Concept Mapping for Faster QoS-Aware Web Service Composition

Viktoriya Degeler, Ilîe Georgievski, Alexander Lazovik, and Marco Aiello
Distributed Systems Group, Johann Bernoulli Institute
University of Groningen
Nijenborgh 9, 9747 AG, The Netherlands
{v.degeler, i.georgievski, a.lazovik, m.aiello}@rug.nl

Abstract—The availability of Web services with similar functionality but different QoS values creates new challenges for Web services composition: not only functional properties of the composed service must be satisfied, but also non-functional properties such as response time and throughput must be optimal.

We present RuGQoS’10 system for QoS-Aware Web services composition. The concept mapping technique allows to find the best possible composition w.r.t. some QoS parameters without performing a brute-force search.

Keywords—Web service composition; quality of service; concept mapping.

I. INTRODUCTION

With the introduction of Web services came the possibility of services composition [1]–[3], the process of combining basic services to satisfy complex tasks. Recently, the quality of service (QoS) values increasingly affect the satisfaction of a user with respect to a composition of Web services [4]. While the number of simple Web services that are used in a composition, or the actual composition structure is usually not important for the end users, the combined non-functional values, like response time or throughput, of a composed service directly affect the willingness of end users to use the composed service.

The Web Services Challenge [5], [6] is a yearly venue with the purpose of bringing together researchers active on Web services composition and offering a uniform benchmark for their proposals. Contestants are expected to create a composition system that is able to return a composition with the best possible QoS values, given large sets of synthetically generated service descriptions [7].

An entry to the challenge receives as an input three description files: (1) A Web Service Description Language (WSDL) file contains a set of services interface descriptions. The interface is defined by concepts that a service needs as an input, and concepts, that a service provides as an output. The number of services can vary greatly. (2) A Web Ontology Language (OWL) file contains a taxonomy of concepts, relating the classes of the WSDL files. A certain concept may be a specialized version of another concept, which means that it can be used as an input and as an answer to a query instead of a more general one. (3) A Web Service Service Level Agreements (WSLA) [8] file contains the QoS non-functional properties for each service in terms of response time and throughput.

One or more queries are sent to the system. Each query is a set of input concepts and output concepts that, in general, require a composition of services to be satisfied.

The output of a composition system to each query is a Business Process Execution Language for Web Services (BPEL) schema of two compositions that satisfy the query: the service composition with the lowest response time and the service composition with the highest throughput.

In this context, response time is expressed in time units and indicates the delay between the time a request is received by a Web service and the time a reply to the request is sent. The throughput measures the amount of requests that a web service can handle in a given time unit. When composing services the qualities of the individual services are aggregated in the following way: (i) the response time of all services in a sequence equals to the sum of all response times of those services; for services that run in parallel the maximum response time among them is taken; (ii) for throughput, the minimum value among all services is taken, whether they run in a sequence or in parallel.

II. THE RUGQoS SYSTEM

Given a set of services with corresponding QoS and OWL descriptions, our system RuGQoS, is able to satisfy queries of a user that requests a certain complex functionality, given in terms of input/output parameters. The answer of RuGQoS is the composition with the fastest response time and the one with the highest throughput, if such compositions exist.

The Quality of Service aware Web service composition system RuGQoS’10 is the evolution of the University of Groningen systems RuGCo’08 [9] and RuGQoS’09 [10], that participated in WS-Challenge in previous years.

With respect to RuGCo, RuGQoS’09 creates richer index structures to take into account the QoS descriptions and utilizes a breadth-first search composition algorithm. The RuGQoS’10 is further improved by devising a new composition algorithm with lower computational complexity than the previous one.

The architecture of RuGQoS’10 underwent no major changes, as the architecture proved to be robust and ade-
quate, thus we do not further describe it here but rather refer to [9], [10].

III. COMPOSITION ALGORITHM

The composition algorithm is the core of the RuGQoS system. We next describe the composition algorithm for finding the minimum response time. The algorithm for finding the maximum throughput is similar.

The main idea of our composition that radically improves performances in comparison to RuGQoS’09 is that one does not need to find all possible compositions in order to find the best response time.

The composition that gives the best response time for output concepts $C_o$, also gives the best response time for all intermediate concepts of the critical path that are calculated along the way. Ad absurdum, if this was not the case, and there existed a solution with a better response time for some concept along the critical path, one could use this solution instead, as it would mean having intermediate concepts with faster response times, thus a better overall solution.

Thus, for each concept one should only keep and update the currently found best time and the composition that gives such optimal response time. In fact, for each concept one only needs to know the last service of this composition (the service that has this concept as an output). All previous services in the composition one can find by looking recursively on the input concepts to the last service.

We create the concept map $CMP$ for this. The map has all concepts as keys. For each concept it keeps the currently found best time and the composition that gives the best time and the composition that gives the best response time.

We continue until we have concepts in a queue. With each run of a loop we process and remove one concept from the queue. When we get a concept from the queue, we know, that this concept has improved its response time with respect to the previously known information. Therefore, it is possible that concepts that are calculated after the current one in a solution also improve the response time. We check this on lines 3-20, for all services $inserv$ that have $concept$ as input do.

Now we can describe the main loop of the composition algorithm as shown in Algorithm 2.

Algorithm 2 Service composition

```java
1: while queue is not empty do
2:   concept ← poll(queue)
3:   for all services $inserv$ that have $concept$ as input do
4:     starttime ← 0
5:     for all $inconc$ ∈ $inserv$.input do
6:       $(t, s) ← CMP$($inconc$)
7:       starttime ← max($t$, starttime)
8:     end for
9:   end for
10:  starttime ← starttime + $inserv$.resptime
11:  for all $outconc$ ∈ $inserv$.output do
12:    $(tbest, s) ← CMP$($outconc$)
13:    if $endtime < tbest$ then
14:      $CMP$($outconc$) ← $(endtime, inserv)$
15:      for all $g ∈ generalizations(outconc)$ do
16:        $CMP$($g$) ← $(endtime, inserv)$
17:      end for
18:    end if
19:  end for
20: end for
21: end while
22: $(composition, besttime) ← ReturnSolution(CMP)$
```

We continue until we have concepts in a queue. With each run of a loop we process and remove one concept from the queue. When we get a concept from the queue, we know, that this concept has improved its response time with respect to the previously known information. Therefore, it is possible that concepts that are calculated after the current one in a solution also improve the response time. We check this on lines 3-20, for all services $inserv$ that have $concept$ as a part of input. The earliest possible starting time of the service $inserv$ is the maximum among response times of all its input concepts. We calculate this time on lines 4-9. If the $concept$ taken from the queue was the one with the maximum response time, the new possible starting time for the service will decrease. Note that in case that at least one of the input concepts is not yet satisfied, its response time (as well as the services’ starting time) will be equal to infinity.
The updated response time of the output concepts of service \( \text{inserv} \) equals to the service’s starting time plus its own response time. We can now check if any of the output concepts had its response time decreased. In this case, we add it and its generalizations to the queue (cf. lines 11-18).

When the queue is empty, we know the best response time of all concepts. We can now give an answer to the composition query. This is shown in Algorithm 3. We find the maximum response time among all concepts of the output set \( C_o \), and return this as a result to the query. If at least one response time is still set to infinity, the query is unsatisfiable.

Algorithm 3 Return solution

1: besttime \( \leftarrow 0 \)
2: for all concept \( \in C_o \) do
3: \((\text{time}, \text{service}) \leftarrow \text{CMP}(\text{concept})\)
4: if \( \text{time} = \text{inf} \) then
5: return Unsatisfiable
6: end if
7: besttime \( \leftarrow \max(\text{besttime}, \text{time})\)
8: Add service to composition
9: AddService(service)
10: end for
11: return \((\text{composition}, \text{besttime})\)
12:
13: function AddService(service)
14: if \( \text{composition} \) doesn’t contain service then
15: for all conc \( \in \text{service}.\text{input} \) do
16: \((t, \text{inserv}) \leftarrow \text{CMP}(\text{conc})\)
17: if \( \text{inserv} \neq \emptyset \) then
18: Add \( \text{inserv} \) to the composition as a predecessor of service
19: AddService(\text{inserv})
20: end if
21: end for
22: end if

To construct a service composition, we start by adding all services that are the last for each of the output concepts. Naturally, we add each service only once. Then for each such a service we check all of its input concepts. If an input concept has its own last service, we add it as a predecessor, and recursively process it in the same way. We do it until we meet a concept with no last service. These concepts are the ones from the input set \( C_i \).

IV. COMPOSITION EXAMPLE

We illustrate the proposed algorithms on a simple composition example. Table I presents a services taxonomy. Suppose the query provides concepts \( a, b \) and demands for the concept \( f \). Table II shows a run of the algorithm for this query.

Table I

<table>
<thead>
<tr>
<th>Services taxonomy</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
</tr>
<tr>
<td>in-concepts</td>
</tr>
<tr>
<td>out-concepts</td>
</tr>
<tr>
<td>response time</td>
</tr>
</tbody>
</table>

Table II

<table>
<thead>
<tr>
<th>CMP map</th>
<th>Current concept</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>((0,0), (0,0), (\text{inf}, 0), (\text{inf}, 0), (\text{inf}, 0), (\text{inf}, 0), (\text{inf}, 0))</td>
<td>( a,b )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>((0,0), (0,0), (3,3), (10,10), (\text{inf}, 0), (\text{inf}, 0), (\text{inf}, 0))</td>
<td>( a, \text{b,c,d} )</td>
</tr>
<tr>
<td>( 2 )</td>
<td>((0,0), (0,0), (3,3), (10,10), (\text{inf}, 0), (\text{inf}, 0), (\text{inf}, 0))</td>
<td>( b, e, d )</td>
</tr>
<tr>
<td>( 3 )</td>
<td>((0,0), (0,0), (3,3), (9,9), (7,7), (\text{inf}, 0), (\text{inf}, 0))</td>
<td>( c, \text{e,d} )</td>
</tr>
<tr>
<td>( 4 )</td>
<td>((0,0), (0,0), (3,3), (9,9), (7,7), (\text{inf}, 0), (\text{inf}, 0))</td>
<td>( d, \text{e,f} )</td>
</tr>
<tr>
<td>( 5 )</td>
<td>((0,0), (0,0), (3,3), (9,9), (7,7), (\text{inf}, 0), (\text{inf}, 0))</td>
<td>( e, \text{f} )</td>
</tr>
<tr>
<td>( 6 )</td>
<td>((0,0), (0,0), (3,3), (9,9), (7,7), (\text{inf}, 0), (\text{inf}, 0))</td>
<td>( f, \text{e,f} )</td>
</tr>
</tbody>
</table>

We start by assigning the time 0 to all provided concepts \((a, b)\) and putting them into the queue (step 0). All other concepts are unsatisfied, thus their time is set to infinite.

At step 1, we process the concept \( a \) extracted from the queue. Services \( s_1, s_4 \) use this concept as an input, and they both can be started directly (for \( s_1 \) the starting time of both its input concepts \( a \) and \( b \) is 0). So we update the possible starting time and service of concepts \( c \) and \( d \), and put them into the queue.

At step 2, we check the concept \( b \), but nothing can be improved. Service \( s_1 \) returns response time 3 for a concept \( c \), but we already have this value in a map, and service \( s_3 \) cannot be invoked, because its other input concept \( e \) is unsatisfied.

At step 3, we check the concept \( c \). By running service \( s_5 \) we can improve the starting time of the concept \( d \) from 10 to 9, thus we update the entry for this concept. We would also add \( d \) to a queue, if it is not already present. We can also satisfy the concept \( c \) now, by running service \( s_2 \). Its starting time \( 7 \) is equal to a sum of the starting time of concept \( c \) and the running time of service \( s_2 \).

At step 4, we finally satisfy the concept \( f \), by running service \( s_6 \) with response time 16. We do not stop however, because the queue is not yet empty. We check the concept \( c \) at step 5 and we see that it improves the response time of the concept \( f \) to 12. At the last step, we check the concept \( f \), however no service has it as an input, so nothing can be improved. The queue is empty now, so the algorithm has finished its work. The best response time of the concept \( f \) is 12, and the composition for this can be seen in Figure 1.

Figure 1. Example composition
V. Evaluation

To evaluate RuGQoS’10 and for testing purposes we generated 80 taxonomies with varying number of services, concepts and solution depths. Figures 2 and 3 indicate that effect of the number of services and the number of concepts on the complexity of the algorithm is very low. However, as illustrated by Figure 4, the composition time greatly increases with the increasing of the solution depth.

VI. Concluding Remarks

RuGQoS’10 is the University of Groningen 2010 entry to the Web services challenge and evolution of the 2008 and 2009 entries. In Table III, we compare the performance of the new system and we remark the radical improvement with respect to the previous year entry.

Acknowledgment

We want to acknowledge the work of Patrick Ratelband, Nico van Benthem, and Elie el Khoury on RugCo and RuGQoS’09 systems.

<table>
<thead>
<tr>
<th>Testset</th>
<th>Parameters</th>
<th>RuGQoS’09</th>
<th>RuGQoS’10</th>
</tr>
</thead>
<tbody>
<tr>
<td>set 1</td>
<td>Lowest response time</td>
<td>750</td>
<td>560</td>
</tr>
<tr>
<td>set 2</td>
<td>Highest Throughput (ms)</td>
<td>1500</td>
<td>15000</td>
</tr>
<tr>
<td>set 3</td>
<td>Highest Throughput (ms)</td>
<td>141</td>
<td>141</td>
</tr>
<tr>
<td>set 4</td>
<td>Highest Throughput (ms)</td>
<td>3627</td>
<td>141</td>
</tr>
<tr>
<td>set 5</td>
<td>Composition time (ms)</td>
<td>201</td>
<td>188</td>
</tr>
<tr>
<td>set 6</td>
<td>Composition time (ms)</td>
<td>/</td>
<td>141</td>
</tr>
<tr>
<td>set 7</td>
<td>Composition time (ms)</td>
<td>/</td>
<td>141</td>
</tr>
<tr>
<td>set 8</td>
<td>Composition time (ms)</td>
<td>/</td>
<td>141</td>
</tr>
<tr>
<td>set 9</td>
<td>Composition time (ms)</td>
<td>/</td>
<td>141</td>
</tr>
</tbody>
</table>

The research is supported by NWO Jacquard SaS-LeG contract no. 638.000.007N07. Project website: http://www.sas-leg.net.

References


