An integrated framework for adapting WS-BPEL scenario execution using QoS and collaborative filtering techniques

D. Margaris,[a,*] C. Vassilakis,[b] P. Georgiadis[a]

[a] Department of Informatics and Telecommunications, University of Athens, GR-15-184, Athens, Greece

[b] Department of Informatics and Telecommunications, University of Peloponnese, GR-221 00, Tripoli, Greece

Abstract

In this paper, we present a framework which incorporates runtime adaptation for BPEL scenarios. The adaptation is based on (a) the quality of service parameters of available services, allowing for tailoring their execution to the diverse needs of individual users and (b) on collaborative filtering techniques, allowing clients to further refine the adaptation process by considering service selections made by other clients, in the context of the same business processes. The proposed framework also caters maintaining the transactional semantics that invocations to multiple services offered by the same provider may bear.

Keywords: WS-BPEL, Adaptation, Quality of service, Collaborative filtering, Metasearch algorithms

1 Introduction

Web Services are considered a dominant standard for distributed application communication over the internet. Consumer applications can locate and invoke complex functionality, through widespread XML-based protocols,

without any concern about technological decisions or implementation details on the side of the service provider. Web Services Business Process Execution Language (WS-BPEL) [1] allows designers to orchestrate individual services so as to construct higher level business processes; the orchestration specification is expressed in an XML-based language, and it is deployed in a BPEL execution engine, made thus available for invocation by consumers.

WS-BPEL has been designed to model business processes that are fairly stable, and thus involve the invocation of web services that are known beforehand. Therefore, the BPEL scenario designer specifies, at the time the scenario is crafted, the exact services to be invoked for the realization of the business process. This setting is however considered inadequate in the context of the current web: many functionalities offered by the services invoked within the scenario (e.g. checking for free rooms in a hotel or booking an air flight) are typically offered by numerous providers (different hotels and flight companies, respectively), and each providers offers its service under different quality of service (QoS) parameters. In this environment, it would be highly desirable for consumers to be able to tailor the WS-BPEL scenario execution according to their QoS requirements. Indeed, [2] lists governance for compliance with QoS and policy requirements as an open issue for the SOA architecture.

To tackle this shortcoming, numerous approaches have been proposed, following two main strategies [3]: (i) *horizontal adaptation*, where the composition logic remains intact and the main adaptation task is to select the service and invoke the service best matching the client's QoS requirements; the selected services are substituted for either *abstract tasks* (e.g. [3]) or *concrete service invocations* (e.g. [5]) and (ii) *vertical adaptation*, where the composition logic may be modified. The incorporation of run-time adaptation introduces the need for service selection affinity maintenance [5]: service selection affinity refers to cases where a service selection in the context of adaptation implies the binding of subsequent selections (e.g. selecting a hotel reservation from a travel agency dictates that the payment will be made to the same travel agency), to cater for preserving the transactional semantics that invocations to a specific service provider may bear.

QoS-based service selection, however, limits the adaptation criteria to aspects such as cost, availability and performance, not being able to take into account the satisfaction of service users “in the real world”; for instance, an airline company may offer low fares and a short trip duration, but the actual traveling experience may be very poor, an aspect not reflected in QoS attributes and therefore unavailable for the purposes of adaptation. On the other hand, collaborative filtering combines the informed opinions of humans (i.e. opinions taking into account the aspect of satisfaction), to make personalized, accurate predictions and recommendations [6]. In the context of collaborating filtering, personalization is achieved by considering ratings of "similar users" (in our case the a user is considered to rate a service favorably if she actually uses it), under the collaborative filtering’s fundamental assumption that if users X and Y have similar behaviors (e.g., buying, watching, listening – in our case, selecting the same services) on some items, they will act on other items similarly [7].

Under this light, a prominent approach would be to combine QoS-based service selection with collaborative filtering to perform adaptation based on both objective data (QoS attributes) and subjective ratings (quality of experience). Similarly, the combination of content-based filtering with collaborative filtering has been proposed in a number of works (e.g. [8][9]).

However, personalization and adaptation needs may extend beyond the specification of QoS requirements. In some cases, users may desire to select the exact services to be invoked for some cases and ask for recommendations on other services; for instance, in a holiday planning application, the user may require that reservation is made in a particular hotel, while at the same time asking for a recommendation about the airline. Once a user has made some explicit service selections, the combined approach can be used to make recommendations on the services that the user has not bound to specific providers.

In this paper we present an approach for adapting the execution of WS-BPEL scenarios taking into account both QoS-based criteria and collaborative filtering techniques, introducing both an adaptation algorithm and an associated execution framework. The adaptation algorithm uses both QoS specifications and semantic-based collaborative filtering

personalization techniques to decide on which offered services best fit the client's profile. To achieve this goal, we follow the metasearch algorithm paradigm [56], using two different candidate adaptation ranking algorithms, the first examining the QoS aspects only and the second being based on collaborative filtering techniques. The adaptation rankings produced by these two algorithms are combined to generate the overall ranking, which then drives the adaptation. The combination of the results is performed using a weighted metasearch score combination algorithm [56][57], however varying weights are used to address issues associated with collaborative filtering, such as cold start (i.e. few entries recorded in the rating database, thus no good matches can be obtained) and gray sheep (i.e. unusual users, which cannot be matched with other users even after the database has been adequately populated) [10].

The proposed BPEL execution framework includes provisions for (a) specifying QoS requirements for invocations of web services within a WS-BPEL scenario (b) specifying specific bindings for selecting services and designating which services are subject to adaptation and (c) adapting the WS-BPEL scenario execution according to the results of the service selection algorithm. Our approach follows the horizontal adaptation paradigm since, as noted in [5], horizontal adaptation preserves the execution flow which has been crafted by the designer to reflect particularities of the business process, while it also allows the exploitation of specialized exception handlers.

This paper extends the work presented in [53] by including the collaborative filtering adaptation algorithm proposed in [54]. The two algorithms run separately, and a combination step is introduced to synthesize the individual algorithm proposals to a single result. Effectively thus, in this work the adaptation process is driven by a metasearch algorithm [56], where each individual algorithm (the QoS-based and the collaborating filtering-based ones) ranks the suitability of each possible adaptation, and the combination step synthesizes these proposals in a way which optimizes the performance of the combination [57]. In order to allow this mode of operation, where each of the algorithms is actually a "voter", the algorithms presented in [53] and [54] have been redesigned to a large extent, in order to use integer programming techniques instead of exhaustive search, in order to cater for efficiency and scalability. To the

best of the authors' knowledge, no other work supports adaptation of WS-BPEL scenario execution based on both QoS-based and collaborative filtering-based criteria.

The rest of the paper is structured as follows: In section 2 we briefly present the QoS concepts and collaborative filtering foundations used in this work. In section 3 we present the proposed service recommendation algorithm in the context of WS-BPEL scenario execution, while section 4 presents the execution adaptation architecture and elaborates on the functionality of its components. Section 5 presents a qualitative (in terms of proposed solution QoS) and a quantitative (in terms of performance) evaluation of the proposed approach. In section 6 we overview related work, while section 7 concludes the paper and outlines future work.

2 QoS concepts and collaborative filtering foundations

In the following subsections we summarize the concepts and underpinnings from the areas of QoS and collaborative filtering, which are used in our work.

2.1 QoS concepts and definitions

QoS may be defined in terms of attributes [27][28], while typical attributes considered are cost, response time, availability, reputation, security etc. [29]. For conciseness purposes, in this paper we will consider only the attributes *responseTime (rt), cost (c)* and *reliability (rel)*, adopting their definitions from [20]. Extension of the framework to include additional attributes is straightforward, thus we have no loss of generality.

In the proposed framework, the QoS specifications for a service within the BPEL scenario may include an upper bound and a lower bound for each QoS attribute, i.e. for service s_j included in a WS-BPEL scenario, the designer formulates two vectors $MIN_j(min_{rt,j}, min_{c,j}, min_{rel,j})$ and $MAX_j(max_{rt,j}, max_{c,j}, max_{rel,j})$. Additionally the designer formulates a *weight* vector $W = (rt_w, c_w, rel_w)$, indicating how important each QoS attribute is considered by the designer in the context of the particular operation invocation (effectively, weight element values are multiplied by the value of

the respective QoS dimension of the service composition, and these products are then summed to produce a total score for the composition). The values of the QoS attributes are assumed to be expressed in a "larger values are better" encoding, hence a service having *response time = 7* actually responds *in less time* than a service with *response time = 3* (better response time). Note that weights apply to the *whole composition*, rather than to individual services, since they reflect the perceived importance of each QoS attribute dimension on the process as a whole, and not its constituent parts [4].

For convenience reasons, we assume that all QoS attributes are normalized in the range [0, 1]; value range normalization is a typical approach in works considering QoS-based adaptation (e.g. [4][11]). Note that this implies that a total ordering relationship is required in the range of the QoS attributes; this is always the case with attributes such as response time and reliability, however some QoS attributes are more complex: for instance [32] identifies seven dimensions related to security, and some security mechanism S1 may outperform some other security mechanism S2 in some dimensions but lag behind S2 in other dimensions. This could be tackled by decomposing the *security* QoS attribute in seven distinct attributes, one for each dimension, ensuring thus the existence of the total ordering relationship in the range of each attribute. In the rest of this paper, we will only consider cases that the total ordering relationship exists.

In order to compute the QoS of services composed through sequential or parallel execution from constituent services s_1, …, s_n having QoS attributes equal to $(rt_1, c_1, rel_1), \ldots, (rt_n, c_n, rel_n)$, respectively, the formulas given in Table 1 [16] can be used. These computation formulas are in line with the Q-algebra operations introduced in [30] and generalized in [31].

As we can see from Table 1, the response time of a sequential composition is equal to the sum of its components' response time, while the response time of a parallel composition is equal to the maximum value. This difference is important in the adaptation process, since different search strategies should be employed to optimally adapt the scenario to the client's QoS specification. Consider for example the case of a BPEL scenario includes sequential invocations to

A and *B*, which is invoked with the setting $W=(1, 1, 0)$ for both service invocations. If the repository of available services were as listed in Table 2, then the adaptation engine should select services (A_2, B_2), with this composition scoring 2 $=(\text{sum}(rt_{A2}, rt_{B2})*1 + \text{sum}(c_{A2}, c_{B2})*1)$, a score higher than any other composition. In a parallel composition however, the adaptation engine should select (A_2, B_1), since these provide an overall score of 1.7 $=(max(rt_{A2}, rt_{B1})*1 + \text{sum}(c_{A2}, c_{B1})*1)$, as opposed to 1.3 of ($A_2$, B_2).

Table 1. QoS of composite services

	QoS attribute		
	responseTime	cost	reliability
Sequential composition	$\sum_{i=1}^{n} rt_i$	$\sum_{i=1}^{n} c_i$	$\prod_{i=1}^{n} rel_i$
Parallel composition	$\max_i(rt_i)$	$\sum_{i=1}^{n} c_i$	$\prod_{i=1}^{n} rel_i$

Table 2. Sample repository contents

Service	responseTime	cost	reliability
A_1	0.6	0.5	0.8
A_2	0.8	0.4	0.7
B_1	0.2	0.5	0.9
B_2	0.7	0.1	0.7

2.2 Subsumption relation representation

In order to perform adaptation, either QoS-based or collaborative filtering-based, we need to represent which services implement the same functionality, and are thus candidate for invocation when this particular functionality is needed. In this work, we use subsumption relations between services [18] to represent this information. In [18] the following subsumption relations are defined:

1. *A exact B*, iff *A* provides the same functionality with *B*.

2. *A plugin B*, iff *A* provides more specific functionality than *B*. For instance *B* could provide *travel*, whereas A could provide *air travel*; in this case *A* could be used whenever the functionality of B is needed, since it delivers (a specialization of) the functionality delivered by B.
3. *A subsume B*, iff *A* provides more generic functionality than *B*. In this case *A* cannot unconditionally be used whenever the functionality of B is needed. For instance *B* could provide *air travel* while *A* provides *travel,* and using a *travel service* (instead of an *air travel* one) could result to transportation by car, which does not comply with the functionality of B.
4. *A fail B*, in all other cases; in this case, *A* cannot be substituted for *B*.

The rationale for adopting the subsumption relations are as follows:

a. subsumption relations effectively broaden the pool of services that can be used in the context of adaptation (a service *A* can be unconditionally substituted by a service *B* if *A exact B or A plugin B*, as opposed to strict equivalence where substitution is only possible when *A exact B*[1]), providing thus more flexibility in the formulation of the execution plan.
b. subsumption relations are suitable for supporting the task of similarity metric computation, which is essential for collaborative filtering-based adaptation.

[18] and [39] address the representation of subsumption relations between service categories (or *abstract tasks*, in horizontal adaptation terminology) using trees, with generic service categories being located towards the tree root and specific service categories being placed towards the leafs (the references above apply this arrangement also to concepts corresponding to service parameters). Since in this work we are interested not only in service categories but in concrete services also (because these will be actually invoked in the context of WS-BPEL scenario execution), we extend the

[1] Note that if *A subsume B*, this does not preclude using B instead of A, however this is only possible under certain conditions. In this work, we will not consider this case.

tree scheme used in [18] and [39] by considering not only *is-a* arcs in the tree (general/specific categories) but also *instance-of* arcs: an arc is drawn in the tree between service category *C* and concrete service *S*, if and only if *S* implements exactly the functionality specified by category C. To illustrate this representation, let us consider the case of a travel planning WS-BPEL scenario containing the following activities: ticket booking, hotel booking, and event attendance. In this case the subsumption relations, including categories and concrete services could be arranged as shown in Figure 1 (categories are denoted using a folder icon; concrete services are denoted using a bullet mark).

We list below how the exact and plugin subsumption relations (the ones sufficient for unconditional service substitution) can be computed using the tree representation. Since the goal of the adaptation is to select the concrete services to be invoked in place of an abstract task or a concrete service, we give only the rules for the cases where the right-hand side operand is a concrete service. In the following, c represents a category, while s1 and s2 represent services:

- *Rule Cext: c exact* s_1 iff *c* is the immediate parent of s_1 (e.g. *Air travel* and *SwissAir* in Figure 1).
- *Rule Cplg: c plugin* s_1 iff *c* is an ancestor of s_1 (e.g. *Ticket* and *SwissAir* in Figure 1).
- *Rule Sext:* s_1 *exact* s_2 iff ∃ *c*: *c* is the immediate parent of s_1 and *c* is the immediate parent of s_2 (e.g. *AirFrance* and *SwissAir* in Figure 1).
- *Rule Splg:* s_1 *plugin* s_2 iff ∃ *c*: *c* is the immediate parent of s_1 and *c* plugin s_2 (e.g. *SportsTicketBooker* and *NBAFinals* in Figure 1).

In all other cases, unconditional substitution cannot occur hence we compute a *fail* result for the operands. Listing 1 and Listing 2 illustrate the computation of the relevant subsumption relation between a service category and a service or between two services, respectively, given the tree representation suggested in [18] and [39]. The representation of subsumption relations illustrated in Figure 1 can be trivially extended to accommodate the QoS attribute values of concrete services, by simply attaching to each concrete service node *s* the vector $QoS_S=(rt_s, rel_s, pr_s)$ corresponding to the particular service's QoS metrics.

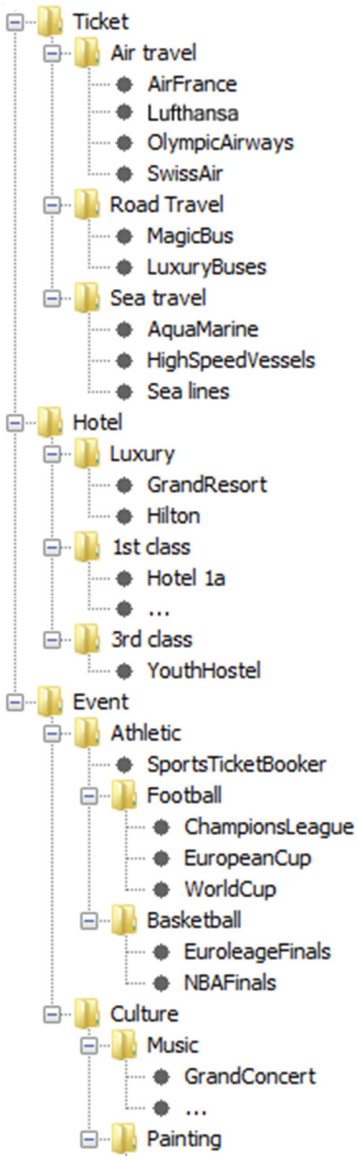

Figure 1. Subsumption relations for the travel planning WS-BPEL scenario

The *exact* subsumption relation is equivalent to *contract equivalence* [67]. Further [58] uses the *subcontract preorder* concept, denoted as $a \preccurlyeq b$, to denote that *b* offers "more capabilities than *a*, and therefore *b* can be substituted for *a*"; the preorder concept is therefore analogous to the *plugin* subsumption relation.

```
subsumptionRelCatSvc(ServiceCategory c, ConcreteService s, SubsumptionTree st) {
       SubsumptionTreeNode categNode = findNodeInTree(c, st);
       SubsumptionTreeNode svcNode = findNodeInTree(s, st);
       if (svcNode.parent == categNode)
              return exact;
       end if
       while (svcNode.parent ≠ null)
              svcNode = svcNode.parent;
              if (svcNode == categNode)
                     return plugin;
              end if
       end while
       return fail;
}
```

Listing 1. Computing the subsumption relation between a service category and a concrete service

```
subsumptionRelSvcSvc(ConcreteService s1, ConcreteService s2, SubsumptionTree st) {
       SubsumptionTreeNode svc1Node = findNodeInTree(s1, st);
       SubsumptionTreeNode svc2Node = findNodeInTree(s2, st);
       if (svc1Node.parent == svc2Node.parent)
              return exact;
       end if
       while (svc2Node.parent ≠ null)
              svcNode = svcNode.parent;
              if (svcNode2 == svc1Node.parent)
                     return plugin;
              end if
       end while
       return fail;
}
```

Listing 2. Computing the subsumption relation between two concrete services

The information regarding the QoS data and the subsumption relations can be stored in repository structures like the ones provided by OPUCE [33]. The population of the repository with QoS values can be performed either by using existing web service QoS datasets (e.g. the QWS dataset [34][35][36] or the WS-DREAM dataset [37][38]), or by using the methodologies described in [34][35][37] to compute the QoS attribute values for any desired set of target web

services. In all cases, the execution framework can monitor the actual quality of service delivered by the individual web services upon their invocation, and update the QoS attribute values in the repository accordingly, as described in [21].

2.3 Designations on specific service bindings

In the considered environment, the WS-BPEL consumer is allowed to make specific service bindings, i.e. request that some particular operation is performed using a designated service. For instance, in the travel planning WS-BPEL scenario discussed above, the consumer may request that ticket reservation is performed through the *SwissAir* service. The consumer may also designate that some functionality included in the WS-BPEL scenario is not executed; for instance, a tourist may not want to attend any event. Typically, the WS-BPEL code will examine input parameters and decide using a conditional execution construct (<*switch*>) whether to invoke the functionality or not.

Finally, functionalities that are neither explicitly bound to a specific service implementation, nor are designated as "not to be executed" are subject to adaptation, using the algorithm described in section 3.

2.4 Usage patterns repository

In order to perform collaborative filtering, a system needs to record evaluations (ratings) or choices (actions taken) made by users. This information is typically stored in a *ratings matrix* [22], where each row corresponds to a user and each column corresponds to an item, and the value of the (i, j) cell of this matrix indicates how user *i* has rated item *j* (or whether user *i* has taken action *j*). Since our aim is to adapt WS-BPEL scenario executions based on past user choices, each row of the ratings matrix in the proposed system corresponds to a particular WS-BPEL scenario execution, while each column corresponds to a concrete web service implementation. The value of a cell *(i, j)* is *1* if the service corresponding to column *j* were executed in the i^{th} execution, and *null* otherwise. We will call this matrix *usage patterns repository*. The execution framework can maintain this usage patterns repository by recording in an appropriate store which services were used in any particular execution of a BPEL scenario.

For notational convenience, in the rest of this paper we will represent the usage patterns repository in a more compact form: taking into account that each functionality included in the WS-BPEL scenario will result to at most one invocation of some service that implements it[2], for each row, at most one of the columns corresponding to the services implementing the functionality will have a value, while the other columns will be null. Thus we can equivalently denote the information within the usage patterns repository using one column for each distinct functionality of the WS-BPEL scenario, with the value of a cell *(i, j)* designating *the service used to deliver the specific functionality* if the functionality corresponding to column *j* were delivered in the context of the i^{th} execution, and *null* otherwise. Table 3 illustrates a usage patterns repository for the travel planning WS-BPEL scenario, using the compact notation. Note that the sparse representation is used internally, as is the case with all collaborative filtering algorithms. In executions 1, 2 and 5 all functionalities were invoked, while in the remaining executions some functionalities were omitted, as per user request.

Table 3. Example usage patterns repository

# exec	*Travel*	*Hotel*	*Event*
1	OlympicAirways	YouthHostel	ChampionsLeague
2	SwissAir	Hilton	GrandConcert
3	HighSpeedVessels	YouthHostel	
4	LuxuryBuses		EuroleagueFinals
5	Lufthansa	YouthHostel	GrandConcert
6	AirFrance	Hilton	
7	SwissAir	YouthHostel	ChampionsLeague

The usage patterns repository is populated by the execution adaptation architecture, described in section 4. When the execution adaptation architecture formulates the execution plan for a particular invocation (i.e. selects the concrete

[2] it may result to zero invocations, if the consumer has designated that this piece of functionality should not be executed. Note that the adaptation procedure presented here does not take into account loops, an aspect which will be addressed as part of our future work.

services to be selected to deliver the requested functionalities), the corresponding row is inserted in the usage patern repository.

3 The service recommendation algorithm

As stated in section 1, our approach follows the *horizontal adaptation* algorithm, i.e. it leaves the composition logic intact and adapts the execution by selecting which concrete service implementation will be used in each specific invocation. In order to perform this task, the algorithm takes into account the following criteria:

i) The consumer's QoS specifications (bounds and weights).

ii) Designations on which exact services should be invoked, if such bindings are requested by the consumer.

iii) Designations on which functionalities should *not* be invoked (e.g. in the example within section 2, the user does not want to attend any event).

iv) The QoS characteristics of the available service implementations.

v) The service subsumption relations.

vi) The usage patterns of services recorded in past WS-BPEL scenario executions.

vii) Statistics on how well-populated the usage patterns repository is; these are used to adjust the weights assigned to the results of the collaborative filtering-based adaptation, considering the fact that a poorly populated usage patterns repository is bound to produce results of poorer quality. To this end, we exploit the concept of user-item *sparsity* [55]; details on this aspect are given below.

Items (i)-(iii) are provided per WS-BPEL scenario execution by the consumer, while items (iv)-(vii) are drawn from repositories maintained by the adaptation scheme. Note that it is not necessary for the end-user to directly specify the QoS bounds or weights for each individual service as an input parameter to the scenario: the user could specify the bounds and weights in a more generic form (e.g. *choose=cheapest*), which would then be mapped to specific bounds for the involved services' QoS attributes through code provided by either the WS-BPEL scenario designer or an

intermediate entity (e.g. the code behind a web page collecting the necessary data from the user, in order to invoke the BPEL scenario).

Recall also that the approach adopted in this paper incorporates two different candidate service ranking algorithms, the first examining the QoS aspects only and the second being based on collaborative filtering techniques. The algorithms run in parallel to formulate their suggestions regarding the services that should be used in the adapted execution, and subsequently their suggestions are combined, through a metasearch score combination algorithm with varying weights. In the following subsections, we will describe the two algorithms (the QoS-based and the collaborating filtering-based) as well as on the combination step. Note that, for the metasearch score combination algorithm to work properly, the scores produced by each individual algorithm should be normalized [56][57][58]. To this end, the individual algorithms adopted from [53] and [54] have been extended to include a score normalization step; additionally the algorithms have been modified to allow them to function as "voters" in the metasearch process adopted in this paper. Figure 2 depicts the operation of the algorithm in the form of an activity diagram.

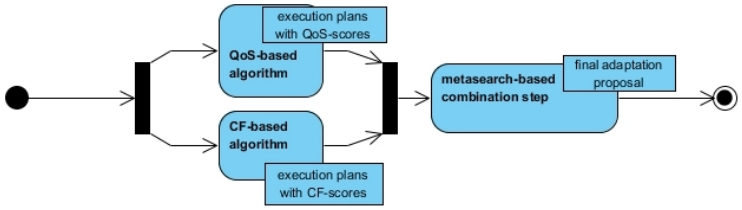

Figure 2. Activity diagram for overall algorithm operation

To illustrate the operation of the algorithms, we will use an example execution request:

TravelPlanner(bindings=(AirTravel(R), GrandResort, NBA), QoSLimits={MIN$_{AirTravel}$*(rt: 4, cost: 3, rel: 3), MAX*$_{AirTravel}$*(rt: 7, cost: null, rel: null)}, QoSWeights={rt: 0.2, cost: 0.5, rel: 0.3})*

which can be effectively read as "I want to stay in Grand Resort, I want to watch an NBA match and I want a recommendation for an air travel; response time for the *AirTravel* service should be at least 4 and at most 7, cost should

be at least 3 (recall that QoS attribute values are expressed in a "larger values are better" encoding scheme, therefore setting a lower bound for the cost excludes the *most expensive* services) and reliability should be at least 3.

We assume that the values of the QoS attributes for the services implementing the AirTravel functionality are as shown in Table 4.

Table 4. QoS attribute values for services

AirTravel Company	*Response_Time*	*Cost*	*Reliability*
Swiss Air	5	7	8
Olympic Airways	4	1	6
Air France	5	8	9
Lufthansa	6	3	4

3.1 The QoS-based algorithm

The QoS-based algorithm initially identifies the services which are candidate to be used for delivering functionalities in the context of the current WS-BPEL scenario, according to their QoS attribute values, as well as the QoS bounds specified by the user for the particular invocation, and subsequently computes a score for each *candidate composition*. Figure 3 illustrates the steps of the QoS-based algorithm in the form of an activity diagram and Figure 4 describes the algorithm in pseudocode, while the following paragraphs provide details on the actions taken within each step.

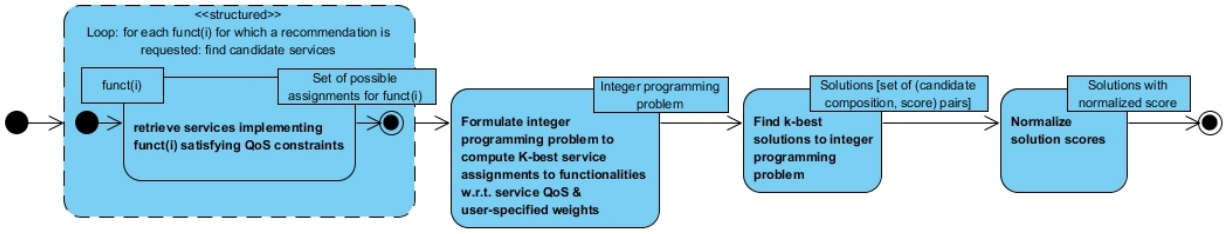

Figure 3. Activity diagram for the QoS-based algorithm

/* QoS-based adaptation algorithm pseudocode
Assumption:
- Scenario includes functionalities (i.e. invocations to services) f_1, f_2, …, f_n

Inputs:

- MIN and MAX (lower and upper bounds for QoS attributes) and weights
- Specification of bindings of functionalities to concrete services $B=(b_1, b_2, \ldots, b_n)$ [b_i == null if no binding provided, else id of service)
- Specifications of functionalities not to be invoked $O=(o_1, o_2, \ldots, o_n)$ [o_i == true if f_i should not be invoked, false otherwise]
- Specifications of functionalities for which recommendations are requested $R=(r_1, r_2, \ldots, r_n)$ [r_i == category of functionality if a recommendation is requested for f_i, null otherwise]
- Subsumption relation tree, including the QoS attribute value of services
- Scenario structure (information about which functionalities are executed sequentially and which in parallel)

Outputs:

- List of <execution plan, score> pairs, with "execution plan" binding services that recommendation is asked for to concrete services

*/

/* 1. For each functionality r_i that a recommendation is requested for, compute the set of services QPA(i) that deliver this functionality (as can be determined by the subsumption relations) and satisfying QoS thresholds. */
foreach $f_i \in R$, $f_i \neq$ null: $QPA(i) = \{s \in serviceRepository: (f_i \text{ exact } s \vee f_i \text{ plugin } s) \wedge MIN \leq QoS(s) \leq MAX\}$

/* 2. formulate and solve integer programming problem, obtaining the top-20 execution plans, taking into account the QoS attribute values and the weights of the QoS attributes */
ipp = formulateIntegerProgrammingProblem(QPA)
QoS_solutions = obtainTop20Solutions(ipp)
/* 3. Finally, normalize scores */
QoS_proposal = normaliseScores(QoS_solutions);

Figure 4. Pseudocode for the QoS-based algorithm

1. for each distinct functionality $funct_i$ within the WS-BPEL scenario which is subject to adaptation (i.e. for each service invocation that the user (a) has not designated an exact service binding and (b) has not designated that the functionality should not be invoked), the service implementations that deliver this functionality are retrieved from the repository. Recall that service functionalities are represented using the subsumption relations tree, and in terms of this tree the retrieval of services implementing a specific functionality is equivalent to retrieving services satisfying either the *Sext* or the *Splg* rule (i.e. services with either identical or more specific functionality, compared to the requested one). Equivalent services retrieval is further constrained by the QoS thresholds specified in the respective invocation. Thus, for each functionality $funct_i$, this step computes a set of possible concrete service assignments $QPA(i)=\{s_{i,1}, s_{i,2}, \ldots, s_{i,T(i)}\}$. Formally, QPA(i) is computed as follows:

$$QPA(i) = \{s \in Repository: (funct_i\ exact\ s\ \vee funct_i\ plugin\ s)\ \wedge\ [(min_{rt,i} \leq rt_s \leq max_{rt,i})\ \wedge\ (min_{c,i} \leq c_s \leq max_{c,i})\ \wedge\ (min_{rel,i} \leq rel_s \leq max_{rel,i})]\}$$

If, for some service that is subject to adaptation, no candidates satisfying the thresholds are found (i.e. QPA(i)=∅), then the algorithm terminates producing no results, indicating thus that the constraints specified by the user cannot be satisfied.

In our example, services *Swiss Air*, *Olympic Airways*, *Air France* and *Lufthansa*, satisfy the *Sext* rule, however service *Olympic Airways* fails to meet the QoS constraint regarding the maximum cost. Hence QPA(AirTravel)={*Swiss Air*, *Air France*, *Lufthansa*}.

2. the algorithm computes the *m-best solutions*, with respect to the QoS value, with each solution being a set of concrete service assignments to the functionalities for which a recommendation has been requested. Since the service consumer's QoS bounds per functionality are an input to the adaptation algorithm, service selection can proceed by exploiting these constraints, performing local selection so as to efficiently compute the optimal concrete service assignments for the functionalities under adaptation [59][58][60][61][62][63][64]. In this work, we adopt the concrete service utility function used in [59][61], which is:

$$U(s_{j,i}) = \sum_{k=1}^{3} \frac{Q_{max}(j,k) - q_k(s_{j,i})}{Q_{max'}(k) - Q_{min'}(k)} * w_k$$

where $q_k(s_{j,i})$ is the value of the k[th] QoS attribute of concrete service $s_{j,i}$ (the first QoS attribute being response time, the second one being cost and the third one being availability), w_k being the weight assigned to the [kth] QoS attribute, $Q_{max}(j,k) = \max_{s \in QPA(j)} q_k(s)$ [i.e. the maximum value of QoS attribute *k* among possible concrete service assignments for functionality *j*], and $Q_{max'}(k)$ [resp. $Q_{min'}(k)$] being the overall maximum (resp. minimum) value of QoS attribute *k* within the repository. Note that services with high QoS values have low

utility function values. Given the utility function, the computation of the *m-best solutions* can be formulated as an integer programming optimization problem as follows: minimize the overall utility value given by:

$$OUV_{QoS} = \sum_{i=1}^{F} \sum_{j=1}^{T(i)} U(s_{j,i}) * x_{j,i}$$

where F is the number of functionalities $funct_k$ requiring adaptation, and each $x_{j,i}$ is a binary variable taking the value 1 if $s_{j,i}$ is selected for delivering functionality $funct_j$, and the value 0, otherwise. Since each functionality $funct_j$ is delivered in the final execution plan by exactly one concrete service, the minimization of the overall utility value is subject to the constraint

$$\sum_{i=1}^{T(j)} x_{j,i} = 1, 1 \leq j \leq F$$

Note that in the above problem formulation an additive formula is used for all QoS attributes, while the formulas listed in Table 1 for composite QoS service computation are not all additive; in particular the reliability attribute of a composite service is shown in Table 1 to be computed as a product, while the response time of a parallel composition is computed as the maximum of the constituent services' response time. In order to transform the non-additive formulas to additive ones, suitable for use in the context of an integer programming problem, we have employed the following techniques:

- regarding the product functions used for computing the reliability of a composite service, we have adopted the approach used in [4][73][74], according to which the logarithm of reliability, rather than the reliability itself is used when writing expressions. Through the use of the logarithm, the equality

  $$\log(\prod_i reliability(i)) = \sum_i \log(reliability(i))$$

  is exploited, and therefore the product function is transformed to a sum function. Observe also that in the computation of the $U(s_{j,i})$ function listed above, the literal value any QoS attribute (including

reliability) is normalized taking into account the minimum and maximum values of the attribute, the range of the normalized attribute value used in the utility function is always in the range [0, 1].

- regarding the *max* function used in computing the response time of a parallel composition, it is possible to follow the method described in [75] to transform an objective minimization problem where the objective contains the *max* function (termed a *minimax objective*) to an integer programming problem. In brief, the transformation procedure of an objective function containing the term

$$\max_{k \in K} \sum_{j \in J} c_{kj} x_j$$

involves the introduction of a new variable z representing the above term, and the introduction of the following extra constraints:

$$\sum_{j \in J} c_{kj} x_j \leq z, \forall k \in K$$

A relevant example is given in [76]. It is however worth noting that the IBM ILOG CPLEX optimizer 12.6.0.0 [65] used in our experiments, directly supports the usage of the *max* operator within the objective function, even in a recursive structure (i.e. the operands of a *max* operator may be *max* operators themselves, as would be the case with nested *flow* constructs in a BPEL scenario), hence we did not need to apply the transformation described above; the transformations can be used, if the integer programming optimization software used does not support the direct use of the *max* operator in the objective function.

If service selection affinity needs to be maintained between functionalities $funct_i$ and $funct_j$, then an additional constraint is added to the problem as follows:

i. Let QPA(i)=$\{s_{i,1}, s_{i,2}, \ldots, s_{i,L(i)}\}$ be the possible concrete service assignments for $funct_i$ and QPA(j)=$\{s_{j,1}, s_{j,2}, \ldots, s_{j,L(j)}\}$ the possible concrete service assignments for $funct_j$, computed in step 1, above. Without loss of

generality, we assume that the first *k* services in QPA(i) and QPA(j) are offered by the same service provider (i.e. services for which the *host* part of their endpoint address [1] is identical), while the remaining services are offered by different providers, or more formally:

$$\begin{cases} provider(s_{i,a}) = provider(s_{j,a}), 1 \leq a \leq k \\ provider(s_{i,x}) \neq provider(s_{j,y}), 1 \leq x \leq k, 1 \leq y \leq k, x \neq y \end{cases}$$

In this case, implementing the functionalities ($funct_i$, $funct_j$) with any choice of services ($s_{i,a}$, $s_{j,a}$) for $1 \leq a \leq k$ leads to maintenance of service selection affinity, while any other choice (i.e. $a > k$, or implementing the functionalities with a choice of services ($s_{i,x}$, $s_{j,y}$) with $x \neq y$) leads to a failure in maintaining service selection affinity.

ii. The constraints

$$x_{i,a} - x_{j,a} = 0, 1 \leq a \leq k$$

are added to the problem; under the presence of these constraints, a service $x_{i,a}$ implementing functionality $funct_i$ and provided by service provider a will be selected if and only if for the realization of functionality $funct_j$ the service $x_{j,a}$ provided by the same provider is selected too. For optimization purposes, all elements corresponding to services $s_{i,b}$ and $s_{s,b}$: $b > k$ are removed from the model, since they cannot be part of any feasible solution.

This procedure can be generalized for cases that service selection affinity needs to be maintained between any number of functionalities: if service selection affinity needs to be maintained between functionalities $funct_{i1}$, $funct_{i2}$, …, $funct_{if}$, then the constraints

$$x_{i1,a} - x_{i2,a} = 0, 1 \leq a \leq k$$

…

$$x_{i1,a} - x_{if,a} = 0, 1 \leq a \leq k$$

Similarly to the case of maintaining service selection affinity between two functionalities, we assume without loss of generality that

$$\begin{cases} provider(s_{i1,a}) = provider(s_{i2,a}), 1 \le a \le k \\ \dots \\ provider(s_{i1,a}) = provider(s_{if,a}), 1 \le a \le k \\ provider(s_{p,x}) \ne provider(s_{q,y}), 1 \le p \le f, 1 \le q \le f, p \ne q, 1 \le x \le k, 1 \le y \le k, x \ne y \end{cases}$$

Subsequently, the integer programming problem is solved, and the *k* best solutions are obtained. Obtaining the *k* best solutions can be achieved by solving the problem once, and then adding a constraint rendering the obtained solution infeasible (for the IBM ILOG CPLEX optimizer [65] used in our experiments, this procedure is described in [66]). In our implementation, we have set *k* to 20; the value of 20 has proven to be sufficient even for applying metasearch techniques to web search engines [68] (where the size of each individual list is considerably higher than the number of alternatives in our case), i.e. further increasing the number of solutions that each voting algorithm provides to the combination phase, does not improve the quality of the result computed by the combination step.

In our example, a single functionality requires adaptation (AirTravel); the utility function for this functionality is

$$U(s_{AirTravel,i}) = \sum_{k=1}^{3} \frac{Q_{max}(AirTravel,k) - q_k(s_{AirTravel,i})}{Q_{max'}(k) - Q_{min'}(k)} * w_k$$

Substituting the values for $Q_{max}(k)$, $Q_{max}(k)$, $Q_{min}(k)$ and w_k, the utility function can be rewritten as:

$$U(s_{AirTravel,i}) = \underbrace{\frac{5 - rt(s_{AirTravel,i})}{5-4} * 0.2}_{runtime} + \underbrace{\frac{8 - cost(s_{AirTravel,i})}{9-2} * 0.5}_{cost} + \underbrace{\frac{9 - rel(s_{AirTravel,i})}{9-4} * 0.3}_{reliability}$$

The utility function values for services in QPA(AirTravel) are as shown in Table 5. Since assignment is performed only for a single functionality, each possible solution obviously consists of selecting one of the candidates in QPA(AirTravel) to implement the AirTravel functionality. In terms of the integer programming

problem solution, the variable $x_{AirTravel,j}$ for this service will be equal to 1 and the variables $x_{AirTravel,k}$ for the remaining services will be 0. The value of the overall utility function for a solution, coincides with this of the sole service selected in this solution. All possible solutions are kept, since their number is less than 20.

Table 5. Utility function values and IP problem solutions for services in QPA(AirTravel)

IP problem solution	*Selected* $s_{AirTravel,i}$	$U(s_{AirTravel,i})$	OUV_{QoS}
$x_{AirTravel,SwissAir}$=1, $x_{AirTravel,AirFrance}$=0, $x_{AirTravel,Lufthansa}$=0	Swiss Air	0.231	0.231
$x_{AirTravel,SwissAir}$=0, $x_{AirTravel,AirFrance}$=1, $x_{AirTravel,Lufthansa}$=0	Air France	0.100	0.100
$x_{AirTravel,SwissAir}$=0, $x_{AirTravel,AirFrance}$=0, $x_{AirTravel,Lufthansa}$=1	Lufthansa	0.657	0.657

3. The scores of the solutions produced by step 2, above, are then normalized in the range [0,1], to enable the combination step to function correctly. Normalization is performed by first computing the minimum and the maximum execution plan scores among all solutions formulated in step 3, above, and then applying the formula

$$normedQoS_{Score_i} = 1 - \frac{QoS_{Score_i} - minQoS_{Score}}{maxQoS_{Score} - minQoS_{Score}}$$

[58] (adapted from [57]; since the target of step (2) was to *minimize* the overall utility function OUV_{CF}, here we subtract the normalization formula suggested in [57] from 1, in order to assign the normalized score 1 to the *best solution* and the normalized score 0 to the worst). The output of the algorithm is the set of candidate execution plans, with each execution plan being tagged with the relevant normalized score (*QoS-score*).

In our example, the normalized solution scores are as shown in Table 6.

Table 6. Normalized solution scores

$s_{AirTravel,i}$	Normalized score
Swiss Air	0.764
Air France	1.000
Lufthansa	0.000

3.2 The collaborating filtering-based algorithm

The collaborating filtering-based algorithm, upon each invocation of a WS-BPEL scenario, proposes sets of actual services that functionalities denoted as "to be recommended" could be bound to. Each proposal is tagged with a *CF-score*, which is determined through a collaborative filtering process. Figure 5 illustrates the steps of the CF-based algorithm in the form of an activity diagram and Figure 6 describes the algorithm in pseudocode, while the following paragraphs provide details on the actions taken within each step.

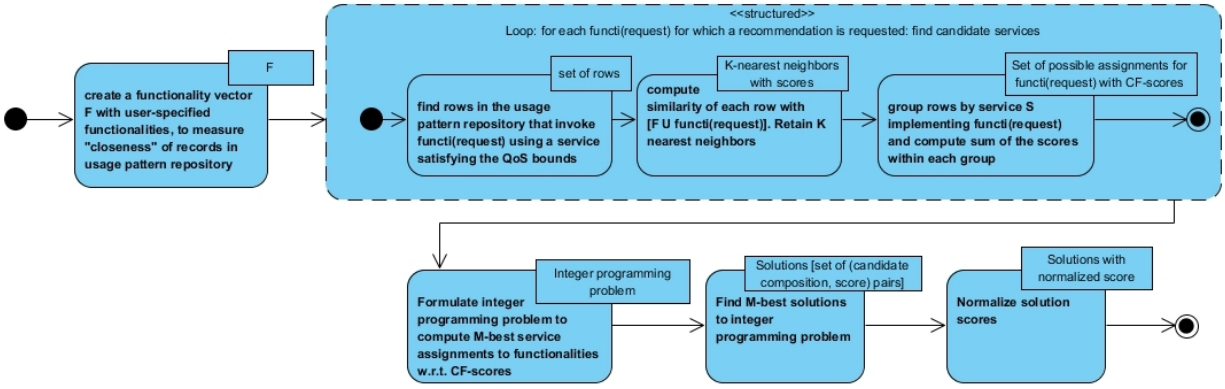

Figure 5. Activity diagram for the CF-based algorithm

/* CF-based adaptation algorithm pseudocode

Assumption:

- Scenario includes functionalities (i.e. invocations to services) $f_1, f_2, \ldots, f_n$

Inputs:

- MIN and MAX (lower and upper bounds for QoS attributes)
- Specification of bindings of functionalities to concrete services $B=(b_1, b_2, \ldots, b_n)$ [b_i == null if no binding provided, else id of service)
- Specifications of functionalities not to be invoked $O=(o_1, o_2, \ldots, o_n)$ [o_i == true if f_i should not be invoked, false otherwise]
- Specifications of functionalities for which recommendations are requested $R=(r_1, r_2, \ldots, r_n)$ [r_i == category of functionality if a recommendation is requested for f_i, null otherwise]
- Subsumption relation tree, including the QoS attribute value of services

Outputs:

- List of <execution plan, score> pairs, with "execution plan" binding services that recommendation is asked for to concrete services

```
*/
/* 1. Build scenario-level functionality vector */
for (i = 1; i <= n; i++)
   if (B[i] != null) then
      F[i] = B[i]; /* set explicit binding information in the functionality vector */
   Else
      F[i] = null;
   end if
end for
/* 2. For each functionality that a recommendation is requested for, fetch matching scenario executions from the usage patterns repository */
for (i = 1; i <= n; i++)
   if (R[i] != null) /* recommendation requested */
      /* Fetch matching rows from the usage patterns repository */
      matchingRows = selectFromUsageRepository row: R[i] exact row[i] ∨ R[i] plugin row[i];
   /* Formulate FV[fi], to be used as the similarity yardstick for functionality fi */
      FV[fi] = F;
      FV[fi][i] = R[i]
/* 3, 4 drop rows not in bounds and compute score for remaining ones */
      foreach (row in matchingRows)
         if (notInBounds(row, MIN, MAX) then
            removeElement(matchingRows, row);
         else
            scores[row] = computeSimilarity(row, FV[fi]);
      end for
/* 5. Retain k-nearest scores and compute group scores */
      qualifyingRows = top30_Scores(matchingRows, scores);
      /* formulate groupScores, containing the candidate services for delivering functionality fi and the respective scores */
      groupedScores[fi] = groupRowScores(qualifyingRows, scores, fi);
   end if /* recommendation requested */
end for
/* 6. formulate and solve integer programming problem, obtaining the top-20 execution plans */
ipp = formulateIntegerProgrammingProblem(groupedScores[fi]: R[i] != null)
CF_solutions = obtainTop20Solutions(ipp)
/* 7. Finally, normalize scores */
CF_proposal = normaliseScores(CF_solutions);
```

Figure 6. Pseudocode for the CF-based algorithm

1. it formulates a scenario-level functionality vector F=(f_1, f_2, …, f_n), where each f_i corresponds to a functionality that is part of the WS-BPEL scenario. The values of the elements f_i are determined as follows:
    - if functionality corresponding to element f_i is bound to a specific service, then the value of f_i is set to the identifier of this service.
    - in all other cases (i.e. if the corresponding functionality will not invoked in the context of the particular WS-BPEL scenario execution or a recommendation is requested for it), the value of f_i is set to *null*.

    Regarding our example, the functionality vector would be set to F=(null, GrandResort, NBA).
2. for each functionality $funct_i(request)$ for which a recommendation is requested, the algorithm retrieves from the usage patterns repository the rows for which

    $funct_i(request)$ *exact* $funct_i(row)$ *or* $funct_i(request)$ *plugin* $funct_i(row)$

    i.e. those patterns containing either an invocation to some identical functionality or a functionality to an invocation to a more specific one. These are the only rows that are useful for formulating a recommendation for $funct_i(request)$, since they involve services that deliver the requested functionality.

    Additionally, a request-level functionality vector F($funct_i$(request)) is formulated, by replacing the null value corresponding to $funct_i$ in vector F with the category corresponding to functionality $funct_i$. The new functionality vector will be used to calculate the similarity of the current request with the usage patterns in the repository, as explained below.

    In our example, rows 1, 2, 5, 6 and 7 of Table 3 would be selected (rows 3 and 4 do not satisfy the subsumption relation criteria). The functionality being considered is $funct_i$=AirTravel and it corresponds to the first element of functionality vector F=(null, GrandResort, NBA) generated in step 1; hence, the functionality vector F(AirTravel) will be formulated with F(AirTravel)=(AirTravel, GrandResort, NBA).

3. the rows for which the QoS characteristics of service $funct_i(row)$ do not satisfy the bounds set through vectors $MIN(funct_i)$ and $MAX(funct_i)$ are dropped; effectively, these rows cannot be used in the recommendation, since they involve services whose QoS characteristics do not satisfy the consumer's requirements.

   Regarding our example, row #1 would be dropped, since the OlympicAirways service implementing the functionality *AirTravel* does not satisfy the bounds related to the cost QoS attribute; therefore only rows 2, 5, 6 and 7 would be retained.

4. for each row retained by step 3, the algorithm computes a similarity score, indicating its similarity with the current request, as the latter is represented by the F(funct$_i$(request)) functionality vector. The similarity score is calculated using the Sørensen similarity index [40] (alternatively known as Dice's coefficient [41]), according to which the similarity of two sets A={a_1, a_2, …, a_n}, B={b_1, b_2, …, b_m}, is equal to $S(A,B) = \frac{2|A \cap B|}{|A|+|B|}$, properly modified to suit a domain with semantic similarities. The modification follows the approach used in the fuzzy set similarity index calculation, where the cardinality of the intersection of two sets (i.e. the nominator in the Sørensen similarity index formula) is computed as the sum of the probabilities that a member belongs to both sets [42]. Correspondingly, when set member similarity is considered, the nominator of the fraction is replaced by $2 * \sum_i sim(a_i, b_i)$, where $sim(a_i, b_i)$ is a metric measuring the similarity between a_i and b_i; analogous approaches are adopted in ontology alignment and ontology matching domains, e.g. [43]. Effectively, the cardinality of the sets' intersection (i.e. the nominator of the fraction) used in cases that only the "equals" operator is available, is replaced by the sum of similarities of the corresponding elements. As a similarity metric between two functionalities (web services or categories), we adopt the one proposed in [39]:

   $sim(s_1,s_2) = C – lw*PathLength – NumberOfDownDirection$

   where:

- C is a constant set to 8 [39][44].
- *lw* is the level weight for each path in subsumption tree (cf. Figure 1). The value of *lw* depends on the depth of the subsumption tree and the level of the node in it. To count the level weight lw, the formula $lw = \frac{subTreeDepth-(ln-1)}{subTreeDepth}$ [39] is used, where *subTreeDepth* is the depth of the subsumption tree, and *ln* is the level of node s_2 in the subsumption tree (the root node has a level equal to 1, its immediate children a level equal to 2 and so forth).
- PathLength is the number of edges counted from functionality s_1 to functionality s_2.
- NumberOfDownDirection is the number of edges counted in the directed path between functionality s_1 and s_2 and whose direction is towards a lower level in the subsumption tree.

We further normalize this similarity metric dividing the result computed in the above formula by 8; this way, the similarity metric is always in the range [0, 1] and so is the value of the modified Sørensen similarity index, consistently with its original definition.

Regarding our example, the similarity measures between the F(AirTravel) functionality vector and the rows retained by the third step of the algorithm are:

sim(F(AirTravel), row2) = (19/24, 19/24, 10.5/24)

sim(F(AirTravel), row5) = (19/24, 14/24, 10.5/24)

sim(F(AirTravel), row6) = (19/24, 19/24, 0)

sim(F(AirTravel), row7) = (19/24, 14/24, 15/24)

and consequently the modified Sørensen similarity index values between F(AirTravel) and these rows are:

S(F(AirTravel), row2) = 2*(19/24+19/24+10.5/24) / 6 = 0.674

S(F(AirTravel), row5) = 2*(19/24 + 14/24 + 10.5/24) / 6 = 0.604

S(F(AirTravel), row6) = 2*(19/24+19/24+0) / 5 = 0.633

S(F(AirTravel), row7) = 2*(19/24+14/24+15/24) / 6 = 0.667

5. Finally, the algorithm retains only the K-nearest neighbors (i.e. the rows with the highest similarity scores), groups the retained rows by the value of the service implementing the $funct_i(request)$ functionality and computes the sum of the scores within each group. Services implementing the $funct_i(request)$ functionality that are associated with higher sums are deemed more suitable, in the sense of collaborative filtering, for being used in order to deliver the functionality $funct_i(request)$. In our implementation, we have set K to 30. This value is higher than the commonly used value of 10 [45] (the value of 10 is also used in the original version of the algorithm [54]): the rationale behind choosing a higher value is to produce a longer list of services that are proposed by the collaborating filtering algorithm, providing thus step 6 that that follows with more options and allowing it to make a choice best suiting the user's request. Effectively, for functionality $funct_i(request)$ this step will compute a set of possible concrete service assignments $CFPA(i)=\{s_{i,1}, s_{i,2}, \ldots, s_{i,L(i)}\}$, with the cardinality of $CFPA(i) = L(i) \leq 30$ (the number of proposed solutions may be less than 30, if less than 30 distinct services are recorded in the usage patterns repository to have been used for delivering $funct_i(request)$). For each possible concrete service assignment $s_{i,j}$, the respective similarity score $CFS(s_{i,j})$ has also been computed.

   In our example, all rows would be retained, since we only have three rows. Since in row 2 and row 7 the same service (SwissAir) is used to deliver the AirTravel functionality, their individual scores would be summed up to formulate the solution *SwissAir* with a score of 1.341. The *AirFrance* service is the second solution with a score of 0.633 (only row 6 contributes to it) and the *Lufthansa* service is the third solution with a score of 0.604 (only row 5 contributes to it).

Steps 2-5 are repeated for each functionality $funct_i(request)$ for which a recommendation is requested.

6. After the lists of candidates for each individual service that is subject to adaptation have been computed, the algorithm selects the *top-20* execution plans with respect to their *CF-score*. The rationale behind choosing the value of 20 has been discussed in subsection 3.1, above. Given an execution plan containing services ($s_{1,i}$, $s_{2,j}$,

…, $s_{N,k}$) with the similarity scores of the services computed in step 5 being (CFS($s_{1,i}$), CFS($s_{2,j}$), …, CFS($s_{N,k}$)), then the *CF-score* of the execution plan is equal to CFS($s_{1,i}$)+CFS($s_{2,j}$)+…+CFS($s_{N,k}$). Using the *sum* function to aggregate the individual service scores into the execution plan *CF-score* ensures that the final result reflects the matching scores of all involved services. Contrary, the use of the *max* function would reflect the matching score of the best matching service only and therefore would render the algorithm prone to selecting an execution plan for which a single service has a very high score, but all other services have very poor ones.

Producing the *top-20* execution plans is modelled as an integer programming optimization problem, which is formulated as follows: maximize the overall utility value given by:

$$OUV_{CF} = \sum_{i=1}^{F} \sum_{j=1}^{L(i)} CFS(s_{i,j}) * x_{i,j}$$

where F is the number of functionalities $funct_k(request)$ requiring adaptation, and each $x_{i,j}$ is a binary variable taking the value 1 if $s_{i,j}$ is selected for delivering functionality $funct_i(request)$, and the value 0, otherwise. Since each functionality $funct_i(request)$ is delivered in the final execution plan by exactly one concrete service, the maximization of the utility value is subject to the constraint

$$\sum_{j=1}^{L(i)} x_{i,j} = 1, 1 \leq i \leq F$$

If service selection affinity needs to be maintained between functionalities $funct_i(request)$ and $funct_j(request)$, then an additional constraint is added to the problem, following the procedure described in subsection 3.1 (step 2), above.

Subsequently, the integer programming problem is solved, and the *k* best solutions are obtained. Obtaining the *k* best solutions can be achieved by solving the problem once, and then adding a constraint rendering the obtained solution infeasible (for the IBM ILOG CPLEX optimizer [65] used in our experiments, this procedure is described in [66]). In our implementation, we have set *k* equal to 20, as explained in subsection 3.1.

In our example, adaptation has been requested for a single functionality, for which three possible solutions have been computed, therefore this step generates threesolutions, as shown

Table 7. Utility function values and CF problem solutions for services in QPA(AirTravel)

CF problem solution	*Selected* $s_{AirTravel,i}$	$CFS(s_{AirTravel,i})$	OUV_{CF}
$x_{AirTravel,SwissAir}=1$, $x_{AirTravel,AirFrance}=0$, $x_{AirTravel,Lufthansa}=0$	Swiss Air	1.341	1.341
$x_{AirTravel,SwissAir}=0$, $x_{AirTravel,AirFrance}=1$, $x_{AirTravel,Lufthansa}=0$	Air France	0.633	0.633
$x_{AirTravel,SwissAir}=0$, $x_{AirTravel,AirFrance}=0$, $x_{AirTravel,Lufthansa}=1$	Lufthansa	0.604	0.604

7. The final step in this algorithm is to normalize the range of the solution's *CF-scores*, expressed through the respective values of the overall utility function, to the range [0, 1], in order for the combination step to function correctly. Normalization is again performed using the formula

$$normedCF\_Score_i = \frac{CF\_Score_i - minCF\_Score}{maxCF\_Score - minCF\_Score}$$

[58], where *minCF_Score* and *maxCF_Score* are the minimum and maximum scores, respectively, of the solutions produced in step 6. The output of the algorithm is the set of candidate execution plans, with each execution plan being tagged with the relevant normalized score (*CF-score*).

Table 8. Normalized solution scores

$s_{AirTravel,i}$	Normalized score
Swiss Air	1.000
Air France	0.039
Lufthansa	0.000

Note that the elimination of rows not satisfying the QoS bounds (step 3) is in this case merely an optimization step: indeed, these rows could be normally processed having their similarity scores computed, and would then be dropped in the combination step; however, performing the filtering at this stage saves unnecessary work (and thus improves performance), while it also facilitates the operation of the combination step, since it will not need to check whether the solutions proposed by each algorithm are conferment to the constraints set by the user.

3.3 The combination step

The role of the combination step is to synthesize the results given by individual algorithms to produce to a single result. Recall from the previous two subsections that each algorithm produces a set of candidate execution plans, with each execution plan being tagged with the relevant normalized score (*QoS-score* or *CF-score*). Figure 7 presents the steps taken into the combination step in the form of an activity diagram, while the following paragraphs provide details on the actions taken within each individual step.

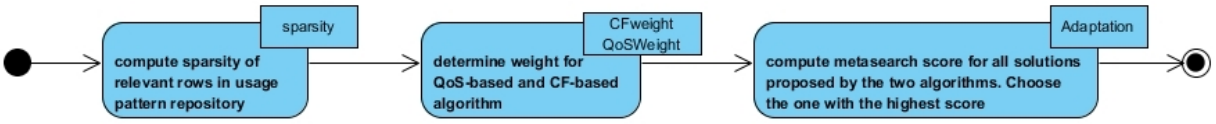

Figure 7. Activity diagram for the combination step

In order to combine the scores, one could use any standard metasearch score combination algorithms, such as CombMIN (minimum of individual scores), CombSUM (sum of individual results), CombMNZ (CombSUM, multiplied by the number of individual algorithms actually suggesting a particular solution[3]) and so forth, with CombMNZ being identified as having the best performance [57]; [58] however suggests that CombSUM and CombMNZ are best suitable for combining results from individual algorithms with relative similar performance. [58] argues that if the individual algorithms have diverse performance, it is best to specify a certain belief about the quality of the recommendations from individual algorithms, proposing thus WCombSUM and WCombMNZ, which extend the

[3] in our case, for an execution plan *ep:*

$$\text{CombMNZ}(ep) = \begin{cases} 2 * \left(\text{QoS\_score}(ep) + \text{CF\_score}(ep)\right), \text{iff } ep \text{ is suggested by both algorithms} \\ 1 * \text{QoS\_score}(ep), \text{iff } ep \text{ is suggested only by the QoS-based algorithm} \\ 1 * \text{CF\_score}(ep), \text{iff } ep \text{ is suggested only by the CF-based algorithm} \\ 0, \text{iff } ep \text{ is not suggested by either algorithm} \end{cases}$$

standard CombSUM and CombMNZ schemes by introducing a weight for the individual ratings produced by the algorithms. More specifically,

$$\text{WCombSUM}_i = \sum_{j=1}^{m_i} w_j * \text{NormalizedScore}_{i,j}$$

where $WCombSUM_i$ is the final score for a particular result i, w_j is a predefined weight associated with the proposing algorithm j, m_i is the number of nonzero scores of result i (i.e. number of algorithms proposing the particular result), and $NormalizedScore_{i,j}$ is the normalized score for result i produced by algorithm j. Also,

$$\text{WCombMNZ}_i = \text{WCombSUM}_i * m_i$$

[55] asserts that collaborating filtering is prone to producing results with low prediction accuracy when the sparsity (i.e. the ratio of empty cells to the total number of cells) of the rating matrix exceeds 99.5%. In more detail, [55] demonstrates that in the sparsity range [99.5%, 99.9%] the mean absolute rate of collaborative filtering increases sharply (and hence the recommendation quality drops) in a linear fashion, while beyond the sparsity limit of 99.9% there is no point in taking into account the results of collaborative filtering-based algorithms. For the sparsity range [0, 99.5%] [55] shows that collaborating filtering has inferior performance as compared to content-based collaborating filtering, exhibiting a mean absolute error rate higher by 6%-9%, with an average of 7.8%.

Taking the above into account, we adopt a modified version of the WCombMNZ metasearch score combination algorithm, in which the weight assigned to each of the algorithms (the QoS-based one and the collaborating filtering-based one) varies according to the sparsity of the rating matrix (i.e. the usage patterns repository). More specifically, the weights are calculated as follows:

$$CFweight = \begin{cases} 0.40, \text{ if sparsity} \le 0.995 \\ \dfrac{0.40 * (0.999 - \text{sparsity})}{0.004}, \text{ if } 0.995 < \text{sparsity} \le 0.999 \\ 0, \text{ if sparsity} > 0.999 \end{cases}$$

$$QoSweight = 1 - CFweight$$

Note here that the *sparsity* metric is calculated against the rows of the usage patterns repository that are retained by step (3) of the CF-based algorithm described in subsection 3.2, therefore the weights are individualized for each distinct adaptation. For efficiency purposes, the calculation of the sparsity is performed during the execution of the CF-based algorithm described in subsection 3.2, where the pertinent data are readily available, and is forwarded to the combination step, along with the list of the execution plans.

In this weight assignment setting, we consider the QoS-based algorithm as a counterpart of the content-based collaborating filtering algorithm examined in [55], under the analogy that the QoS-based algorithm examines *the values of item metadata* (QoS attribute value of services), instead of the item contents. The value of 0.40 for *CFweight* in the range [0, 0.995] has been selected due to the fact that collaborating filtering exhibits higher mean absolute error rate as compared to the content-based collaborating filtering algorithm, and hence its weight should be set lower than 0.5, according to the rationale of the WCombMNZ metasearch score combination algorithm. On the other hand, the value of 0.4 is large enough to allow the collaborative filtering algorithm to influence the final result. The appropriateness of value 0.4 has been also experimentally verified, as described in subsection 5.1.

After computing the WCombMNZ metasearch for all candidate execution plans, the combination step selects the execution plan with the highest score, which will be used to drive the adaptation process. If more than one candidate execution plans have the same score, one of them is selected randomly.

In our example, the sparsity of the table consisting of rows 2, 5, 6 and 7 is equal to 0.891. This stems from the facts that (a) each row in the usage patterns repository has 23 columns, i.e. equal to the number of concrete services in the subsumption tree[4], (b) rows 2 and 7 have 3 cells equal to "1" and (c) rows 5 and 6 have 2 cells equal to "1", making

[4] Note that the "compact notation" used in Table 3 to represent the usage pattern repository has been employed for notational convenience only; the usage pattern repository is analogous to the *ratings matrix* [22] used in collaborative filtering algorithms, as noted in subsection 2.4.

thus a total of 10 cells having a value of "1" out of a total of 92, or, inversely, 82/92 cells are empty, leading to a sparsity of 0.891. Using the rules for computing *CFweight* and *QoSweight* given above, we get *CFweight=0.40* and *QoSweight = 0.6*.

Using these weights, we may now compute the WCombMNZ value for each solution, as shown in Table 9.

Table 9. Computing the WCombMNZ score for the solutions

Solution	QoS-based algorithm score	CF-based algorithm score	WCombMNZ formula	WCombMNZ score
Swiss Air	0.764	1.000	2 * (0.6 * 0.764 + 0.4 * 1.000)	1.717
Air France	1.000	0.039	2 * (0.6 * 1.000 + 0.4 * 0.039)	1.231
Lufthansa	0.000	0.000	2 * (0.6 * 0.000 + 0.5 * 0.000)	0

4 The execution adaptation architecture

The execution adaptation architecture follows the middleware-based approach, which is common in QoS-based adaptation frameworks (e.g. [5][11][46]). In this approach, an *adaptation layer* intervenes between the BPEL execution engine and the actual service providers, selecting the services that will actually be invoked in the context of any single WS-BPEL scenario execution, using the algorithm described in section 3 and arranging for redirecting the actual invocations to the selected services. Redirection is performed through a specially crafted web service, namely *adaptInvocation*. The adaptation layer implements three additional utility web services, namely *getSessionId* (assigning unique session identifiers to individual WS-BPEL scenario executions), *prepareAdaptation* (accepting information about the invocations that should be adapted, QoS bounds and service binding information, and executing the algorithm of section 3 to determine the execution plan that should be followed in the particular WS-BPEL scenario execution) and *releaseSession* (cleaning up the information regarding the WS-BPEL scenario execution, upon its termination). Furthermore, the execution adaptation architecture includes a *preprocessor module* which transforms "ordinary" (i.e. nonadaptive) WS-BPEL scenarios to a form that their execution can be readily adapted by the middleware. The preprocessor module relieves the burden of having to redesign the BPEL scenarios, limiting the necessary changes to

(a) allowing the user to specify the QoS requirements and the service bindings (including requests for recommendation) at the front-end application (e.g. a web form) and (b) preprocessing and redeploying the WS-BPEL scenarios. Using existing WS-BPEL scenario, where functionality invocations are specified by means of concrete services, allows for using standard WS-BPEL scenario editors, instead of specialized software that would be required if functionalities were specified by means of abstract tasks.

Figure 8 presents the overall execution adaptation architecture. In the following, we will elaborate on the operation of the preprocessor and the operation of the adaptation layer.

4.1 Specifying the QoS information in the scenario

The first step towards enabling the QoS-based adaptation is the specification of the required QoS bounds for service invocations and the specification of the weights of each QoS attribute. In order to provide this feature in a WS-BPEL compliant fashion, the proposed framework adopts the following conventions:

1. the designer should use the WS-BPEL variable *QoS_weights* to designate the weight vector W, i.e. the vector designating the weights of all QoS attributes (cf. subsection 2.1). The BPEL designer may set the value of these variables after inspecting input parameters to the scenario (e.g. “choose=cheapest”), arranging thus for tailoring the QoS specification to the invoking user’s preferences.
2. the designer should include in each *invoke* construct in the WS-BPEL scenario the optional attribute *name* [1], assigning distinct names to the *invoke* constructs.
3. for the *invoke* construct having name *invX*, the designer should use the WS-BPEL variables *QoSmax_invX* and *QoSmin_invX*, which define the respective QoS specifications for the particular invocation. Similarly to the *QoS_weights* variable, the designer may set the values of the *QoSmax_invX* and *QoSmin_invX* variables after inspecting input parameters to the scenario.

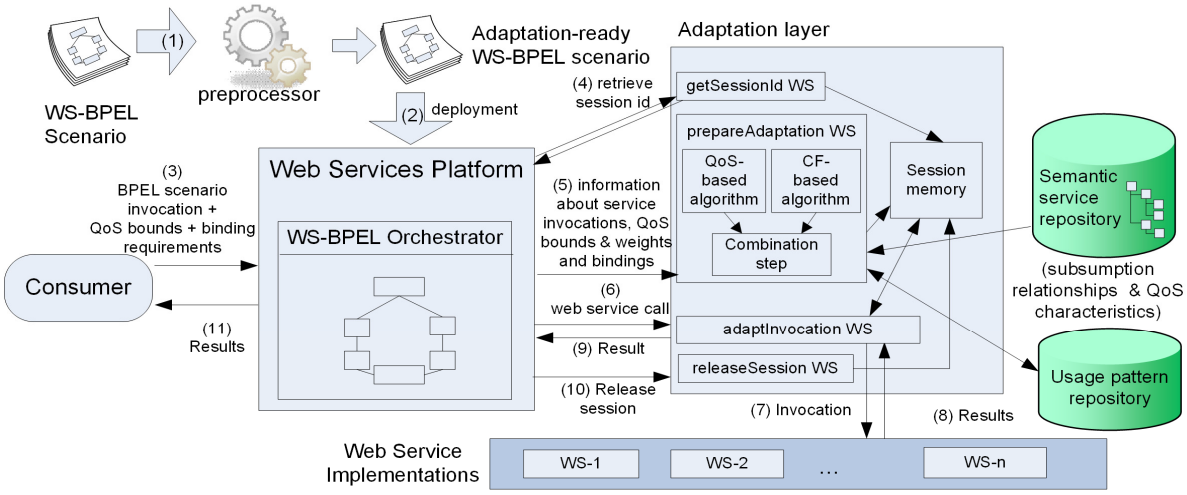

Figure 8. The Execution Adaptation Architecture

Note that the setting above allows the designer to set different QoS bounds for each distinct invocation. This provides more flexibility with the adaptation of the scenario, e.g. in the travel planning scenario considered in subsection 2.2, the bounds for the *event attendance* functionality may be set to high values (e.g. min=0.8, max=1), while the bounds for *ticket booking* and *hotel booking* may be set to low values, to indicate that the client wants good seats for the event to be attended, but economy travel tickets and an inexpensive hotel to limit the overall cost. On the other hand, the weights apply to the whole composition, rather than to individual services, since they reflect the client's perceived importance of each QoS attribute dimension on the process as a whole, and not its constituent parts [4].

Listing 3 presents an excerpt of a BPEL scenario setting QoS specifications for an invocation (named *invoke1*). Since variable *QoSmin_bookTicket* does not include a setting for *respTime* and *cost*, no lower bounds will be considered for these attribute values in the process of adapting the service invocation *bookTicket*. Similarly, since the *QoSmax_bookTicket* does not include a setting for the *reliability* QoS attribute, no upper bound for this attribute value will be considered in the process of adapting the service invocation *bookTicket*.

```
<!-- assign global weights -->
<assign>
   <copy>
      <from><literal>respTime: 0.3; cost:-0.5; reliability: 0.2</literal></from>
      <to variable="QoSweights"/>
   </copy>
</assign>
<!-- assign invocation-specific bounds for invocation “bookTicket”-->
<assign>
   <copy>
      <from><literal>respTime:0.5;cost:0.3</literal></from>
      <to variable="QoSmax_bookTicket"/>
   </copy>
   <copy>
      <from><literal> reliability: 0.6</literal></from>
      <to variable="QoSmin_bookTicket"/>
   </copy>
</assign>
<invoke name="bookTicket" partnerLink="lnk1" portType="port1" operation="op1" inputVariable="input1" outputVariable="output2"/>
```

Listing 3. QoS specification in the BPEL scenario

4.2 Preprocessing the WS-BPEL scenario

As stated above, the preprocessor accepts as input a WS-BPEL scenario and transforms it into an “adaptation-ready” form. More specifically, the transformed scenario differs from the original one into the following respects:

1. it includes, as its first operation an invocation to the web service *getSessionId* provided by the middleware; the result of the invocation is stored in a variable and used in subsequent operations.
2. it collects the information regarding (a) the QoS bounds for the functionalities, (b) the QoS attribute weights and (c) which functionalities should be bound to specific services, which will not be invoked and for which a recommendation is requested. This information is then transmitted, together with the result of the *getSessionId* invocation, to the adaptation layer, through an invocation to the *prepareAdaptation* WS.

   As stated in subsection 4.1, the QoS bounds for a functionality used in the WS-BPEL scenario are represented using input parameters *QoSmax_invName* and *QOSmin_invName*, where *invName* is the value of the *name* attribute in the <*invoke*> construct realizing the functionality (<*invoke name="invName" partnerLink="lnk1” ...>)*, while the QoS attribute weights are stored in variable *QoSweights*. Similarly, the designations regarding

functionality bindings are represented using variables *BINDING_invName*, which will again be set by the designer, probably after examining some input parameters. The value of a *BINDING_invName* variable may be one of (i) the id of the service to which the functionality should be bound, (ii) the literal *SKIP* if the functionality should not be invoked and (iii) the literal *RECOMMEND*, if a recommendation is requested for the specific functionality.

Regarding our example in section 2.2 (where a user wants to stay in Grand Resort and attend the NBA Finals, and requests a recommendation for ticket booking), we will assume that the names of the *invoke* constructs corresponding to the *ticket booking*, *hotel booking*, and *event attendance* functionalities are *bookTicket*, *bookHotel* and *attendEvent*, respectively. If the QoS attribute weights and QoS bounds for the *bookTicket* service were as shown in Listing 3, the information transmitted to the middleware through the *prepareAdaptation* WS would be as follows:

SessionId=<as returned by getSessionId>
QoSweights=respTime: 0.3; cost:-0.5; reliability: 0.2
QoSmax_bookTicket=respTime:0.5;cost:0.3
QoSmin_bookTicket=reliability:0.6
BINDING_bookTicket=RECOMMEND
BINDING_bookHOTEL=GrandResort
BINDING_attendEvent=NBAFinals

This information, together with the data contained in the semantic service repository and the usage patterns repository are adequate for the middleware layer to execute the algorithm described in section 3 and determine the execution plan to be used. The implementation arranges for executing the QoS-based and the CF-based parts in parallel, so that the optimization step can benefit from the presence of multiple processors and/or cores (a commodity in contemporary hardware).

The invocation to the *prepareAdaptation* WS is inserted before the first <*invoke*> construct of the original WS-BPEL scenario, to ascertain that the adaptation-related information have been transferred to the adaptation layer (and processed by it) before the first invocation is intercepted and adapted.

3. each service invocation is redirected to the *adaptInvocation* service, complemented with a header which includes the session id for the current WS-BPEL scenario execution (the value returned by the getSessionId WS) and the value of the name attribute of the particular *invoke* construct. Although header manipulation not a standard WS-BPEL feature, contemporary WS-BPEL orchestration engines provide means to set request headers, e.g. [47][48].
4. an invocation to the *releaseSession* service of the adaptation layer is included as a final operation in the transformed scenario.

The adaptation-ready WS-BPEL scenario, as produced by the preprocessor, is then deployed to the WS-BPEL orchestration engine and made available for execution.

4.3 Executing the WS-BPEL scenario

When a WS-BPEL scenario commences execution, its first action will be to invoke the *getSessionId* web service hosted in the adaptation layer, so as to retrieve a unique session identifier. Afterwards, it invokes the *prepareAdaptation* web service of the adaptation layer, transmitting to it (i) the session identifier (b) the information regarding the QoS bounds for each functionality, (c) the QoS weights and (d) the functionality binding and the recommendation request information. At this point, the adaptation layer has all the information needed to execute the algorithm described in section 03, so as to produce the execution plan to be followed in the current instance of the WS-BPEL scenario. After the algorithm has been applied, all service bindings (both those specified by the consumer and which were provided as input to the *prepareAdaptation* web service, as well as those produced as output of the adaptation algorithm) are stored into the *session memory*, tagged with the session identifier. Effectively, the consumer session memory stores, for each

session, the mappings between functionality invocations within the particular session and the concrete services that these invocations should be directed to. Assuming that in the example presented in subsection 4.2 the algorithm would determine that the *bookTicket* functionality should be bound to the *Swissair* service, the information that would be inserted into the session memory would be as shown in Listing 4.

```
(Session: <as returned by getSessionId>;
 Bindings: (  bookTicket: Swissair;
              bookHotel: GrandResort;
              attendEvent: NBAFinals) )
```

Listing 4. Information inserted in session memory

The *prepareAdaptation* web service finally stores the service bindings it received as input parameters into the usage patterns repository, making thus the usage information available for future recommendation formulations.

When the WS-BPEL orchestration engine executes an *invoke* construct, the invocation is directed to the *adaptInvocation* service of the adaptation layer, due to the transformations made by the preprocessor (cf. item 3 in subsection 4.2, above). Upon reception of an incoming request, the *adaptInvocation* service proceeds as follows:

1. it extracts from the request headers the session identifier and name of the *invoke* construct.
2. Using the session identifier, it retrieves from the session memory the service bindings pertinent to the particular session, and then it uses the name of the invoke construct to extract the binding of the specific functionality.
3. The request is then forwarded to the service indicated by the binding, the result is collected and finally it is returned to the WS-BPEL orchestration engine, as a reply to the original invocation.

Finally, when the WS-BPEL scenario reaches its end, it invokes the *releaseSession* web service, providing the session identifier as a parameter. The *releaseSession* service will then remove from the session memory all information pertaining to this session.

5 Experimental evaluation

In order to determine the optimal value for parameter *CFweight* and assess the performance of our approach and the quality of the adaptations it produces, we have conducted a set of experiments. The first experiment aimed to offer insight on the effect of the *CFweight* parameter on the formulation of the solution chosen by the algorithm and the solution's quality. The performance-related experiments aim to measure and quantify the overhead incurred due to the introduction of the middleware. On the other hand, the experiments assessing the quality of the adaptations aim to provide insight on how the QoS of the execution plan proposed by the algorithm in section 03 compares with the QoS of the execution plans proposed by the plain QoS-based algorithm (which is optimal) and the plain CF-based algorithm. In this experiment we have also included a "random" algorithm (i.e. randomly select a service fulfilling the QoS constraints specified by the user), in order to determine whether how the QoS of the execution plans formulated by the algorithm in section 3 compares with the average execution plan.

In more detail, in the performance-related experiments we have evaluated the overhead imposed due to the following activities executed in the proposed scheme: (a) execution plan formulation (invocation of the *prepareAdaptation* service) (b) invocation redirection (the extra network messages to and from the middleware during each service invocation and the querying of the session memory) and (c) housekeeping activities (i.e. invocations to *getSessionId* and *releaseSession* services, as well as update of the usage patterns repository). In these experiments we have varied the following parameters:

1. the number of concurrent invocations,
2. the size of the usage patterns repository,
3. the number of functionalities in the WS-BPEL scenario and
4. the number of recommendations requested per invocation.

The time needed by the preprocessor to transform the original WS-BPEL scenario into its "adaptation-ready" form has not been included in this evaluation, because the preprocessor operates in an offline fashion, not imposing thus any overhead to the performance of the production system.

In all experiments, the semantic service repository was populated with synthetic data having an overall size of 2.000 web services[5]; these services account for 20 different functionalities, with each functionality having 100 alternative providers. The QoS attribute values in this repository were uniformly drawn from the domain [0, 1]. When conducting a test for a particular number of functionalities, we synthetically generated 20 BPEL scenarios, randomly drawing implementations of distinct functionalities from the repository, and the performance evaluation tests were run for each of the generated scenarios. In the scenario generation process, two consecutive functionality invocations were selected to be executed sequentially (<*sequence*> construct) with a probability of 0.8 and in parallel (<*flow*> construct), with a probability of 0.2. Therefore, in all cases, there existed BPEL scenarios involving parallel invocations.

[5] Since in our experiment we used a single machine to host the target web services, and deploying 2,000 target web services on a single machine would degrade its performance, for the experiment purposes it was arranged that within the semantic service repository all services delivering *exactly the same* functionality were mapped to the same implementation on the machine hosting the services. We used DNS aliases to make the service endpoints different at the repository level (e.g. a service implementation *svc1* deployed on the second machine would appear in 100 entries in repository as http://exp1.sdbs.uop.gr/svc1/endpoint, …, http://exp100.sdbs.uop.gr/svc1/endpoint, with all addresses exp1.sdbs.uop.gr, …, exp100.sdbs.uop.gr resolving to the same IP address). This arrangement reduced the number deployed services to 20, and does not affect the adaptation algorithm operation, since endpoints are only considered in the execution phase, after the adaptation is performed. Note also that in a real-world setting, these services would be deployed on different machines (the alternative providers' machines).

The differences observed, regarding the execution time, when the performance evaluation tests were conducted on different BPEL scenarios involving the same number of functionalities were negligible (less than 2% of the overall time). Some differences were observed in the quality of the proposed solution, which depend on the functionalities that were actually included in the services and the contents of the usage patterns repository; this is to be expected, since the score computed by the collaborative filtering algorithm . Each unique performance evaluation test was run 100 times, and the average value was computed and is shown in the following diagrams; error bars are used in the diagrams to show the minimum and maximum values. The lower QoS bounds for the functionalities were randomly drawn from the domain [0,0.4], while the upper QoS bounds for the functionalities were randomly drawn from the domain [0.6,1]. The weights of the QoS attributes were again randomly selected from the domain [0,1]. In all cases, a uniform distribution was used.

For our experiments we used two machines: the first one (a workstation equipped with one Intel Xeon E5-2620@2.0GHz / 6 cores CPU and 16 GB of RAM) hosted the preprocessor and the clients, while the second one (a workstation with identical configuration to the first, except for the memory which was 64GBytes) hosted the BPEL orchestration engine (a Glassfish application server [49] using the Metro web service stack [50]), the adaptation layer, the target web services, the service repository (which included the tree representation of the subsumption relations and the services' QoS characteristics) and the usage patterns repository. The machines were connected through a 100Mbps local area network. Both repositories (the semantic service repository and the usage patterns repository) were implemented as in-memory hash-based structures, which proved more efficient that using a separate, in-memory database engine (e.g. HSQLDB [51] used in [53] and [54]). The new rows of the usage patterns repository are written periodically to the disk to ensure persistence, while provisions have been made to allow for reloading the semantic service repository in case it changes (e.g. addition or deletion of services, or changes in the QoS values).

5.1 Determination of CFweight

In this experiment, we randomly generated 1,000 BPEL scenarios, and for each scenario we ran the adaptation algorithm varying the values of the CFweight parameter from 0% to 60%. In the scenario generation process, two consecutive functionality invocations were selected to be executed sequentially (<*sequence*> construct) with a probability of 0.8 and in parallel (<*flow*> construct), with a probability of 0.2. For each value of the CFweight parameter, we collected the following metrics:

1. *percentage of solutions affected by the CF-based dimension*: this metric is computed as the ratio of solutions where the final solution chosen was different than the one proposed by the QoS-based algorithm to the number of examined cases, and it is a measure of the effect that the CF-based component has on the formulation of the final solution. Since the introduction of the CF-based component aimed to allow for taking into account the users' subjective ratings (quality of experience), we would like to increase this effect.
2. *Normalized quality of the solution:* this metric is computed as the ratio of QoS-based overall utility value (OUV_{QoS}) of the chosen solution to the maximum QoS-based utility value computed by the QoS-based algorithm. Effectively, this metric represents how close the finally chosen solution is to the optimal QoS. Naturally, we would like this metric to be as high as possible.

Figure 9 presents the findings of our experiment. For values of *CFweight* ≤ 20%, the normalized quality of the solution is very close to the optimal (97.3% for *CFweight*=20%), however only up to 18.7% of the solutions are affected by the CF-based dimension (or, equivalently, the proposal of the QoS-based algorithm is adopted in the 81.3% of the adaptations); this indicates that when the value of *CFweight* falls in this range, the result of the CF-based algorithm is not adequately taken into account.

For values of *CFweight* ≥ 50%, the majority of decisions is affected by the QoS-based dimension (53.1% for *CFweight*=50% and 60. 3% for *CFweight*=60%) however the normalized quality of the solution drops to 76.20% for *CFweight*=50% and to 67.47% for *CFweight*=60%.

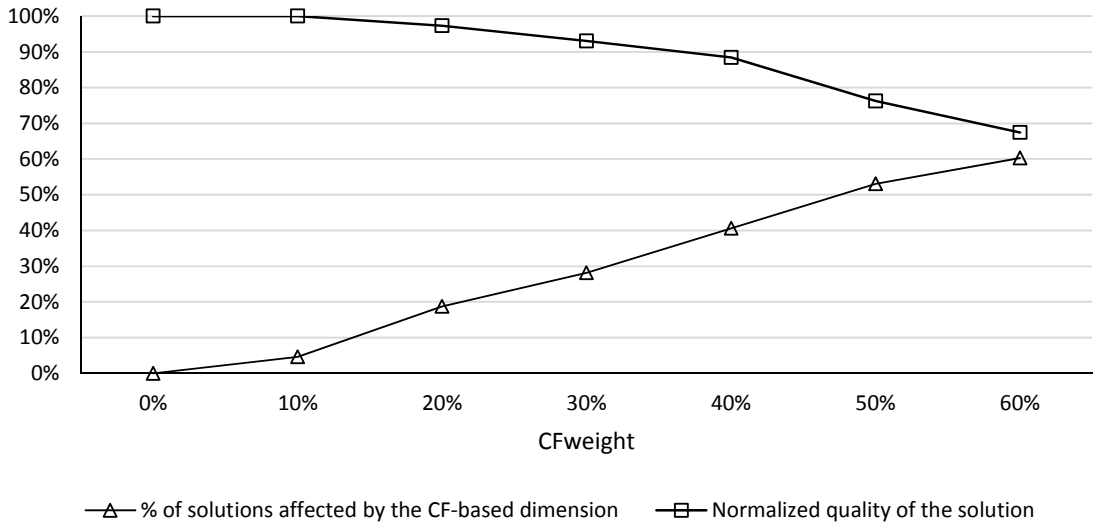

Figure 9. The effect of the *CFweight* parameter on the on the formulation of the solution and the solution's quality

Setting *CFweight* to one of the values (30%, 40%) appears to provide a good balance between the goals of maximizing the effect of the CF-based dimension and maintaining a high normalized quality of the solution. If a value of 30% is used, 28.1% of the solutions are affected by the CF-based dimension and the normalized quality of the solution is 93.07, while the respective numbers for *CFweight*=40% are 40.60% and 88.43% respectively. Considering that a value of 88.43% for the normalized quality is acceptable, we have set *CFweight* to 40%, in order to maximize the effect of the CF-based dimension.

5.2 Execution time

Figure 10 presents the execution plan formulation and housekeeping overhead, i.e. the overhead imposed by the use of the invocations to the *getSessionId, prepareAdaptation* and *releaseSession* web services, for a varying degree of

concurrent WS-BPEL scenario invocation requests arriving to the WS-BPEL orchestration engine. In this experiment, the usage patterns repository was set to include 1,000 qualifying entries (i.e. the usage patterns repository contained 1,000 entries matching the functionality for which a recommendation was requested), the number of functionalities in the WS-BPEL scenario was set to 10 and one recommendation was requested (i.e. 9 service bindings were fixed by the consumer). We can notice that the overhead is increasing linearly with the number of concurrent invocations, while we can also observe that even with 250 concurrent invocations the overhead is less than one second.

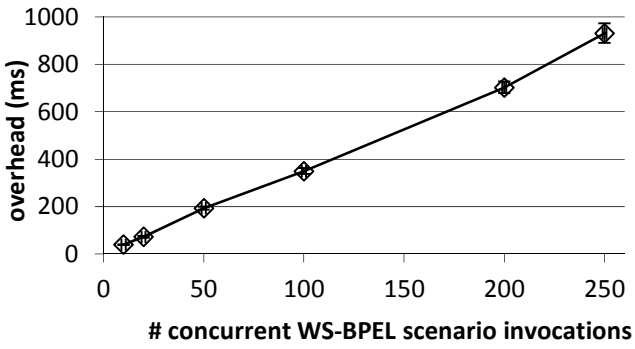

Figure 10. Execution plan formulation and housekeeping overhead for varying degrees of concurrency

Figure 11 illustrates the time needed to perform the computation of the execution plan and the housekeeping tasks under a varying number of functionalities within the WS-BPEL scenario and for different counts of qualifying records in the usage patterns repository (250, 500 and 1000 records). In all cases, the concurrency level was set to one (a single request was submitted and processed) and one recommendation was requested. The recorded times were found to increase between 8% and 14% when the number of functionalities increases by one; this is owing to the time needed for the extra processing to compute the similarity score for the extra service, since the integer programming problems formulated in both cases are identical and therefore their solution time is not affected. The time to compute the similarity score has been optimized, by keeping the similarity metrics between all service/category pairs pre-computed in a hash

table and looking them up as needed, instead of computing them on demand. The hash table is populated when the middleware bootstraps and when the service repository is reloaded.

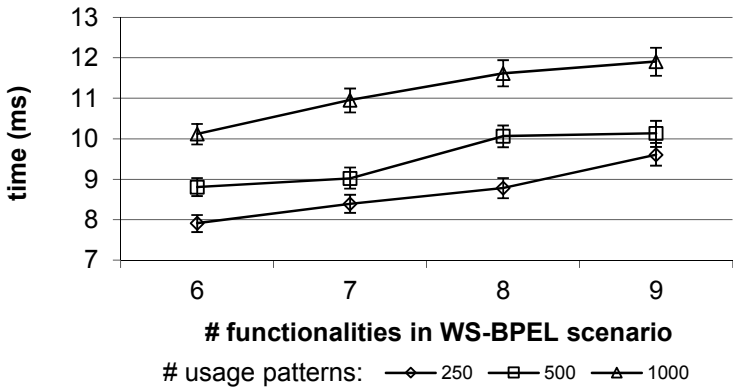

Figure 11. Execution plan formulation overhead for varying number of functionalities within the WS-BPEL scenario and qualifying usage patterns

In Figure 11 we can also notice that the execution plan formulation overhead is increasing at a slower with respect to the number of the qualifying usage patterns increases (on average, an increase of 45.68% when the number of qualifying usage patterns increases by 4). This indicates that the algorithm scales well with the size of the usage patterns repository, however for large usage patterns repository sizes, techniques reducing the execution time, such as the one proposed in [70] can be employed.

Figure 12 illustrates the time needed to formulate the execution plan, with respect to the number of qualifying records in the usage patterns repository and the number of recommendations requested in a single WS-BPEL scenario invocation. In all cases, the concurrency level was set to one (a single request was submitted and processed) and the WS-BPEL scenario whose execution (and adaptation) was requested contained 10 functionalities. We can observe that the time needed for each recommendation is fairly stable, e.g. the time needed for making three recommendations is approximately triple the time needed for making one recommendation when the size of the usage patterns repository

remains stable. This is due to the fact that each recommendation in the CF-based is made individually by repeating the same steps of the algorithm, and this extra time is merely added to the overall algorithm execution time. The behavior regarding the number of qualifying usage patterns is analogous to the one observed in Figure 11. In summary, the recommendation overhead increases linearly with respect to the number of recommendations requested, hence the proposed approach is scalable.

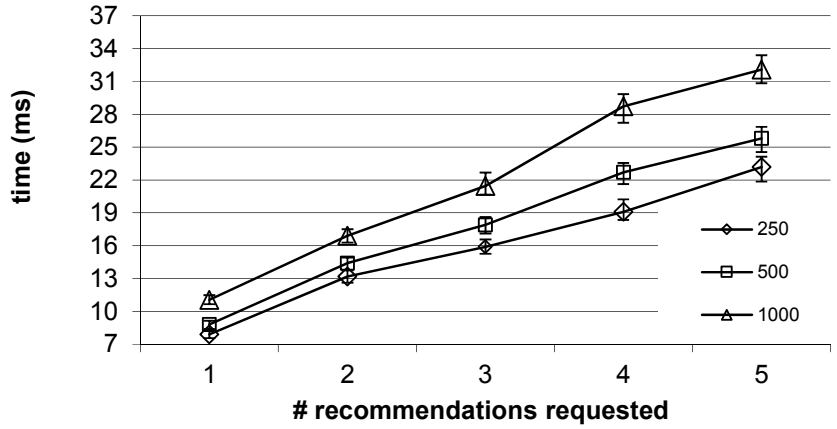

Figure 12. Recommendation overhead for varying number of qualifying usage patterns and number of requested recommendations

Figure 13 evaluates the *overall overhead* incurred for the execution of a BPEL scenario due to the introduction of the adaptation mechanism proposed in this paper, and compares this overhead to the overhead introduced by the algorithm described in [5] (without the XSLT transformation feature) which is only based on the QoS features; the latter algorithm was deployed and run in the same test environment. The overall overhead accounts for (a) the time to compute the service that each invocation should be redirected to (b) housekeeping activities such as session establishment and release (c) transmission and processing time of the two extra messages per service invocation, which are introduced due to the presence of the middleware. The overall overhead is the only metric that can be used for comparing the performance of the two algorithms, since the algorithm in [5] operates in a greedy fashion, selecting the

service to be invoked at the time the request to execute some functionality is made, while the algorithm presented in this paper computes the overall execution plan *before* any actual functionality invocations are made. [5] was chosen for among the works listed in the related work section (section 6) to perform this comparison, because it is one of the few works that describe not only the adaptation algorithm but the execution architecture as well.

In this comparison, a scenario with 10 functionalities was used, and the number of adaptations performed was varied from 1 to 5. Note that the algorithm in [5] adapts by default all functionalities; for the cases however that service selection affinity should be maintained among a group of services, adaptation is only performed on first invoked functionality and the subsequent ones are bound to the provider selected during the first adaptation, due to the greedy nature of the algorithm. This mode of operation was exploited to set the number of adaptations performed within the BPEL scenario execution: using a BPEL scenario necessitating the maintenance of service selection affinity across all 10 functionalities (by having all invocations pointing to endpoints of a single service provider), leads to the execution of a single adaptation (upon the first invocation); using a BPEL scenario necessitating the maintenance of service selection affinity across 9 services (by having 9 services pointing to endpoints of a single service provider and one service pointing to a different service provider) leads to the execution of two adaptations (one upon the first invocation to a member of the group of the 9 services and one upon the invocation of the tenth service, which is adapted in isolation); and so forth. Again, 20 randomly generated BPEL scenarios were used, and experiments for each scenario were run 100 times.

In Figure 13 we can observe that the overall overhead incurred when the proposed adaptation scheme is used (lines *p(250)* and *p(1000)* in the diagram, where p(X) represents the execution of the proposed adaptation scheme with the usage patterns repository containing X matching records per service adaptation) is 26.1%-35.4% higher than the one incurred when the QoS-based scheme proposed in [5] (line *q* in the diagram) is employed. In absolute numbers, this accounts for an increase ranging from 36.2 to 47.66 msec, which is acceptable, considering the fact that the proposed algorithm introduces new functionality (the CF-based dimension). Note that the algorithm in [5] follows a greedy

strategy, therefore it exhibits good performance, running however the risk that the adaptations it computes can be suboptimal in terms of the QoS dimension. On the contrary, both the QoS-based and the CF-based parts of the algorithm proposed in this paper use integer programming strategies to examine all adaptations combinatorially, hence this risk is avoided.

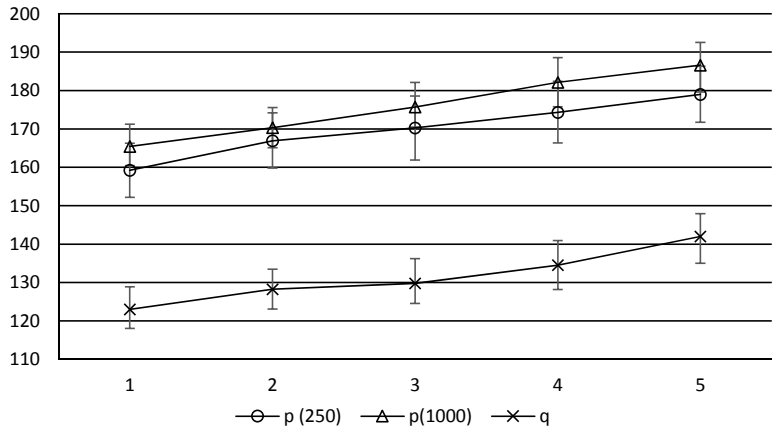

Figure 13. Performance comparison of the proposed algorithm against the QoS-based algorithm in [5]

Finally, since the computation of the execution plan involves the solution of integer programming problems, and the complexity of solving integer programming problems is varying depending on the structural properties of the problem [71], we have conducted an experiment to verify that the optimization times observed are not coincidental (i.e. owing to the fact that the tested scenarios have favorable properties), but correspond to the standard behavior of the algorithm. To this end, we have synthetically generated 5,000 BPEL scenarios, with 10 functionalities, and measured the overall overhead incurred when the proposed adaptation scheme is used, again considering cases when the number of matching lines from the usage patterns repository are (a) 250 and (b) 1000, and when the number of requested adaptations ranges from 1 to 5, similarly to the case of Figure 13. In the scenario generation process, two consecutive functionality invocations were selected to be executed sequentially (<*sequence*> construct) with a probability of 0.8

and in parallel (<*flow*> construct), with a probability of 0.2. The results of this experiment are depicted in Figure 14, and their similarity with the results of Figure 13 indicates that the algorithm performs fairly stable, regardless of the specific scenario that it makes recommendations on.

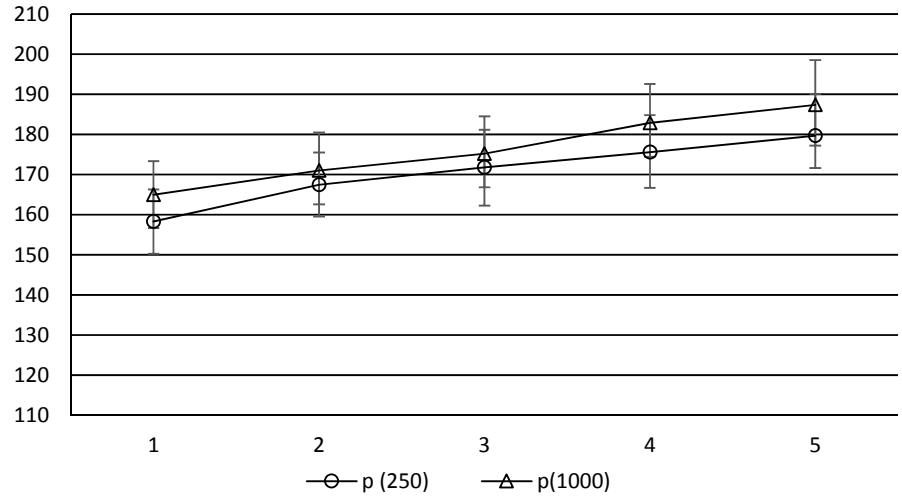

Figure 14. Validating the stability of the proposed algorithm's performance

5.3 Execution plan QoS

Figure 15 depicts the QoS of the execution plan formulated by different algorithms for a number of trial cases, aiming to provide insight on how the QoS of the execution plan proposed by the algorithm in section 03 compares with the QoS of the execution plans proposed by the plain QoS-based algorithm described in [53] and the plain CF-based algorithm described in [54]; as stated above, a "random" algorithm is also included in order to determine whether how the QoS of the execution plans formulated by the algorithm in section 3 compares with the average execution plan.

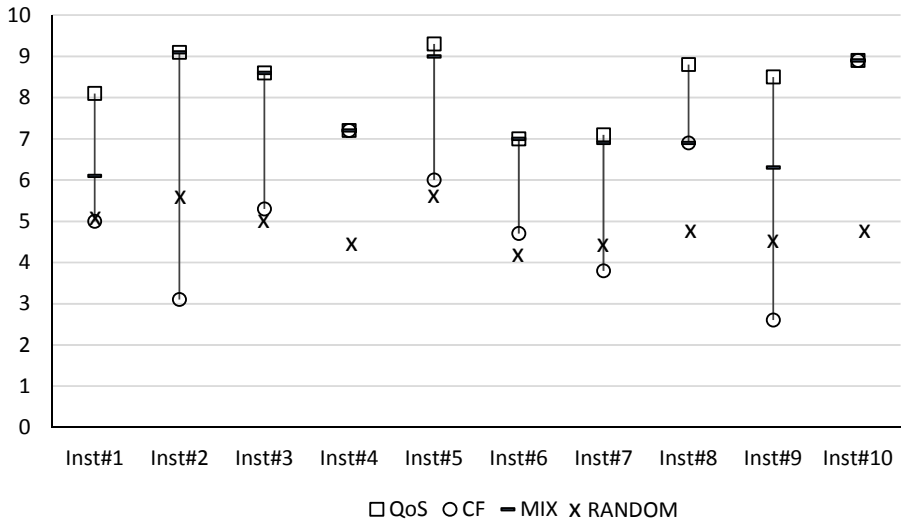

Figure 15. QoS of the execution plans proposed by different algorithms for ten trial cases

The trial cases in this diagram correspond to the invocation of a WS-BPEL scenario containing 10 functionalities in total and requesting 1 recommendation (the remaining 9 functionalities were bound to specific services). The lower QoS bounds for the functionalities were randomly drawn from the domain [0,0.4], while the upper QoS bounds for the functionalities were randomly drawn from the domain [0.6,1]. The weights of the QoS attributes were again randomly selected from the domain [0,1]. In all cases, a uniform distribution was used.

Note that since the QoS-based and the CF-based algorithms, as described in subsections 3.1 and 3.2 respectively, formulate *lists of execution plans*, the QoS value shown in the diagram for these algorithms is the one corresponding to the execution plan that has attained the biggest score, i.e. the execution plan that would be selected by each algorithm, if it were to decide on its own for the adaptation. Note also that since the *random* algorithm is non-deterministic, each test case for this algorithm was run 100 times and the mean execution plan QoS was used in the diagram.

In the diagram we can notice that the combined algorithm chooses execution plans whose QoS are very close to the optimal ones: the ratio (combinedQos / optimalQoS) varies between 0.75 and 1, with an average of 0.92. This is considerably higher that the corresponding ratio achieved by the CF-based algorithm (min: 0.34, max: 1, average: 0.65; this is in-line with the results presented in [55]) and the ratio attained by the random algorithm (min: 0.62, max: 0.75, average: 0.65). In all cases, the combined algorithm achieves a QoS equal or higher than pure CF-based one, indicating its ability to tailor the execution considering the proposals of the CF-based algorithm and at the same time maintain a high QoS.

Of particular interest in Figure 15 are trials 2, 5 and 6, in which the deviation between the QoS computed by the CF-based and the QoS-based algorithms is high, however the QoS of the composition is equal to the one proposed by the QoS-based algorithm: these cases correspond to occasions that the usage patterns repository sparsity is high, and therefore the confidence to the proposal of the CF-based algorithm is low, leading the combination step to practically disregard the CF-based algorithm's proposals. In trial 3 sparsity was low, however the execution plan proposed by the QoS-based algorithm was finally adopted: this is because the same execution plan was ranked third by the CF-based algorithm, and hence it achieved the highest WCombMNZ score in the combination step. Conversely in trial 8 the proposal of the CF-based algorithm was adopted, being the fourth runner up in the QoS-based algorithm list and achieving the highest WCombMNZ score in the combination step. In trial 4, both algorithms ranked first the same execution plan, and hence it was selected. In all other trials, an execution plan that was high in the list of both algorithms, but not first in either list, was selected.

6 Related work

As stated in section 1, adaptation approaches presented insofar follow either the horizontal and vertical level adaptation [2]. AgFlow, introduced in [4] provides WS-BPEL clients with QoS constrains, by revising the execution

plan (i.e. the set of bindings of functionalities referenced in the WS-BPEL scenario to concrete services [4]) either on a global or a local planning. In case of local planning, web service selection is done during execution time and conforms to a greedy strategy. On the other hand, in case of global planning, WS-BPEL scenario is partitioned in regions of collaboration and web service selection strategy is concentrated on a group of actions. Work in [11], introduces VieDAME, which adapts the execution of BPEL scenarios according to QoS parameters; however these parameters and the selection strategy are pre-determined through pluggable modules. Additionally, VieDAME does not support service selection affinity and is platform-dependent since it relies on extensions of the ActiveBPEL engine.

Work in [12] considers service selection in the presence of QoS constraints and aiming to minimize an objective function for the entire orchestration employing both brute force (OPTIM_S) and heuristic (OPTIM_HWEIGHT) algorithms. This work presents the service selection algorithms but does not propose an architecture on top of which the adaptation can be realized, while it additionally does not consider service selection affinity. [13] introduces the MOSES approach, which performs web service selection by means of formulating and solving an integer programming problem, considering different patterns [par_or (i.e. concurrent execution of atomic services, with the construct successfully concluding when one service execution finishes successfully), par_and (i.e. concurrent execution of atomic services, with the construct successfully concluding when all services finish their execution successfully) etc.], and monitoring QoS execution during runtime. MOSES assumes that business processes are written as abstract compositions (contrary to our approach where business processes are specified through actual WS-BPEL scenarios, enabling the use of existing ones without any modification), while QoS requirements are stated through an SLA, giving the *average value* of QoS attributes, not allowing distinct QoS specifications per web service invocation. [72] presents a technique based on mixed-integer programming for QoS-based web service matchmaking, and argues that mixed-integer programming (MIP) should be used as a matchmaking technique instead of constraint programming, since it proves to be more efficient.

Several works exist which not only optimize business processes, but also implement exception handling mechanisms in order to provide solutions in unpredicted run time environments. More specifically, the framework proposed in [14] uses autonomic computing concepts for providing execution plan formulation for business processes, taking into account QoS parameters, monitors dynamically QoS violations at runtime and provides instrumentation for the handling of these exceptions. The work in [5] combines adaptation to QoS specifications and exception resolution (reverting to wherever possible to suboptimal plans, in the presence of exceptions), undertaking however a greedy strategy to service binding, which may lead to suboptimal solutions. [53] arranges for performing horizontal adaptation based on QoS criteria, taking into account the sequential and parallel execution structures within the BPEL scenario, while catering for preservation of service selection affinity. Work in [16] proposes a multi-tier architecture, TailorBPEL, which supports the tailoring of personalized BPEL-based workflow compositions, enabling end-users to tailor personalized BPEL-based workflow compositions at runtime.

Note that none of the approaches listed above incorporates collaborative filtering techniques to enhance the quality of the adaptation. Interestingly, [15] uses techniques from the collaborative filtering domain for generating recommendations in the context of BPEL scenario adaptation, however the work reported therein is based on QoS data contributed by users, as opposed to our work which is based on previous personalizations of BPEL scenario executions. [54] employs collaborative filtering techniques to drive the adaptation, and presents an associated execution framework. This work however uses very limited QoS-based criteria (only a lower and an upper bound optionally specified for each QoS attribute), hence it runs the risk of formulating solutions whose overall QoS is much inferior to the optimal composition QoS that can be attained, especially in the cases that collaborative filtering has known issues (e.g. cold start and gray sheep).

In order to perform adaptation, exception resolution, or collaborative filtering-based adaptation, all approaches employ some means to formally specify the services' functionality; techniques involving QoS characteristics need additionally to have access the services' QoS attribute values. Regarding services' functionality, some approaches need

only record which services are equivalent [17], while others use more elaborate schemes, such as the one described in [18][19], according to which a service A may be related a service B through one of the following subsumption relations: exact (services have identical functionality, e.g. they both book a flight), plugin (service A is more specific than B and can thus be used in its place; for instance, if service A is "book a flight" and service B is "book transport", then we can use A in the place of B since A actually books a transport), subsume (service A is more general than B and therefore cannot always be used in its place; e.g. if A books a transport and B books a flight, we cannot always use A in the place of B since A may lead to booking a trip by ship instead of a trip by plane) and fail (none of the exact, plugin and subsume relations holds). [19] details on how a service matchmaker can operate over an OWL-S ontology to computes the degree of semantic matching for a given pair of service advertisement and request, using subsumptions.

METEOR-S [20] can be used for storing and querying both service functionalities and QoS characteristics. Since METEOR-S adopts ontologies for representing information about services, it is powerful enough to express both the service equivalence notions and the subsumption relations; WSMO [21] is another widely adopted option with similar expressive power, regarding the features listed above. [33] describes the OPUCE platform repository, which could be used to provide both the necessary QoS data and the subsumption relations. In the OPUCE platform repository, service functionalities are described by means of an ontology concept, which can be subclassed to provide the taxonomy illustrated in Figure 1.

In the collaborative filtering domain, several methods have been proposed, however their incorporation in the WS-BPEL execution adaptation process has only received little attention. [22] surveys collaborative filtering, focusing on is use within the adaptive web. Interestingly, [22] lists the basic properties of domains suitable for collaborative filtering classified under three major categories (data distribution; underlying meaning; data persistence) and all the listed properties hold for the context of WS-BPEL scenario adaptation. [23] presents an evaluation of collaborative filtering algorithms. Collaborative filtering is in many cases combined with another personalization technique, namely content based filtering; [24] and [25] are examples of such approaches. However, content-based filtering needs items with

content to analyze [22], and in the context of WS-BPEL scenario adaptation we cannot use the items' content (responses of individual web services) since the responses of equivalent web services are identical (apart from invocation-specific data), hence they are not useful for choosing between different service implementations. Moreover, the responses contain personal data (and some of them sensitive data, e.g. credit card numbers or health-related data), and therefore they cannot be retained to be used in further analysis. Thus, in this work we use only the collaborative filtering approach, retaining only service usage patterns, in an anonymized form. Finally, [26] examines the use of collaborative filtering techniques for suggesting web services to the users. The work in [26] however considers individual web services, not WS-BPEL scenarios (being however able to suggest compositions of services), and suggestions are forwarded to the user, and not integrated into the scenario execution adaptation process. As stated above, the work in [54] uses collaborative filtering techniques for adapting the execution of WS-BPEL scenarios, uses only limited QoS criteria, running thus the risk of formulating suboptimal solutions.

7 CONCLUSION AND FUTURE WORK

In this paper we have presented a framework for adapting the execution of WS-BPEL scenarios, taking into account both the QoS requirements of the invoking users and the behavior of other users having invoked the same scenario in the past. The algorithm used for performing the adaptation follows the metasearch algorithm paradigm, using two different candidate adaptation ranking algorithms, the first examining the QoS aspects only and the second being based on collaborative filtering techniques. The adaptation rankings produced by these two algorithms are combined to generate the overall ranking, which then drives the adaptation. The proposed framework is complemented with an execution architecture for enacting the adaptation. The execution architecture includes a scenario preprocessing step, which transforms existing WS-BPEL scenarios into an "adaptation ready" form, and an adaptation layer (a middleware component intervening between the WS-BPEL execution engine and the actual service implementation), which undertakes the tasks of executing the adaptation algorithm and redirecting invocations to the selected services.

The proposed framework has been experimentally evaluated, both in terms of performance and in terms of the QoS of the adaptation it produces. The overhead imposed for performing the adaptation has been found to be small (approximately 5msec per concurrent user), while the execution architecture's scalability has also proven to be satisfactory, exhibiting a linear increase for all tested factors (concurrent users, size of usage patterns repository, number of recommendations requested per invocation and number of functionalities within the WS-BPEL scenario). The proposed algorithm has also been found to choose execution plans whose QoS are very close to the optimal ones, while at the same time crafting the execution plans taking into account the proposals of the CF-based algorithm.

Our future work will focus on the integration of QoS-adherence monitoring mechanisms, such as those described in [11], as well as gathering and utilizing statistical information from prior scenario executions as information sources to the adaptation process. This statistical information will be used to quantify aspects regarding the execution of control constructs in the scenario, e.g. the probability that a conditional branch is executed or the distribution of the number of executions of a loop. These quantities will be then taken into account by the adaptation algorithm. QoS-adherence monitoring mechanisms could also provide feedback to the adaptation algorithm, by either changing the services' QoS parameters in the repository (e.g. increasing the value of a services' response time QoS attribute if it consistently exhibits worse response time than the one recorded in the repository) or by instructing the algorithm to refrain temporarily from choosing a service (e.g. if the service has been recently invoked many times and its degraded performance could thus be attributed to a temporary overload condition). We also plan to examine how the algorithm can be extended to consider different adaptation strategies [69], and to evaluate the performance of these strategies. Finally, a user survey is planned to assess the degree to which users are satisfied by the plans generated by the various adaptation algorithms; work in this direction will consider recent results in the area of exploiting social networks for software evaluations, as described in [52].

REFERENCES

[1] OASIS WSBPEL TC. WS-BPEL 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[2] MP. Papazoglou, P. Traverso, F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges", IEEE Computer vol. 40, no 11, 2007, pp. 38-45.

[3] V. Cardellini, V. Di Valerio, V. Grassi, S. Iannucci, F. Lo Presti, "A Performance Comparison of QoS-Driven Service Selection Approaches", Proceedings of ServiceWave 2011, Abramowicz W et al. (Eds.): LNCS 6994, 2011, pp. 167–178.

[4] LB. Zeng, AHN. Benatallah, M. Dumas, J. Kalagnanam, H. Chang, "QoS-aware middleware for web services composition". IEEE Transactions on Software Engineering, vol. 30, no 5, 2004.

[5] C. Kareliotis, C. Vassilakis, S. Rouvas, P. Georgiadis. "QoS-Driven Adaptation of BPEL Scenario Execution", Proceedings of ICWS 2009, E. Damiani, R. Chang, J. Zhang (eds), 2009, pp. 271-278.

[6] M. Claypool, A. Gokhale, Y. Miranda, P. Murnikov, D. Netes, M. Sartin, "Combining Content-Based and Collaborative Filters in an Online Newspaper". SIGIR '99 Workshop on Recommender Systems: Algorithms and Evaluation, I. Soboroff, C. Nicholas, M. Pazzani (eds), Berkeley, California, 1999,

[7] CL. Hwang, K. Yoon, "Multiple Criteria Decision Making", Lecture Notes in Economics and Mathematical Systems, Springer-Verlag, 1981.

[8] J. Basilico, T. Hofmann, "Unifying collaborative and content-based filtering". Proceedings of the twenty-first international conference on Machine learning (ICML 04), C.E. Brodley (ed), 2004, pp. 9-16.

[9] B.M. Kim, Q. Li, C.S. Park, S.G. Kim, J.Y. Kim. "A new approach for combining content-based and collaborative filters". Journal of Intelligent Information Systems, July 2006, vol. 27, Issue 1, pp 79-91.

[10] J. Fürnkranz, Eyke Hüllermeier (eds), "Preference Learning". Springer; 2011 edition (October 13, 2010), ISBN: 3642141242

[11] O. Moser, F. Rosenberg, S. Dustdar, "Non-Intrusive Monitoring and Service Adaptation for WS-BPEL", Proceedings of WWW 2008, J. Huai, R. Chen, H-W. Hon, Y. Liu, W-Y. Ma, A. Tomkins, X. Zhang (eds), Beijing, China, 2008, pp. 815-824.

[12] Y. Xia, P. Chen, L. Bao, M. Wang, J. Yang, "A QoS-Aware Web Service Selection Algorithm Based on Clustering", Proceedings of ICWS11, I. Foster, L. Moser, J. Zhang (eds), 2011.

[13] V. Cardellini, S. Iannucci, "Designing a Broker for QoS-driven Runtime Adaptation of SOA Applications", Proceedings of ICWS10, Florida, USA, 2010, pp. 504–511.

[14] AE. Arpacı, AB. Bener, "Agent Based Dynamic Execution of BPEL documents", Proceedings of ISCIS 2005, LNCS 3733, P. Yolum, T. Güngör, F.S. Gürgen, C.C. Özturan (eds), 2005, pp. 332 – 341.

[15] Z. Zheng, H. Ma, M. Lyu, I. King, “QoS-Aware Web Service Recommendation by Collaborative Filtering”, IEEE Transactions on Services Computing vol. 4 no 2, 2011, pp. 140-152.

[16] G. Canfora, M. Di Penta, R. Esposito, ML. Villani, “An Approach for QoS-aware Service Composition based on Genetic Algorithms”, Proceedings of the 2005 conference on Genetic and evolutionary computation, H-G. Beyer, U-M. O'Reilly (eds), 2005, pp. 1069-1075.

[17] S. Rinderle-Ma, M. Reichert, M. Jurisch, “Equivalence of Web Services in Process-Aware Service Compositions”, Proceedings of ICWS'09, E. Damiani, R. Chang, J. Zhang (eds), 2009, pp. 501 – 508.

[18] M. Paolucci, T. Kawamura, T. Payne, T. Sycara, “Semantic Matching of Web Services Capabilities”, Proceedings of the International Semantic Web Conference, I. Horrocks, J.A. Hendler (eds): Sardinia, 2002, pp. 333-347.

[19] M. Klusch, B. Fries, K. Sycara, “Automated Semantic Web Service Discovery with OWLS-MX”, Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), H. Nakashima, M.P. Wellman, G. Weiss, P.Stone (eds) May 8-12, 2006, Hakodate, Hokkaido, Japan.

[20] J. O’Sullivan, D. Edmond, A. Ter Hofstede, “What is a Service?: Towards Accurate Description of Non-Functional Properties”, Distributed and Parallel Databases, vol. 12, 2002.

[21] J. Cardoso, A. Sheth, “Semantic e-Workflow Composition”, Journal of Intelligent Information Systems, vol. 21 no 3, pp. 191-225, 2003.

[22] JB. Schafer, D. Frankowski, J. Herlocker, S. Sen, “Collaborative Filtering Recommender Systems”, in “The Adaptive Web”, Lecture Notes in Computer Science Volume 4321, P. Brusilovsky, A. Kobsa, W. Nejdl (eds), 2007, pp 291-324.

[23] JL. Herlocker, JA. Konstan, LG. Terveen, JT. Riedl, “Evaluating collaborative filtering recommender systems”, ACM Transactions on Information Systems vol. 22, no 1, January 2004, pp. 5-53.

[24] M. Balabanovic, Y. Shoham. “Fab: content-based, collaborative recommendation”, Communications of the ACM, vol. 40, issue 3, 1997, pp 66-72.

[25] MJ. Pazzani, “A Framework for Collaborative, Content-Based and Demographic Filtering”, Artificial Intelligence Review, vol. 13, issue 5-6, December 1999, pp 393-408.

[26] US. Manikrao, TV. Prabhakar, “Dynamic Selection of Web Services with Recommendation System” Proceedings of the International Conference on Next Generation Web Services Practices, A. Abraham, S.Y. Han, D. Hung-Chang Du, M. Paprzycki (eds), 2005, pp. 117-121.

[27] ISO. “UNI EN ISO 8402 (Part of the ISO 9000 2002): Quality Vocabulary”, 2002.

[28] ITU. “Recommendation E.800 Quality of service and dependability vocabulary”, 1998.

[29] J. Cardoso, “Quality of Service and Semantic Composition of Workflows”, PhD thesis, Univ. of Georgia, 2002.

[30] T. Chothia, J. Kleijn, “Q-Automata: Modelling the Resource Usage of Concurrent Components”, Electronic Notes in Theoretical Computer Science 175, 2007, pp. 153–167.

[31] L. Barbosa, S. Meng, “QoS-aware Component Composition”, Proceedings of CISIS 2010, L. Barolli, F. Xhafa, S. Vitabile, H-H. Hsu (eds), 2010, pp. 1008 – 1013.

[32] S. Ran, “A Model for Web Services Discovery With QoS”, ACM SIGecom Exchanges, vol. 4(1), Spring, 2003, pp. 1-10.

[33] J. Yu, Q. Sheng, J. Han, Y. Wu, C. Liu, “A semantically enhanced service repository for user-centric service discovery and management”, Data & Knowledge Engineering, vol. 72, Feb 2012, pp. 202-218.

[34] E. Al-Masri, Q.H. Mahmoud, “Discovering the best web service”, (poster) 16th International Conference on World Wide Web (WWW), P. Bouquet, H. Stoermer, G. Tummarello, H. Halpin (Eds.) 2007, pp. 1257-1258.

[35] E. Al-Masri, Q.H. Mahmoud, "QoS-based Discovery and Ranking of Web Services", IEEE 16th International Conference on Computer Communications and Networks (ICCCN), 2007, pp. 529-534.

[36] E. Al-Masri, Q.H. Mahmoud, “The QWS data set”, http://www.uoguelph.ca/~qmahmoud/qws/

[37] Z. Zheng, M.R. Lyu, “Collaborative Reliability Prediction for Service-Oriented Systems”, in Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering (ICSE2010), J. Kramer, J. Bishop, P. T. Devanbu, S. Uchitel (Eds.), Cape Town, South Africa, May 2-8, 2010, pp. 35 – 44.

[38] Z. Zheng, M.R. Lyu, “WS-DREAM: Web Service QoS Datasets”, http://www.wsdream.net/dataset.html

[39] A. Bramantoro, S. Krishnaswamy, M, Indrawan, “A semantic distance measure for matching web services”, Proceeding of the 2005 International Conference on Web Information Systems Engineering, A.H.H. Ngu, M. Kitsuregawa, E.J. Neuhold, J-Y. Chung, Q.Z. Sheng (eds), 2005, pp 217-226.

[40] TA. Sorensen, “A method of establishing groups of equal amplitude in plant sociology based on similarity of species content, and its application to analyses of the vegetation on Danish commons”, K dan Vidensk Selsk Biol Skr 5, 1948, pp. 1-34.

[41] LR. Dice, “Measures of the Amount of Ecologic Association Between Species”, Ecology vol. 26, no 3, 1945, pp. 297–302, doi:10.2307/1932409

[42] E. Hullermeier, M. Rifqi, S. Henzgen, R. Senge, “Comparing Fuzzy Partitions: A Generalization of the Rand Index and Related Measures”, IEEE Transactions on Fuzzy Systems, vol. 20, no 3, June 2012, pp. 546-556.

[43] V. Cross, X. Hu, “Using Semantic Similarity in Ontology Alignment”, Proceedings of the The Sixth International Workshop on Ontology Matching (collocated with the 10th International Semantic Web Conference ISWC-2011), P. Shvaiko, J. Euzenat, T. Heath, C. Quix, M. Mao, I. Cruz (eds), 2011, pp. 61-72.

[44] G. Hirst, D. St-Onge, “Lexical Chains as Representations of Context for the Detection and Correction of Malapropisms”, chapter in “WordNet: An Electronic Lexical Database”, Christiane Fellbaum (ed), chapter 13, 1998, pp. 305-332, The MIT Press, Cambridge, MA.

[45] T. Seidl, HP. Kriegel, “Optimal multi-step k-nearest neighbor search”, Proceedings of the 1998 ACM SIGMOD international conference on Management of data, L.M. Haas, A. Tiwary (eds), 1998, pp. 154-165.

[46] NB. Mabrouk, S. Beauche, E. Kuznetsova, N. Georgantas, V. Issarny, “QoS-aware Service Composition in Dynamic Service Oriented Environments”, Proceedings of Middleware 2009, LNCS vol. 5896, J.M. Bacon, B.F. Cooper (eds), 2009, pp 123-142.

[47] Apache Group, “Apache ODE Headers Handling”, 2012, http://ode.apache.org/headers-handling.html

[48] Oracle, “Manipulating SOAP Headers in BPEL”, 2013, http://docs.oracle.com/cd/E14571_01/integration.1111/e10224/bp_manipdoc.htm#CIHFCBAD

[49] GlassFish Community, “Glassfish”, 2013, http://glassfish.java.net/

[50] Glassfish Community, “Metro”, 2013, https://metro.java.net/

[51] HSQLDB, http://hsqldb.org/

[52] R. Ali, C. Solis, I. Omoronyia, M. Salehie, B. Nuseibeh, “Social Adaptation: When Software Gives Users a Voice”, Proceedings of the 7th Interantional Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2012), J. Filipe, L.A. Maciaszek (eds), 2012, pp. 75-84.

[53] D. Margaris, C. Vassilakis, P. Georgiadis, “An integrated framework for QoS-based adaptation and exception resolution in WS-BPEL scenarios”, Proceedings of the 28th ACM Symposium on Applied Computing, S.Y. Shin, J.C. Maldonado (eds), Coimbra, Portugal, 2013, pp. 1900-1906.

[54] D. Margaris, C. Vassilakis, P. Georgiadis, “Adapting WS-BPEL scenario execution using collaborative filtering techniques”, Proceedings of the IEEE 7th International Conference on Research Challenges in Information Science, RCIS 2013, R. Wieringa, S. Nurcan, C. Rolland, J-L. Cavarero (eds), Paris, France, 2013.

[55] P. Melville, RJ. Mooney, R. Nagarajan, “Content boosted collaborative filtering for improved recommendations”, Proceedings of the Eighteenth National Conference on Artificial Intelligence, R. Dechter, R.S. Sutton (eds), Canada, 2002, pp. 187-192.

[56] J. Aslam, M. Montague, “Models for metasearch”, Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval, SIGIR 2001, W.B. Croft, D.J. Harper, D.H. Kraft, J. Zobel (eds), 2001, pp. 276-284.

[57] M. Montague, J.A. Aslam, “Relevance Score Normalization for Metasearch”, Proceedings of the tenth international conference on Information and knowledge management, CIKM 2001, H. Paques, L. Liu, D. Grossman, C. Pu (eds), 2001, pp. 427-433.

[58] D. He, D. Wu, “Toward a Robust Data Fusion for Document Retrieval”, IEEE 4th International Conference on Natural Language Processing and Knowledge Engineering - NLP-KE, 2008.

[59] M. Alrifai, T. Risse, “Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition”, Proceedings of the 18th international conference on World wide web (WWW '09), Th. Karagiannis and M. Vojnovic (Eds.), 2009, pp. 881-890.

[60] F. Lécué, N. Mehandjiev, “Towards Scalability of Quality Driven SemanticWeb Service Composition”, Proceedings of the IEEE International Conference on Web Services (ICWS 2009), E. Damiani, R. Chang and J. Zhang (eds), 2009, pp. 469–476.

[61] M. Alrifai, T. Risse, “Efficient QoS-aware Service Composition”, In Emerging Web Services Technology Volume III, W. Binder, S. Dustdar (eds), Birkhäuser Basel, 2009.

[62] W. Mayer, R. Thiagarajan, M. Stumptner, “Service Composition as Generative Constraint Satisfaction”, IEEE International Conference on Web Services, 2009 (ICWS 2009), E. Damiani, R. Chang and J. Zhang (eds), 2009, pp. 888-895.

[63] L. Qi, X. Xia, J. Ni, Ch. Ma, Y. Luo, “A Decomposition-based Method for QoS-aware Web Service Composition with Large-scale Composition Structure”, Proceedings of the Fifth International Conferences on Advanced Service Computing, A. Koschel, J.L. Mauri (eds), 27 May - 1 Jun, 2013, Valencia, Spain, pp. 81-86.

[64] A.B. Hassine, S. Matsubara, T. Ishida, “A Constraint-Based Approach to Horizontal Web Service Composition”, Proceedings of the 5th International Semantic Web Conference, ISWC 2006, I.F. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, L. Aroyo (Eds.) Athens, GA, USA, November 5-9, 2006, pp. 130-143.

[65] IBM, “IBM ILOG CPLEX Optimizer”, 2013, http://www.ibm.com/software/commerce/optimization/cplex-optimizer/

[66] IBM, “Using CPLEX to examine alternate optimal solutions”, 2013, http://www.ibm.com/support/docview.wss?uid=swg21399929

[67] G. Castagna, N. Gesbert, L. Padovani, “A theory of contracts for Web services”, ACM Transactions on Programming Languages and Systems, vol. 31, issue 5, June 2009, article no 19.

[68] Y. Yanbe, A. Jatowt, S. Nakamura, and K. Tanaka, “Towards Improving Web Search by Utilizing Social Bookmarks”, In Web Engineering, LNCS vol. 4607, L. Baresi, P. Fraternali, G.J. Houben (eds), 2007, pp 343-357.

[69] C. Zeginis, D. Plexousakis, “Web Service Adaptation: State of the art and Research Challenges”, Institute of Computer Science, FORTH-ICS, Technical Report 410, ICS-FORTH, October 2010, http://www.ics.forth.gr/tech-reports/2010/2010.TR410_Web_Service_Adaptation.pdf

[70] R. Salakhutdinov, A. Mnih, G. Hinton, “Restricted Boltzmann machines for collaborative filtering”, Proceedings of the 24th international conference on Machine learning (ICML 07), Z. Ghahramani (ed), 2007, pp. 791-798.

[71] R. Beier, B. Vocking, “Typical Properties of Winners and Losers in Discrete Optimization”, Proceedings of the 36th ACM Symposium on Theory of Computing (STOC 2004), June 13–15, 2004, Chicago, Illinois, USA.

[72] K. Kritikos, D. Plexousakis, “Mixed-Integer Programming for QoS-Based Web Service Matchmaking”, IEEE Transactions on Services Computing, Volume 2 Issue 2, April 2009, pp. 122-139.

[73] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, R. Mirandola, “MOSES: A Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems”, IEEE Transactions on Software Engineering, vol. 38, no. 5, September/October 2012, pp. 1138-1159.

[74] T. Yu, K-J. Lin, “Service selection algorithms for Web services with end-to-end QoS constraints”, Proceedings of IEEE International Conference on e-Commerce Technology, 2004, pp. 129 - 136.

[75] J. Bisschop, “Linear Programming Tricks”, chapter in “AIMMS Optimization Modeling”, ISBN: 9781847539120. The chapter is available at http://www.aimms.com/aimms/download/manuals/aimms3om_linearprogrammingtricks.pdf

[76] StackOverflow, “Using min/max *within* an Integer Linear Program”, http://stackoverflow.com/questions/10792139/using-min-max-within-an-integer-linear-program