2013

Ontology-Based QoS Aggregation for Composite Web Services

Paul Karaenke
University of Hohenheim, Department of Information Systems 2, Stuttgart, Germany; FZI Forschungszentrum Informatik, Karlsruhe, Germany, paul.karaenke@uni-hohenheim.de

Joerg Leukel
University of Hohenheim, Department of Information Systems 2, Stuttgart, Germany, joerg.leukel@uni-hohenheim.de

Vijayan Sugumaran
Oakland University, School of Business Administration, Rochester, MI, USA; Sogang University, Department of Global Service Management, Seoul, Republic of Korea, sugumara@oakland.edu

Ontology-Based QoS Aggregation for Composite Web Services

Paul Karaenke[1,2], Joerg Leukel[1], and Vijayan Sugumaran[3,4]

[1] University of Hohenheim, Department of Information Systems 2, Stuttgart, Germany
{paul.karaenke,joerg.leukel}@uni-hohenheim.de
[2] FZI Forschungszentrum Informatik, Karlsruhe, Germany
[3] Oakland University, School of Business Administration, Rochester, MI, USA
sugumara@oakland.edu
[4] Sogang University, Department of Global Service Management, Seoul, Republic of Korea

Abstract. Determining the QoS (quality of service) of composite Web services is of high importance for both service providers and service consumers. Heterogeneity of service descriptions, however, often hinders the aggregation of QoS parameters. We propose ontology-based QoS aggregation that integrates the semantics of QoS parameters and their aggregation into the overall aggregation process. The contribution is a QoS aggregation ontology and a QoS aggregation method that uses this ontology. We demonstrate the usefulness of our proposal for designers of composite services and assess its computational efficiency.

Keywords: Ontology, QoS, Service Composition, SOA, Web Service

1 Introduction

Composite Web services play an important role in services computing, since they allow realizing business processes by composing elementary services under a shared workflow and providing this business process "as a service" [1]. An essential task within service composition [2] is determining the quality of service (QoS) of composite services. This task is called QoS aggregation.

QoS aggregation is non-trivial due to the heterogeneity of service descriptions, and in particular its QoS parameters. Heterogeneity is concerned with syntax, i.e., the lexicon for representing QoS, and semantics, i.e., the meaning of a QoS. Existing specifications for service descriptions (e.g., WSDL [3]) and service level agreements (SLAs) (e.g., WS-Agreement [4]) provide constructs for QoS parameters by a small set of attributes including name, data type, and unit of measurement. Service providers may use these constructs to define specific parameters. Whereas these specifications provide a common syntax, the problem of semantic heterogeneity still remains.

Any endeavor to resolve semantic heterogeneity faces the trade-off between standardization and flexibility. Standardization would restrict the use of QoS parameters to a particular set of standard parameters. Whereas aggregating these parameters is easy, the providers must revise their custom service descriptions according to the standard.

11th International Conference on Wirtschaftsinformatik,
27th February – 01st March 2013, Leipzig, Germany

The main disadvantage of this approach is that it assumes the ontological commitment of all the service providers to that standard. This assumption, however, contradicts the idea of loosely coupled Web services in services computing [5]. Flexibility is achieved by giving the service providers full control over their service descriptions. The problem of heterogeneity must then be solved by the services users, who compose services from different providers under a shared workflow.

We aim at balancing the trade-off between standardization and flexibility by maintaining the custom service description and aligning them to an intermediate semantic layer. Unlike current semantic approaches to QoS standardization, we restrict the ontological commitment to one annotation per QoS parameter. Thus, the objectives of this research are to: (1) develop an ontology-based QoS aggregation method for composite Web services with well-formed workflows, and (2) apply this artifact to composite Web service of different complexity to demonstrate its usefulness and assess its computational efficiency. These objectives constitute design science research [6], because it designs and evaluates an artifact (method) that is informed by prior artifacts (composition patterns [14]) and theories (workflow [15], description logic [21]). We evaluate the proposed artifact by rigorously demonstrating its usefulness via a two-level evaluation, which consists of a detailed scenario (descriptive evaluation method) and a simulation using a prototype implementation (experimental evaluation method). The contributions of this research are the QoS aggregation ontology and aggregation method for composite Web services. In our previous work [7], we have presented a preliminary ontology. In this paper, we (1) propose and formalize the aggregation method, (2) revise the ontology, and (3) report the two-level evaluation.

The remainder of this paper is structured as follows. In the next section, we discuss the approaches for QoS aggregation. Subsequently, we provide the basic notions for composite service and QoS. Then, we present the QoS aggregation ontology and the method. We evaluate our proposal through a set of experiments and draw conclusions.

2 Related Work

QoS aggregation has been the subject of much research in services computing. We examine two aspects, heterogeneity of QoS parameters and aggregation methods.
Standardization on the syntactical level provides formats for representing parameters in service descriptions and SLAs. WS-Agreement [4] defines a format for SLAs. The QoS parameters are regarded as domain-dependent; hence WS-Agreement does not define specific parameters and does not solve semantic heterogeneity. Harmonization on the semantic level could overcome this problem, by defining common QoS parameters [8]. Ontologies have been proposed that specify the conceptualization of QoS formally [9-10], though they do not provide information for QoS aggregation.

For QoS aggregation, it would be sufficient to amend the service description with information about aggregation procedures. Haq et al. propose to state aggregation functions for SLA parameters explicitly [11]. These functions are stored in a specific attribute that extends the WS-Agreement specification. The drawback of this approach is service providers need to determine the function for all parameters correctly.

Aggregation functions for QoS parameters are used in [12-13], which stress that the specific parameter sets used are extensible without fundamentally altering the overall approach. This fact sheds light on the important insight to abstracting from diverse and domain-specific QoS parameters. A significant contribution stems from Jaeger et al. [14] who ground QoS aggregation on workflow patterns [15] and deduce so-called composition patterns and respective aggregation functions. Similarly, Cardoso et al. [12] propose a graph reduction algorithm for QoS aggregation.

The rationale for ontology-based QoS aggregation is to integrate constructs from Web services research, in particular, the composition patterns and aggregation functions, into an ontology and an aggregation method. This approach could minimize the additional effort for service providers in annotating their service descriptions as well as increase the flexibility for service consumers to explore a wider range of services.

3 Basic Model

We consider composite service as a collection of services that are governed under a shared workflow (directed acyclic graph, DAG). Figure 1 shows an example workflow, which contains six tasks and four gateways (represented by diamond shapes).

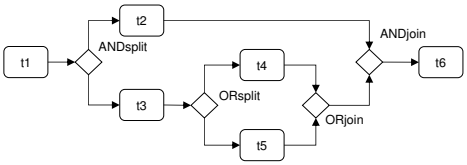

Fig. 1. Example workflow

To instantiate the workflow for different services, the tasks are separated from actual services by means of a binding, which is part of the succeeding definition.

Definition 1 (Composite Service). A composite service is a tuple $CS = (W, B)$ of workflow W and binding B. W is a directed acyclic graph $W = (T, G, A, CG)$, where

- T is a finite non-empty set of tasks $t \in T$, G is a finite set of gateways $g \in G$,
- $N = T \cup G$ is a finite set of nodes, with $T \cap G = \emptyset$,
- A is a set of arcs connecting tasks with tasks, tasks with gateways, and gateways with tasks, thus $A \subseteq (T \times T) \cup (T \times G) \cup (G \times T)$,
- CG is a function which assigns a type to each gateway; $CG: G \rightarrow$ *{XORsplit, XORjoin, ANDsplit, ANDjoin, ORsplit, ORjoin, Loop}*.
- B is a binding function, which assigns a service $s \in S$ to each task, i.e., $B: T \rightarrow S$.

The model is restricted to well-formed (structured) workflows, where W has exactly one start and end node and is weakly connected. The correct usage of the different gateway types, thus how to connect nodes by arcs, is given as follows:

- Each task $t \in T$ has exactly one input arc and one output arc, i.e., $|\bullet t| = |t\bullet| = 1$.
- Each gateway $g' \in G$ with $CG(g') = \{XORsplit, ANDsplit, ORsplit\}$ has exactly one input arc and at least two output arcs, i.e., $|\bullet g'| = 1$ and $|g'\bullet| \geq 2$.
- Each gateway $g' \in G$ with $CG(g') = \{XORjoin, ANDjoin, ORjoin\}$ has at least two input arcs and exactly one output arc, i.e., $|\bullet g'| \geq 2$ and $|g'\bullet| = 1$.
- Each gateway $g' \in G$ with $CG(g') = \{Loop\}$ has one input arc and two output arcs, i.e., $|\bullet g'| = 1$ and $|g'\bullet| = 2$. Let $a_1 = (n_1, g')$ be the input arc, then one output arc a_2 connects back to node n_1 with $a_2 = (g', n_1)$, and one output arc a_3 connects to another node with $a_3 = (g', n_2)$ and $n_1 \neq n_2$.

The *CS* model is not bound to a particular workflow language such as the Business Process Model and Notation (BPMN) and the Web Services Business Process Execution Language (WS-BPEL), but conveys common constructs of workflow languages.

QoS is described in a SLA, which contains QoS information by means of guarantees on parameters; the most common guarantee is that the parameter fits into a given domain of minimum or maximum values, e.g., maximum execution time. The basic model of SLA is made of service, parameters, and parameter values (definition 3). The classification *C* is important for QoS aggregation, because otherwise service parameters of different services cannot be identified as those to be aggregated.

Definition 2 (Composite QoS). *QoS(CS)* is a function that determines the composite QoS for the services *S* of a composite service *CS*.

Definition 3 (SLA). A service level agreement is a tuple $SLA_s = (P, V, C)$. *P* is a set of parameters $p_1, \ldots, p_j$, *V* is a set of values $v_1, \ldots, v_j$ for these parameters, and *C* is a function which assigns a type to each parameter.

4 Ontology-based QoS Aggregation

We present our ontology-based QoS aggregation method by (1) deducing the aggregation ontology from composition patterns and (2) specifying the aggregation algorithm.

4.1 QoS Aggregation Ontology

Rationale. The composite QoS depends on two determinants: QoS parameters and workflow. QoS parameters are diverse with regard to number, name, data type, and conceptualization and rarely adhere to any standard. Instead of contributing to their harmonization, we propose a classification exclusively with regard to their aggregation. This classification is built upon the following principle: If two parameters share the same aggregation function, then they belong to the same parameter type, regardless of other characteristics. Applying this principle results in five parameter types:

- Type 1: Parameters that are always summed up along all deterministic paths of the workflow (e.g., cost of service execution). For non-deterministic paths, the aggregation depends on whether the lower or upper bound is calculated.

- Type 2: Parameters for which the critical path is determined by the maximal values in parallel executions (e.g., duration of execution time).
- Type 3: Parameters that denote a capacity (e.g., throughput).
- Type 4: Parameters that denote a probability (e.g., uptime probability).
- Type 5: Parameters for which the critical path is determined by the minimal values in parallel executions (e.g., key length of the service's encryption algorithm).

The second determinant workflow is analyzed by means of workflow patterns [14]; this analysis arrives at a set of composition patterns *CP = {Sequence, Loop, XORXOR, ANDAND, ANDDISC, ORDISC, ORDISC}*. For instance, the *ORDISC* pattern describes an OR-split followed by OR-join. *ANDDISC* and *ORDISC* describe AND-split and OR-split followed by a m-out-of-n join (discriminator). The latter is used when an activity is triggered after m out of n branches have been completed (e.g., to improve response time, two databases are queried and the first result is processed, the second is ignored) [15]. Both determinants span a matrix of cases, with each cell giving the respective aggregation function. We define the set of all aggregation functions as:

$$X = \{x_{cp,p} \mid cp \in CP \wedge p \in P_s, x_{cp,p} : R^n \rightarrow \mathbb{R}, n \in N^+\} \quad (1)$$

P_S is the set of parameters of service *s*. *n* denotes the number of parameters to be aggregated; the domain of $x_{cp,p}$ consists of *n*-tuples with QoS parameters of the constituent services. Since the aggregation function depends on *(p, cp)*, there exist 5 • 7 = 35 aggregation functions. These assume that the parameters share the same unit of measurement (UoM). The literature provides mature methods for converting UoM [19]. In addition, an OWL ontology for these UoM is available [20].

Several pairs *(p, cp)* share the same aggregation function. Thus, we reduce the number of functions to only seven. We define generic aggregation functions $x \in X$ as shown in Table 1, with $x_1, \ldots, x_n$ denoting the parameter values of the services to be aggregated, and *k* for the number of iterations in a *Loop* pattern. To each pair *(p, cp)*, we assign the respective aggregation function. Following [14], we distinguish upper and lower bound (as shown in Table 2). This distinction is necessary to assess whether a particular parameter value meets the guarantee defined in the SLA (see section 3). Calculating expected or average parameter values is only possible, if the distribution of the non-deterministic control flows (i.e., XORXOR, ANDDISC, ORDISC, and ORDISC) is known [14]; however, this information is not available.

Table 1. Generic aggregation functions.

Aggregation function	Definition	Aggregation function	Definition
x_{sum}	$x_{sum}(x_1,\ldots,x_n) = \sum_{i=1}^{n} x_i$	x_{power}	$x_{power}(x) = x^k$
$x_{product}$	$x_{product}(x_1,\ldots,x_n) = \prod_{i=1}^{n} x_i$	x_{linear}	$x_{linear}(x) = kx$
x_{max}	$x_{max}(x_1,\ldots,x_n) = max(x_1,\ldots,x_n)$	$x_{identity}$	$x_{identity}(x) = x$
x_{min}	$x_{min}(x_1,\ldots,x_n) = min(x_1,\ldots,x_n)$		

Table 2. Aggregation functions for upper and lower bounds of five QoS parameter types.

Type	Bound	Sequence	Loop	XORXOR	ANDAND	ANDDISC	OROR	ORDISC
1	Upper	x_{sum}	x_{linear}	x_{max}	x_{sum}	x_{sum}	x_{sum}	x_{sum}
	Lower	x_{sum}	x_{linear}	x_{min}	x_{sum}	x_{sum}	x_{min}	x_{min}
2	Upper	x_{sum}	x_{linear}	x_{max}	x_{max}	x_{max}	x_{max}	x_{max}
	Lower	x_{sum}	x_{linear}	x_{min}	x_{max}	x_{min}	x_{min}	x_{min}
3	Upper	x_{min}	$x_{identity}$	x_{max}	x_{min}	x_{min}	x_{sum}	x_{sum}
	Lower	x_{min}	$x_{identity}$	x_{min}	x_{min}	x_{min}	x_{min}	x_{min}
4	Upper	$x_{product}$	x_{power}	x_{max}	$x_{product}$	$x_{product}$	x_{max}	x_{max}
	Lower	$x_{product}$	x_{power}	x_{min}	$x_{product}$	$x_{product}$	$x_{product}$	$x_{product}$
5	Upper	x_{min}	$x_{identity}$	x_{max}	x_{min}	x_{min}	x_{max}	x_{max}
	Lower	x_{min}	$x_{identity}$	x_{min}	x_{min}	x_{min}	x_{min}	x_{min}

4.2 Ontology Specification

The ontology formally defines the conceptualization of composition patterns, parameter types, aggregation functions, and their relations. We specify the ontology using description logics (DL), which is a family of formalisms for representing knowledge within a domain. DL provides high expressiveness, while it retains computational completeness and decidability [21]. This logic is also the basis of the Web Ontology Language OWL DL [22]. The methodology for creating the ontology is a rigorous deduction process from the content of table 1 and 2. Syntactical correctness is checked by implementing the ontology in OWL DL and performing standard checks that are built in the OWL editor used. Completeness is guaranteed by covering all the content of table 2.

Figure 2 gives an overview of the concepts and roles. The ontology consists of three concept hierarchies for parameter types *P*, composition patterns *CP*, and aggregation functions *AF*. The actual formula term is represented by *F* and the functional role *hasF*. For each parameter type concept, we add one sub-concept for commonly used parameters (i.e., cost, execution time, throughput, uptime probability, and encryption). These parameters constitute examples only.

The dependencies are expressed by restrictions over the two roles *forP* and *forCP*. Table 3 summarizes these restrictions. For instance, a sequence of *Type1* parameters is aggregated by sum; hence the concept *Sum* is extended by $Sum \equiv \exists forP.Type1 \sqcap \forall forCP.Sequence$. Since *Sum* is valid for more than one pair, the concept definition consists of several pairs of restrictions being concatenated by a logical *OR* (in DL signified by $\sqcup$); i.e., for *Type1* it is valid for *Sequence*, *ANDAND*, and *ANDDISC*, whereas for *Type2* it is valid for *Sequence* only.

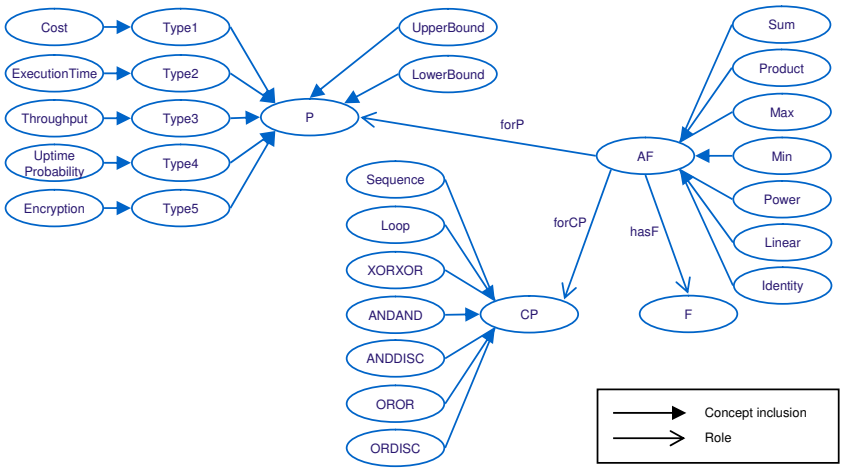

Fig. 2. QoS aggregation ontology (overview)

Table 3. Concept definitions for aggregation functions.

Concept	**Definition**
Sum	$\exists forP.Type1 \sqcap (\forall forCP.(Sequence \sqcup ANDAND \sqcup ANDDISC)) \sqcup (\exists forP.Type2 \sqcap \forall forCP.Sequence) \sqcup (\exists forP.(Type1 \sqcup Type3) \sqcap \forall forCP.(ORDR \sqcup ORDISC) \sqcap (\forall forP.UpperBound))$
Product	$\exists forP.Type4 \sqcap ((\forall forCP.(Sequence \sqcup ANDAND \sqcup ANDDISC)) \sqcup (\forall forCP.(ORDR \sqcup ORDISC)) \sqcap (\forall forP.LowerBound))$
Min	$(\exists forP.(Type1 \sqcup Type2 \sqcup Type3 \sqcup Type5) \sqcap (\forall forCP.(XORXOR \sqcup ORDR \sqcup ORDISC) \sqcap (\forall forP.LowerBound)) \sqcup (\exists forP.(Type3 \sqcup Type5) \sqcap (\forall forCP.(Sequence \sqcup ANDAND \sqcup ANDDISC)) \sqcup \exists forP.Type4 \sqcap \forall forCP.XORXOR \sqcap \forall forP.LowerBound) \sqcup (\exists forP.Type2 \sqcap \forall forCP.ANDDISC \sqcap \forall forP.LowerBound)$
Max	$(\forall forCP.XORXOR \sqcap \forall forP.UpperBound) \sqcup (\exists forP.Type2 \sqcap \forall forCP.ANDAND) \sqcup (\exists forP.Type2 \sqcap \forall forCP.(ANDDISC \sqcup ORDR \sqcup ORDISC) \sqcap \forall forP.UpperBound) \sqcup (\exists forP.(Type4 \sqcup Type5) \sqcap \forall forCP.(ORDR \sqcup ORDISC) \sqcap \forall forP.UpperBound))$
Power	$\exists forP.Type4 \sqcap \forall forCP.Loop$
Linear	$\exists forP.(Type1 \sqcup Type2) \sqcap \forall forCP.Loop$
Identity	$\exists forP.(Type3 \sqcup Type5) \sqcap \forall forCP.Loop$

4.3 QoS Aggregation Algorithm

The aggregation is performed on workflow *W*. Algorithm 1 (Figure 3) is executed for *W*'s start node *ns*. If the workflow begins with an *XOR/AND/ORsplit*, all branches are processed recursively. The aggregated nodes are collected in the set *N'* (lines 01-05). The actual aggregation is performed based on the respective pattern using algorithm 2 (Figure 3) and the result is stored in the join nodes (lines 06-13). These nodes are used by algorithm 1 to detect subsequent *Sequence* and *Loop* patterns (line 14).

Algorithm 1. *QoS(B(n)) = aggregateQoS(n)*
inputs: DAG node *n*
outputs: aggregated QoS for node *n*
01 **if** $n \in \{ORsplit, ANDsplit, ORsplit\}$ **then**
02 $N'=\emptyset$ //nodes to be aggregated
03 **foreach** $n' \in N : (n, n') \in A$: $n' \in T \cup \{XORsplit, ANDsplit, ORsplit\}$
04 $QoS(B(n'))=aggregateQoS(n')$
05 $N'=N' \cup \{n'\}$
06 **if** $n=XORsplit$ **then**
07 $QoS(B(n))=aggregateValues(XORXOR, N')$
08 **else if** $n=ANDsplit$ **then**
09 $QoS(B(n))=aggregateValues(ANDAND, N')$
10 **else** // $n=ORsplit$
11 $QoS(B(n))=aggregateValues(OROR, N')$
12 $n'=getJoinNode(n)$
13 $QoS(B(n'))=QoS(B(n))$ // QoS for join node
14 $n=n'$
15 **if** $\exists n' \in N : (n, n') \in A$: $n' \in T \cup \{XORsplit, ANDsplit, ORsplit\}$ **then**
16 $QoS(B(n))=aggregateValues(Sequence, getSequenceNodes(n))$
17 **if** $\exists n' \in N : (n, n') \in A \wedge n'=Loop$ **then**
18 $QoS(B(n))=aggregateValues(Loop, \{n\})$
19 **return** $QoS(B(n))$

Algorithm 2. *QoS(B(N'))=aggregateValues(cp, N')*
inputs: composition pattern $cp \in CP$, set of DAG nodes N'
outputs: aggregated QoS values of N'
01 **foreach** $p \in P$
02 $myP=createIndividual(p)$
03 $myCP=createIndividual(cp)$
04 $myAF=createIndivudual(AF)$
05 $myAF.setProperty(forP, myP)$
06 $myAF.setProperty(forCP, myCP)$
07 **foreach** $f \ni myAF$
08 **if** $f \sqsubset AF$ **then**
09 **break** //end inner for loop
10 $SV(QoS(B(N')), p) = f(SV(QoS(n_1 \in N'), p), ..., SV(QoS(n_{|N'|} \in N'), p))$
11 **return** $QoS(B(N'))$

Fig. 3. Aggregation algorithms

The existence of a subsequent node in *DAG* results in the detection of *Sequence* patterns (line 15). However, determining the nodes that belong to a sequence is a non-trivial process, as sequence patterns are not made explicit in the *DAG*. Thus, we introduce the function *getSequenceNodes(n)* that performs this task. The function works as follows: For task nodes, non-nested *XORsplit*, *ANDsplit*, *ORsplit*, and subsequent task nodes are collected, whereas *XOR/AND/ORsplit* nodes are recursively aggregated using algorithm 1. For *XOR/AND/ORsplit* nodes, i.e., for the detection of nested *Sequence* patterns in *XORXOR*, *ANDAND*, as well as *XORXOR* patterns, *DAG* is traversed until the corresponding join node is reached for each branch. The nested patterns are also recursively aggregated using algorithm 1. *Loop* nodes indicate multiple executions of the preceding node. Algorithm 1 detects these patterns and performs the aggregation using algorithm 2 for the preceding node (algorithm 1, lines 17-18).

The concrete aggregation (i.e., calculation of the aggregated values) is performed by algorithm 2, which uses the aggregation ontology. For inferring the correct aggregation function, it first creates individuals for the parameter (line 02), composition pattern (line 03), and aggregation function (line 04). *myAF*, the instance of the latter, belongs to the general concept *AF* only, since we do not know the aggregation function yet. Next, we relate these three individuals by the roles *forP* and *forCP* (lines 05-06). Then, the knowledge base is queried for all concept memberships of *myAF*. DL reasoning returns three memberships (loop in line 07): *T* (the top concept to which all individuals belong), *AF* (asserted in line 4), and the specific one denoted by *f*. We use *f* for calculating the parameter value by considering all nodes of the input set *N'* (line 10). The QoS parameter aggregation is performed separately for each parameter (line

01). The function $SV: V \times P \rightarrow V$ reduces a vector of values V to a subvector of parameter values described by P. Finally, we return the aggregated QoS values.

5 Evaluation

We demonstrate the usefulness in a scenario of service composition. We assess the efficiency in a set of experiments using a prototype implementation.

5.1 Usefulness

We consider the following scenario: A designer has defined the workflow as shown in Figure 1. He has selected for each task one service and the workflow has been executed several times successfully. After some time, the execution fails due to the provider of the service for task t2, who is unable to guarantee the agreed service level. Therefore, this service must be replaced by a suitable service. By using a service discovery functionality (which is not subject of our proposal), three candidate services are found. These services have been described using custom parameters as shown in Table 4.

The problem with these services is that their QoS parameters are not aligned to a shared conceptualization. If a particular service is selected for t2, its parameters cannot be aggregated with the parallel service of task t3 and all subsequent tasks in the workflow. A potential solution for this problem would be that the designer rejects the service and asks the service providers to submit a revised service description, which conforms to the standard syntax of the designer; for example, replacing "Price" by "Cost". The disadvantage of this approach is that the service composition process is broken and the designer must wait for the new service description to arrive.

Table 4. Example service offers.

Service provider	Service ID	Parameter	Value
SP1	S1	Cost	0.85
		ExecutionTime	500
SP2	S2	Price	0.75
		ResponseTime	600
	S3	Price	0.70
		ResponseTime	700

By using the proposed aggregation method, there are two options for solving the problem of heterogeneous QoS parameters. First, the designer could take the current service descriptions and add a semantic annotation to each parameter of service S1 through S3. Figure 4 illustrates such an annotation by showing the SLA for S2 specified in the WS-Agreement format. The original SLA offer is then amended by a reference to a concept from the ontology as follows: the attribute *sawsdl:modelReference* contains a mapping of the literal *ResponseTime* to the concept *ExecutionTime* of the ontology which is identified by the namespace *qosns*). Having made this annotation, the service parameters are aligned to the conceptualization that is also used in the

description of the other services in the workflow, in particular the services that are bound to task t3. Therefore, their QoS parameters can be aggregated and thus the stopped process of service composition can be continued. In this sense, the ontology-based aggregation method assists the designer in maintaining workflows and choosing substitutes for failed services. The concrete decision of what service to select, however, depends on the designer's preferences and is not affected by the aggregation method.

```
<wsag:AgreementOffer ...>
  ...
  <wsag:ServiceDescriptionTerm ...
     wsag:ServiceName="S2">
  ...
  <wsla:SLAParameter name="ResponseTime"
     type="wsla:float" unit="ms">
  ...
  </wsag:ServiceDescriptionTerm>
  ...
</wsag:AgreementOffer>
```

```
<wsag:AgreementOffer ...>
  ...
  <wsag:ServiceDescriptionTerm ...
    wsag:ServiceName="S2">
  ...
  <wsla:SLAParameter name="ResponseTime"
    type="wsla:float" unit="ms"
    sawsdl:modelReference="
       &qosns;ExecutionTime">
    ...
  </wsag:ServiceDescriptionTerm>
 ...
</wsag:AgreementOffer>
```

Fig. 4. Example SLA offer by the service provider (left) and with annotation (right)

The second option assumes an ontological commitment of the providers of the services S1 through S3 to the ontology. In this case, each provider still uses his/her own syntax for service parameters, but provides a semantic annotation for each parameter as part of the service description. These annotations are the same as the ones shown in Figure 4. The difference is that annotations are made by the service provider. The annotation process is supported by virtually any XML tool, since the SA-SLA specification provides a standard-conform XML schema for SLA specifications [10].

The advantage of this option is that the process of service composition can be continued immediately by aggregating the QoS parameters under consideration for task t3 and the entire workflow. These aggregates are handed over to a service selection functionality, which selects the best service from a set of services; the selection is either performed by the designer or a component that automates this process. Particular methods for service selection are beyond the scope of our proposal. Existing service selection methods [23] can be combined with our proposal, which complements the selection problem by resolving the heterogeneity of QoS parameters.

5.2 Efficiency

Experimental Setup. We have implemented the proposed aggregation method in a Java prototype application. Workflows are stored in WS-BPEL and SLAs in SA-SLA format. SA-SLA constitutes a combination of WSLA [24] and WS-Agreement for the SLA schema and SAWSDL [25] for semantic annotations. Experiments are performed on Scientific Linux 5.5 on a machine with two Intel Xeon E5630 CPUs (4C, 2.53 GHz) and 16 GB RAM. The prototype is a single-threaded application though.

We study how the proposed aggregation method works for different types of composite services with regard to their complexity and coverage. Complexity is measured

by the following metrics: number of tasks *(|T|)*, number of gateways *(|G|)*, number of arcs *(|A|)*, and number of parameters *(|P|)* in *QoS(CS)*. We consider three workflows *W1*, *W2*, *W3* with 4, 8, and 12 tasks and bind one service to each task. Fig. 5 shows these workflows as BPMN processes. In BPMN, loop is not represented by a gateway node but a modification of the task node. Therefore, the number of gateways in *CS* is 3, 6, and 10, with number of arcs being 10, 19, and 31 respectively.

In the experiments, we instantiate each workflow multiple times for different SLAs. We describe each service by five parameters. The reason for this choice is to have full coverage of the ontology's concepts for parameters, i.e., *type1* to *type5*. Coverage is thus measured by the percentage of the ontology's concepts that the QoS aggregation experiments use. Since we are interested in the effectiveness of the approach, each workflow covers all parameter concepts (5) and composition pattern concepts for *Sequence*, *Loop*, *XORXOR*, and *ANDAND*. This coverage of CPs is higher than in [11] and [16], and equal to [18].

The experiments study two variations. The first variation is concerned with the workflow's complexity. We will study the influence of |*T*| on the time required for QoS aggregation. By setting the range of this variable to {4, 8, 12}, the simulation yields three data sets that provide indications of the system's scalability. The second variation is concerned with the reasoner that is used, since ontology-based systems add computational effort for asserting and retrieving facts. We use the two reference Java ontology frameworks Jena 2.6.4 and OWL API 3.2.2 and three reasoners in the following setup: Jena with OWLMicroReasoner, Jena with Pellet 2.2.2, OWL API with HermiT 1.3.3, OWL API with Pellet 2.2.2, OWL API with FaCT++ 1.5.2.

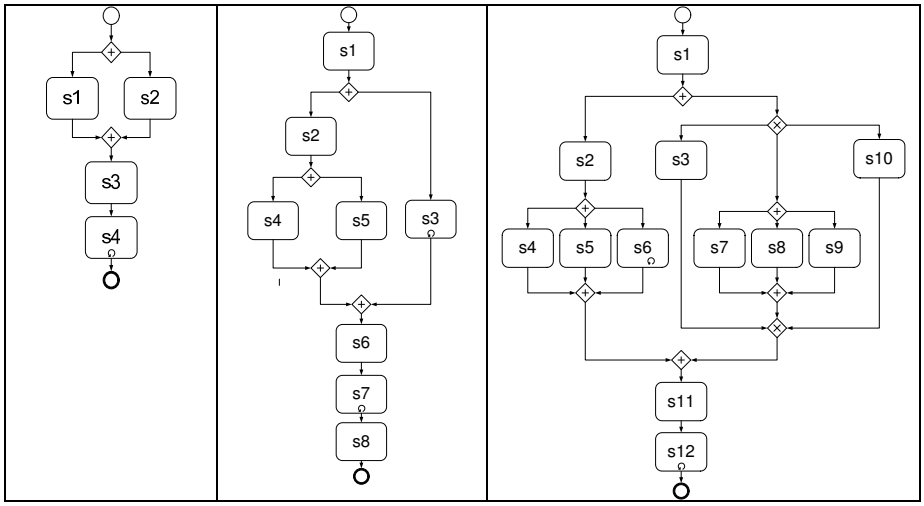

Fig. 5. BPMN workflows *W1* (left), *W2* (middle) and *W3* (right)

We compare the ontology-based method with conventional aggregation, i.e., system with no reasoner, but explicit aggregation functions. This comparison is possible,

because the workflows and SLAs are known prior to execution and have been annotated with aggregation functions exclusively for conventional aggregation.

These two variations span a set of 3 • 6 = 18 experiments. In each experiment, we apply the aggregation method to five different SLAs per service and compute the QoS 50 times (thus 250 runs per experiment). We arrive at 18 • 5 • 50 = 4,500 runs. The SLAs are generated prior to executing the experiments and values are assigned to each parameter (uniform distribution, interval [1,100]). The concrete values do not affect the method's performance, but are used to check the correctness of the composite QoS (effectiveness of the method).

Results. With regard to varying the number of tasks, Figure 6 shows the mean execution time for all 18 experiments. Each of the six configurations suggests a linear computation time. The figure also indicates that the baseline implementation of the aggregation method without reasoning is valid due to its linear complexity.

First, we assess the distribution of execution time in each experiment by using the following distribution properties: mean, median, minimal value (Min), maximal value (Max), standard deviation (STD), and coefficient of variation (CV). Table 5 shows these measures for the largest workflow *W3*. We observe the following: The data points tend to be very close to the mean (CV between 1.7 and 5.2%). Whereas Min and Max span a relatively wide interval, very few data points are close to Min and Max. These findings hold for all reasoner setups.

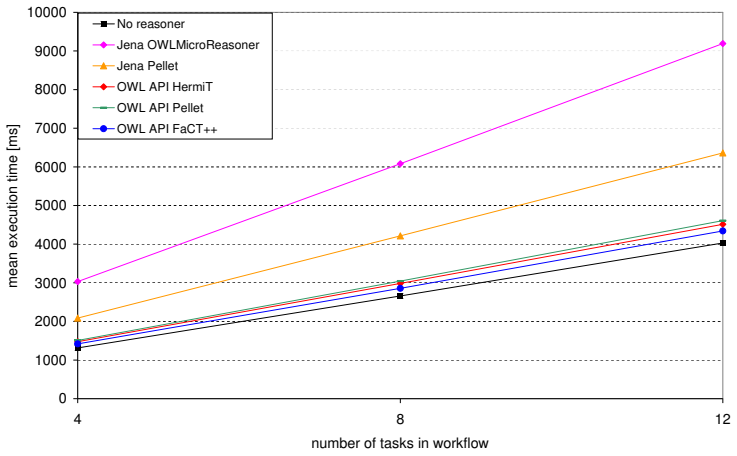

Fig. 6. Mean execution time for six reasoner setups

Second, we assess the efficiency of each setup by comparing their mean execution time to the fastest setup (no reasoner). Table 6 shows the relative increase caused by using the ontology: The increase ranges from [0.097; 0.101] for OWL API FaCT++ to [1.282; 1.1309] for Jena OWLMicroReasoner. These results lead to two important findings: On one hand, all OWL API setups outperform the Jena setups and require between 9.7% and 15.9% additional time compared to no reasoner. This increase is

rather small for an ontology-based method and could be attributed to efficient reasoner systems. On the other hand, the relative additional time does not increase with the number of workflow tasks (as shown by the series *W1*, *W2*, and *W3*). This finding is encouraging for ontology-based QoS aggregation and suggests that the approach is also efficient for workflows of higher complexity.

Table 5. Distribution properties of execution time in ms for workflow *W3*.

Reasoner setup	Mean	Median	Min	Max	STD
No reasoner	4,029	4,003	3,932	6,667	178.91
Jena Micro	9,193	9,157	9,013	13,457	282.81
Jena Pellet	6,363	6,325	6,183	9,062	212.64
OWL API HermiT	4,514	4,486	4,326	6,251	158.97
OWL API Pellet	4,608	4,580	4,508	5,212	87.25
OWL API FaCT++	4,340	4,307	4,231	4,745	85.82

Table 6. Relative increase of mean execution time due to OWL reasoning.

Reasoner setup	W1	W2	W3
Jena Micro	130.9%	128.8%	128.2%
Jena Pellet	60.5%	60.1%	59.5%
OWL API HermiT	13.4%	13.4%	13.2%
OWL API Pellet	15.9%	15.7%	15.5%
OWL API FaCT++	10.1%	9.7%	9.9%

6 Conclusion

We have presented a QoS aggregation ontology and an aggregation method that uses this ontology for composite Web services. We have shown the usefulness of our proposal in a use case scenario of service failure that requires to repair an existing composite service by substituting the failed service. In particular, the approach suggests that designers of composite services have more flexibility in choosing services that have not been aligned to standard QoS parameters. In this sense, the developed artifact helps in balancing the trade-off between standardization and flexibility, which arises from resolving semantic heterogeneity. The assessment of the method's efficiency using the prototype implementation suggests that adding an ontology into the aggregation process increases the computational effort by about 10% for the fastest reasoner setup (OWL API FaCT++). We argue that this increase is compensated by the higher flexibility available for both the service provider and the service consumer.

The results of this work are not limited to a specific domain, since QoS aggregation is a problem in any domain with composite services. The proposed ontology enables the providers to expose their capabilities to potential user, who request a particular service in the course of his/her service composition. Unlike current QoS ontologies, our approach does not rely on a comprehensive semantic description of Web services nor requires commitment to a Web service ontology. Adding annotations to existing SLAs does not affect conformance to the WS-Agreement specification.

The current QoS aggregation ontology contains one example parameter for each parameter type. It is important to note that these parameters represent examples only.

Using the ontology would typically start with linking application- or domain-specific parameters to one parameter type (by making the respective parameter a sub-concept of one of the type-x concepts). In our current work, we could have added many more parameters, however, the only added value is to help the user find the right parameter type. Such an extension of the ontology does not require any modification to the concepts *P*, *CP*, and *AF* including all role restrictions. The ontology, however, is limited to its five parameter types. If a new parameter type is needed, then a new sub-concept of *P* must be defined. In addition, the definitions of all *AF* sub-concepts must be extended with additional role restrictions.

Another limitation of our approach is that it requires all service parameters to be annotated. Even if one annotation is missing or incorrect, the entire QoS aggregation process will fail. This limitation is important in cross-organizational workflows incorporating autonomous service providers. A promising solution to circumvent this drawback is the integration of fault-tolerant or probabilistic [17] QoS aggregation methods into our approach.

Acknowledgments. The work of P. Karaenke and J. Leukel has been supported by the German Federal Ministry of Education and Research (BMBF) under the project InterLogGrid (FKZ 01IG09010E) and by the German Federal Ministry of Economics and Technology (BMWi) under the project MIGRATE! (FKZ 01ME11052). The work of V. Sugumaran has been supported by Sogang Business School's World Class University Program (R31-20002) funded by Korea Research Foundation and the Sogang University Research Grant of 2011.

References

1. Dustdar, S.: Web Services Workflows – Composition, Co-Ordination, and Transactions in Service-Oriented Computing. Concurrent Eng. Res. A. 12, 237–245 (2004)
2. Ardagna, D., Pernici, B.: Adaptive Service Composition in Flexible Processes. IEEE T. Software Eng. 33, 369–384 (2007)
3. W3C: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C Recommendation 26 June 2007, http://www.w3.org/TR/wsdl20
4. Open Grid Forum: Web Services Agreement Specification (WS-Agreement). OGF Proposed Recommendation, http://www.ogf.org/documents/GFD.107.pdf
5. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. IEEE Computer 40, 38–45 (2007)
6. Hevner, A., March, S., Park, J., Ram, S.: Design science in Information Systems Research. MIS Quarterly 28, 75–105 (2004)
7. Karaenke, P., Leukel, J.: Towards Ontology-based QoS Aggregation for Composite Web Services. In: Fähnrich, K.-P., Franczyk, B. (eds.): Informatik 2010. LNI, Vol. 175, pp. 120-125. GI, Bonn (2010)
8. O'Sullivan, J., Edmond, D., ter Hofstede, A.H.M.: What's in a Service? Distrib. Parallel Dat. 12, 117–133 (2002)
9. Dobson, G., Lock, R., Sommerville, I.: QoSOnt: A QoS Ontology for Service-Centric Systems. In: 31st EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 80–87. IEEE Press, New York (2005)

10. Muñoz Frutos, H., Kotsiopoulos, I., Vaquero Gonzalez, L., Rodero Merino, L: Enhancing Service Selection by Semantic QoS. In: Aroyo, L. et al. (eds.): ESWC 2009. LNCS, Vol. 5554, pp. 565–577. Springer, Heidelberg (2009)
11. Ul Haq, I., Huqqani, A., Schikuta, E.: Hierarchical aggregation of Service Level Agreements. Data Knowl. Eng. 70 (5), 435-447 (2011)
12. Cardoso, J., Sheth, A.P., Miller, J.A., Arnold, J., Kochut, K.: Quality of service for workflows and web service processes. J. Web Semant. 1, 281–308 (2004)
13. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. IEEE Transactions on Software Eng. 30, 311–327 (2004)
14. Jaeger, M.C, Rojec-Goldmann, G., Mühl, G.: QoS Aggregation for Web Service Composition using Workflow Patterns. In: 8th IEEE Enterprise Distributed Object Computing Conference (EDOC 2004), pp. 149–159. IEEE Press, New York (2004)
15. van der Aalst, W., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases 14 (1), 5–51 (2003)
16. Canfora, G., Di Penta, M., Esposito, R., Perfetto, F., Villani, M.L.: A framework for QoS-aware binding and re-binding of composite web services. J. Syst. Software 81, 1754–1769 (2008)
17. Hwang, S.-Y., Wang, H., Tang, J., Srivastava, J.A.: Probabilistic approach to modeling and estimating the QoS of web-services-based workflows. Inform. Sciences 177, 5484–5503 (2007)
18. Unger, T., Mauchart, S., Leymann, F., Scheibler, T.: Aggregation of Service Level Agreements in the Context of Business Processes. In: 12th IEEE Enterprise Distributed Object Conference (EDOC 2008), pp. 43–52. IEEE Press, New York (2008)
19. Beaty, H.W.: Units, Symbols, Constants, Definitions, and Conversion Factors. In: Fink, D.G., Beaty, H.W. (eds.): Standard Handbook for Electrical Engineers. McGraw-Hill Professional, New York (2006)
20. Hodgson, R., Keller, P.J.: QUDT – Quantities, Units, Dimensions and Data Types in OWL and XML. http://www.qudt.org (2011)
21. Baader, F., Horrocks, I., Sattler, U.: Description Logic. In: Harmelen, F. van, Lifschitz, V. (eds.): Handbook of Knowledge Representation. Elsevier, Amsterdam (2007)
22. W3C: OWL Web Ontology Language. W3C Recommendation 10 February 2004, http://www.w3.org/TR/owl-ref
23. Alrifai, M., Risse, T.: Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition. In: 18th International Conference on World Wide Web (WWW 09), pp. 881-890. ACM, New York (2009)
24. Keller, A., Ludwig, H.: The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. J. Netw. Syst. Manag. 11, 57–81 (2003)
25. Kopecky, J., Vitvar, T., Bournez, C., Farrell, J.: SAWSDL: Semantic Annotations for WSDL and XML Schema. IEEE Internet Comput. 11, 60–67 (2007)