onQoS-QL: A Query Language for QoS-Based Service Selection and Ranking

Giuseppe Damiano, Ester Giallonardo, and Eugenio Zimeo

Research Centre on Software Technology
Department of Engineering
University of Sannio
Benevento, 82100 Italy
gppdamy@virgilio.it, ester.giallonardo@unisannio.it,
zimeo@unisannio.it

Abstract. Although service consumers need to communicate effectively their quality of service requests, today a standard QoS query language has not yet been defined. This paper proposes a query language, named onQoS-QL, to properly capture QoS requirements. It is based on the onQoS ontology and on a set of statements that the user adopts to express the desiderata subjective or context-aware constraints on QoS measurable values. Currently, the language is based on SPARQL to implement service retrieval and selection through a SPARQL-engine integrated in our discovery engine that is able to rank the selected services according to user requirements and specifications. Some preliminary tests show the correctness and the power of the proposed approach.

Keywords: Metrics, Ontology, Quality of Service, Semantic Web, Service Discovery.

1 Introduction

Discovery engines are key architectural components of Service Oriented Architecture (SOA). Service requestors use them to properly select the services to bind to a Web process by specifying both functional ("what the service does") and non-functional ("how the service is supported") requirements.

Accessing a growing number of Web Services requires more and more sophisticated discovery engines that extend the basic capabilities of UDDI registries in order to find the set of services that satisfy user requests and to select one or more of them that better meet the requirements [1].

Although functional requirements allows for building a desired business process by composing existing functionalities, non-functional requirements are important to reduce the search space during the discovery process by identifying only those services that guarantee the desired level of quality of service (QoS). The role of these requirements is much more important if we consider the dynamic nature of business processes, especially in the Web environment. Here QoS constraints can be usefully adopted to autonomously handle Web processes at run-time, dynamically guiding the binding of discovered services and re-binding them in presence of failures.

E. Di Nitto and M. Ripeanu (Eds.): ICSOC 2007 Workshops, LNCS 4907, pp. 115–127, 2009.

QoS is defined in ISO 8402 as: "The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs". This definition clearly highlights the necessity of defining an unambiguous understanding of QoS concepts and their relations in order to establish valid agreements between providers and consumers with limited user intervention, even at run-time. To this end, in a previous work [3], we have proposed an ontological-based approach (onQoS) that provides a QoS Description Model, which helps to eliminate ambiguities during the QoS Discovery Process, when the same ontology is shared between providers and consumers.

However, a correct matching of user desiderata with published onQoS service descriptions requires a template description prepared by the requestor, reporting the features of the desiderated service. This approach presents three main problems: (1) it is hard and cumbersome to define template services; (2) templates are not sufficiently expressive to capture user desiderata; (3) service ranking is often subjective and needs to specify user-centric utility functions. In this work we face and overcome these expressivity problems exploiting onQoS to define a query language, called in the following *onQoS-QL* (Query Language about QoS), which we use to define complex queries on QoS constraints (related to both functional and non-functional service features).

With the help of onQoS-QL, service consumers can express more effectively their QoS requests, so better formalizing their real intentions. In fact, these measures often depend on subjective expectations of the service or on context information and vary with the type of service and where it is used. Therefore, providing clients with a way to clearly and formally define QoS criteria and complex utility functions, the QoS discovery engine will be able to select automatically the proper service, reasoning not only on the QoS shared knowledge but also ranking the services according to the requestor criteria.

The definition of onQoS-QL gave us also the opportunity of re-factoring onQoS by introducing some new concepts that enable onQoS to better explicit the concept of measurement process of QoS attributes, which represents a key element in a dynamic market of networked services where monitoring and reputation drive the evolution.

The remainder of this paper is structured as follows. Section 2 describes the related works. Section 3 shows the main concepts of the onQoS ontology. Section 4 describes the onQoS-QL query language. Section 5 presents the onQoS-QL reasoner. Section 6 discusses the results of our study. Finally, Section 7 summarizes the conclusions and highlights future works.

2 Related Works

Queries used during service discovery have a direct influence on the precision and recall of the results, since they are the means for the requestor to explicit the knowledge about its intention. Hence, to improve the quality of service matching, the design of a specific QoS query language could be useful, as demonstrated by the focus on query and rule layers in the context of Semantic Web [12-14].

OWL-QL [13] has a restricted access to the TBox so that only named classes and individuals can be retrieved. Utilizing this formal language, values of variables cannot

be filtered and no ranking can be produced. On the other hand SPARQL [12], the W3C's proposed query language to access RDF and RDF schema information, is able to retrieve from a pool the services that satisfy a query. Constraints on the retrieved solutions can be expressed by the keyword FILTER. In this way a sorting on a single attribute can be implemented, but no ranking on aggregated variables is possible. So it can be mainly used as a basis for building other more complex query languages.

Liu, Ngu and Zeng in [16] presented a QoS computation model for web services selection. They utilize weights, groups of parameters, preferred directions on QoS values (e.g. the highest or the lowest ones) in order to specify the client demand. However, the model does not provide support for expressing logical or value constraints on QoS parameters.

The definition of a language for aiding service discovery and ranking requires to be founded on a stable and well-organized ontology. The research on this field has produced many proposals in recent years that we briefly overviewed to establish our reference QoS ontology before defining a new language.

The DAML-QoS ontology [5] presents the following three levels. The QoS profile layer for matching purpose; the QoS property definition layer for defining domain and range constraints on QoS features; the metrics layer for specifying QoS metrics. DAML-QoS presents also a QoS vocabulary, called Basic Profile. The approach allows a service requestor to define QoS constraints in the QoS profile layer, but the authors utilize incorrectly the OWL cardinality construct to impose bounds on QoS parameter values.

The scope of the WS QoS ontology [6] is QoS service discovery. The *QoSInfo* element describes server performance and protocols required for providing QoS features; the *WSQoSOntology* element is used to define QoS parameters and their protocols; instead the *QoSOfferDefinition* element contains one or more *QoSInfo* elements. The implemented WS-QoS Editor allows both requestor and provider to easily edit their QoS requirements or offers without knowing the details of the WS-QoS XML Schema. One or more XML-based files are generated automatically. QoSOnt [7] has been designed in a modular way with the scope of developing service centric systems. The QoSOnt basic layer contains generic concepts relevant to QoS as for example unit ontologies. The QoSOnt attribute layer defines particular QoS attributes and their metrics. The last layer contains domain ontologies.

QoS Ontology [8] captures QoS vocabulary and the basic QoS concepts in the upper ontology. While in the middle ontology, QoS features on distribute systems are specified. The implemented SQRM tool provides graphical means to specifying QoS requirements.

The analysis of the literature showed that the semantics of query operators and of query measurement processes have not yet been formalized in any existing QoS ontology. For this reason we worked to define onQoS [3] and to refine it in this paper in order to better formalize the concept of *MeasurementProcess*.

3 onQoS Ontology

By reasoning on QoS Discovery, we identified as key aspect the importance of delivering measurable values to Service Consumers for advanced monitoring and process

handling. Such values are instances of concepts of a QoS Description Model that uses QoS metrics to share QoS knowledge between providers and consumer. The ontological concepts derive from the analysis of process measurement and from its employment in the specification of each QoS entity.

The onQoS [3] ontology (óvQoS, i.e. QoS entity, ὄν in the early Greek, part. of εἶναι: to be) is able to support the reasoning power required to navigate the QoS terminology correctly and efficiently and it is able to formalize QoS knowledge, i.e. QoS parameters and their relationships. We utilized the ontology as the means to resolve terminology mismatches between the vocabularies, misunderstandings at message level, unsuccessful couplings between measurement processes, scales and units of measurement and as the means to derive knowledge utilizing the domain-dependent functions that can be specified among QoS metrics [3].

Fig. 1 shows the root concepts of onQoS according to the OntoShere global view approach [15]. It presents a big earth-like sphere bearing on its surface the main onQoS concepts represented as small spheres. Atomic nodes are smaller and in dark colour. The scene shows the main direct semantic relations between the concepts. *QoSParameter* is a (measurable or quantifiable) QoS characteristic or feature. *QoSMetric* is a type of measurement tied to a QoS parameter. *MeasurementProcess* is the process by which numbers or symbols are assigned to QoS parameters according to clearly defined rules. *Scale* specifies the nature of the relationships among a set of values (*ScaleValue*). *ScaleValue* is a number or symbol that identifies a category in which the QoS parameters can be placed. The *Participant* identifies the resource that performs the measurement process. The onQoS *Profile* describes a QoS policy through the definition of one or more QoS metrics.

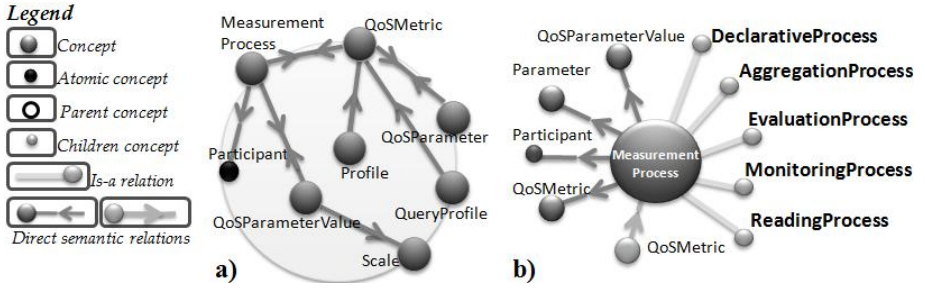

Fig. 1. a) The main concepts of onQoS. **b)** The *MeasurementProcess* concept.

The *QueryProfile* is a particular *Profile* that presents a unique QoS metric relating to the overall QoS. *QueryProfile* presents the single property *hasQoSMetric* on the range of *WSQoSMetric. WSQoSMetric* is a metric relating to the overall QoS. We associate a *Profile* with a service advertisement, while a *QueryProfile* describes the QoS profile requested by a user. *MeasurementProcess* presents the specializations *DeclarativeProcess*, *AggregationProcess*, *EvaluationProcess*, *MonitoringProcess* and *ReadingProcess* as we can see in the image b) of Fig. 1. The *DeclarativeProcess* is a direct measurement process that enables statement on QoS parameter values. The *AggregationProcess* is an indirect measurement process that reduces a set of measures

into a unique representative one, making a synthesis of different aspects. The *EvaluationProcess* is the process whose objective is to evaluate QoS constraints. These can be expressed utilizing different predicates, i.e. a specialization of the evaluation process (in our context), such as the comparison operators (<, >, =, !=, …) that operate on different scales. The *ReadingProcess* is a dummy measurement process that simply reads the QoS values. Instead, the *MonitoringProcess* is the process whose objective is to supervise the QoS.

4 onQoS-QL: A Query Language on QoS Attributes

onQoS-QL is a QoS query language for Web Service selection based on the knowledge represented in onQoS and its lower ontologies, i.e. the elements it defines are interpreted utilizing the onQoS semantics and its own domain specializations. The language gives users a way to express in a query some subjective measurement processes to better evaluate a service. Consequently, the language was designed also to be extended in a particular domain, to let users defining personalized and contextualized ways to retrieve and rank services.

The *WSQoSMetric* concept in onQoS ontology represents the main building block for query formulation. This metric measures the degree of compatibility between two descriptions on a scale that defines an ordinal relation. User formulates a query through several instances of *WSQoSMetric* organized in hierarchical manner and according to a logical/arithmetic expression. This concept has the following restrictions:

$$\forall\ hasMeasurementProcess\ .\ (AggregationProcess \sqcup EvaluationProcess) \quad (1)$$

$$\forall\ measuredParameter\ .\ WSQoS \quad (2)$$

An instance of it can be defined only if the measurement process is an instance of *AggregationProcess* or *EvaluationProcess* (1) and the measured parameter is the *WSQoS* as stated in (2). Therefore, an instance of *WSQoS* is functionally linked with an instance of *WSQoSMetric* by means of the object property *isMeasuredBy* inverse of *measuredParameter*. Indeed, the scope of a query is to capture the way in which users want to measure the quality of a Web service, represented by the class *WSQoS*, evaluating constraints defined on selected QoS parameters and aggregating these partial results. Aggregation can be performed only if a reference measurement scale exists for any value involved in this process and according to the operations it defines. That scale is represented by means of the class *WSQoSScale* and is a numeric and totally ordered scale whose values span in the range [0, 1] of the real numbers. The object property *isMadeUpOf* of the scale is restricted in a way that any values of it belong to the range *{hasWSQoSScaleValue}*. The data property *hasWSQoSScaleValue* links a float value in [0, 1] with a variable on the scale *WSQoSScale*. Formally, a variable whose values belong to *WSQoSScale* is represented with the class *WSQoSValue*, subclass of *NumericValue*. *WSQoSValue* also, defines the data property *hasWeight* that can be used to weight a QoS parameter constraint evaluation or aggregation result as context of the ranking function specified in the query. Therefore, an instance of QueryProfile, which represents a user's query by means of an expression

of user selected QoS parameters, defines other than the Profile instances to retrieve, the function to rank the result set.

Following a bottom-up approach to formulate a query, a user defines a *DeclarativeMetric* or *ReadingMetric* metric for every selected QoS parameter, evaluates the measured values by means of a predicate defined on the same measurement scale of the selected parameters and aggregates these partial measurements by means of a function defined on the *WSQoSScale*. Predicates and functions are measurement processes of *WSQoSMetric* and define how to measure (partial value of) the *WSQoS* parameters. The composition of evaluation and aggregation measurement process is driven by the compatibility of the input/output measured variables. onQoS defines a variable for each measurement scale, also for the *WSQoSScale*. As Fig. 2.a) illustrates, the object property *hasMeasuredValue* inherited from *MeasurementProcess* by the class *Predicate* has a range restriction on class *WSQoSValue*. The properties *hasFirstArgument* and *hasSecondArgument* specializing *hasParameter* of *MeasurementProcess* identify the arguments of a non-commutative predicate and have restrictions on subclass of *QoSParameterValue* corresponding with the scale (*NominalScale*, *OrdinalScale* or *RatioScale*) for which predicates are defined.

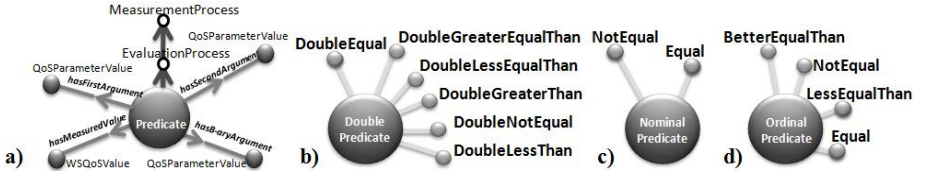

Fig. 2. a) Predicates for **b)** RatioScale, **c)** NominalScale and **d)** OrdinalScale

As we said, a query defines how to measure from a user point of view the whole quality of a Web service. Fig. 3 shows some aggregation functions (*WeightedMean*, *WSQoSOr* and *WSQoSAnd*) that let a user to aggregate partial Web Services QoS evaluations to specify the wished measure in an expression.

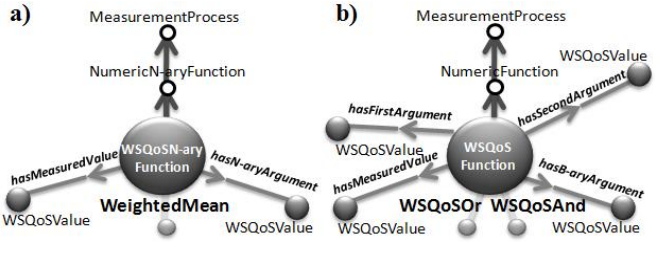

Fig. 3. a) *WSQoSN-aryFunction* and **b)** *WSQoSFunction* with their specializations

As we have stated in section 2, no language has yet defined to formulate a query as a set of statements to retrieve and rank Web Services satisfying user defined QoS requirements. onQoS-QL tries to fill up that gap giving the user a set of operators that merge retrieval constrains and ranking directives. As such, predicates and some

aggregation functions have extended semantics to support both retrieval and ranking phases. As example, *DoubleLessThan* predicate (Fig. 2.a) defined for the ratio scale *DoubleRatioScale* not only evaluates whether the first operand is smaller than (<) the second one but, if that condition is satisfied, also a value in]0, 1] is computed as evaluation on the *WSQoSScale*.

Table 1 shows the main predicates and aggregation functions with their semantics. The first two columns report the operators of the language and the operands with their measurement scales. The second two columns report retrieval and ranking semantics of each operator. The former means a service profile will be retrieved if and only if the logical condition it states is satisfied. The latter indicates the rank value of each satisfied operator, according to the mathematical expression provided in the table.

Table 1. Main predicates and aggregation functions with their semantics

Vocabulary Term	Measurement Scale of Arguments	Retrieval Semantics	Ranking Semantics
Equal(x_i, x_j)	NominalScale $\{x_i\}_{i=1...N}$	$i = j$	1
NotEqual(x_i, x_j)		$i \neq j$	1
BetterEqualThan(x_i, x_j)	OrdinalScale $\{x_i \mid x_i < x_j \Leftrightarrow i < j\}_{i=1...N}$	$i \in \{j,...,N\}$	$\frac{1}{N}(1+i-j)$
LessEqualThan(x_i, x_j)		$i \in \{1,...,j\}$	$\frac{1}{N}(1+j-i)$
DoubleLessThan(x, y)	DoubleRatioScale $[X_{inf}, X_{sup}]$	$x < y$	$\frac{2}{1+e^{-\frac{y-x}{k}}} - 1$
DoubleGreaterThan(x, y)		$x > y$	$\frac{2}{1+e^{-\frac{x-y}{k}}} - 1$
WSQoSAnd(x, y)	WSQoSScale $\{p \equiv (x,w) \mid x \in [0,1] \wedge w \in [0, X_{sup}]\}$	$x \wedge y$	$min(x, y)$
WSQoSOr(x, y)		$x \vee y$	$max(x, y)$
WeightedMean$(p_i)_{i=1,...,N}$		$\exists p_i$, $i = 1,...,N$	$\sum_{i=1}^{N} w_i x_i \Big/ \sum_{i=1}^{N} w_i$

4.1 Query Examples

To explain in details the query formulation process, let us introduce a short notation for a QoS metric:

$$QoSMetric = < QoSParameter, MeasurementProcess, Scale, ?QoSParameterValue > \quad (3)$$

where *?QoSParameterValue* is the measured variable as the prefixed symbol "?" shows. By means of the *MeasurementProcess* it is possible to measure a particular *QoSParameter* whose values belong to the Scale as the metric defines.

Our first example deals with some computer network QoS parameters such as jitter and round trip time (*RTT*):

$$?WSQoSValue = ?RTTValue <_{WSQoS}\ k_1 \vee_{WSQoS}\ ?JitterValue \leq_{WSQoS}\ k_2 \quad (4)$$

where operators $<_{WSQoS}$, $\vee_{WSQoS}$ and $\leq_{WSQoS}$ have an extended semantics according to Table 1, where constants k_1 and k_2 belong to the same measurement scale of ?*RTTValue* and *?JitterValue*. To define the metrics for the values in (4), we use the notation

ntroduced in (3). The declarative metrics *RTTMetric* = <*RTT, RTTProcess, DoubleScale, ?RTTValue*> and *JitterMetric* = <*Jitter, JitterProcess, DoubleScale, ?JitterValue*> are defined to measure *?RTTValue* = *RTTProcess()* and *?JitterValue* = *JitterProcess()*. Hence, to evaluate expression (4) a bottom-up approach is followed, evaluating firstly the inner expressions. So, the partial *WSQoS* evaluation metrics *WSQoSRTTEval* = <*WSQoSRTT, DoubleLessThan, WSQoSScale, ?WSQoSRTTValue*> and *WSQoSJitterEval* = <*WSQoSJitter, DoubleLessEqualThan, WSQoSScale, ?WSQoSJitterValue*> are defined to measure *?WSQoSRTTValue* = *DoubleLessThan(RTTProcess(), k_1)* and *?WSQoSJitterValue* = *DoubleLessEqualThan(JitterProcess(), k_2)*. The overall *WSQoS* aggregation metric *WSQoSMetric* = <*WSQoS, WSQoSOr, WSQoSScale, ?WSQoSValue*> is defined to measure *?WSQoSValue* = *WSQoSOr(?WSQoSRTTValue, ?WSQoSJitterValue)*.

Another example comes from the automotive domain where we have specialized onQoS and have extended onQoS-QL with some new operators such as a "quantizer" evaluation function. In (5) we show such a query whose *?WSQoSValue* is the result of a weighted mean computation over some QoS parameter measurements and evaluations in the automotive domain.

$$\frac{w_{C0}\left\lceil\frac{\sum_{i=1}^{F_{C0}}?C0_i}{F_{C0}}\right\rceil + w_{C1}\left\lceil\frac{\sum_{i=1}^{F_{C1}}?C1_i}{F_{C1}}\right\rceil + w_{C2}\left\lceil\frac{\sum_{i=1}^{F_{C2}}?C2_i}{F_{C2}}\right\rceil + w_{Rules}\left\lceil\frac{\sum_{i=1}^{F_{Rules}}?Rules_i}{F_{Rules}}\right\rceil + w_{Sched}\left\lceil\frac{\sum_{i=1}^{F_{Sched}}?Sched_i}{F_{Sched}}\right\rceil + w_{Std}\lceil?Sdt\rceil}{w_{C0}+w_{C1}+w_{C2}+w_{Rules}+w_{Sched}+w_{Std}} \quad (5)$$

Evaluations are performed by means of the "quantizer", a function whose output is computed through thresholds comparison and indicated in (5) with the symbol $\lceil\cdot\rceil$.

The query in (5) can be regarded as a utility function that the matchmaker will maximize over the whole service space. QoS parameters are *C0*, *C1* and *C2* dealing with the quality of geometric integrity of CAD design for the contact, tangency and curvature continuity between patches (contiguity surfaces): these parameters measure (in percentage) how many patches are not compliant with a fixed tolerance; Rules deals with design rules adherence of CAD modeling; Sched is for the scheduling constraints satisfaction; finally, *Std* deals with supported automotive standard. For the query in (5) we have to define the following metrics.

The reading metrics *CxMetric* = <*Cx, CxProcess, DoubleScale, ?Cx*>, *RulesMetric* = <*Rules, RulesProcess, DoubleScale, ?Rules*> and *SchedMetric* = <*Sched, SchedProcess, DoubleScale, ?Sched*> and a declarative metric *StdMetric* = <*Std, StdProcess, IntScale, ?Std*>. The role of reading and monitoring metrics in query-answering is illustrated in Fig. 4. The monitoring components are configured by means of the monitoring metrics to collect measures about dynamic QoS parameters. For each monitoring metric an onQoS domain ontology defines also a reading metric utilized by users to formulate their queries. A reading metric defines its measurement frame, i.e. the number of samples the matchmaker can read from the requestor's (or provider's) monitoring component. So, the derived metrics *CxDerivedMetric* = <*CxMean, ArithmeticMean, DoubleScale, ?CxMean*>, *RulesDerivedMetric* = <*RulesMean, ArithmeticMean, DoubleScale, ?RulesMean*> and *SchedDerivedMetric* = <*SchedMean, ArithmeticMean, DoubleScale, ?SchedMean*> are defined to compute a synthetic value (arithmetic mean) for each requestor's monitored QoS parameter. The partial WSQoS evaluation metrics *CxWSQoSEvalMetric* = <*CxWSQoS,* $\lceil\cdot\rceil$*, DoubleScale, ?CxWSQoS*>, *RulesWSQoSEvalMetric*

= <*RulesWSQoS*, $\lceil \cdot \rceil$, *DoubleScale*, *?RulesWSQoS*>, *SchedWSQoSEvalMetric* = <*SchedWSQoS*, $\lceil \cdot \rceil$, *DoubleScale*, *?SchedWSQoS*> and *StdWSQoSEvalMetric* = <*StdWSQoS*, $\lceil \cdot \rceil$, *DoubleScale*, *?StdWSQoS*> are defined to evaluate each QoS parameter value on the *WSQoSScale*. Finally, the overall WSQoS aggregation metric *WSQoSMetric* = <*WSQoS*, *WeightedMean*, *WSQoSScale*, *?WSQoSValue*> is defined to measure *?WSQoSValue* according to the expression in (5).

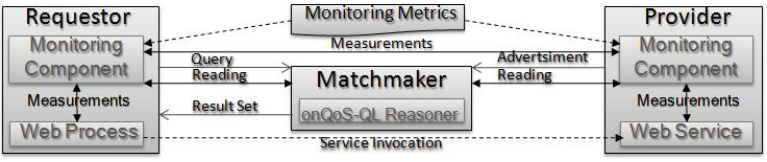

Fig. 4. Role of monitoring and reading metrics in query-answering

5 onQoS-QL Reasoner

The sample expressions presented above and the metrics used are analyzed by the onQoS-QL reasoner that supports onQoS-QL query-answering by using the reasoning capabilities of the open source DL reasoner Pellet [9] and the framework Jena [10]. Fig. 5 shows the onQoS-QL Reasoner architecture. The main component is the *onQoS-QL Engine* that manages and controls the *SPAR-QL Engine* and the *Ranking Engine*.

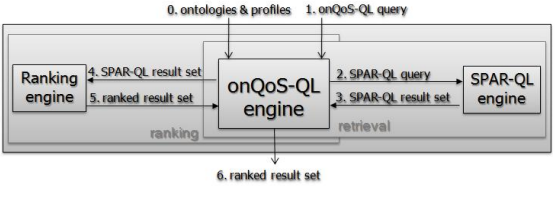

Fig. 5. onQoS-QL Reasoner Architecture

The former selects from a pool the services that satisfy the query whereas the latter establishes a ranking among the retrieved services. To this end, the onQoS-QL engine rewrites the onQoS-QL query in a SPARQL one in order to give it in input to the SPAR-QL engine. The result set obtained from the SPAR-QL engine can be ranked according to the matching process defined in the query.

The onQoS-QL reasoner presents the following interface:

- *loadOntology (String uri)*: loads in the knowledge-base the onQoS-based domain ontology;
- *loadProfile(Profile p)*: loads in the knowledge-base a domain ontology compliant target profile;
- *executeQuery(QueryProfile q, Boolean rank)*: retrieves the profiles that satisfies the input query. The argument *rank* enables or disables the ranking process.

5.1 Profiles Retrieval

As first step to reply to a user's query, the onQoS-QL engine interacts with the SPAR-QL engine to retrieve from its knowledge-base the Profile instances satisfying the query according to the retrieval semantics explained in Table 1. As such, the onQoS-QL engine rewrites dynamically the submitted onQoS-QL query in SPAR-QL, with two main sections: an OPTIONAL blocks sequence and a FILTER statement. Indeed, each declarative or reading metric is rewritten as an OPTIONAL block and each constraint (or predicate) for a QoS parameter value and some aggregation functions have a corresponding piece in the FILTER statement.

Example of SPAR-QL rewriting of the query shown in (4).

```
SELECT DISTINCT ?profile WHERE { ?profile rdf:type onQoS:Profile.
                OPTIONAL{
                          ?profile onQoS:hasQoSMetric ?metricRTT.
                          ?metricRTT onQoS:measuredParameter ?mparRTT.
                          ?metricRTT onQoS:hasMeasurementProcess ?mpRTT.
                          ?mpRTT onQoS:hasMeasuredValue ?mvRTT.
                          ?mvRTT onQoS:hasDoubleScaleValue ?RTT.
                          ?mparRTT rdf:type qosNet:RTT.
                }
                OPTIONAL{
                          ?profile onQoS:hasQoSMetric ?metricJitter.
                          ?metricJitter onQoS:measuredParameter ?mparJitter.
                          ?metricJitter onQoS:hasMeasurementProcess ?mpJitter.
                          ?mpJitter onQoS:hasMeasuredValue ?mvJitter.
                          ?mvJitter onQoS:hasDoubleScaleValue ?Jitter.
                          ?mparJitter rdf:type qosNet:Jitter.
                }
                FILTER ( (?RTT < k1) || (?Jitter <= k2) )
}
```

5.2 Profiles Ranking

The ranking engine orders the Profile instances retrieved by the SPAR-QL engine according to the predicates and aggregation functions reported in Table 1. To compute a ranking value, our component substitutes the Profile instance specific QoS parameter values in the measurement process of the *WSQoSMetric* representing the user query. Recursively, the execution of that process starts the computation of the input arguments by means of their specific measurement processes.

$$?WSQoSValue = max\left(\frac{2}{1+e^{-\frac{k_1-?RTTValue}{k}}}-1, \frac{2}{1+e^{-\frac{k_2-?JitterValue}{k}}}-1\right) \tag{6}$$

The top-down measurement processes execution terminates when atomic parameters are found. So, partial values on the *WSQoSScale* can be aggregated to compute a globally value for the *WSQoS* parameter. In (6) we show the function used to rank the result set obtained from the SPAR-QL engine for the query in (4).

6 Experimental Results

Once tested the correctness of the onQoS-QL reasoner, the proposed approach was compared with manual matches and a previous implemented matching strategy [3], called QoSAggregator. In [3], each test case QoS description defines a set of metrics with expected values for QoS parameters, that are assumed to be in logical "And". Instead, here in order to formulate the onQoS-QL queries, we are able to use

explicitly the *WSQoSAnd* aggregation function, whose ranking semantics is been defined comparing the following three different functions: *min(x, y), max(x, y)* and *(x+y)/2*. These respectively return the smallest value between *x* and *y*, the largest one and the average.

The recall curves in Fig. 6. a) were obtained constraining the expected *WSQoS* value in the range [0.1, 1] with incremental steps of 0.1. We selected the aggregation function related to the worst case (i.e., *min(x, y)*) to have a better level of robustness, and we adopted a similar approach to define the ranking semantics of other functions.

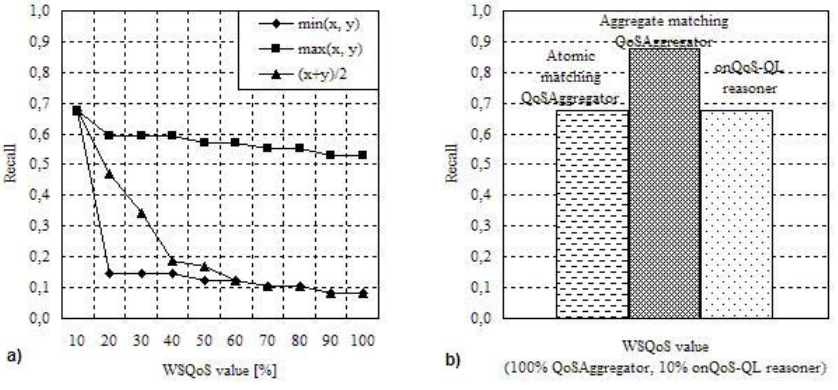

Fig. 6. Comparison of **a)** *WSQoSAnd* ranking semantics and **b)** matching strategy

After established the ranking semantics of *WSQoSAnd*, we compared the recall curve of *QoSAggregator* based on atomic matching with that of the onQoS-QL reasoner. Indeed, the comparison with *QoSAggregator* is significant only for a constrained value of *WSQoS* equal to 100%. With that assumption, the final recall value for the *QoSAggregator* is equal to the initial value for the onQoS-QL reasoner because in the former case the retrieval is performed only with a constrained value of *WSQoS* equal to 100% while in the latter case it is performed for every value and before the ranking phase. So, for a constrained value of WSQoS equal to 0.1, the onQoS-QL reasoner has already filtered out irrelevant profiles. Fig. 6. b) shows the comparison between the QoSAggregator and the onQoS-QL reasoner. It shows also that the adoption of a selection strategy exploiting domain knowledge about QoS parameters also for onQoS-QL would improve of about 30% the recall index, without degrading the precision, since the aggregation decreases false negatives without adding false positives.

In spite of the same value of recall shown in Fig. 6 b) between QoSAggregator (based on the Zeng algorithm) and onQoS-QL, it is worth noting that for the experiments we used simple queries based on the "And" operator to correlate the parameters of each template, so assuming the same simple query semantics used in [3]. However, if the client wishes to use more complex queries, Zeng and QoSAggregator algorithms would show a precision loss, whereas onQoS-QL, by defining operators with an extended semantics, would be able to express complex queries for subjective and context-aware service retrieval and ranking.

7 Conclusion and Future Works

In this paper we addressed the problem of effectively expressing user preferences on QoS selection and ranking. In particular we faced the challenge of design a QoS query language in order to add a query logical layer in our discovery engine architecture. Being QoS a fundamental aspect in business decision-making, discovery strategies need to ensure that non-functional requirements to be satisfied.

In the progress of defining a QoS Discovery Engine, the design of an appropriate query language is a fundamental phase. In fact, it enables users to request services through a set of statements, instead of asking services through services descriptions. onQoS-QL can be regarded as a bridge between onQoS and Service Requestors.

We will perform other experiments to evaluate the improvement of precision of onQoS-QL compared with other techniques when more complex queries are issued. To this end, we propose to interpret the user requests also using context information.

Acknowledgments. The work described in this paper is framed within the LOCOSP[11] and ArtDeco[17] projects funded dy Italian Ministry of University and Research (MIUR).

References

1. Verma, K., et al.: METEOR-S WSDI: A Scalable Infrastructure of Registries for Semantic Publication and Discovery of Web Services. J. of Information Technology and Management 6(1), 17–39 (2005); Special Issue on Universal Global Integration
2. Kramer, J.: Is Abstraction the Key to Computing. Communications of the ACM 50(4) (2007)
3. Giallonardo, E., Zimeo, E.: More Semantics in QoS Matching. In: IEEE International Conference on Service-Oriented Computing and Applications, Newport Beach, pp. 163–171. IEEE Press, Los Alamitos (2007)
4. Wache, H., et al.: Ontology-based integration of information - a survey of existing approaches. In: Stuckenschmidt, H. (ed.) IJCAI 2001 Workshop: Ontologies and Information Sharing, pp. 108–117 (2001)
5. Zhou, C., Chia, L., Lee, B.: DAML-QoS Ontology for Web Services. In: ICWS 2004, pp. 472–479 (2004)
6. Tian, M., Gramm, A., Ritter, H., Schiller, J.: Efficient Selection and Monitoring of QoS-Aware Web Services with the WS-QoS Framework. In: IEEE/WIC/ACM International Conference on Web Intelligence. Proceedings, pp. 152–158 (2004)
7. Dobson, G., Lock, R., Sommerville, I.: QoSOnt: a QoS Ontology for Service-Centric Systems. In: 31st EUROMICRO Conference on Software Engineering and Advanced Applications (2005)
8. Maximilien, E.M., Singh, M.P.: Toward autonomic web services trust and selection. In: ICSOC, pp. 212–221 (2004)
9. Pellet Reasoner, v. 1.3 (April 17, 2006), `http://www.mindswap.org/2003/pellet/`
10. Jena 2.4, `http://jena.sourceforge.net/`
11. LOCOSP, `http://plone.rcost.unisannio.it/locosp`

12. Prud'hommeaux, E.: SPARQL Query Language for RDF, W3C Working Draft A.S. (ed.) (October 4, 2006)
13. Fikes, R., Hayes, P., Horrocks, I.: OWL-QL - A Language for Deductive Query Answering on the Semantic Web. Web Semantics: Science, Services and Agents on the WWW 2(1), 19–29 (2004)
14. Antoniou, G., Bikakis, A.: DR-Prolog: A System for Defeasible Reasoning with Rules and Ontologies on the Semantic Web. IEEE Transactions on Knowledge and Data Engineering, 233–245 (2007)
15. Bosca, A., Bonino, D., Pellegrino, P.: OntoSphere: more than a 3D ontology visualization tool. In: The 2nd Italian Semantic Web Workshop SWAP CEUR Workshop Proceedings, Trento, Italy, December 14-16 (2005), http://ceur-ws.org/Vol-166/70.pdf ISSN 1613-0073
16. Liu, Y., Ngu, A.H.H., Zeng, L.: QoS computation and policing in dynamic web service selection. WWW (Alternate Track Papers & Posters), 66–73 (2004)
17. ArtDeco, http://artdeco.elet.polimi.it/Artdeco