A web service composition approach based on QoS preferences (Short paper)

Raluca Iordache
University ”POLITEHNICA” of Bucharest, Romania
Email: riordache@outlook.com

Florica Moldoveanu
University ”POLITEHNICA” of Bucharest, Romania
Email: fm@cs.pub.ro

***Abstract*—One important step in dynamic web service composition is to bind concrete web services to the activities involved in the composition. Ideally, the component services are assigned such that the resulting composite web service meets the service level agreements and offers the best trade-off between the various QoS parameters. This implies the ability to express preferences related to the trade-offs. In this paper, we propose a *binding-as-a-service* (BaaS) approach based on a powerful, but at the same time simple and intuitive notation for specifying user preferences. The typical client of a BaaS provider is a module of a service composition framework. The service request describes the abstract composition model, the available component services and the QoS constraints and preferences. A prototype implementation of this *binding-as-a-service* approach validates our method.**

***Keywords*—multidimensional QoS preferences, service binding, service composition.**

I. INTRODUCTION

Web service composition is a complex task, which involves three challenging steps: (1) composite web service specification, (2) selection of the component web services and (3) execution of the composite web services [1]. In this paper, we focus on the second step, whose goal is to bind concrete web services to the activities involved in the composition, in order to produce the most suitable composite service. Since service binding is required by any service composition middleware, we advocate the use of a *binding-as-a-service* (BaaS) approach for implementing this operation.

For each task in the abstract composition model created in step (1), there are usually several web services offering the required functionality. However, they may differ in non-functional properties such as reliability, cost, or response time. Therefore, the *quality of service* (QoS) is the deciding factor in choosing which concrete web service to bind to each task. The preferred selection strategy is a global planning approach, where QoS constraints and preferences are expressed with respect to the composite service as a whole [2]. An issue related to the global planning is the computation of the QoS of a composite service based on the QoS of its component services. Most solutions to this problem are limited to composition models that can be represented as well-structured workflows. Our *binding-as-a-service* (BaaS) implementation uses the aggregation method proposed by Yang *et al.* [3], which overcomes this restriction.

The ability to express trade-offs between different QoS parameters is critical in order to provide a binding that produces the most suitable composite service. Common approaches are based on specifying priorities or associating weights to the different QoS dimensions. The drawback of these methods is that they cannot accurately capture users' preferences. The BaaS approach presented in this work uses the conditional lexicographic method of articulating non-functional preferences introduced in one of our previous papers [4]. This method offers great flexibility, while being easy to use and understand. It uses an intuitive notation and leads to a simple algorithm for selecting web services, which does not require sophisticated multi-criteria decision techniques.

The rest of this paper is organized as follows: Section II introduces an illustrative example highlighting some of the issues related to the QoS-aware dynamic web service composition. Section III discusses the problem of estimating the QoS of a composite service and presents the aggregation technique chosen in this work. Section IV describes our conditional lexicographic approach for expressing QoS preferences. The last section concludes the paper and outlines future work directions.

II. ILLUSTRATIVE EXAMPLE

In this section, we give an illustrative example that will be used throughout this paper in order to expose some of the issues related to the QoS-aware dynamic web service composition. We consider an online trading system offering services for trading various financial instruments. One of these services allows customers to buy both domestic and foreign stocks. Its business process model is depicted in Fig. 1.

Some of the tasks in this model (such as those for order registration) represent internal actions of the online trading system. Other tasks (such as those for getting stock quotes) require interaction with external systems. The online trading system implements the stock buying service as a composite web service. For the tasks involving interaction with external systems, it is necessary to find providers offering the required functionality as a web service. Usually, there are several alternatives for each of these tasks. For example, there are many web services that provide stock quotes. The online trading system has to decide which of the possible service components to bind to each task in its composition model. This service binding is a dynamic process, because over time, some component services may cease to exist and new ones may become available.

In our example, we consider that only the following QoS attributes are interesting for the online trading system: execution time, cost, and reliability. The QoS of the composite

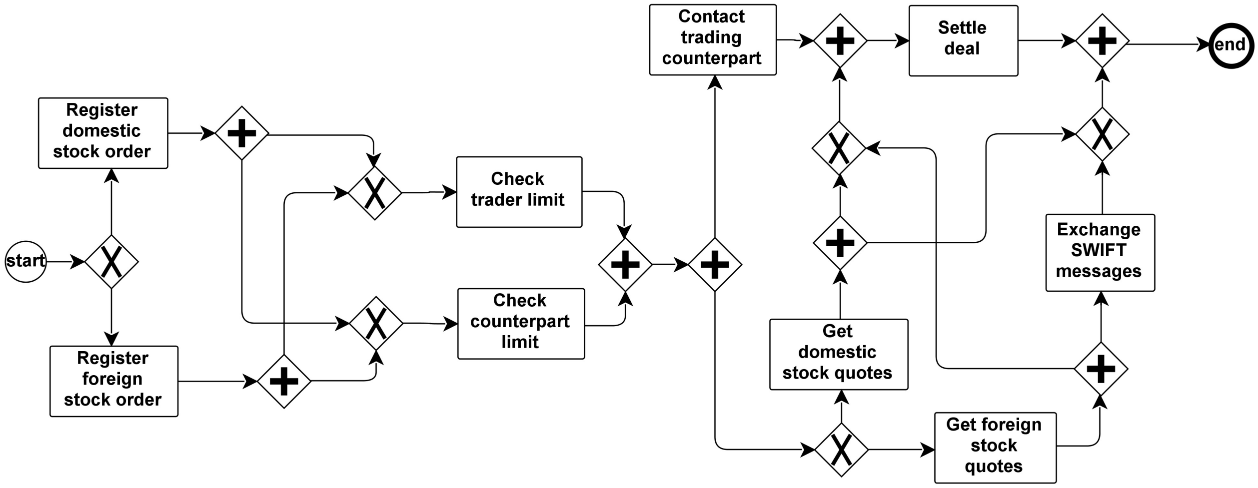

Fig. 1. BPMN model of a process for buying stocks

stock buying service is determined by the QoS of its service components, which are assumed to be known. However, it is not clear how to estimate the QoS of the composite service. While the aggregated cost can be easily computed by adding the costs of all component services, there is no obvious method for estimating execution time and reliability.

In order to be able to dynamically bind component services to the tasks specified in the composition model of the stock buying service, the online trading system must have an automated method of comparing composite services based on their QoS. We add a few more details about the online trading system to illustrate why comparing composite services characterized by multiple QoS attributes is not a trivial task. The executives of this system try to maximize their profit, therefore they see the cost as the most important QoS parameter. However, they are willing to ignore small cost differences (not exceeding 10 cents) if the composite service with a higher cost has better values for reliability and execution time. For the customers of this system, it is very important that trading orders are executed as soon as possible. Therefore, the online trading systems guarantees that the execution time of its stock buying service does not exceed 30 seconds. For every violation of this agreement, the owners of the online trading system must pay a penalty proportional with the delay. This means that, when comparing two composite services, the execution time becomes the most important parameter if at least one of the compared services has an execution time exceeding the 30 seconds limit. It is clear that traditional methods such as weighted sum or parameter ranking are not appropriate for this scenario. In the next sections we address the problems highlighted by this example.

III. THE AGGREGATED QOS OF COMPOSITE SERVICES

Various solutions have been proposed for the problem of estimating the aggregated QoS of a composite service, but they differ in the restrictions they impose on the topology of the composition. Most of them are limited to orchestration models that can be represented as well-structured workflows. Yang *et al.* [3] have introduced a method that overcomes these restrictions. This method, which is used in our *binding-as-a-service* (BaaS) implementation, is presented in the remaining of this section.

The input of this method is an orchestration model together with a binding that maps tasks to component services. An orchestration model is a directed graph with execution probabilities attached to its edges. The orchestration models are decomposed into *orchestration components*, which are subgraphs with a single-entry and single-exit point. The QoS is computed in a bottom-up manner for each orchestration component. Well-structured orchestration models, that is, models where each split gateway has a corresponding join gateway, are straightforward to analyze. Different aggregation formulas are provided depending on the type of the QoS attribute, which can be classified into three categories: *critical path*, *additive* and *multiplicative*.

A preliminary step of the QoS aggregation method is to use the block-structuring technique introduced in [5] to transform an unstructured orchestration model into a *maximally-structured* orchestration model. The model in Fig. 2 is behaviorally equivalent with the one in Fig. 1, but the left side half of the transformed model is now well-structured.

The components that are irreducible using this technique are called *rigid components* and they are of two types: irreducible Directed Acyclic Graphs (DAG) and irreducible multiple-entry, multiple-exit (MEME) loops. The authors of [3] provide an algorithm that transforms irreducible DAG components in equivalent choice components. Irreducible MEME loops can be transformed using the block-structuring technique into equivalent rigid components where the concurrency is fully encapsulated within child components. For these equivalent components, the expected number of times that a node in the MEME loop is visited can be calculated using standard methods. This allows computing the QoS of the irreducible component by applying the aggregation formulas characteristic to each category of QoS parameters.

IV. QOS PREFERENCE SPECIFICATION

As mentioned before, the ability to capture and handle trade-offs between QoS preferences plays a crucial role in creating a high quality composite service. In this work, we use a method of expressing non-functional preferences that we

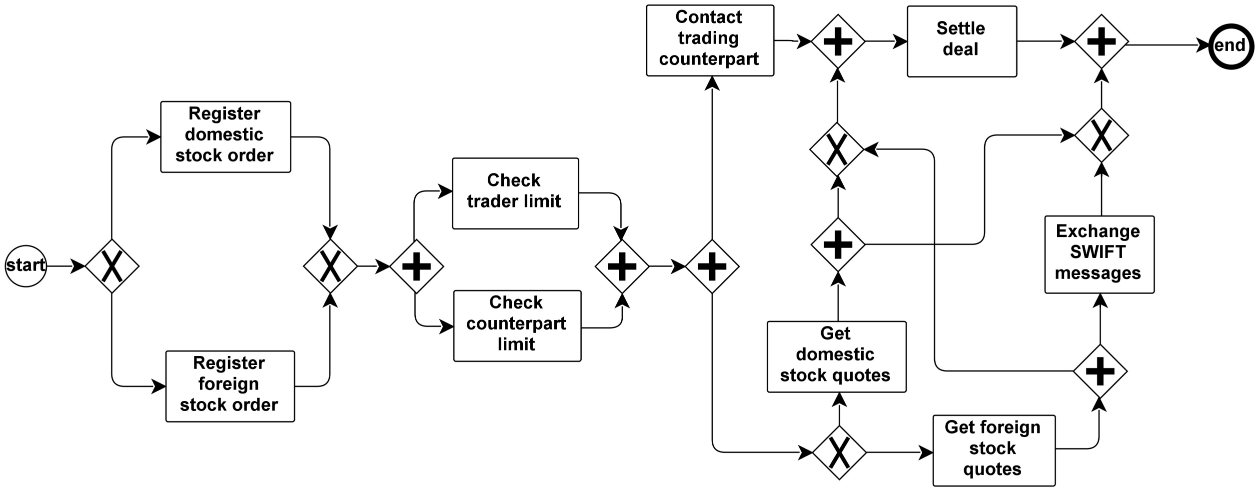

Fig. 2. Maximally structured model of the process for buying stocks

have introduced in [4]. This method 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 alternatives, by attaching conditions to lexicographic preferences and provides a preference specification language that can be used for authoring QoS preferences.

For illustration purposes, we refer again to the online trading system example introduced in Sect. II. We consider that the executives of the online trading see the *cost* as the most important QoS attribute, followed by *reliability* and then by *execution time*. As mentioned before, the executives are willing to ignore small cost differences (not exceeding 10 cents) if the composite service with a higher cost has better values for reliability and execution time. Furthermore, there are penalties to be paid if the execution time of the composite service exceeds 30 seconds. Therefore, when comparing two composite services, the execution time becomes the most important parameter if at least one of the compared services has an execution time exceeding the 30 seconds limit.

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 I.

TABLE I. PREFERENCE OPERATORS

Preference operator	Meaning
AT_LEAST_ONE(condition)	condition($service_1$) OR condition($service_2$)
EXACTLY_ONE(condition)	condition($service_1$) XOR condition($service_2$)
ALL(condition)	condition($service_1$) AND condition($service_2$)
DIFF(attribute)	$\|service_1.attribute - service_2.attribute\|$

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 uses a *preferences* block that includes a comma separated list of entries, called *preference rules*, listed in the order of their importance. 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.

In our specification language, the preferences corresponding to the above described scenario can be articulated as shown in Fig. 3. (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) [AT_LEAST_ONE(execTime $>$ 30)] execTime : low,
2) [DIFF(cost) $>$ 10] cost : low,
3) reliability : high,
4) execTime : low,
5) cost : low,
}

Fig. 3. A specification of preferences

The specification language can deal with situations where people are not fully aware of their preferences. When users notice that the current rules do not accurately capture their preferences, they can simply add a new conditional rule, thus incrementally improving the preference specification.

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 (*i.e.*, s_1 and s_2 are equally preferred). Additionally, we introduce the notation $s_1 \underset{k}{\succ} s_2$ 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. The complementary operators $\prec$ and $\underset{k}{\prec}$ are defined in a similar manner.

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 ← 1 .. length(preferences) 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 $\neq$ 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. 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 $\underset{k}{\prec}$ or $\underset{k}{\succ}$ between them.

In a series of experiments, Tversky [6] has shown that people have sometimes intransitive preferences. Therefore, being able to capture such preferences is an important feature of our specification language. However, a consequence of allowing intransitive preferences is that 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 consider a set of 5 composite web services (WS_1 through WS_5) with the aggregated QoS values specified in Table II.

TABLE II. RELEVANT QOS ATTRIBUTE VALUES

	WS_1	WS_2	WS_3	WS_4	WS_5
execTime	27	24	31	28	26
cost	536	548	520	525	540
reliability	0.97	0.96	0.98	0.98	0.96

Using the preferences specified in Fig. 3, the relations identified by the pairwise comparison of the 5 composite services above are depicted in Table III, 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 services WS_i and WS_j.

TABLE III. PAIRWISE COMPARISON OF THE 5 COMPOSITE WEB SERVICES

1/2	1/3	1/4	1/5	2/3	2/4	2/5	3/4	3/5	4/5
$\underset{2}{\succ}$	$\underset{1}{\succ}$	$\underset{2}{\prec}$	$\underset{3}{\prec}$	$\underset{1}{\succ}$	$\underset{2}{\prec}$	$\underset{4}{\succ}$	$\underset{1}{\prec}$	$\underset{1}{\prec}$	$\underset{2}{\succ}$

In the above table the following intransitive relationship can be observed: $WS_1 \succ WS_2 \succ WS_5$ and $WS_5 \succ WS_1$.

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. 5, where n denotes the number of web service alternatives.

```
procedure createScoreVectors()
    for i ← 1 .. n do
        for k ← 1 .. r do
            $V_i^k$ ← no. of times $WS_i$ is preferred to another service
                    due to decisive rule k (i.e., due to a $\underset{k}{\succ}$ relation).
        end for
        $V_i^{r+1}$ ← no. of times $WS_i$ is indifferent to another service
    end for
end procedure
```

Fig. 5. 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. 6.

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. 7. 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.

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

	$\underset{1}{\succ}$	$\underset{2}{\succ}$	$\underset{3}{\succ}$	$\underset{4}{\succ}$	$\underset{5}{\succ}$	$\sim$
WS_1	1	1	0	0	0	0
WS_2	1	0	0	1	0	0
WS_3	0	0	0	0	0	0
WS_4	1	3	0	0	0	0
WS_5	1	0	1	0	0	0

Fig. 6. Score vectors of the 5 web service alternatives

```
 1) function compareScores($V_1$, $V_2$)
 2)     $count_1 \leftarrow \sum_{i=1}^{r} V_1^i$
 3)     $count_2 \leftarrow \sum_{i=1}^{r} V_2^i$
 4)     if $count_1 \neq count_2$ then
 5)         return $count_1 - count_2$
 6)     end if
 7)     for i $\leftarrow$ 1 .. r + 1 do
 8)         if $V_1^i \neq V_2^i$ then
 9)             return $V_1^i - V_2^i$
10)         end if
11)     end for
12)     return 0
13) end function
```

Fig. 7. Function for score vector comparison

corresponding to the higher value is chosen as the better one (lines 8-10). 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. If the score vectors are identical, the function returns 0 (line 12).

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_5, WS_2, WS_3), with WS_4 being the best alternative.

V. THE BINDING-AS-A-SERVICE (BAAS) IMPLEMENTATION

A large number of web service composition frameworks have been developed, with different architectures and methodologies. Nonetheless, service binding is a task required by all these frameworks. Therefore, it is useful to offer this functionality as a service. The typical client of a *binding-as-a-service* (BaaS) provider is a module of a web service composition framework, which needs to find the best mapping of concrete web services to the tasks of a composition model.

We provide a QoS-aware BaaS implementation based on the QoS aggregation method of Yang *et al.* [3] and on our preference handling approach detailed in the previous section. The prototype implementation is written in Java and it is available as open source at: http://baas.sourceforge.net/.

A web service request sent to our BaaS provider must contain the following information: the orchestration model; the list of QoS attributes; for each task in the orchestration model, a list of concrete web services offering the required functionality; the QoS constraints; the QoS preferences.

The orchestration model is represented as a workflow with execution probabilities attached to its edges. If probabilities are missing, our implementation will assign default probabilities. Edges starting from an XOR gateway are assigned a probability of $1/k$, where k is the number of outgoing edges of the given XOR gateway. All other edges are assigned a probability of 1.

The list of QoS attributes must contain information about the aggregation category of each attribute. The list of concrete web services offering the required functionality of a given task must specify for each concrete web service its QoS values. Not all web services have all QoS attributes of the composite service. For example, a composite service that converts data sets to graphic charts may have a QoS attribute indicating the number of colors of the resulting image. The composite service may have a component service that sorts the data set. The number of colors is clearly not a QoS attribute of the sorting service. In situations where a QoS attribute is missing for a component service, our implementation provides default values, in accordance with the aggregation category of the missing QoS attribute.

Some of the tasks in an orchestration model may be internal actions. For these tasks, the list of concrete web services implementing their functionality is empty.

The QoS preferences are specified using the preference notation introduced in the previous section. Our BaaS implementation uses a genetic algorithm in order to find the best mapping of component services to tasks. This algorithm and the exprimental results will be discussed in a subsequent paper.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a dynamic web service composition approach that can deal with complex QoS preferences. We have focused on the problem of binding concrete web services to the activities involved in the composition and we have offered a prototype implementation in the form of a *binding-as-a-service* (BaaS) provider.

In our work, we have combined two powerful technologies. The first one is our method of dealing with QoS preferences, which offers great flexibility in managing trade-offs, but is at the same time very intuitive. The second one is the QoS aggregation method of Yang *et al.* [3], which has the major advantage of being able to deal with unstructured orchestration models.

Our current efforts are directed toward devising an ontology compatible with the QoS preference approach used in this paper. This ontology should also be able to deal with the aggregation categories of QoS attributes.

REFERENCES

[1] El Haddad, J., Manouvrier, M., Rukoz, M., *TQoS: Transactional and QoS-Aware Selection Algorithm for Automatic Web Service Composition*, IEEE Trans. on Services Computing, 3, issue 1, pp. 73–85, 2010

[2] Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H., *QoS-Aware Middleware for Web Services Composition*, IEEE Trans. on Software Engineering, 30, issue 5, pp. 311-327, 2004

[3] Yang, Y., Dumas, M., Garca-Bauelos, L., Polyvyanyy, A., Zhang, L, *Generalized aggregate Quality of Service computation for composite services*, Journal of Systems and Software, 85(8), pp. 18181830, 2012

[4] Iordache, R., Moldoveanu, F., *A Conditional Lexicographic Approach for the Elicitation of QoS Preferences*, In: Meersman, R., Panetto, H., Dillon, T.S., Rinderle-Ma, S., Dadam, P., Zhou, X., P, S., Ferscha, A., Bergamaschi, S., Cruz, I.F. (eds.) OTM Conferences (1), LNCS, vol. 7565, pp. 182-193. Springer 2012

[5] Polyvyanyy, A., Garca-Bauelos, L., Dumas, M., *Structuring Acyclic Process Models*, In: Proceedings of the International Conference on Business Process Management, pp. 276–293,New York, NY, USA, 2010

[6] Tversky, A., *Intransitivity of Preferences*, Psychological Review, vol.76, no.1, pp. 31–48, 1969.