**University of Peloponnese**
**Department of Informatics and Telecommunications**
**Software and Database Systems Laboratory**

# Preprocessor transformations and adaptation operations for improving QoS delivered by WS-BPEL scenario adaptation through service execution parallelization

Technical Report TR-15002

Dionisis Margaris, Costas Vassilakis, Panagiotis Georgiadis
margaris@di.uoa.gr, costas@uop.gr, p.georgiadis@uoa.gr

September, 2015
Tripoli, Greece

September, 2015

# 1. Introduction

In this technical report, we present the preprocessor transformations and adaptation operations for improving QoS delivered by WS-BPEL scenario adaptation through service execution parallelization. The preprocessor transformations aim at restructuring the parallelizable operations to be executed in parallel, even though the WS-BPEL scenario designer has specified sequential execution. Exploitation of parallelism can serve as an aid to the adaptation process by broadening the set of alternatives available to the adaptation mechanism: since parallelism reduces the overall execution time, in the parallelized scenario it is possible to choose operations with higher response times but better values in other QoS dimensions (e.g. cost), with the composition respecting the overall WS-BPEL scenario execution time limits, but scoring higher in the other dimensions (e.g. having lower costs).

The preprocessor also caters for making the scenario *adaptation ready*, i.e. inserting appropriate code to pass the data required to perform the adaptation to a newly introduced *adaptation layer* and redirect service invocations to this layer.

The adaptation layer is responsible for computing the optimal execution plan, i.e. find the service implementations that best suit the user-specified QoS policy and invoke these implementations to realize the functionalities specified in the WS-BPEL scenario.

Since the WS-BPEL scenario is restructured, the presented approach does not strictly follow the horizontal adaptation paradigm [1], however the changes to the composition logic are limited and performed in a fashion that enables the exploitation of exception handlers provided by the scenario designer, which may have been elaborately crafted to correspond to the particularities of the business process modeled by the WS-BPEL scenario.

The rest of this report is organized as follows: in section 2, we present the underlying foundations regarding QoS. Section 3 lists the preprocessor transformations introduced to enable QoS-based adaptation. In section 4, we present the algorithm used for parallelization detection and the relevant transformations employed by the preprocessor. Section 5 details the execution architecture and elaborates on the adaptation operations. Finally, section 6 concludes the report.

## 2. QoS concepts and collaborative filtering foundations

QoS is generally defined in terms of attributes corresponding to non-functional aspects of services, with typical attributes being response time, availability, price, reputation, security and so forth [2]. In this paper, we will limit our discussion to attributes *response time* (rt), *availability* (av) and *cost* (c), for brevity reasons, adopting their definitions from [3]. This limitation does not lead to loss of generality since the extension of the proposed algorithm to include more QoS attributes is straightforward. Taking the above into account, each functionality implementation (realized as a *service operation*) considered in the adaptation process has a known QoS vector $QoS_S=(rt_S, av_S, c_S)$ which is recorded in an appropriate repository (e.g. METEOR-S [4] or WSMO[5]). The same repository should also provide information regarding which operations are equivalent. Within a WS-BPEL scenario, individual functionalities are composed into sequential or parallel flows to implement the business process. Considering the QoS parameters of the individual functionalities invoked and the type of their compositions (sequential or parallel), it is possible to compute the QoS value of the overall composition using the formulas shown in Table I [8]. As we can see from table 1, the response time of a sequential composition is equal to the sum of its components' response time, while the response time of a parallel composition is equal to the maximum value. This difference is important in the context of this work, since the exploitation of available parallelization can lead to reduction of the overall response time.

**Table 1: QoS of composite services**

| | QoS attribute | | |
|---|---|---|---|
| | response time | cost | availability |
| Sequential composition | $\sum_{i=1}^{n} rt_i$ | $\sum_{i=1}^{n} c_i$ | $\prod_{i=1}^{n} av_i$ |
| Parallel composition | $\max_i rt_i$ | $\sum_{i=1}^{n} c_i$ | $\prod_{i=1}^{n} av_i$ |

In the context of adaptation, selection of the concrete service that will realize some functionality is typically driven by parameters specifying the upper and lower bounds for each QoS attribute. QoS bounds may either be defined as *global constraints* (i.e. express the desired values for the whole WS-BPEL scenario) or as *local constraints* (each such constraint expresses the desired values for a particular service invocation). When adaptation problems need to address global constraints performance is poor [6], therefore either local constraints are directly used (e.g. [7][8]) or methods for mapping global constraints to local constraints are employed (e,g, [6]). Complementary to the QoS bounds, a *weight* is assigned to each QoS attribute, indicating how important each QoS attribute is considered by the designer in the context of the particular business process modeled by the scenario. Weights always apply to the whole composition, rather than to individual services, since they reflect the perceived importance of each

QoS attribute dimension on the process as a whole, and not its constituent parts [9]. In the proposed algorithm, the QoS specifications for a service within the WS-BPEL scenario may include an upper bound and a lower bound for each QoS attribute, i.e. for service $s_j$ included in a WS-BPEL scenario, the designer formulates two vectors MINj($min_{rt,j}$, $min_{av,j}$, $min_{c,j}$) and $MAX_j$($max_{rt,j}$, $max_{av,j}$, $max_{c,j}$). Additionally the designer formulates a weight vector W = ($rt_w$, $av_w$, $c_w$), indicating how important each QoS attribute is considered by the designer in the context of the particular operation invocation. The values of the QoS attributes are assumed to be expressed in a “larger values are better” setup, e.g. a service having *cost* = *6* means that that it is *cheaper* than a service having *cost* = *4* [8].

## 3. Transformations to enable QoS-based adaptation

In order to enable QoS-based adaptation, the WS-BPEL scenario designer must provide the relevant QoS parameters that apply to each invocation. In this work, we adopt the approach used in [8], according to which:

1. for each *invoke* construct, the designer should provide the optional attribute name, assigning distinct names to the invoke constructs,
2. for the *invoke* construct having the name attribute equal to *invX*, the designer should use the WS-BPEL variables *QoSmax_invX* and *QoSmin_invX* which designate the QoS bounds for the particular invocation and
3. the designer should provide the WS-BPEL variable QoS_weight to specify the weight of each QoS attribute (recall that Weights always apply to the whole composition, rather than to individual services).
4. The WS-BPEL designer may set the values for variables QoSmax_invX and QoSmin_invX and QoS_weight after examining the values of user-provided parameters, tailoring thus the QoS specification to the preferences of the user invoking the scenario. Fig. 2 shows an example of maximum QoS bounds specification for an operation invocation.

Figure 1 presents an example of QoS parameter specification.

```
<assign>
        <copy>
                <from><literal>7</literal></from>
                <to variable="QoSmax_getQuote" part="respTime"/>
        </copy>
        <copy>
                <from><literal>4</literal></from>
                <to variable="QoSmax_getQuote" part="cost"/>
        </copy>
</assign>
<invoke name="getQuote" partnerLink="hotel1" operation="getQuote"
     outputVariable="quote" inputVariable="roomTypeAndPeriod" />
```

**Figure 1. QoS specification in the WS-BPEL scenario**

When the WS-BPEL scenario designer has crafted the scenario, including the QoS parameter specification, it submits it to the preprocessor, which is responsible for:

1. arranging for connecting to the adaptation layer and retrieving a session id; the session id is used to distinguish among invocations to the middleware pertaining to individual executions of WS-BPEL scenarios performed within the WS-BPEL orchestrator. This is accomplished through an invocation to the *getSessionId* operation provided by the adaptation layer, which is inserted at the beginning of the scenario.
2. providing for passing to the middleware all appropriate information regarding the operation invocations performed within the WS-BPEL scenario, their structure (parallel vs. sequential), the affinity groups, the QoS weights for the scenario (as set using the *QoS_weight* variable) and the QoS bounds for the different invocations (as specified through the *QoSmin_invX* and *QoSmax_invX* variables). This is realized

through an invocation to the *bpelScenarioInfo* operation of the adaptation layer (c.f. section 5). This information will be used by the adaptation layer to formulate the execution plan, which is inserted before the first *invoke* operation within the scenario.

3. modifying individual operation invocations so that they are redirected to the adaptation layer, instead of the actual web service operation initially specified in the WS-BPEL scenario. The adaptation layer will then forward the invocation to the appropriate implementation, according to the formulated execution plan. This is accomplished by modifying the *partnerlink* specification. The preprocessor arranges so that the *headers* of each invocation contain the session identifier, to enable the adaptation layer to distinguish the particular WS-BPEL scenario execution in the context of which this invocation is performed.
4. inserting an invocation to the *releaseSession* operation of the adaptation layer as the last activity of the WS-BPEL scenario, to allow for release of resources dedicated to the particular execution.

When the modified scenario is generated, it is deployed to the WS-BPEL orchestrator and made available for invocation.

## 4. The parallelization algorithm and pre-processor transformations

Although WS-BPEL provides the mechanisms to designate parallel execution of operation invocations, WS-BPEL scenario designers may not fully exploit the potential for arranging operations into parallel execution structures, similarly to the case that programmers typically write their programs in a single-threaded fashion [10][11]. This is owing to the fact that execution parallelization is a laborious task and WS-BPEL designers mostly focus on accurately realizing the business logic behind the WS-BPEL scenario they create, rather than pursue execution time optimizations. To this end, a tool that would be able to detect and exploit the parallelization opportunities available in WS-BPEL scenarios, would deliver the benefits of parallel execution without placing the parallelization burden on WS-BPEL scenario designers.

In our approach, WS-BPEL scenario parallelization is undertaken by a preprocessor, which preprocesses the scenario before it is deployed to the WS-BPEL execution engine; parallelization can be driven by *data flow and dependence analysis* used in instruction-level parallelism [11], supplemented with techniques aiming to address the particularities of WS-BPEL execution (exceptions, compensations and side-effects) and aspects related to the QoS of the invoked services. The criteria for identifying invocations that can be executed in parallel are detailed in the following paragraphs; in these paragraphs, we will consider that operation invocation *$op_1$* appears in the WS-BPEL scenario before operation invocation *$op_2$*.

WS-BPEL provides two main control flow structures for composing operation invocations into business processes, namely the *sequence* element which arranges for sequential execution of the invocations it contains, and the *flow* element which arranges for parallel execution of the invocations it contains.

1. Two operation invocations *$op_1$* and *$op_2$* can be scheduled to run in parallel, if they have been designated to be executed in parallel in the original WS-BPEL scenario (as crafted by the WS-BPEL designer).
2. Operations *$op_1$* and *$op_2$* are analyzed for existence of data dependence between them. Four types of dependences may exist between operation invocations [10][10]:
   a. *True (or flow) dependence: $op_2$* uses a parameter that is either directly returned by *$op_1$* as its result, or computed using the result of *$op_1$*. In this case, clearly *$op_2$* cannot be executed before *$op_1$* concludes its execution, since the value of some input parameter of *$op_2$* is yet unknown.
   b. *Anti-dependence: $op_2$* modifies a variable V by assigning to it its result value, and the same variable V is used as an input parameter to *$op_1$*. In this case the operations cannot be executed in parallel because if *$op_2$* concludes before *$op_1$* is processed, variable V will be modified and thus the parameter passed to *$op_1$* will not have the correct value.
   c. *Output dependence:* both operations store their result to the same variable V. In a sequential execution, after *$op_2$* has concluded the value returned by *$op_2$* will be stored in V. If however *$op_1$* and *$op_2$* are scheduled to be executed in parallel, the value of the operation invocation that concluded last will be finally stored in V; therefore in the case that *$op_1$* concludes after *$op_2$*, the execution result will be erroneous.
   d. *Input dependence:* both operations share an input parameter.

If true dependence, anti-dependence or output dependence is identified between two invocations, then they cannot be scheduled to run in parallel; input dependence does not preclude parallel execution of the involved operation invocations [11].

3. Operations *op₁* and *op₂* cannot be scheduled to be executed in parallel if the invocation of op2 either (a) incurs some cost or (b) has some side-effect (e.g. creating a session, booking a ticket etc.)[5][4], unless the results of the invocation of *op2* are undoable, through a compensation handler provided in the WS-BPEL scenario. This criterion targets the case in which an exception is raised during the invocation of *op₁*: if *op₁* failed due to an exception and *op₂* were scheduled to run after *op₁*, then *op₂* would not be executed at all (and thus the associated cost would not be incurred) since either the scenario would be terminated or control would be transferred to the appropriate fault handler. If however the invocations were executed in parallel, *op₂* would run and therefore the associated cost would be incurred, which is undesirable; nevertheless, if the WS-BPEL scenario included a compensation handler for *op₂* it would be possible to execute the services in parallel and provide a fault handler which would arrange for invoking *op₂*'s compensation handler to recuperate the cost stemming from the invocation of *op₂*.
4. Two operation invocations *op₁* and *op₂* cannot be scheduled to run in parallel if *op₁* creates a side-effect (e.g. creation of a session/login, sending goods) and *op₂* depends on the existence of the side-effect.
5. In all other cases, *op₁* and *op₂* are able to run in parallel.

At the current development stage, only invocations belonging either to (i) the same *sequence* structured activity or (ii) nested *sequence* and *flow* activities, with no intervening conditional (*if)* or repetitive (*while, repeatUntil, foreach*) structured activities are considered. The development of the necessary techniques for control dependence checking and loop unrolling [11] to foster parallelization among invocations nested in different structured activities are part of our future work. Criteria 1 and 2 in the above list, as well as the existence of the compensation handler stated in criterion 3 can be directly evaluated by analyzing the WS-BPEL scenario. The existence of a cost associated with the invocation of a service mentioned in criterion 3 can be directly retrieved from the service repository (e.g. METEOR-S [4]). Finally, side-effects either created by the service (criteria 3 and 4) or needed by the service (criterion 4) can be retrieved from a repository such as WSMO [5]. Obviously, instead of using two distinct repositories, the information needed may be stored into a single, comprehensive repository; in our implementation we have used a unified repository. When two (or more) operation invocations that were initially designated to run sequentially are restructured to run in parallel, their QoS limits regarding the response time can be relaxed. For instance, consider the case that a WS-BPEL scenario comprises of operation invocations $O_1$ and $O_2$ that are designated to be executed sequentially, with an upper bound on the response time 3 and 7 seconds, respectively; therefore the upper bound on the scenario execution time would be 10 seconds. If the scenario is restructured so that $O_1$ and $O_2$ are executed in parallel, then the upper bound of both operations' execution time can be set to 10 seconds, a setting which provides guarantees that the WS-BPEL scenario will conclude in 10 seconds, but it also broadens the pool of operations that the adaptation mechanism can choose from to realize $O_1$ and $O_2$. Generalizing, if operations $O_1$, $O_2$, …, $O_n$ were initially restructured to run sequentially and are restructured to run in parallel, then the upper bounds of their response time are set to $\sum_{i=1}^{n} U_i^{RT}$, where $U_i^{RT}$ is the initially set upper bound for the run time of operation $O_i$.

Taking the above criteria into account, the preprocessor analyzes the structure of the WS-BPEL scenario and determines which invocations can be parallelized. Operations within a sequential structure that are found to be parallelizable, are organized in a *flow* construct. Consider for instance the WS-BPEL scenario fragment illustrated in Figure 2 (for conciseness purposes, only the relevant parts/attributes of the scenario are shown), while the corresponding graphical representation of the same scenario is illustrated in Figure 3. This scenario fragment arranges for getting a quote for a hotel room and booking it, renting a car and then paying for both items. The invocations are arranged in a sequential structure, however in this sequence, we can identify that invocations to *getRoomQuote* and *rentCar* may proceed in parallel, since they (a) have no interdependencies and (b) *rentCar* has an associated cost (the cost of invoking the service e.g. a commission; the actual fee for renting the car is paid later through *finalizeReservation*) and a side-effect (recording the car rental in the service provider's database), however a compensation handler exists, therefore any incurred costs and/or side effects are undoable by invoking this compensation handler.

```
<sequence>
  <invoke operation="getRoomQuote" outputVariable= "quote"
    inputVariable="roomTypeAndPeriod" name= "getQuote"/>
  <invoke operation="reserveRoom" inputVariable= "quote"
    outputVariable="reservationInfo" name= "reserveRoom"/>
  <invoke operation="rentCar" inputVariable= "carTypeAnd Period"
    outputVariable="carRentalInfo" name="rentCar">
      <compensationHandler>
          <invoke operation="cancelRentCar" inputVariable= "carRentalInfo">
     </compensationHandler>
  </invoke>
  <assign>
    <copy>
      <from expression="$quote.price + $rentalInfo.price" />
      <to variable="paymentInfo" part="amount" />
    </copy>
    <copy>
      <from variable="creditCardInfo" />
      <to variable="paymentInfo" part="creditCard" />
    </copy>
  </assign>
  <invoke operation="finalizeReservation" name="doReserve"
      inputVariable="paymentInfo" outputVariable="receipt" />
</sequence>
```

**Figure 2: Excerpt of sequential WS-BPEL scenario**

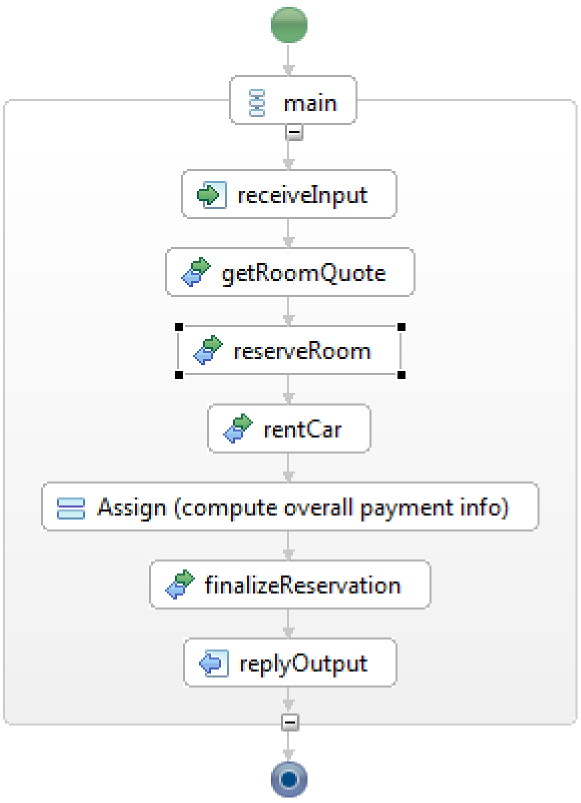

**Figure 3: Graphical representation of the sequential WS-BPEL scenario**

Contrary, the invocation to *reserveRoom* must strictly be performed after the invocation to *getRoomQuote* has concluded, since *reserveRoom* uses variable *quote* as its input, which is produced by *getRoomQuote* (direct dependency). Similarly the invocation to *finalizeReservation* should follow the conclusion of both *getRoomQuote* and *rentCar* because variable *paymentInfo* (the input of *finalizeReservation*) is indirectly dependent on the output of *rentCar* (variable *carRentalInfo*) and *getRoomQuote* (variable *quote*), since the *copy* construct in Figure 2 uses the *carRentalInfo* and *quote* variables to calculate the value to be assigned to (a part of) *reserveRoom*'s input *paymentInfo.* A more subtle dependence exists between services *reserveRoom* and *finalizeReservation*, which cannot be determined by analyzing the scenario code alone: *finalizeReservation* can be performed only when a room has been reserved; this is a required side-effect for operation *finalizeReservation*, and this side-effect is produced by operation *reserveRoom*, hence *reserveRoom* must have concluded before *finalizeReservation* is invoked. The information regarding the side effects is drawn by the preprocessor from the service repository, where it is recorded that *reserveRoom* creates the side effect and *finalizeReservation* depends on it. After the dependence analysis results have been computed, the WS-BPEL scenario is restructured to accommodate the available parallelism, as shown in Fig. 4 (only the first part which has changed is shown; the part that has remained intact has been omitted for brevity), while the corresponding graphical representation is shown in Fig. 5. Regarding the upper response time bound of the services that are restructured to be executed in parallel, the preprocessor arranges for designating that the upper response time bound of *each of the invocations* to *getRoomQuote* and *rentCar* is equal to the sum of the individual invocations, with the sum being again normalized to the [1, 10] scale.

```
<sequence>
  <flow>
    <invoke operation="getRoomQuote" inputVariable= "roomTypeAndPeriod"
            outputVariable="quote" name="getQuote" >
      <compensationHandler>
              <invoke operation="cancelRentCar" inputVariable="carRentalInfo"/>
      </compensationHandler>
    </invoke>
    <invoke operation="rentCar" inputVariable="carTypeAndPeriod"
            outputVariable="carRentalInfo" name="rentCar" />
  </flow>
  <sequence>
          <invoke operation="reserveRoom" inputVariable="quote"
                  outputVariable="reservationInfo" name="reserveRoom"/>
      <assign>
      …
  </sequence>
</sequence>
```

**Figure 4: Excerpt of transformed (parallelized) WS-BPEL scenario**

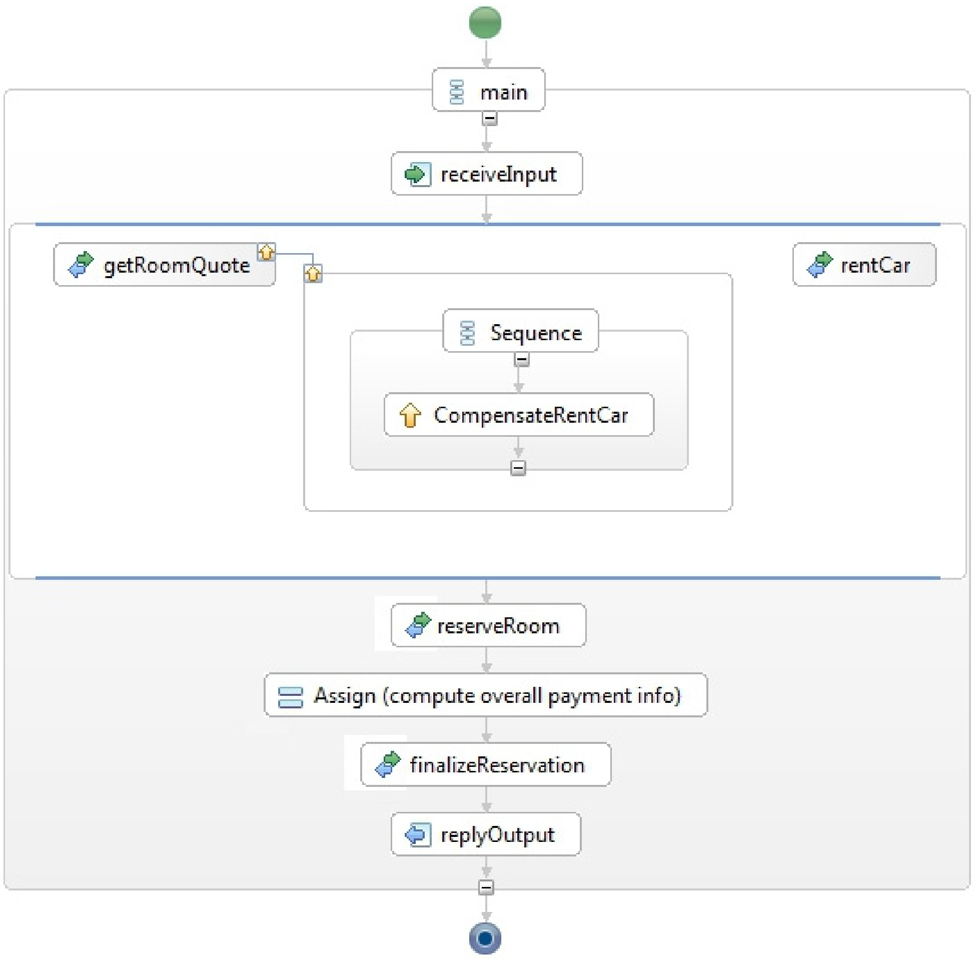

**Figure 5: Graphical representation of transformed (parallelized) WS-BPEL scenario**

An issue that needs to be addressed regarding these transformations, is the fact that one of the criteria for determining whether operations are parallelizable, and in particular the criterion examining whether the involved service incurs some cost (criterion 3 above) is based on the service repository contents. However, the service repository contents regarding this dimension may change i.e. either (a) a provider may begin charging a previously free service, hence operation invocations that were previously parallelizable cease to be so, or (b) a provider may stop charging a previously non-free service, in which case two invocations that were previously non-parallelizable can now be scheduled to be executed in parallel. A similar issue exists for side-effect creation and requirement. To tackle this issue, the preprocessor takes the following two measures:

1. to guard against selecting a non cost-free service, the preprocessor arranges for setting the upper bound for the cost of the particular invocation to zero (normalized to the [1, 10] scale).
2. in all cases, the preprocessor establishes *redeployment triggers*, which consist of monitoring updates to the repository that fall into the previously described categories (cost, side-effect creation and side-effect requirement). When such a change is detected, the affected WS-BPEL scenarios are identified and a preprocessing and redeployment action is initiated for them, so that the preprocessor takes into account the updated contents of the repository (c.f. section 5).

## 5. Adaptation architecture and operations

The adaptation architecture, illustrated in fig. 6, adds to the standard SOA architecture three additional modules, the *preprocessor*, the *adaptation layer* and the *redeployment triggers*.

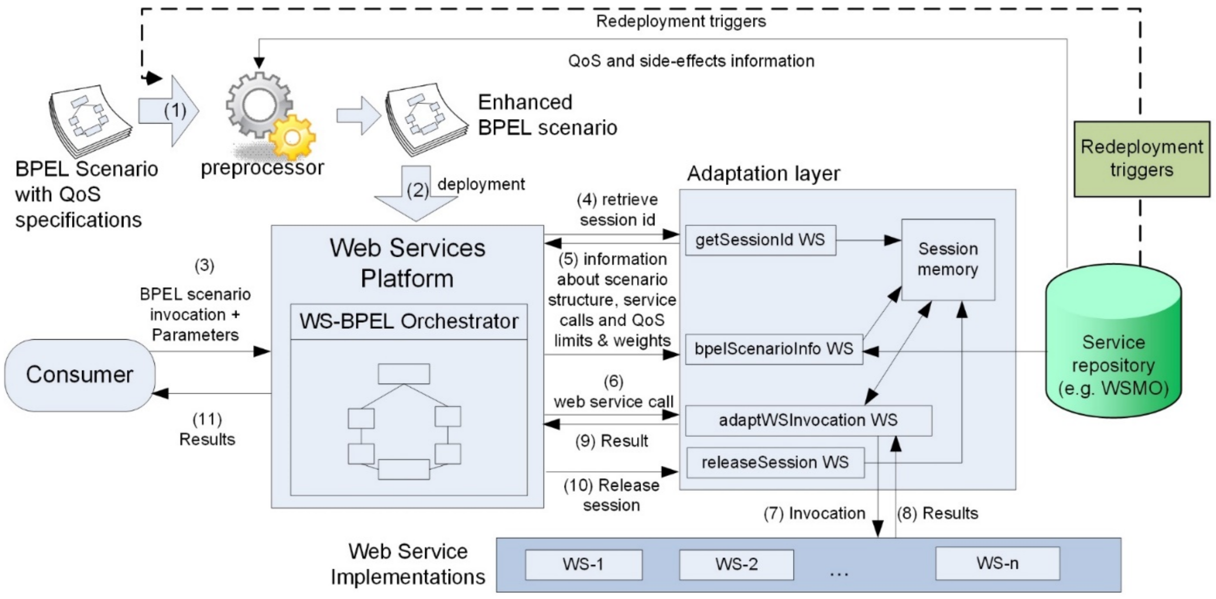

**Figure 6: The adaptation architecture**

The *preprocessor* performs transformations on the original WS-BPEL scenario by (a) restructuring service invocations to be performed in parallel under the conditions described in section 4 above (b) arranging for passing appropriate data to the adaptation layer to drive the adaptation and (c) redirecting service invocations to the adaptation layer, so as to be sent to the service implementations best matching the QoS specifications. The preprocessing step produces an *enhanced WS-BPEL scenario*, which is then deployed to the WS-BPEL orchestrator. The transformations performed by the preprocessor have been discussed in sections 3 and 4.

*Redeployment triggers* periodically whether changes have occurred to the data within the repository on the basis of which decisions regarding parallelization capability have been made. This includes (a) cost of services (b) creation of side-effects by services and (c) requirement of side-effects by services. When such a modification is expected, the affected WS-BPEL scenarios are identified and, for each of them, the preprocessor is invoked to perform the applicable transformations, considering the updated service repository contents. Redeployment of the new preprocessed file is performed without affecting currently running instances of the scenario, exploiting the hot redeployment feature of contemporary WS-BPEL orchestrators (e.g. [12]).

The *adaptation layer* intervenes between the WS-BPEL orchestrator and the actual web service implementations, arranging for formulating the WS-BPEL scenario *execution plan*, i.e. to choose for each operation invocation designated in the executing scenario the most appropriate implementation with respect to the QoS policy defined for the current execution. The adaptation layer uses integer programming to determine the optimal execution plan for the specific WS-BPEL scenario execution, subject to the

QoS policy specified by the consumer, and stores this execution plan to the *session memory*. Subsequently intercepts service invocations performed in the context of the WS-BPEL scenario execution and redirects them to the chosen service implementations. In more detail the execution of a WS-BPEL scenario is performed as follows:

## *5.1 Execution plan computation*

When the WS-BPEL scenario commences execution, it will first retrieve from the adaptation layer the session identifier, and afterwards it will invoke the *bpelScenarioInfo* operation of the adaptation layer to provide to it all the information required for the formulation of the execution plan. The adaptation layer will then compute the execution plan as follows:

1. For each operation invocation, the adaptation layer retrieves from the repository those operations that are equivalent to the operation being invoked and satisfy the QoS bounds for the particular invocation. Thus, the set of possible operation assignments for operation $op_i$ POA(op$_i$)={op$_{i,1}$, op$_{i,2}$, …, op$_{i,x}$} is formulated as follows: $POA(op_i) = \{ op \in Repository{:}\ op\ equivalent\ op_i \wedge [(min_{rt,i} \leq rt_s \leq max_{rt,i}) \wedge (min_{c,i} \leq c_s \leq max_{c,i}) \wedge (min_{rel,i} \leq rel_s \leq max_{rel,i})]\}$; recall that the min and max bounds per operation have been received as input to the *bpelScenarioInfo* operation.
2. The adaptation layer formulates an integer programming (IP) problem, in order to produce the execution plan, i.e. a set of concrete operation assignments EP={cop$_1$, cop$_2$, …, cop$_n$} where cop$_i$∈POA(op$_i$), such that this set of assignment best matches the QoS parameters specified by the user. To this end, the candidate *i* for realizing operation $op_j$ (i.e. the ith element in POA(op$_j$)) is assigned a utility value calculated by the following function [6]:

$$U(op_{j,i}) = \sum_{k=1}^{3} \frac{Q_{max}(j,k) - q_k(o_{j,i})}{Q_{max'}(k) - Q_{min'}(k)} * w_k$$

where $q_k(o_{j,i})$ is the value of the kth QoS attribute of operation $o_{j,i}$ (the first QoS attribute corresponds to response time, the second one to cost and the third one to availability), w$_k$ being the weight assigned to the kth QoS attribute, $Q_{max}(j,k) = \max_{s \in POA(j)} q_k(s)$ [i.e. the maximum value of QoS attribute *k* among possible concrete operation assignments for operation *j*], and $Q_{max'}(k)$ [resp. $Q_{min'}(k)$] being the overall maximum (resp. minimum) value of QoS attribute *k* within the repository. Note that services with high QoS values have low utility function values. Given the utility function, the computation of the *m-best solutions* can be formulated as an integer programming optimization problem as follows: minimize the overall utility value given by:

$$OUV_{QoS} = \sum_{j=1}^{N} \sum_{i=1}^{|POA(op_j)|} U(op_{j,i}) * x_{j,i}$$

where *N* is the number of operation invocations within the WS-BPEL scenario, $|POA(op_i)|$ is the cardinality of POA(op$_i$) and $x_{j,i}$ is a binary variable taking the value 1 if $op_{j,i}$ is chosen for implementing operation $op_j$ of the WS-BPEL scenario and the value 0 otherwise.

Since each operation $op_i$ designated in the original WS-BPEL scenario must be realized through exactly one operation in POA($op_i$), the constraint set

$$\sum_{i=1}^{|POA(op_j)|} x_{j,i} = 1, 1 \leq j \leq N$$

is added to the problem. In order to support transactional semantics of operations, which are possibly present when two or more invocations are made to services delivered by the same provider[1], the adaptation layer adds constraints to the problem as follows: let us assume that operations $op_x$ and $op_y$ belong in the same affinity group, and POA($op_x$) = {$o_{x,1}$, $o_{x,2}$, …, $o_{x,L(x)}$}, POA($op_y$) = {$o_{y,1}$, $o_{y,2}$, …, $o_{y,L(y)}$}, the possible operation assignments for $op_x$ and $op_y$, respectively. Without loss of generality, we assume that for the first *k* services in POA($op_x$) and POA($op_y$):

$$provider(op_{x,m}) = provider(op_{y,m}) \ \forall \ 1 \leq m \leq k$$

while

$$provider(op_{x,m}) \neq provider(op_{y,m}) \ \forall \ m > k$$

Under this setting, transactional semantics are maintained if for the realization of $op_x$ and $op_y$ within the scenario, the corresponding services (i.e. services with the same index) from POA($op_x$) and POA($op_y$) are selected. To ensure this in the solution of the IP problem, the following constraints are added to the problem:

$$op_{x,a} - op_{y,a} = 0, 1 \ \leq a \ \leq \ k$$

This method is directly generalizable to cases where the affinity group contains more than two operation invocations; for example if the affinity group contained a third operation invocation $op_z$, the set of constraints

$$op_{x,a} - op_{z,a} = 0, 1 \ \leq a \ \leq \ k$$

would be added to the problem. Notice that there is no need to include additional constraints to explicitly maintain transactional semantics between $op_y$ and $op_z$ (i.e. $op_{y,a} - op_{z,a} = 0$), since this is guaranteed by the two introduced set of constraints by virtue of transitivity.

This approach (a) exploits the concept of service replacement candidate, limiting service selection affinity only to the operations actually involved in the scenario and (b) further broadens the options available to the middleware by removing the need for affinity maintenance among services that coincidentally were set in the original scenario to be directed to the same service provider.

Afterwards, the IP problem is solved and the solution is stored –coupled with the session id– to the *session memory* (cf. Fig. 6).

## *5.2 Invocation adaptation*

When the adaptation layer intercepts an operation invocation (recall from section 3 that the preprocessor arranges so that invocations are redirected to the adaptation layer), it retrieves from the session memory the selection made for the realization of the particular invocation in the context of the current WS-BPEL scenario execution (i.e.

[1] An example of a case with transactional semantics is when a service invocation books a room while a subsequent one pays for the booking; clearly, the booking must be made to the hotel in which the room was booked, so if the adaptation mechanism redirects the booking request to hotel A, then the payment request should be redirected to Hotel A too.

within the execution plan formulated as described in subsection 5.1) and redirects the invocation to that service. The reply is then collected and returned as a reply to the WS-BPEL orchestrator.

## 5.3 Cleanup and housekeeping

Finally, when the WS-BPEL scenario reaches its end, it invokes the *releaseSession* web service, providing the session identifier as a parameter. The *releaseSession* service will then remove from the session memory all information pertaining to this session.

## 6. Experimental evaluation

In this section, we report on our experiments aiming to substantiate the feasibility of the proposed approach, both in terms of execution time (quantifying the introduced overhead and performance gains) and solution quality. For our experiments we used two machines: (a) a workstation, equipped with one 6-core Intel Xeon E5-2620@2.0GHz CPU and 16 GB of RAM, which hosted the preprocessor and the clients and (b) a workstation with identical configuration to the first, except for the memory which was 64GBytes, that hosted the WS-BPEL orchestration engine (Apache ODE 1.3.6), the adaptation layer, the target web services deployed on a Glassfish 4.1 application server and the service repository. The machines were connected via a 1Gbps LAN. The service repository was implemented as in-memory hash-based structure, which proved more efficient than using a separate (memory or disk-based) database. Preprocessing time is not included in the overheads, since this is performed in an off-line fashion and does not penalize the WS-BPEL scenario execution performance. In all experiments, the service repository was populated with synthetic data having an overall size of 1,000 web services; each web service included 3-8 operations and each operation was offered by a number of alternative providers, ranging from 5 to 50. Each service had at least 5 other services equivalent to it (i.e. having equivalents for *all* its operations). QoS attribute values in this repository were uniformly drawn from the domain [0, 10]. The WS-BPEL scenarios used in the experiments were synthetically generated by randomly drawing operations from the repository, and the performance evaluation tests were run for each of the generated scenarios; 1,000 scenarios were generated in total. We resorted to synthetic data due to the lack of a real-world test suite. In the scenario generation process, two consecutive functionality invocations were selected to be executed sequentially (*sequence* construct) with a probability of 0.7 and in parallel (*flow* construct) with a probability of 0.3. In our first experiment, we quantify (a) the time needed to formulate the WS-BPEL scenario execution plan, for varying degrees of concurrency (incurred once per execution), (b) the overhead imposed by the middleware intervention during service invocation (incurred for each invocation; the diagram illustrates the overhead sustained for *all* invocations within the scenario executions) and (c) the overall overhead per WS-BPEL scenario execution (Fig. 7). We can observe that all overheads remain relatively low, even for high degrees of concurrency, (an overall penalty of 250 msec for 200 concurrent invocations) and scales linearly with the concurrency degree.

Fig. 8 compares the QoS of the execution plan formulated for a number of representative trial cases and on average by (i) the simple QoS-based algorithm described in [8] and (ii) the approach proposed in this paper. The average shown in the diagram has been computed considering all 1,000 WS-BPEL scenarios used in the experiment, while the representative trial cases were chosen so as to include different number of operation invocations (scenarios 1-3 contain 3 invocations, scenarios 4-6 contain 6 invocations and scenarios 7-10 contain 8 invocations), varying settings regarding parallel flows (scenarios 1, 2, 4 and 7 contain no parallel flows, scenarios 3, 5, 8 and 9 contain one parallel flow and scenarios 6 and 10 contain two parallel flows) and different numbers of data-dependent invocations (from one to seven; some data dependencies formed chains e.g. *s1 is dependent on s2* $\wedge$ *s2 is dependent on s3*, while other data dependencies were unconnected, e.g. *s1 is dependent on s2* $\wedge$ *s3 is dependent on s4*).

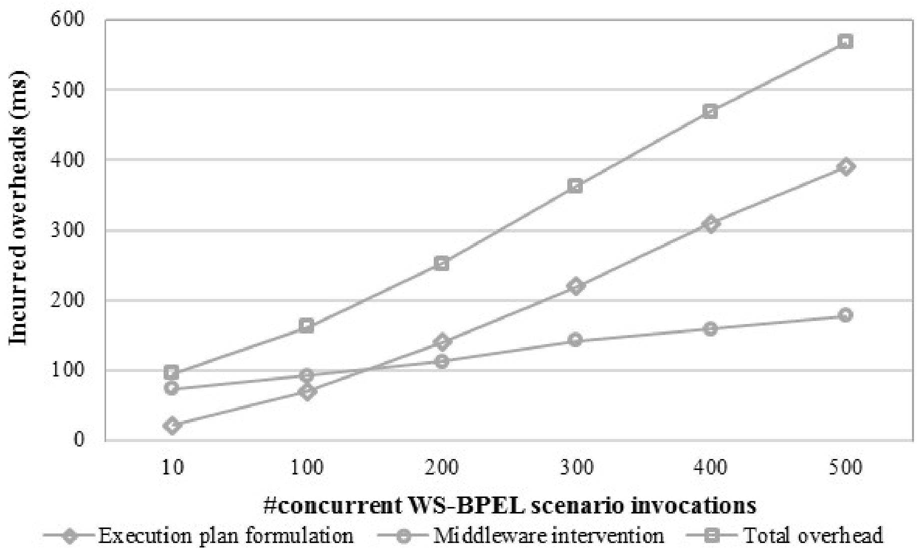

**Figure 7: Execution plan formulation overhead**

We chose to compare the proposed approach against the one described in [8], since the latter handles parallel flows and is exhaustive, always thus locating the optimum solution. The lower and upper QoS bounds for operation invocations were randomly drawn from the domains [0,4] and [6,10, respectively]. The weights of the QoS attributes were randomly selected from the domain [0,1]. In all cases, a uniform distribution was used. The diagram shows that the algorithm proposed in this paper achieves solutions whose QoS is on average higher by 22% than the corresponding solutions formulated by the algorithm described in [8]. This is due to the parallelization of operation invocations, which (a) lead to reduced response time and (b) due to the relaxation of the response time constraints allowed by the parallelization, the set of alternatives available to the adaptation mechanism is broadened (through allowing for selection of implementations with higher execution times than would be possible in the original scenario with sequential execution); this in turn provides opportunities for formulation of better execution plans, in the cases that the implementations that can now be selected score better in the rest QoS dimensions.

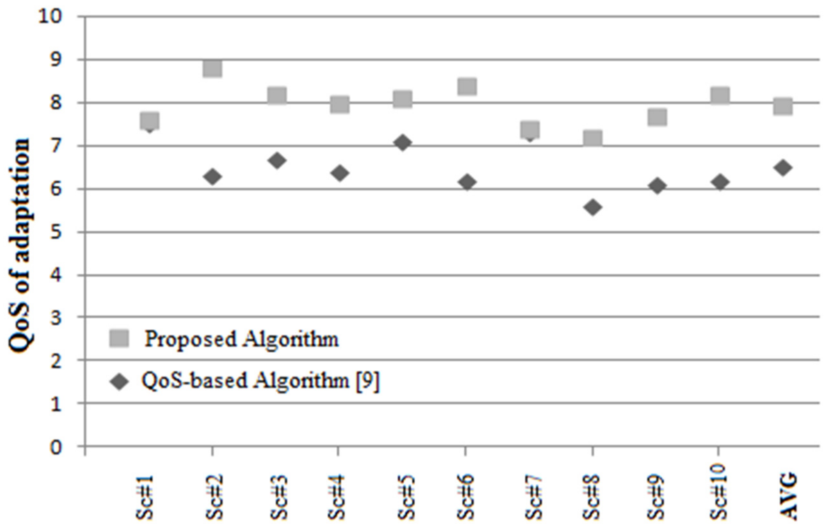

**Figure 8: QoS of solutions formulated by the proposed approach and the algorithm described in [8].**

We can notice that in cases 1 and 7 the proposed approach, as well as the algorithm described in [8], achieve exactly the same score. This is due to the fact that the optimal services had already been chosen in the first place, hence even after the parallelization process, the web services selection is exactly the same.
More specifically, in case 7, before the parallelization adaptation, processes A1 and B1 had been chosen with QoS values of (7,7,8) and (8,6,8), respectively, as far as response time, cost and availability are concerned. The other options (equivalent web services) available for these two processes were A2 with QoS values of (3,4,5) and A3 with QoS values of (2,6,7), as far as A1 is concerned, and B2 with QoS values (5,3,2), as far as B1 is concerned, hence even after the parallelization process, the web services selection is exactly the same and the overall adaptation QoS score (considering $rt_w=av_w= c_w$) is equal to (7+7+8+8+6+8)/6 = 7.33, as depicted in figure 8.

## 7. Conclusions

In this technical report we have presented the preprocessor transformations and adaptation operations for improving QoS delivered by WS-BPEL scenario adaptation through service execution parallelization. The preprocessor transformations aim at restructuring the parallelizable operations to be executed in parallel, even though the WS-BPEL scenario designer has specified sequential execution. Exploitation of parallelism can serve as an aid to the adaptation process by broadening the set of alternatives available to the adaptation mechanism: since parallelism reduces the overall execution time, in the parallelized scenario it is possible to choose operations with higher response times but better values in other QoS dimensions (e.g. cost), with the composition respecting the overall WS-BPEL scenario execution time limits, but scoring higher in the other dimensions (e.g. having lower costs).

The preprocessor also caters for making the scenario adaptation ready, i.e. inserting appropriate code to pass the data required to perform the adaptation to a newly introduced adaptation layer and redirect service invocations to this layer.

## 8. References

[1] V. Cardellini, V. Di Valerio, V. Grassi, S. Iannucci, F. Lo Presti, “A Performance Comparison of QoS-Driven Service Selection Approaches”, Proceedings of ServiceWave 2011, Abramowicz W et al. (Eds.): LNCS 6994, 2011, pp. 167–178.

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

[3] O’Sullivan, J., Edmond, D., Ter Hofstede, A.: What is a Service?: Towards Accurate Description of Non-Functional Properties. Distributed and Parallel Databases, vol. 12 (2002)

[4] M. Alrifai, T. Risse, “Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition”, Proc.s of the 18th international conference on World Wide Web (WWW '09), 2009, pp. 881-890.

[5] D. Ardagna, B. Pernici, “Adaptive Service Composition in Flexible Processes”, IEEE Transactions on Software Engineering, vol. 33, no. 6, June 2007, 369 – 384

[6] M. Alrifai, T. Risse, “Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition”, Proc.s of the 18th international conference on World Wide Web (WWW '09), 2009, pp. 881-890.

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

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

[9] X. Fei, S. Lu, “A Dataflow-Based Scientific Workflow Composition Framework”, IEEE Transactions on Services Computing 5(1), 2012, pp.45-58

[10] G. Goff, K. Kennedy, C.W. Tseng, “Practical Dependence Testing”, Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, 1991, pp. 15-29.

[11] A.J. Bernstein, “Analysis of Programs for Parallel Processing”, IEEE Trans. on Electronic Computers” Volume:EC-15(5), 1996, pp. 757–763.

[12] Red Hat. JBoss Enterprise SOA Platform 5: ESB Services Guide. https://access.redhat.com/documentation/en-US/JBoss_Enterprise_SOA_Platform/5.