QoS-Aware Service Composition in Dynamic Service Oriented Environments

Nebil Ben Mabrouk, Sandrine Beauche, Elena Kuznetsova, Nikolaos Georgantas, and Valérie Issarny

INRIA Paris-Rocquencourt, France
{nebil.benmabrouk,sandrine.beauche,elena.kuznetsova}@inria.fr,
{nikolaos.georgantas,valerie.issarny}@inria.fr

Abstract. QoS-aware service composition is a key requirement in Service Oriented Computing (SOC) since it enables fulfilling complex user tasks while meeting Quality of Service (QoS) constraints. A challenging issue towards this purpose is the selection of the best set of services to compose, meeting global QoS constraints imposed by the user, which is known to be a NP-hard problem. This challenge becomes even more relevant when it is considered in the context of dynamic service environments. Indeed, two specific issues arise. First, required tasks are fulfilled on the fly, thus the time available for services' selection and composition is limited. Second, service compositions have to be adaptive so that they can cope with changing conditions of the environment. In this paper, we present an efficient service selection algorithm that provides the appropriate ground for QoS-aware composition in dynamic service environments. Our algorithm is formed as a guided heuristic. The paper also presents a set of experiments conducted to evaluate the efficiency of our algorithm, which shows its timeliness and optimality.

1 Introduction

Service Oriented Computing (SOC) and its underlying technologies such as Web Services have emerged as a powerful concept for building software systems [1]. An interesting feature of SOC is that it provides a flexible framework for reusing and composing existing software services in order to build value-added service compositions able to fulfill complex tasks required by users. A key requirement in services' composition is to enable these tasks while meeting Quality of Service (QoS) constraints set by users.

QoS-aware service composition underpins this purpose since it allows for composing services able to fulfill user required tasks while meeting QoS constraints. Assuming the availability of multiple resources in service environments, a large number of services can be found for realizing every sub-task part of a complex task. A specific issue emerges to this regard, which is about selecting the best set of services (i.e., in terms of QoS) to participate in the composition, meeting user's global QoS requirements.

J.M. Bacon and B.F. Cooper (Eds.): Middleware 2009, LNCS 5896, pp. 123–142, 2009.

QoS-aware composition becomes even more challenging when it is considered in the context of dynamic service environments characterized with changing conditions. The dynamics of service environments bring about two specific problems in service selection. First, as dynamic environments call for fulfilling user requests on the fly (i.e., at run-time) and as services' availability cannot be known a priori, service selection and composition must be performed at runtime. Hence, the execution time of service selection algorithms is heavily constrained, whereas the computational complexity of the problem is NP-hard. The second issue is about the fluctuation of QoS conditions due to the dynamics of such environments. This problem arises for example when one or more services that make part of a service composition are no longer available or their QoS decreases (e.g., due to network disconnection or weak network connectivity) during the execution of the composition. Thus, a service selected to participate in a composition based on its QoS may no longer provide the same QoS when the time comes to be actually invoked. The overall question asked to this regard is: how to cope with the dynamics of service environments during the selection, the composition and the execution of services?

In this paper, we present a service selection algorithm that copes with the above issues. Our algorithm is designed in the context of the SemEUsE research project[1], which targets semantic QoS-aware middleware for dynamic service oriented environments. The middleware architecture presented in SemEUsE is centered on *dynamic binding* [2,3] of services, i.e., binding one out of multiple possible services just-in-time before its invocation according to its QoS measured at runtime (hereafter referred to as *runtime QoS*). Our selection algorithm underpins this purpose since it selects multiple services for every sub-task part of a complex task required by users, based on their nominal QoS (hereafter referred to as *advertised QoS*). Our algorithm consists in a guided heuristic. Our choice of a heuristic-based approach addresses the two issues stated above for dynamic environments. First, since the time available for service selection is limited, brute-force-like algorithms are inappropriate for such purpose, as they target determining the optimal composition, which is NP-hard. Second, finding the optimal composition may prove useless in the end since, due to dynamics, there is no guarantee that the selected composition will be possible at runtime or that its runtime QoS will not decrease with respect to the advertised one. To this regard, our algorithm aims at determining a set of near-optimal service compositions, i.e., compositions that: (i) respect global QoS constraints imposed by the user on the whole composition, and (ii) maximize a QoS utility function. At runtime, if a specific service composition is no longer possible or its QoS decreases, an alternative composition will be executed. To give a concrete example where our approach can be applied, we present the following scenario.

Motivating Scenario. An important use case where our solution can take place is the management of medical visits in large hospitals. Traditionally the management of medical visits in hospitals is static with predetermined

[1] SemEUsE project: http://www.semeuse.org

allocation of visits to doctors. Nevertheless, the availability of doctors can change with respect to some conditions. For instance, one or more doctors may be absent or they may be overloaded with new visits (e.g., due to some emergency cases unforeseen during the scheduling of visits). Human-based re-scheduling of medical visits is a time-consuming process entailing negotiations with doctors with respect to their specialties and agreements on the number of additional visits to be taken in charge.

A second issue concerns the process (i.e., the different activities) entailed by medical visits. Related to this, patients need to move between different points in the hospital in order to fulfill their visits. Ordinarily, they have to register, to pay for the visit, to meet the doctor and then to go to the pharmacy for buying medicines, which is a long and hard process especially for patients.

To avoid such complicated situations and to prevent patients unnecessarily moving between different points, hospitals need to manage their medical visits as a single request by composing the aforementioned activities in a unique process. Moreover, they need to dynamically handle these processes in order to cope with changing conditions in the hospital.

The SOC paradigm offers a flexible framework for managing the medical visits by reusing and composing existing software services of the hospital. Medical visits will be thus formed as processes (e.g., BPEL processes) underpinned by Web Services (e.g., registration, payment, doctor's service, chemist's service).

Let us consider a scenario where patients use the terminals available in the waiting room of the hospital to submit their medical visit requests. Using our solution, the hospital software system will be able to discover, select and compose the medical visit services (e.g., registration, payment, doctor's service, chemist's service) on-the-fly with respect to their QoS. Our solution considers common QoS features (e.g., response time) and domain-specific QoS features (e.g., doctors' specialties). Additionally, if the doctor's availability changes in-between, the hospital system will be able to dynamically update the composition by affecting the visit to another available doctor having the same specialty.

The remainder of this paper is structured as follows. In Section 2, we give an overview of related work. In Section 3, we present our service composition approach and we define the QoS model and the composition model underpinning this approach. In Section 4, we give the details of our selection algorithm, and we conduct a set of experiments to evaluate its timeliness and optimality in Section 5. Finally, in Section 6, we conclude with a summary of our contributions and the future perspectives of this work.

2 Related Work

Several selection algorithms have been proposed to select service compositions with different composition structures and various QoS constraints. A taxonomy of these solutions may be produced based on their objectives and the way they proceed. According to this, a first class of approaches aim at determining the optimal service composition (i.e., composition with the highest QoS utility) using

brute-force-like algorithms (e.g., Global Planning [4], BBLP, MCSP, WS-IP [5]). These solutions have high computational cost and they can not provide a solution in a satisfying amount of time, thus they are inappropriate to be used in the context of dynamic service environments.

To cope with this issue, other approaches propose heuristic-based solutions (e.g., WS-HEU and WFlow [5], Genetic algorithm [8,9,6,7,10,11,12]) aiming to find near-optimal compositions, i.e., compositions that respect global QoS constraints and maximize a QoS utility function. Yu et al. [5] present two heuristics, WS-HEU and WFlow, for the service selection problem. WS-HEU is specific heuristic applied to sequential workflows (i.e., workflows structured as a sequence of activities), whereas WFlow is designed for general workflow structures (i.e., sequential, conditional, parallel). The main idea of WFLow is to decompose workflows into multiple execution routes. WFlow considers a parameter ξ_i for every route indicating its probability to be executed. Therefore, it focuses on the route with the highest probability, whereas in our approach we aim at giving feasible service compositions regardless of the way the workflow will be executed.

Other approaches [8,9,6,7,10,11,12] present heuristics based on a genetic algorithm. The application of such algorithm to the service selection problem presents two main drawbacks: first, the order in which service candidates are checked is randomly chosen (e.g., Crossing [6]), whereas in our approach we aim at checking services in an ordered way to optimize the timeliness and the optimality of our algorithm. Second, as the genetic algorithm can run endlessly, the users have to define a constant number of iterations fixed *a priori*. However, fixing a high number of iterations does not give any guarantee about the quality of the result. Therefore, the genetic algorithm is deemed non useful for our purpose (i.e., selecting near-optimal compositions).

More recently, Alrifai et al. [13] presented a novel approach that combines local and global optimization techniques. This approach starts from the global level and resolves the selection problem at the local level. It proceeds by decomposing global QoS constraints (i.e., imposed by the user on the whole composition) into a set of local constraints (i.e., for individual sub-tasks, part of the composition). To do so, it uses MILP (mixed integer linear programming) techniques to find the best decomposition of QoS constraints. The main drawback of this approach is that it represents a greedy selection method, since it selects services at the local level and does not ensure that the global QoS constraints are respected.

3 Composition Approach Overview

Our approach starts from the assumption that the user (e.g., the patient in our scenario) uses a Graphical User Interface (e.g., terminals available in the waiting room of the hospital) to submit his/her request (e.g., medical visit). The interface guides the user to express his request in terms of functional and QoS requirements, and then it formulates these requirements as a machine-understandable specification.

User functional requirements are formulated as an abstract task (hereafter referred to as *abstract service composition*) brought about by the composition of a set of abstract sub-tasks (hereafter referred to as *activities*) (e.g., registration, payment, doctor's service, chemist's service). These activities are described with abstract information (i.e., function, I/O description). Abstract service compositions are later transformed into *concrete service compositions* by assigning a concrete service to every activity in the composition. Considering the multiple resources available in service environments, it is common that several concrete services are found for every activity; we refer to these services as *service candidates* of the considered activity.

Concerning user QoS requirements, they are formulated as a set of constraints (hereafter referred to as *global QoS constraints*) on the whole composition. These constraints cover several QoS attributes specified by the user. Further details about QoS attributes are given in Section 3.1, where we present the QoS model underpinning our approach.

Once user requirements are specified, we proceed by automatically building executable service compositions with respect to user requirements and the dynamics of the service environment. Building executable compositions consists of: (i) discovering, (ii) selecting, and (iii) composing services on-the-fly (i.e., at runtime).

Concerning services' discovery, we adopt a semantic-based approach introduced by Ben Mokhtar et al. [14,15]. This approach uses domain-specific and QoS ontologies to match user functional and QoS requirements to services available in the environment. The matching is based on an efficient semantic reasoning performed at runtime. For every activity in the composition, the discovery phase gives the set of service candidates able to fulfill the activity (i.e., functional aspect) and to respect user QoS requirements. Services' discovery uses advertised QoS of services to perform a preliminary filtering ensuring that individual service candidates respect user QoS requirements.

Refining the first filtering, the selection phase ensures user QoS requirements at the global level (i.e., for the whole composition) based on the advertised QoS of services. That is, it selects a set of service candidates for each abstract activity that, when composed together, meet global QoS constraints. To achieve this, we introduce a heuristic algorithm based on clustering techniques, notably K-Means [16]. Clustering techniques, applied to our purpose, allow for grouping services with respect to their QoS into a set of clusters, to which we refer as *QoS levels*. Further, we use the resulting QoS levels to determine the utility of service candidates regarding our objective, i.e., selecting near-optimal compositions. More specifically, our heuristic algorithm deals with the service selection problem in two phases: (1) a local classification phase, which aims at determining the utilities of service candidates using clustering; this phase is performed for every activity in the composition; (2) a global selection phase which uses the obtained utilities to guide the selection of near-optimal compositions.

Once the global selection is fulfilled, the composition phase uses the selected services to define an executable service composition, by replacing every abstract

activity in the composition with a 'dynamic binding' activity that takes as input the set of selected candidate services for this activity. At runtime, a unique service is selected and enacted among the provided ones with respect to its runtime QoS.

3.1 QoS Model

We consider a generic QoS model based on our previous work [17], in which we introduced a semantic QoS model formulated as a set of ontologies for QoS specification in dynamic service environments. This model allows for specifying cross-domain QoS attributes like response time, availability, reliability, throughput as well as domain-specific QoS attributes, e.g., medical visit price with respect to our scenario. Our model provides a detailed taxonomy of QoS which is flexible and easily extendible. Herein, we introduce an extension that concerns a particular classification of QoS attributes needed for our composition approach. QoS attributes can be divided into two groups: quantitative attributes (e.g., response time, availability, reliability, throughput) and qualitative attributes (e.g., security, privacy of medical information in our scenario). The former attributes are quantitatively measured using metrics, whereas the latter attributes can not be measured, they are rather evaluated in a boolean manner (i.e., they are either satisfied or not). For the sake of simplicity and without loss of generality, in this work we will consider only quantitative QoS attributes, since qualitative attributes can be represented as quantitative attributes determined by boolean metrics (i.e., 0 and 1).

Quantitative QoS attributes are in turn divided into two classes: negative attributes (e.g., response time, medical visit price) and positive attributes (e.g., availability, reliability, throughput). The first class of attributes has a negative effect on QoS, (i.e., the higher their values, the lower the QoS), hence they need to be minimized. On the contrary, positive QoS attributes need to be maximized, since they increase the overall QoS (i.e., the higher their values, the higher the QoS).

On the other hand, QoS attributes' values are determined in two ways: During the selection of services, these values are given by service providers (e.g., based on previous executions of services or using users' feedback). As already stated, we refer to these values as *advertised QoS*, which is specified in services' descriptions. At runtime, QoS values are provided by a monitoring component to enable further dynamic evaluation of services. As already stated, we refer to these values as *runtime QoS*.

3.2 Composition Model

Our algorithm aims at determining a set of near-optimal compositions. Such purpose requires evaluating the QoS of possible service compositions with respect to their structure and the way QoS is aggregated. That is, the evaluation of QoS depends on the structuring elements used to build the composition, to which we refer as *composition patterns*, and also QoS aggregation formulas associated with each pattern. Next, we describe the composition patterns on which our approach

is based and we give the aggregation formulas associated with QoS attributes and composition patterns.

Composition Patterns. We consider a set of patterns commonly used by composition approaches [4,5], which cover most of the structures specified by composition languages (such as BPEL) [18,19]:

- Sequence: sequential execution of activities
- AND: parallel execution of activities
- XOR: conditional execution of activities
- Loop: iterative execution of activities

Computing the QoS of Composite Services. For every activity in the abstract service composition, we represent the QoS of a single candidate service S_i by using a vector $QoS_{S_i} = \langle q_{i,1}, ..., q_{i,n} \rangle$, where n represents the number of QoS attributes required by the user and $q_{i,j}$ represents the value of the QoS attribute j $(1 \leq j \leq n)$. The QoS of a service composition is evaluated based on the QoS vectors of its constituent services while taking into account the composition patterns. Regarding QoS associated with AND and XOR, we adopt a pessimistic approach that considers worst-case QoS values. That is, to determine the values of the QoS attributes of a service composition, we consider the worst QoS values of all the possible executions of the composition. For instance, to determine the response time of parallel activities (i.e., AND), we consider the activity with the longest response time. Concerning the particular case of iterative activities (i.e., structured as a loop), we adopt a history-based estimation that considers the maximum number of loops (i.e., pessimistic approach). This number is determined from previous executions of the activity. In Table 1, we show examples of QoS computation with respect to QoS attributes and composition patterns. These examples can be classified as cross-domain QoS attributes (e.g., *response time, reliability, availability, throughput*) and domain-specific QoS attributes (e.g., *medical visit price*), but also as negative attributes

Table 1. QoS computation examples: $rt_i, re_i, av_i, th_i, p_i$ represent respectively, response time, reliability, availability, throughput and the medical visit price of services candidates structured with respect the composition patterns, whereas RT, RE, AV, TH, P represent the aggregated values of response time, reliability, availability, throughput and the medical visit price, respectively

QoS attributes	Composition Patterns			
	Sequence	AND	XOR	Loop
Response time (RT)	$\sum_{i=1}^{n} rt_i$	$max(rt_i)$	$max(rt_i)$	$rt \times k$
Reliability (RE)	$\prod_{i=1}^{n} re_i$	$\prod_{i=1}^{n} re_i$	$min(re_i)$	re^k
Availability (AV)	$\prod_{i=1}^{n} av_i$	$\prod_{i=1}^{n} av_i$	$min(av_i)$	av^k
Throughput (TH)	$min(th_i)$	$min(th_i)$	$min(th_i)$	th
Medical visit price (P)	$\sum_{i=1}^{n} p_i$	$\sum_{i=1}^{n} p_i$	$max(p_i)$	$p \times k$

(e.g., *response time, medical visit price*) and positive attributes (e.g., *reliability, availability, throughput*). Let us consider for example, the QoS computation of the medical visit price. Concerning the Sequence and AND patterns, the price is the sum of p_i values associated with the involved services (e.g., meeting doctors, buying medicines). For the XOR pattern (e.g., meeting two doctors with different specialties in an exclusive manner decided based on pre-diagnosis) the price is the maximum among p_i values of the involved services. Finally, for the iterative pattern (i.e., loop), the aggregated price is the value p of the repeated service multiplied by the number of loops k.

Notations. To state the problem that we are addressing in a formal way, we use the following notations:

- $AC = \{A_1, ..., A_x\}$ is an abstract service composition with x activities.
- $CC = \{S_1, ..., S_x\}$ is a concrete service composition with x service candidates, every service candidate S_i is bound to an abstract activity A_i $(1 \leq i \leq x)$.
- $U = \{U_1, ..., U_n\}$ is a set of global QoS constraints imposed by the user on n QoS attributes.
- QoS of a service candidate S_i is represented as a vector $QoS_{S_i} = \langle q_{i,1}, ..., q_{i,n} \rangle$ where $q_{i,j}$ is the advertised value of QoS attribute j $(1 \leq j \leq n)$.
- QoS of a concrete service composition CC is represented as a vector $QoS_{CC} = \langle Q_1, ..., Q_n \rangle$ where Q_j is the aggregated value of QoS attribute j $(1 \leq j \leq n)$.
- Each service candidate S_i has an associated utility function f_i.
- Each concrete service composition CC has an associated utility function $\mathcal{F}$.

4 Service Selection Algorithm

In the literature, service selection algorithms fall under two general approaches: (i) local [4] and (ii) global selection [5]. The former proceeds by selecting the best services (in terms of QoS) for every abstract activity individually. This approach has a low computational cost but it does not guarantee meeting global QoS constraints imposed by the user. For instance, regarding our scenario, this approach proceeds by selecting services offering the best trade-off between the required QoS attributes (e.g., response time, availability, reliability, throughput and medical visit price) for every activity apart. Thus, it cannot handle, for example, the global response time of the whole composition.

Conversely, global selection ensures meeting global QoS constraints since it selects the optimal service composition, i.e, a composition which respects global QoS constraints and has the highest QoS. This approach considers all possible compositions of services and selects the optimal one.

Nevertheless, the computational cost of global selection is NP-hard. To meet global QoS constraints in a timely manner, we present a heuristic algorithm

that combines local and global selection techniques. Starting from the assumption that service candidates (for every activity in the abstract process) are already given by the semantic discovery phase, our algorithm proceeds through the following phases:

1. **Scaling phase,** which is a pre-processing phase aiming to normalize QoS values associated with negative and positive QoS attributes;
2. **Local classification,** which aims at classifying candidate services (for every activity in the abstract process) according to different QoS levels; this classification is further used to determine the utilities of every service candidate regarding our purpose;
3. **Global selection,** which aims at using the obtained utilities to guide the selection of near-optimal compositions.

4.1 Scaling Phase

As already mentioned, QoS attributes can be either negative or positive, thus some QoS values need to be minimized whereas other values have to be maximized. To cope with this issue, the scaling phase normalizes every QoS attribute value by transforming it into a value between 0 and 1 with respect to the formulas below [4].

$$\text{Negative attributes :} \quad q'_{i,j} = \begin{cases} \frac{q_j^{max} - q_{i,j}}{q_j^{max} - q_j^{min}} & \text{if} \quad q_j^{max} - q_j^{min} \neq 0 \\ 1 & \text{else} \end{cases} \tag{1}$$

$$\text{Positive attributes :} \quad q'_{i,j} = \begin{cases} \frac{q_{i,j} - q_i^{min}}{q_j^{max} - q_j^{min}} & \text{if} \quad q_j^{max} - q_j^{min} \neq 0 \\ 1 & \text{else} \end{cases} \tag{2}$$

where $q'_{i,j}$ denotes the normalized value of QoS attribute j associated with service candidate S_i. It is computed using the current value $q_{i,j}$ and also q_j^{max} and q_j^{min}, which refer respectively to the maximum and minimum values of QoS attribute j among all service candidates.

The same formulas are also used to normalize the aggregated QoS values of concrete service compositions. Each composition CC is represented by a vector $QoS_{CC} = \langle Q_1, ..., Q_n \rangle$ with n QoS attributes. The normalization produces a QoS vector $QoS_{CC} = \langle Q'_1, ..., Q'_n \rangle$. The values of Q'_j $(1 \leq j \leq n)$ are computed based on the current value Q_j, and also Q_j^{max} and Q_j^{min}, which refer respectively to the maximum and minimum values of Q_j among all concrete service compositions.

4.2 Local Classification

Local classification is performed locally for every activity in the abstract service composition. It aims at classifying service candidates associated with a given activity into multiple QoS levels (i.e., clusters) with respect to their QoS. Each level contains the set of service candidates having roughly the same QoS. This classification is further used to determine the relative importance of service candidates regarding our objective (i.e., selecting near-optimal compositions). To do so, we use clustering techniques, notably the K-means [16] algorithm.

Classification Overview. K-means provides a simple and efficient way to classify a set of data points into a fixed number of clusters. These data points are characterized by their N-dimensional coordinates $\langle x_1, x_2, .., x_n \rangle$. The main idea of K-means is to define a centroid $c = \langle x_{c,1}, x_{c,2}, .., x_{c,n} \rangle$ for every cluster and to associate each data point $dp_i = \langle x_{i,1}, x_{i,2}, .., x_{i,n} \rangle$ to the appropriate cluster by computing the shortest N-dimensional Euclidian distance D between the data point and each centroid:

$$D_{(c,dp_i)} = \sqrt{\sum_{j=1}^{n} (x_{c,j} - x_{i,j})^2} \tag{3}$$

Further, the values of centroids are updated by computing the average of their associated data points. The clustering iterates by alternating these two steps (i.e., updating centroids, clustering data points) continuously until reaching a fixpoint (i.e., centroids' values do not change any more). The result of K-means will be the set of final clusters and their associated data points. It is worth noting that K-means has a polynomial computational cost in function of the number of iterations [20].

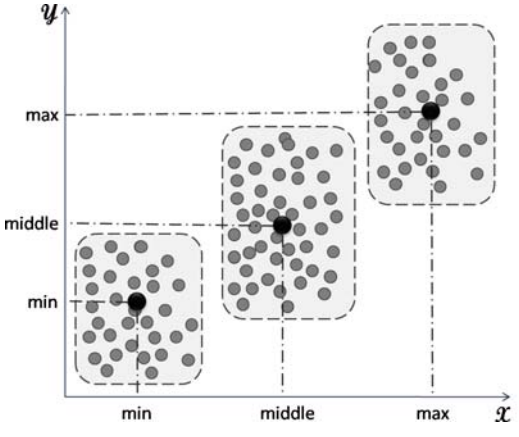

Fig. 1. Example of K-means with 2 dimensions (x, y) and 3 clusters: *min, middle* and *max*

In our context, we use K-means to group service candidates of every activity in the abstract service composition into multiple QoS levels. QoS levels are thus represented as clusters and service candidates are considered as data points determined by the QoS vectors $QoS_{S_i} = \langle q_{i,1}, ..., q_{i,n} \rangle$.

QoS Levels Computation. To cluster service candidates, we need first to determine the initial values of QoS levels (i.e., centroids). For this matter, we

define m QoS levels (i.e., QL_l, $(1 \leq l \leq m)$), where m is a constant number fixed a priori (Fig. 2). The value of m differs from an activity to another and it is supposed to be given by domain experts with respect to the *service density* [17] of the considered activity. For instance, in our medical visit scenario, the number of QoS levels related to the doctors' activity is fixed by the hospital system administrator with respect to the number of doctors in the hospital. Once the number of QoS levels is fixed, the value of each level is determined by dividing the range of the n QoS attributes (fixed by the global QoS constraints) into m equal quality ranges qr with respect to the following formula:

$$qr_j^l = q_j^{min} + \frac{l-1}{m-1} * (q_j^{max} - q_j^{min}) \qquad 1 \leq l \leq m \tag{4}$$

where qr_j^l denotes the quality range l of QoS attribute j with $(1 \leq j \leq n)$, whereas q_j^{max} and q_j^{min} refer to the maximum and minimum values of the attribute j, respectively. The initial value of each QoS level is then:
$QL_l = \langle qr_1^l, ..., qr_n^l \rangle$ with $1 \leq l \leq m$.

Once the initial values of QoS levels are determined, we perform the clustering of service candidates, and then we obtain the final set of QoS levels which is used to determine the utility of service candidates.

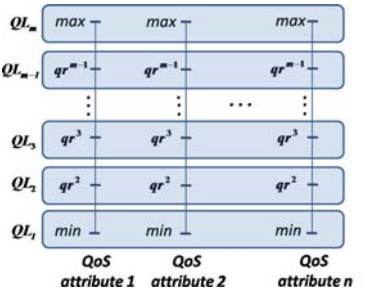

Fig. 2. Computation of Quality Levels

Service Utility Computation. The objective of our algorithm is selecting near-optimal compositions, but also obtaining a number of near-optimal compositions as large as possible. Indeed, the larger the number of selected compositions is, the larger is the choice of services allowed during dynamic binding. Additionally, providing a large number of compositions helps preventing the *starvation* problem during dynamic binding of services. This problem arises when, e.g., a few number of services are selected for dynamic binding but none of them is available at runtime.

For this matter, we consider a utility function f_i which characterizes the relative importance of a service candidate S_i regarding the objective above. The utility f_i is calculated based on two parameters: (i) QoS of S_i and, (ii) the number

of services in the QoS level to which S_i belongs. The first parameter is interpreted as follows: the higher QoS of S_i, the higher its ability to be part of feasible compositions. Concerning the second parameter, it represents the importance of the QoS level QL_l to which S_i belongs, i.e., if the number of service candidates associated with QL_l is large, this means that using QL_l would eventually lead to finding more feasible compositions. Therefore, f_i is computed as follows:

$$f_i = (r/t) * qos_i \qquad where \quad qos_i = (\sum_{j=1}^{n} q'_{i,j})/n \tag{5}$$

where r is the number of services in the QoS level QL_l to which S_i pertains, t is the total number of service candidates for the activity, and qos_i is the QoS utility of service S_i. It is computed as the average of the normalized QoS attributes' values $q'_{i,j}$. As the values (r/t) and qos_i are comprised between 0 and 1 (i.e., since $r \leq t$ and $0 \leq q_{i,j} \leq 1$, respectively), the value of f_i is also comprised between 0 and 1.

4.3 Global Selection

Global selection aims at selecting near optimal compositions, i.e., compositions that (i) respect global QoS constraints and (ii) maximize the utility function $\mathcal{F}$. The utility function $\mathcal{F}$ of a concrete service composition CC with $QoS_{CC} = \langle Q'_1, ..., Q'_n \rangle$ is defined as the average of its normalized QoS values Q'_j:

$$\mathcal{F} = (\sum_{j=1}^{n} Q'_j)/n \tag{6}$$

Therefore, the problem that we are addressing can be stated as finding concrete service compositions that fulfill these two conditions:

1. For every QoS attribute j $(1 \leq j \leq n)$,
   - $Q_j \leq U_j$ for negative attributes;
   - $Q_j \geq U_j$ for positive attributes;
2. The QoS utility $\mathcal{F}$ is maximized.

Heuristic Overview. The goal of our heuristic is to use the utilities f_i resulting from the local classification phase to select near-optimal compositions without considering all possible combinations of services. Towards this purpose, we fix a utility threshold $\mathcal{T}$ that allows for considering only service candidates with a utility value $f_i \geq \mathcal{T}$, thus enabling to focus on the most eligible services (i.e., services with the highest f_i values).

The choice of the threshold $\mathcal{T}$ is of great importance in our algorithm since it allows for tuning the trade-off between the number of resulting compositions and the timeliness of the algorithm. Indeed, if $\mathcal{T}$ increases, the number of considered services possibly decreases and consequently so will the number of compositions to check. Hence, the execution time of the algorithm decreases, but the number of

resulting near-optimal compositions decreases as well. Conversely, if $\mathcal{T}$ decreases, the number of services to consider possibly increases and hence, the number of obtained compositions possibly increases, too. However, the execution time of the algorithm increases as well.

The latter point leads to another important result, which is about the application of our algorithm. Indeed, tuning $\mathcal{T}$ makes our algorithm generic and flexible, so that it can be applied to multiple dynamic service environments according to their characteristics, particularly their *service density* [17] and also time constraints in such environments. For instance, if a service environment has a high service density, the system can tune $\mathcal{T}$ so that the algorithm will be more selective and give a satisfying number of service compositions. By the same, if the execution time in dynamic service environments is heavily constrained (e.g., highly dynamic environments), the system can also tune $\mathcal{T}$ to make the algorithm check a limited number of service compositions, thus enabling to respect timeliness constraints of such environments.

Pruning the Search Tree. Our algorithm proceeds by exploring a combinatorial search tree built from candidate services according to the following rules:

- Every service candidate S_i having $f_i \geq \mathcal{T}$ is a node in the search tree;
- If there is a link (i.e., control flow) from activity A_x to activity A_y in the abstract service composition, then the candidate services of A_x will be the child nodes of every service candidate in A_y;
- Child nodes (i.e., services associated with an activity A_i) are sorted from left to right according to their utility values f_i. Services with higher values of f_i are on the left and those with lower values are on the right.
- Add a virtual root node to all the nodes without incoming links.

Once the search tree is built, our heuristic algorithm ensures that its constituent service compositions meet user QoS requirements. Towards this purpose, it first generates a global QoS aggregation formula (i.e., for the whole composition) for every QoS attribute by exploring the structure of the composition. Then it uses the generated formulas to compute the aggregated QoS value of each attribute and the QoS utility of service compositions. The algorithm further checks the feasibility of these compositions by setting the global QoS constraints given by the user as upper bounds for the aggregated QoS values. The above step in performed along with the following optimizations aiming to prune the search tree of our algorithm.

– **Pruning using incremental computation.** As our algorithm traverses down the search tree from the root node to the leaf nodes, the aggregated QoS values increase along with the traversal of the tree. Consequently, if the aggregated QoS values calculated at any non-leaf node in the traversal of the tree, does not respect QoS constraints, then all the sub-tree under the non-leaf node will be pruned. This optimization is useful when we deal with long running processes having a large number of activities.

– **Pruning using utility values approximation.** This idea concerns an approximation rather than an exact optimization. It utilizes the fact that our algorithm explores the search tree in an ordered way, i.e., it checks services with higher f_i values first. Therefore, if a service candidate S_i does not lead to any feasible composition, all its following nodes (i.e., service candidates of the same activity but with lower f_i values) will be not considered for the rest of the computation, which reduces the number of services to check. This approximation is convenient when we have a large number of candidate services per activity.

Our algorithm uses the above optimizations together, along with an additional improvement allowing to enhance the timeliness of the algorithm. Indeed, to reduce the time needed for computing the aggregated QoS values of service compositions, we ensure that only one service candidate changes when the algorithm switches from a composition to another. That is, the difference between two consecutive compositions CC_v and CC_w is that a service candidate S_i in the first composition will be replaced by a service S_j in the second one. Thus, instead of computing the whole aggregated QoS values of CC_w, the algorithm updates the aggregated QoS values of CC_v with respect to QoS_{S_i} and QoS_{S_j}.

Finally, our algorithm produces as output the set of near-optimal compositions ranked according to their utilities $\mathcal{F}$. The obtained compositions are then used for the generation of an executable service composition underpinning dynamic binding of services.

5 Experimental Evaluation

5.1 Experimental Setup

We conducted a set of experiments to evaluate the quality of our algorithm. These experiments were conducted on a Dell machine with two AMD Athlon 1.80GHz processors and 1.8 GB RAM. The machine is running under Windows XP operating system and Java 1.6. In these experiments, we focus on two metrics:

– **Execution time.** This metric measures the response time of our algorithm with respect to the size of the problem in terms of the number of activities and the number of services per activity. In these experiments, we measure separately the execution time of local classification and global selection.
– **Optimality.** This metric measures how close the utility of the best composition given by our algorithm to the utility of the optimal composition given by the brute-force algorithm. The optimality metric is then given by the following formula:

$$Optimality = \mathcal{F}/\mathcal{F}_{opt} \tag{7}$$

where $\mathcal{F}$ is the utility of the best composition given by our heuristic algorithm and $\mathcal{F}_{opt}$ is the utility of the optimal composition given by the brute force algorithm.

In our experiments, we use the data given by previous studies about Web Services' QoS [21,22]. In these studies, the authors provide a set of QoS metrics (i.e., response time, throughput, availability, validation accuracy, cost) related to current email validation Web services (Table 2). We use these metrics as a sample input data for our algorithm. Nevertheless, the number of Web services considered in these studies is too limited compared to the number of services that we need to assess the scalability of our algorithm. To this regard, we developed a *Data Generator* that randomly generates input data for our algorithm between the minimmum and maximum values of the QoS metrics given in Table 2. Further, we developed a *Process Generator* that randomly generates abstract

Table 2. QoS metrics for email validation Web Services

Service Provider	Response Time (ms)	Throughput (req./min)	Availability (%)	Validation Accuracy (%)	Cost (cents/invoke)
XMLLogic	720	6.00	85	87	1.2
XWebservices	1100	1.74	81	79	1
StrikeIron	912	10.00	96	94	7
CDYNE	910	11.00	90	91	2
Webservicex	1232	4.00	87	83	0
ServiceObjects	391	9.00	99	90	5

processes to use as input for experimenting with our algorithm. The Process Generator takes as arguments the number of activities and the number of candidate services per activity, and it yields as output a process by structuring the activities with respect to randomly chosen composition patterns. The Process Generator uses the Data Generator to provide the QoS values associated with service candidates of each activity in the process.

For the purpose of these experiments, we vary the number of activities and the number of services per activity between 10 and 50. Concerning the number of QoS constraints, it is comprised between 2 and 5 constraints. Finally, for the sake of precision we execute each experiment 20 times and we calculate the mean value of the obtained results.

Once data input is generated, we need to fix the values of the following parameters before launching the experiments:

- We set the values of global constraints given by the user to the mean value m of every QoS attribute aggregated with respect to the structure of the generated process composition.
- We use the method of computing QoS levels described in Section 3.2 to cluster service candidates according to 3 clusters: *Min, Middle* and *Max*.
- Concerning the computation of the utility threshold $\mathcal{T}$, we fix it to $(m + \sigma)$ where m and σ denote respectively, the mean value and standard deviation of f_i utilities of all service candidates. As we have a large number of service

candidates, we assume that the values of f_i are normally distributed. According to this, the *central limit theorem* [23] states that the value $(m + \sigma)$ allows for discarding approximately 74% of service candidates.

5.2 Experimental Results

During the experiments, we aimed to compare the execution time of our algorithm to the execution time of a brute-force algorithm that we developed for the purpose of these experiments. Nevertheless, the latter algorithm takes a long time to execute (i.e., several hours) for a number of activities more than 20. Hence, we are not going to present the execution time of both algorithms, we will rather present the measurements obtained for our algorithm.

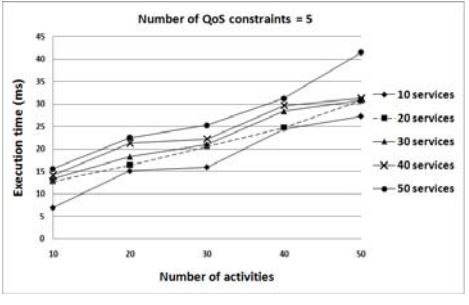

Fig. 3. Execution time of the local classification phase (for a fixed number of QoS constraints)

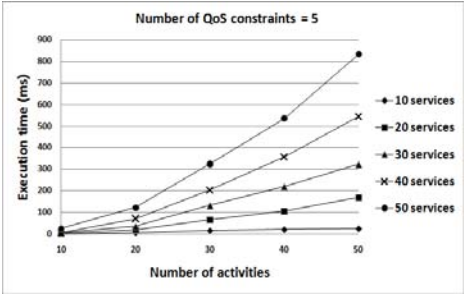

Fig. 4. Execution time of the global selection phase (for a fixed number of QoS constraints)

Figures 3 and 4 show the execution time of local classification and global selection, respectively. These measurements are obtained by fixing the number of QoS constraints to 5 and varying the number of activities and the number of service candidates per activity between 10 and 50. The obtained measurements show that the execution time of our algorithm increases along with the number of activities and the number of services per activity, which is an expected result. Conversely, in Figures 5 and 6, we measure the execution time of our algorithm while fixing the number of service candidates per activity to 50, and varying the number of activities between 10 and 50 and the number of QoS constraints between 2 and 5. These figures show that the execution time of our algorithm also increases along with the number of activities and the number of QoS constraints.

Additionally, it is worth noting that the execution time of the local classification phase is approximately negligible compared to the execution time of the global selection phase (i.e., 45ms $\ll$ 0.8s), which is an expected result given that K-Means is a simple algorithm with a polynomial computational cost [20]. Overall, in almost all cases our algorithm is executed in a reasonable amount of

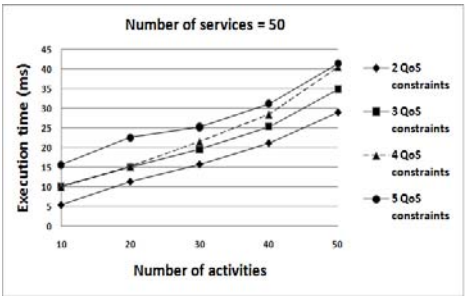

Fig. 5. Execution time of the local classification phase (for a fixed number of services)

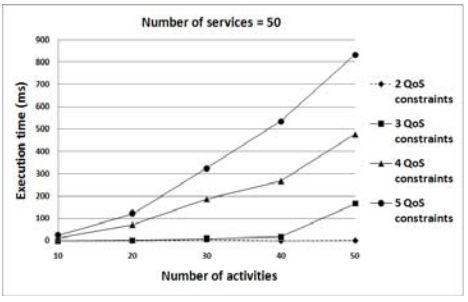

Fig. 6. Execution time of the global selection phase (for a fixed number of services)

time (i.e., less than 0.9s) if we compare it, e.g., to the response time of the email validation Web services described in Table 2.

Concerning the optimality of our algorithm, we measure it while fixing the number of QoS constraints to 5, and varying the number of activities and the number of services per activity between 10 and 50. Figure 7 shows that the optimality of our algorithm increases along with the number of activities and the number of services per activity. This means that, when it deals a large number of compositions, our algorithm finds more feasible compositions that may provide a better utility. This is explained by the fact that the utility of the best composition increases along with the probability to find services with QoS values close to the optimal QoS (i.e., near-optimal QoS values). As the service candidates are randomly generated, this probability increases along with the number of generated services and also with the number of activities, thus increasing the utility of the overall composition.

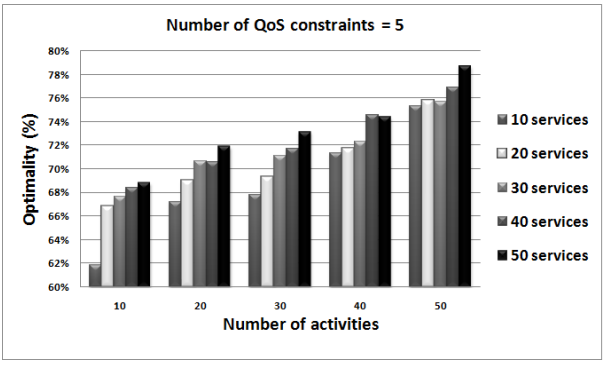

Fig. 7. Optimality of our algorithm

In general, our algorithm produces a satisfying optimality (i.e., more than 62%). However, this metric can be further enhanced by tuning the utility threshold $\mathcal{T}$ with respect to the trad-off between the desired optimality and the timeliness of the algorithm.

6 Conclusion

The objective of this work has been to address services' selection and composition in the context of a QoS-aware middleware for dynamic service environments. For this purpose, we have proposed an efficient QoS-based selection algorithm. The importance of our algorithm is three-fold. First, it introduces a novel approach based on clustering techniques. Applying such techniques for services' selection brings new ideas in this research area. Second, by producing not a single but multiple service compositions satisfying the QoS constraints, our algorithm underpins the concept of dynamic binding of services, which allows for coping with changing conditions in dynamic environements. Third and most importantly, our algorithm shows a satisfying efficiency in terms of timeliness and optimality, which makes it appropriate for on-the-fly service composition in dynamic service environments.

The presented work makes part of our ongoing research addressing QoS-aware middleware for pervasive environments. Our next steps concern further inverstigating clustering techniques for improving our heuristic algorithm, and considering in our QoS model network-level QoS and middleware-based QoS enhancement for service compositions.

Acknowledgement

This research is partially supported by the SemEUsE project[2] funded by the french National Research Agency (ANR).

References

1. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. Computer 40(11), 38–45 (2007)
2. Pautasso, C., Alonso, G.: Flexible Binding for Reusable Composition of Web Services. In: Gschwind, T., Aßmann, U., Nierstrasz, O. (eds.) SC 2005. LNCS, vol. 3628, pp. 151–166. Springer, Heidelberg (2005)
3. Di Penta, M., Esposito, R., Villani, M.L., Codato, R., Colombo, M., Di Nitto, E.: WS Binder: a framework to enable dynamic binding of composite web services. In: SOSE 2006: Proceedings of the 2006 international workshop on Service-oriented software engineering, pp. 74–80. ACM, New York (2006)
4. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. IEEE Trans. Softw. Eng. 30(5), 311–327 (2004)

[2] SemEUsE project: http://www.semeuse.org

5. Yu, T., Zhang, Y., Lin, K.-J.: Efficient Algorithms for Web Services Selection with End-to-End QoS Constraints. ACM Trans. Web 1(1), 6 (2007)
6. Jaeger, M.C., Mühl, G.: QoS-based Selection of Services: The Implementation of a Genetic Algorithm. In: Braun, T., Carle, G., Stiller, B. (eds.) Kommunikation in Verteilten Systemen (KiVS 2007) Industriebeträge, Kurzbeiträge und Workshops, Bern, Switzerland, March 2007, pp. 350–359. VDE Verlag, Berlin und Offenbach (2007)
7. Kobti, Z., Zhiyang, W.: An Adaptive Approach for QoS-Aware Web Service Composition Using Cultural Algorithms. In: Orgun, M.A., Thornton, J. (eds.) AI 2007. LNCS (LNAI), vol. 4830, pp. 140–149. Springer, Heidelberg (2007)
8. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: An approach for qos-aware service composition based on genetic algorithms. In: GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, pp. 1069–1075. ACM, New York (2005)
9. Zhang, C., Su, S., Chen, J.: A Novel Genetic Algorithm for QoS-Aware Web Services Selection. In: Lee, J., Shim, J., Lee, S.-g., Bussler, C.J., Shim, S. (eds.) DEECS 2006. LNCS, vol. 4055, pp. 224–235. Springer, Heidelberg (2006)
10. Cao, L., Li, M., Cao, J.: Using genetic algorithm to implement cost-driven web service selection. Multiagent Grid Syst. 3(1), 9–17 (2007)
11. Gao, C., Cai, M., Chen, H.: QoS-aware Service Composition Based on Tree-Coded Genetic Algorithm. In: COMPSAC 2007: Proceedings of the 31st Annual International Computer Software and Applications Conference, Washington, DC, USA, pp. 361–367. IEEE Computer Society, Los Alamitos (2007)
12. Vanrompay, Y., Rigole, P., Berbers, Y.: Genetic algorithm-based optimization of service composition and deployment. In: SIPE 2008: Proceedings of the 3rd international workshop on Services integration in pervasive environments, pp. 13–18. ACM, New York (2008)
13. Alrifai, M., Risse, T., Dolog, P., Nejdl, W.: A Scalable Approach for QoS-based Web Service Selection. In: 1st International Workshop on Quality-of-Service Concerns in Service Oriented Architectures (QoSCSOA 2008) in conjunction with ICSOC 2008, Sydney (December 2008)
14. Mokhtar, S.B., Kaul, A., Georgantas, N., Issarny, V.: Efficient semantic service discovery in pervasive computing environments. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 240–259. Springer, Heidelberg (2006)
15. Mokhtar, S.B., Preuveneers, D., Georgantas, N., Issarny, V., Berbers, Y.: EASY: Efficient semAntic Service discoverY in pervasive computing environments with QoS and context support. J. Syst. Softw. 81(5), 785–808 (2008)
16. Lloyd, S.P.: Least squares quantization in PCM. Unpublished memorandum, Bell Laboratories (1957)
17. Mabrouk, N.B., Georgantas, N., Issarny, V.: A Semantic End-to-End QoS Model for Dynamic Service Oriented Environments. In: Principles of Engineering Service Oriented Systems (PESOS 2009), held in conjunction with the International Conference on Software Engineering, ICSE 2009 (2009)
18. Moscato, F., Mazzocca, N., Vittorini, V., Di Lorenzo, G., Mosca, P., Magaldi, M.: Workflow Pattern Analysis in Web Services Orchestration: The BPEL4WS Example. In: Yang, L.T., Rana, O.F., Di Martino, B., Dongarra, J. (eds.) HPCC 2005. LNCS, vol. 3726, pp. 395–400. Springer, Heidelberg (2005)

19. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Pattern Based Analysis of BPEL4WS. In: Song, I.-Y., Liddle, S.W., Ling, T.-W., Scheuermann, P. (eds.) ER 2003. LNCS, vol. 2813, pp. 200–215. Springer, Heidelberg (2003)
20. Arthur, D., Vassilvitskii, S.: On the Worst Case Complexity of the k-means Method. Technical Report 2005-34, Stanford InfoLab (2005)
21. Al-Masri, E., Mahmoud, Q.H.: QoS-based Discovery and Ranking of Web Services, August 2007, pp. 529–534 (2007)
22. Al-Masri, E., Mahmoud, Q.H.: Discovering the Best Web Service. In: WWW 2007: Proceedings of the 16th international conference on World Wide Web, pp. 1257–1258. ACM, New York (2007)
23. Hogben, L., Greenbaum, A., Brualdi, R., Mathias, R.: Handbook of Linear Algebra. Chapman & Hall, Boca Raton (2007)