Available online at www.sciencedirect.com

ScienceDirect

Procedia Engineering 69 (2014) 1152 – 1161

Procedia Engineering

www.elsevier.com/locate/procedia

24th DAAAM International Symposium on Intelligent Manufacturing and Automation, 2013

QoS-Aware Web Service Semantic Selection Based on Preferences

Raluca Iordache*, Florica Moldoveanu

University POLITEHNICA of Bucharest, Splaniul Independentei nr. 313,Bucharest, CP 060042, Romania

Abstract

The emergence of service oriented computing brought major research challenges in the process of discovery and selection of web services. Key to success in obtaining the best fitting web service out of a multitude of alternatives is finding the service that both satisfies the requester's functional needs and non-functional requirements. Clients should also be able to indicate how to make trade-offs when some of thir requirements cannot be met. The ability to capture trade-off preferences is critical for selecting the best fitting web service. In this paper, we address the problem of expressing the requester's preferences in a semantic context. We analyze the existing QoS ontologies and choose to extend the OWL-Q ontology to capture trade-off preferences expressed using QoSPref, our own conditional lexicographic approach for QoS preference specification. We extend the QoSSpec specification facet of OWL-Q by adding a new class QoSSelectionWithTradeoffs, that offers the possibility of describing the list of preferences and trade-offs similar to our QoSPref notation, thus offering the possibility of using our algorithm for ranking web service alternatives.

Keywords: QoS preferences; semantic web; service selection; service discovery; QoS ontology; multicriteria decision making

1. Introduction

Semantic web has gained popularity in the recent years, in part due to the need to automate the service discovery process. The use of ontologies facilitates machine reasoning, allowing the implementation of complex web service selection engines that offer accurate results.

Usually, the web service selection process includes the discovery step and the selection step. The discovery step involves searching for appropriate web services, based on functional criteria. During the selection step, the best fitting service is selected from the services discovered in the previous step, by taking into account the client's non-functional requirements. Therefore, ontologies for semantic web should be able to deal with both functional and non-functional requirements.

*Corresponding author. Tel.: 0049-171-5349595
E-mail: riordache@outlook.com

doi:10.1016/j.proeng.2014.03.104

We argue that the capability to express QoS preferences and trade-offs between them is crucial for selecting the best web service. While hard constraints are relatively easy to formulate, there is no standard way to deal with soft constraints that should reflect client's preferences in situations where no web service is capable of satisfying all QoS requirements.

In a previous paper [1], we have introduced a conditional lexicographic approach of articulating non-functional preferences, which offers great flexibility in managing trade-offs while being easy to use and understand. In this paper, we propose a QoS ontology compatible with this preference specification approach. The proposed ontology extends OWL-Q [2] with concepts allowing to capture preference trade-offs expressed using our conditional lexicographic method.

The rest of this paper is organized as follows: Section 2 outlines existing QoS ontologies. Section 3 describes the conditional lexicographic approach for specifying QoS Preferences. Section 4 introduces our QoS ontology that can handle preference trade-offs. The last section concludes the paper and outlines future work directions.

2. Existing QoS Ontologies

Currently, there isn't any established standard for describing the QoS properties. The ontology OWL-S allows the description of QoS properties as name-value pairs, but this approach isn't rich enough for describing complex QoS policies of service requesters and providers [3].

There are several Quality of Service ontology proposals trying to provide a shared vocabulary for the QoS concepts and to facilitate intercommunication regarding QoS in a heterogeneous environment [4]. None of these independent proposals is mature enough to be established as a standard. Among the current approaches we choose to analyze the ones based on OWL-S.

DAML-QoS [5] is an ontology that extends DAML-S, the predecessor of OWL-S, and works with OWL-S. The authors develop external ontologies for metrics and units and offer a basic QoS profile, containing all the basic metrics, that can be inherited und extended. The main problem is that the QoS property constraints are described by cardinality constraints, which actually restricts the number of values a property can take, not on the values themselves. This approach doesn't take into consideration web service requester's preferences and trade-offs.

The QosOnt [4] ontology is a modular ontology, which features several base QoS attributes that can be measured by several metrics, that allows unit conversion and requirements specification. The requirements matching is though simple and only allows expressions of AND / OR constructions to be used. The requirement specification doesn't allow the description of preferences and trade-offs.

OWL-Q [2] is another complex ontology, carefully designed into several facets that can be extended and enriched independently. The ontology allows a symmetric specification of the QoS properties of the request and the offer. For the matchmaking algorithm the approach transforms the comparison to a Constraint Satisfaction Problem and extends the existing CSP-based approaches. OWL-Q addresses the problem of the specification of the requester's priority for QoS constraints by allowing the requester to provide weights to metrics of his interest. The weight expresses the impact of the attribute and allows the ranking of the offers. We argue that providing weights for the metrics isn't enough for capturing complex QoS requirements.

We find the OWL-Q ontology to be the most mature among the existing proposals and address in this paper the important problem of capturing the requester's preferences and trade-offs between them. Our OWL-Q extension proposed in Section 4 also addresses the drawbacks of associating weights to the different QoS dimensions as a method for preference articulation. The association of weights to the soft constraints can be difficult and cannot alone capture accurately the requester's preferences. In the next section, we outline our own approach of capturing trade-offs between QoS preferences in a flexible manner.

3. QoSPref - The conditional lexicographic approach for the elicitation of QoS Preferences

Preference models can be found in various areas like psychology, mathematics, philosophical literature, in economics and game theory, in operations research and decision analysis and in various disciplines of computer science. The choices that we make are guided by our preferences. Understanding preference handling is relevant when attempting to build systems that make choices on behalf of users [6].

The best known and simplest method for preference articulation is the weighted sum method. The method uses weight values supplied by the user to describe the importance of the objectives. One drawback of this method is that the weights must both compensate for differences in objective function magnitudes and provide a value

corresponding to the relative importance of an objective. Another drawback is that it is not able to find certain solutions in the case of a non-convex Pareto curve. The authors of [7] conclude that the weighted sum method "is fundamentally incapable of incorporating complex preference information".

Lexicographic preferences is another simple method used for modeling rational decision behavior. Preferences are defined by a lexical ordering, which leads to a strict ranking. While being very easy to use, lexicographic preferences have the major drawback of being non-compensatory. An extension of this method is lexicographic semiorder, where a trade-off is addressed in situations where there is a significant improvement in one objective that can compensate an arbitrarily small loss in the most important objective. An alternative x is considered better than an alternative y if the first criterion that distinguishes between x and y ranks x higher than y by an amount exceeding a fixed threshold [8]. The advantage of this method is that it ensures that a solution that is slightly better on the most important objective but a lot worse on the other objectives will not be selected.

Our approach to articulate the QoS preferences [1] is based on the observation that, when trying to find a set of rules allowing them to choose between several alternatives, people start by ranking their preferences, in accordance with their perceived importance. This action is equivalent to imposing a lexicographic order on the different criteria that have to be considered. In most situations, using such a strict hierarchy is not sufficient to capture people's real preferences. In this case, people usually introduce additional rules that change the criteria priorities when some specific condition is met.

Our method establishes a total order on the set of existing web service alternatives, by attaching conditions to lexicographic preferences and using a simple algorithm that allows dealing with trade-offs of the preferences. The method also offers a simple and intuitive preference specification language that can be used for authoring QoS preferences.

We illustrate our approach and the use of its associated specification language by considering a hypothetical company that offers data visualization services.

One of these services is the generation of charts based on data sets. Instead of performing itself such tasks, the company delegates them to other business partners, which offer web services for chart generation. A service selection broker chooses the most suitable web service, based on QoS requirements formulated by clients.

The chart generation web services are characterized by domain-independent QoS attributes (e.g., availability, response time) and domain-specific ones (e.g., chart type, cost per chart, number of colors, image resolution).

As client of the hypothetical company we consider a data acquisition system, which regularly sends charts depicting the state of an industrial process to a 1280 x 720 monitor capable of displaying 65536 colors. The image displayed on the monitor can be updated only at fixed intervals of 5 seconds. If a new chart is not available at the end of a 5 seconds interval, the monitor update is postponed until the next end of a 5 seconds interval. Our preference specification language allows specifying both constraints and preferences. Constraints are declared as a list of comma separated boolean conditions that must be satisfied by the service. They are enclosed in a *constraints* block, as shown below:

```
constraints {
        chartType = "time series",
        cost < 10,
        availability > 0.95,
        imageResolution = "1280x720",
        responseTime < 10}
```

Fig. 1. Hard contraints specification.

The order of constraint conditions is irrelevant, but order plays a key role in the articulation of QoS preferences. For the beginning, we consider that the client provides a strict ranking of preferences. This is expressed in our specification language by using a *preferences* block that includes the comma separated list of relevant QoS attributes in the order of their importance:

```
preferences {
        cost,
        availability : high,
        responseTime,
        colors : high}
```

Fig. 2. A simplistic specification of preferences.

For each QoS attribute the client should indicate the direction associated with better values. This piece of information appears after the attribute name, separated by a colon. Possible values for direction are *low* and *high*, where *low* is the default direction and can be omitted.

In the example above, *cost* is the most important QoS attribute, and services with a lower cost are considered better. However, this specification of preferences does not accurately capture client's preferences. A first problem is that selecting a web service with a response time greater than 5 seconds would result in skipping an update of the monitor. This is a serious issue, and such a scenario should be prevented even if this leads to a higher cost. The *responseTime* attribute should be ranked higher only when exactly one of the two web services compared has a value higher than 5 seconds for this attribute. If, for example, both web services considered are able to provide the chart in less than 5 seconds, the problem of missing an update does no longer exist and *responseTime* does not need a higher ranking. Conversely, if both web services compared have a *responseTime* higher than 5 seconds, an update of the monitor will be inevitably skipped, and the actual value of this attribute is no longer of critical importance.

Another problem arises when at least one of the web services compared provides a number of colors less than 65536. Since this will lead to a loss of quality, the client may want to increase in such situations the importance of the *colors* attribute. If both web services provide a number of colors higher than 65536, the *colors* attribute become irrelevant, because the difference in quality cannot be detected on the available monitor.

Finally, a small difference in the values of the *cost* attribute should be ignored if the selection of the slightly more expensive web service leads to a better color quality.

In order to be able to articulate preferences for scenarios like the one above, our specification language provides four unary preference operators, which are shown in :

Table 1. Preference operators.

Preference operator	Meaning
AT_LEAST_ONE* (condition)	condition(service1) OR condition(service2)
EXACTLY_ONE (condition)	condition(service1) XOR condition(service2)
DIFF (attribute)	|service1.attribute - service2.attribute|
ALL (condition)	condition(service1) AND condition(service2)

* default operator (can be omitted)

The first three operators take as argument a boolean formula, which usually involves one or more QoS attributes. The formula is evaluated twice, once for each of the web services to be compared. The two resulting boolean values are passed as arguments to the boolean operator (OR, XOR, or AND) associated with the given preference operator, in order to obtain the return value.

The preference operator DIFF takes as argument a QoS attribute and returns the modulus of the difference of its corresponding values from the two web services compared. Our specification language also allows the definition of virtual QoS attributes, which will be treated as genuine QoS attributes by the preference operators. This can be done by means of the *def* directive, as seen in the example below:

$$\text{def colorDepth} = \log_2(\text{colors})$$

We use the term *preference rule* to denote an entry in the *preferences* block. A preference rule has three components: an optional *condition*, an *attribute* indicating the QoS dimension used in comparisons and a *direction* flag stating which values should be considered better.

The preferences corresponding to the above described scenario can be articulated as follows in our specification language:

```
preferences {
        [EXACTLY_ONE(responseTime >5)] responseTime,
        [DIFF(cost) > 2] cost,
        [colors < 65536] colors : high,
        cost,
        availability : high,
        responseTime,
        colors : high
```

Fig. 3. A more elaborate specification of preferences.

The condition part of the third preference rule in the above *preferences* block (i.e., [colors < 65536]) does not explicitly specify a preference operator, which means it uses the default operator AT_LEAST_ONE.

In what follows, we use the notation $s_1 \succ s_2$ to indicate that the web service s_1 is preferred to the web service s_2, and the notation $s_1 \sim s_2$ to indicate that the service s_1 is indifferent to the web service s_2. Additionally, we introduce the notation $s_1 \succ_k s_2$ to indicate that the web service s_1 is preferred to the web service s_2 and that the preference rule k has been decisive in establishing this relationship. We also introduce the complementary operators $\prec$ and $\prec_k$, defined by the following relations:

$s_1 \prec s_2$, iff $s_2 \succ s_1$

$s_1 \prec_k s_2$, iff $s_2 \succ_k s_1$

An algorithm for comparing two web services based on the preferences expressed using our conditional lexicographic approach is shown in Fig. 4.

```
1   function compareServices(service1, service2, preferences)
2       for i ← 0 .. length(preferences) - 1 do
3           cond ← preferences[i].condition
4           attr ← preferences[i].attribute
5           dir ← preferences[i].direction
6           if cond = null OR cond(service1, service2)=true then
7               result ← compare(service1.attr, service2.attr, dir)
8               if result ≠ 0 then
9                   return {result, i}
10              end if
11          end if
12      end for
13      return null
14  end function
```

Fig. 4. Pairwise comparison of two web services.

The algorithm examines all entries in the *preferences* block in the order in which they appear (line 2). If the current preference rule has no attached condition or the attached condition evaluates to *true* (line 6), the values corresponding to the attribute specified by this entry are compared (line 7). The *compare* function returns a

numerical value that is positive if the first argument is better, negative if the second argument is better and 0 if the arguments are equal (see pseudocode in Fig. 5). If the attribute values are not equal (line 8), the algorithm returns a tuple containing the result of the current comparison and the index of the preference rule that has been decisive in establishing the preference relationship (line 9). Otherwise, the algorithm continues its execution with the next preference rule. A null return value (line 13) indicates an indifference relation between the two web services, while a not-null tuple identifies a relation of type $\prec_k$ or $\succ_k$ between them.

```
function compare (attr1, attr2, dir)
        if attr1 = attr2 then
                result ← 0
        else if attr1 < attr2 then
                result ← 1
        else
                result ← -1
        end if
        if dir = high then
                result ← - result
        end if
        return result
end function
```

Fig. 5. Pairwise comparison of QoS attribute values.

Because our specification language allows the declaration of intransitive preferences, the pairwise comparison of all web service alternatives is in general not sufficient to impose a total order on these services. In order to illustrate this, we use a simplified version of the preferences specified in Fig. 3. As seen in Fig. 6, the simplified version does no longer involve the QoS attribute *availability*. Therefore, this attribute is no longer relevant for the web service ranking. The simplified specification is unrealistic, but it is easier to analyze and it helps us highlight the intransitivity issues. (The preference rule indexes appearing at the left side of the figure are only informative and are not part of the preference specification .)

```
    preferences {

1           [EXACTLY_ONE(responseTime >5)] responseTime,
2           [DIFF(cost) > 2] cost,
3           [colors < 65536] colors : high,
4           responseTime
    }
```

Fig. 6. A simplified specification of preferences used for exemplification.

We consider a set of 5 web service alternatives (WS_1 through WS_5) with the relevant QoS attribute values specified in Table 2. 0

Table 2. Relevant QoS attribute values.

	WS_1	WS_2	WS_3	WS_4	WS_5
responseTime	7.0	7.0	5.5	4.5	7.5
cost	4.0	5.0	6.5	8.0	7.5
colors	256	256	256	65536	65536

The relations identified by the pairwise comparison of the 5 web services considered in our example are depicted in Table 3, where header notations use the format i / j to indicate that the corresponding symbol in the line below represents the preference relation between the web service WS_i and the web service WS_j.

Table 3. Pairwise comparison of the 5 web services.

1/2	1/3	1/4	1/5	2/3	2/4	2/5	3/4	3/5	4/5
$\sim$	$\succ_2$	$\prec_1$	$\succ_2$	$\prec_4$	$\prec_1$	$\succ_2$	$\prec_1$	$\prec_3$	$\succ_1$

Several cases of intransitivity of preferences can be observed in the above table. A first example is given by the following relations:

$WS_1 \succ WS_3 \succ WS_2$
$WS_1 \sim WS_2$

Although WS_1 is indifferent to WS_2, WS_1 is preferred to WS_3, while WS_2 is not preferred to WS_3.
Another example is the rock-paper-scissors relationship induced by:

$WS_2 \prec WS_3 \prec WS_5$
$WS_5 \prec WS_2$

In order to obtain a total order on the set of web service alternatives, we attach to each web service i a *score vector* of integer values: $V_i \in \mathbb{N}^{r+1}$, where r is the number of preference rules. The algorithm used to compute the score vectors is presented in Fig. 7, where n denotes the number of web service alternatives.

```
procedure createScoreVectors ()
        for i ← 1 .. n do
                for k ← 1 .. r do
                        V_i^k ← number of times service WS_i is preferred to another web service due to
                                                                decisive rule k (i.e., due to a ≻_k relation)
        end for
                V_i^{r+1} ← number of times service WS_i is indifferent to another web service.
        end for
end procedure
```

Fig. 7. Procedure to create the score vectors.

For the 5 web service alternatives considered in our example, the corresponding score vectors computed with the above algorithm are presented in Fig. 8.

	$\succ_1$	$\succ_2$	$\succ_3$	$\succ_4$	$\sim$
WS_1	0	2	0	0	1
WS_2	0	1	0	0	1
WS_3	0	0	0	1	0
WS_4	3	0	0	0	0
WS_5	0	0	1	0	0

Fig. 8. Score vectors of the 5 web service alternatives.

Using the score vectors, we are able to provide an algorithm for the ranking of web service alternatives. This algorithm is based on the function *compareScores*, described in pseudocode in Fig. 9. Again, r is used to denote the

number of preference rules. The function takes as arguments two score vectors and returns a numerical value that is positive if the web service corresponding to the first score vector is preferred, negative if the web service corresponding to the second score vector is preferred and 0 if the corresponding web services are indifferent to each other.

```
1  function compareScores(V1, V2)
2      count1 ← Σ_i^r V1^i
3      count2 ← Σ_i^r V2^i
4          if count1 ≠ count2 then
5              return count1 - count2
6          end if
7          for i ← 1 .. r + 1 do
8              if V1^i ≠ V2^i then
9                  return V1^i − V2^i
10             end if
11         end for
12         return 0
13 end function
```

Fig. 9. Function for score vector comparison.

For each of the two corresponding web services, the function computes the number of times it has been preferred to other web services (lines 2, 3). This computation does not take into account the number of times a web service has been found to be indifferent to another one (hence the sum is taken up to the value r, not $r + 1$).

If the previously computed values $count_1$ and $count_2$ are not equal (line 4), the web service with the higher value is chosen as the better one (line 5).

Otherwise, the algorithm scans each position in the score vectors (line 7) and if it finds different values, the web service corresponding to the higher value is chosen as the better one. The scanning of the values in the vector scores starts with the position corresponding to the first preference rule, because this is considered the most important one, and it ends with the position corresponding to the number of indifference relations (i.e., $r + 1$), because this is considered the least important one.

In contrast with the function *compareServices* presented in Fig. 4, the function *compareScores* induces a total order on the set of web service alternatives, thus allowing us to rank them accordingly. Using this algorithm, the 5 web service alternatives considered in our example will be ranked in the following order:
(WS_4, WS_1, WS_2, WS_5, WS_3),
with WS_4 being the best alternative.

4. An OWL-Q extension compatible with the QoSPref approach

In order to use the QoSPref requirements specifications presented above in a semantic context, we extend the OWL-Q ontology to use our conditional lexicographic method.

4.1. OWL-Q Requirements Specification

In OWL-Q [2] the QoS offers and requests are defined by the QoSSpec Facet. The class QoSSpec class contains the QoS description of a web service. It contains several attributes like the cost of using the service and the currency, security and transaction protocols, the validity of the offer. QoSSpec is separated into two subclasses, disjoint to each other: QoSOffer and QoSDemand where the WS providers and requesters can define in the same way, symmetrically, their QoS constraints.

The WS requester can provide constraints by using the QoSDemand class and can also provide weights to metrics of his interest, by using the QoSSelection class. The QoSSelection class contains list of <metric,weight> entries. The weight value can have the value of 2.0 if it is a hard constraint or a value in (0:0; 1:0) if it is soft.The entry <metric,weight> is defined by the QosSelectElem class and the QoSSelectElemeList contains a list of QosSelectElem-s. For a better understanding, Fig. 10 shows the graph representing the QosSpec facet, generated by the ontology editor Protégé.

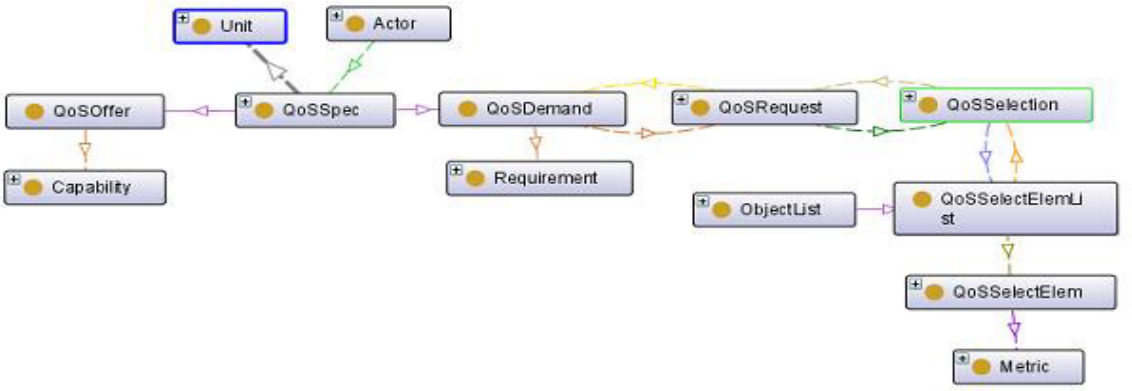

Fig. 10. The QoSSpec facet of OWL-Q.

OWL-Q also contains a Metric Value Type Facet that describes the types of values a QoS metric can take. The MetricValueType class has two subclasses indicating the direction of values for the QoS metric that owns one of these two subtypes: PositivelyMonotonic value types have a direction of values from the lowest to the highest value where the highest value is mapped to the highest quality level that can be achieved while a NegativelyMonotonic value type has an opposite direction of values where the highest quality level is mapped to the lowest value. As an example, a QoS metric measuring the Availability QoS attribute has as value type a positively monotonic value type while a ReponseTime QoS Attribute has a negatively monotonic value type, as the lowest values are the better ones. Thus, when describing a preference rule from out QosPref approach, we don't need to explicitly write the direction flag of a metric anymore, as this is already described by the metric itself.

4.2. Extension of the OWL-Q Requirements Specification to capture preference trade-offs

From the existing specification in OWL-Q we can keep the QoSOffer description as the provider specification remains unmodified. For this reason, we choose to extend the existing facet by adding new classes for the preference selection, and not create another facet.

We define a new QoSSelectionWithTradeoffs class that will incorporate a list of preferences, described similar to our QosPref's notation. The hard constraints are specified using the QoSDemand class, just like in the OWL-Q ontology. A preference is described by a QoSPreference class that incorporates a preference rule. As already seen, a preference rule has three components: an optional condition, an attribute indicating the QoS metric used in comparisons and a direction flag stating which values should be considered better. As the direction flag is specified using the MetricValueType class from OWL-Q, a preference rule has to include the optional condition and the metric. A preference rule entry looks like that: <(optional condition) metric>.

For the optional condition the three logical operators AT_LEAST_ONE (OR operator), EXACTLY_ONE (XOR operator), ALL (AND operator) and the arithmetic operator DIFF (-) as seen in Table 1 have to be included in the ontology. The optional condition is described by can be described as <operator, metric>.

Another observation is that the order of the preferences is an important factor for the ranking algorithm, with the first preference being the most important. So, the order of preferences in the preferences list is important. To make the semantic notation easy to follow we choose to explicitly attach the priority to every preference rule so that an entry in the QoSPreference class will look like that: <priority, (condition) metric>.

The ontology extension is shown in Fig. 11.

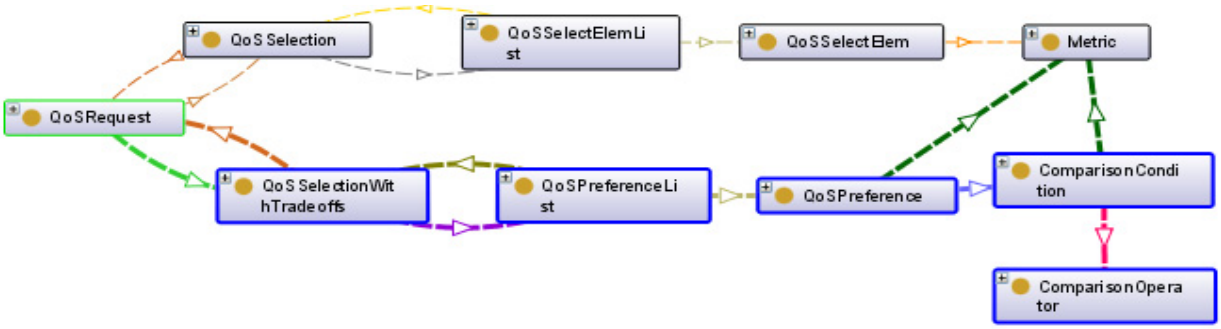

Fig. 11. OWL-Q extension compatible with the QoSPref approach.

The QoSRequest class will either point to both QoSSelectionWithTradeoffs and QoSSelection class. The requester can beside which QoS specification he wants to use depending on the use case.

The ontology extension allows the requester to use a notation to flexibility define complex constraints and trade-offs of preferences in a semantic context.

5. Conclusion and future work

In this paper we analyse the existing QoS ontologies from the service requester's perspective. We extend an existing, carefully designed ontology OWL-Q, to use our rich and intuitive specification notation, QoSPref, for capturing the user's QoS constrains and tradeoffs. This allows us to make a semantic web service discovery and selection using our simple algorithm that allows dealing with tradeoffs.

Our current efforts are directed toward designing and implementing a framework for dynamic semantic web service selection that supports the handling of QoS preferences based on our conditional lexicografix method. An open source prototype implementation of the comparison engine is already available and we plan to also offer open source implementations for the other components of our framework.

References

[1] R. Iordache and F. Moldoveanu, "A conditional lexicographic approach for the elicitation of QoS Preferences," in *Lecture Notes in Computer Science Volume 7565, pp 182-193*, Rome, 2012.

[2] K. Kritikos and D. Plexousakis, "OWL-Q for Semantic QoS-based Web Service Description and Discovery," in *Proceedings of the SMR2 Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web*, Heraklion, Greece, 2007.

[3] Li, Fei; He, Yanxiang; Hu, Wensheng; Wu, Libing; Wen, Peng, "Web Service Selection Based on Fuzzy QoS Attributes," *Journal of Computational Information Systems,* vol. 7, no. 1, pp. 198-205, 2011.

[4] G. Dobson, R. Lock and I. Sommerville, "QoSOnt: an Ontology for QoS in Service-Centric Systems," in *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, Computing Department, Lancaster University, 2005.

[5] C. Zhou, L.-t. Chia and B.-s. Lee, "DAML-QoS Ontology for Web Services," in *International Conference on Web Services*, 2004.

[6] R. Brafman and C. Domshlak, "Preference Handling - An Introductory Tutorial," *AI Magazine,* vol. 30, no. 1, pp. 58-86, 2009 .

[7] R. Marler and J. Arora, "The weighted sum method for multi-objective optimization: new insights," *Structural and Multidisciplinary Optimization,* p. 853–862, 2010.

[8] P. Manzini and M. Mariotti, "Choice by lexicographic semiorders," *Theoretical Economics,* 2010.