*, pages 152–166, 2010.

[8] Danilo Ardagna and Barbara Pernici. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, 33:369–384, 2007.

[9] Yanlong Zhai, Jing Zhang, and Kwei-Jay Lin. Soa middleware support for service process reconfiguration with end-to-end qos constraints. *ICWS'09*, 0:815–822, 2009.

[10] Girish Chafle, Koustuv Dasgupta, Arun Kumar, Sumit Mittal, and Biplav Srivastava. Adaptation in web service composition and execution. *ICWS'06*, 0:549–557, 2006.

[11] Kunal Verma, Prashant Doshi, Karthik Gomadam, John Miller, and Amit Sheth. Optimal adaptation in web processes with coordination constraints. *ICWS'06*, 0:257–264, 2006.

[12] Nebil Ben Mabrouk, Sandrine Beauche, Elena Kuznetsova, Nikolaos Georgantas, and Valérie Issarny. Qos-aware service composition in dynamic service oriented environments. In *Middleware '09*, pages 7:1–7:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[13] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430, Washington, DC, USA, 2001. IEEE Computer Society.

[14] Wei Jiang, Songlin Hu, and Zhiyong Liu. Top k query in qos-aware automatic service composition. technical report. http://debs.ict.ac.cn/topk.pdf.

[15] Wei Jiang, Songlin Hu, and Zhiyong Liu. Qos-aware automatic service composition: A graph view. *Journal of Computer Science and Technology.*, 26(5):837–853, 2011.

[16] Web service challenge 2009. [online] http://ws-challenge.georgetown.edu/wsc09/.

[17] Wei Jiang, Songlin Hu, Dongwon Lee, Shuai Gong, and Zhiyong Liu. Continuous query for qos-aware automatic service composition [extended]. http://debs.ict.ac.cn/cq-extended.pdf.

[18] Wei Yan, Songlin Hu, Vinod Muthusamy, Hans-Arno Jacobsen, and Li Zha. Efficient event-based resource discovery. In *DEBS*, 2009.

[19] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[20] Yuhong Yan, Pascal Poizat, and Ludeng Zhao. Self-adaptive service composition through graphplan repair. *ICWS'10*, 0:624–627, 2010.

[21] Swaroop Kalasapur, Mohan Kumar, and Behrooz A. Shirazi. Dynamic service composition in pervasive computing. *IEEE Trans. Parallel Distrib. Syst.*, 18:907–918, July 2007.

[22] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02*, pages 1–16, NY, USA, 2002. ACM.

[23] Arvind, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26:2003, 2003.

[24] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: a scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2):379–390, 2000.