Optimal QoS-Aware Web Service Composition

Marco Aiello, Elie el Khoury, Alexander Lazovik, and Patrick Ratelband
Distributed Systems Group
Mathematics and Computing Science Department
University of Groningen, The Netherlands
Emails: m.aiello@rug.nl, e.el.khoury@rug.nl, a.lazovik@rug.nl, ratelband@gmail.com

Abstract

The availability of many independent services on an open network opens the opportunity of composing individual instances to achieve complex functionality. Most often there are several possible compositions to achieve the same high-level functionality; the advantage of choosing one composition instead of another one may lie in the different quality of the composition, e.g., one might be cheaper, faster, or more reliable. In this paper, we focus on services described with XML documents and accessed via XML Protocols, known as Web services, and enriched with semantic and Quality of Service (QoS) annotations. We propose an algorithm that, given a desired functionality, returns a composition of services from a repository with the optimal response time or throughput. Services are composed taking into account an ontology of operation names expressed in OWL. RuGQoS is the related implementation.

1. Introduction

The Web is evolving. From the original use as a vast distributed multimedia hyperlinked content sharing system, it is becoming more and more a computational infrastructure where not only content is shared, but also services. An important step in this shift is the availability of open standards, like those based on XML and collectively known as Web services [1]. Such publicly available services open a number of challenges and opportunities to be exploited: services need to be found, they need to be invoked, but they also can be composed together in order to achieve more complex functionality. The recent years have witnessed quite some effort on the last aspect, e.g. [2], [3], [4]. Here, we concentrate on a related problem, that is, the composition of services based on their advertised Quality of Service. The latter is a somewhat less investigated topic [5]. Given a set of services with corresponding QoS and OWL descriptions, the system, which we name *RuGQoS*, is able to satisfy queries of the user requesting 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 a compositions exists.

RuGQoS is the evolution of our earlier system RuGCo [6], the main difference is that instead of returning all possible compositions to the user, only the ones with best response time and throughput are returned. To manage such change, a modified version of the breadth first algorithm is developed and the indexing of WSLA description is added to the former RuGCo system. *RuGQoS* is the University of Groningen 2009 entry to the Web service challenge.

The remainder of the paper is organized as follows. Section 2 contains an overview of the 2009 Web-service challenge. In Section 3, we describe an entry for the challenge, namely, *RuGQoS*. It also contains preliminary considerations on how to evaluate the performance of a system such as *RuGQoS*. Final considerations are presented in Section 4.

2. Challenge Definition

The Web-Service Challenge focuses on the retrieval of service definitions and their composition, given large sets of synthetically generated descriptions [7]. In the last edition, entities appearing in the descriptions were related via an OWL ontology. The novelty of the fifth edition lies in the appearance of quality of service annotation coupled with service descriptions. Intuitively, a solution to the 2009 Web service challenge can be seen as a tool to find chain of services that optimally satisfy a user query.

First, let us consider all the elements given as input to an entry to the 2009 Web service challenge. (1) A Web Service Description Language (WSDL) file containing a set of services interface descriptions. The files are synthetically generated. The number of services can vary greatly. (2) A Web Ontology Language (OWL) file containing a taxonomy of concepts relating the classes of the WSDL files. (3) A Web Service Service Level Agreements (WSLA) [8] file containing the QoS non-functional properties for each service in terms of response time and throughput. (4) One or more queries to the system to be run against the repository (WSDL, WSLA) and the ontology (OWL). Each query is a set of input names and output names that, in general, require a composition of services to be satisfied.

The output of an entry to the challenge is a Business Process Execution Language for Web Services (BPEL) schema

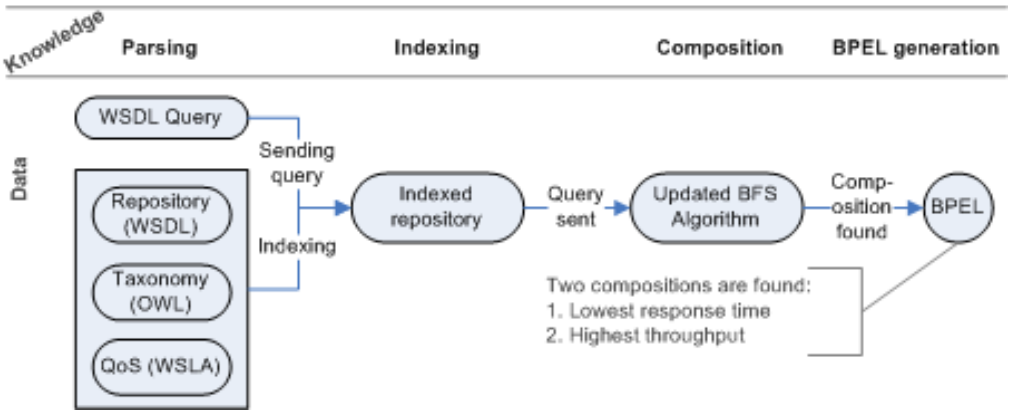

Figure 1. Data processing flow of RuGQoS.

of two compositions satisfying the user query: the service composition with the lowest response time and the service composition with the highest throughput. *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) for response time, we add the response time of all services in a sequence and we take the maximum of those in parallel; (ii) for throughput, we take the minimum value of all services in a sequence or in parallel. In the challenge, entries are evaluated mainly by finding the composition with the lowest response time and the composition with the highest throughput in the fastest time, without considering the time necessary to parse the service repository description, WSLA and ontology (Bootstrap phase).

3. The RuGQoS System

The Quality of Service aware Web service composition entry *RuGQoS* is the evolution of the University of Groningen entry to the WS-Challenge 2008, RuGCo [6]. The main components are: (i) a *XML parser* that translates the WSDL, WSLA and OWL descriptions into a set of indexes where each WSDL operation name is associated with a service and its response time and throughput quality; (ii) a *composition engine* based on a backward breadth first search algorithm; (iii) a *BPEL code generator* used to convert the compositions found by the engine into one generic business process according with new response time and throughput calculation rules. With respect to RuGCo, *RuGQoS* creates richer index structures to take into account the QoS descriptions and utilizes a different composition algorithm, as described in the next section.

From the data processing point of view, the flow of *RuGQoS* goes through four major phases, Figure 1. In the first phase, all the challenge data is parsed. The service information is used to create reverse indexes [9], that is, operation names are linked to services and their QoS. The OWL ontology is also used in this process. In fact, elements within a message structure are annotated, linking an element to a semantic individual, which is an instance of an OWL ontology class. Complex elements are dealt with by looking at their sub-elements. The ontology defines an hierarchy of classes. The creation of indexes occurs once for each data set. The other thing that is parsed is the user query. This can be done several times for the same data set. For each query the composition algorithm is invoked. This generates graphs representing the solution to the query (Composition phase). Finally, the solutions found are transformed into BPEL format, as required by the challenge rules.

3.1. The Composition Algorithm

The core of the *RuGQoS* system lies in its composition algorithm. This is based on an extended *Breadth First Search* (BFS) algorithm that uses a *priority queue* with cost based on response time and throughput. The priority queue contains unfinished compositions, and thus, cost and selection on partially completed compositions, with cost calculated according to two different cost functions (response time and throughput), we run the algorithm twice, and then combine the results into one unique BPEL schema. The proposed algorithm, shown in (Algorithm 1), always finds a solution if it exists and returns a failure otherwise.

The Algorithm 1 takes as input the input/output query, the cost function (different for response time and throughput) and the indexes from the service description repository (WSDL, OWL, WSLA processed data). The priority queue contains compositions, where the priority is calculated by the cost function f (line 3). Initially it contains one composition with a single node, with attached the input operation names i and initial cost 0. A priority queue q supports two operations *enqueue* and *dequeue*, denoted by $q \leftarrow \langle \star, \star \rangle$ and $\star \leftarrow q$. An

```
1: Input: Cost function f, input params i, desired output
   params g, indexed repository r
2: Output: Optimal composition or failure if it is not found
3: initialize priority queue q ← ⟨g, 0⟩
4: repeat
5:    c ← q
6:    if unsat(c) ⊆ i then
7:       return c
8:    else
9:       for all parameters e ∈ unsat(c) do
10:         if find(r, e) is empty then
11:            mark(r, q, e)
12:         else
13:            for all services s ∈ find(r, e) do
14:               q ← ⟨c+s, f(c+s)⟩
15:            end for
16:         end if
17:      end for
18:   end if
19: until q is empty
20: return failure
```

Algorithm 1: Composition algorithm.

operation **unsat** (line 6 and 9) returns a set of yet unsatisfied parameters for a given composition. An operation **find** (line 10 and 13) returns a set of services that satisfies (have as output) the given parameter. In line 14 operation **+** adds a service to a composition at a node where corresponding parameters are required. An operation **mark**(r, q, e) removes all services from the repository as well as all compositions in q that require e. Note that the algorithm always terminates as the number of parameters and services is finite, and at each step we resolve at least one parameter and we do not add services that are already part of the composition.

In the actual implementation of the algorithm, some further performance enhancements are present. E.g., several search steps are combined into one whenever possible by satisfying several required inputs at a time; one also keeps track of already used services and satisfied inputs to avoid redundant service selection; one never adds duplicate entries to the priority queue; moreover, whenever a single service satisfies several inputs in different branches, one can share it between all branches involved.

3.2. A run of the algorithm

Let us now consider a small example to illustrate the working of the Algorithm 1. Consider the set of six Web services listed in Table 1 with their respective inputs, outputs and cost (response time in this example). For simplicity, we use only simple types, omitting complex OWL taxonomy and class inferences. Now consider the user query taking no input and outputting a and b.

Initially (line 3 in Algorithm 1) the priority queue contains one composition with one node, and $\{a, b\}$ are the unsatisfied parameters (top left corner in Figure 2).

Step 1. In the first step, one finds satisfying either a or b, and then extend the composition. New services are added as nodes, and a link is added from the new service node to each of the nodes that new service resolves. Then, we remove satisfied parameters, and add new ones from the service input at the corresponding service node. In the example, we have three services (s_3, s_4, and s_5) that satisfy either a or b, and we add them all to the priority queue with their corresponding response times.

Step 2–8. At each following iteration, we take the first composition from the priority queue, and then for each yet unresolved parameter we add it back with all possible services that have this parameter as output, adding the updated response time. Note that there a number of special cases: at Step 3, we remove service s_2 from the repository and from the priority queue as it has an input parameter d which cannot be resolved by any other service and it is not part of the global input (lines 10–11 in Algorithm 1). At step 4 we find a first solution, with a total response time equal to 5. From this point on, the priority queue will not contain any composition that have response time equal or greater than 5. At step 7, the priority queue already contains compositions that are to be added and are dropped to avoid processing of identical compositions twice. At step 8, we find a solution with a total response time equal to 4, therefore we remove all entries from the priority queue that have cost equal or greater than 4. At the final step (lower left corner in Figure 2), one cannot build a composition that has a response time less than 4, thus, at the next step we extract the composition from the queue which has no unresolved parameters, that is, the solution (lines 6–7 in Algorithm 1).

	Input	Output	Response Time
s_1	-	C	2
s_2	D	C	1
s_3	C	A	1
s_4	-	B	4
s_5	E	B	2
s_6	-	C,E	3

Table 1. A Sample input to RuGQoS.

4. Concluding remarks

RuGQoS is the University of Groningen 2009 entry to the Web service challenge. It performs service composition using a generalized backward breadth first search with cost function based on two QoS parameters: response time and throughput. Future research will focus on the following aspects: (i) performance evaluation of the system on synthetically generated data; (ii) tuning of the composition algorithm to increase performance; (iii) combining into one search the identification of the optimal composition with respect to response time and throughput, rather than performing two

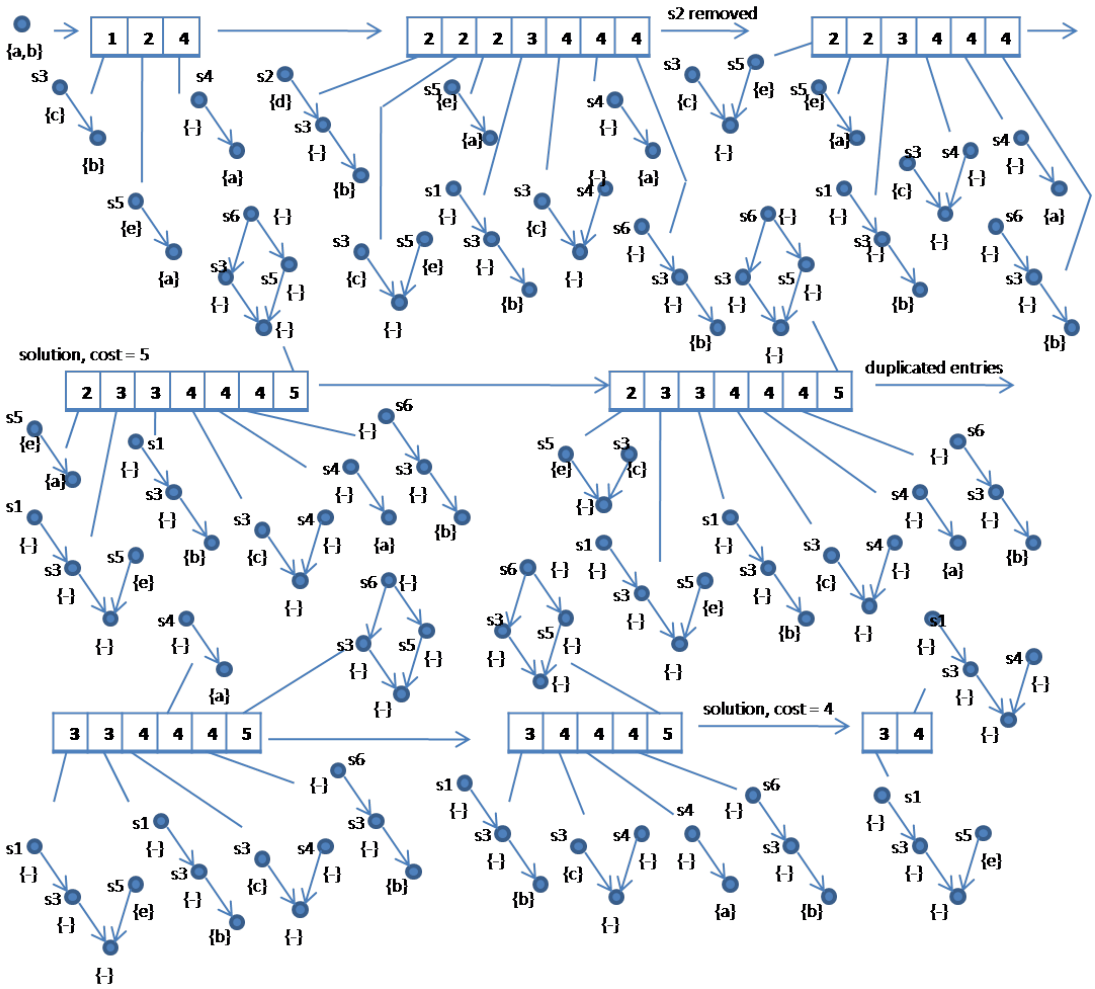

Figure 2. A run with input: {-} and output: {a,b}.

independent searches. Doing so will lower the execution time of the Algorithm, allowing results to be obtained faster, although memory usage might be an issue in this case. (iv) further improve the indexing algorithm to reduce the bootstrap time of the system.

References

[1] M. Papazoglou, *Web Services: Principles and Technology*. Prentice Hall, 2007.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[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, C. Platzer, F. Rosenberg, H. Tran, M. Vasko, and S. Dustdar, "Web service indexing for efficient retrieval and composition," in *IEEE EEE/CEC 2006*, 2006, pp. 63–65.