Generalized Aggregate Quality of Service Computation for Composite Services

Yong Yang[a], Marlon Dumas[b], Luciano García-Bañuelos[b], Artem Polyvyanyy[c], Liang Zhang[a]

[a] *School of Computer Science, Fudan University, Handan 220, Shanghai 200433, China*
[b] *Institute of Computer Science, University of Tartu, J. Liivi 2, Tartu 50409, Estonia*
[c] *Hasso-Plattner-Institute, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany*

Abstract

This article addresses the problem of estimating the Quality of Service (QoS) of a composite service given the QoS of the services participating in the composition. Previous solutions to this problem impose restrictions on the topology of the orchestration models, limiting their applicability to well-structured orchestration models for example. This article lifts these restrictions by proposing a method for aggregate QoS computation that deals with more general types of unstructured orchestration models. The applicability and scalability of the proposed method are validated using a collection of models from industrial practice.

Keywords: Service-Oriented Computing, Service Composition, Service Orchestration, Quality of Service

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

Email addresses: `081024011@fudan.edu.cn` (Yong Yang), `marlon.dumas@ut.ee` (Marlon Dumas), `luciano.garcia@ut.ee` (Luciano García-Bañuelos), `artem.polyvyanyy@hpi.uni-potsdam.de` (Artem Polyvyanyy), `lzhang@fudan.edu.cn` (Liang Zhang)

pillars of Service-Oriented Computing (SOC). Service orchestration is a popular approach for service composition [1]. 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 an *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 Model and Notation (BPMN).

One of the key issues in service composition is that of predicting and managing the Quality-of-Service (QoS) of composite services. In particular, providers of composite services need to assess the expected quality of these services and to detect and act upon unexpected QoS variations [2, 3, 4].

In this context, this article addresses the problem of computing the expected (mean) QoS of a composite service given:

- Its orchestration model specified in a language such as WS-BPEL or BPMN.
- A *binding* that assigns each task in the orchestration model to a service. Services bound to at least one task in an orchestration model are called *component services.*
- The mean QoS of each component service.

In line with previous work, we assume that QoS is captured in terms of numerical attributes (e.g., time, cost and reputation). It is also assumed that the QoS attribute values of component services are either disclosed by the providers of these services (as part of their Service Level Agreements) or derived by service consumers or by third parties based on past observations (cf., [5] for example).

Previous solutions to the above QoS aggregation problem [6, 7, 8, 9, 10] impose strong restrictions on the topology of the input models. For the most, these solutions are designed for well-structured orchestration models, that is, models described as graphs, such that for every node with multiple outgoing arcs (a *split* node) there is a unique corresponding node with multiple incoming arcs (a *join* node) such that the region of the graph between the split and the join is a single-entry-single-exit (SESE) region.[1] Yet, mainstream languages for defining orchestration models, such as WS-BPEL and BPMN, allow orchestration models to be unstructured. Mukherjee *et al.* [9] outlined an approach to partially lift the well-structuredness restriction in order to cover the case of orchestration models containing acyclic SESE regions, while Zheng *et al.* [10] proposed an algorithm for dealing with orchestration models containing some types of unstructured cycles. However, these methods still impose restrictions on the input models that hinder their applicability in practical settings.

The contribution of this article is a generalized method for QoS aggregation that lifts the restrictions imposed by existing methods. The proposed method has been implemented and tested on a collection of process models taken from industrial practice, including models with topologies that cannot be handled by existing QoS aggregation methods.

The rest of the article is organized as follows. Section 2 introduces a running example and defines the notions of orchestration model and QoS model used in this article. Next, Section 3 describes the data structures used to represent orchestrations, while Section 4 outlines the QoS aggregation method. Section 5 then discusses the implementation of the method and its empirical evaluation. 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 article.

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

[1]And vice-versa, for every join there should be a corresponding split with this property.

in which the tasks are either internal actions (e.g., internal data transformations) or interactions with services drawn from a service repository (the *component services*).

Definition 2.1 (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. 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.

Nodes in an orchestration model 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), whereas AND gateways represent parallel forking (AND-split) or synchronization points (AND-join). Split gateways are gateways with one incoming and at least two outgoing edges, while join gateways are gateways with one outgoing and at least two 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*. Empty tasks represent internal actions, like for example a data transformation performed internally by the orchestrator, without involving any component service.

Without loss of generality, we impose the following *well-formedness* conditions:

1. 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. This is a natural assumption given that an orchestration should have a start and an end.
2. Every task node has a single incoming and a single outgoing edge, and every gateway is either a split or a join.
3. The sum of the probabilities attached to the outgoing edges of an XOR-split gateway is 1. If this condition is not satisfied, it is trivial to rewrite the probabilities of XOR-split gateways so that they add up to one by means of normalization.

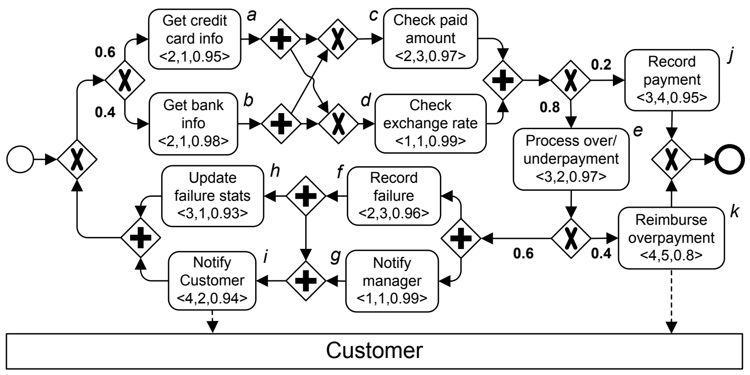

Get credit card info <2,1,0.95>
a
Get bank info <2,1,0.98>
b
0.6
0.4
Check paid amount <2,3,0.97>
c
Check exchange rate <1,1,0.99>
d
0.2
0.8
Record payment <3,4,0.95>
j
Process over/ underpayment <3,2,0.97>
e
Update failure stats <3,1,0.93>
h
Record failure <2,3,0.96>
f
Notify Customer <4,2,0.94>
i
Notify manager <1,1,0.99>
g
0.6
0.4
Reimburse overpayment <4,5,0.8>
k
Customer

Figure 1: Example of Composite Service: Payment

4. 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. This is a natural assumption since an edge that does not stem from an XOR-split must always be taken after its source node has been executed.

As an illustrative example, we consider a simplified *Payment* composite service depicted in Figure 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 '$\times$' (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).

We note that the above orchestration model assumes that the "branching" probabilities for each arc of an XOR-split are known. This assumption is shared with other quantitative process analysis techniques such as process simulation. A typical method for estimating branching probabilities for quantitative process analysis is by analyzing logs of past executions of the composite service. The ProM framework [11] for example – via its "heuristic net miner" – is able to extract such branching probabilities from execution logs. If no logs of previous executions are available, domain expert opinion is an alternative, albeit arguably less reliable.

Also, it should be noted that the adopted orchestration meta-model is not intended to be a comprehensive or complete process modeling language. In this respect it is customary in previous work to evaluate the comprehensiveness of process meta-models using the so-called workflow patterns [12]. With respect to these patterns, the adopted orchestration meta-model supports the basic control-flow patterns, namely sequence, parallel split (AND-split), synchronization (AND-join), exclusive choice (XOR-split), simple merge (XOR-merge). Additionally, it supports the *multi-choice* pattern insofar as a multi-chocie can be directly rewritten as a combination of AND-split and XOR-split as explained in [12]. In a similar vein, the orchestration model supports the *synchronizing merge* pattern when it is used in a structured block (also called *structured synchronizing merge*) since this pattern can be rewritten in terms of AND and XOR splits. The orchestration meta-model also supports the *deferred choice pattern* since it abstracts away from the way choices are made. All that is required is that the modeler specify the branching probability for each decision point, and not the mechanism of choice. In particular, the choice mechanism could be based on a race condition as in the deferred choice.

Other workflow patterns are not (directly) supported by the adopted orchestration meta-model although in some cases work-arounds are possible to fit them into the constructs of our orchestration meta-model as explained in [12]. In summary, the orchestration meta-model supports the basic patterns and some additional patterns. This subset, while not complete, is arguably representative of a large class of orchestration models found in practice.

Finally, it should be noted that the adopted orchestration models abstract away from the data manipulated by the orchestration. Again, this assumption is shared with other quantitative process analysis techniques. In general, analysis of process models with data is undecidable [13]. Some recent advances have led to promising techniques for analyzing temporal properties on restricted classes of process models with data via abstraction techniques [13], but further research is needed to make these or similar techniques applicable to quantitative process analysis problems such as QoS estimation.

2.2. Quality of Service Model

QoS computations on service orchestrations 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. 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 2.2 (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., [6, 14, 15, 7, 8, 5]). With respect to the method for computing QoS attribute values for composite services, we classify the QoS attributes studied in this previous body of work 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) [6].

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. Typically, these attributes arise when capturing *failures*. Indeed, the higher is the number of tasks executed during a service orchestration, the greater are the chances that at least one task will fail, which in turn results in a failure of the orchestration. In other words, failure rates of tasks are multiplicative. Examples include *reliability* (percentage of composite service executions where no component service fails) and *availability* (percentage of composite service executions where all component services are available when invoked) [5].

For the sake of illustration, the rest of the article focuses on three representative attributes (one per category). For each service s, $QoS(s) = \langle T, C, R \rangle$, where T, C and R stand for time, cost and reliability. In Figure 1, for example, the numbers in each service denote its QoS attributes. Note that (for simplicity) this example does not include any empty tasks. If there was an empty task, its QoS would by default be $\langle 0, 0, 1\rangle$, meaning that an empty task is assumed to take zero time, zero cost and has 100% reliability. These default values could, of course, be overridden, for example to capture the fact that a data transformation takes some time.

Also, the article focuses on computing the *mean* of each of these attributes for a service orchestration, as opposed for example to other aggregation functions such as the *median* or the Xth percentile. In other words, we assume that the input values of T, C and R for each task represent means and we aim to compute means values for these attributes at the level of the orchestration.

Notwithstanding the fact that the article focuses on three attributes, the method is more general and can be extended to other attributes and aggregation functions. To extend the method, a tool developer would need to define the additional attributes and aggregation functions as discussed later in the article.

3. Anatomizing Orchestration Models

In order to analyze the QoS of orchestration models, we decompose them into *orchestration components*. An orchestration component is a subgraph of the orchestration model with a single-entry and single-exit point. The largest orchestration component is the entire orchestration model, while the smallest orchestration components are the individual tasks. The overarching idea is that QoS is computed independently for each orchestration component in a bottom-up manner, starting from individual tasks and ending with the entire orchestration model, which is the goal of the method. In this section, we introduce the approach we employ for identifying and representing orchestration components.

3.1. Refined Process Structure Tree and Maximally-Structured Orchestrations

The Refined Process Structure Tree (RPST) [16, 17] is a technique to parse a process model (and in particular an orchestration model) into a tree of single-entry, single-exit (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

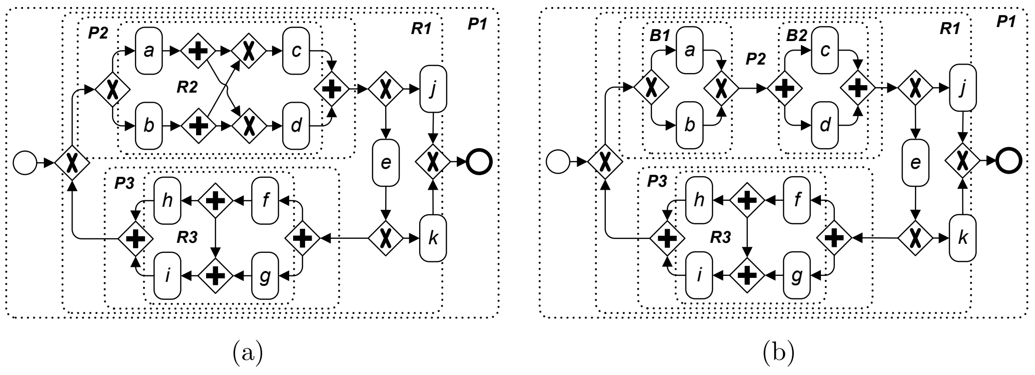


Figure 2: (a) Service orchestration, (b) maximally-structured representation of (a)

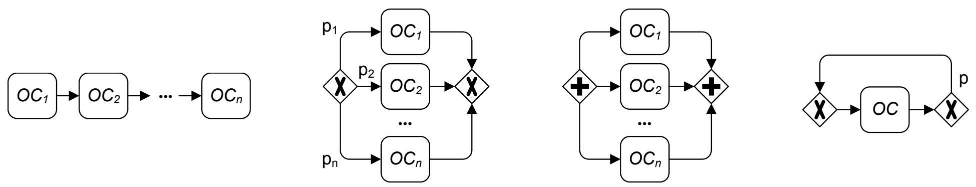
components that share two common nodes. Any other component is a *rigid* (R) component. Importantly, the RPST exists for any process model and it is unique [16, 17].

Figure 2(a) exemplifies the RPST of the running example given in Figure 1. Note that Figure 2(a) uses short-names for tasks $(a, b, c, \ldots)$, which appear next to each task in Figure 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 displayed for simplicity reasons.

Bond and polygon components are well-structured and straightforward to analyze as discussed later in the article. On the other hand, rigid components correspond to the unstructured parts of the orchestration model. The service orchestration in Figure 2(a) contains three rigid components.

In order to maximize the degree of well-structuredness and thus facilitate the analysis of QoS, we employ the structuring technique proposed in [18, 19]. This technique transforms (whenever possible) an unstructured orchestration model into a *maximally-structured* orchestration model under fully concurrent bisimulation equivalence [20]. In particular, by applying the technique from [18] to the model in Figure 2(a), one obtains the service orchestration given in Figure 2(b). Note that rigid component $R2$ gets an equivalent representation consisting of a polygon with two bond components as children, namely $B1$ and $B2$.

Some rigid components are irreducible, meaning that they cannot be rewritten into bond and polygon components. In other words, some rigid



(a) *SEQ* (b) *CHC* (c) *PAR* (d) *RPT*

Figure 3: Structured orchestration components

components may still appear in a maximally-structured orchestration model. Such irreducible rigids can be classified into two categories [18]: irreducible Directed Acyclic Graphs (DAG) and irreducible multiple-entry, multiple-exit (MEME) loops. Note that despite their name, these MEME loops are in fact SESE components. Indeed, while a loop may have multiple entries and multiple exits, it is still enclosed within a minimum-bounding SESE component, and this latter component is the one that we manipulate.

In light of the above, we can represent a maximally-structured orchestration model using the following abstract syntax.

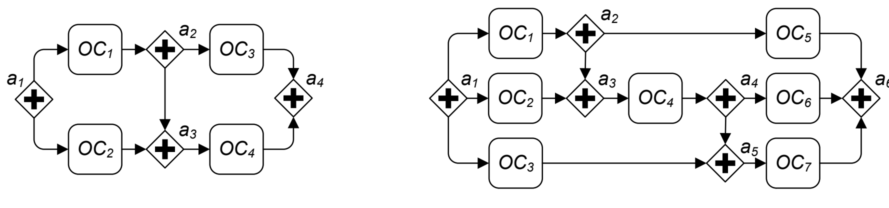
Definition 3.1 (Syntax of Maximally-Structured Orchestrations). Let P be the range of real numbers from 0.0 to 1.0.

$$\begin{aligned}
OrchestrationComponent(OC) &::= \tau \mid \mathit{Service} \mid \mathit{SC} \mid \mathit{UC} \\
OrchestrationElement(OE) &::= \mathit{OC} \mid \mathit{AND} \mid \mathit{XOR} \\
StructuredComponent(SC) &::= \mathit{SEQ}(\{\mathit{OC}\}) \mid \mathit{CHC}(\{\mathit{OC} \times P\}) \\
&\quad\ \mid \mathit{PAR}(\{\mathit{OC}\}) \mid \mathit{RPT}(\mathit{OC} \times P) \\
UnstructuredComponent(UC) &::= \mathit{DAG}(\{\mathit{OE} \times P \times \mathit{OE}\}) \\
&\quad\ \mid \mathit{MEMELoop}(\{\mathit{OE} \times P \times \mathit{OE}\})
\end{aligned}$$

This abstract syntax is based on a distinction between the following types of orchestration components: empty (τ) tasks, tasks bound to services (*Service*), structured orchestration components (*SC*), and unstructured orchestration components (*UC*). Further, there are four types of structured orchestration components, namely sequence (*SEQ*), choice (*CHC*), parallel (*PAR*), and repeat (*RPT*), as exemplified in Figure 3. A sequence component is a list of orchestration components. A choice component is a set of orchestration components, each one associated with a probability P of being chosen. A parallel component is a set of orchestration components. Finally, a repeat component is an orchestration component along with the probability that the back-edge from the exit point to the entry is taken (denoted by P in the abstract syntax). For example, in Figure 2(b), bond $B1$ is a choice component, bond $B2$ is a parallel component, and polygon $P2 = \{B1, B2\}$ is a sequence component. Unstructured components (UC) are classified into acyclic components (DAG) and cyclic components ($MEMELoop$). These two latter types of components are described in further details in the following subsections. Given the above abstract syntax, a maximally-structured orchestration model is represented by means of its top-level orchestration component (i.e., the orchestration component corresponding to the entire model).

3.2. DAG Components

As stated above, acyclic rigids that are present in maximally-structured orchestration models are hereby called irreducible DAG components. Figure 4 gives an example of such a DAG component: Figure 4(a) shows the simplest form of an irreducible DAG component (a well-known Z-structure studied in [21]), whereas Figure 4(b) displays a more complex irreducible DAG component that can be seen as a composition of Z-structures. Observe that rigid $R3$, both in Figure 2(a) and in Figure 2(b), is an irreducible DAG component.



(a) Z-structure (b) Composition of Z-structures

Figure 4: DAG components

We treat an (irreducible) DAG component as a set of tuples (OE_1, p, OE_2), where OE_1 and OE_2 are orchestration elements (orchestration components or gateways) and p is the probability that OE_2 will be executed after accomplishment of OE_1. For instance, the Z-structure in Figure 4(a) is described by the set $\{(a_1, 1.0, OC_1)$, $(a_1, 1.0, OC_2)$, $(OC_1, 1.0, a_2)$, $(OC_2, 1.0, a_3)$, $(a_2, 1.0, a_3)$, $(a_2, 1.0, OC_3)$, $(a_3, 1.0, OC_4)$, $(OC_3, 1.0, a_4)$, $(OC_4, 1.0, a_4)\}$.

In order to analyze DAG components, we introduce the notion of a *run*. A run is a subgraph of a DAG component comprising the set of all edges traversed in one possible execution of the component. The concept is akin to the notion of *execution trace*, but it should be kept in mind that in a run, there can be parallel splits and joins (and thus a run may contain multiple paths), whereas traditionally, a trace is defined as a specific path in a graph. Each run in a DAG can be associated with the probability that a given execution of the component follows this particular run. Below, we are interested in extracting both the runs of a DAG component and the probability associated to each run.

To explain how runs are extracted, we make use of three notions: *choice edge*, *compatibility* of choice edges and *configuration.* A *choice edge* is an edge whose source is an XOR-split. Two choice edges are said to be *compatible* iff both may be traversed in a single run. Formally, two choice edges $e_i, e_j \in E$ are compatible iff $(\text{src}(e_i), \text{src}(e_j)) \in E^+ \wedge\ (\text{tgt}(e_i), \text{tgt}(e_j)) \notin E^+$, where E^+ denotes the transitive closure of E. To illustrate this concept, consider the DAG component presented in Figure 5(a). If we take the pair of choice edges e_1 and e_2, we can see that they are compatible because if e_1 is taken it is still possible to follow e_2. If we take the pair of edges e_1 and e_5, however, we can easily see that no execution in which both e_5 and e_1, because e_5 becomes unreachable once e_1 has been chosen. Hence e_1 and e_5 are incompatible.

A *configuration* of a DAG component is a maximal set of mutually compatible choice edges from the DAG component. By maximal, we mean that there is no superset of a configuration that contains only mutually compatible choice edges. The set of configurations of a DAG can be computed using an algorithm described in [22]. This algorithm has an exponential-time complexity, which inherently is due to the fact that the number of configurations (and the number of runs) of an acyclic process model is in the worst-case exponential on the number of nodes in the graph.

Given a DAG component and one of its configurations, Algorithm 1 computes the run associated to this configuration as well as the probability of this run. The algorithm starts by making a copy of the entire graph G (line 1). Then, it iterates over the set of choice edges of configuration θ and removes all the outgoing edges incident to each XOR split gateway in the DAG, except for those included in the configuration (lines 2-3). As a result of this step, a number of elements in the copy of the graph become unreachable from entry node x_ε. Therefore, all dangling elements are removed (lines 4-6). The algorithm proceeds by calculating the aggregate probability for the entire run, which is equal to the product of the probabilities of the choice edges in the input configuration. Next, the algorithm resets the value of the choice edge in the resulting graph to 1.0 (lines 7-10). Finally, the algorithm returns the tuple (γ, p_γ) representing the run and its associated probability.

```
Algorithm 1: Compute the run induced by a configuration
  Input:  $G$ – DAG component as a set of edges
          $x_\varepsilon$ – Entry node of the DAG component
          $\theta$ – A configuration of DAG component
  Output: $(\gamma, p_\gamma)$ – Run induced by configuration $\theta$
1  $\gamma \leftarrow G$
2  foreach $(x, p, y) \in \theta \ \wedge \ (x, q, z) \in \gamma : y \neq z$ do
3      $\gamma \leftarrow \gamma \setminus \{(x, q, z)\}$
4  $E \leftarrow \{(x, y) \mid (x, p, y) \in \gamma\}$
5  foreach $(x, p, m) \in \gamma : (x_\varepsilon, x) \notin E^+$ do
6      $\gamma \leftarrow \gamma \setminus \{(x, p, m)\}$
7  $p_\gamma \leftarrow 1.0$
8  foreach $(x, p, y) \in \theta$ do
9      $\gamma \leftarrow \gamma \setminus \{(x, p, y)\} \cup \{(x, 1.0, y)\}$
10     $p_\gamma \leftarrow p_\gamma \cdot p$
11 return $(\gamma, p_\gamma)$
```

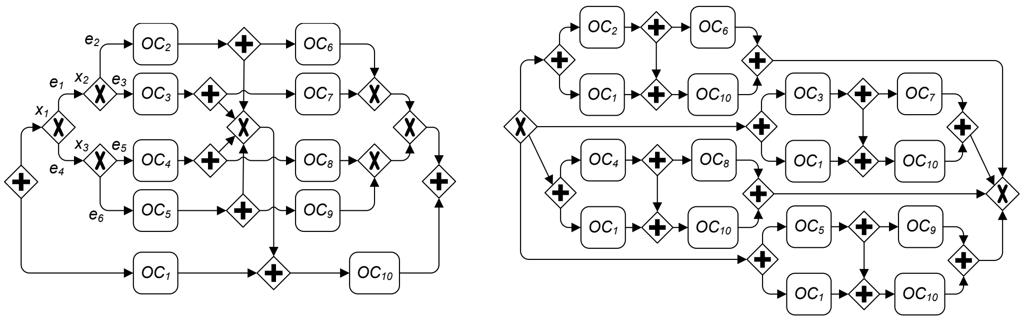
By applying the above algorithm to each configuration of a DAG, we can extract the set of runs of a DAG component and their associated probabilities. With the set of runs thus computed, we can construct a structured choice component that captures the behavior of the original DAG component. To achieve this, one XOR-split gateway is introduced that has a branch leading to each run with its corresponding probability. Conversely, one XOR-join gateway is added to merge the exit node of every run. For example, the choice component corresponding to the DAG component in Figure 5(a) is given in Figure 5(b).

In more specific terms, if we use the *CHC* constructor introduced in Definition 3.1, the structured choice component corresponding to a DAG component is defined by the expression $CHC(\{(r_1, p_1), (r_2, p_1), \ldots (r_n, p_n)\})$ where r_i represents a run of the DAG component, while p_i represents the probability of r_i.

We note that the choice component constructed in this way is *trace equiv-*

alent with the original DAG component, meaning that they have the same set of traces. Although trace equivalence is generally a weak notion of equivalence [20], it is sufficient in our context since we are interested in the mean QoS values of the executions of the orchestration model (or of a component thereof). By definition, the QoS of an orchestration component is the mean of the QoS of the executions of the component, when the number of executions tends to infinity. If we executed the DAG component a very large number of times, we would observe that each possible trace of the component is executed a certain number of times. Let us call f_i the percentage of times that a given possible trace t_i is observed, relative to the total number of executions (e.g. $f_1 = 0.1$ means that 10% of the executions followed trace t_1). The mean of the QoS of the set of executions is then:

$$\sum_{i=1}^{n} f_i \times QoS(t_i) \tag{1}$$



(a) DAG component (b) Structured choice component

Figure 5: Transformation of a DAG component into a (structured) choice component

As the number of executions becomes large, f_i tends to a certain value F_i. We observe that each trace corresponds to a traversal of the component from the entry to the exit node, and each edge is traversed with a probability that is equal to 1.0 for all edges except the edges emanating from XOR-splits (i.e. the conditional branches). Accordingly, F_i is the product of the probabilities of the conditional branches of the orchestration model traversed in trace t_i. Here, we observe that this product is exactly the value of p_i for the run that subsumes trace t_i. Repeating the same reasoning starting from the choice component, we observe that the QoS is also given by an equation of the same form as Equation 1, and that each f_i tends to p_i. Thus the mean QoS of the DAG component is the same as the mean QoS of the corresponding choice component.

Complexity. We note that the complexity of computing the runs is exponential due to the fact that the number of runs is exponential. Algorithm 1 adds a polynomial factor to this inherently exponential complexity. However, the computation of runs is only performed for irreducible acyclic DAGs, which contain only a subset of the edges of the orchestration model. Furthermore, as discussed in Section 5, the exponential worst-case complexity is not necessarily an obstacle when dealing with models found in commercial practice. Should the exponential complexity raise a practical problem, one could optimize the technique for generating runs by stopping the production of runs once the sum of the probabilities of the runs accumulated so far is above a certain threshold (e.g. 99%). This would mean that the runs corresponding to 99% of the executions have been obtained and this set can be used as an approximation.

3.3. MEME Loop Components

MEME loop components are unstructured (i.e. rigid) components that embed loops in topologies that disallow their transformation into equivalent structured components. Such irreducible components arise for example when a loop has multiple exit points as shown in Figure 6. Indeed, it is well-known that restructuring loops with multiple exit points cannot be done solely via node duplication, but requires the introduction of variables and XOR gateways to "emulate" the control flow of the loop [23]. For example, the MEME loop shown in Figure 6 contains two loops, namely $\mathcal{L}_1$ and $\mathcal{L}_2$. The reader can easily check that loop $\mathcal{L}_2$ can be transformed into a single-entry/single-exit loop via node duplication, but loop $\mathcal{L}_1$ cannot unless we introduce variables. Introducing such variables would be problematic for QoS computation since we would need to take these variables into account in the QoS computation. Instead of trying to re-write multiple-exit loops into single-exit loops using variables, we propose a technique for computing the QoS of MEME loop components under certain conditions. This technique for computing QoS of MEME loop components (cf., Section 4.3) is general enough so that the transformation of $\mathcal{L}_2$ into an equivalent structured component is not required.

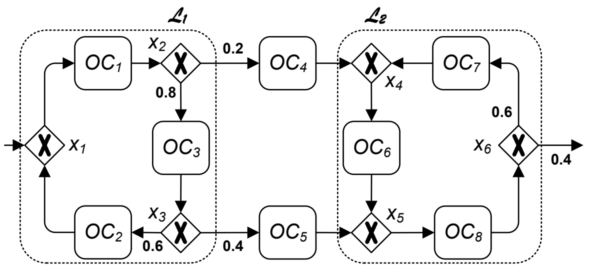
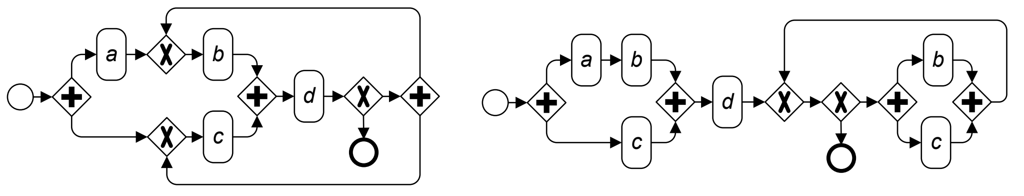


Figure 6: MEME loop component

Similar to DAG components, a MEME loop component is represented as a set of tuples $(OE_i, p_{i,j}, OE_j)$, each tuple corresponding to one edge. OE_i and OE_j are the source and target nodes, respectively, while $p_{i,j}$ is the probability of traversing the edge. For instance, the MEME loop in Figure 6 is represented as the set $\{(x_1, 1, OC_1), (OC_1, 1, x_2), (x_2, 0.8, OC_3), (x_2, 0.2, OC_4), \ldots, (x_6, 0.6, OC_7)\}$. As in the case of DAG components, we assume that the probability associated to choice edges (edges stemming from XOR-split gateways) is given, and that probability of all other edges in the component is equal to 1.

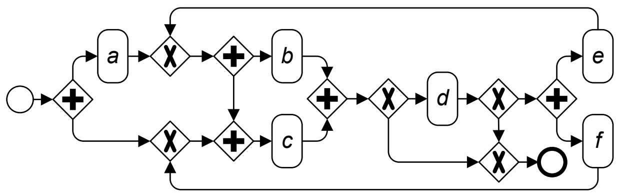
In the general case, a MEME loop component may involve complex combinations of concurrency and conditional branching as in the example shown in Figure 7(a). The structuring technique implemented in BPStruct tool [24] transforms a rigid component into a combination of polygon and bond components (i.e. structured components), whenever this transformation is possible. For example, given the model in Figure 7(a), BPStruct produces the structured model given in Figure 7(b). Since the resulting component is well-structured, its QoS can be calculated using the same method as for structured components.



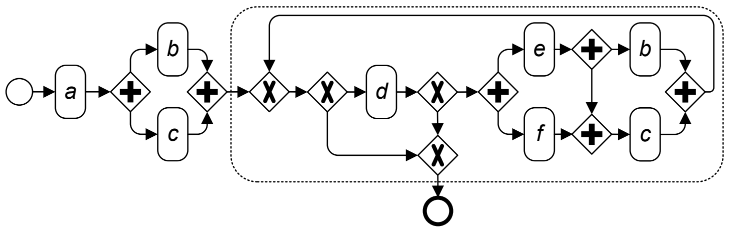
(a) Cyclic rigid with concurrency (b) Structured component equivalent to (a)

Figure 7: Cyclic rigid component and equivalent well-structured component.

Some rigid components cannot be transformed into equivalent structured components as discussed above. In this case, BPStruct attempts to transform the rigid component into an equivalent rigid component where the concurrency is fully encapsulated within child components. This latter type of cyclic rigid component is hereby called a *MEME loop component with encapsulated concurrency*. MEME loop components with encapsulated concurrency only contain two types of nodes: XOR gateways and nodes representing child components. An example of this case is shown in Figure 8. Here the concurrency (captured by the AND gateways) has been factored out into a DAG component that is a child of a MEME loop component – the latter is shown in dotted lines in Figure 8(b). This MEME loop component contains only XOR gateways and child components, i.e. it is a MEME loop component with encapsulated concurrency.



(a) Cyclic rigid component with concurrency



(b) MEME loop with encapsulated concurrency equivalent to (a)

Figure 8: Cyclic rigid component and equivalent MEME loop with encapsulated concurrency.

Finally, a third case is the one where BPStruct is not able to fully encapsulate the concurrency into child components. In this case, BPStruct produces a rigid component that contains both XOR gateways and AND gateways mixed together. This is the case of the example shown in Figure 9, which cannot be further refactored by BPStruct. To the best of our knowledge, it is

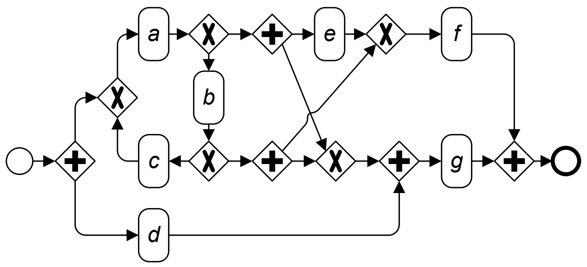


Figure 9: Example of cyclic rigid component with non-encapsulated concurrency that cannot be transformed into an equivalent rigid component with encapsulated concurrency by BPStruct.

an open research question whether or not this type of rigid component can be rewritten into an equivalent one where the AND gateways and the XOR gateways are encapsulated in separate components (under fully concurrent bisimulation equivalence which is the equivalence notion used in BPStruct).

The QoS aggregation method proposed below assumes that the MEME loop components that remain after BPStruct's transformation are MEME loop components with encapsulated concurrency – or MEME loop components that, in their initial form, contain only XOR gateways. In other words, the proposed method cannot handle MEME loop components such as the one in Figure 9. Lifting this restriction is left as future work.

4. Quality of Service Aggregation

In this section, we show how to compute the QoS of orchestration models. First, we define functions for computing the QoS of each type of orchestration component and then we combine these component-specific functions into an algorithm for computing the aggregate QoS of an orchestration model.

4.1. QoS of Structured Components

Assuming that the QoS of the orchestration elements under an orchestration components is known, the QoS of structured orchestration components is computed based on the following equations, which are taken from [6].

$$
\begin{aligned}
QoS(SEQ) &= \left\langle \sum_{oc\in SEQ} T(oc),\ \sum_{oc\in SEQ} C(oc),\ \prod_{oc\in SEQ} R(oc) \right\rangle \\
QoS(CHC) &= \left\langle \sum_{(oc,p)\in CHC} p\ T(oc),\ \sum_{(oc,p)\in CHC} p\ C(oc),\ \sum_{(oc,p)\in CHC} p\ R(oc) \right\rangle \\
QoS(RPT) &= \left\langle \frac{T(oc)}{1-p},\ \frac{C(oc)}{1-p},\ R(oc)^{(1-p)^{-1}} \right\rangle \quad \text{with } RPT = (oc, p) \\
QoS(PAR) &= \left\langle \max_{oc\in PAR}\{T(oc)\},\ \sum_{oc\in PAR} C(oc),\ \prod_{oc\in PAR} R(oc) \right\rangle \qquad (2)
\end{aligned}
$$

In these formulas, $T(oc)$, $C(oc)$ and $R(oc)$ denote the time, cost and reliability of orchestration element oc.

For repeat composition $RPT(oc, p)$, the computation considers that the body of the loop (orchestration element oc) may be executed one or more times. Following the well-known geometric series, oc is then expected to be executed $(1 - p)^{-1}$ times[2], where p is the probability of staying in the loop.

Also, note that the formula for calculating the aggregate execution time for parallel components assumes that the value of the sub-component $oc \in PAR$ with the maximum mean execution time is always greater than the value of the QoS of the other components $oc' \in PAR$ with smaller mean execution times. This assumption is implicitly made in [6] and other previous work on QoS aggregation for composite services. If this assumption is not fulfilled, different formulas need to be employed depending on the probability distributions of the sub-components $oc \in PAR$. Formulas for calculating the mean of the maximum of multiple random variables can be found in [25].

4.2. QoS of DAG Components

A DAG orchestration component can be transformed into an equivalent choice component as explained in Section 3.2. Each of the branches in this choice component corresponds to a run of the DAG component. The QoS values calculated for individual runs are aggregated taking into account the probability of each run as follows:

[2]Geometric series: $p^0 + p^1 + p^2 + \ldots = \sum_{i=0}^{\infty} p^i = (1 - p)^{-1}$

$$QoS(DAG) = \left\langle \begin{array}{l} \sum_{(\gamma,p_\gamma)\in\Gamma(DAG)} p_\gamma \ \text{CriticalPath}(\gamma), \\ \sum_{(\gamma,p_\gamma)\in\Gamma(DAG)} p_\gamma \ \sum_{oc\in\gamma} C(oc), \\ \sum_{(\gamma,p_\gamma)\in\Gamma(DAG)} p_\gamma \ \prod_{oc\in\gamma} R(oc) \end{array} \right\rangle \tag{3}$$

where $\Gamma(DAG)$ denotes the set of runs of DAG component DAG. This set is computed by iteratively calling Algorithm 1. For a given run $(\gamma, p_\gamma) \in \Gamma(DAG)$, 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.

4.3. QoS of MEME Loop Components

The method to compute QoS of MEME loop components relies on the theory of Markov chains. The use of Markov chain for handling this type of component is possible because we have made the assumption that MEME loop components do not include concurrent behavior (cf. Section 3.3). All concurrency, if any, should be enclosed within child components.

A Markov chain is a state-transition system in which the transitions are associated with probabilities. The states in an absorbing Markov chain are classified into transient and absorbing. An absorbing state has a single self-transition with a probability of one, i.e., once the Markov chain reaches an absorbing state it is impossible to leave it. Any other state is called a transient state. A Markov chain is said absorbing if it has at least one absorbing state and if it is possible to reach an absorbing state from every other (transient) state.

Given the assumption that a MEME loop component only contains XOR gateways, it is trivial to convert it into a Markov chain. To illustrate the mapping, consider the MEME loop component presented in Figure 10(a) and its corresponding absorbing Markov chain as presented in Figure 10(b).

Each node in the MEME loop component is mapped to a state in the Markov chain. To achieve traceability of the mapping, the states in the Markov chain have the same label as in their corresponding node in the MEME loop component. Then, each flow edge in the MEME loop component is mapped to a transition in the Markov chain. Note that transitions on the Markov chain carry the corresponding transition probability. Finally, a fresh

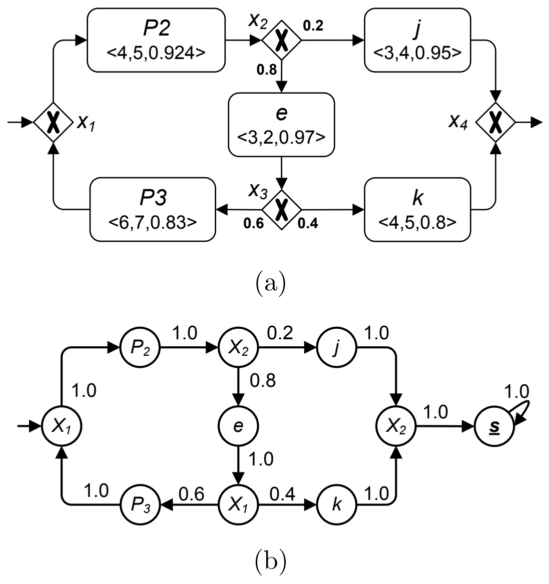


Figure 10: A sample MEME loop component (a), and its corresponding Absorbing Markov Chain (b).

absorbing state is added to the Markov chain, namely $\underline{\mathbf{s}}$. The intuition behind is that state $\underline{\mathbf{s}}$ models the successful completion of an execution of the MEME loop component.

A Markov chain can be represented by means of a *transition probability matrix*, referred to as $\mathbf{P}$. When columns and rows are arranged such that all transient states come first, the matrix is said to be in its *canonical form*. In this representation, the transition probability matrix has the following form:

$$\mathbf{P} = \begin{array}{c|c|c} & TR & ABS \\ TR & \mathbf{Q} & \mathbf{R} \\ \hline ABS & \mathbf{0} & \mathbf{I} \end{array}.$$

In this representation, the rows/columns that encode transient states are labeled TR while the rows/columns that encode absorbing states are labeled ABS. Since the Markov chain of a MEME loop has only one absorbing state, its transition probability matrix has only one ABS row and one ABS column.

The interest of mapping a MEME loop component to an absorbing Markov chain stems from the fact that we can calculate the expected number of times that a node in the MEME loop is visited, using standard methods. Indeed, if we have the transition probability matrix in its canonical form, the expected number of times a transient state j is visited, starting from transient state i (e.g. the entry node of the component) is given by the ij^{th} element of the so-called *fundamental matrix* $\mathbf{S}$ of the chain. A well-known result is that the fundamental matrix of an absorbing Markov chain is:

$$\mathbf{S} = (\mathbf{I} - \mathbf{Q})^{-1} \tag{4}$$

Thus, the average cost and time associated to a MEME loop component L can be calculated using the following equations:

$$QoS_{\text{time}}(L) = \sum_{oc \in L} \mathbf{S}[\text{entry-of}(L), oc]\ T(oc) \tag{5}$$

$$QoS_{\text{cost}}(L) = \sum_{oc \in L} \mathbf{S}[\text{entry-of}(L), oc]\ C(oc) \tag{6}$$

where entry-of(L) refers to the entry node in the MEME loop component L.

To compute reliability, we use the approach proposed in [26]. The key idea of this approach is to map the MEME loop component into a new Markov chain, namely $\hat{\mathbf{P}}$, that takes into account the fact that an execution of a child component can either succeed or fail. A run of a MEME loop component is considered to be successfully completed if and only if none of the children involved in the run fails. Conversely, the failure of one child component entails the failure of the entire MEME loop component. Hence, the Markov chain $\hat{\mathbf{P}}$ has to reflect these two possible outcomes: successful completion and failure. This observation leads us to the idea of mapping the MEME loop component in Figure 10(a) into the Markov chain $\hat{\mathbf{P}}$ shown in Figure 11.

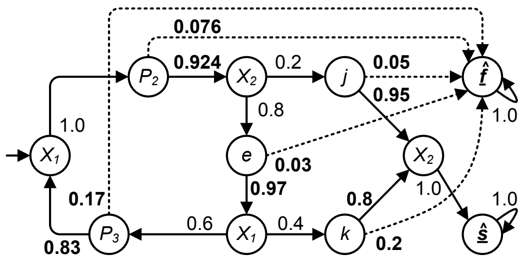


Figure 11: Absorbing Markov Chain modeling the reliability of the MEME loop component in Figure 10(a).

In the Markov chain $\hat{\mathbf{P}}$, the absorbing state $\hat{\underline{\mathbf{s}}}$ specifies the successful completion, whereas the absorbing state $\hat{\underline{\mathrm{f}}}$ models failures. Note that the states in the Markov chain that represent a child of the MEME loop component now have two outgoing transitions. The first transition corresponds to the original control flow, i.e., the path towards the successful completion. The transition probability of this transition corresponds to the reliability of the underlying child component. The second transition goes to the "failure" state $\hat{\underline{\mathrm{f}}}$ (cf. dashed lines in Figure 11). The probability of the transition from a state s to the failure state $\hat{\underline{\mathrm{f}}}$ is 1 minus the reliability of the child component represented by s. Using the resulting Markov chain, the problem of calculating the reliability of a MEME loop component corresponds to computing the probability of reaching the failure state $\hat{\underline{\mathbf{s}}}$ when starting from the initial state. This probability is also referred to as the *absorbing probability*, and it can be derived from using the following equation:

$$\hat{\mathbf{B}} = \hat{\mathbf{S}} \cdot \hat{\mathbf{Q}} \tag{7}$$

where $\hat{\mathbf{Q}}$ is the upper right submatrix of the Markov chain $\hat{\mathbf{P}}$ and $\hat{\mathbf{S}}$ is the fundamental matrix of $\hat{\mathbf{P}}$. Since $\hat{\mathbf{P}}$ has exactly two absorbing states, $\hat{\mathbf{B}}$ is a two-column matrix. Henceforth, the reliability associated to a MEME loop component L is the value of $\hat{\mathbf{B}}[\text{entry-of}(L), \hat{\underline{\mathbf{s}}}]$, where entry-of($L$) is the entry node to the MEME loop component.

4.4. QoS of Composite Services

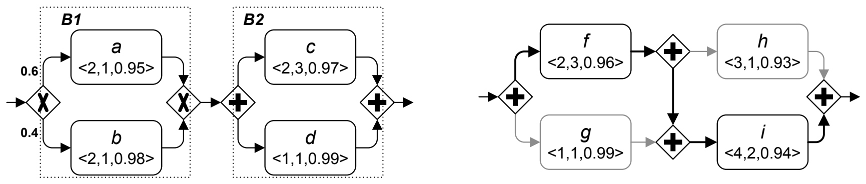
In order to compute the aggregate QoS of a composite service, its maximally-structured orchestration model (represented as an RPST) is traversed in breadth-first-search post-order, i.e., starting from the leaf nodes and moving upwards to the root node. At each node – which corresponds to an orchestration component – the QoS computation methods previously outlined are applied. The overall procedure is described in Algorithm 2.

To illustrate Algorithm 2, we compute the QoS values for the composite service presented in Figure 1. The computation is based on its maximally-structured orchestration model given in Figure 2(b).

Let us assume that the first structured component to be considered is the CHC component B1 followed by the PAR component B2 (cf., Figure 12(a)). Their corresponding QoS tuples are:

Algorithm 2: Compute QoS of a component: ComputeQoS(OC)

```
Input: $OC$ – Node of the RPST
Global: $QoS$ – Map holding QoS tuples (i.e., $QoS : OC \mapsto \langle T, C, R \rangle$)
Inline : $T(OC) =_{def} QoS(OC)|_T$, $C(OC) =_{def} QoS(OC)|_C$, and
         $R(OC) =_{def} QoS(OC)|_R$ are the projection of attributes of
         QoS in the tuple associated to $OC$
1  foreach $oc \in$ children-of($OC$) do
2      ComputeQoS($oc$)
3  switch type-of($OC$) do
4      case $SEQ$
5          $QoS_{OC} \leftarrow \left\langle \sum_{oc \in OC} T(oc), \sum_{oc \in OC} C(oc), \prod_{oc \in OC} R(oc) \right\rangle$
6      case $CHC$
7          $QoS_{OC} \leftarrow$
           $\left\langle \sum_{(oc,p) \in OC} p\ T(oc), \sum_{(oc,p) \in OC} p\ C(oc), \sum_{(oc,p) \in OC} p\ R(oc) \right\rangle$
8      case $RPT$
9          $QoS_{OC} \leftarrow \left\langle \frac{T(oc)}{1-p}, \frac{C(oc)}{1-p}, R(oc)^{(1-p)^{-1}} \right\rangle$    with $OC = (oc, p)$
10     case $PAR$
11         $QoS_{OC} \leftarrow \left\langle \max_{oc \in OC} T(oc), \sum_{oc \in OC} C(oc), \prod_{oc \in OC} R(oc) \right\rangle$
12     case $DAG$
13         $QoS_{OC} \leftarrow \left\langle \begin{array}{l} \sum_{(\gamma,p_\gamma) \in \Gamma(OC)} p_\gamma\ CriticalPath(\gamma), \\ \sum_{(\gamma,p_\gamma) \in \Gamma(OC)} p_\gamma \sum_{oc \in \gamma} C(oc), \\ \sum_{(\gamma,p_\gamma) \in \Gamma(OC)} p_\gamma \prod_{oc \in \gamma} R(oc) \end{array} \right\rangle$
14     case $MEMELoop$
15         Map OC to $\mathbf{P}$ (for computing time and cost values) and to $\hat{\mathbf{P}}$
           (for computing the reliability value).
16         Compute $\mathbf{S}$ from $\mathbf{P}$ according to Equation 4
17         Compute $\hat{\mathbf{S}}$ from $\hat{\mathbf{P}}$ according to Equation 4
18         Compute $\hat{\mathbf{B}}$ according to Equation 7
19         $QoS_{OC} \leftarrow \left\langle \begin{array}{l} \sum_{oc \in OC} \mathbf{S}[\text{entry-of}(OC), oc]\ T(oc), \\ \sum_{oc \in OC} \mathbf{S}[\text{entry-of}(OC), oc]\ C(oc), \\ \hat{\mathbf{B}}[\text{entry-of}(OC), \underline{\hat{\mathbf{s}}}] \end{array} \right\rangle$
20 $QoS \leftarrow QoS \cup \{(OC \mapsto QoS_{OC})\}$
```



(a) Component P2 (b) Component P3

Figure 12: Excerpt of the Running Example

$$\begin{aligned}
QoS(B1) &= \langle 2 \cdot 0.6 + 2 \cdot 0.4, \quad 1 \cdot 0.6 + 1 \cdot 0.4, \quad 0.95 \cdot 0.6 + 0.98 \cdot 0.4 \rangle \\
&= \langle 2,\ 1,\ 0.962 \rangle \\
QoS(B2) &= \langle \max\{2, 1\}, \quad 3 + 1, \quad 0.97 \cdot 0.99 \rangle \\
&= \langle 2,\ 4,\ 0.9603 \rangle
\end{aligned}$$

The results can be aggregated to determine the QoS tuple of Component P2 as follows:

$$\begin{aligned}
QoS(P2) &= \langle 2 + 2, \quad 1 + 4, \quad 0.962 \cdot 0.9603 \rangle \\
&= \langle 4,\ 5,\ 0.924 \rangle
\end{aligned}$$

Now let us consider the case of DAG component R3 (cf., Figure 12(b)). Note that it corresponds to a single Z-structure such that there is only one run to be analyzed.[3] Moreover, the QoS tuple for component P3 is the same as for component R3, and corresponds to the following:

$$\begin{aligned}
QoS(P3) &= \\
QoS(R3) &= \langle \max\{2 + 3, 2 + 4, 1 + 4\},\ 3 + 1 + 1 + 2,\ 0.96 \cdot 0.99 \cdot 0.93 \cdot 0.94 \rangle \\
&= \langle 6,\ 7,\ 0.83 \rangle
\end{aligned}$$

The next component to be analyzed is $R1$, which is a MEME Loop component. The QoS of $R1$ is indeed the same as for component $P1$ and therefore for the entire service orchestration. To ease the analysis, we have replaced

[3]To ease the identification of the critical path, the set of tasks that are not part of it have gray borders.

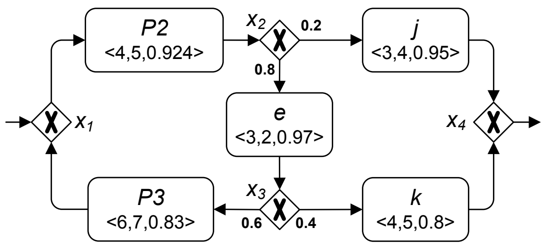


Figure 13: A MEME loop component.

the components $P2$ and $P3$ with tasks displaying the corresponding QoS, as shown in Figure 13. The absorbing Markov chain of $R1$ is the same that we used for illustrating the mapping in the previous section (cf., Figure 10(b)), and its transition probability matrix is the following:

$$\mathbf{P} = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ P2 \\ P3 \\ e \\ j \\ k \\ \underline{\mathsf{s}} \end{array} \left(\begin{array}{ccccccccc|c} x_1 & x_2 & x_3 & x_4 & P2 & P3 & e & j & k & \underline{\mathsf{s}} \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.8 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.6 & 0 & 0 & 0.4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right)$$

We have arranged the matrix to be in the canonical form. Therefore, $\mathbf{Q}$ corresponds to the upper left sub-matrix. Then we compute the fundamental matrix with equation 4. The expected number of times every orchestration element is visited is given by the following row vector:

$$\mathbf{S}[x_1, *] = \begin{array}{c} x_1 \quad x_2 \quad x_3 \quad x_4 \quad P2 \quad P3 \quad e \quad j \quad k \\ \left(\begin{array}{ccccccccc} 1.92 & 1.92 & 1.54 & 1 & 1.92 & 0.92 & 1.54 & 0.38 & 0.62 \end{array} \right) \end{array}$$

Similarly, the absorbing Markov chain that models the reliability of this MEME loop component is given in Figure 11. The corresponding transition probability matrix is the following:

$$\hat{\mathbf{P}} = \begin{array}{c|ccccccccc|cc}
 & x_1 & x_2 & x_3 & x_4 & P2 & P3 & e & j & k & \hat{\underline{\mathsf{s}}} & \hat{\underline{\mathsf{f}}} \\
x_1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
x_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0.8 & 0.2 & 0 & 0 & 0 \\
x_3 & 0 & 0 & 0 & 0 & 0 & 0.6 & 0 & 0 & 0.4 & 0 & 0 \\
x_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
P2 & 0 & 0.92 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.08 \\
P3 & 0.83 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.17 \\
e & 0 & 0 & 0.97 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.03 \\
j & 0 & 0 & 0 & 0.95 & 0 & 0 & 0 & 0 & 0 & 0 & 0.05 \\
k & 0 & 0 & 0 & 0.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0.2 \\
\cline{2-12}
\underline{\mathsf{s}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\underline{\mathsf{f}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}$$

We compute the fundamental matrix $\hat{\mathbf{S}}$ with equation 4, and then the absorption probability matrix $\hat{\mathbf{B}}$ with equation 7 as follows:

$$\hat{\mathbf{B}} \;=\; \hat{\mathbf{S}} \cdot \hat{\mathbf{Q}}$$

$$= \begin{array}{c|ccccccccc|}
 & x_1 & x_2 & x_3 & x_4 & P2 & P3 & e & j & k \\
x_1 & 1.56 & 1.44 & 1.12 & 0.63 & 1.56 & 0.67 & 1.15 & 0.29 & 0.45 \\
x_2 & 0.6 & 1.56 & 1.21 & 0.68 & 0.6 & 0.72 & 1.24 & 0.31 & 0.48 \\
x_3 & 0.77 & 0.72 & 1.56 & 0.63 & 0.77 & 0.93 & 0.57 & 0.14 & 0.62 \\
x_4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
P2 & 0.56 & 1.44 & 1.12 & 0.63 & 1.56 & 0.67 & 1.15 & 0.29 & 0.45 \\
P3 & 1.29 & 1.19 & 0.93 & 0.52 & 1.29 & 1.56 & 0.95 & 0.24 & 0.37 \\
e & 0.75 & 0.69 & 1.51 & 0.61 & 0.75 & 0.91 & 1.56 & 0.14 & 0.6 \\
j & 0 & 0 & 0 & 0.95 & 0 & 0 & 0 & 1 & 0 \\
k & 0 & 0 & 0 & 0.8 & 0 & 0 & 0 & 0 & 1
\end{array}
\begin{array}{|cc|}
\hat{\underline{\mathsf{s}}} & \hat{\underline{\mathsf{f}}} \\
0 & 0 \\
0 & 0 \\
0 & 0 \\
1 & 0 \\
0 & 0.08 \\
0 & 0.17 \\
0 & 0.03 \\
0 & 0.05 \\
0 & 0.2
\end{array}$$

$$\hat{\mathbf{B}}[x_1, *] = \begin{array}{cc}
\hat{\underline{\mathsf{s}}} & \hat{\underline{\mathsf{f}}} \\
\big(\; 0.63 & 0.37 \; \big)
\end{array}$$

We now have all the elements to calculate the QoS values for $R1$ (and hence for $P1$). Time and cost values are calculated with equations 5 and 6, reliability is retrieved from $\hat{\mathbf{B}}$, such that we get the following:

$$\begin{aligned}
QoS(P1) &= \\
QoS(R1) &= \left\langle \begin{array}{l} 1.92 \cdot 4 + 0.92 \cdot 6 + 1.54 \cdot 3 + 0.38 \cdot 3 + 0.62 \cdot 4, \\ 1.92 \cdot 5 + 0.92 \cdot 7 + 1.54 \cdot 2 + 0.38 \cdot 4 + 0.62 \cdot 5, \\ 0.63 \end{array} \right\rangle \\
&= \langle \mathbf{21.44},\ \mathbf{23.74},\ \mathbf{0.63} \rangle
\end{aligned}$$

4.5. Extensibility

In the above presentation of the QoS aggregation method, we have assumed that the QoS vectors include three attributes, namely cost, time and reliability. However, the method is extensible to other QoS attributes as explained below.

Essentially, what the method does is to traverse the tree of orchestration components from bottom to top. At each level, one aggregation function for each QoS attribute is invoked in order to compute the attribute value of the component given the attribute values of its sub-components. In order to extend the proposed method to deal with an additional attribute, one would need to define an aggregation function for each type of component listed in Algorithm 2. For a given type of component, the aggregation function takes as input the parameters associated with this component as per Definition 3.1. In the case of DAG components, the aggregation function additionally takes as input the set of runs and their associated probabilities, while in the case of MEME loop components, it takes as input the expected number of executions of each sub-component (i.e. the cell $\mathbf{S}[\text{entry-of}(L), oc]$ in the fundamental matrix of the Markov chain derived from the MEME loop component as in Equations 5 and 6).

In a similar vein, tool developers could extend the proposed method to deal with other statistics besides the mean. For example, a tool developer could introduce an attribute *maxCost* corresponding to the maximum cost of one execution of a service. Having introduced this attribute, the tool developer would then implement aggregation functions to compute the *maxCost* at the level of the orchestration components. Note that if loops are not bounded, the notion of maximum cost is undefined. Accordingly, in order to calculate the maximum value, the orchestration component types in Definition 3.1 would have to be extended to capture the maximum number of times that Repeat loop can be executed as well as the maximum number of times that an arc in a MEME loop component can be traversed during one execution of the component.

Finally, the method presented here is restricted to taking as input scalar values for each attribute (e.g. mean QoS). Another possible extension would be to handle probability distributions for each QoS attribute as opposed to just the mean or the maximum value. In theory, the proposed method could also be extended by introducing QoS aggregation functions that compute the probability distribution of the QoS of a component, given the probability distribution of the QoS of a sub-component. Such aggregation functions have been studied for example in [27].

5. Implementation and Evaluation

We have implemented the proposed QoS aggregation method in a tool, called BPStructQoS, that takes as input orchestration models in BPMN 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. BPStructQoS is distributed as an extension of the BPStruct tool[4] and is available at: `http://sep.cs.ut.ee/Main/BpstructQoS`. Below we present an evaluation of the scalability of the QoS aggregation method using BPStructQoS.

5.1. Dataset

We collected a dataset consisting of 561 BPMN models from the following sources: 8 models from BPMN-to-BPEL case study of the Grabats'2009 graph transformation challenge[5], 8 models from the public Oryx repository[6], 12 models from a repository of process models for local government authorities in China collected by Fudan University, and 533 models from the IBM BIT process library.[7] The models in the dataset where structured using BPStruct, as a preprocessing step. The sizes of models in the dataset (number of process nodes) range from 3 to 47, with an average of 17 nodes. Some of

[4]`http://code.google.com/p/bpstruct`

[5]`http://fots.ua.ac.be/events/grabats2008/cases.html`

[6]`http://oryx-editor.org/`

[7]Available from `http://www.zurich.ibm.com/csc/bit/downloads.html`. The 533 models do not include unsound models (e.g., models containing deadlocks), which were manually eliminated prior to the tests.

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: 1480 SEQ components, 544 CHC components, 205 PAR components, 79 SESE Loop components, 41 DAG components, and 38 MEME loop components. All the models in the dataset (or in some cases links to these models) are included in BPStructQoS' Web page. We assigned a random probability value to each choice edge using a uniform distribution and ensuring that for each XOR-split, the sum of the probabilities of its outgoing edges adds up to one. We also assigned random QoS values (for time, cost and reliability) to each component service.

5.2. Verification of accuracy

The QoS aggregation method combines multiple techniques, including a technique for structuring process models (implemented by BPStruct), an algorithm for computing runs and their associated probabilities, as well as Markov chain analysis techniques. One question one might ask is whether the composition of such techniques introduces approximations in the estimation of the QoS that have visible end-effects. Accordingly, we undertook to test the accuracy of the QoS aggregation method by comparing its outputs with those obtained via process simulation. To this end, we simulated each model in the experimental dataset using a process simulator, namely BIMP.[8] Each simulation was composed of 1000 instances created simultaneously. The models given as input to the simulator already contained QoS values for each task (cost and time).[9] These values are the same ones that were used when running BPStructQoS.

At the end of each simulation, we extracted the mean execution time and mean cost per process instance. Each model was simulated four times and the mean cost and execution times obtained during each simulation run were averaged in order to even out the variance created by the stochastic nature of the simulation. The averaged values were compared with those obtained by running BPStructQoS on the same model.

We observed that the difference between the execution time and costs computed by BPStructQoS and those computed via simulation largely coin-

[8]`http://bimp.cs.ut.ee`

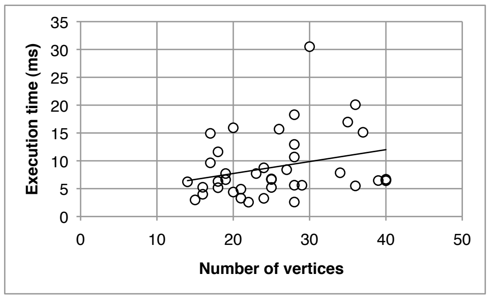
[9]We did not include the "reliability" attribute in these tests since the process simulator does not support this attribute.

cided: the mean percentage deviation was slightly below 0.1% Although not comparable to a formal correctness proof, this result provides some empirical validation of the accuracy of the proposed method.

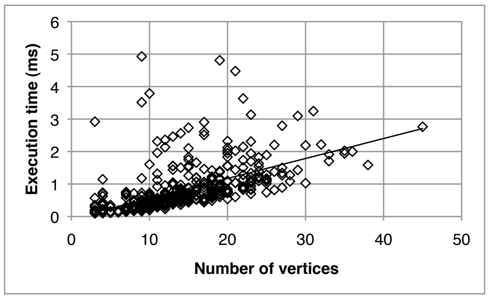
5.3. Performance evaluation

BPStructQoS was able to compute the aggregate QoS of all models in the dataset. During the tests, we measured the execution time for each model, including the time required to compute the RPST and to calculate the QoS. All tests were performed on a laptop with a dual core Intel processor, 2.53 GHz, 4 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 execution times (in milliseconds) are plotted in Figure 14. Figure 14(a) plots the execution times for models that contain at least one DAG component, while Figure 14(b) plots the times for models containing no DAG components. As expected, models containing DAG components incurred a relatively larger execution times. Still, the execution times are in the order of milliseconds even for the largest models, thus showing that the QoS aggregation method can deal with models of realistic size and complexity.



(a) Models with DAGs



(b) Models without DAGs

Figure 14: Scatter plot of execution times for QoS aggregation. Each point in a plot represents one model. The size of the model is given by the x-coordinate of its point while the execution time for the model is given by the y-coordinate. The line shown in each scatter-plot correspond to the linear regression trendline.

6. Related Work

Several previous studies have addressed the problem of aggregating QoS of composite services based on orchestration models. Jaeger *et al.* [7, 8] discuss the QoS aggregation problem for orchestration models consisting of sequence, choice and parallel flow blocks. This approach does not deal with loops. This restriction is lifted by Cardoso *et al.* [6] who proposed a Stochastic Workflow Reduction (SWR) algorithm that takes as input a process graph and computes the expected QoS by repeatedly applying a set of reduction rules for well-structured sequential, parallel, choice and SESE loops (specifically "repeat-until" loops). In a similar vein, Hwang *et al.* [14, 15] represent composite services using a tree structure and compute the aggregate QoS of composite services by traversing the tree using breadth-first search. This tree is similar to the RPST structure, but the trees in the work of Hwang *et al.* do not contain any unstructured blocks (i.e., rigid components). The same QoS aggregation functions for well-structured constructs are given in Canfora *et al.* [3], who use these functions to tackle the problem of binding and re-binding component services to an orchestration model in order to maximize the QoS of the final binding.

Mukherjee *et al.* [9] propose a model to estimate QoS (specifically time and cost) of an orchestration models defined in BPEL. Their method handles the same four well-structured constructs as Cardoso *et al.* Additionally, Mukherjee *et al.*'s method can deal with BPEL "flow" activities containing control links, which are akin to DAG orchestration components in our nomenclature. However, Mukherjee *et al.* do not give details of how the mean execution time of such DAG components is computed. Also, the method of Mukherjee *et al.* does not handle unstructured cycles.

Zheng *et al.* [10] propose yet another model for estimating QoS that can deal with the same four well-structured constructs, plus some forms of multi-exit loops. However, the class of multi-exit loops defined by Zheng *et al.* are not of a general form. Firstly, Zheng *et al.* excludes loops containing AND-split gateways such as the loop shown in Figure 6. Secondly, the loop pattern defined by Zheng *et al.* is such that each node in the loop is connected to one successor in the loop (i.e., the loop is a single cyclic path such that each node in the path can be an exit point of the loop). Hence, overlapping loops are excluded. In contrast, our method can deal with all types of MEME loop components involving only XOR gateways as well as loops involving concurrency, provided that the concurrency can be encapsulated inside child components by means of restructuring. Furthermore, the method of Zheng *et al.* does not cover DAG components.

More recently, Zheng *et al.* [27] extended their previous work to deal with aggregation of QoS attributes defined by means of probability distributions instead of single values. However, this extension still has the same limitations as [10] with respect to the types of components it can handle. An interesting direction for future work is to integrate our QoS aggregation approach with the method for handling QoS probability distributions developed in [27].

The problem of computing QoS for composite services is related to that of QoS-aware service composition [28, 29, 5]. 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.* [5] 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.* [29] 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.* [28]. 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.* [5] 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.

This article is an extended and revised version of a previous conference article [30]. The aggregation method outlined in this previous publication was only able to deal with restricted forms of unstructured loops. Specifically, the loops in [30] were allowed to have multiple exit points, but could not deal with nested or overlapping unstructured loops. This restriction is lifted in this article by means of the mapping to Markov chains. Also, the empirical evaluation was extended to include models from the IBM BIT library.

7. Conclusion

In this article, we proposed a method for computing the QoS of a composite service, given the QoS of the services bound to its orchestration model. Unlike previous work, the proposed method can deal with orchestration models containing complex types of unstructured components, both with and without cycles.

While being more general than previous proposals in the field, the proposed QoS aggregation method is still not complete. Specifically, the method is not applicable when concurrency and conditional branching are intertwined in the same cyclic SESE component. In some cases, the structuring method embodied in BPStruct manages to disentangle concurrency and conditional branching within a cyclic SESE component by factoring out the concurrency into separate SESE components. But in other cases it fails to do so. An example of a model that cannot be handled by the proposed QoS aggregation method and that cannot be further refactored by BPStruct was shown in Figure 9. While such cases are arguably corner-cases and do not seem to appear in practice, developing QoS aggregation methods that are able to handle them is an interesting theoretical question and a possible avenue for future work.

In its current form, the proposed method treats each QoS attribute as being single-valued. It is assumed that the value of a QoS attribute represents an average (e.g., average cost or average execution time). A more general case is one where the QoS is given as a probability distribution. In theory, the proposed method can be extended to deal with aggregation of probability distribution, but this extension would require one to design aggregation functions that determine the probability distribution of a parent component, given those of its child components. Addressing the challenges posed by this extension is another direction for future work.

Acknowledgments

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] C. Peltz, Web services orchestration and choreography, IEEE Computer 36 (10) (2003) 46–52.

[2] L. Zeng, H. Lei, H. Chang, Monitoring the QoS for Web Services, in: Proc. of the 4th International Conference on Service-Oriented Computing (ICSOC), Springer, 2007, pp. 132–144.

[3] G. Canfora, M. Di Penta, R. Esposito, M. L. Villani, A framework for QoS-aware binding and re-binding of composite web services, Systems and Software 81 (10).

[4] C. Becker, H. Kulovits, M. Kraxner, R. Gottardi, A. Rauber, An Extensible Monitoring Framework for Measuring and Evaluating Tool Performance in a Service-Oriented Architecture, in: Proc. 9th Int. Conf. on Web Engineering, Springer, 2009, pp. 221–235.

[5] L. Zeng, B. Benatallah, A. H.H. Ngu, M. Dumas, J. Kalagnanam, H. Chang, QoS-Aware Middleware for Web Services Composition, IEEE Transactions on Software Engineering 30 (5) (2004) 311–327.

[6] J. Cardoso, A. Sheth, J. Miller, J. Arnold, K. Kochut, Quality of Service for Workflows and Web Service Processes, Web Semantics 1 (3) (2004) 281–308.

[7] M. C. Jaeger, G. Rojec-Goldmann, G. Muhl, QoS Aggregation for Web Service Composition using Workflow Patterns, in: Proc. of the Int. Conf. on Enterprise Distributed Object Computing (EDOC), IEEE, 2004, pp. 149–159.

[8] M. C. Jaeger, G. Rojec-Goldmann, G. Muhl, QoS Aggregation in Web Service Compositions, in: Proc. of the IEEE Conf. on E-Commece, E-Services and E-Government (EEE), 2005, pp. 181–185.

[9] D. Mukherjee, P. Jalote, M. Gowri Nanda, Determining QoS of WS-BPEL compositions, in: Proc. 5th Int. Conf. on Service-Oriented Computing (ICSOC), Springer, 2008, pp. 378–393.

[10] H. Zheng, W. Zhao, J. Yang, A. Bouguettaya, QoS analysis for web service composition, in: Proc. of the IEEE International Conference

on Services Computing (SCC 2009), Bangalore, India, IEEE Computer Society, 2009, pp. 235–242.

[11] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, W. M. P. van der Aalst, The prom framework: A new era in process mining tool support, in: Proc. of the 26th International Conference on Applications and Theory of Petri Nets (ICATPN), Miami, USA, Springer, 2005, pp. 444–454.

[12] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, A. P. Barros, Workflow Patterns, Distributed and Parallel Databases 14 (1) (2003) 5–51.

[13] F. Belardinelli, A. Lomuscio, F. Patrizi, Verification of deployed artifact systems via data abstraction, in: Proc. of the 9th International Conference on Service-Oriented Computing (ICSOC), Paphos, Cyprus, Springer, 2011, pp. 142–156.

[14] S.-Y. Hwang, H. Wang, J. Srivastava, R. A. Paul, A Probabilistic QoS Model and Computation Framework for Web Services-Based Workflows, in: Proc. 23rd Int. Conf. on Conceptual Modeling, Springer, 2004, pp. 596–609.

[15] S.-Y. Hwang, H. Wang, J. Tang, J. Srivastava, A probabilistic approach to modeling and estimating the QoS of web-services-based workflows, Information Sciences 177 (23) (2007) 5484–5503.

[16] A. Polyvyanyy, J. Vanhatalo, H. Völzer, Simplified computation and generalization of the refined process structure tree, in: Web Services and Formal Methods (WS-FM), Vol. 6551 of Lecture Notes in Computer Science, Springer, 2010, pp. 25–41.

[17] J. Vanhatalo, H. Völzer, J. Koehler, The Refined Process Structure Tree, Data and Knowledge Engineering 68 (9) (2009) 793–818.

[18] A. Polyvyanyy, L. García-Bañuelos, M. Dumas, Structuring acyclic process models, Information Systems (IS) 37 (6) (2012) 518–538.

[19] A. Polyvyanyy, L. García-Bañuelos, D. Fahland, M. Weske, Maximal structuring of acyclic process models, The Computing Research Repository (CoRR) abs/1108.2384, 2011.

[20] E. Best, R. R. Devillers, A. Kiehn, L. Pomello, Concurrent Bisimulations in Petri Nets, Acta Informatica 28 (3) (1991) 231–264.

[21] B. Kiepuszewski, A. H. M. ter Hofstede, C. Bussler, On Structured Workflow Modelling, in: Proc. 12th Int. Conf. on Advanced Information Systems Engineering, Springer, 2000, pp. 431–445.

[22] S. Perumal, A. Mahanti, Applying Graph Search Techniques for Workflow Verification, in: Proc. of the 40th Annual Hawaii Int. Conf. on System Sciences, 2007, pp. 48–55.

[23] G. Oulsnam, Unravelling Unstructured Programs, Computer Journal 25 (3).

[24] A. Polyvyanyy, Structuring Process Models, Ph.D. thesis, University of Potsdam, Germany (January 2012).

[25] D. Blumenfeld, Operations Research Calculations Handbook, CRC Press, 2012.

[26] R. C. Cheung, A User-Oriented Software Reliability Model, IEEE Transactions on Software Engineering SE-6 (2) (1980) 118–125.

[27] H. Zheng, J. Yang, W. Zhao, A. Bouguettaya, QoS analysis for web service compositions based on probabilistic QoS, in: Proc. of the 9th International Conference on Service-Oriented Computing (ICSOC), Paphos, Cyprus, Springer, 2011, pp. 47–61.

[28] M. Alrifai, T. Risse, Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition, in: Proc. 18th Int. Conf. on World Wide Web, ACM, 2009, pp. 881–890.

[29] Y. Liu, A. H. Ngu, L. Z. Zeng, QoS Computation and Policing in Dynamic Web Service Selection, in: WWW Alt., 2004, pp. 66–73.

[30] M. Dumas, L. García-Bañuelos, A. Polyvyanyy, Y. Yang, L. Zhang, Aggregate quality of service computation for composite services, in: Proc. of the 8th International Conference on Service-Oriented Computing (IC-SOC), San Francisco, CA, USA, 2010, pp. 213–227.