Adapting WS-BPEL scenario execution using collaborative filtering techniques

Dionisis Margaris and Panagiotis Georgiadis
Department of Informatics and Telecommunications
University of Athens
Athens, Greece
margaris@di.uoa.gr, p.georgiadis@di.uoa.gr

Costas Vassilakis
Department of Computer Science and Technology
University of Peloponnese
Tripoli, Greece
costas@uop.gr

***Abstract*— WS-BPEL has been adopted as the predominant method for composing individual web services into higher-level business processes. The designers of WS-BPEL scenarios define at development time the specific web services that will be invoked in the context of the business process they model; in the context however of the current web, where each functionality is offered by multiple service providers, under different quality of service parameters, using a fixed BPEL scenario has been recognized to be inadequate for servicing the diverse needs of business processes clients. To this end, WS-BPEL scenario execution adaptation has been proposed, mainly allowing clients to specify quality of service policies, which drive the dynamic selection of the services that will be invoked. In this paper, we present a framework extending the quality of service-based adaptation mechanisms with 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.**

Keywords— WS-BPEL; adaptation; personalization; collaborative filtering; quality of service

I. INTRODUCTION

Web Services are considered as 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, and 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.

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 (e.g. in a holiday planning application, the user may require that reservation is made in a particular hotel). Once a user has made some explicit service selections, recommender system technology could be used, complementary to the QoS specifications, to assist in the selection of services not explicitly set by the user, 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), they will act on other items similarly [7].

In this paper we present an approach for integrating collaborative filtering techniques into the WS-BPEL execution adaptation procedure, 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, while the BPEL execution framework includes provisions for (a) specifying QoS requirements for invocations of web services within a WS-BPEL scenario (b) selecting exact services to be invoked and (c) adapting the WS-BPEL scenario executions according to the recommendations of the 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. The novel features of this proposal lie in the use of the collaborative filtering in the adaptation procedure, as well as in the creation of the execution framework.

The rest of the paper is structured as follows: section II overviews related work, while section III presents the algorithm for service recommendation in the context of WS-BPEL scenario execution. Section IV presents the execution adaptation architecture and elaborates on the functionality of its components. Section V presents a performance evaluation study, aiming to substantiate the feasibility of the proposed approach. Finally, section VI concludes the paper and outlines future work.

II. Related Work

Recall from section I, that existing WS-BPEL execution adaptation approaches focus mainly on tailoring the execution of WS-BPEL scenarios according to the consumer's QoS requirements, following either the *horizontal-* or the *vertical-level* adaptation [3]. [4] presents AgFlow, which revises the execution plan in order to conform the user's QoS constraints. AgFlow may operate either using global planning, in which the execution plan is revised in order to conform the user's QoS constraints, or using local optimization, in which optimization is made on individual task basis, using the Simple Additive technique Weighting [5] to select the optimal service for a given task. VieDAME introduced in [7], 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 is platform-dependent, since it relies on extensions of the ActiveBPEL engine. Work in [8] addresses selection among equivalent services under QoS constraints, aiming to optimize the value of an objective function for the entire orchestration. To this end, it may employ either a heuristic (OPTIM_HWEIGHT) or a brute force (OPTIM_S) algorithm. While this work discusses the service selection algorithms, it does not describe an execution architecture for accommodating the proposed adaptation. [9] considers adaptation in the context of exception resolution (i.e. automatically substituting a failed service by an equivalent one, in order to guarantee successful completion of the business process), while [5] combines exception resolution with QoS-based adaptation; [10] extends this approach with the ability to handle parallel flows. Work in [11] 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, [12] proposes the use of collaborative filtering techniques for generating recommendations, 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.

In order to perform adaptation and/or exception resolution, all approaches employ some means to formally specify the necessary properties of the services and importantly (a) their functionality and (b) their QoS attribute values. Regarding services' functionality, some approaches need only record which services are *equivalent* [11], while others use more elaborate schemes, such as the one described in [14] according to which a service A may be related a service B through one of the following subsumption relationships: *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* relationships holds). The subsumption relationships 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. Hence, in this work we will adopt the subsumption relationships representation.

METEOR-S [15] 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 relationships; WSMO [16] is another widely adopted option with similar expressive power, regarding the features listed above.

In the collaborative filtering domain, several methods have been proposed, however their incorporation in the WS-BPEL execution adaptation process has not been considered. [17] surveys collaborative filtering, focusing on is use within the adaptive web. Interestingly, [17] 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. [18] presents an evaluation of collaborative filtering algorithms. Collaborative filtering is in many cases combined with another personalization technique, namely content based filtering; [19] and [20] are examples of such approaches. However, content-based filtering needs items with content to analyze [17], 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; this is mandatory or they would not be equivalent), 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. Thus, in this work we use only 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.

collaborative filtering approach, retaining only service usage patterns, in an anonymized form.

Finally, [21] examines the use of collaborative filtering techniques for suggesting web services to the users. The work in [21] 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.

III. The Service Recommendation Algorithm

As stated in section I, the adaptation algorithm chooses the services to be invoked in the context of the particular execution taking into account the following criteria:

i) The consumer's QoS specifications.

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

iii) The QoS characteristics of the available service implementations.

iv) The service subsumption relationships.

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

Items (i) and (ii) are provided per WS-BPEL scenario execution by the consumer, while items (iii)-(v) are drawn from repositories maintained by the adaptation scheme. In the following paragraphs, we briefly describe the QoS concepts and we elaborate on the representation of the subsumption relationships and the usage patterns, as well as the overall adaptation algorithm.

A. QoS concepts

QoS is generally defined in terms of attributes corresponding to non-functional aspects of services [21][23], with typical attributes being response time, availability, price, reputation, security and so forth [24]. The range of the QoS attributes is considered to be normalized in the range [0, 10]; value range normalization is a typical approach in works considering QoS-based adaptation (e.g. [4][7]). 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 availability, however some QoS attributes are more complex: for instance [25] identifies seven dimensions related to security, and some security mechanism S_1 may outperform some other security mechanism S_2 in some dimensions but lag behind S_2 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.

Without loss of generality, we will limit our discussion to attributes *response time* (rt), *availability* (av) and *price* (pr); the extension of the proposed algorithm and framework to include more QoS attributes is straightforward. Thus, each service implementation *S* considered in the adaptation process has a known QoS vector $QoS_S=(rt_s, av_s, pr_s)$ which is recorded in the repository (e.g. METEOR-S).

In the proposed approach, each WS-BPEL scenario execution request is coupled with a lower and an upper bound for each QoS attribute, indicating that only services with values between these two bounds should be considered in the adaptation process. Thus, the request-specific QoS specifications for each functionality *f* to be invoked in the context of the request may be defined in the form of two vectors, namely $MAX_f = (rt_f^{max}, av_f^{max}, pr_f^{max})$ and $MIN_f = (rt_f^{min}, av_f^{min}, pr_f^{min})$.

B. Subsumption relationship representation

In order to perform adaptation we need to represent the subsumption relationships between services, so as to be able to determine which services can be used to deliver a specific functionality. [14] and [27] address the representation of subsumption relationships 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 works 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 (since these will be actually invoked in the context of WS-BPEL scenario execution), we extend the tree scheme used in [14] and [27] 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 relationships, including categories and concrete services could be arranged as shown in Fig 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 relationships (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 s_1 and s_2 represent services.

- *Rule* C_{ex}*: c exact* s_1 iff *c* is the immediate parent of s_1 (e.g. Air travel and SwissAir in Fig. 1)
- *Rule* C_{pl}*: c plugin* s_1 iff *c* is an ancestor of s_1 (e.g. Ticket and SwissAir in Fig. 1)
- *Rule* S_{ex}*:*s_1 *exact* s_2 iff $\exists$ *c*: *c* is the immediate parent of s_1 and *c* is the immediate parent of s_2 (e.g. AirFrance and SwissAir in Fig. 1).
- *Rule* S_{pl}*:*s_1 *plugin* s_2 iff $\exists$ *c*: *c* is the immediate parent of s_1 and *c* plugin s_2 (e.g. SportsTicketBooker and NBAFinals in Fig. 1).

In all other cases, unconditional substitution cannot occur hence we compute a *fail* result for the operands.

The representation of subsumption relationships illustrated in Fig. 1 can be trivially extended to accommodate the QoS attribute values of concrete services, by simply attaching to each concrete service node the vector $QoS_S=(rt_s, av_s, pr_s)$ corresponding to the particular service's QoS metrics.

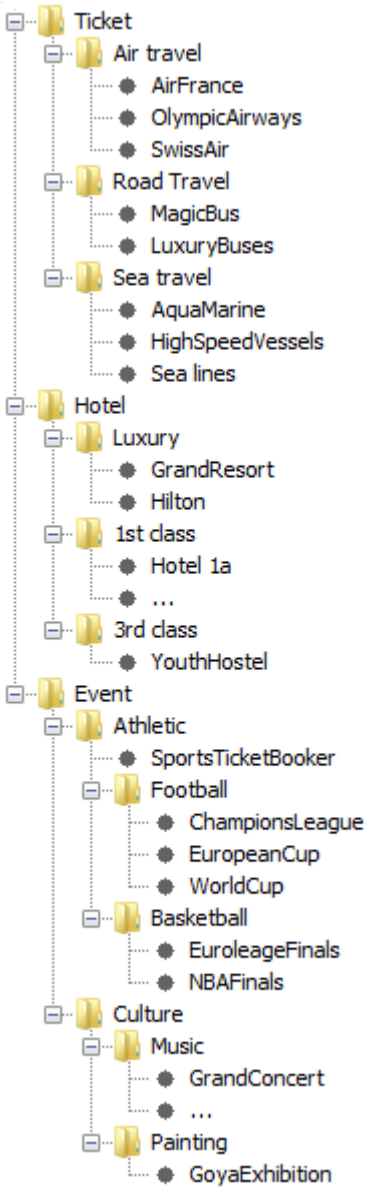

Fig. 1. Subsumption relationships for the travel planning WS-BPEL scenario

[26] describes the OPUCE platform repository, which could be used to provide both the necessary QoS data and the subsumption relationships. 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 Fig. 1.

C. *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. For simplicity purposes, in this paper we consider if a service should not be invoked in the context of a particular execution, this is explicitly signified through a specific input parameter.

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 subsection E, below.

D. *Service 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* [17], where each row corresponds to a user and each row corresponds to an item. Since our aim is to adapt WS-BPEL scenario executions based on past user choices, each row of the rating 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.

Such a ratings matrix tends to be sparse: since each functionality included in the WS-BPEL scenario will result to at most one invocation of some service that implements it (recall that it may result to zero invocations, if the consumer has designated that this piece of functionality should not be executed), 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. A more compact representation of the same data would be to use one column for each distinct functionality within the WS-BPEL scenario. Under this arrangement, the value of a cell *(i, j)* is *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 I illustrates a service usage pattern repository for the travel planning WS-BPEL scenario. In executions 1, 2 and 5 all functionalities were invoked, while in the remaining executions some functionalities were omitted, as per consumer request.

E. *The adaptation algorithm*

Having available the information listed in subsections A-D above, upon each invocation of a WS-BPEL scenario the execution adaptation algorithm determines the actual services that functionalities denoted as "to be recommended" should be

bound to. This binding is determined through the collaborative filtering process, and observing the bounds set for the QoS attributes of each functionality. The steps followed by the algorithm are described below; to illustrate the algorithm operation, we will use an example execution request:

TravelPlanner(bindings=(AirTravel(R), GrandResort, NBAFinals), QoSLimits={})

which can be effectively read as "I want to stay in GrandResort, I want to watch the NBA finals and I want a recommendation for an air travel; no QoS limits are imposed for the recommendation".

TABLE I. EXAMPLE USAGE PATTERNS REPOSITORY

# exec	Travel	Hotel	Event
1	OlympicAirways	YouthHostel	ChampionsLeague
2	SwissAir	Hilton	GrandConcert
3	HighSpeedVessels	YouthHostel	
4	LuxuryBuses		EuroleagueFinals
5	MagicBus	YouthHostel	NBAFinals
6	SwissAir	Hilton	

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, NBAFinals).

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 a 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.

   In our example, rows 1, 2 and 6 of Table 1 would be retrieved, since all other rows result to a *fail* subsumption relationship, regarding the ticket functionality "AirTravel".

   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.

   Regarding our example, $funct_i$=AirTravel and it corresponds to the first element of functionality vector F=(null, GrandResort, NBAFinals); hence, the functionality vector F(AirTravel) will be formulated with F(AirTravel)=(AirTravel, GrandResort, NBAFinals).

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.

   In our example, no QoS bounds have been set, therefore no rows are eliminated and the algorithm will proceed by considering rows 1, 2 and 6.

4. For each row retained by step 3, we compute its similarity with the current request, as the latter is represented by the F($funct_i$(request)) functionality vector. The similarity is calculated using the Sørensen similarity index [28] (alternatively known as Dice's coefficient [29]), 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 adapted to suit a domain with semantic similarities. The adaptation 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 in both sets [30]. 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. [31]. 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 distance metric between two functionalities (web services or categories), we adopt the one proposed in [27]:

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

   where C is a constant set to 8 [27][32], *lw* is the level weight for each path in subsumption tree (cf. Fig 1; the value of *lw* depends on the depth of the subsumption tree and the level of the node in it), PathLength is the number of edges counted from functionality s_1 to functionality s_2 and 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 tree level. 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), row1) = (19/24, 14/24, 15/24)

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

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

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

S(F(AirTravel), row1) = 2*(19/24+14/24+15/24)/6 = 0.667

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

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

5. Finally, the algorithm retains only the K-nearest neighbors (i.e. the rows with the highest similarity scores), it 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. The service corresponding to the group having the greatest sum is then selected to deliver the specific functionality in the context of the current execution. In our implementation, we have set the value of K to 10, which is a commonly used value [33].

   In our example, all rows would be retained, since we only have 3 rows, i.e. less than the limit of 10. Rows 2 and 6 form one group corresponding to service *Swissair* and achieving an overall score of 1.307, while row 1 forms a second group corresponding to service *OlympicAirways* with an overall score of 0.667 (service *AirFrance* has no corresponding rows and is not considered in this step). Thus, service *Swissair* is selected to implement the AirTravel functionality in the context of the current scenario execution.

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

Fig 2 illustrates the pseudocode for the recommendation algorithm.

```
/* adaptation algorithm pseudocode
 Assumption:
    - Scenario includes functionalities (i.e. invocations to services) f1, f2, …, fn
Inputs:
    - MIN and MAX (lower and upper bounds for QoS attributes)
    - Specification of bindings of functionalities to concrete services B=(b1, b2, …, bn) [bi == null if no binding provided, else id of service)
    - Specifications of functionalities not to be invoked O=(o1, o2, …, on) [oi == thrue if fi should not be invoked, false otherwise]
    - Specifications of functionalities for which recommendations are requested R=(r1, r2, …, rn) [ri == category of functionality if a recommendation is requested for fi, null otherwise]
 Outputs:
    - Bindings for services that a recommendation has been requested for P=(p1, p2, …pn) [pi == null if no recommendation was requested for fi, otherwise it contains the id of the service]
 */
 /* 1. Build scenario-level functionality vector */
 for (i = 1; i <= n; i++)
    if (B[i] != null) then
        F[i] = B[i];
    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 pattern repository */
 for (i = 1; i <= n; i++)
    if (R[i] != null) /* recommendation requested */
        matchingRows = selectFromUsageRepository row where
            B[i] exact row[i] ∨ B[i] plugin row[i]: ∀i: B[i] != null;
    /* FV[fi] is 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 formulate suggestions */
        qualifyingRows = topK_Scores(matchingRows, scores);
        groupedScores = groupRowScores(qualifyingRows, scores, fi);
        P[fi] = topScore(groupedScores);
    end if /* /* recommendation requested */
 end for
```

Fig. 2. The Recommendation algorithm pseudocode

IV. The Execution Adaptation Architecture

The execution adaptation architecture adopts the middleware-based approach, typically used in QoS-based adaptation frameworks (e.g. [5][7][34]). 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 III 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 utility web services, namely *getSessionId* (assigning unique session identifiers to individual WS-BPEL scenario executions), *prepareAdatation* (accepting QoS bounds and service binding information, and executing the algorithm of section III to select services for the functionalities that adaptation is requested for) 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. non-adaptive) WS-BPEL scenarios to a form that their execution can be readily adapted by the middleware. The preprocessor

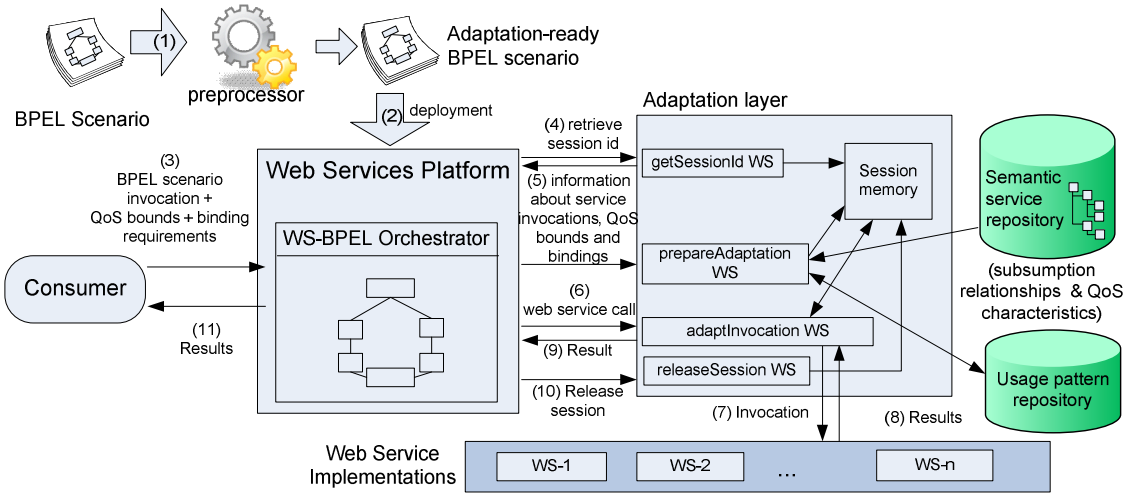

Fig. 3. The Execution Adaptation Architecture

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.

Fig. 3 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.

A. Preprocessing the 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 extracts from the incoming request the information regarding (a) the QoS bounds for the functionalities and (b) the designations regarding 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. For simplicity purposes, we will assume that the QoS bounds for a functionality *funct* used in the WS-BPEL scenario are represented using input parameters *QOSMAXinvName* and *QOSMAXinvName*, where *invName* is the value of the *name* attribute in the <*invoke*> construct realizing the functionality (<*invoke name="invName" partnerLink="lnk1" ...>)*. Similarly, the designations regarding functionality bindings are represented using input parameters *BINDINGinvName*, whose value 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 III, 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. Since the example invocation discussed in section III contained no QoS bounds, no *QOSMIN** and *QOSMAX** input parameters need to be set; the settings of the *BINDING** variables will be *BINDINGbookTicket=RECOMMEND*, *BINDINGbookHOTEL=GrandResort, BINDINGattendEvent=NBAFinals.*

   The code realizing this functionality is injected immediately after the invocation to *getSessionId*.

3. each service invocation is redirected to the *adaptInvocation*, 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, most contemporary WS-BPEL orchestration engines provide means to set request headers, e.g. [35][36].

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 deployed to the WS-BPEL orchestration engine and made available for execution.

B. *Executing the 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 and (c) 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 III, so as to produce any recommendations requested. After the algorithm has been applied, all service bindings (both those specified by the consumer and were provided as input to the *prepareAdaptation* web service, as well as those produced as output of the recommendation 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 sessions and the concrete services that these invocations should be directed to. In the example presented in section 3, the information that would be inserted into the session memory (assuming that the session identifier would be equal to s1) would be as shown in Fig 4.

```
(Session: s1;
    Bindings: (  bookTicket: Swissair;
                 bookHotel: GrandResort;
                 attendEvent: NBAFinals) )
```

Fig. 4. Information inserted in session memory

The *prepareAdaptation* web service finally stores the service bindings it received as input parameters into the usage pattern 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 IV.A, 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.

V. PERFORMANCE EVALUATION

In order to assess the performance of our approach and validate its feasibility, we have conducted a set of experiments, aiming to measure and quantify the overhead incurred due to the introduction of the middleware, the semantic trees and the BPEL scenarios stored in our database. The extent to which users are satisfied by the recommendations generated by the proposed algorithm is also of importance, and it is scheduled as part of our future work.

In these experiments we measured (a) the overhead imposed by the use of the invocations to the *getSessionId, prepareAdaptation* and *releaseSession* web services (invoked once per execution of a WS-BPEL scenario), (b) the overhead imposed for each web service invocation due to the intervention of the *adaptInvocation* service. In the experiment we have varied the following parameters:

1. the number of concurrent invocations,
2. the size of the usage pattern 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 evaluated, because the preprocessor operates in an offline fashion, not imposing thus any overhead to the performance of the production system.

For our experiments we used two machines: the first one (a workstation equipped with one Pentium 4@2.8GHz CPU and 512MB of RAM) hosted the preprocessor and the clients, while the second one (a workstation equipped with one Pentium i7@1.6 GHz and 4 GBytes of RAM) hosted the BPEL orchestration engine (a Glassfish application server [37]), the adaptation layer, the target web services, the service repository (which included the tree representation of the subsumption relationships and the services' QoS characteristics) and the usage pattern repository. Both repositories were implemented as HSQLDB [38] databases. The machines were connected through a 100Mbps local area network.

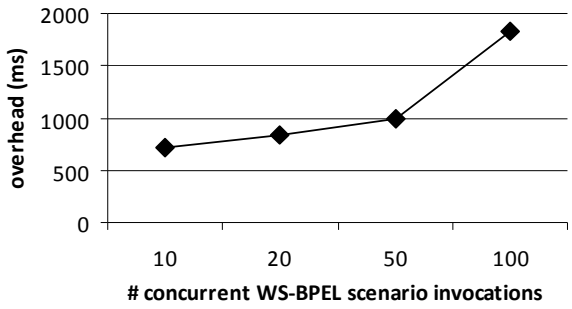

Fig. 5. Recommendation and housekeping overhead for varying degrees of concurrency

Fig. 5 presents the recommendation and housekeeping overhead, i.e. the overhead imposed by the use of the

invocations to the *getSessionId, adaptInvocation* 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 size of the usage pattern repository was set to 1,000 qualifying entries (i.e. the usage pattern repository contained 1,000 entries matching the functionality for which a recommendation was requested; cf. step 2 in subsection II.E), 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 observe a sharp increase on the overhead when the number of concurrent invocations raises from 50 to 100 concurrent invocations; this is due to the depletion of the second workstation's resources at this load range. Moving some tasks assigned to the second workstation (e.g. the adaptation layer and the repositories) to an extra machine is expected to provide smoother performance scaling.

Fig. 6 illustrates the time needed to perform recommendation and housekeeping tasks under a varying number of functionalities within the WS-BPEL scenario and for different counts of qualifying records in the usage pattern 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 by about 10%-12% when the number of functionalities increases by one; this is owing to the time needed for the extra processing to compute the semantic distances for the extra service.

In Fig. 6 we can also notice that the recommendation and housekeeping overhead is increasing linearly with the number of qualifying usage pattern repository entries. This was to be expected, since the current version of the algorithm considers and processes all entries fetched from the repository. The pruning of unpromising entries, e.g. entries whose cardinality is significantly greater than the cardinality of the functionality vector (increasing thus the denominator in the Sørensen's similarity index formula without a prospective increase in the nominator, hence leading to a poorer score) could contribute towards achieving better scalability with respect to the number of qualifying usage pattern repository entries.

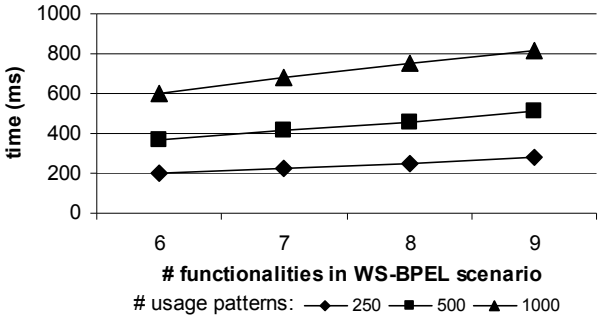

Fig. 6. Recommendation and housekeping overhead for varying number of functionalities within the WS-BPEL scenario and qualifying usage patterns

Additionally, maintaining pre-computed similarity metrics between all pairs of functionalities in the subsumption relationship tree, would help achieve better absolute times, trading off time for space, albeit it would not contribute towards improving scalability.

Fig. 7 illustrates the time needed to generate recommendations, with respect to the number of qualifying records in the usage pattern 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 was requested contained five functionalities. We can observe that the time needed for each recommendation is fairly stable, e.g. the time needed for making two recommendations is approximately double the time needed for making one recommendation, and similarly for making three recommendations. This is to be expected, since each recommendation is made individually, by repeating the same steps of the algorithm. The behavior regarding the number of qualifying usage patterns is analogous to the one observed in Fig. 6.

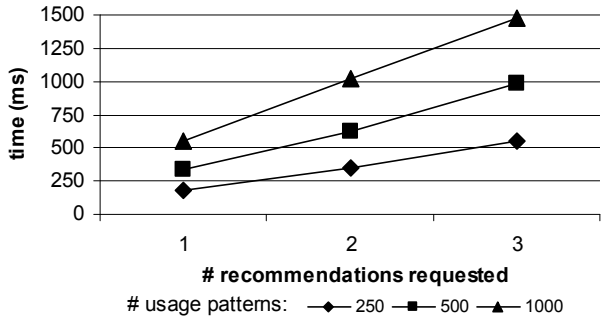

Fig. 7. Recommendation and housekeping overhead for varying number of qualifying usage patterns and number of requested recommendations

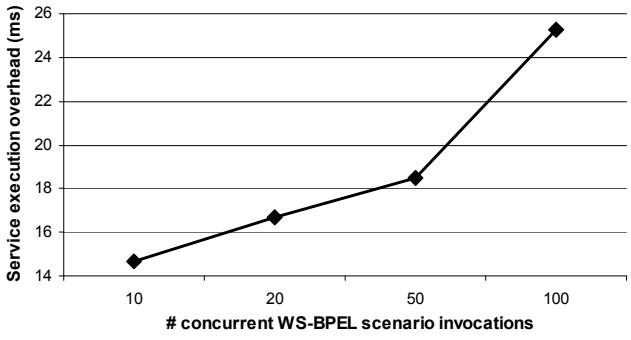

Fig. 8. Service execution overhead for varying degrees of concurrency

Finally, Fig. 8 depicts the overhead imposed for each service execution. This overhead corresponds to the two extra network messages required per service invocation (four network messages are needed when the adaptation layer intervenes as opposed to two messages only when services are directly invoked), plus the time needed for the adaptation layer to access the session memory, retrieve the service mappings and redirect the request appropriately. Similarly to Fig. 5, the overhead rises sharply when the number of concurrent invocations raises from 50 to 100 concurrent invocations, due

to the depletion of the second workstation's resources. The overhead is small, therefore once the *prepareAdaptation* service has concluded, the WS-BPEL scenario's performance is only minimally affected.

VI. CONCLUSION AND FUTURE WORK

In this paper we have presented an algorithm that uses collaborative filtering to adapt the execution of WS-BPEL scenarios so as to best suit the user requirements. The algorithm is complemented with an execution architecture, allowing for transforming existing WS-BPEL scenarios into an "adaptation ready" form, and adapting them on-the-fly according to the requirements posed by consumers. The execution architecture has been evaluated under varying degrees of concurrency, numbers of functionalities in the WS-BPEL scenario, numbers of requested recommendations, and sizes of the usage pattern repository to assess its feasibility for application in a production environment. The overhead incurred has been quantified to be reasonable, in most cases being less than 1 second, while the architecture's performance scales acceptably in regards to the factors listed above.

Our future work will focus on the optimization of the architecture's performance and the integration of QoS-adherence monitoring mechanisms, such as those described in [7]. We will also consider the incorporation of QoS weighting mechanisms, as used e.g. in [4] and [5]; under this scheme, both the QoS weights and the scores computed by the recommendation algorithm will be taken into account, to formulate the service bindings. 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 [39].

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, pp. 271-278.

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

[7] O. Moser, F. Rosenberg, S. Dustdar, "Non-Intrusive Monitoring and Service Adaptation for WS-BPEL", Proceedings of WWW 2008, Beijing, China, , 2008, pp. 815-824.

[8] Y. Xia, P. Chen, L. Bao, M. Wang, J. Yang, "A QoS-Aware Web Service Selection Algorithm Based on Clustering", Proceedings of ICWS11, 2011.

[9] C. Kareliotis, C. Vassilakis, P. Georgiadis, "Enhancing BPEL scenarios with dynamic relevance-based exception handling", Proceedings of ICWS07, Salt Lake City, Utah, USA, 9–13 July 2013, pp.751–758.

[10] D. Margaris, C. Vassilakis, P. Georgiadis, "An integrated framework for QoS-based adaptation and exception resolution in WS-BPEL scenarios", Proceedings of the ACM Symposium on Applied Computing, 2013, Coimbra, Portugal.

[11] 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, 2005, pp. 1069-1075.

[12] 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.

[13] S. Rinderle-Ma, M. Reichert, M. Jurisch, "Equivalence of Web Services in Process-Aware Service Compositions", Proceedings of ICWS'09, 6-10 July 2009.

[14] M. Paolucci, T. Kawamura, T. Payne, T. Sycara, "Semantic Matching of Web Services Capabilities", Proceedings of the International Semantic Web Conference, Sardinia, 2002, pp. 333-347.

[15] 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.

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

[17] JB. Schafer, D. Frankowski, J. Herlocker, S. Sen, "Collaborative Filtering Recommender Systems", in "The Adaptive Web", Lecture Notes in Computer Science Volume 4321, 2007, pp 291-324.

[18] 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.

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

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

[21] US. Manikrao, TV. Prabhakar, "Dynamic Selection of Web Services with Recommendation System" Proceedings of the International Conference on Next Generation Web Services Practices, 2005, pp. 117-121.

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

[23] ITU. Recommendation E.800 Quality of service and dependability vocabulary.

[24] J. Cardoso, "Quality of Service and Semantic Composition of Workflows", PhD thesis, Univ. of Georgia, 2002.

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

[26] 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.

[27] 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, 2005, pp 217-226.

[28] 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.

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

[30] 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.

[31] 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), 2011, pp. 61-72.

[32] 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.

[33] T. Seidl, HP. Kriegel, “Optimal multi-step k-nearest neighbor search”, Proceedings of the 1998 ACM SIGMOD international conference on Management of data, 1998, pp. 154-165.

[34] 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, 2009, pp 123-142.

[35] Apache Group, Apache ODE Headers Handling. http://ode.apache.org/headers-handling.html

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

[37] GlassFish Community, http://glassfish.java.net/

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

[39] 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).