QoS-Aware Replanning of Composite Web Services

Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, Maria Luisa Villani

RCOST — Research Centre on Software Technology

Department of Engineering - University of Sannio

Viale Traiano - 82100 Benevento, Italy

{canfora, dipenta, r.esposito, villani}@unisannio.it

Abstract

Run-time service discovery and late-binding constitute some of the most challenging issues of service–oriented software engineering. For late-binding to be effective in the case of composite services, a QoS-aware composition mechanism is needed. This means determining the set of services that, once composed, not only will perform the required functionality, but also will best contribute to achieve the level of QoS promised in Service Level Agreements (SLAs).

However, QoS-aware composition relies on estimated QoS values and workflow execution paths previously obtained using a monitoring mechanism. At run-time, the actual QoS values may deviate from the estimations, or the execution path may not be the one foreseen. These changes could increase the risk of breaking SLAs and obtaining a poor QoS. Such a risk could be avoided by replanning the service bindings of the workflow slice still to be executed.

This paper proposes an approach to trigger and perform composite service replanning during execution. An evaluation has been performed simulating execution and replanning on a set of composite service workflows.

Keywords: Dynamic binding, Replanning, Composite Web Services

1 Introduction

Web service technology is greatly contributing to change the landscape of today's software engineering. One of the most promising advantages of the service-oriented paradigm is related to service run-time discovery and late-binding mechanisms. Run-time discovery implies that our program or workflow, instead of containing a direct invocation to a service, contains an unambiguous, semantic description of the functionality needed at that point. At run-time, a matchmaking mechanism is used to discover services that match that description, and one of these services is automatically invoked.

In many cases given a semantic, unambiguous description of a service (hereby referred to as *abstract service*), several services (hereby referred to as *concrete services*) may exist that match such a description. Late-binding mechanisms provide means to select those services that best contribute to maximize a set of local (i.e., related to the single invocation) and global (i.e., related to the whole system or composite service) optimum criteria (e.g., related to response time, cost and, in general, to any QoS attribute), and, above all, to meet QoS constraints formalized in the SLA.

Much in the same way, a composite service results from a composition of abstract services. Thus, it is necessary to define a QoS–aware composition mechanism (such as the ones defined by Cardoso et al. [7] or Zeng et al. [17]) aiming to determine the set of concrete services so that the composite service QoS is optimized and, above all, meets the SLA. The composite service QoS can be computed according to some aggregation formulae, such as those defined by Cardoso et al. [7]. The formulae define aggregation functions for each pair QoS attribute – workflow construct (e.g., node type in a workflow description language such as BPEL4WS [2]). Clearly, the formulae applicability relies on estimates of service execution parameters, namely QoS values for invoked services, and execution traces in workflows of composite services (used to determine the likelihood each branch in the workflow has to be followed). All the mentioned information can be obtained by monitoring the composite service execution.

At execution time, the actual values will almost surely deviate from the estimates. Iterations and execution paths depend on the user inputs, and actual QoS values can vary, for example, because of the network load. Finally, in the worst case, some services may not be available when requested. The effect of deviations from estimated values of QoS attributes and execution profiles is that the actual QoS of a composite service would not be the one agreed in the SLA. To avoid this, it is necessary to replan the service composition, i.e., to remake the bindings between abstract and

concrete services.

This paper proposes an approach to trigger replanning opportunities during composite service execution. In summary, replanning is triggered as soon as it is possible to predict that the actual service QoS will deviate from the initial estimates (an early version of the replanning trigger, as well as the adopted QoS aware composition mechanism, have been presented in [4]). Then, the slice, i.e., the part of the service workflow that still has to be executed, will be determined and replanned. The proposed approach relies on a proxy-based architecture to easily permit the binding between abstract and concrete services, as well as to perform replanning.

The validity of the approach has been evaluated simulating the execution of composite services and their replanning, and monitoring the actual QoS values. The tradeoff between having an early sensitive replanning and the consequent overhead is evaluated and discussed.

The remainder of this paper is organized as follows. After a review of the literature in Section 2, Section 5 describes the architecture for service binding. The composition approach proposed in the paper [4] is summarized in Section 3, while the replanning is detailed in Section 4. Section 5 describes the architecture used to enable replanning. Section 6 reports and discusses results obtained in the simulations. Finally, Section 7 concludes.

2   Related Work

In service-oriented systems, 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. In this respect, some formalism proposals for the service QoS specification and SLAs have been provided, such as the Web Service Offerings Language (WSOL) [15] or the IBM's Web Service Level Agreement (WSLA) language [12]. Instead, the currently available technology still lacks of facilities for a complete QoS estimation, management and monitoring for processes.

Cardoso et al. [7] propose a mathematical model for workflow QoS computation, described by some metrics aggregation functions which are defined for time, cost, reliability, and fidelity. The QoS computation algorithm consists of applying a set of reduction rules to the workflow until one atomic task is obtained. We use a similar reduction approach for QoS estimation, and have modified the aggregation functions to enable dynamic service binding and replanning (see also a previous paper [4] for details).

The work of Zeng et al. [17] proposes a global planning approach to reach overall QoS optimality through integer programming techniques. Their method consists on unfolding loops, once estimated the number of iterations,

and the binding for each task is decided based on the most frequently executed path containing that task. We use an alternative solution, namely binding all the occurrences of an abstract service in the workflow to the same concrete service, that might lead to a suboptimal solution but with better performances [4]. Also, we improve their idea of workflow replanning and introduce a triggering algorithm to predict such an event as soon as possible during execution.

A paper by Aggarwal et al. [1] describes a framework for a constraint driven service composition. Abstract processes are defined in BPEL4WS and the specification is completed after the binding with the concrete services and before execution. Performance issues of the constraint solver are not discussed nor dynamic workflow replanning.

Web-Flow [11] and eFlow [8] are workflow management systems that offer some support to selection of services according to quality constraints but this is limited to individual tasks.

Some very recent works propose to include ad-hoc service calls in the process description to enable dynamic service discovery and late binding [13]. An approach for monitoring service compositions is presented in a paper by Baresi et al. [3]. Monitors are defined as additional services of a process and used to validate contracts of the individual services, expressed through assertions in the process specification. Spanoudakis et al. [14] propose a framework for checking requirements compliance during process execution, where the expected behavior and assumptions are expressed in event calculus.

3   QoS-aware Composition

The first step is to estimate the QoS of a composite service, and to determine the optimal set of concrete services to be bound to the abstract services composing the workflow.

3.1   Computing the QoS of Composite Services

To compute the QoS attributes of a composite service, we used the aggregation formulae proposed by Cardoso et al. [7] for each pair QoS-attribute/composition language control statement (e.g., *sequence*, *switch*, *loop* or *flow*). QoS is computed by recursively applying these formulae for compound nodes of the service workflow. Similarly to what proposed by Cardoso et al., for a *switch* construct in the workflow, each *case* statement is annotated with the probability to be chosen. 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:

$$p\,C_1\ +\ (1-p)\,C_2 \tag{1}$$

Also, *sequence* and *flow* are handled similarly to Cardoso et al., basically using additive formulae for attributes such as cost, and multiplicative formulae for attributes such as availability or reliability. *Loops* (i.e., *while* constructs) are handled differently from Cardoso et al. 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 by taking into account the factor k. For example, if the *loop* compound node has a cost C_l, then the estimated cost of the *loop* will be $k\,C_l$.

Actually, for the QoS estimation, we create a new workflow where *loops* are removed, and each invoke node has a weight, representing the number of times that the node should be accounted for in the QoS formulae. This workflow, hereby referred to as *unlooped workflow*, will be used throughout the process execution only for QoS estimation purposes, whenever replanning is triggered.

3.2 Searching for an optimal solution

Determining the best concretization of a composite service is an optimization problem, aiming to i) maximize a fitness function of the available QoS attributes; and ii) 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 abstract service composing our service, need to be checked when choosing the set of candidate concrete services to bind.

Finding a solution for the above problem is NP-hard [9]. In this case, different strategies can be adopted, for example Integer Programming [17] or Genetic Algorithms (GAs) [10], or else Constraint Programming [16]. In our work we have tried some of these approaches with the aim of choosing the most appropriate for each case. As discussed in the paper [5], GAs better handle non-linearity of aggregation formulae, and better scale up when the number of concrete services for each abstract service is high. For simpler cases, integer programming is quicker.

The fitness function may need 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, a fitness function for a solution g can be defined as follows:

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

where $w_1, \ldots, w_4$ are real, positive weighting factors. As mentioned above, some of the variables could be constrained and in this case the best solution could be found at the cost of a more expensive search.

```
Q_EST ← Estimated overall QoS;
Q_COS ← QoS parameter upper bound;
Q_ACT ← Q_EST;
Q_TOT ← 0 ;
NTH ← replanning threshold;
QTH ← single node 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)/Q_EST > NTH
            or Q_ACT > Q_COS then
                triggerReplan(INNER,
                Q_TOT) ;
            end
            for j ← 1 to k' do
                visit(INNER);
                if (Q_ACT − Q_EST)/Q_EST >
                NTH or Q_ACT > Q_COS
                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)/Q_EST > NTH
            or Q_ACT > Q_COS then
                triggerReplan(INNER,
                Q_TOT)
            end
            visit(INNER);
        case sequence
            foreach Node INNER in
            sequence do
                visit(Node);
                if (Q_ACT − Q_EST)/Q_EST >
                NTH or Q_ACT > Q_COS
                then
                    Q_TOT ← Q_TOT −
                    ActQoS(INNER) ;
                    triggerReplan(INNER,
                    Q_TOT);
                end
            end
        case flow
            foreach Node CHILD in flow
            do
                flow(Node);
            end
        case invocation
            INNER ← node
            Q_INNER ← QoS(node);
            Execute(node);
            Q_INNERACT ← ActQoS(node);
            if (Q_INNERACT −
            Q_INNER)/Q_INNER > QTH
            then
                triggerReplan(INNER,
                Q_TOT);
            end
            Q_ACT ←
            Q_ACT − Q_INNER + Q_INNERACT;
            Q_TOT ← Q_TOT + Q_INNERACT ;
            if (Q_ACT − Q_EST)/Q_EST > NTH
            or Q_ACT > Q_COS then
                triggerReplan(INNER, Q_TOT);
            end
        end
    end
end function
```

Figure 1. Re-planning triggering algorithm

4 Replanning

This section describes how, during a composite service execution, replanning is triggered by refining the QoS estimation whenever new information is available, how the workflow slice to be replanned is determined and finally how replanning will be performed.

4.1 Triggering Service Replanning

The algorithm presented in Figure 1, describes the proposed replanning triggering approach. The basic idea is to re-estimate the workflow QoS as soon as new information is available. The number of times a loop will be iterated, the branch followed in a conditional node, or the QoS values measured when a service is invoked, permit to correct initial QoS estimates, which are updated on the unlooped workflow. Whenever this new estimate indicates a large deviation from the initial one and, above all, that a risk for SLA violation, services that still remain to be executed must be replanned to try to "reduce the damage".

The algorithm is described for the cost QoS attribute. For response time the algorithm is quite similar (being it also an additive attribute), except for the *flow* node, for which differences are discussed. For multiplicative QoS attributes, the algorithm remains the same, except that Q_{ACT} and Q_{TOT} are updates of performed multiplications and divisions instead of additions and subtractions. 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. Replanning is triggered if i) the actual cost goes beyond the estimated one over a given percentage; or, clearly, ii) if the actual cost violates the SLA.

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 replanning is triggered. Then, the *loop* inner node is visited k' times (or while the *loop* condition is *true*), triggering replanning 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 replanning is triggered each time the deviation of the actual QoS from the estimate is above the threshold.

For *flow* nodes, the overall actual cost is augmented while services are invoked in the different, parallel flow children. Instead, the response time is measured separately

```
NODE ← unlooped workflow root node;
LIST ← list of stopped nodes;
SLICE ← triggeringNode;
SLICE ←
computeSlice(NODE, LIST, SLICE) ;
exit();
begin function computeSlice(in
NODE, in LIST, out SLICE)
    CONTAINER ← SLICE.container;
    if !CONTAINER then
        return SLICE;
    end
    if (sizeOfList > 1) then
        FORK ← most external fork of first
        node of LIST;
        N ← # of children of FORK;
        newFORK ← create new fork;
        for  i ← 1 to i < N + 1 do
            CHILD ← i-th child of FORK;
            CLIST ←
            sublist(LIST, CHILD);
            CSLICE ← first node of
            CLIST;
            CSLICE ←
            computeSlice(CHILD, CLIST, CSLICE);
            newFORK.addChild(CSLICE);
        end
        SLICE ← newFORK;
        SLICE.next ← FORK.next;
        SLICE.container ←
        FORK.container;
    end
    else
        if (most external loop of SLICE)
        then
            SLICE ← unlooped sequence;
            CONTAINER ←
            SLICE.container;
            if (!CONTAINER) then
                return SLICE;
            end
        end
    end
    switch CONTAINER is type of do
        case sequence
            newSEQUENCE ← new
            sequence node ;
            TSLICE ← SLICE ;
            while (TSLICE) do
                newSEQUENCE.add(TSLICE);
                TSLICE ← TSLICE.next;
            end
            newSEQUENCE.container ←
            CONTAINER.container;
            newSEQUENCE.next ←
            CONTAINER.next;
            SLICE ← newSEQUENCE;
        case case statement
            SLICE.container ← container
            of Switch;
            SLICE.next ← next of Switch;
    end
end function
```

Figure 3. Slice computation algorithm

for each child. Thus, a response time constraint violation may be triggered on each different child. After all children have terminated their execution, the maximum response time is kept as the parent process actual response time.

Finally, the actual QoS values of each invocation are measured and the actual overall QoS is updated accordingly. Also, if the estimation on the single node has a strong deviation, above a threshold indicated by QTH in Figure 1, replanning is triggered as well.

4.2 Determining the replanning slice

The function *triggerReplan()* invoked in the algorithm of Figure 1 has as effect that the workflow execution is stopped and replanning is performed on a slice of the un-
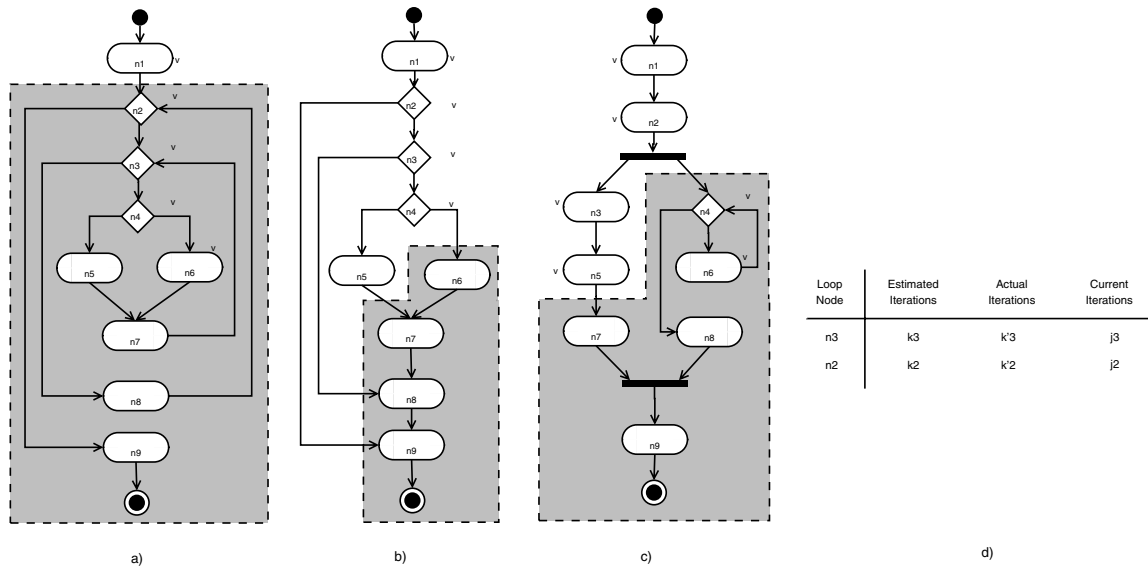
Figure 2. Replanning slice a) Node inside a loop b) Node inside a switch c) Node inside a flow

looped workflow (see Section 3.1), containing all the nodes that are still to be executed, with their weights. Note that the weight of each node is decreased by 1 at every invocation. The algorithm presented in Figure 3 describes how this slice is identified.

Given a node that, after its execution, triggers a replanning, the list of the nodes executed last before the workflow execution was stopped is considered. The slice to be replanned is recursively computed depending on the position of each of these nodes in the workflow control structure.

Let us assume that the triggering is not performed inside a child process (i.e., inside a *flow*). Initially, the slice refers to the triggering node n_p. Now, each invoke node of the original workflow has a reference to the most external *loop* to which it belongs, if any. This reference becomes a reference to a *sequence* node on the unlooped workflow. Also, each node has a reference to a next node and a reference to a container node, if they exist.

If the node n_p is inside a *loop* in the original workflow, then the slice is changed to the referred *sequence* node (corresponding to the most external *loop* in the original workflow), not yet *estimated to be completed*, i.e., whose inner invoke nodes have "estimated" weights greater than zero.

If the current slice is inside a *sequence* construct, then the slice is set to a new *sequence* node that collects the old slice, and its following nodes, of the *sequence* construct. As an effect, we have that only the non-executed nodes of the original sequence are considered (Figure 2-a, 2-d). If the node is part of a *case* statement of a *switch* construct, then the slice is set to have pointers to the next node and the container node of the *switch* construct. In fact, as the

decision for the *case* was taken already, the alternative *cases* will never be executed (Figure 2-b). Finally, the algorithm ends if the current slice is not included in any other control construct, as this will be the replanning slice.

In case there are more stopped nodes in the list, other then the replanning triggering node, then the most external *flow* construct containing these nodes is considered first for slice computation. The procedure described above is applied to its children threads to get their slices, which will be then composed again in a *flow* construct (Figure 2-c) to be the new slice, from where a second recursion of the algorithm above starts.

Given the replanning 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:

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

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

5 Architecture

The implementation of late-binding and service replanning mechanisms require a proper architecture that goes beyond the Service-Oriented-Architecture (SOA). Our approach stems from an architecture, also used by Mandell and McIlraith [13], to permit run-time discovery and composition. The binding between abstract and concrete services is realized by means of a proxy service. Instead
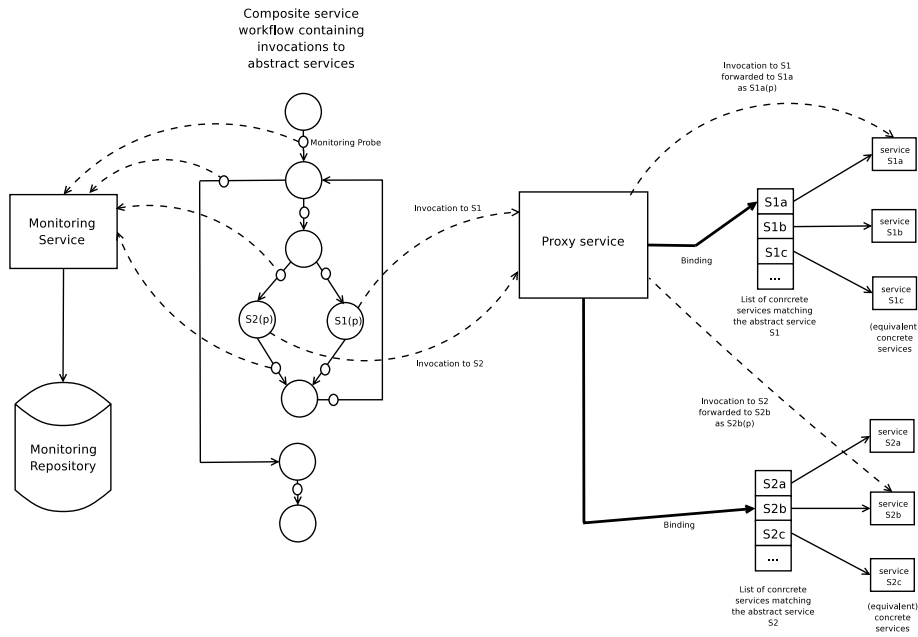
Figure 4. Dynamic binding and replanning via proxy services

of containing directly partner links to concrete services, BPEL4WS processes contain invocations to the proxy service, specifying i) the query, i.e., the semantic description of the desired functionality; and ii) the list of provided inputs and expected outputs.

The proxy service allows to retrieve the set of services matching the query and to choose the one that best contribute to fulfill the QoS requirements of the composite service and to meet the SLA. The invocation message is then forwarded to this service, eventually solving the type mappings of the input parameters through the underlying ontology. Before the execution of a composite service starts, the proxy service retrieves and caches the lists of concrete services that match the abstract services (represented in Figure 4). Right before the execution, in fact, it is necessary to perform a QoS-aware service composition, following the approach described in Section 3. Once a solution of the composition optimization problem is found, the binding is performed through a routing from abstract to concrete services. Whenever workflow replanning is triggered by the triggering algorithm explained in Section 4.1,

1. the workflow execution is stopped;

2. a workflow slice, i.e. the part of workflow still remaining to be executed, is determined;

3. new bindings are determined so to maximize the fitness function on the service slice that remains to be executed;

4. bindings are enacted by simply moving the bold links to different concrete services in the list; and, finally

5. the engine is re-started to continue the workflow execution.

The approach described above and the architecture present several advantages. Namely, there is no need to perform any change in the BPEL4WS language nor in the workflow engine to support the use of abstract services in the workflow. Also, when performing a replanning, the only intervention to be done is to change the bindings to concrete services, and then to restart the workflow execution.

The proposed architecture also encompasses monitoring facilities[1]. Briefly, all the needed information (QoS estimates, paths followed in the workflow) is collected by means of services invoked by probes inserted in the workflow using instrumentation (see Figure 4).

6 Empirical Study

We used numerical simulations to evaluate the proposed approaches for triggering and realizing workflow replanning during execution. The experiments were run on a Pentium IV with 1800 MHz processor, 640MB of RAM. The QoS values of semantically equivalent services were varying according to some Gaussian distribution function, and

---

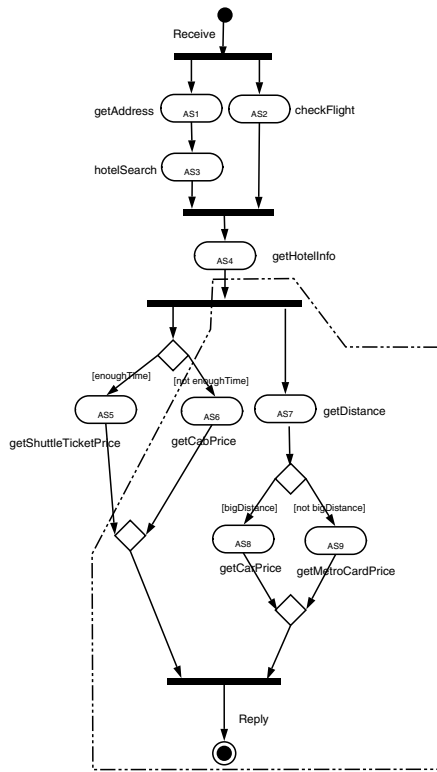[1]Out of scope of this paper, details can be found in the technical report [6].

Figure 5. TravelPlan process

Figure 6. CDMaker process

better response time and availability offers corresponded to higher costs.

For both experiments, we used an elitist GA (the best two individuals were kept alive over subsequent generations), a population of 50 individuals and 200 GA generations. The crossover probability was set to 0.7 while the mutation probability to 0.01.

For our experiments we took some realistic examples of service composition and reproduced situations for workflow replanning on them. Then we made several runs in order to evaluate the overall benefit of replanning on the resulting QoS. Figures 5 and 6 show two concrete examples of abstract service orchestrations. The first is concerned with a travel planning service, whose control structure includes *sequence*, *flow* and *switch* constructs. The process essentially describes a search for availability of a flight to, and accommodation in, a certain city, the nearest possible to some target place in that city, which could be, for example, a tourist attraction. The output of the service should also include information about the total cost of the travel. According to the arrival time of the flight and the latest possible hotel check-in, the cost of a cab rather then a shuttle ticket should be included. Also, two possible alternatives, either a car rental or a metro card, are offered according to the distance of the tourist attraction from the hotel. The
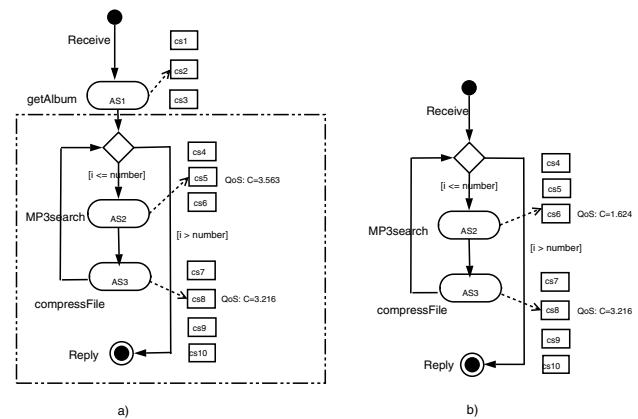
whole plan and its costs are then presented to the customer for approval and booking. The abstract services needed by the process are: at one side a tourist information service to provide the address of the target attraction, with hotel search, and a flight information service to plan the travel at the other side. Then services, such as car rental or transport information are used to inquire about additional costs. For this example, we have assumed that, with probability 0.75, a shuttle would be taken to reach the hotel, while, less probably, i.e., 0.6, (a convenient) hotel would be close to the attraction. Also, we suppose that a customer has required that the response time of the whole service should not be above 14 seconds and that the cost for its usage should be minimal. The second example is a very simple CD composition process, including *sequence* and *loop* constructs. Given the title of an album, it describes a search for, and compression of, the MP3 files composing it. Here, we have assumed that any album would contain about 20 songs on average, while a customer has required that the overall cost be less then 14 dollars and the response time be minimal.

6.1 Empirical Study Results

These examples have been thought to highlight the three possible causes for replanning, that is: deviation of the actual QoS of the candidate services, more accurate re-estimations of the overall QoS after decision points of the workflow, and under-estimation of the number of iterations for *loops*. Obviously, a consequence of these changes could be the violation of some global constraints. A selection of representative results from our experiments is shown in Table 1.

The first row of the table corresponds to the scenario where a high deviation of the actual response time (T) from the estimations occurred for some of the composing services. Consequently, a time constraint violation was foreseen, as

Process	Initial Est QoS		Replanning QoS		Final QoS	
	T [s]	C	T [s]	C	T [s]	C
TravelPlan	13.86	1.9	22.1	1.87	13.9	1.7
TravelPlan	13.86	1.89	24.62	1.97	14.81	1.82
TravelPlan	12.45	1.96	22.8	2.03	13.39	1.97
CDMaker	1927.65	13.8	2021.88	14.46	2567.75	10.39
CDMaker	1927.65	13.8	-	-	2471.65	10.26

Table 1. Initial, replanning and final QoS values

Figure 8. CDMaker results

the second column shows, before the execution of the shuttle ticket service, and a replanning was applied, which allowed for the time constraint to be met. It should be noted that the final cost (C) was even less then the one initially estimated as optimal because the aggregation formula for the branch construct had resulted in an over-estimation of the cost and time values. The final response time value also accounts for the time spent by the replanning algorithm, which was of about 1 sec.

The second and third rows of the table refer to the case where the less likely branch was executed, i.e., the cab choice, so that the initial QoS estimation was not at all accurate and, again, the time constraint would have been violated. The workflow slice to which replanning was applied in both cases is shown as a dashed area in Figure 5. However, in the first case, the replanning did not solve the problem, as the final response time was still above the limit. While an acceptable solution was found by the optimization algorithm, the additional time of the replanning algorithm (i.e., 1 sec.) caused the violation of the constraint. Thus, in general, the cost of the replanning process should be estimated and a decision on whether or not trying to do replanning should be taken accordingly. Instead, the third experiment shows a successful replanning with respect to the time constraint, but with a sub-optimal final cost.
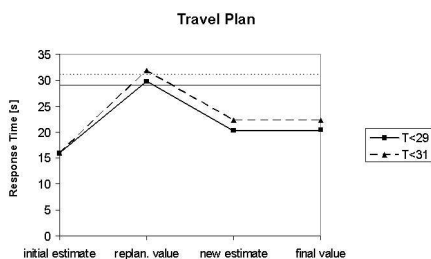

Figure 7. TravelPlan results

The forth row of the table reports the experiment where a replanning was performed due to an under-estimation on the number of songs, i.e., 20 instead of 21. This would have lead to the violation of the cost constraint. The replanning was triggered before entering the *loop*, when the new cost estimation 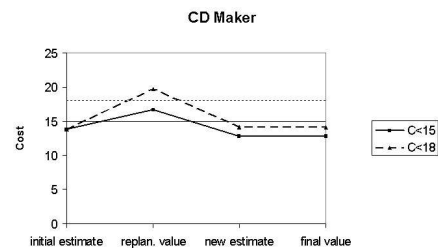was computed as soon as the actual number of songs of the requested album was known. The workflow slice and the new bindings are shown in Figure 6-b). The fifth row shows the results from the situation where the song download service selected turned not to be available. The replanning lead to a much higher time, i.e., 2471 sec. instead of 1927, but with a save in cost.

Figures 7 and 8 report the evaluation of the effects of replanning on the final QoS, considering two possible constraints for each process. We made about 1100 runs of each workflow, where the actual response time of the chosen services was varying according to a standard deviation up to 30%, and considered the mean QoS values. The essential observation is that, in both examples, replanning was successful on average, i.e., the constraints were finally met. Further, the results we obtained indicated a loss in cost of about 7% after replanning in the case of the TravelPlan process and up to 44% in the response time in the case of the CDMaker process.

7 Conclusions

This paper presented an approach to replan, at execution time, the binding between a composite service and its invoked services. During service execution, the QoS attributes are monitored and replanning is triggered if the SLA is violated, if there is a high likelihood to violate it, or if deviation between estimated and actual QoS is very high. The triggering algorithm proposed permits an early activation of the replanning, so to prevent risks as soon as possible. Once replanning is triggered, the workflow slice that (possibly) still remains to be executed is determined and rebound to concrete services. Finally, the workflow execution is restarted. A proxy architecture is used to enable dynamic binding and replanning.

The case studies showed the feasibility of the approach. During service execution, if the QoS constraints were violated or if a service was not available, the replanning was able to determine new bindings that permitted to complete the service execution while satisfying the SLA. For services requiring a quick response, it is necessary to pursue a trade-off between a possible improvement gained with replanning

and the replanning overhead on response-time. Future work is devoted to thoroughly study the above mentioned trade-off issues, as well as to perform an *in field* evaluation of the proposed approach on complex composite services.

8 Acknowledgments

References

[1] R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint driven web service composition in METEOR-S. In *Proc. IEEE International Conference on Services Computing (SCC'04)*, pages 23–30, Shanghai, China, Sept. 2004.

[2] 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/*.

[3] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *Proc. 2nd International Conference on Service Oriented Computing (ICSOC'04)*, pages 193–202, New York, USA, November 2004. ACM.

[4] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. A lightweight approach for QoS-aware service composition. In *Proc. 2nd International Conference on Service Oriented Computing (ICSOC'04) - short papers*, New York, USA, Nov. 2004. IBM Technical Report.

[5] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An approach for QoS-aware service composition based on genetic algorithms. In *To appear in Proc. of Genetic and Evolutionary Computation Conference (GECCO 2005)*, Washington, DC, USA, June 2005.

[6] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. QoS-aware replanning of composite web services. Technical Report TR 001-05, RCOST, University of Sannio, *http://www.rcost.unisannio.it/mdipenta/papers/rcost-tr001-05.ps.gz*, 2005.

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

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

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

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

[11] U. Greiner and E. Rahm. 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.

[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] D. Mandell and S. McIlraith. Adapting BPEL4WS for the semantic web: The bottom-up approach to web service interoperation. In *The SemanticWeb - ISWC 2003*, volume 2870/2003, pages 227–241. LNCS, 2003.

[14] G. Spanoudakis and K. Mahbub. Requirements monitoring for service-based systems: Towards a framework based on event calculus. In *Proc. 19th International Conference on Automated Software Engineering (ASE'04)*, pages 379–384, Linz, Austria, Sept. 2004. IEEE.

[15] V. Tosic, B. Pagurek, and K. Patel. 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] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, USA, 1989.

[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.

IEEE COMPUTER SOCIETY