A Dynamic QoS-Aware Semantic Web Service Composition Algorithm

Pablo Rodriguez-Mier, Manuel Mucientes, and Manuel Lama

Centro de Investigación en Tecnoloxías da Información (CITIUS)
Universidade de Santiago de Compostela, Spain
{pablo.rodriguez.mier,manuel.mucientes,manuel.lama}@usc.es

Abstract. The aim of this work is to present a dynamic QoS-aware semantic web service composition algorithm that finds the minimal solution graph that satisfies the composition request considering multiple QoS criteria and semantic input-output message structure matching restrictions. Our proposal starts computing an initial solution by selecting only those services from the dataset that are relevant to the user request and meet the semantic restrictions. Then, an optimal QoS-aware composition is calculated using Dijkstra shortest path algorithm. Once the solution is obtained, the number of services is minimized using the optimal aggregated QoS value calculated in the previous step as a bound to prune the state space search. Moreover, a set of extensive experiments with five different datasets from the Web Service Challenge 2009-2010 is presented to prove the efficiency of our proposal.

Keywords: Automatic composition, Shortest Path, QoS optimization, Semantic Web Services.

1 Introduction

QoS-Aware web service composition has attracted a lot of attention from different fields in recent years. In [6], the authors distinguish two different types of composition algorithms: static and dynamic algorithms. Static algorithms require a predefined workflow with abstract processes. Each abstract process can be implemented by a wide variety of web services with different QoS measures that meet the functionality requirements of the process. The goal is to select the best services for each abstract process that fulfills the QoS constraints imposed by the user. Thus, these algorithms are only focused on service selection based on QoS and therefore are more related to the service discovery field. Relevant examples of this category are [10,9,2]. None of these approaches generate composite web services by combining different atomic services automatically. Dynamic algorithms, on the other hand, are more focused on calculating the overall composition structure, satisfying the global QoS. Within this category, the most interesting proposals are [5,8,1,3]. However, most of them can maximize only one QoS attribute and do not consider service minimization, leading to huge solutions with redundant services.

C. Liu et al. (Eds.): ICSOC 2012, LNCS 7636, pp. 623–630, 2012.

This paper addresses the problem of the dynamic QoS-Aware semantic web service composition considering multiple QoS attributes and minimizing the total number of services from the composition result. The novelties of our proposal are: 1) A multi-objective Dijkstra-based label setting algorithm that finds the optimal QoS composition (minimizing the total response time and maximizing the throughput) and 2) a combinatorial search algorithm that minimizes the number of services from a solution, keeping the optimal QoS. The algorithm uses the optimal values calculated in the previous phase to effectively reduce the search space size.

The rest of the paper is organized as follows: Sec. 2 introduces the basis of the semantic web service composition and explains the QoS model used to compute the global QoS. Sec. 3 illustrates the proposed algorithm for web service composition. Sec. 4 analyzes the algorithm with five different repositories and section 5 concludes the paper.

2 QoS-Based Semantic Composition Model

We define a web service by a 3-Tuple $S = \{I_S, O_S, Q_S\}$ where $I_S = \{I_S^1, I_S^2, ...\}$ is the set of inputs consumed by the service, $O_S = \{O_S^1, O_S^2, ...\}$ is the set of outputs retrieved when the service is invoked and $Q_S = \{Q_S^1, Q_S^2, ..., Q_S^n\}$ is the set of quality attributes of the service. Inputs and outputs of a service are semantically annotated by concepts that are defined in an ontology. Although concepts from an ontology can be related to each other by different types of relations in our approach we only use the subclass/superclass relationship, so we consider that an output of a service o_{S1} matches the input of other service i_{S2} when o_{S1} is equal or a subclass of i_{S2} ($o_{S1} \subseteq i_{S2}$).

A web service can be invoked only if all their inputs are matched. Given a request $R = \{I_R, O_R\}$, and given a web service $S = \{I_S, O_S, Q_S\}$, the web service S can be invoked only if $I_R \subseteq I_S$ (all inputs matched), i.e., for each input $i_s \in I_S$ there exists an input $i_r \in I_R$ such that $i_r \subseteq i_s$. Also, O_R will be satisfied only if $O_S \subseteq O_R$, i.e., for each output $o_r \in O_R$ there exists an output $o_s \in O_S$ such that o_s is equal or subclass of o_r ($o_s \subseteq o_r$).

2.1 QoS Computation Model for DAG Compositions

Considering the previous description, the QoS-aware composition problem tackled in this paper can be formulated as the automatic construction of a directed acyclic graph (DAG) that models the dependencies among the different web services involved in the composition with a global optimal value of QoS. The DAG contains two special nodes, *Source* (without incoming edges) and *Sink* (without outgoing edges), which provides the requested inputs and consumes the requested outputs respectively. Each directed edge is an ordered pair of two connected vertex (services) (S_i, S_j) of the graph and represents a semantic matching between S_i and S_j (i.e., one or many outputs from S_i match one or many inputs from S_j).

The calculation of the global value of QoS for a composite web service depends directly on the DAG structure. We consider the two quality QoS attributes defined in the Web Service Challenge 2009-2010: response time, which should be minimized, and throughput, which should be maximized. The total QoS value of a composite service corresponds with the aggregated QoS of the $Sink$ node of the composition DAG. To compute the best QoS of a composite service, we define a recursive function for each QoS attribute over the service domain ($QN_R(S)$, $QN_T(S)$):

- Resp. time: $QN_R(S_i, \{S_i^1, ..., S_i^n\}) = Max\{QN_R(S_i^1), ..., QN_R(S_i^n)\} + R(S_i)$
- Throughput: $QN_T(S_i, \{S_i^1, ..., S_i^n\}) = Min\{QN_T(S_i^1), ..., QN_T(S_i^n), T(S_i)\}$

Where $\{S_i^1, ..., S_i^n\}$ are the direct predecessors from the service node S_i and $R(S_i), T(S_i)$ are the functions that returns the response time and the throughput respectively associated to the service S_i. $QN_R(Sink)$ returns the total QoS of a composite service. Note that $R(Source), R(Sink) = 0$ and $T(Source), T(Sink) = \infty$ since $Source$ and $Sink$ are not real services.

3 Algorithm Description

The problem tackled in this paper consists of generating the best composition from the point of view of the QoS and cost (number of services) given a semantic request provided by an user. The steps followed by our proposal are: 1) Discover relevant services for the query; 2) Construct a matching digraph representing all possible matchings between these services; 3) Find the composition DAG with the optimal QoS value using a Dijkstra-shortest path algorithm over the matching digraph and 4) minimize the number of services of the solution using a backward search.

Finding the web service composition with the minimal cost has been proved to be NP-Complete [4]. However, in most cases the optimal QoS can be used as a bound to prune effectively the search space, discarding all those states that worsen the optimal value. In these section, we explain these steps in detail.

3.1 Service Filtering

The first step before calculating the composition is to filter all those services from the repository that are relevant to the request, discarding the rest. The filtering technique is explained in detail in [7]. Given a user request $R_{user} = \{I_R, O_R, W_R\}$ a matching digraph with the relevant services and all the matching relations among their inputs and outputs is generated layer by layer. Each layer contains those services whose required inputs are generated in previous layers. First and last layers contain the virtual services $Source = \{\emptyset, I_R, \{0, \infty\}\}$ and $Sink = \{O_R, \emptyset, \{0, \infty\}\}$, respectively, where $Source$ provides the inputs of the request and $Sink$ receives the outputs specified in the request. The calculation of the layers stops when there are no more services to add. When the process completes, the resultant graph contains all relevant services with their input/output concept matching relationships. The services contained in each layer are:

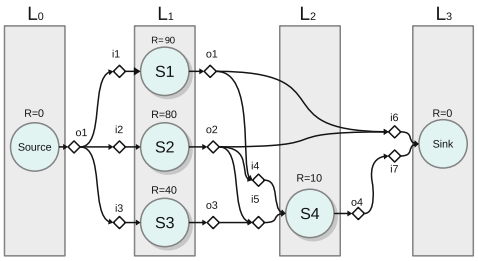

Fig. 1. A matching digraph representing the relations between the filtered (relevant) services for a request. Circles are services and diamonds are concepts (inputs and outputs). A directed edge between two concepts ($c1$, $c2$) represents a $c1 \subseteq c2$ relationship. Note that services from subsequent layers can provide inputs to services from previous layers, and therefore cycles are allowed.

- $L_0 = \{Source\}$, $L_N = \{Sink\}$
- $L_i = \{S_i : S_i \notin L_j (j < i) \wedge I_{Si} \cap O_{i-1} \neq \emptyset \wedge I_{Si} \subseteq I_R \cup O_0 \cup \ldots \cup O_{i-1}\}$

3.2 Optimal QoS-Aware Composition

The matching digraph represented in Fig. 1 has two type of nodes: services and concepts. Concepts are the traditional OR-nodes in a directed graph. Each incoming edge to a concept node represents a different path to obtain that concept. Thus, the optimal cost of a concept is determined by the best value among all their incoming paths. Conversely, services are a special type of nodes (AND-nodes) as they are unreachable until all their inputs are matched. The cost to reach a service node is calculated using the worst value among all their concepts. If a concept of a service has not been resolved (has a cost of ∞) then the cost to reach the service (and hence the cost of their outputs) is ∞ too. As our algorithm is multi-objective (minimizes response time and maximizes throughput) both QoS attributes have to be scaled and combined properly using the weights assigned to the request. The normalization of the QoS attributes is described in detail in [10] so is omitted here.

To find the optimal providers for each concept, we define a Dijkstra-based label setting algorithm that minimizes the objective function by exploring the service graph from the *Source* to the *Sink* node. The objective function of a composition is defined as $Global_{QoS}(R, T) = w1 * R + w2 * (1 - T)$, where R and T are the total response time and total throughput (scaled between [0,1]) of the composition and $w1, w2 \in [0, 1] \wedge w1 + w2 = 1$.

The pseudocode of the Dijkstra algorithm is shown in Alg. 1. The algorithm starts adding the *Source* service to the queue. Then, services in the queue are analyzed in order of increasing cost. The cost of each service S_i is their aggregated value of QoS. This value is calculated as $aggregatedQoS(S_i) = Global_{QoS}(QN_R(S_i, Pred), QN_T(S_i, Pred))$. $Pred = \{S_i^1, ..., S_i^n\}$ is the set of the optimal predecessors for each input of S_i, i.e., $Pred = \{i_1.op, ..., i_j.op\}$ ($i.op$

Algorithm 1. Optimal QoS-Aware Service Composition

```
#Services are ordered in queue by their aggregatedQoS(Service) value
queue ← (0, Source) #0 = best cost, 1 = worst cost
while queue ≠ ∅ do
    S_A ← queue #Extract lower cost service
    newAggregatedQoS = aggregatedQoS(S_A)
    for all S_B matched_by S_A do
        inputsMatched = {i_m : i_m ∈ (O_{S_A} ∩ I_{S_B}) ∧ i_m ∈ I_{S_B})}
        for all i_m ∈ inputsMatched do
            if newAggregatedQoS < i_m.aggregatedQoS then
                i_m.aggregatedQoS = newAggregatedQoS
                i_m.op = S_A #op means optimalPredecessor
            end if
        end for
        newCost = aggregatedQoS(S_B)
        queue ← (newCost, S_B) #(Re)order the neighbor in queue
    end for
end while
```

is the optimal predecessor that provides the input i) where $\{i_1, ..., i_j\} \in I_{S_i}$. If optimal predecessors have not been determined yet, then $aggreatedQoS(S_i) = \infty$.

3.3 Service Minimization

The reconstruction of the optimal QoS-Aware service composition using the optimal providers leads, in most cases, to inefficient compositions with redundant services, which increases the cost of the final composition. For example, following Alg. 1, we obtain that the best compositions contains the services $\{S2, S3, S4\}$ as they are the best providers for each input. However, as the best *aggregatedQoS* for $S4$ is determined by the worst cost (input $i4$), $S3$ can be removed without affecting the global value of QoS (input $i5$ can be provided by $S2$ with a cost of 80 ms). Thus, we develop a state space search algorithm that finds the composition with the minimum number of services using Dijkstra backwards (from *Sink* to *Source*), keeping the optimal QoS value calculated previously. The algorithm navigates state by state, selecting in each transition the best combination of services that provides the required inputs for each state, using the optimal QoS as a bound to discard all those actions that worsen the optimal value of QoS.

The search space is the set of all reachable states from the initial state by any sequence of actions. We define the minimization problem as a backward search over the state space. The elements that conform the search space problem are:

- *State*: is defined as a 2-tuple $\{I, O\}$ where $I = \{i_1, ..., i_n\}$ is the set of required inputs and $O = \{o_1, ..., o_n\}$ is the set of the provided outputs.
- *Initial state*: $\{I_{Sink}, \emptyset\}$ where I_{Sink} are the required inputs by the *Sink* node.
- *Goal state*: $\{\emptyset, O_{Source}\}$ where O_{Source} are the outputs provided by the *Source* service.

- *Action*: $A = \{S_1, ..., S_n\}$ is the set of services that provides the required outputs.
- $\phi(A)$: operator function that collects all outputs generated by an *Action*.
- $\gamma(A)$: operator function that collects all inputs required by an *Action*.
- *Transition function*: $f : State_A \times Action \rightarrow State_B$. The resulting state is defined as $State_B = \{\gamma(Action), \phi(Action)\}$.
- *Path cost function* $\delta(S)$: function that returns the size of the path $P = A_1 \cup A_2, ..., \cup A_N$ where P is the union set of all actions from initial state to S. Note that P contains all the different services selected from the initial state to S. The problem is to reach the goal state with the minimum cost.

Given a state $S = \{I, O\}$, the possible actions that can be applied to S are all those combinations of services from the matching digraph that covers the inputs $I \in S$, i.e., $\phi(A) \subseteq I$. Since we know the best aggregated QoS value of the composition, we can filter all those actions that exceed the bound. Consider the example in Fig. 1 and suppose that Alg. 1 determined that the best providers for all inputs are $(Source, S2, S3, S4, Sink)$. The global QoS of the composite service using these services is $QN_R(Sink) = Max(80, 90) = 90$. The initial state can be defined as $S_I = \{\{i6, i7\}, \emptyset\}$. The possible actions that can be applied to this state are $A_1 = \{S1, S4\}$ and $A_2 = \{S2, S4\}$. Although $S1$ is not considered by the algorithm as the best provider for $i6$, $S1$ can replace $S2$ without affecting the global QoS. The resulting states after applying actions A_1 and A_2 are $S_{A_1} = \{\{i1, i5\}, \{o1, o4\}\}$ and $S_{A_2} = \{\{i2\}, \{o2, o4\}\}$ Note that in the next iteration, S_{A_2} reach the solution with the minimum path cost, so the optimal solution consists only of services $S2$ and $S4$. Using Dijkstra to traverse the graph, we can guarantee the optimality of the solutions found.

4 Experiments

In order to prove the validity and efficiency of our algorithm in different situations, we carried out some experiments using five datasets from Web Service Challenge 2009-2010. Table 1 shows the results obtained for each dataset using different weights for response time (w1) and throughput (w2).

The minimization of the services for each solution can be done by searching over the entire service graph (global minimization, GM) or considering only the optimal providers obtained for each input (local minimization, LM). When the LM is performed, instead of considering all alternatives for each input, the algorithm prunes all those optimal redundant services from the original result that are not necessary to obtain the best aggregated value of QoS.

Column *#I. Serv* shows the initial services obtained before applying the minimization. These services are the optimal providers for each input found with the Alg.1. Column *#S. (LM/GM)* shows the minimum number of services obtained using local or global minimization. Columns *#Rt.(LM/GM)* and *#Th.(LM/GM)* present the results for the response time and the throughput of the composite service. Note that results obtained for response time when $w_1 = 0$ and for the

throughput when $w_2 = 0$ are not relevant, as the algorithm does not minimize/maximize the attributes weighted with 0. The last column shows the time elapsed (in milliseconds) between the initial user request and the delivery of the composition result (results are not translated to BPEL, they are provided as DAGs).

4.1 Results Discussion

Table 1 shows the results of the algorithm describing all the characteristics defined in the Web Service Challenge 2009-2010. All tests were executed in a Intel Core 2 Quad Q9550 2.83 GHz with 8 GB RAM, under Ubuntu 10.04 64-bit, with a time limit of 30 seconds for each test (results marked with a dash are those that took more than 30 seconds). The quality of the results is evaluated measuring the best response time, the best throughput and the number of services. Since we do not generate BPEL code, we cannot measure the total composition length.

An important difference between the solutions of the participants from the Web Service Challenge 2009-2010 and our solutions is that they do not minimize both quality attributes (they use the same algorithm to minimize each QoS attribute independently). Thus, their results should be compared with our solutions when $w_1 = 0, w_2 = 1$ or $w_1 = 1, w_2 = 0$, as they cannot provide intermediate solutions. In all cases we obtained the same best solutions as the winners, with less number of services for datasets 4 and 5. Note that the performance of our algorithm is slightly worse due to the minimization process. Solutions for the dataset 4 with the global service minimization cannot be obtained in a reasonable period of time due to the combinatorial explosion. However, local minimization can be used efficiently when the priority is to obtain good quality solutions in a short time.

Table 1. Results obtained by our algorithm

Dataset	**Optimal QoS solution**					
WSC-2009'01	w1/w2	#I. Serv.	#S. (LM/GM)	Rt.(LM/GM)	Th.(LM/GM)	Time (ms) (LM/GM)
	1.0/0.0	13	5/5	500/500	3000/3000	274/389
	0.5/0.5	7	5/5	760/760	15000/15000	277/291
	0.0/1.0	7	5/5	930/930	15000/15000	270/298
WSC-2009'02	1.0/0.0	25	20/20	1690/1690	3000/2000	868/1988
	0.5/0.5	24	20/20	1800/1770	6000/6000	860/3103
	0.0/1.0	24	20/20	1970/2000	6000/6000	117/7530
WSC-2009'03	1.0/0.0	11	10/10	760/760	2000/4000	1071/1545
	0.5/0.5	33	10/10	840/760	4000/4000	1069/1533
	0.0/1.0	31	18/11	1780/1110	4000/4000	1101/5249
WSC-2009'04	1.0/0.0	50	40/-	1470/-	2000/-	4399/-
	0.5/0.5	73	64/-	3540/-	4000/-	4586/-
	0.0/1.0	72	62/-	3840/-	4000/-	4506/-
WSC-2009'05	1.0/0.0	41	32/32	4070/4070	1000/1000	2646/2801
	0.5/0.5	41	32/32	4280/4200	4000/4000	2667/2680
	0.0/1.0	41	32/30	5470/4750	4000/4000	2657/10953

5 Conclusions

In this paper we have presented a dynamic QoS-Aware semantic web service composition that finds optimal compositions minimizing the total response time and maximizing the throughput. We also presented a method to effectively reduce the total number of services from a composition without affecting the global value of QoS. This technique can also perform a local or a global search to minimize the total services depending on time requirements. Moreover, a full validation has been done using five different datasets from the Web Service Challenge 2009-2010, showing a good performance as in all cases the best solutions with the best values of QoS and the minimum number of services were found.

Acknowledgement. This work was supported by the Spanish Ministry of Economy and Competitiveness (MEC) under grant TIN2011-22935. Pablo Rodríguez-Mier is supported by the Spanish Ministry of Education, under the FPU national plan. Manuel Mucientes is supported by the Ramón y Cajal program of the MEC.

References

1. Aiello, M., Khoury, E.E., Lazovik, A., Ratelband, P.: Optimal QoS-Aware Web Service Composition. In: IEEE CEC 2009, pp. 491–494 (2009)
2. Ardagna, D., Pernici, B.: Adaptive Service Composition in Flexible Processes. IEEE Trans. on Soft. Eng. 33(6), 369–384 (2007)
3. Jiang, W., Zhang, C., Huang, Z., Chen, M., Hu, S., Liu, Z.: QSynth: A Tool for QoS-aware Automatic Service Composition. In: IEEE ICWS 2010, pp. 42–49 (2010)
4. Oh, S.C., Lee, D., Kumara, S.R.T.: Effective Web Service Composition in Diverse and Large-Scale Service Networks. IEEE Trans. on Soft. Eng. 1(1), 15–32 (2008)
5. Oh, S.C., Lee, J.Y., Cheong, S.H., Lim, S.M., Kim, M.W., Lee, S.S., Park, J.B., Noh, S.D., Sohn, M.M.: WSPR*: Web-Service Planner Augmented with A* Algorithm. In: IEEE CEC 2009, pp. 515–518 (2008)
6. Rao, J., Su, X.: A Survey of Automated Web Service Composition Methods. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 43–54. Springer, Heidelberg (2005)
7. Rodriguez-Mier, P., Mucientes, M., Lama, M.: Automatic web service composition with a heuristic-based search algorithm. In: IEEE ICWS 2011, pp. 81–88 (2011)
8. Yan, Y., Xu, B., Gu, Z., Luo, S.: A QoS-Driven Approach for Semantic Service Composition. In: IEEE CEC 2009, pp. 523–526 (2009)
9. Yu, T., Lin, K.-J.: Service Selection Algorithms for Composing Complex Services with Multiple QoS Constraints. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 130–143. Springer, Heidelberg (2005)
10. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. IEEE Trans. on Soft. Eng. 30(5), 311–327 (2004)