A Lightweight Approach for QoS–Aware Service Composition

Gerardo Canfora, Massimiliano Di Penta
RCOST - Research Centre on Software Technology
University of Sannio, Department of Engineering
Palazzo ex Poste, Via Traiano
82100 Benevento, Italy
canfora@unisannio.it, dipenta@unisannio.it

Raffaele Esposito, Maria Luisa Villani
RCOST - Research Centre on Software Technology
University of Sannio, Department of Engineering
Palazzo ex Poste, Via Traiano
82100 Benevento, Italy
r.esposito@unisannio.it, villani@unisannio.it

ABSTRACT

One of the most challenging issues of service–centric software engineering is the QoS–aware composition of services. The aim is to search for the optimal set of services that, composed to create a new service, result in the best QoS, under the user or service designer constraints. During service execution, re-planning such a composition may be needed whenever deviations from the QoS estimates occur.

Both QoS–aware composition and re-planning may need to be performed in a short time, especially for interactive or real–time systems. This paper proposes a lightweight approach for QoS–aware service composition that uses genetic algorithms for the optimal QoS estimation. Also, the paper presents an algorithm for early triggering service re-planning. If required re-planning is triggered as soon as possible during service execution. The performances of our approach are evaluated by means of numerical simulation.

1. INTRODUCTION

Web services constitute a promising technology landscape for software engineering. During the last 20 years, component based software engineering aimed at the application of principles used for years in other engineering disciplines to software development activities. In fact, electronics, mechanics and other engineering systems are commonly built by assembling pre-defined components, such as memories, CPUs, etc.

In the same way, the spread of reusable software components avoids software engineers to "re-invent the wheel" each time. Building software systems by "gluing" components enables them to concentrate more on the problems their systems aims to solve. This scenario changes when moving from component-based systems to service-centric systems, where a functionality is realized by searching, composing and executing services. In the particular case of web services, this is done by using a set of XML-based standards, known as UDDI, WSDL and SOAP [16].

The service-centric systems scenario poses several additional challenges with respect to component-based software engineering. First and foremost, in a component-based software system components are physically integrated and, except for distributed systems, they are executed as a whole in the end-user's environment. This is not usually the case of web services as they are executed on the service provider server, thus raising issues on the run-time service (and of course network) availability and performances.

Secondly, several services may be available with the same function (we call them *semantically equivalent* services), however they surely exhibit different Quality of Service (QoS). According to Std. ISO 8402 [9] and ITU [10], QoS may be defined in terms of attributes such as price, response time, availability, reputation (further details can be found in Cardoso's PhD thesis [2]). Moreover, it may be possible to have some domain-specific QoS attributes (e.g., a temperature service could have QoS attributes such as precision or refresh frequency). The choice between different but semantically equivalent services is a function of such QoS attributes: one may decide to choose the cheapest service, the fastest, or maybe a compromise between the two. Moreover, an user may specify constraints on the values of some attributes (e.g., the price cannot be higher than a given value), which could influence the choice. On the other hand, the service provider can estimate ranges for the QoS attribute values as part of the contract with potential users. Also, the QoS guarantees for the same service could be customer-dependent, and so they would apply each to a different instance of that service. For example, an user that buys a service at a given price is not expected to get a response time below a given threshold.

Once service annotation and matching mechanisms are available, a semantic description of the service may be included in the user system as a reference to it. Such a description, that we call an *abstract service*, corresponds to a specific functionality, while not necessarily to a single service implementation. At run-time, using a matching algorithm, this description is used to retrieve some services (that we will refer to as *concrete services*), and then select one

among them that meets the constraints and maximizes our QoS objective function. Several matching approaches have been proposed in literature, see for example Paolucci et al. [13].

A composite service is a service resulting from a composition of other services whose interaction is described by some workflow description language (e.g., BPEL4WS [1]). The component services can be, on their own, *abstract services*, therefore two different problems need to be solved:

- determine the QoS of a composite service as a function of the QoS of its components; and
- determine the set of *concrete services* that maximize the QoS of the composite service. In other words, for each *abstract service* of the workflow, determine a concretization such that the total QoS is maximized and the global constraints are met.

Clearly, the QoS values of the single service components used for computation may be estimations in turn, declared by each service provider or obtained by computing statistics during previous executions of the service. At run-time, the actual (measured) QoS values may deviate from the estimate ones or, simply, one of the services may not be available. Thus the composite service may have to be re-planned, so to still meet the constraints and maximize the QoS. Some approaches have been proposed in literature [17] to this aim.

All of this, from the QoS-aware composition to re-planning, often needs to be performed very quickly. Especially for interactive systems, long delays may be unacceptable. For example, the user of a booking ticket system might not want to wait for a long time while the system searches for candidate services offering flight tickets with the lowest booking fare. Gaining a few cents after several minutes of waiting may make the user disappointed. Even for some non-interactive service a fast composition just before execution may be desired: performing service composition long before execution may lead to unattended results (e.g., some services may not be available anymore or, conversely, new, more convenient services can be available).

The paper aims are the following:

1. it proposes an approach for a quick, coarse–grained QoS-aware service composition, where some of the composition rules are the same as those proposed by Cardoso [2];
2. when the QoS optimality is more relevant than the performance of the service composition algorithm, like for scientific computations, an alternative approach is proposed;
3. finally, the paper proposes a re-planning algorithm aiming to anticipate the re-planning decisions as soon as possible during the composite service execution. Re-planning is then performed on a slice of the original workflow, avoiding to unfold loops that could worsen the performances.

Numerical simulations have been used to evaluate the results.

The remainder of this paper is organized as follows. After a review of the literature in Section 2, Section 3 details the approach we propose. In particular, the QoS composition rules are shown, the evolutionary approach for QoS-aware composition is described and, finally, the re-planning algorithm is detailed. Section 4 describes the toolkit aiming to support the work, while Section 5 reports and discusses results obtained in the simulations. Finally, Section 6 concludes.

2. RELATED WORK

QoS-aware discovery and composition of services has been recognized as a crucial aspect in the web services era, where companies are starting to deliver their products as services over the Internet and the service-oriented architecture paradigm has become a new reference for the software business. In this context, providers need ways to express their quality guarantees on the service being advertised, and technological support should be given to customers to search for and select the best available service. Furthermore, satisfaction of the quality requirements he/she specified should be assured during execution. This is more difficult for composite services, where the overall QoS relies on that of each component service, and it depends on how services are integrated and interact with each other. General issues for Web Services QoS are discussed in a paper by Ludwig [11].

Some formalism for the service QoS specification and Service Level Agreements (SLAs) has been provided, such as the IBM's Web Service Level Agreement (WSLA) language [12], or the Web Services Offer Language (WSOL) [15]. On the other hand, some web service orchestration languages are being proposed as standard, like Business Process Execution Language for web services (BPEL-4WS) [1]. Nevertheless, the currently available workflow technology still lacks of facilities for a complete QoS estimation, management and monitoring for processes. In fact, most established solutions in the area of workflow focus only on time management or load balancing and web-service-based systems that account for other QoS criteria are still being experimented ([3], [14], [17]). In particular, some on-going research activities deal with QoS prediction and dynamic adaptation of the workflow to face unexpected QoS progress during execution. In this respect, our work is positioned within that of Cardoso [2], and that of Zeng et al. [17].

The former proposes a mathematical model for workflow QoS computation, described by some metrics aggregation functions which are defined for time, cost, reliability, and fidelity. The meaning of each metrics is precisely given. The model uses stochastic information indicating the probability of transitions being fired at run-time, which can be initially set by the designer, wherever possible, and then periodically adjusted, based on data on previous executions stored in the workflow system log. The same method is also used to re-compute tasks' QoS. The QoS computation algorithm (SWR) proposed by Cardoso consists of applying a set of reduction rules to the workflow until one atomic task is obtained. We also use a reduction approach for our workflow QoS estimation, because of the advantage of fast computation, and mostly consider the same aggregation functions for each metrics, some of which are reported in Table 1. However, Cardoso's method does not consider optimal binding of the service components nor re-negotiation at run-time. In particular, the loop reduction rule does not seem to be suitable for re-planning. In fact, this rule is based on a probability for the feedback transition to fire, thus it does not allow distinguishing the workflow part to be re-planned during execution, in case of errors happening within the loop. We propose a different reduction rule for the loop construct which enables dynamic service binding and re-negotiation.

The work of Zeng et al. focuses on dynamic and quality-driven

selection of the service implementations of the workflow that accounts for local and global quality constraints and user preferences. Most notably, they propose a global planning approach to reach overall QoS optimality through linear programming techniques. The method is discussed, and empirical data given, under the assumptions of the workflow being acyclic, the constraints and objective function being linear, and the workflow consisting of one execution path, for which the QoS estimate is made. Then the method could be generalized by unfolding loops, based on the estimated number of iterations, binding a concrete service to each task by considering the "hot path" for that task, i.e. the most frequently executed path containing that task, while the constraint on linearity is overcome by applying logarithmic transformation functions. While this approach is quite effective with respect to reaching QoS optimality, in case of a complex workflow with branches and frequent loop iterations the concretization process they propose seems to be less efficient. In this paper, we discuss the trade-off between efficiency of the concretization algoritm and optimality of the solution obtained. Also, we extend their method for workflow re-planning.

Web-Flow [14] and eFlow [5] are workflow management systems that offer some support to selection of services according to quality constraints. However, these constraints are only applied at task level. In particular, eFlow allows to bind a service implementation to a *generic node* at run-time through a *search recipe*, while Web-Flow includes an exception handling mechanism, based on the Event-Condition-Action paradigm, triggered by the violation of constraints or other events like service faults, that may occur during process execution. A model for exception analysis, prediction and prevention in business processes, based on data warehousing and mining techniques, was presented by Casati et al. [4]. The exceptional events are stated by the user and are defined by conditions over process execution data. No dynamic recovery is addressed.

Finally, solutions for resource allocation and performance management that can be borrowed from the distributed systems area have been suggested [11], along with an analysis on web services-specific issues.

3. APPROACH DESCRIPTION

As stated in the introduction, the proposed approach is mainly devoted to allow for a fast, overall computation of the QoS of a composite service, as well as to determine the optimal set of *concrete services* to be bound to the *abstract services* composing the workflow. The model also enables re-planning.

In the sequel we shall consider a composite service S of n *abstract services*, $S \equiv \{s_1, s_2, \ldots, s_n\}$, whose structure is defined through some workflow description language. Each component s_i can be bound to one of the m *concrete services* $cs_{i,1}, \ldots, cs_{i,m}$, which are functionally equivalent.

3.1 Computing the QoS of Composite Services

This section describes our approach, hereby referred as the *unlooping* approach, for computing the QoS of composite service. Similarly to what proposed by Cardoso [2], for a Switch construct in the workflow, each `case` statement is annotated with the probability to be chosen (see Figure 1-a). For example, for a workflow containing a Switch composed of two Cases, with costs C_1 and C_2 respectively and probabilities p and $1-p$, the overall cost is computed as follows:

$$p\,C_1 \;+\; (1-p)\,C_2 \qquad (1)$$

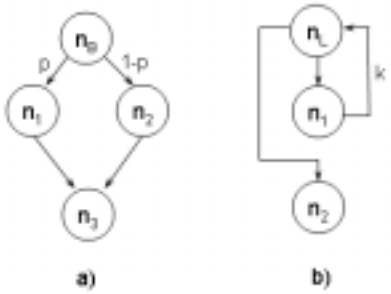

Figure 1: Workflow annotation: a) Switch b) Loop

Clearly, probabilities are initialized by the workflow designer, and then eventually updated considering the information obtained by monitoring the workflow executions.

Loops are handled differently from Cardoso [2], that basically proposes to adopt a mechanism (based on the probabilities of entering/exiting the Loop) as for the `switch` construct. Our approach is more similar to what proposed by Zeng et al [17], i.e., Loops are annotated with an estimated number of iterations k. Instead of unfolding Loops (like Zeng et al.), here the QoS of the Loop is computed taking into account the factor k (see Figure 1-b) . For example, if the Loop compound has a cost C_l, then the estimated cost of the Loop will be $k\,C_l$.

This approach for handling Loops presents two advantages:

- It allows for a quick computation of the overall workflow QoS, without the need to unfold Loops;
- The estimated QoS accounts for the estimated number of Loop iterations.

Given a *concretization* of a composite service, i.e., a composite service description where each *abstract service* has been bound to one of its corresponding *concrete services*, the overall QoS can be computed by applying the rules described in Table 1, which shows an aggregation function for each pair workflow construct and QoS attribute. While for some standard QoS attributes the aggregation function has been explicitly specified ([17], [2]) there may be other attributes (for example, domain-dependent attributes) for which the aggregation function is user–specified (see the last row of Table 1).

It should be noted that the table is not complete (it only contains rules to be used in our examples) and, except that for Loops, the aggregation functions correspond to those proposed by Cardoso [2]. These functions are recursively defined on compound nodes of the workflow, although the table only shows their definition on sets of elementary tasks. Namely, for a Sequence construct of tasks $\{t_1, \ldots, t_m\}$, the *Time* and *Cost* functions are additive while *Availability* and *Reliability* are multiplicative. The Switch construct of Cases $1, \ldots, n$, with probabilities $p_{a1}, \ldots, p_{an}$ such that $\sum_{i=1}^{n} p_{ai} = 1$, and tasks $\{t_1, \ldots, t_n\}$ respectively, is always evaluated as a sum of the attribute value of each task, times the probability of the Case to which it belongs. The aggregation functions for the Fork construct, are essentially the same as those for the Sequence construct,

QoS Attr.	Sequence	Switch	Fork	Loop
Time (T)	$\sum_{i=1}^{m} T(t_i)$	$\sum_{i=1}^{n} p_{ai} * T(t_i)$	$Max\{T(t_i)_{i\in\{1...p\}}\}$	$k * T(t)$
Cost (C)	$\sum_{i=1}^{m} C(t_i)$	$\sum_{i=1}^{n} p_{ai} * C(t_i)$	$\sum_{i=1}^{p} C(t_i)$	$k * C(t)$
Availability (A)	$\prod_{i=1}^{m} A(t_i)$	$\sum_{i=1}^{n} p_{ai} * A(t_i)$	$\prod_{i=1}^{p} A(t_i)$	$A(t)^k$
Reliability (R)	$\prod_{i=1}^{m} R(t_i)$	$\sum_{i=1}^{n} p_{ai} * R(t_i)$	$\prod_{i=1}^{p} R(t_i)$	$R(t)^k$
Custom Attr. (F)	$f_S(F(t_i)_{i\in\{1...m\}})$	$f_B((p_{ai}, F(t_i))_{i\in\{1...n\}})$	$f_F(F(t_i)_{i\in\{1...p\}})$	$f_L(k, F(t))$

Table 1: Aggregation functions per workflow construct and QoS attribute

except for the *Time* attribute where this is the maximum time of the parallel tasks $\{t_1, \ldots, t_p\}$. Finally, a Loop construct with k iterations of task t is equivalent to a Sequence construct of k copies of t.

3.2 Searching for a solution with Genetic Algorithms

Determining the best concretization of a composite service is an optimization problem:

1. Maximize a fitness function of the available QoS attributes; and
2. Meet the constraints specified for some of the attributes. In particular, these are the global constraints, i.e. assertions on the overall QoS attribute values. Local constraints, i.e. constraints on each service s_i composing our service, need to be checked when choosing the set of candidate *concrete services* for s_i.

Finding a solution for the above problem is NP-hard [7]. In this case, different strategies can be adopted, for example integer programming [17] or Genetic Algorithms (GA). In our work we chose to adopt GA because the problem, as stated in our case, is well suited to be encoded with a genome and solved using GA evolution. Differently to linear programming approaches, GA does not impose constraints on the linearity of the QoS composition operators (and thus of objective function and constraints). This permits to adopt our approach for all possible (even customized) QoS attributes, without the need for linearization.

GA originated with an idea, born over 30 years ago, of applying the biological principle of evolution to artificial systems. Roughly speaking, a GA is an iterative procedure that searches for the best solution of a given problem among a constant-size population, represented by a finite string of symbols, named the *genome*. The search is made starting from an initial population of individuals, often randomly generated. At each evolutionary step, individuals are evaluated using a *fitness function*. High–fitness individuals will have the highest probability to reproduce.

The evolution (i.e., the generation of a new population) is made by means of two operators: the *crossover operator* and the *mutation operator*. The crossover operator takes two individuals (the *parents*) of the old generation and exchanges parts of their genomes, producing one or more new individuals (the *offspring*). The mutation operator has been introduced to prevent convergence to local optima, in that it randomly modifies an individual's genome (e.g., by flipping some of its bits, if the genome is represented by a bit string). Crossover and mutation are performed on each individual of the population with probability *pcross* and *pmut* respectively, where $pmut \ll pcross$. Further details on GA can be found, for example, in the Goldberg's book [8].

To let the GA search for a solution of our problem, we first need to encode the problem with a suitable genome. In our case, the genome is represented by an integer array with a number of items equals to the number of distinct *abstract services* composing our service. Each item, in turn, contains an index to the array of the *concrete services* matching that *abstract service*. Figure 2 gives a better idea of how the genome is made.

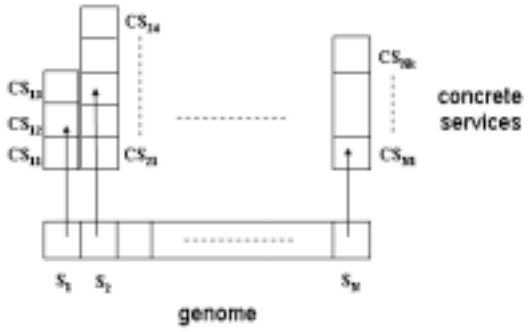

Figure 2: Genome Encoding

The crossover operator is the standard two-points crossover [8], while the mutation operator randomly selects an *abstract service* (i.e., a position in the genome) and randomly replaces the corresponding *concrete service* with another one among those available. Clearly, *abstract services* for which only one *concrete service* is available are taken out from the GA evolution.

The problem can now be modeled by means of a fitness function and, eventually, some constraints. The fitness function needs to maximize some QoS attributes (e.g., reliability), while minimizing others (e.g., cost). When user–defined, domain–specific QoS attributes are used, the specification of the fitness function is left to the workflow designer. For standard QoS attributes, we define a fitness function for a genome g as follows:

$$F(g) = \frac{w_1\ Availability\ +\ w_2\ Reliability}{w_3\ Cost\ +\ w_4\ Response\,Time} \qquad (2)$$

where $w_1, \ldots, w_4$ are real, positive weighting factors. As shown, our fitness function is multi-objective. Different approaches have been proposed in literature to deal with this kind of fitness function [6]. Generally, calibrating weights is guided by observing the fitness function's landscape, as well as from the analysis of the evolution of the different factors.

As mentioned above, constraints are assertions on the overall values of QoS attributes, e.g.:

$$\begin{aligned} Cost &< 50 \\ Time &< 100 \\ \frac{1}{2}\, Availability + \frac{1}{2} Reliability &> 0.95 \end{aligned}$$

When starting and evolving the GA, constraints need to be met anyway. This requires the genome initialization, the crossover operator and the mutator operator be modified. In fact, any randomly–generated genome that constitutes an individual of the starting population must be rejected (and thus generated again) if its QoS violates the constraints. Similarly, crossover and mutation operations must be roll-backed when, respectively, the offspring or the mutated individual violate the constraints.

3.3 Unfolding Approach: when better QoS is more essential than Quickness

The described approach permits to quickly find a solution to the QoS optimization problem. However, when QoS attributes are constrained, there could be better solutions at the price of a more expensive search.

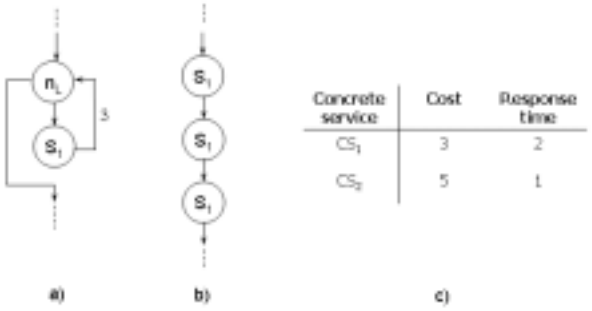

Figure 3: When unfolding may be convenient

Let us consider, for example, a simple workflow shown in Figure 3-a), where the Loop is estimated to be executed 3 times. Suppose now that our optimization problem is given by:

$$\begin{aligned} F(x) = 1/(Cost + Time) \\ Time \leq 5 \end{aligned}$$

The GA described in Section 3.2 suggests that the *concrete service* CS_2 binds the *abstract service* S_1; the latter does not maximize the fitness function (that is equal to 1/18, while it would be 1/15 if CS_1 is chosen), although the constraints are met.

Let us suppose now that we unfold the Loops in our workflow, obtaining a workflow depicted in Figure 3-b. Let us consider a genome composed of all the nodes of the unfolded workflow, as represented in Figure 3-c. This encoding permits to have different bindings for different invocations of a service component, that can be useful to obtain a better QoS while meeting the constraints. In our example, a binding $\{CS_1, CS_2, CS_1\}$ will ensure the constraint to be met, and also produces a fitness value of $1/16 > 1/18$.

Clearly, as it will be shown in Section 5.1, Loop unfolding could mean an explosion of the genome size, and consequently of the time the GA requires to converge. Therefore, a tradeoff should be pursued. Interactive applications may accept a weaker QoS in favor of a short service negotiation time. Besides, for long–time–run scientific computations a long service negotiation time can be acceptable if, for example, we are composing a workflow that implements an algorithm to be run over weeks of computation time. In that case, the choice of the best combination of services, although requiring a longer time (e.g., some hours), could save days of computation time.

3.4 Triggering Service Re-planning

During workflow execution, the actual QoS may deviate from the estimated one, according to formulae shown in Table 1. Furthermore, there could be services not more available when invoked. In the two situations above, the slice of the workflow still to be executed may need to be re-planned.

The algorithm presented in Figure 5 describes the proposed re-planning triggering approach. Actually, this algorithm needs to be integrated with the workflow engine to allow measuring the actual QoS during execution and to perform re-planning when needed. The algorithm is described for any additive QoS attribute (e.g., cost) however it is still valid (with proper changes in the QoS formulae) for other (e.g., multiplicative) attributes. Also, for simplicity, the algorithm shows how re-planning works for constructs such as Loop, Switch, Sequence and invocation, while it can be easily extended to other constructs (e.g. Fork).

Given the overall estimated QoS (Q_{EST}), initially the actual workflow QoS (Q_{ACT}) is equal to it. Then, the workflow execution starts visiting the root node, and each node is recursively visited. Each time the absolute difference between the actual QoS and the estimated QoS is above the fixed threshold NTH, a re-planning is triggered.

For Loop nodes, the actual number of iterations k' is determined if possible (when the Loop exit is bound to a condition, this might not be possible), and the actual QoS is refined varying it by $(k' - k) * Q_{INNER}$ (i.e., considering that the number of iterations is varied by $k' - k$). In case this difference evaluates above the threshold, a re-planning is triggered. Then, the Loop inner node is visited k' times (or while the Loop condition is *true*), triggering re-planning each time this is necessary.

For Switch nodes, the actual Case to be executed (the $j-th$ one) is determined, and the Switch inner QoS (originally a weighted sum, as shown in equation (1)) is updated, considering, instead, only the QoS of the Case chosen.

For Sequence nodes, each child is visited, and re-planning is triggered each time the deviation of the actual QoS from the estimate is above the threshold. Finally, for service invocation nodes the

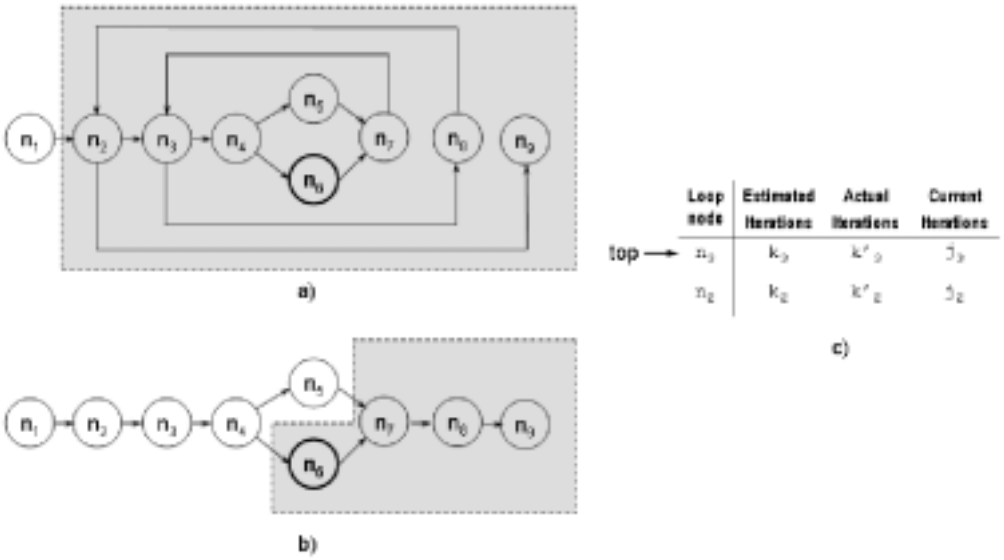

Figure 4: Re-planning slice a) Node outside any Loop or Sequence b) Node inside a not yet estimated to be completed Loop c) Node inside completed Loops

measured QoS after service invocation is used to update the actual QoS.

3.5 Determining the re-planning slice

The function *triggerReplan()* invoked in the algorithm of Figure 5 acts on a slice of the original workflow, representing the set of nodes that still are to be executed.

Given a node n_p that, after its execution, triggers a re-planning, the slice to be re-planned is computed depending on the position of the node n_p in the workflow control structure.

- If the node is outside any Loop or Switch statement, all the nodes that follow n_p (including the node n_p itself) are part of the slice (Figure 4-a);
- If the node is part of a Switch, while outside any Loop, the slice will include again all the nodes that follow n_p, excluding the nodes of the alternative Cases (Figure 4-b);
- Whenever the workflow execution enters a Loop i, the following information is pushed on a stack: a reference to the Loop node, the estimated number of iterations k_i, the re-estimation k'_i (if available, otherwise $k'_i = k_i$), and the actual number of executed iterations j_i. When a re-planning is triggered inside a Loop, the slice is obtained starting from the most external, not yet *estimated to be completed*, Loop. A Loop i is estimated to be completed *iff* $j_i \geq k'_i$. Clearly, the resulting slice will contain, for such most external Loop, an estimated number of iterations equals to $k'_i - j_i$, i.e., only accounting for the iterations left to be executed.

Given the re-planning slice, the same approach described in Section 3.2 is used to find its (sub)-optimal concretization. However, this time the overall QoS that maximizes the fitness function while meeting the constraints is given by:

$$Q_{OVERALL} = Q_{TOT} + QoS(slice) \quad (3)$$

i.e., the QoS of already executed nodes, plus the estimated QoS of the slice.

4. TOOL DESCRIPTION

We have implemented a tool prototype for a QoS-aware composition of services that we used for the experimentation of the presented approaches. Some of the components we realized could be integrated with a real web services orchestrator, while for the purpose of our experiments it was enough implementing a quite simple workflow engine simulator. The architecture of the tool is composed of:

- a *Service Repository*, where concrete services are stored along with their QoS information, and classified according to their semantic descriptions;
- a *Workflow Generation Tool*, used to automatically produce XML workflow representations of composite services. The workflow tasks are abstract services, while BPEL4WS-like constructs are used for the control flow specification. Also, it is possible to attach QoS constraints to workflow tasks;
- a *Concrete Workflow Builder*, for retrieving concrete services for the workflow tasks from the Service Repository, eventually according to local QoS constraints;
- a *Workflow QoS Estimator*, used for the overall QoS estimation of a concrete workflow;
- a *Workflow QoS Optimizer*, to find the optimal set of concrete services for the workflow, with respect to the overall QoS, that maximizes a specified objective function and meets global QoS contraints. This component uses the *Concrete*

```
Q_EST ←Estimated overall QoS;
Q_ACT ← Q_EST;
Q_TOT ← 0 ;
NTH ←replanning threshold;
NODE ← Workflow root node;
visit(NODE);
exit();
begin function visit(node)
    switch node is type of do
        case loop
            k ← Estimated loop iterations ;
            k' Actual # of loop iterations ;
            INNER ← Loop inner node
            Q_INNER ← QoS(INNER);
            Q_ACT ←
            Q_ACT + (k' − k) * Q_INNER ;
            if |Q_ACT − Q_EST| > NTH then
                triggerReplan(INNER,
                Q_TOT) ;
            end
            for j ← 1 to k' do
                visit(INNER);
                if |Q_ACT − Q_EST| > NTH
                then
                    Q_TOT ← Q_TOT −
                    ActQoS(INNER) ;
                    triggerReplan(INNER,
                    Q_TOT);
                end
            end
        case switch
            j ← Case statement chosen ;
            INNER ← Inner node of the
            j − th case ;
            Q_SWITCH ← QoS(node);
            Q_INNER ← QoS(INNER);
            Q_ACT ←
            Q_ACT − Q_SWITCH + Q_INNER;
            if |Q_ACT − Q_EST| > NTH then
                triggerReplan(INNER,
                Q_TOT)
            end
            visit(INNER);
        case sequence
            foreach Node INNER in
            sequence do
                visit(n);
                if |Q_ACT − Q_EST| > NTH
                then
                    Q_TOT ← Q_TOT −
                    ActQoS(INNER) ;
                    triggerReplan(INNER,
                    Q_TOT);
                end
            end
        case invocation
            INNER ← node
            Q_INNER ← QoS(node);
            Execute(node);
            Q_INNERACT ← ActQoS(node);
            Q_ACT ←
            Q_ACT − Q_INNER + Q_INNERACT;
            Q_TOT ← Q_TOT + Q_INNERACT ;
        if |Q_ACT − Q_EST| > NTH then
            triggerReplan(INNER, Q_TOT)
        end
    end
end function
```

Figure 5: Re-planning triggering algorithm

Workflow Builder to obtain the concrete workflows needed for the evaluation of the objective function through the *Workflow QoS Estimator*, and an *Optimization Library* to solve the global optimization problem;

- a *Workflow Simulator*, for the service composition simulation. This component can be used to simulate the execution of a concrete workflow, to analyze the actual path followed and the actual local and global QoS values. When performing a simulation, we considered: i) the actual response times of the component services and the actual number of loop iterations, varying them according to a gaussian distribution function centered in the estimated value, and with a specified standard deviation; ii) the probability of choosing a case statement inside a switch construct, and the estimated availability of a service; iii) the values of the cost attributes of the component services are all constant, since it is unlikely (even if possible) that these could change at run–time. To avoid bias due to the result randomness, simulations are performed for a high number of times (say 1000), and average values of the actual QoS are considered. The *Workflow Simulator* has also been extended with the implementation of our Re-planning Triggering algorithm, and a *QoS Monitoring Tool*. Whenever a re-planning trigger occurs during execution, the slice of the workflow to be re-planned is computed and the *Workflow QoS Optimizer* is invoked to do the re-planning work on that slice. The *QoS Monitoring Tool* keeps track of the past workflow executions for each customer. The workflow log data is then used to refine the estimations on the loop iterations and to update the probabilities on case statements.

5. EMPIRICAL STUDY

The evaluation of the workflow QoS estimation and re-planning approaches has been performed through numerical simulation. The experiments were run on a *Compaq Proliant*^{TM} with Dual Xeon^{TM} 900 MHz processor, 2MB Cache and 4GB of RAM. To this aim, we used a simplified representation of the services, including a name, a reference to its semantic description and estimated values for cost, response time and availability attributes. The QoS values of semantically equivalent services were varying according to some gaussian distribution function, and better response time and availability offers corresponded to higher costs. Also, workflows of different sizes were generated with random probabilities on Switch and Loop iteration estimations.

Some sets of experiments were set up to reason about unlooping vs unfolding for a given workflow, when the estimated number of Loop iterations increases, and the frequency of re-planning during execution of workflows, due to deviations of the actual overall QoS from the estimated one. These experiments and the resulting data are discussed in the following subsections.

The GA was set up with the following parameters:

- Elitist GA, i.e., the best two individuals were kept alive over subsequent generations;
- Population of 100 individuals;
- Crossover probability=0.7; and
- Mutation probability=0.1.

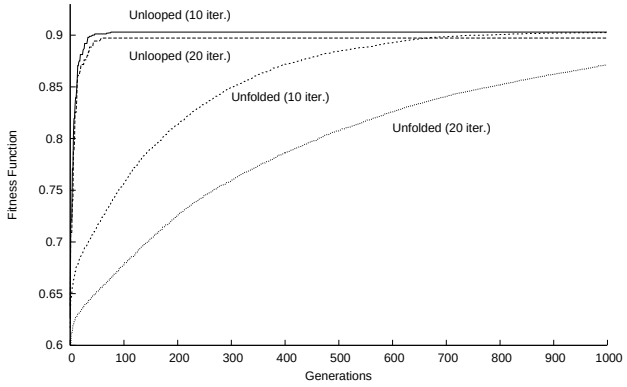

Figure 6: Unfolding vs. unlooping: fitness function convergence

To avoid biasing results because of randomness, the GA executions were repeated 10 times, and average values used.

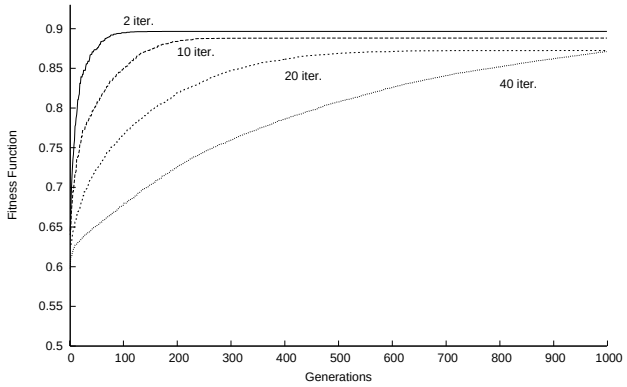

Figure 7: Unfolding rise time for different numbers of iterations

5.1 Unlooping vs. Unfolding

The first part of the experiments was concerned with testing both approaches for a benefits-costs analysis. We initially considered a simple workflow consisting of a Loop construct including 10 distinct abstract services, and repeated the experiments with a number of iterations ranged from 2 to 40, hence with 400 as a maximum number of service invocations. Also, for each abstract service, up to 15 concrete services were considered.

Figure 6 plots the fitness function evolution across GA generations, for worflows with 10 and 20 Loop iterations. It is important to note that without constraints, we get convergence to the same fitness value with both unfolding and unlooping approaches. However, the unlooping approach is able to ensure convergence to be reached much faster, after about 30-40 generations, regardless of the number of iterations considered. Vice–versa, as shown in Figure 8, the time rise of the unfolding case (i.e., the number of generations required for fitness convergence) depends on the number of iterations. Moreover, if we consider the GA execution performances [1] (see Figure 8), while those of the unfolding case are significantly higher (because of the largest genome) with an exponential grow at the increase of the number of iterations, the performances of the unlooping approach are lower and approximately constant with respect to the iterations.

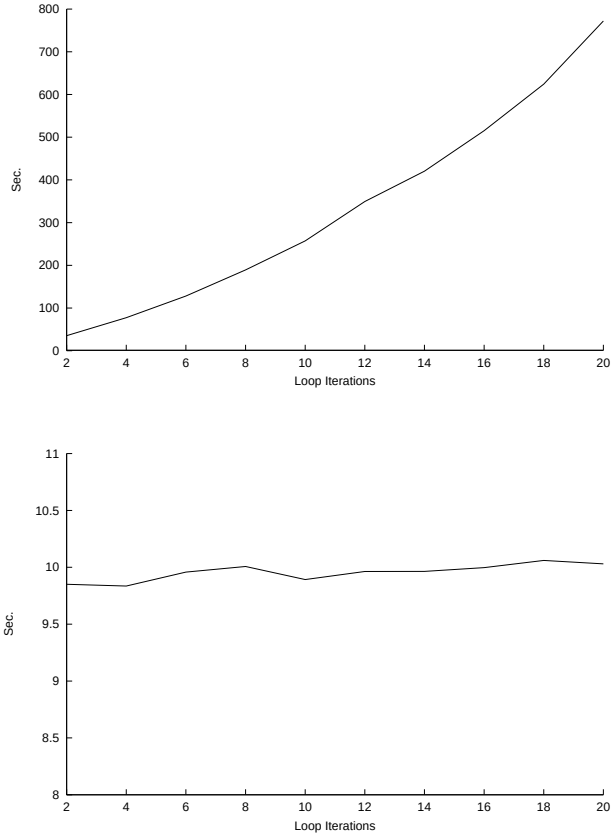

Figure 8: Unfolding vs. unlooping: timing comparison

5.2 Constraints

As described in Section 3, the unfolding approach may result useful in presence of global constraints. To support this hypothesis, we performed a second set of experiments, imposing constraints on the response time attribute (of type $overall\ response\ time \leq value$) , in order to evaluate the extent to which a loss of the overall quality with our unlooping approach, but with a time gain for finding an acceptable solution, is worth against absolute QoS optimality reached with the unfolding approach.

In particular, a workflow containing a Loop over 3 services was considered, with the number of iterations varying from 3 to 20. As shown in Table 2, the unfolding approach can lead to an increase of the fitness function value, although this is almost always limited by the maximum time to be taken. In our experiments, we found improvements of the fitness function values varying from 7 to 11%, while the additional time needed for convergence was up to 2000%. Noticeably, varying the constraint proportionally to the number of

[1] *user* CPU times computed with the Unix utility `time`.

# of Loop Iterations	Fitness (Unlooped)	Fitness (Unfolded)	Fitness Increase (%)	Time (Unlooped)	Time (Unfolded)	Time Increase (%)
3	0.71	0.79	10.78%	2.80	8.47	202.50%
5	0.71	0.77	8.98%	2.81	12.63	349.47%
10	0.69	0.74	7.18%	2.84	26.84	845.07%
15	0.68	0.73	7.44%	2.87	43.28	1408.01%
20	0.67	0.72	8.29%	2.91	62.97	2064.66%

Table 2: Using the unfolded workflow in presence of constraints: cost/benefits

iterations, we did not observe a corresponding increment in the difference between the fitness function values on the unfolded and unlooped workflows.

Thus, we believe that unfolding Loops may only be convenient when, in presence of constraints, even a slight improvement of the fitness value is worth hours taken to search for a solution. For example, we may desire to minimize the cost as much as possible, without trying to minimize the response time, on which, however, a constraint has been specified. Once the constraint is satisfied, the unfolding approach tends to choose the cheapest services rather than the fastest ones. On the contrary, the unlooping approach may require that each abstract service be bound to the fastest concrete service, since each abstract service is only bound once in the workflow. The tradeoff between the unfolding (better fitness) and unlooping (quick convergence to a sub-optinal solution) approaches is represented in Figure 9.

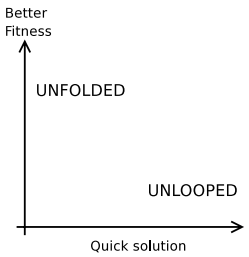

Figure 9: Tradeoff between unfolding and unlooping

5.3 Dealing with Uncertainty in QoS Estimation

The estimated overall QoS may deviate from the actual values obtained during service execution. The QoS of a Switch node is estimated, according to Table 1, as a weighted sum of the QoS values of the different cases. At run–time, only the QoS of the case actually followed is considered. Similarly, the actual number of Loop iterations can deviate from the estimated one. Finally, some QoS estimations of the invoked nodes can vary.

We used the *Workflow Simulator* to compute differences between estimated and actual QoS values. Simulations were performed varying from 10% to 50% the standard deviation on the estimated number of Loop iterations, and from 5% to 15% that on the QoS estimates (only for response time, since we considered cost values to be constant).

Figure 10 plots the error occurred on overall cost and response time estimates when the standard deviation on Loop iterations estimates varies from 10 to 50. In this case the standard deviation on the QoS estimate was kept constant to 5%. Errors on availability were always below 1%, thus giving high confidence on service availability even in the presence of big estimation errors. When varying the standard deviation for the response time estimate, the error measured for 5%, 10% and 15% were, respectively, of 3%, 5.5% and 8%. In this experiment, the standard deviation on Loop iterations estimates was kept fixed to 1%.

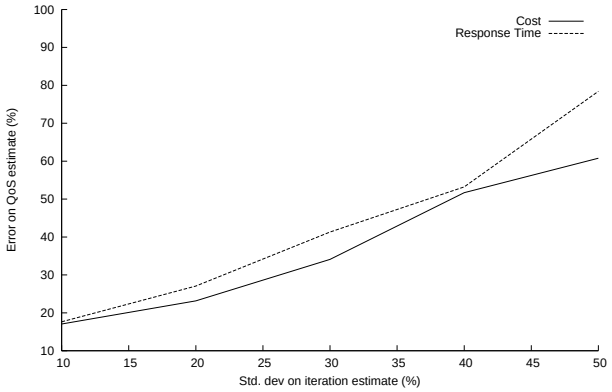

Figure 10: Error on QoS estimate due to uncertainty in the # of iterations

The general indication given by the simulations we performed is that the actual QoS obtained during execution deviates in the presence of wrong QoS estimates and, above all, of wrong estimates on the paths to be followed in the workflow. This would require to: i) refine the estimates as much as possible, using actual data obtained during different service executions, and ii) trigger re-planning as soon as this is necessary.

5.4 Triggering Re-plan during Simulations

The last set of experiments aimed to simulate the behavior of the re-planning trigger during workflow executions. We considered two different workflows, composed of 10 and 12 nodes respectively, a standard deviation of 5% on QoS estimates, and of 10% on the number of iterations estimates. Finally, we calibrated the re-planning threshold to 10% (i.e., the difference between Q_{EST} Q_{ACT} needs to be bigger than 10%). In our experience (on different workflows, with the above specified estimate errors), thresholds bigger than 30% would not lead to any re-planning.

Figure 11-a shows that, for the first workflow, the re-planning likelihood is of 56.8%, distributed on different nodes (two Sequence, one Switch and one invoke node, indicated as $S18\#0$). In the second case (Figure 11-b), the likelihood of re-planning is of 45.6%, however limited to a unique, Switch node. This because the node

splits the whole workflow in two cases with similar likelihood and, once the choice of the branch to be followed has been actually made, a re-planning is triggered to (eventually) re-plan only from that branch. It should be noted that the re-planning trigger can be useful not only at run-time, but also for analyzing how (and where) wrong QoS estimates could make a re-planning necessary during future executions thus wasting execution time.

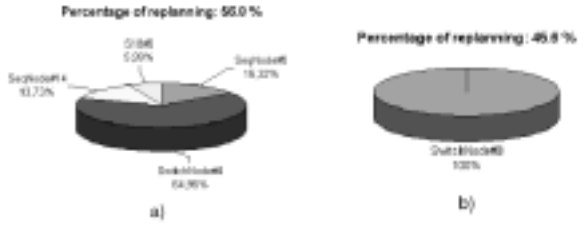

Figure 11: Triggering re-plan on different workflow nodes

6. CONCLUSIONS

In this paper we described an approach for QoS–aware service composition, based on composition of the QoS attributes of the component services and on Genetic Algorithms (GA). The aim of the approach is to provide a fast way, even if rough, to find the (sub)–optimal service composition and estimate its overall QoS. Constraints on the QoS attribute values are also kept into account. Obtaining an estimate quickly is particularly relevant for interactive services, where the time allowed to make the choice is limited. Numerical simulation showed the effectiveness of the approach and how an alternative approach, namely unfolding the workflow loops, can lead to better QoS values at the price of a higher search time. For example, the latter may be more suitable for the composition of non–interactive, computational intensive services.

In addition to the approach for composition, we proposed a re-planning algorithm to compute the deviation between the estimated QoS and the QoS measured at run–time, whenever possible. When the deviation goes above a threshold, the algorithm triggers a re-planning action, that is performed on the workflow slice that still remains to be executed.

Work–in–progress is devoted to better validate the approach on a large set of real services, as well as to further optimize the composition approach, for example with an hybrid optimization technique that combines GA and hill climbing. GA performances will also be compared with those of other optimization approaches. The whole toolkit is going to be integrated on a service broker we are developing in a project together with a large Italian software company.

7. REFERENCES

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, S. T. D. Smith, I. Trickovic, and S. Weerawarana. Business process execution language for web services. *http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/*.

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

[3] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Quality of service for workflows and web service processes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):281–308, April 2004.

[4] F. Casati, U. Dayal, D. Grigori, and M. Shan. Improving business process quality through exception understanding, prediction, and prevention. In *Proc. 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 159–168, Rome, Italy, Sept. 2001.

[5] F. Casati and M. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–162, May 2001.

[6] K. Deb. Multi-objective genetic algorithms: Problem difficulties and construction of test problems. *Evolutionary Computation*, 7(3):205–230, 1999.

[7] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[8] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub Co, Jan 1989.

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

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

[11] H. Ludwig. Web services qos: External slas and internal policies or: How do we deliver what we promise? In *Proc. 4th International Conference on Web Information Systems Engineering Workshops (WISEW'03)*. IEEE, 2004.

[12] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck. Web service level agreement (WSLA) language specification. *http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf*.

[13] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proceedings of the first International Semantic Web Conference (ISWC 2002)*, volume 2348 of *Lecture Notes on Computer Science*, pages 333–347. Springer-Verlag, June 2002.

[14] E. R. U.Greiner. Quality-oriented handling of exceptions in web-service-based cooperative processes. In *Proc. EAI-Workshop 2004 - Enterprise Application Integration*, pages 11–18. GITO-Verlag, 2004.

[15] K. P. V. Tosic, B. Pagurek. Wsol - a language for the formal specification of classes of service for web services. In *Proc. of the 2003 International Conference on Web Services (ICWS'03)*, pages 375–381. CSREA Press, 2003.

[16] W3C Working Group. Web services architecture. *http://www.w3.org/*.

[17] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5), May 2004.