Aggregate Quality of Service Computation for Composite Services

Marlon Dumas[1], Luciano García-Bañuelos[1], Artem Polyvyanyy[2], Yong Yang[3], and Liang Zhang[3]

[1] Institute of Computer Science, University of Tartu, Estonia
{marlon.dumas,luciano.garcia}@ut.ee

[2] Hasso Plattner Institute at the University of Potsdam, Germany
Artem.Polyvyanyy@hpi.uni-potsdam.de

[3] School of Computer Science, Fudan University, China
{081024011,lzhang}@fudan.edu.cn

Abstract. This paper addresses the problem of computing the aggregate QoS of a composite service given the QoS of the services participating in the composition. Previous solutions to this problem are restricted to composite services with well-structured orchestration models. Yet, in existing languages such as WS-BPEL and BPMN, orchestration models may be unstructured. This paper lifts this limitation by providing equations to compute the aggregate QoS for general types of irreducible unstructured regions in orchestration models. In conjunction with existing algorithms for decomposing business process models into single-entry-single-exit regions, these functions allow us to cover a larger set of orchestration models than existing QoS aggregation techniques.

1 Introduction

The ability to rapidly and effectively build new services by composing existing services – a practice known as *service composition* – is one of the key pillars of Service-Oriented Computing (SOC). Service orchestration is a popular approach for service composition [14]. The idea of service orchestration is to assign the responsibility for coordinating the execution of a composite service to a single entity (the *orchestrator*). The orchestrator is responsible for handling incoming requests for the composite service and to interact with the services participating in the composition (the *component services*) in order to fulfill these requests. The interactions between the orchestrator and the component services are governed by a *orchestration model* that usually takes the form of a process model in which each task represents either an internal action (e.g. a data transformation) or an interaction with a component service. In practice, these process models are specified using a specialized language such as the Business Process Execution Language (WS-BPEL) or the Business Process Modeling Notation (BPMN).

One of the key issues in service composition is that of predicting and managing the Quality-of-Service (QoS) of composite services. If we assume that each component service advertises its QoS, or that this QoS information can be derived based on past observations (as detailed in [18] for example), we can estimate the QoS of the composite service by aggregating the available information about the component services' QoS. This estimation can then be used to detect undesirable QoS variance as early as possible [19, 2] and to trigger corrective actions when such variance is detected [4].

In this setting, this paper addresses the following problem: How to compute the expected QoS of a composite service given its orchestration model specified in a language such as WS-BPEL or BPMN, and a binding that assigns each task in the orchestration to a concrete service? Following previous work, we assume that QoS is captured in terms of numerical attributes (e.g. time, cost and reputation) and that the QoS attribute values for each component service are given. Gathering QoS attribute values for non-composite services is a separate problem addressed in previous work [18].

Previous solutions to this problem [5, 8, 18, 9, 12] only work for composite services with well-structured orchestration models, that is, models described as graphs made up of split and join points, such that for every split there is a corresponding join such that the region of the graph between the split and the join is a single-entry-single-exit region. Yet, both WS-BPEL and BPMN allow orchestration models to be unstructured. In the case of WS-BPEL, one can obtain unstructured models by using so-called *control links*. These links allow tasks to be connected in arbitrary topologies, with the restriction that links cannot cross the boundaries of loop activities. Therefore, WS-BPEL orchestration models may contain unstructured acyclic fragments that cannot be handled by existing QoS aggregation methods. BPMN orchestration models are even less restricted, and they may contain both acyclic and cyclic unstructured fragments. The contribution of this paper is a generalized method for computing the QoS of composite services that can handle unstructured acyclic fragments, and a larger set of unstructured cyclic fragments than existing methods.

The rest of the paper is organized as follows. Section 2 introduces the orchestration model and the QoS model. Next, Section 3 describes the data structures used to represent service orchestrations, while Section 4 outlines the QoS aggregation method. Section 5 then discusses the implementation of the method and its evaluation using models of various sizes and topologies. Finally, Section 6 discusses related work and Section 7 draws conclusions.

2 Background

In this section, we introduce an orchestration model covering the core features of languages used in practice for specifying orchestration models, particularly WS-BPEL and BPMN. We also introduce the basic model for capturing Quality of Service that is used in the rest of the paper.

2.1 Orchestration Model

We consider service compositions whose internal logic is specified in terms of orchestration models. An orchestration model is essentially a process graph in which the tasks are mapped to interactions with the client of the composite service and with services drawn from a service repository (the component services).

Definition 1 (Composite Service, Orchestration Model).
A *composite service* is a tuple *(Orc, Binding)*, where *Orc* is a service orchestration model and *Binding* is a function that maps tasks in the orchestration model to component services or to a predefined *Client* role. An *orchestration model* is a directed graph consisting of edges (n_1, p, n_2) such that n_1 and n_2 are process nodes (the source and the target of the edge) and p is the probability of taking the edge assuming that the execution of the orchestration has reached node n_1.

Process nodes are of two types: *tasks* and *gateways*. Tasks represent units of work that are delegated to component services, while gateways represent control-flow routing points. There are two types of gateways: XOR gateways represent conditional branching (XOR-split) or merging of exclusive branches (XOR-join), wheres AND gateways represent parallel forking (AND-split) or synchronization points (AND-join). Split gateways are gateways with multiple outgoing edges, while join gateways are gateways with multiple incoming edges.

The binding of a composite service is not necessarily a total function – some tasks might not be bound to any service. A task in a composite service that is not bound to a service is called an *empty task*.

We impose the following *well-formedness* conditions: (i) an orchestration model has a single source node (i.e., a node with no incoming edges), and a single sink node (i.e., a node with no outgoing edges), and every node is on a path from the source to the sink; (ii) every task node has a single incoming and a single outgoing edge, and every gateway is either a split or a join. If these latter conditions are not satisfied, the orchestration model can be trivially restructured into one that satisfies these conditions; (iii) the sum of the probabilities attached to the outgoing edges of an XOR-split gateway is 1; (iv) an edge whose source is not an XOR-split gateway has a probability of 1, meaning that such edges are always traversed when their source node is reached.

As an illustrative example, we consider a simplified *Payment* composite service depicted in Fig.1. The figure shows the orchestration model of the composite service in BPMN. Tasks are represented as rounded rectangles while gateways are represented as diamonds labelled with 'X' (XOR) or '+' (AND). Not shown in the figure is the binding of the composite service which maps each task to a service (except tasks "Notify Customer" and "Reimburse Overpayment" which consist of interactions with the customer).

2.2 Quality of Service Model

QoS computations on composite services are performed with respect to a fixed set of QoS attributes $\{Attr_i \mid i \in 1..n\}$ such as execution time, cost and reliability.

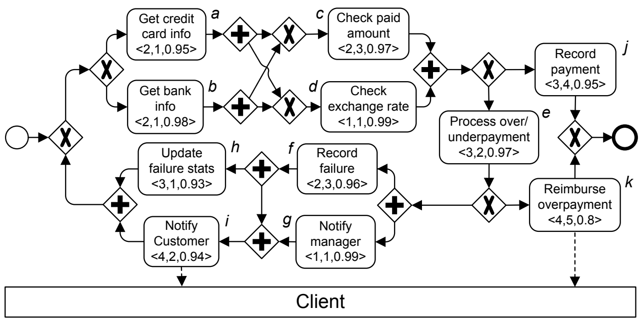

Get credit card info <2,1,0.95>
a
Get bank info <2,1,0.98>
b
Check paid amount <2,3,0.97>
c
Check exchange rate <1,1,0.99>
d
Process over/ underpayment <3,2,0.97>
e
Record payment <3,4,0.95>
j
Reimburse overpayment <4,5,0.8>
k
Update failure stats <3,1,0.93>
h
Record failure <2,3,0.96>
f
Notify Customer <4,2,0.94>
i
Notify manager <1,1,0.99>
g
Client

Fig. 1. Example of Composite Service

The assumption of a fixed set of attributes is made for presentation purposes and does not constitute a limitation since we can make this set as large as required.

We further postulate the existence of a function that given a service, returns its QoS. This function is initially given for pre-existing (non-composite) services. Our goal is to lift this function so that it can also be applied to composite services.

Definition 2 (QoS Function). The QoS of a service s, denoted by $QoS(s)$, is a vector $\langle v_1, \cdots, v_n \rangle$, where v_i is the value of QoS attribute $Attr_i$ for service s. By extension, QoS is also defined over tasks as follows: $QoS(T) = QoS(binding(T))$.

Numerous QoS attributes have been proposed in previous studies (e.g., [5–9, 18]). With respect to the method for computing QoS attribute values for composite services, we classify existing QoS attributes into three categories:

1. **Critical path** The value of the QoS attribute for the composite service is determined by the *critical path* of the orchestration. Examples include *execution time* (longest critical path) and fault-tolerance (weakest path) [5].
2. **Additive** The value of the QoS attribute for the composite service is a sum of the QoS values of the component services taking into account how often each service is invoked. Examples include *cost* and *carbon footprint.*
3. **Multiplicative** The QoS attribute value for the composite service is a product of the QoS values of the component services taking into account how often each service is invoked. Examples include *reliability* and *availability* [18].

Below, we only consider three representative attributes. For each service s, $QoS(s) = \langle T, C, R \rangle$, where T, C and R stand for time, cost and reliability. In Fig. 1, for example, the numbers in each service denote its QoS attributes.

3 Anatomizing Service Orchestration

This section presents an approach for parsing service compositions. Given a service orchestration, it gets decomposed into a collection of orchestration components, each with clear structural characteristics. The approach is founded on

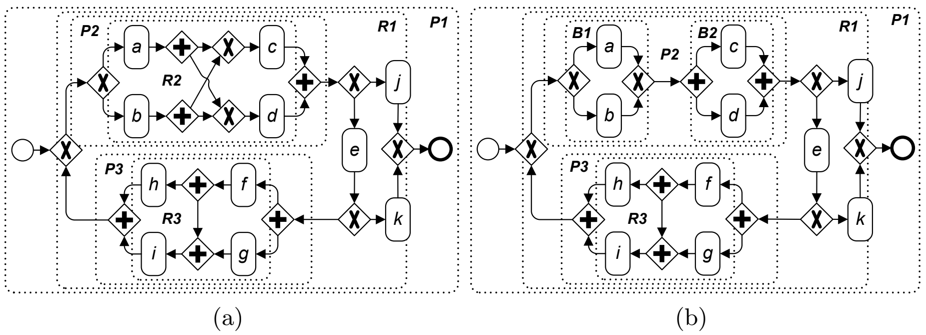


Fig. 2. (a) Service orchestration, (b) maximally-structured representation of (a)

two techniques: a technique for structuring orchestration models [15] and a technique for discovery of SESE components in orchestration models [16, 17]. Sect.3.1 presents the overall approach. Sect.3.2 and Sect.3.3 discuss two special types of orchestration components: SEMELoop and DAG components.

3.1 Orchestration Component

In order to analyze service orchestrations, we decompose them into *orchestration components*. An orchestration component is a subgraph of the orchestration model with a single-entry and single-exit point (including individual tasks in the orchestration, but also larger subgraphs). QoS is then computed independently for each orchestration component. This section discusses the approach we employ for identifying orchestration components in orchestrations.

The Refined Process Structure Tree (RPST) [16, 17] is a technique to parse orchestration models into a tree of SESE components. A component in the RPST contains all components at the lower level, whereas all components at a given level are disjoint. Each component in the RPST belongs to one out of four classes: A *trivial* (T) component consists of a single flow arc. A *polygon* (P) represents a sequence of components. A *bond* (B) stands for a set of components that share two common nodes. Any other component is a *rigid* (R) component.

Fig.2(a) exemplifies the RPST of the running example given in Fig.1. Note that Fig.2(a) uses short-names for tasks ($a, b, c, \ldots$), which appear next to each task in Fig.1. In the figure, each dotted box represents a component in the RPST that is formed by flow arcs that are inside or intersect the box. Names of components hint at their class, e.g., $P1$ is a polygon component and $R1$ is a rigid component. Each flow arc forms a trivial component. Trivial components, as well as polygons that are composed of two flow arcs, are not visualized for simplicity reasons.

Rigid components determine what makes a service orchestration unstructured. The service orchestration in Fig.2(a) contains three rigid components. To maximize the amount of structural information derived at the parsing step,

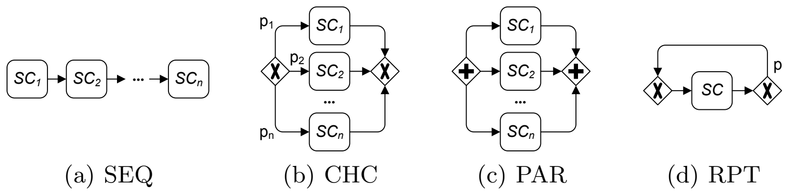


Fig. 3. Structured orchestration components

we employ the technique from [15]. The technique allows to derive maximally-structured representation of a service orchestration under the fully concurrent bisimulation equivalence notion [3]. By employing the technique from [15] to the model in Fig.2(a), one obtains a service orchestration that is given in Fig.2(b); rigid component $R2$ gets an equivalent representation that consists of bond components $B1$ and $B2$. Importantly, at the stage of maximally-structured representation we are able to define syntax of an orchestration component as follows.

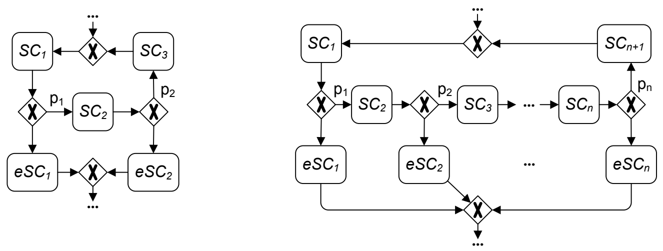
Definition 3 (Syntax of an Orchestration Component).
Let P be the range of real numbers from 0.0 to 1.0.

$$\begin{aligned}
ServiceComponent(SC) &::= \tau \mid Service \mid C^+ \mid C^- \\
OrchestrationElement(OE) &::= SC \mid AND \mid XOR \\
StructuredComponent(C^+) &::= SEQ([SC]) \mid CHC(\{P \times SC\}) \mid \\
&\quad \mid PAR(\{SC\}) \mid RPT(SC \times P) \\
UnstructuredComponent(C^-) &::= SEMELoop([SC \times P \times SC]) \mid \\
&\quad \mid DAG(\{OE \times P \times OE\})
\end{aligned}$$

We distinguish the following types of orchestration components: empty (τ) tasks, regular tasks (tasks bound to services), structured orchestration components, and unstructured orchestration components. Fig.3 exemplifies four types of structured orchestration components: sequence (SEQ), choice (CHC), parallel (PAR), and repeat (RPT). A sequence component is a list of orchestration components. A choice component is a set of orchestration components along with probabilities for executing each orchestration component. A parallel component is a set of orchestration components. Finally, a repeat component is an orchestration component along with the probability of repeating it. In Fig.2(b), bond $B1$ is a choice component, bond $B2$ is a parallel component, and polygon $P2 = [B1, B2]$ is a sequence component.

3.2 Single-Entry-Multi-Exit Loop Component

In the context of flowcharts, it has been shown that loops with multiple entry points can be restructured into loops with single entry point by means of node duplication [13]. However, subsequent transformation of loops with multiple exit points into loops with single exit point requires introduction of variables and



(a) 1-Entry-2-Exit Loop (b) 1-Entry-n-Exit Loop

Fig. 4. SEMELoop components

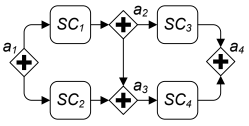
branching conditions on these variables. In these cases, it is not straightforward to abstract branching conditions as a branching probability. Therefore, we explicitly deal with single-entry-multi-exit loop components (SEMELoop components) that capture single entry point loop topologies. Fig.4(a) shows a 1-Entry-2-Exit loop component, whereas Fig.4(b) gives a general topology of a SEMELoop component of size $n \in \mathbb{N}, n \geq 2$.

We treat a SEMELoop component as a list of tuples (SC_i, p_i, eSC_i), where p_i is the probability of proceeding with loop execution after an accomplishment of an orchestration component SC_i and eSC_i is an orchestration component that is executed if the loop is left after an accomplishment of SC_i. For instance, a SEMELoop component that is given in Fig.4(a) is represented by the list $[(SC_1, p_1, eSC_1), (SC_2, p_2, eSC_2), (SC_3, 1.0, \tau)]$. Note that the last element in the list shows that the loop component cannot be left after SC_3 (the probability of staying in the loop is equal to 1.0) and, hence, no orchestration component can be executed after leaving the loop after SC_3 (denoted by a silent service τ). Observe that $R1$, both in Fig.2(a) and in Fig.2(b), is a 1-Entry-2-Exit loop component. Loop topologies that cannot be classified as SEMELoop components within service orchestrations are left for future work.

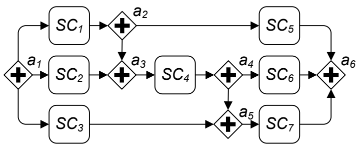
3.3 DAG Component

Acyclic rigids that are present in maximally-structured representations of service orchestrations are classified as irreducible acyclic components, or DAG components. Fig.5 exemplifies DAG components: Fig.5(a) shows the simplest DAG component—a well-known N-structure [10], whereas Fig.5(b) visualizes a DAG component that is a composition of N-structures. Observe that rigid $R3$, both in Fig.2(a) and in Fig.2(b), is a DAG component.

We treat a DAG component as a set of tuples (OE_1, p, OE_2), where OE_1 and OE_2 are orchestration elements, i.e., either an orchestration component or a gateway, and p is the probability that OE_2 will be executed after accomplishment of OE_1. For instance, the N-structure in Fig.5(a) is described by the set $\{(a_1, 1.0, SC_1), (a_1, 1.0, SC_2), (SC_1, 1.0, a_2), (SC_2, 1.0, a_3), (a_2, 1.0, a_3), (a_2, 1.0, SC_3), (a_3, 1.0, SC_4), (SC_3, 1.0, a_4), (SC_4, 1.0, a_4)\}$.



(a) N-structure



(b) Composition of N-structures

Fig. 5. DAG components

DAG components are analyzed by employing the notion of a *run*. A *run* is a subgraph of a DAG component that can be interpreted as its concurrent execution along with the probability to observe this run. The notion which is relevant to runs is that of *configuration*. A configuration is a set of tuples (OE_1, p, OE_2), where OE_1 is a XOR-split gateway, OE_2 is an orchestration element, and p is the probability that OE_2 will be executed after visiting OE_1. Moreover, a configuration must define a run. For instance, $\{(x_1, p_{11}, x_2), (x_2, p_{21}, SC_2)\}$ and $\{(x_1, p_{11}, x_2), (x_2, p_{22}, SC_3)\}$ are configurations of the DAG in Fig.6(a).

Individual runs allow us to treat each DAG component as a choice component. A choice component that corresponds to a DAG component is a set of runs of the DAG, each together with the probability to observe the run. We use Alg.1 to compute all configurations of a process graph.

Algorithm 1: Compute Configurations of a DAG Component

```
Input: $G$—a DAG component
Output: $\Theta$—the set of configurations of $G$
$X = \{x_1, x_2, \ldots, x_n\}$;
// XOR-split gateways of $G$
$\Theta = \prod_{i=1}^{|X|} out(x_i)$;
// Cartesian product of outgoing flow arcs
foreach $\theta = (e_1, e_2, \ldots, e_n) \in \Theta$ do
    foreach $e_i, e_j \in set(\theta)$ do
        if $\exists\, path : src(e_i), e_j \in path \wedge tgt(e_i) \notin path$ then $\theta = \theta - e_j$;
Remove duplicates from $\Theta$;
return $\Theta$;
```

Example 1. We exemplify the steps of Alg.1 for the DAG component in Fig.6(a):

1. $X = \{x_1, x_2, x_3\}$ is the set of XOR-split gateways, with outgoing flow arcs given as: $out(x_1) = \{e_1, e_4\}$, $out(x_2) = \{e_2, e_3\}$, $out(x_3) = \{e_5, e_6\}$.
2. $\Theta = \prod_{i=1}^{3} out(x_i) = \{(e_1, e_2, \underline{e_5}), (e_1, e_2, \underline{e_6}), (e_1, e_3, \underline{e_5}), (e_1, e_3, \underline{e_6}), (e_4, e_5, \underline{e_2}), (e_4, e_5, \underline{e_3}), (e_4, e_6, \underline{e_2}), (e_4, e_6, \underline{e_3})\}$, where the underlining elements will be removed in the next step.
3. $\Theta = \{(e_1, e_2), (e_1, e_2), (e_1, e_3), (e_1, e_3), (e_4, e_5), (e_4, e_5), (e_4, e_6), (e_4, e_6)\}$.
4. $\Theta = \{(e_1, e_2), (e_1, e_3), (e_4, e_5), (e_4, e_6)\}$, after removing duplicate entries.

Given a DAG component and the set of its configurations, Alg.2 computes the set of its runs. Consequently, one can construct a choice component that corresponds

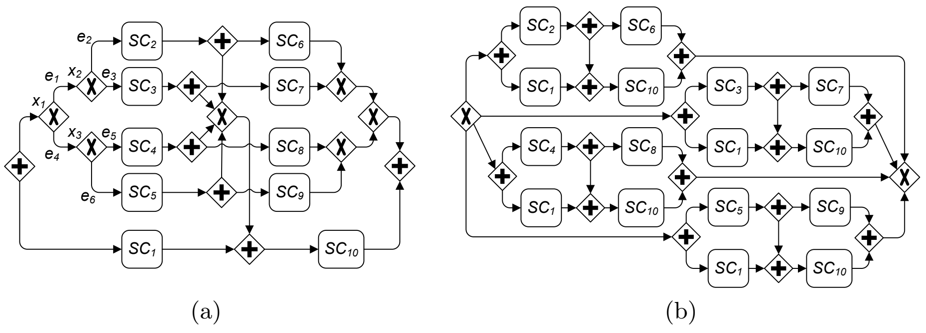


Fig. 6. (a) a DAG component, (b) a choice component that corresponds to (a)

to the original DAG component. A choice component is obtained by introducing a single XOR-split gateway which leads to entry of each run with the probability that reflects the chance to observe the run. Accordingly, exits of all runs must be merged by a single XOR-join gateway. A choice component that corresponds to the DAG component in Fig.6(a) is given in Fig.6(b).

Algorithm 2: Compute Runs of a DAG Component

```
Input: $G$—a DAG component, $\Theta$— the set of configurations of $G$
Output: $\Gamma$—the set of runs of $G$
$\Gamma = \{\}$;
// Initialize $\Gamma$ as empty set
foreach $\theta \in \Theta$ do Compute corresponding run for $\theta$
    $\gamma = G$;
    // Initialize the run
    $q = 1.0$;
    // Initialize the probability of observing the run
    foreach $(x, p, m) \in \theta$ do
        foreach $(x, h, n) \in \gamma \wedge n \neq m$ do $\gamma = \gamma - (x, h, n)$;
    while $\exists (y, k, z) \in \gamma : \nexists (y', k', y) \in \gamma$ do $\gamma = \gamma - (y, k, z)$;
    foreach XOR-split gateway $x \in \gamma$ do $\gamma = \gamma - (x, p, y) + (x.precede, 1.0, y)$;
    $q = q \times p$ // Update the probability of observing the run
    $\Gamma = \Gamma \cup (q, \gamma)$;
return $\Gamma$;
```

4 Quality of Service Aggregation

In this section, we discuss the aggregation of QoS of orchestration components. Sect.4.1 proposes functions for computing QoS of service orchestration components, and Sect.4.2 proposes an algorithm that combines these component-specific functions into comprehensive approach for computing QoS.

4.1 Aggregation of Orchestration Components

Structured Component The QoS of structured orchestration components is computed based on the following equations, which are taken from [5].

$$\begin{aligned} QoS(SEQ[SC_i]) &= \langle \sum T_{SC_i},\ \sum C_{SC_i},\ \prod R_{SC_i} \rangle \\ QoS(CHC\{(p_i, SC_i)\}) &= \langle \sum p_i T_{SC_i},\ \sum p_i C_{SC_i},\ \sum p_i R_{SC_i} \rangle \\ QoS(RPT(SC, p) &= \langle (1-p)^{-1} T_{SC},\ (1-p)^{-1} C_{SC},\ R_{SC}^{(1-p)^{-1}} \rangle \\ QoS(PAR\{SC_i\}) &= \langle max\{T_{SC_i}\},\ \sum C_{SC_i},\ \prod R_{SC_i} \rangle \end{aligned} \tag{1}$$

Note that, for repeat composition $RPT(SC, p)$, the computation considers that SC may be executed one or more times. Following the well-known power series relation, SC is expected to be executed $(1-p)^{-1}$ times[4], where p is the probability of staying in the loop.

Single-Entry-Multi-Exit Loop Component As for repeat components, in SEMELoop components a collection of orchestration components is executed one or more times. However, for a given component the expected number depends on its position in the loop.

Let $L = SEMELoop[(SC_i, p_i, eSC_i)], i \in [1..n+1]$, be a SEMELoop and $\{\alpha_i\} = \{SC_i\} \cup \{eSC_i\}$ be an orchestration component of L. Then, QoS of L can be computed as follows:

$$\langle \sum_{i=1}^{n+1} avg(\alpha_i) \bullet T_{\alpha_i}, \sum_{i=1}^{n+1} avg(\alpha_i) \bullet C_{\alpha_i}, \prod_{i=1}^{n+1} R_{SC_i}^{avg(SC_i)} \times \sum_{i=1}^{n} p(eSC_i) \bullet R_{eSC_i} \rangle \tag{2}$$

Here, $avg(\alpha_i)$ stands for the average number of times that α_i gets executed in the loop, $p(eSC_i)$ stands for the probability of exiting the loop along orchestration component eSC_i, and $avg(eSC_i) = p(eSC_i)$ for each service eSC_i.

For each service α_i, $avg(\alpha_i)$ can be computed as follows:

$$\begin{aligned} avg(SC_1) &= 1 + \rho_n + \rho_n^2 + \cdots = \sum_{i=0}^{\infty} \rho_n^i = (1-\rho_n)^{-1} \\ avg(SC_k) &= \rho_{k-1} + \rho_{k-1}\rho_n + \rho_{k-1}\rho_n^2 + \cdots = \rho_{k-1} \sum_{i=0}^{\infty} \rho_n^i = \rho_{k-1}(1-\rho_n)^{-1} \\ avg(eSC_k) &= p(eSC_k) = \rho_{k-1}(1-p_k) + \rho_{k-1}(1-p_k)\rho_n + \rho_{k-1}(1-p_k)\rho_n^2 + \cdots \\ &= (1-p_k)\rho_{k-1} \sum_{i=0}^{\infty} \rho_n^i = (1-p_k)\rho_{k-1}(1-\rho_n)^{-1} \end{aligned}$$

Wherein, $\rho_k = \prod_{i=1}^{k} p_i$, $k \in [1..n]$ and $\rho_0 = 1$.

[4] Power series: $p^0 + p^1 + ... + p^n = \sum_{i=0}^{\infty} p^i = (1-p)^{-1}$

Example 2. The average number of times of each orchestration component in Fig.4(a) is computed as follows:

$avg(SC_1) = \sum_{i=0}^{\infty}(p_1p_2)^i = (1-p_1p_2)^{-1}$
$avg(SC_2) = p_1\sum_{i=0}^{\infty}(p_1p_2)^i = p_1(1-p_1p_2)^{-1}$
$avg(SC_3) = p_1p_2\sum_{i=0}^{\infty}(p_1p_2)^i = p_1p_2(1-p_1p_2)^{-1}$
$p(eSC_1) = (1-p_1)\sum_{i=0}^{\infty}(p_1p_2)^i = (1-p_1)(1-p_1p_2)^{-1}$
$p(eSC_2) = p_1(1-p_2)\sum_{i=0}^{\infty}(p_1p_2)^i = p_1(1-p_2)(1-p_1p_2)^{-1}$

DAG Component A DAG orchestration component can be transformed into an equivalent choice component as explained in Sect.3.3. Each of the branches in this choice component corresponds to a run of the DAG component. The QoS values calculated for individual runs are then aggregated taking into account the probability of each run as follows:

$$\langle \sum p_{\gamma_k} T_{\gamma_k}, \sum p_{\gamma_k}\ C_{\gamma_k}, \sum p_{\gamma_k}\ R_{\gamma_k}\rangle \tag{3}$$

For a given run γ_k, the execution time can be computed with the well-known *critical path method*, i.e. compute the longest duration path in the run. Meanwhile, the cost of a run is the sum of the costs of the orchestration components in the run, and the reliability of a run is the product of the reliabilities of the orchestration components in the run.

For each run γ, therefore, QoS can be computed as follows:

$$\langle CriticalPath(T_{path_i}), \sum C_{SC_i}, \prod R_{SC_i}\rangle \tag{4}$$

Example 3. The QoS of the DAG component shown in Fig.5(a) is the following: $\langle max\{T_{SC_1} + max\{T_{SC_3}, T_{SC_4}\}, T_{SC_2} + T_{SC_4}\}, \sum C_{SC_i}, \prod R_{SC_i}\rangle$.

4.2 QoS of Composite Services

The overall QoS for a composite service is computed by aggregating the QoS of its orchestration components according to their structure and relation, i.e., maximally-structured representation in Fig.2(b). To this end, the RPST of the composite service is traversed in pre-order, i.e., computing the aggregate QoS from leaf nodes up to the root node.

Alg. 3 details the procedure of computing QoS for a service orchestration.

Example 4. To exemplify the Alg.3, we compute QoS for the composite service that is proposed in Fig.1.

$QoS(B1) = \langle 2\cdot 0.6 + 1\cdot 0.4, 1\cdot 0.6 + 2\cdot 0.4, 0.95\cdot 0.6 + 0.98\cdot 0.4\rangle = \langle 1.6, 1.4, 0.962\rangle$;
$QoS(B2) = \langle max\{2,1\}, 3+1, 0.97\cdot 0.99\rangle = \langle 2, 4, 0.9603\rangle$;
$QoS(P2) = \langle 1.6+2, 1.4+4, 0.962\cdot 0.9238\rangle = \langle 3.6, 5.4, 0.8887\rangle$;
$QoS(P3) = QoS(B2) = \langle max\{2+3, 2+4, 1+4\}, 3+1+1+2, 0.96\cdot 0.99\cdot 0.93\cdot 0.94\rangle = \langle 6, 7, 0.8308\rangle$;
$avg(P2) = 1/(1-0.8\cdot 0.6) = 1.9231$;

Algorithm 3: Compute QoS for Service Orchestration: ComputeQoS(SC)

```
Input: $SC$ — node of the RPST
Output: $QoS$ — QoS of $SC$
if $SC$ is an atomic service then return $\langle T_{SC}, C_{SC}, R_{SC}\rangle$;
foreach $SC_i \in$ ChildrenOf($SC$) do ComputeQoS($SC_i$);
if $SC$ is a structured orchestration component then
    Compute $QoS(SC)$ as according to Formula 1;
if $SC$ is a SEMELoop component then
    Compute $QoS(SC)$ according to Formula 2;
if $SC$ is a DAG component then
    Compute configurations $\Theta$ of $SC$ according to Alg. 1;
    foreach $\theta \in \Theta$ do
        Compute runs $\gamma$ for $\theta$ according to Alg. 2;
        Compute $QoS(\gamma)$ according to Formula 4;
    Compute $QoS(SC)$ according to Formula 3;
return $QoS$;
```

$avg(j) = p(j) = (1 - 0.8)/(1 - 0.8 \cdot 0.6) = 0.3846;$
$avg(e) = 0.8/(1 - 0.8 \cdot 0.6) = 1.5385;$
$avg(k) = p(k) = 0.8 \cdot (1 - 0.6)/(1 - 0.8 \cdot 0.6) = 0.6154;$
$avg(P3) = 0.6 \cdot 0.8/(1 - 0.8 \cdot 0.6) = 0.9231;$

$QoS(P1) = QoS(R1) = \langle 3.6 \cdot 1.9231 + 3 \cdot 0.3846 + 3 \cdot 1.5385 + 4 \cdot 0.6154 + 6 \cdot 0.9231, 5.4 \cdot 1.9231 + 4 \cdot 0.3846 + 2 \cdot 1.5385 + 5 \cdot 0.6154 + 7 \cdot 0.9231, 0.8887^{1.9231} \cdot 0.97^{1.5385} \cdot 0.80^{0.6154} \cdot (0.95 \cdot 0.3846 + 0.8308 \cdot 0.9231)\rangle = \langle 20.6927, 24.5388, 0.7506\rangle.$

5 Implementation and Evaluation

We have implemented the proposed QoS aggregation method in a tool that takes as inputs service orchestrations described in BPMN[5] and computes the aggregate value for each QoS attribute. The QoS values for each service and the branching probabilities of gateways in the BPMN model are defined in separate (text) files. The tool is distributed as an extension of the BPStruct tool and is available at: `http://sep.cs.ut.ee/Main/Bpstruct`. Below we present an evaluation of the scalability of the QoS aggregation method using the implemented tool.

5.1 Dataset

We collected a dataset consisting of 28 BPMN models from the following sources: 8 models from the public Oryx repository[6], 8 models from BPMN-to-BPEL case study of the Grabats'2009 graph transformation challenge[7], and 12 models from

[5] Specifically, the tool accepts BPMN models exported from Oryx (`http://oryx-project.org/`)
[6] `http://oryx-editor.org/`
[7] `http://fots.ua.ac.be/events/grabats2008/cases.html`

a repository of process models for local government authorities in China collected by Fudan University. We discarded incomplete/incorrect models, and models containing OR gateways, complex gateways, error events and boundary events, which are out of the scope of this paper. The size of models in the dataset (number of process nodes) range from 5 to 32, with an average of 17.5 nodes. Some of these models were larger, but they were structured into a top-level process with subprocess invocations. In this case, the process and its subprocesses are handled separately. The models cover all types of components: 72 SEQ components, 19 CHC components, 24 PAR components, 20 SESE Loop components, 2 DAG components, and 4 SEME Loop components. Links to all the models in the dataset are included in the tool distribution Web page. We assigned random probability to each XOR-split branch of each model (using a uniform distribution), and random QoS to each service (i.e., time, cost and reliability).

5.2 Results

We used the tool to compute the aggregate QoS for each models in the dataset and measured execution times (in milliseconds). All tests were performed on a laptop with a dual core Intel processor, 2.53 GHz, 3 GB memory, running Microsoft Vista and SUN Java Virtual Machine version 1.6 (with 512MB of allocated memory). To eliminate load time from the measures, each test was executed five times, and we recorded the average execution time of the second to fifth run. The measured execution time included the time required to compute the RPST and to calculate the QoS. The resulting histogram of execution times is plotted in Fig. 5.2. The figure shows that the QoS aggregation technique can deal with models of realistic size and complexity and that it scales quasi-linearly.

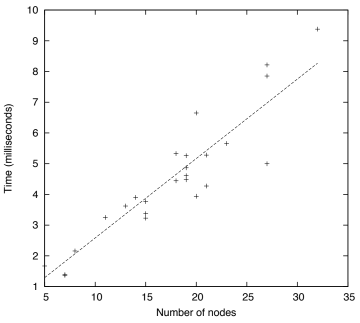


Fig. 7. Execution times for QoS aggregation

6 Related Work

Several previous studies have addressed the problem of aggregating QoS in terms of different structures in the orchestration model. Jaeger *et al* [8, 9] discuss the QoS aggregation problem for process models consisting of sequence, conditional and parallel. The approach does not deal with loops. In order to cope with the problem of binding and re-binding for composite services, Canfora *et al* [4] propose QoS aggregation functions for four constructs: sequence, switch, flow and loop, while Mukherjee *et al* [12] propose a model to estimate QoS of an executable BPEL process definition, but without considering unstructured BPEL activities (i.e. BPEL “flow” activities with control links). Cardoso *et al* [5] proposed a Stochastic Workflow Reduction (SWR) algorithm which takes as input a process graph and computes the expected QoS by repeatedly applying a set of reduction rules for sequential, parallel, conditional and simple loop. Hwang *et al* [6, 7] represent composite services using a tree structure and compute the aggregate QoS of composite services recursively by traversing the tree. This tree is similar with the RPST structure, but the trees in the work of Hwang *et al* do not contain any unstructured blocks. In summary, all of the above approaches are related to ours, but all of them deal with well-structured orchestration models only.

The problem of computing QoS for composite services is related to that of QoS-aware service composition [1, 11, 18]. The goal is to find a binding that optimizes a given objective function while satisfying a given set of constraints. The input is an orchestration model and a set of service candidates for each task in the orchestration model. Zeng *et al* [18] study a local and a global optimization approach to this problem using Simple Additive Weighting (SAW) and Integer Programming (IP), respectively. Meanwhile, Liu *et al* [11] propose a dynamic QoS computation model for web services selection in order to deal with runtime QoS selection. The authors construct a QoS matrix and compute QoS of a composite service via normalization and then multiplication with weights given by a user. A combination of local optimization and global optimization approaches is studied in Alrifai *et al* [1]. This latter work considers three types of QoS aggregation functions: summation, multiplication and minimum relation. Our classification of QoS attributes is inspired by this latter work.

The above studies address a more complex problem, in the sense that the binding is not given, but instead needs to be computed based on the set of candidate services for each task. On the other hand, the above work also suffer from an inability to deal with unstructured components. In addition, the global optimization approach proposed by Zeng *et al* [18] cannot deal with loops (not even structured loops). Instead, it is assumed that loops are expanded by putting an upper-bound to the number of times a loop is executed and unfolding the loop into a sequential structure.

7 Conclusion

In this paper, we proposed a method for computing the QoS of composite services. Unlike previous work, the proposed method can deal with orchestration models containing unstructured components, specifically models containing single-entry-multi-exit loop (SEMELoop) and DAG components. The proposed method has been implemented as a tool and tested with a collection of models taken from multiple sources.

Our future work includes computing QoS for composite services with more complex types of loops (e.g., overlapping loops) and extending the proposed method to address the problem of QoS-aware service composition.

Acknowledgment. Work supported by the NSF of China grant 60873115, the Chinese National Basic Research Program (973) grant 2005CB321905, the Fudan Short-Term Visits Program for Doctoral Students, the Estonian Science Foundation and ERDF via the Estonian Centre of Excellence in Computer Science.

References

1. Alrifai, M., Risse, T.: Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition. In: Proc. 18th Int. Conf. on World Wide Web. pp. 881–890. ACM (2009)
2. Becker, C., Kulovits, H., Kraxner, M., Gottardi, R., Rauber, A.: An Extensible Monitoring Framework for Measuring and Evaluating Tool Performance in a Service-Oriented Architecture. In: Proc. 9th Int. Conf. on Web Engineering. pp. 221–235. Springer (2009)
3. Best, E., Devillers, R.R., Kiehn, A., Pomello, L.: Concurrent Bisimulations in Petri Nets. Acta Informatica 28(3), 231–264 (1991)
4. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: A framework for QoS-aware binding and re-binding of composite web services. Systems and Software 81(10) (2008)
5. Cardoso, J., Sheth, A., Miller, J., Arnold, J., Kochut, K.: Quality of Service for Workflows and Web Service Processes. Web Semantics 1(3), 281–308 (2004)
6. Hwang, S.Y., Wang, H., Srivastava, J., Paul, R.A.: A Probabilistic QoS Model and Computation Framework for Web Services-Based Workflows. In: Proc. 23rd Int. Conf. on Conceptual Modeling. pp. 596–609. Springer (2004)
7. Hwang, S.Y., Wang, H., Tang, J., Srivastava, J.: A probabilistic approach to modeling and estimating the QoS of web-services-based workflows. Information Sciences 177(23), 5484–5503 (2007)
8. Jaeger, M.C., Rojec-Goldmann, G., Muhl, G.: QoS Aggregation for Web Service Composition using Workflow Patterns. In: Proc. of Enterprise Distributed Object Computing Conf. pp. 149–159. IEEE (2004)
9. Jaeger, M.C., Rojec-Goldmann, G., Muhl, G.: QoS Aggregation in Web Service Compositions. In: Proc. IEEE Int. Conf. EEE. pp. 181–185 (2005)
10. Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.: On Structured Workflow Modelling. In: Proc. 12th Int. Conf. on Advanced Information Systems Engineering. pp. 431–445. Springer (2000)
11. Liu, Y., Ngu, A.H., Zeng, L.Z.: QoS Computation and Policing in Dynamic Web Service Selection. In: WWW Alt. pp. 66–73 (2004)
12. Mukherjee, D., Jalote, P., Gowri Nanda, M.: Determining QoS of WS-BPEL compositions. In: Proc. 5th Int. Conf. on Service-Oriented Computing. pp. 378–393. Springer (2008)
13. Oulsnam, G.: Unravelling Unstructured Programs. Computer Journal 25(3) (1982)
14. Peltz, C.: Web services orchestration and choreography. IEEE Computer 36(10), 46–52 (2003)
15. Polyvyanyy, A., García-Bañuelos, L., Dumas, M.: Structuring Acyclic Process Models. In: Proc. Int. Conf. on Business Process Management (2010), (in press)
16. Polyvyanyy, A., Vanhatalo, J., Völzer, H.: Simplified Computation and Generalization of the Refined Process Structure Tree. In: Proc. of the 7th Int. Workshop on Web Services and Formal Methods (WS-FM) (2010), (in press)
17. Vanhatalo, J., Völzer, H., Koehler, J.: The Refined Process Structure Tree. Data and Knowledge Engineering 68(9), 793–818 (2009)
18. Zeng, L., Benatallah, B., H.H. Ngu, A., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. IEEE Transactions on Software Engineering 30(5), 311–327 (2004)
19. Zeng, L., Lei, H., Chang, H.: Monitoring the QoS for Web Services. In: Proc. 4th Int. Conf. on Service-Oriented Computing. pp. 132–144. Springer (2007)