QoS-Aware Multi-granularity Service Composition Based on Generalized Component Services

Quanwang Wu, Qingsheng Zhu, and Xing Jian

Computer College, Chongqing University, Chongqing, China
{wqw,qszhu,jx}@cqu.edu.cn

Abstract. QoS-aware service composition aims to maximize overall QoS values of the resulting composite service. Traditional methods only consider service instances that implement one abstract service in the composite service as candidates, and neglect those that fulfill multiple abstract services. To overcome this shortcoming, we present the concept of generalized component services to expand the selection scope to achieve a better solution. The problem of QoS-aware multi-granularity service composition is then formulated and how to discover candidates for each generalized component service is elaborated. A genetic algorithm based approach is proposed to optimize the resulting composite service instance. Empirical studies are performed at last.

1 Introduction

Service composition is staged at two phases: at first, the abstract composite service, consisting of a collection of abstract services orchestrated by kinds of workflow patterns, is defined, and then at running time, it is instantiated and executed by binding abstract services to concrete ones. Since many service instances could provide equivalent functionality with different Quality of Service (QoS) values, an efficient optimization approach for automatic service composition is required to optimize the overall QoS and meet global QoS constraints. This so-called QoS-aware service composition problem is a hot research topic and a lot of efforts have been devoted to it in recent years. Zeng *et al.* [1] use integer programming to find the optimal solution but the approach suffers from poor scalability due to its exponential computational complexity. Canfora *et al.* [2] present a genetic algorithm based approach to enhance the efficiency. The overall QoS reflected by the fitness value of the genome increases from generation to generation and the best one is returned as the solution. Other technologies are also applied to tackle this problem such as skyline query [3] and ant colony optimization [4].

However, current methods mostly lack flexibility of selection. That is, they only consider service instances that implement one abstract service in the composite service as candidates, and neglect those that fulfill multiple abstract services. To illustrate, consider a composite service consisting of three abstract services s_1, s_2 and s_3, which are executed in sequence. Assume there are service instances si_1, si_2 and si_3 which fulfill the functionality of services s_1, s_2 and s_3, respectively, and meanwhile there exists another service instance si_4, which implements the functionalities of s_1 and s_2 in sequence. Current composition approaches will limit candidates to si_1, si_2

S. Basu et al. (Eds.): ICSOC 2013, LNCS 8274, pp. 446–455, 2013.
© Springer-Verlag Berlin Heidelberg 2013

and si_3, but not consider si_4 even if its QoS is better than the aggregated QoS of si_1 and si_2, as the process definition of the composite service does not contain a single service that can accommodate si_4.

To the best of our knowledge, only a few works have tried to overcome this shortcoming. Barakat *et al.* [5] utilize the planning knowledge hierarchy to allow the expression of multiple decompositions of tasks, but how to construct the hierarchy among tasks automatically is not mentioned. Zhou *et al.* [6] present the problem of QoS-based multi-granularity service selection, and propose an integer programming based method. They only consider composite services orchestrated in the sequence pattern and do not explain how to discover candidates in various granularities. Feng *et al.* [7] study how to produce a new service composition plan with better QoS, while preserving its original behaviors, by replacing the service with another service or a set of services of finer or coarser grain.

In this paper, we present the concept of generalized component services (GCSs) to expand selection scope for service composition to achieve a better solution. The GCS is defined in a semantic manner, and the QoS-aware multi-granularity service composition model is formulated on the basis of this concept. In this model, any service instance which can fulfill partial functionality of the composite service with the same execution sequence can be discovered and employed for composition. A genetic algorithm based approach is presented to tackle this optimization problem and how the proposed approach outperforms the traditional one is described.

2 QoS-aware Multi-granularity Service Composition Model

2.1 Preliminaries

The functionality description of a semantic service can be denoted as a quadruple (I, O, P, E), such as in OWL-S[1], where:

(1) I and O are the inputs and outputs of the service. I and O consist of one or multiple parameter types and a parameter type is associated to a concept of a shared ontology. Two types C_1 and C_2 can either be equal ($C_1 \equiv C_2$), in a subclass relationship ($C_1 \sqsubseteq C_2$) or not related.

(2) P is the precondition which must hold before service execution and E is the effect which holds after service execution. P and E can be expressed by rule syntaxes such as SWRL[2].

Apart from the functional description, a service instance[3] owns a non-functional description: QoS. QoS attributes can be classified into two categories: positive and negative (denoted as Q^+ and Q^-). For the former, larger values indicate better performance (e.g. reliability and availability) while for the latter, smaller values indicate better performance (e.g. price and response time).

---

1 http://www.w3.org/Submission/OWL-S

2 http://www.w3.org/Submission/SWRL

3 In our discussion the term service refers to the abstract functionality and the term service candidate or instance refers to a concrete service provided to be consumed (e.g. web service).

Definition 1: Composite Service. A composite service is a value-added service, formed as a number of component services orchestrated according to a set of control-flow and data-flow dependencies.

From the view of process orchestration, a composite service can be represented as a directed acyclic graph (V_G, E_G), where, V_G is the set of vertices including services, gateways, the source and sink vertices, and E_G is the set of edges including control edges and data edges. Gateways encode the routing logic of control-flow dependencies. A split gateway has a single incoming control edge and multiple outgoing control edges, while a join gateway has multiple incoming control edges and a single outgoing control edge. We assume the process orchestration is structured, i.e., for each split gateway, there exists a corresponding join gateway merging the forked flows (e.g. XOR-join to XOR-split, AND-join to AND-split).

Control edges represent logical dependencies between services by specifying the order of interactions, and together with gateways, control edges determine the execution flow of the composite service. At the same time, data edges represent data dependencies between services and a data edge is a 3-tuple (s_{from}, s_{to}, C) meaning that the service s_{from} supplies the concept C to s_{to}. This supply relation holds, iff:

$$(\exists o \in s_{from}.O,\ o \sqsubseteq C) \wedge (\exists i \in s_{to}.I,\ C \sqsubseteq i)$$

To ensure uniqueness of traversal sequence on the vertices V_G in the following sections, let τ be a topological ordering of V_G, which should always be followed during traversing. Fig. 1 depicts the process orchestration of a composite service for illustration and the topological ordering τ can be as follows: $s_1 \oplus s_2 s_3 \oplus s_4 \otimes s_5 s_6 \otimes$ (the source and sink vertices are omitted here).

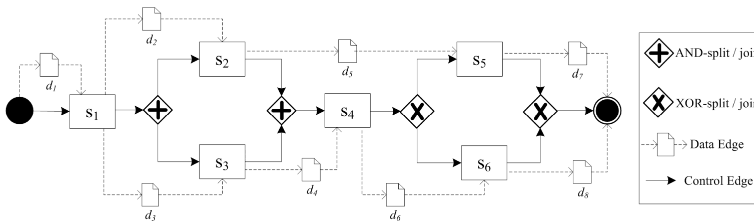

Fig. 1. Process orchestration of a composite service

Meanwhile, from the view of the functional description, a composite service can also be represented as (I, O, P, E) like a common service. Each element of the quadruple can be deduced from component services and the process orchestration. The data in data edges from the source vertex are the inputs, and the data in data edges to the sink vertex are the outputs. P can be deduced by aggregating the preconditions of the first-executed services in the composite service, and E can be deduced by aggregating the effects of the last-executed services. For a concrete composite service instance, it also has the non-functional attribute QoS and the QoS values are determined by QoS values of its concrete components and orchestration patterns. The detailed aggregation functions can be found in [2, 8, 9].

2.2 Granularity Model for Service Composition

Definition 2: Generalized Component Service (GCS). A generalized component service represents the functionality of a well-formed substructure in the composite service and in turn it can be used to compose this composite service as a component. The substructure can contain one or more services and it is well-formed if:

(1) all the services in it are connected via gateways and control edges;

(2) for each service, all its data edges are included;

(3) for any split gateway contained by the substructure, its corresponding join gateway is also included and vice versa; furthermore, all the vertices between them are included as well.

The first two requirements are intuitive and the reason to add the third one is to obviate GCSs that can not be used to compose the original composite service due to the violation of the control-flow dependencies. Take the substructure of s_1 and s_2 executed in sequence in Fig. 1 as an example: because of lack of s_3, which is also between the split & join gateways like s_2, this substructure is not well-formed.

GCS can also be expressed as (I, O, P, E), and each element can be deduced from the included services and the process orchestration. For example, if the source (sink) vertex is in the substructure, the data in data edges from the source (sink) vertex are the inputs (outputs). Otherwise, the data in data edges which have no starting (ending) vertices are the inputs (outputs).

For two GCSs from a specific composite service, if their included services are exactly the same, their functionality will be also completely the same according to the definition and requirements of GCSs. Thus, for a GCS, its set of services can be utilized as its identity and representation. Examples of GCSs in Fig. 1 are as follows: $gcs_1 = \{s_2, s_3\}$, $gcs_2 = \{s_1, s_2, s_3\}$, $gcs_3 = \{s_4, s_5, s_6\}$, $gcs_4 = \{s_1\}$. There are always many ways to decompose a composite service into multiple GCSs. For example, the composite service in Fig. 1 can be decomposed into gcs_2 and gcs_3, or into gcs_4, gcs_1 and gcs_3, and so on.

Definition 3: GCS Granularity. The granularity of a GCS is defined as the number of services it contains and it is denoted as $gra(GCS)$. For example, $gra(gcs_1) = 2$, $gra(gcs_2) = 3$, $gra(gcs_4) = 1$. A GCS is called fine-grained if its granularity is equal to 1, and otherwise it is called coarse-grained.

2.3 Problem Formulation

The target for QoS-aware service composition is to optimize overall QoS of the resulting composite service. The simple additive weighting (SAW) is adopted as the QoS utility function to facilitate ranking of composite service instances in terms of QoS. According to SAW, the QoS utility of a composite service instance csi_k can be calculated in Eq. 1, where, w_t is the preference weight and $q_t(csi_k)$ is the aggregated value of the tth QoS attribute of csi_k, and $q_{t,max}$, $q_{t,min}$ denote the minimal and maximal possible aggregated values of the tth QoS attribute, respectively.

$$U(csi_k) = \sum_{q_t \in Q^-} \frac{q_{t,\max} - q_t(csi_k)}{q_{t,\max} - q_{t,\min}}.w_t + \sum_{q_t \in Q^+} \frac{q_t(csi_k) - q_{t,\min}}{q_{t,\max} - q_{t,\min}}.w_t \quad (1)$$

Besides, users may impose global constraints on QoS attributes, e.g., the reliability should be larger than 95%. Hence, the QoS-aware multi-granularity service composition problem can be summarized as a two-step process:

1. When the user request for a specific composite service is received, the composition engine first identifies all the GCSs of the composite service, and then starts to discover instances for each GCS through the service registry using functional matching based on semantic descriptions;
2. With a number of service instances available for each GCS, the composition engine instantiate the composite service to a concrete one who is the optimal in terms of QoS utility and satisfies user's global QoS constraints.

In this context, the traditional QoS-aware service composition problem can be regarded as a special kind of our problem, where, granularity of GCSs is limited to 1.

3 Identification of GCSs and Discovery of Service Instances

In order to discover service instances in various granularities for the composite service *CS*, all its generalized component services should be first identified. Since in a GCS the set of services can be utilized as its indicator, an intuitive method is to enumerate all the combinations of services in *CS* and check whether requirements of GCSs are satisfied. However, the time complexity of this method is exponential, as the number of all the combinations is 2^n provided that the number of services is *n*.

```
Algorithm 1 constructGCS(CS,startId,endId,GCSSet)
for i=startId; i≤endId; i++ do
    v_1=CS.V_G.get(i,τ);
    if(isService(v_1)) then
        constructRest(CS,i+1,endId,GCSSet,v_1);
    else if(isSplit(v_1)) then
        nestDepth=0; branchStart=i+1; SComb.clear();
        for i=i+1; i≤ endId; i++ do
            v_2=CS.V_G.get(i,τ);
            if(isService(v_2)) then
                SComb.append(v_2);
                if(nestDepth==0 && pointToJoin(v_2)) then
                    constructGCS(CS,branchStart,i,GCSSet);
                    branchStart=i+1;
                end if
            else if(isSplit(v_2)) then
                ++nestDepth;
            else if(isJoin(v_2) && nestDepth--==0) then
                break;
            end if
        end for
        constructRest(CS,i+1,endId,GCSSet,SComb);
    end if
end for
```

Hence, we use the three requirements to construct GCSs, which is shown in Algorithm 1. *constructGCS* is a recursive function, *startId* and *endId* represent the indexes of the first and last vertex in *CS*, respectively, and *GCSSet* stores the constructed GCSs. The function first traverses vertices of *CS* from *startId* to *endId* successively following the topological ordering τ. If the vertex v_1 is a service, the function *constructRest* is invoked, which constructs GCSs whose initial part is fixed to v_1. If it is a split gateway, the traverse of vertices is continued in order to find each branch between this pair of split & join gateways. Inside this pair of gateways there may be nested with other split gateways, and thus *nestDepth* is used to measure the depth of nesting. It ascends when another split gateway is encountered and descends when the join gateway is encountered. A branch is determined when the outgoing control edge from the vertex v_2 points to a join gateway and *nestDepth* is equal to 0, and then the function recurs for each branch. When the corresponding join gateway to this split gateway is found, this inner traverse breaks. For all the services between this pair of split & join gateways, stored in *SComb*, *constructRest* is also invoked.

The function *constructRest(CS, startId, endId, GCSSet, SComb)* focuses on how to construct the rest part of a GCS when its initial part is fixed to the service combination *SComb*. Since the current *SComb* is a well-formed GCS itself, it is added into *GCSSet* first. Then vertex traverse is started from *startId* to *endId* successively, also following τ. When the vertex is a service, it is appended into *SComb*, and when the depth of nest is equal to 0 and the vertex is a join gateway or a service, *SComb* is added to *GCSSet* as a well-formed GCS.

After *GCSSet* is identified, the composition engine looks up for instances from the registry for each GCS in it. A service instance *si* is categorized as a candidate of a *gcs*, if its functionality exactly matches *gcs* with respect to logic-based equivalence of their formal semantics [10]. The matching in terms of inputs and outputs exploits defined semantics of the associated concepts as values of service parameters and the exact matching between *si* and *gcs* is formally expressed as:

$$\forall i_1 \in si.I,\ \exists i_2 \in gcs.I : i_1 \equiv i_2 \ \wedge\ \forall o_1 \in gcs.O,\ \exists o_2 \in si.O : o_1 \equiv o_2$$

The relaxed matching levels in terms of inputs and outputs such as subsuming and plugging in can be considered depending on the application requirement. Besides, the matching in terms of the precondition and effect can also be performed if necessary [11]. After service discovery, each GCS has a list of service candidates and $cnd(gcs_i)$ is used to denote all the discovered instances of gcs_i.

4 Genetic Algorithm for Optimizing Service Composition

Ahead of optimization, we first present the concept of generalized candidates to associate instances for GCSs in various granularities to services in the composite service.

Definition 4: Generalized candidates. Let $gs(s_i)$ denote the set of GCSs in *GCSSet*, whose first-traversed service is s_i. The generalized candidates $gcnd(s_i)$ of s_i represents the union set of $cnd(gcs_k)$ whose GCS gcs_k belongs to $gs(s_i)$. Formally, it is defined as:

$$gcnd(s_i) = \bigcup_{gcs_k \in gs(s_i)} cnd(gcs_k)$$

Therefore, $gcnd(s_i)$ contains candidates in varying granularities for s_i and the granularity of the candidate determines how many services from s_i in the composite service it can fulfill. Let $s_{i,j}$ represent the jth candidate in $gcnd(s_i)$. For example, if $gra(s_{i,1}) = 1$, it indicates $s_{i,1}$ can only fulfill the functionality of s_i, and if $gra(s_{i,1}) = 3$, $s_{i,1}$ can fulfill not only the functionality of s_i, but also that of s_{i+1} and s_{i+2}. Fig. 2 depicts services in Fig. 1 associated with their generalized candidates.

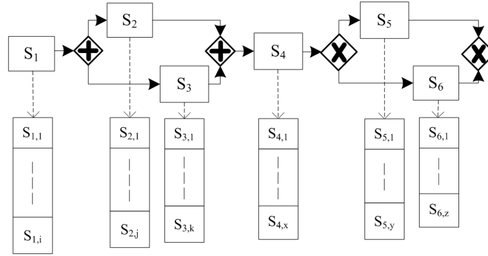

Fig. 2. Services and their generalized candidates

4.1 Genetic Encoding and Fitness Function

A concrete composite service instance is encoded as a genome for our problem. The genome is represented by an array with its length equal to the number of component services and the ith entry in the array refers to the selection result of the ith service. That is to say, given that the value of the ith entry is j, it indicates that $s_{i,j}$ is selected to execute s_i.

When a coarse-grained instance $s_{i,j}$ from $gcnd(s_i)$ with the granularity of k is selected for s_i, it can not only fulfill the functionality of s_i, but also fulfill the functionality of s_{i+1}, s_{i+2}, …, s_{i+k-1}. In this case, it is not necessary to select instances for those services, and the corresponding genes in the genome are filled with the pound sign "#" to indicate that these services have been implemented. Based on this representation rule, each service in the composite service is implemented by and only by one service instance in the composite service instance represented by a valid genome. Fig. 3 depicts an example of the genome. In the composite service instance represented by this genome, there are five service instances: $s_{1,3}$, $s_{2,9}$, $s_{4,2}$, $s_{5,5}$, $s_{6,4}$, where $s_{2,9}$ implements the tasks of s_2 and s_3, and $s_{6,4}$ implements the tasks of s_6, s_7 and s_8.

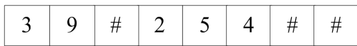

Fig. 3. An example of the genome

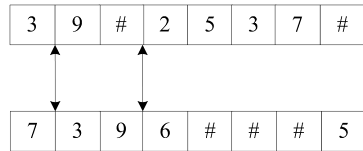

Fig. 4. An example of crossover

The fitness function measures the fitness of the represented solution. As clarified in Subsection 2.3, the fitness of a composite service instance csi_k relies on its QoS utility, and if QoS constraints are satisfied. Thus, it is defined as the sum of QoS utility

value and penalty for violations of QoS constraints. Eq. 2 is the fitness function, where *pnl* is negative, representing the penalty value for one violation, and x_t is a binary value defined in Eq. 3, denoting whether the tth QoS constraint qc_t is satisfied.

$$F(csi_k) = U(csi_k) + \sum_{t=1}^{|Q|} pnl \times x_t \tag{2}$$

$$x_t = \begin{cases} 1 & if\ q_t \in Q^+\ and\ q_t(csi_k) \le qc_t\ or\ \ q_t \in Q^-\ and\ q_t(csi_k) \ge qc_t \\ 0 & else \end{cases} \tag{3}$$

4.2 Genetic Operators

To guarantee that the representation rule of coarse-grained instances is always followed during the evolution of GA (i.e., to keep the genome valid), we extend each genetic operator with special adaptation.

Initialization Operator: An empty array with the length equal to the number of services is initialized and the random assignment is performed from the first gene to the last. An instance *c* from the generalized candidates $gcnd(s_1)$ of s_1 is randomly selected and bound to the first gene. If $gra(c) \ge 2$, the following $gra(c)$-1 genes are assigned with "#". Then the ith gene ($i = 1+gra(c)$) is selected to be assigned and this process loops until the last gene is assigned.

Crossover Operator: For a genome with a length of *n*, there are totally *n-1* splitting points. However, choosing some of them as splitting points will render the resulting genome invalid after crossover, and thus in a genome, the genes belonging to the same coarse-grained service instance should not be split. Let sp_1 be the set of feasible splitting points in $parent_1$, and sp_2 for $parent_2$. The splitting points the crossover operator can use are limited to the intersection of sp_1 and sp_2. For instance, in Fig. 4, sp_1 is {1, 3, 4, 5, 6}, sp_2 is {1, 2, 3, 7}, and thus feasible splitting points is {1, 3}.

Mutation Operator: Traditionally, each gene in the genome is selected and mutated with the same probability and in this case, coarse-grained service instances will be more likely to be replaced. Therefore, instead, a service instance is randomly selected with the same probability from all the service instances contained in the represented solution. The corresponding genes of the selected instance are then reassigned while complying with the representation rule.

4.3 Empirical Studies

A composite service with *n* abstract services is simulated and there are m_1/n service instances for each fine-grained GCS in it, m_2 instances totally for all coarse-grained GCSs of various granularities. Let λ represent the ratio that the number of course-grained candidates divided by that of fine-grained candidates, i.e., $\lambda = m_2/m_1$. The QWS dataset [12] is adopted to associate the service candidates. For a candidate with the granularity of *k*, *k* pieces of QoS data randomly selected from QWS dataset are first aggregated and then the aggregated datum is associated to the candidate.

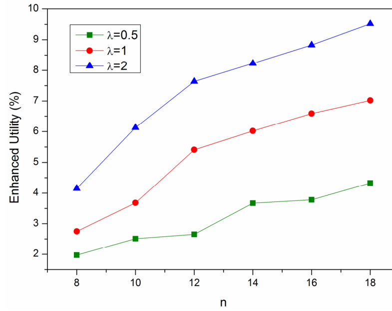

Fig. 5. Enhanced utility w.r.t. n

We evaluate the effectiveness of the proposed GA by comparing it with the traditional GA from [2], which only considers instances of fine-grained GCSs as candidates. The effectiveness is measured by using the enhanced percentage of the QoS utility value in the best solution and it is defined as $1 - u_{traditionalGA} / u_{GA}$. Figure 5 depicts values of enhanced utilities in three case of λ=0.5, 1 and 2, with n growing from 8 to 18 and m_1 set to 5*n. When λ becomes larger, i.e., the number of service instances for coarse-grained GCSs grows, the enhanced utility ascends. This value also goes up with the increase of n. Our approach outperforms the traditional one because the selection scope for service composition is expanded.

5 Conclusions

Traditional approaches for QoS-aware service composition lacks flexibility of selection, as only service instances which have corresponding functionality specified in the composite service via a single service are considered as candidates. This paper presents the concept of generalized component services to expand the choice space for QoS-aware service composition, and then proposes GA to solve the problem. The effectiveness is shown at last via empirical studies.

References

1. Zeng, L., et al.: QoS-aware middleware for Web Services Composition. IEEE Transactions on Software Engineering 30(5), 311–327 (2004)
2. Canfora, G., et al.: An approach for QoS-aware service composition based on genetic algorithms. In: Proceedings of GECCO 2005, pp. 1069–1075 (2005)
3. Alrifai, M., Skoutas, D., Risse, T.: Selecting skyline services for QoS-based web service composition. In: Proceedings of WWW 2010, pp. 11–20 (2010)
4. Wu, Q., Zhu, Q.: Transactional and QoS-aware dynamic service composition based on ant colony optimization. Future Generation Computer Systems 29(4), 1112–1119 (2013)
5. Barakat, L., Miles, S., Poernomo, I., Luck, M.: Efficient multi-granularity service composition. In: 2011 IEEE International Conference on Web Services, ICWS (2011)
6. Zhou, B., Yin, K., Jiang, H., Zhang, S., Kavs, A.J.: QoS-based selection of multi-granularity web services for the composition. Journal of Software 6(3), 366–373 (2011)

7. Feng, Z., et al.: QoS-aware and multi-granularity service composition. Information Systems Frontiers 15(4), 553–567 (2013)
8. Jaeger, M.C., et al.: Qos aggregation for web service composition using workflow patterns. In: International Enterprise Distributed Object Computing Conference, pp. 149–159 (2004)
9. Xia, Y., Luo, X., Li, J., Zhu, Q.: A Petri-Net-Based Approach to Reliability Determination of Ontology-Based Service Compositions. IEEE Transactions on Systems, Man, and Cybernetics: Systems 43(5), 1240–1247 (2013)
10. Klusch, M., Fries, B., Sycara, K.: OWLS-MX: A hybrid Semantic Web service matchmaker for OWL-S services. Web Semantics: Science, Services and Agents on the World Wide Web 7(2), 121–133 (2009)
11. Bartalos, P., Bieliková, M.: Qos aware semantic web service composition approach considering pre/postconditions. In: IEEE International Conference on Web Services (ICWS), pp. 345–352 (2010)
12. Al-Masri, E., Mahmoud, Q.H.: Investigating web services on the world wide web. In: Proceeding of WWW 2008, pp. 795–804 (2008)