Concept mapping for faster QoS-Aware Web Service Composition

Viktoriya Degeler, Ilče 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.**

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.

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 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 bandwidth 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 adequate, 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 we 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 response time and the last service, that returns the concept as an output. The initialization of this map is shown in Algorithm 1.

Algorithm 1 Initialisation of CMP map and $queue$

```
1:  $queue \leftarrow \emptyset$
2:  for all $concept \in C$ do
3:      if $concept \in C_i$ then
4:          $CMP(concept) \leftarrow (0, \emptyset)$
5:          Add $concept$ to $queue$
6:      else if $concept \in generalizations(C_i)$ then
7:          $CMP(concept) \leftarrow (0, \emptyset)$
8:      else
9:          $CMP(concept) \leftarrow (\inf, \emptyset)$
10:     end if
11: end for
```

The best time for input concepts C_i and their generalizations is 0, and these concepts do not have an associated last service, since they are given as input. In other words, one does not need to execute anything in order to know their values. The best time for all other concepts is initially set to infinity. That means that at the current level of our knowledge those concepts cannot be satisfied. We also have a $queue$ of concepts, that "improved their response time, and should be processed". At the beginning the queue only contains input concepts C_i.

Before running the main composition loop, we create a map of all input services per concept. During the composition loop, it is required to know which services use the particular (or more general) concept as an input. Creating such a map at the beginning saves time instead of searching the needed services each time they are requested, thus improving the algorithm's performance. The map is used on line 4 in Algorithm 2.

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

Algorithm 2 Service composition

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

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

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.

Table I
Services taxonomy

	s1	s2	s3	s4	s5	s6
in-concepts	a,b	c	b,e	a	c	d
out-concepts	c	e	f	d	d	f
responce time	3	4	5	10	6	7

Table II
A run of the composition algorithm

Step	CMP map						Current concept	Queue
	a	b	c	d	e	f		
0	(0,∅)	(0,∅)	(inf,∅)	(inf,∅)	(inf,∅)	(inf,∅)	∅	a,b
1	(0,∅)	(0,∅)	(3,s1)	(10,s4)	(inf,∅)	(inf,∅)	a	b,c,d
2	(0,∅)	(0,∅)	(3,s1)	(10,s4)	(inf,∅)	(inf,∅)	b	c,d
3	(0,∅)	(0,∅)	(3,s1)	(9,s5)	(7,s2)	(inf,∅)	c	d,e
4	(0,∅)	(0,∅)	(3,s1)	(9,s5)	(7,s2)	(16,s6)	d	e,f
5	(0,∅)	(0,∅)	(3,s1)	(9,s5)	(7,s2)	(12,s3)	e	f
6	(0,∅)	(0,∅)	(3,s1)	(9,s5)	(7,s2)	(12,s3)	f	∅

At step 1, we process the concept a extracted form the queue. Services $s1$, $s4$ use this concept as an input, and they both can be started directly (for $s1$ 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 $s1$ returns response time 3 for a concept c, but we already have this value in a map, and service $s3$ cannot be invoked, because its other input concept e is unsatisfied.

At step 3, we check the concept c. By running service $s5$ 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 e now, by running service $s2$. Its starting time 7 is equal to a sum of the starting time of concept c and the running time of service $s2$.

At step 4, we finally satisfy the concept f, by running service $s6$ with response time 16. We do not stop however, because the queue is not yet empty. We check the concept e 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.

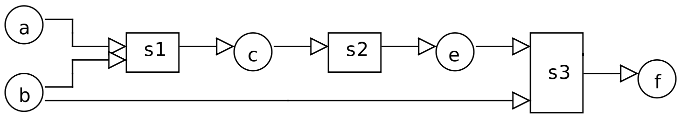

Figure 1. Example composition

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.

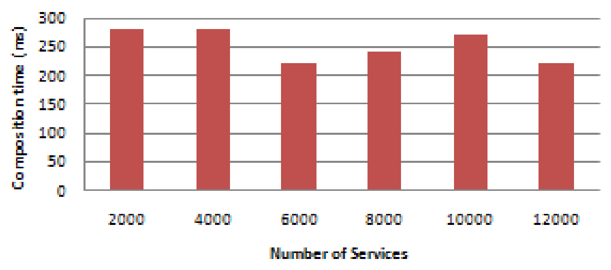

Figure 2. Average computation time vs number of services

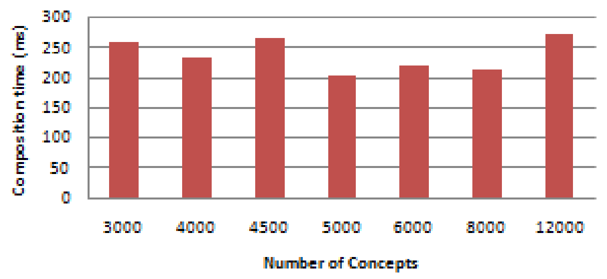

Figure 3. Average computation time vs number of concepts

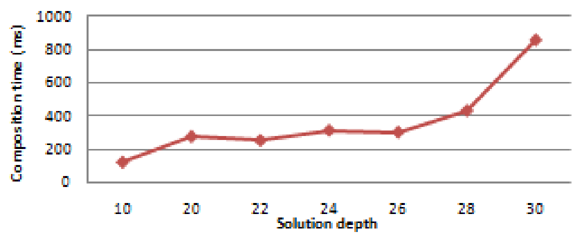

Figure 4. Average computation time vs depth of solution

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 on RuGQoS'09 system.

Table III
Compared performance of RuGQoS'09 and RuGQoS'10

Testset	Parameters	RuGQoS'09	RuGQoS'10
'09 Challenge set 1	Lowest response time	750	500
	Highest Throughput	15000	15000
	Composition time(ms)	202	88
'09 Challenge set 2	Lowest response time	2200	1690
	Highest Throughput	6000	6000
	Composition time(ms)	3627	141
'09 Challenge set 3	Lowest response time	810	760
	Highest Throughput	4000	4000
	Composition time(ms)	2601	188
'09 Challenge set 4	Lowest response time	/	1470
	Highest Throughput	/	4000
	Composition time(ms)	Out of time	266
'09 Challenge set 5	Lowest response time	5410	4070
	Highest Throughput	4000	4000
	Composition time(ms)	Out of time	519

The research is supported by NWO Jacquard **SaS-LeG** contract no. 638.000.000.07N07. Project website: http://www.sas-leg.net.

References

[1] F. Casati, M. Sayal, and M. Shan, "Developing e-services for composing e-services," in *Conf. on Advanced Information Systems Engineering (CAiSE)*, ser. LNCS 2068. Springer, 2001, pp. 171–186.

[2] A. Lazovik, M. Aiello, and M. Papazoglou, "Planning and monitoring the execution of web service requests," in *Int. Conf. on Service-Oriented Computing (ICSOC-03)*, ser. LNCS 2910. Springer, 2003, pp. 335–350.

[3] S. Tai, R. Khalaf, and T. Mikalsen, "Composition of coordinated Web services," *ACM/IFIP/USENIX Int. Conf. on Middleware*, vol. 78, no. 5, pp. 294–310, 2004.

[4] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut, "Quality of service for workflows and web service processes," *Journal of Web Semantics*, vol. 1, no. 2, pp. 281–308, 2004.

[5] (2010) The Web Services Challenge. [Online]. Available: http://www.wschallenge.org/

[6] M. Blake, W. Cheung, M. Jaeger, and A. Wombacher, "WSC-06: The Web Service Challenge," in *Joint Proceedings of the CEC/EEE 2006*, 2006.

[7] S. Bleul, T. Weise, and K. Geihs, "The web service challenge – a review on semantic web service composition," in *Service-Oriented Computing (SOC'2009)*, Mar. 5 2009.

[8] E. Keller and H. Ludwig, "The WSLA framework: Specifying and monitoring service level agreements for web services," *J. of Network and Systems Management*, vol. 11, p. 2003, 2003.

[9] M. Aiello, N. van Benthem, and E. el Khoury, "Visualizing compositions of services from large repositories," in *Joint Proceedings of the IEEE CEC/EEE 2008*, 2008, pp. 359–362.

[10] M. Aiello, E. el Khoury, A. Lazovik, and P. Ratelband, "Optimal qos-aware web service composition," in *Joint Proceedings of the IEEE CEC/EEE 2009*, 2009.