Flexible Probabilistic QoS Management of transaction based Web services orchestrations

Sidney Rosario
INRIA
Centre Rennes Bretagne Atlantique
Rennes, France
sidney.rosario@inria.fr

Albert Benveniste
INRIA
Centre Rennes Bretagne Atlantique
Rennes, France
albert.benveniste@inria.fr

Claude Jard
ENS Cachan Bretagne,
Université Européenne de Bretagne
Bruz, France
jard@bretagne.ens-cachan.fr

***Abstract*—In this paper we extend our previous work on *soft probabilistic contracts* for QoS management, from the particular case of "response time", to general QoS parameters. Our study covers *composite* QoS parameters dealing not only with time aspects but also with *Quality of Data*. We also study contract *composition* (how to derive QoS contracts for an orchestration from the QoS contracts with its called services), and contract *monitoring*. Our approach supports comprehensive and flexible QoS management within a probabilistic framework.**

I. Introduction

Web services and their orchestrations are now considered an infrastructure of choice for managing business processes and workflow activities over the Web infrastructure [21]. Besides BPEL, the ORC formalism has been proposed to specify orchestrations, by W. Cook and J. Misra at Austin [15]. ORC is a simple and clean academic language for orchestrations with a rigorous mathematical semantics. For this reason, our study in this paper relies on ORC. Its conclusions and approaches, however, are also applicable to BPEL [3].

Contract based QoS management: When dealing with the management of QoS, *contracts*—in the form of *Service Level Agreements*, SLA [5]—specify the commitments of each subcontractor with regard to the orchestration. Most SLAs commonly tend to have QoS parameters which are mild variations of the following: response time (latency); availability; maximum allowed query rate (throughput); and security [12].

From QoS contracts with sub-contractors, the overall QoS contract between orchestration and its clients can be established. This process is called *contract composition*. Then, since contracts cannot only rely on trusting the sub-contractors, *monitoring* techniques must be developed for the orchestrator to be able to detect possible violation of a contract, by a sub-contractor. Finally, upon contract violation, the orchestrator may consider *reconfiguration*, i.e., replacing some called services by alternative, "equivalent" ones — we do not address this last task here.

Hard Contracts versus Soft Probabilistic Contracts: To the best of our knowledge, with the noticeable exception of [13], [10], [11], all composition studies consider performance related QoS parameters of contracts in the form of *hard bounds*. For instance, response times and query throughput are required to be less than a certain fixed value and validity of answers to queries must be guaranteed at all times. When composing contracts, hard composition rules are used such as addition or maximum (for response times), or conjunction (for validity of answers to queries). Whereas this results in elegant and simple composition rules, this general approach by using hard bounds does not fit the reality well *and may lead to over pessimistic promises*. Indeed, real measurements of response times for existing Web services reveal that they vary a lot and are better represented through their histogram. Thus we have proposed using *soft probabilistic contracts* instead. In such contracts, hard bounds are replaced by probabilistic obligations, *i.e.*, a QoS parameter Q is considered probabilistic and a *distribution function*, or *distribution* for short, $F_Q(x) = \mathbf{P}(Q \leq x)$ is agreed for all relevant values x of Q. The obligation is that the called service should behave "no worse" than F_Q regarding Q, in a sense that will be formalized later.

Contributions of this paper: In this paper we extend and systematize the approach of [18], [19] by extending it beyond the only case of Response Time. Our first contribution consists in proposing a comprehensive approach for Soft Probabilistic QoS Contracts encompassing a large class of QoS parameters taking values in partially ordered domains, together with means to build *composite* QoS parameters and contracts and reason about them. A second contribution consists in a procedure to perform flexible *contract composition*, which consists in relating the obligations binding the pair {client, orchestration}, to the obligations binding the different pairs {orchestration, called service}. A third (minor) contribution consists in the extension of the technique proposed in [19] for contract *monitoring* to our generalized case. This extension turns out to be straightforward, as we shall see. Last but not least, we discuss *languages features* that are useful in making our approach effective. Not surprisingly, QoS domains must be declared along with their characteristics allowing to perform contract composition. We also found it very useful to introduce a language *feature* that is generic with respect to the various QoS domains and performs a filtering of responses from called services or from pools thereof, according to best

This work was partially funded by the ANR national research program DOTS (ANR-06-SETI-003), DocFlow (ANR-06-MDCA-005) and the project CREATE ActivDoc.

QoS performance. We illustrate this with the ORC language. Our whole approach is supported by the *TOrQuE* tool (**T**ool for **Or**chestration **Qu**ality of Service **e**valuation), from which we present experimental results for contract composition.

Organization of the paper: Our study is supported by the UsedCarOnLine Example that we present and discuss in section II. Based on this example, we discuss in particular why QoS domains should be partially, not totally, ordered. Section III develops our general framework for flexible QoS management, including the procedure for contract composition. Experiments are reported in section IV.

II. THE USEDCARONLINE EXAMPLE

A. Informal description

The UsedCarOnLine example is shown in Figure 1. In search

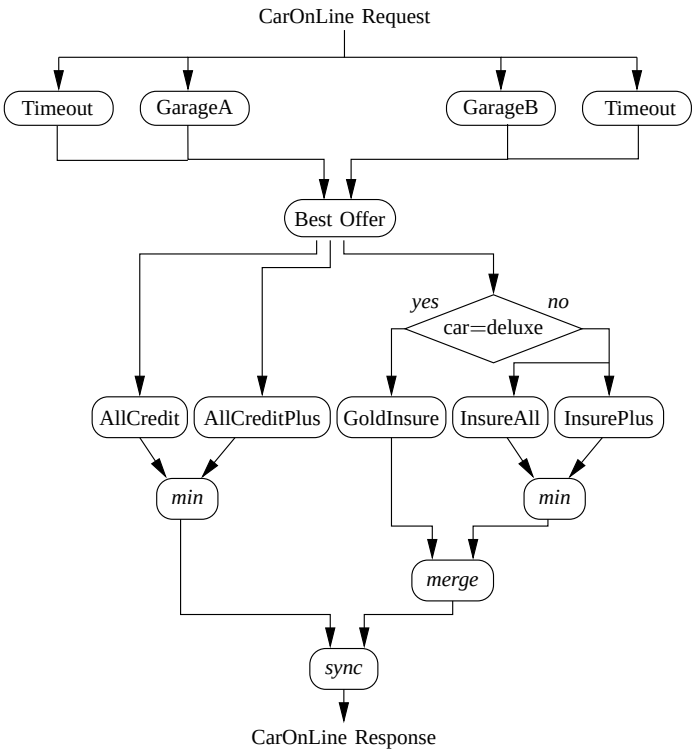

Figure 1. Schematic representation of the UsedCarOnLine example.

for a used car, a client calls the UsedCarOnLine orchestration with a car type — small car, family car, SUV, etc — as the input. The orchestration calls two garages, *GarageA* and *GarageB*, in parallel, with the client's car type as an input parameter. The garages respond with their price quote for that car. Another dimension to the response from the garage is the car's environmental friendliness that we assess using a *green level*.[1] The green level is used to select the environment friendlier car in *Best Offer*. The calls to the garages are guarded by a timer *Timeout*. If only one garage has responded when a timeout occurs, it is taken as the best offer and any eventual response of the other garage is simply ignored. If no garage responds before timeout, then a *Fault* message is returned to the client, indicating an exception.

After selecting the best offer for the car, UsedCarOnLine finds insurance and credit offers for this car. For credit offers, two services *AllCredit* and *AllCreditPlus* are called in parallel and the offer having the lower interest rate is chosen. The insurance services called depends on the type of car which needs to be insured. If the car requested by the client is of some "deluxe" category, then only one service — *GoldInsure* — can offer insurance for such cars, and any offer made by it is taken. If the car is not a "deluxe" car, then two services, *InsureAll* and *InsurePlus* are called in parallel and the insurance that costs the least amongst the two offers is selected. In the end, the (price, credit, insurance) is sent back to the client.

Each query to the orchestration comes as a token having data and QoS attributes. Each of these tokens are broadcast to the two garages and timers, which all produce a token at their output. These tokens are then combined when selecting the best offer. And so on. The traversal of any of the displayed sites by a token modifies both data and QoS attributes for the output tokens. Finally, one output token exits the orchestration for each given input token.

B. The ORC specification for UsedCarOnLine

The ORC program for UsedCarOnLine is given in Table I.

Assumptions QoS parameters :
δ : inter-query time, $D_\delta = \mathbb{R}_+$
Guarantees QoS parameters :
d : latency, $D_d = \mathbb{R}_+$
Q : clean level, $D_Q = \{0 \ldots 4\}$

$$UsedCarOnLine(car) \triangleq CarPrice(car) >p> \mathbf{let}(p, c, i)$$
$$\quad \mathbf{where}\ c :\in GetCredit(p)$$
$$\quad\quad\quad\ i :\in GetInsur(p, car)$$

$$CarPrice(car) \triangleq \{\ \mathbf{let}(p)$$
$$\quad \mathbf{where}$$
$$\quad\quad p :\in_Q \{GarCall(GarageA[\delta, d, Q](car)) \mid$$
$$\quad\quad\quad\quad GarCall(GarageB[\delta, d, Q](car))\}$$
$$\quad \}\ >p>\ \{\ \mathbf{if}(p \neq Fault)) \gg \mathbf{let}(p)\ \}$$

$$GarCall(g) \triangleq \mathbf{let}(a)\ \mathbf{where}\ a :\in \{\ g\ \mid\ \mathbf{RTimer}(T)\}$$

$$GetCredit(p) \triangleq Min[\delta, d](c1, c2)$$
$$\quad \mathbf{where}$$
$$\quad\quad c1 :\in AllCredit[\delta, d](p)$$
$$\quad\quad c2 :\in AllCreditPlus[\delta, d](p)$$

$$GetInsur(p, car) \triangleq \{\ \mathbf{if}(car = deluxe) \gg GoldInsure[\delta, d](p)\} \mid$$
$$\quad \{\ \mathbf{ifnot}(car = deluxe) \gg \{min(ip, ia)$$
$$\quad\quad\quad \mathbf{where}\ ip :\in InsurePlus[\delta, d](p)$$
$$\quad\quad\quad\quad\quad\ ia :\in InsureAll[\delta, d](p)$$
$$\quad \}\}$$

Table I
UsedCarOnLine in ORC. We have shown in light gray the add-ons to ORC.

ORC offers three primitive operators. For ORC expressions f, g, "$f \mid g$" executes f and g in parallel. "$f >x> g$" evaluates f first and for *every* value returned by f, a *new* instance of

[1] This refers to the current situation in France where a bonus/penalty system is attached to each car: when buying a clean car, you may get up to a 700 Euro bonus and you pay a penalty if the car is environment hostile. There are finitely many tax levels in this system.

g is launched with variable x assigned to this return value; in particular, "$f \gg g$" (which is a special case of the former where returned values are not assigned to any variable) causes *every* value returned by f to create a *new* instance of g. "f **where** $x :\in g$" executes f and g in parallel. When g returns its *first* value, x is assigned to this value and the computation of g is terminated. All site calls in f having x as a parameter are blocked until x is defined (*i.e.,* until g returns its first value).

The operator $:\in_Q$ is a new operator introduced for our QoS studies, where Q is the (static) parameter of this operator. Q is a QoS parameter whose domain is a partially ordered set $(D_Q, \leq)$ that is an upper and lower lattice; by convention, "best" will refer to a minimal element among a set. The expression "f **where** $x :\in_Q g$" does not take the first value returned by g as x. Instead it waits for a "best quality" response among all responses from g to that call, irrespective of the time taken to generate them — since the domain of Q is only partially ordered in general, a best response may not be unique. Observe that $:\in$ is a particular case of $:\in_Q$ by taking for Q the latency or response time of the call — in this case it is not needed to wait for all the responses from g to get the best one, since the first one received will, by definition, be the best.

C. QoS management and contracts

Semantics of QoS parameters: The orchestration wishes to establish SLA or QoS contracts with its clients. It wishes to provide *Guarantees* on response time (or latency) when subject to a query and *Guarantees* on the "green level" profile of its used car offers.

In turn, the orchestration will assume that its client complies with the *Assumption* that the query rate will not exceed some agreed profile; query rate is equivalently captured by inter-query time. Such an assumption is needed to avoid overloading and thus making it impossible for the orchestration server to process queries in due time. Thus, contracts are implications *assumptions* $\Rightarrow$ *guarantees*.

The example of table I begins with QoS parameters declarations, split into Assumptions and Guarantees. For each of these a list of QoS parameters and their domains is given.

Then, for each site call, a sub-list of all the QoS parameters is declared as QoS attributes for this site. For example, the declaration of the triple $[\delta, d, Q]$ in $GarageA[\delta, d, Q](car)$ indicates that site $GarageA$ knows the listed three QoS parameters. A token traversing this site will therefore exit with the following QoS attributes, which are the QoS Guarantees offered by the site: 1/ the latency d in traversing this site, and 2/ the green level Q associated to the response of this site. On the other hand, this site computes the elapsed time δ since the previous token was received; this belongs to the Assumptions under which the site should work and must therefore be monitored by the site, see section III-C.

Now, some sites have only a subset of the QoS parameters in their scope, e.g., both $AllCredit$ and Min know $[\delta, d]$ but ignore green level Q. Tokens traversing such sites keep their value unchanged for the ignored QoS parameter Q. If more than one input token are combined to form a single output token, as for Min, then two cases can occur: 1) All input tokens carry the same value for Q. In this case the output token has the same value for Q as the inputs. 2) The input tokens carry different values for Q. In this case the output token carries a special value *inconsistent* for Q. However if a token exiting the orchestration has the value *inconsistent* for Q, but Q appears as a parameter in the orchestration's contract with its client, then the orchestration will throw at error in this case. In the UsedCarOnLine example we are in the first case.

Probabilistic contracts: As advocated in [18], [19], guarantees on latency as well as assumptions regarding inter-query time must be of statistical nature. The "green level" QoS parameter relates to the general concept of Quality of Data (QoD) and has a different nature. Typically, QoD is qualitative (cheap/medium/expensive or silver/gold/platinum and so on); in our case, the green level may, for instance, be captured by the french categories of bonus/penalty, see footnote 1. If new cars were considered, QoD contract regarding green level would boil down to a hard constraint: make only offers involving cars with a green level not less than x. Since we deal with used cars however, availability is not always guaranteed and the distribution of "green level" in the available population of used cars at a given garage may very well be random. Thus "green level" will also be a random QoS parameter.

Why considering partially ordered domains for QoS parameters? There are two kinds of reasons for that.

First, some QoS domains may not be totally ordered. Consider the case of a pool of garages — here it consists of the two garages GarageA and GarageB — from which we expect getting a good offer. The larger the number of responses from the pool by its different garages, the better is the chance of getting a good offer. Therefore, a natural QoS parameter for a pool is the *subset of members of the pool* having responded. Now, subsets of a given set are partially ordered by inclusion.

Another reason for considering partial orders is the need for building composite QoS parameters. For example, in our case, we could make the pair {latency, green level} a single parameter. This composite QoS parameter would be naturally partially ordered using the product of the two orders: {latency', green level'}$\geq$ {latency, green level} iff latency'$\geq$ latency and green level' $\geq$ green level. Of course, we could instead totally order this pair by introducing a utility function (as in economy) u(latency, green level) taking values in, e.g., $\mathbb{N}$ or $\mathbb{R}_+$; utility functions may be sometimes arbitrary, however.

Now, there may be compelling reasons for considering composite QoS parameters, as we follow a probabilistic approach. Focus again on the two parameters "latency" and "green level". It clearly makes sense to assume that these two parameters are "uncorrelated" — formally, that they are probabilistically independent. For other cases, QoS parameters may not be assumed to be independent. They they must be packaged as a composite parameter endowed with a joint probability distribution taking correlation effects into account.

In the rest of the paper we will develop a framework for flexible QoS management based on a probabilistic ap-

proach. We leave aside the particular issue of *monotonicity* [6]: QoS management implicitly assumes that, the better a called service performs, the better the orchestration will perform. This reference provides conditions for monotonicity as well as guidelines for the design of monotonic orchestrations.

III. A FRAMEWORK FOR FLEXIBLE QOS MANAGEMENT

A. QoS domains and probabilistic contracts

QoS domains play a different role for Assumptions and Guarantees. Assumptions concern the flows of queries submitted to the orchestration or services. In contrast, Guarantees are performance obligations of the orchestration or the called services, i.e., they concern the servers supporting the considered services, and/or the performance of the orchestration seen as a composite service. Hence we address QoS domains for Assumptions and Guarantees separately. We first begin with Guarantees and then discuss Assumptions.

QoS domain for Guarantees: such a QoS domain is a tuple $(D, \leq, \oplus)$ where:

- $(D, \leq)$ is a partial order which is a complete lower and upper lattice; thus infimums, denoted by $\wedge$ and supremums, denoted by $\vee$, can be considered, with the usual algebraic properties. The two operators $\wedge$ and $\vee$ will be useful in combining QoS of services that are called concurrently or in conflict.
- $(D, \leq, \oplus)$ is a commutative semi-ring, meaning that $\oplus$ is a commutative and associative operation on D that distributes over $\wedge$. This operator will be used when composing QoS for services called in sequence.

Examples of such QoS domains include:

- $(\mathbb{R}_+, \leq, +)$ for *latency*. Best is shortest for latencies. Responses awaited in conflict yields the min of the latencies; the basic example is the ORC statement "f **where** $x :\in g$". Synchronous waiting for responses of concurrent calls yields the max of the latencies — since all returns must be received. Latencies add for calls performed in sequence.
- $([0 \ldots L], \leq, \vee)$ for *green level*. 0 is the best value (lowest tax, or, equivalently, max bonus) and L the worst. If a composite service consists of a sequence of calls involving environmental issues, one could consider that the worst level encountered in the successive responses yields the level of the whole.

QoS domain for Assumptions: such a QoS domain is a partial order $(D, \leq)$ that is a complete lower and upper lattice. Examples of such QoS domains include:

- $(\mathbb{R}_+, \geq)$ for *inter-query time*. From the server's point of view, best is longest, whence the choice of $\geq$. The inter-query time is measured by the orchestration and each service, by comparing the dates of successive queries.

For Q a QoS parameter, we will denote by D_Q its domain. QoS domains *compose* by taking their products when seen as partial orders or as semi-rings.

Probabilistic contracts: The domain D_Q of a QoS parameter Q can be randomized by equipping it with a probability $\mathbf{P}$.[2]

Consider first the case where $(D_Q, \leq)$ is a total order. For this case we can reuse the approach of [19] where the probabilistic behavior of Q is represented by its *distribution*:

$$F_Q(x) = \mathbf{P}(Q \leq x).$$

Now, suppose that $F_{Q,S}$ has been agreed for the QoS parameter Q, between the orchestration and some called service S. How to formalize that "S performs at least as good as agreed"? (In this case the orchestration should be happy with S regarding QoS parameter Q.) We need an order on probability distributions. It turns out that such a *stochastic order* for distributions exists [2], [17]. For F and G two probability distributions over a totally ordered domain D, say that

$$G \leq_s F \quad \text{iff} \quad \forall x \in D \Rightarrow G(x) \geq F(x) \tag{1}$$

This definition reads as: there are more chances of being less than x if the random variable is drawn according to G than according to F — whence the reverting of inequalities. If X and Y are two random variables with respective distributions F and G, then

$$G \leq_s F \text{ if and only if } \mathbf{E}(\varphi(X)) \leq \mathbf{E}(\varphi(Y)) \tag{2}$$

holds, for any real valued *increasing* function φ. Now, if $F_{Q,S}$ has been agreed as said above, and service S actually responds with probability distribution $G_{Q,S}$, the agreement is met iff $G_{Q,S} \leq_s F_{Q,S}$ holds.

Stochastic ordering has been considered in [17] for the case when $(D, \leq)$ is only a partial order. Observe that the characterization provided in (2) can be taken as a definition of stochastic ordering in this case. We give a new characterization here, not given in [17]. This is obtained by considering *ideals* of D, *i.e.*, subsets I of D that are downward closed:

$$x \in I \text{ and } y \leq x \implies y \in I$$

Examples of ideals are: for $\mathbb{R}_+$, the intervals, $[0, x]$ for all x; for $\mathbb{R}_+ \times \mathbb{R}_+$ equipped with the product order, arbitrary unions of rectangles $[0, x] \times [0, y]$. Now, if Q is a QoS parameter over a partially ordered QoS domain $(D_Q, \leq)$, we define its *distribution* by

$$F_Q(I) = \mathbf{P}(Q \in I),$$

for I ranging over the set of all ideals of D_Q. Again, we then define, for F and G two distributions over D_Q,

$$G \leq_s F \quad \text{iff} \quad \text{for any ideal } I \text{ of } D \Rightarrow G(I) \geq F(I) \tag{3}$$

We now have the needed apparatus for defining probabilistic contracts — in this paper we restrict ourselves to contracts involving only two parties; we will discuss the case of orchestration versus called service, but the same concepts apply to client versus orchestration.

[2] We omit the technicalities behind this notion, e.g., measurability and so on; a demanding reader may, for simplicity, restrict herself to finite domains.

Following the established approach of WSLA [12], a contract must specify the obligations of the two parties. Since we deal with the asymmetric pairs {client, orchestration} or {orchestration, called service}, we will use an asymmetric wording for the obligations. Let us focus from now on, on a pair {orchestration, called service} and take the point of view of the called service:

- the obligations that the orchestration has regarding the service are seen as *assumptions* by the service; the orchestration is supposed to meet them and the service is bound to its obligations as long as assumptions are met;
- the obligations that the service has regarding the orchestration are seen as *guarantees* by the service; the service commits to meeting them as long as assumptions are met.

Definition 1 (probabilistic contract): A probabilistic contract *is a pair* $\{A, G\} = \{Assumptions, Guarantees\}$*, which both are lists of tuples* (Q, D_Q, F_Q)*, where* Q *is a QoS parameter with QoS domain* D_Q *and distribution* F_Q*.*

The precise mathematical semantics of such a contract will be made clear when discussing contract composition and monitoring. The QoS declaration part of table I provides an example of QoS parameter declaration. Specific contracts are established with each called service regarding relevant QoS parameters for this service, by providing a distribution for it. Details regarding this will be provided in section IV.

B. Contract composition

Contract composition is the process by which the orchestration can build a contract with its client, considering the contracts it has with the services it calls. Due to the assume/guarantee type of reasoning, contract composition is an intricate problem. It is further complicated by the fact that our contracts are probabilistic and orchestrations involve complex interactions between control, QoS parameters, and data — see [19] for a detailed discussion of the latter point.

Fortunately, QoS management of orchestrations exhibits a special structure regarding causality between assumptions and guarantees. Consider again the UsedCarOnLine example of figure 1. Consider first the latency, which is declared a Guarantee. Causality regarding latency flows outward in the following sense: from knowing the latencies of each service, one can deduce the overall latency of the orchestration. In contrast, the throughput (represented by the inter-query time) is declared an Assumption. Regarding throughput, causality flows inward, from client to services, in the following sense: knowing the dates of arrival of the calls to the orchestration, one can observe the dates of resulting calls, for the different services. Accordingly, contract composition is performed as explained next, using Monte-Carlo simulations.

Contract composition procedure

In this procedure, all probability distributions are assumed independent in the probabilistic sense, for QoS parameters associated to different services and for different QoS parameters associated to a same service.

a) Initial Conditions:

- *Assumptions*: the distribution of the Assumptions A_O for the orchestration is specified — for our example: inter-query time.
- *Guarantees*: the distribution of each Guarantee G_S is specified, for each called service — for our example: latency and green level (whenever relevant). Sometimes, the orchestration may contain calls to "public" services (like Google) which are freely available and cannot be contracted. For such services, a contract is replaced by an estimation of the service's performance, which can be done through measurements.

b) Inward Sweep:

1) Generate calls to the orchestration randomly, according to the agreed distribution for their inter-query time;
2) For each query to the orchestration, run a Monte-Carlo simulation of the orchestration. Corresponding occurrences and dates of calls to (a subset of) the different services are observed.
3) Collect the dates of the successive calls to a same service during the series of calls to the orchestration. This yields the successive inter-query times for each called service and allows to specify the Assumptions for each called service. Also, the resulting QoS parameters "latency" and "green level" for the orchestration are stored for possible subsequent reuse.

c) Outward Sweep: at this point all Assumptions have been specified. Denote the orchestration by the symbol O. For each service S, the pair (O, S) has both its Assumptions A_S and Guarantees G_S specified. Then, two cases may occur.

- For the *good case*, all contracts $\{A_S, G_S\}$ form an acceptable contract for all services S. In this case, we can reuse the data generated at step 3) of the inward sweep to get an empirical estimate of the distributions constituting the Guarantees offered by the orchestration to its client, completing contract composition.
- For the *bad case*, some pairs $\{A_S, G_S\}$ do not constitute an acceptable contract, i.e., guarantees G_S may be too demanding considering the Assumptions A_S. We can then adopt two alternative iterative approaches: For the first approach, we iterate on Guarantees offered by the called services, *i.e.*, given the Assumptions A_S for each service S, contracts are re-negotiated, which results in a new setting for the Guarantees G_S. For the second approach, we iterate on Assumptions applied to the orchestration, *i.e.*, we redesign A_O. With any of the two approaches, we have updated the *Initial Conditions* and are now ready to re-run the process, until all contracts are accepted.

This iterative approach resembles the technique of *policy iteration*, used in dynamic programming and game theory to find similar equilibria [4]. We have no convergence proof yet, but policy iteration techniques are known to converge in a few iterations in many cases — as exemplified by our experiments of section IV.

Discussing the independence hypothesis

Independence of all distributions is assumed while performing contract composition. Is this acceptable? Consider first the case where different QoS parameters are associated to a service. If independence is not an acceptable hypothesis, then just make the tuple of these QoS parameters a new composite QoS parameter, see section II-C — our framework is powerful enough to allow for this. Of course, this comes with a price: estimating independent probability distributions is cheaper (requires less data) than estimating a joint distribution. Now consider the case of different services. The independence hypothesis is generally accepted here. It is needed if contracts are to be negotiated on a pairwise basis, between the orchestration and each individual service. Otherwise, group negotiation would be needed, a much heavier process.

C. Contract monitoring

Once contracts have been agreed, they must be monitored by the orchestration for possible violation. Contract monitoring is studied in detail in [19] for the case of a single QoS parameter, namely the latency. The same technique, however, extends without change, to our case. We nevertheless reproduce it here because QoS domains can be partially, not totally, ordered in our case. Monitoring applies to each contracted distribution F individually, where F is the distribution associated to some QoS parameter X having partially ordered domain D. By monitoring the considered service, the orchestration can get an estimate of the actual distribution of X, we call it G. The problem is, for the orchestration, to decide whether or not G complies with F, where compliance is defined according to formula (3), rewritten as

$$\sup_{I \in \mathcal{I}_D} F(I) - G(I) \quad \leq \quad 0 \tag{4}$$

where $\mathcal{I}_D$ denote the set of ideals of D. However, $G(I)$ in (4) is not given to the orchestration, it can only be estimated by collecting actual values for QoS parameter X. To this end, we consider the following basic *empirical estimate* for G, namely:

$$\widehat{G}_\Delta(I) \quad = \quad \frac{|\{x \in \Delta | x \in I\}|}{|\Delta|}$$

where Δ is a sample of values for X collected at run time by the orchestration and $|A|$ is the cardinal of set A. Estimate $\widehat{G}_\Delta$ converges toward G when the size of Δ grows to infinity. In practice, successive values for $\widehat{G}_\Delta$ are updated on-line at run time by collecting in Δ buffered values for X in a buffer of size N large enough. If Δ_t is the content of the buffer at time t, we thus get an estimate $\widehat{G}_{\Delta_t}$, which we denote by $\widehat{G}_t$ for simplicity. Then, the indicator in (4) is replaced by:

$$\chi_t \quad =_{\text{def}} \quad \sup_{I \in \mathcal{I}_D} F(I) - \widehat{G}_t(I)$$

At a first sight, a violation should be declared at the first instant t when $\chi_t > 0$ occurs. The problem is that estimate $\widehat{G}_t(I)$ can randomly fluctuate around $G(I)$, especially for N not large enough. Hence, applying the brute force stopping rule $\chi_t > 0$ will inevitably result in many false alarms. A countermeasure consists in having a *tolerance zone* above the critical value 0. This yields the following stopping rule for declaring violation: $\chi_t \geq \lambda$, where $\lambda > 0$ is a design parameter of the procedure, defining the tolerance zone. We do not provide here the details of how monitoring is implemented, the reader is referred to [19], section V for this.

D. Language features for flexible QoS management

As evidenced from the UsedCarOnLine example of figure 1, ORC requires some additional language features to make it QoS-enabled. QoS parameter declarations, including the specification of probability distributions,[3] fully comply with the state-of-practice in WSLA. We will therefore rather focus on the needed *operators*:

(a) Wait synchronously for the returns of concurrent service calls and combine the QoS values for the collected returns. This does not require any specific language feature. The designer should simply state herself how the combination is performed by specifying a formula for the combination. Default combination is by assigning the worst collected value to the tuple.

(b) Combine QoS values of same parameters, for calls performed in sequence. Again, this does not require any specific language feature but the declaration of the operation $\oplus$ for this type of QoS parameter, see section III.

(c) Wait asynchronously for the returns of concurrent service calls and select a *best candidate* among the responses, based on a given QoS parameter. *This leads to considering the* $:\in_Q$ *operator*, see table I. An in-depth study of this operator and its mathematical semantics is beyond the scope of this paper and will be reported elsewhere.

To summarize, only one specific new operator needs to be considered to make ORC QoS-enabled, namely the $:\in_Q$ operator.

IV. EXPERIMENTS ON CONTRACT COMPOSITION

We do our experiments on the UsedCarOnLine example of section II. Our experiments are on the contract composition technique detailed in this paper. Experiments on contract monitoring are not done here, the interested user may refer to [19] for this. All experiments were done on a machine with a 1,000 MHz Pentium Centrino CPU, with 2 GB of memory.

The sites *GarageA*, *GarageB*, *AllCredit*, *AllCreditPlus*, *GoldInsure*, *InsurePlus* and *InsureAll* were assigned latency behaviours inferred from measured values of calls to services over the web. For this, we invoked six web services — USWeather, Bushism, XMethods, StockQuote, Caribbean and CongressMembers — found in the XMethods online repository [22]. We made 20,000 calls to each of these six services and recorded the response time for each of these calls. We then increasingly reordered these measurements and picked a certain number (in our case seven) quantiles. The response times are assumed to be uniformly distributed between these

[3] In practice, distribution F will be abstracted by either a finite set of *quantiles* $(F(x_1), \ldots, F(x_K)$, for a fixed family $x_1, \ldots, x_K$ of values for the QoS parameters) or a finite set of *percentiles* (e.g., the set of values $y_1, \ldots, y_9$ such that $F(y_1) = 10\%, \ldots, F(y_9) = 90\%$). Such contracts are easily expressible in terms of the WSLA standard [12].

quantiles, except after the highest one, after which the response time decreases exponentially. The estimated distribution for each of the contract sites of UsedCarOnLine is given in figure 2. Other ways of using these measurements were experimented [19] but are not reported here, due to lack of space.

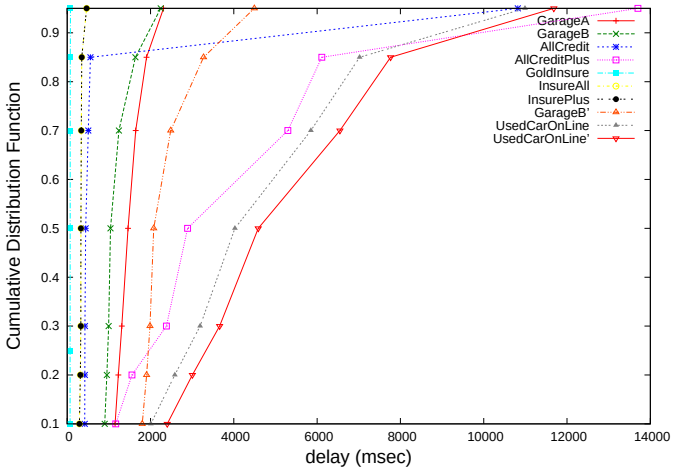

Figure 2. Interpolated distributions for each of the contracted sites, and the end to end distribution for UsedCarOnLine.

All the other sites of UsedCarOnLine like **let**, **if**, etc are internal to the orchestration. Their response times are negligible in comparison to that of the contracted sites, and so are assumed to respond instantaneously with zero delay.

We take the "green level" to be a random parameter with values in $\{0 \ldots 4\}$, for the two sites where it is contracted with (*i.e., GarageA* and *GarageB*). The corresponding probability is given in the second and third column of Table II.

Green Level	Probability GarageA	Probability GarageB	Probability UsedCarOnLine
0	0.25	0.2	0.207
1	0.25	0.25	0.25
2	0.25	0.25	0.25
3	0.15	0.25	0.236
4	0.1	0.05	0.055

Table II
Probability for "green level", for GarageA, GarageB and the UsedCarOnLine orchestration.

Sweep 1: We first do an inward sweep, as described in the contract composition method of section III-B. We generate random calls to the orchestration following the exponential distribution for inter-arrival times with a rate parameter of 5 requests/second. We ran 100,000 iterations of the orchestration. The resulting throughput for each of the contracted sites is given in Table III. The end to end latency for UsedCarOnLine is given by the *UsedCarOnLine* curve in figure 2.

Sweep 2: During the negotiation phase, say that *GarageB* finds that the request rate of 5.028 calls per second is too demanding for the performance it guarantees. This

Site Name	Throughput (sweep 1)	Throughput (sweep 2)
GarageA	5.028	5.0
GarageB	5.028	5.0
AllCredit	5.028	4.99
AllCreditPlus	5.028	4.99
GoldInsure	1.679	1.674
InsureAll	3.342	3.332
InsurePlus	3.342	3.332

Table III
Average throughput for each of the contracted sites.

corresponds to the *Bad Case* as mentioned in section III-B. In this case, UsedCarOnLine could reduce its own input rate so that *GarageB* is not invoked that often. Another possibility would be that *GarageB* agrees to support this request rate, but at a decreased performance. The new contract is given in figure 2 by the *GarageB'* curve. The resulting throughput for the sites is given in table III and the end to end orchestration delay in this case is given by the *UsedCarOnLine'* curve. In our case, the process converges after a single iteration (the throughput remain almost unchanged for the two sweeps), despite the drastic decrease in GarageB's response time (almost twice slower). In the general case, this could need more sweeps. The quality distribution for whole UsedCarOnLine program after Sweep two is given in the last column of table II. The overall execution time for both the sweeps was about 23.5 seconds.

V. Related Work

Proposals for QoS-based SLA composition are few and no well-accepted standard exists to date. Menascé [14] discusses QoS issues in Web services, introducing the response times, availability, security and throughput as QoS parameters. He also talks about the need of having SLAs and monitoring them for violations. He does not however, advocate a specific model to capture the QoS behaviour of a service, or a composition approach to compose SLAs. Agarwal et. al [1] view QoS based SLA composition as a constraint satisfaction/optimization problem solved by linear programming. Cardoso et al. in [8] follow a rule based approach to derive QoS parameters for a workflow, given the QoS parameters of its component tasks. Zeng et al. [23] use Statecharts to model composite services and use linear programming techniques such that it optimizes a specific global QoS criteria. In [16], the authors propose using fuzzy distributed constraint satisfaction programming (CSP) techniques for finding the optimal composite service. Canfora et. al [7] use Genetic Algorithms for deriving optimal QoS compositions. Compared to the linear programming method of Cardoso et. al [8], the genetic algorithm is typically slower on small to moderate size applications, but is more scalable.

A distinguishing feature of our proposal is that we deviate from using hard bounds and handle soft probabilistic contracts.

In [9] the authors use WSFL (Web Service Flow Language) and enhance it with the capability to specify QoS attributes. Web service Performance Analysis Center (sPAC)

[20], is another similar approach for performance evaluation of services and their compositions. For both works, probabilistic models are translated into simulation engines for performance analysis. The fundamental difference from our approach is that the approach assumes a "closed world" scenario, assuming that the services of the orchestration can be instrumented with measurement code to get information about its performance. We rely on contracts, instead.

The notion of probabilistic QoS has been introduced and developed in [10], [11] with the ambition to compute an exact formula for the composed QoS, which is only possible for restricted forms of orchestrations without any data dependency. We propose using simulation techniques to analyze the QoS of a composite service, this allows us to use non-trivial distributions as models for performance and also permits analysis of orchestrations whose control flow have data and time related dependencies. A distinct feature of our approach is that the quality domains can be partially ordered which allows expressing rich and possibly complex QoS parameters.

VI. Conclusion and Perspectives

In this paper we have proposed a framework for QoS management based on *soft probabilistic contracts*. This work is a step forward toward establishing QoS management of Web services on a mathematically sound basis. Our vision is targeted to the use of Web services for business processes, in a semi-open world such as multi-tier supplier chains. According to this vision, Web services provider interact via *contracts*.

More precisely, Web service interfaces must expose information regarding the following: 1/ *how they should be queried* — this involves conformance of the query with regard to data types, semantic aspects of data, and, for more sophisticated services involving complex, dynamic, interaction, the allowed dialogs between the service and the client who queries it; but this also involves QoS aspects, e.g., maximal query throughput or allowed complexity of the submitted query; 2/ *how they respond when properly queried* — this involves conformance of the return with regard to data types, semantic aspects of data, and, for more sophisticated services involving complex, dynamic, interaction, the possible dialogs between the service and the client who queries it; but this also involves QoS aspects, e.g., maximal latency, quality of response, etc. Today, Web service interfaces are mostly poor in many of these aspects — emphasis is on typing issues and data semantics and QoS issues are handled in a primitive way.

Orchestrations allow composing services to form new services. This immediately raises the issue of how to compose contracts. Contract composition was the heart of this paper. To account for variability in Web servers and networks, QoS parameters were considered to be random. This is still not common, but we strongly believe it should be like this. In such a probabilistic framework, QoS contracts consist in exposing percentiles for the QoS parameter in consideration, e.g., 95% of responses better than…, 75% of responses better than…, etc. Our main contribution is a general procedure for multi-QoS contract composition in such a probabilistic context.

QoS relates to performance. Therefore it makes sense assuming that a Web service outperforming its contract should do well for the orchestration — actually all SLA are designed with this implicit assumption in mind. Formal study of monotonicity of orchestrations w.r.t. QoS parameters is the subject of other ongoing work.

References

[1] Rohit Aggarwal, Kunal Verma, John A. Miller, and William Milnor. Constraint Driven Web Service Composition in METEOR-S. In *IEEE SCC*, pages 23–30, 2004.

[2] G Anderson. Nonparametric Tests of Stochastic Dominance in Income Distributions. *Econometrica*, 64(5):1183–1193, 1996.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, I. Thatte D. (Editor), Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services. [BPEL4WS.] version 1.1, May 2003.

[4] Dimitri P. Bertsekas and John Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996. ISBN: 1-886529-10-8.

[5] Preeti Bhoj, Sharad Singhal, and Sailesh Chutani. SLA management in federated environments. *Computer Networks*, 35(1):5–24, 2001.

[6] Anne Bouillard, Sidney Rosario, Albert Benveniste, and Stefan Haar. Monotonicity in Service Orchestrations. Technical Report 6658, Inria, April 2008.

[7] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for QoS-aware service composition based on genetic algorithms. In *GECCO*, pages 1069–1075, 2005.

[8] Jorge Cardoso, Amit P. Sheth, John A. Miller, Jonathan Arnold, and Krys Kochut. Quality of Service for workflows and Web service processes. *J. Web Sem.*, 1(3):281–308, 2004.

[9] Senthilanand Chandrasekaran, Gregory A. Silver, John A. Miller, Jorge Cardoso, and Amit P. Sheth. XML-based Modeling and Simulation: Web Service Technologies and their Synergy with Simulation. In *Winter Simulation Conference*, pages 606–615, 2002.

[10] San-Yih Hwang, Haojun Wang, Jaideep Srivastava, and Raymond A. Paul. A Probabilistic QoS Model and Computation Framework for Web Services-Based Workflows. In *ER*, pages 596–609, 2004.

[11] San-Yih Hwang, Haojun Wang, Jian Tang, and Jaideep Srivastava. A probabilistic approach to modeling and estimating the QoS of web-services-based workflows. *Inf. Sci.*, 177(23):5484–5503, 2007.

[12] Alexander Keller and Heiko Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *J. Network Syst. Manage.*, 11(1), 2003.

[13] Zhen Liu, Mark S. Squillante, and Joel L. Wolf. On maximizing service-level-agreement profits. In *ACM Conference on Electronic Commerce*, pages 213–223, 2001.

[14] Daniel A. Menascé. Qos issues in web services. *IEEE Internet Computing*, 6(6):72–75, 2002.

[15] Jayadev Misra and William R. Cook. Computation Orchestration: A Basis for Wide-Area Computing. *Journal of Software and Systems Modeling*, May, 2006. Available for download at http://dx.doi.org/10.1007/s10270-006-0012-1.

[16] Xuan Thang Nguyen, Ryszard Kowalczyk, and Manh Tan Phan. Modelling and Solving QoS Composition Problem Using Fuzzy DisCSP. In *ICWS*, pages 55–62, 2006.

[17] Yosef Rinott and Marco Scarsini. Total positivity order and the normal distribution. *Journal of Multivariate Analysis*, 97:1251–1261, 2006.

[18] Sidney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard. Probabilistic qos and soft contracts for transaction based web services. In *ICWS*, pages 126–133, 2007.

[19] Sidney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard. Probabilistic QoS and soft contracts for transaction based Web services orchestrations. *IEEE Transactions on Service Computing*, 1(4), 2008.

[20] Hyung Gi Song and Kangsun Lee. sPAC (Web Services Performance Analysis Center): Performance Analysis and Estimation Tool of Web Services. In *Business Process Management*, pages 109–119, 2005.

[21] Wil M. P. van der Aalst and Kees M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[22] XMethods. http://www.xmethods.net.

[23] Liangzhao Zeng, Boualem Benatallah, Anne H. H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Trans. Software Eng.*, 30(5):311–327, 2004.