Context-based transactional service selection approach for service composition*

Joyce El Haddad[1], Maude Manouvrier[1], Stephan Reiff-Marganiec[2], and Marta Rukoz[1,3]

[1] Université Paris-Dauphine, LAMSADE, France
[2] University of Leicester, UK
[3] Université Paris Ouest Nanterre La Défense, France

Abstract. Building applications dynamically from individual services offered by different organizations is one of the opportunities that Service-oriented Architecture (SoA) provides. With increasing number of providers, there is a choice among several functionally equivalent services resulting in a need to select the most appropriate ones in terms of Quality of Service (QoS) and transactional properties. In this paper, we propose a new web service composition approach bringing together the ideas of dynamically selecting services based on context with techniques to ensure that transactional properties are met across the application.

1 Introduction

Service-oriented Computing (SoC) is establishing itself as the dominant paradigm for developing distributed software applications. The basic building block in SoC is a 'service' – a computational unit that exists at a high abstraction level, usually closely related to a business functionality. Services are self-describing (e.g. through WSDL or OWL-S), discoverable (through registries) and adhere to common communication standards making them compatible with each other accross programming language and operating system boundaries. Applications are collections of services, where services are assembled in a loosely coupled fashion to achieve a larger business goal. One way to describe these applications is through business processes expressed through BPEL (or other process notations). Services are bound to tasks in a business process to enable its execution.

While services fulfil a specific functionality, there might be a plethora of services with the same functionality available. This leads to the important questions of how can we differentiate services and how can we choose the most appropriate service. Answers to the former come in terms of quality of service attributes and more generally non-functional properties – that is descriptors of availability,

* This work was conducted while Reiff-Marganiec was on study leave from the University of Leicester and was visiting Professor at the Université Paris-Dauphine and has been partially supported by the project "PERvasive Service cOmposition" (PERSO) funded by the French National Agency for Research (ANR JC07_186508).

reliability but also cost or security aspects. Service selection is concerned with either selecting the most appropriate service for a specific task (local selection) or finding an instantiation of services for a whole workflow (global selection) that optimises some global measure of desirability. While we will look at the respective advantages later, we briefly note that selecting one service for a task that is about to be executed can of course pick a service that is good for the user's and service's current context while preselecting all services might mean that those later in the chain were selected based on context information that has changed by the time they will be executed. On the other hand making many individual, independent decisions means that we cannot achieve a solution that is globally optimal.

Considering a process as a whole, we might have additional criteria that need to be satisfied, with transactions being predominant in business processes. It is often desirable to ensure that the business process overall satisfied a certain transactional property, and clearly the individual selected services will contribute to the overall property. Indeed, delivering reliable service composition over unreliable services is a challenging problem [1]. The interoperation of distributed software-systems is always affected by failures, dynamic changes, availability of resources, and others. In this context, a service that does not provide a transactional property might be as useless as a service not providing the desired functional results [2]. If the composition is based on services by only considering functional requirements, then it is possible that during the execution, the whole system becomes inconsistent in presence of failures. Thus, selection of transactional services allows the system to guarantee reliable composition execution. Indeed, the execution of transactional services will leave the system in a consistent state even in presence of failures.

In this paper we are addressing a service selection problem, which arises from the above: we are in a context-aware setting (that is user and service context changes for example by user's moving or services becoming busy or unavailable) while the business process is executing. However, the overall process is subject to said transactional requirements. We are presenting a novel method to select the most appropriate service for a task optimising the interplay of local and global non-functional (i.e. QoS) and transactional properties dynamically while the business process executes.

The paper is organised as follows: the next two sections present background on some fundamental techniques and an introduction to the workflow notation and naming conventions used. Section 4 introduces the context based transactional service selection approach. The remaining sections identify related work and conclude the paper with an outlook to future work.

2 Background

2.1 Transactional Properties of Services

As explained earlier, the execution of a business process is often subjected to overarching Transactional Properties (TP) to ensure the overall consistency. Web

Services (WSs) may provide distinct transactional behaviour: the main transactional properties [3] are pivot (p) and compensatable (c) and retriable(r). A service is said to be *pivot* if its behaviour supports atomic transactions. In other words, a service is pivot if when it successfully completes, its effects remains forever and cannot be semantically undone; if it fails it has no effect at all. A service is said to be *compensatable* if its behaviour supports compensatable transactions, that is it offers operations to semantically undo the original effects. A service is *retriable* if it guarantees a successfully termination after a finite number of invocations. In other words, a service with this property offers forward recovery.

A service can combine behavioural properties. For instance, a service can combine the pivot and retriable properties which leads to a new behavioural property, denoted pr. Similarly, a service can be compensatable and retriable which leads to a new behavioural property, denoted cr. Thus the set of all possible combinations for the behavioural property of a service is $\{p, c, pr, cr\}$.

When each task of a workflow is implemented by a component Web service, we obtain a Composite Web Service (CWS). Several CWSs can be associated to the same workflow, depending on the assigned component WSs. The transactional properties of a CWS highly depend on the transactional properties of its component WSs and on the structure of the workflow. Similarly, we can define transactional properties of CWSs [4]; we provide an overview. A CWS can be retriable (r), atomic ($\vec{a}$) or compensatable (c), where again compensatable and atomic can be composed with retriable (to cr and ar). A CWS is *retriable* when all its tasks (component services) are retriable. A CWS is *atomic* if it can be treated as an unit of work. In other words, if all the tasks of the CWS complete successfully then their effect remain forever and cannot be semantically undone. On the other hand, if one task does not complete successfully then all previously successful tasks have to be compensated. We use $\vec{a}$ to denote that a CWS is atomic while $\tilde{a}$ denotes non-atomic CWS. Finally a CWS is compensatable if all its tasks can be compensated.

Fig. 1. Computing transactional properties of a CWS from components' properties

Once individual services occur as part of a workflow, we can compute the behavioural property of a Transactional Composite Web Service (TCWS) from those of the contributing individual services and the structure of the workflow. A TCWS is a CWS whose transactional behavioral property is in $\{\vec{a}, \vec{a}r, c, cr\}$.

In a previous work [4], we defined several rules allowing to compute any combination of transactional properties based on the respective workflow operators (sequential or parallel composition). Figure 1 shows a statemachine resulting from these rules, where the final states $\{c, \vec{a}, cr, \vec{a}r\}$ correspond to the set of transactional behavioural properties for a CWS. For example, the parallel (//) or sequential (;) composition between a compensatable retriable CWS (see state *cr* in Figure 1) and a pivot Web service (see transitions $//p$ and $;p$ from state *cr* in Figure 1) is atomic because if both components complete successfully then the result cannot be semantically undone and if the second component fails then the first one can be compensated. Using these rules, we also have defined in a WS selection approach to obtain a TCWS satisfying a global transactional properties. We have studied two global transactional properties: *Risk 0* and *Risk 1*. *Risk 0*, grouping c and *cr* properties, means that the obtained TCWS (one that satisfies the global property), once it successfully completes, can be compensated. On the other hand, *Risk 1*, grouping $\vec{a}$, $\vec{a}r$, c and *cr* properties, means that the obtained TCWS is a transactional one. Note that obtaining a TCWS satisfying *Risk 1* does not mean that all its component services must have $\vec{a}$ or $\vec{a}r$ as transactional property.

2.2 Backwards Composition Context based Service Selection

In previous work [5] we introduced a runtime service selection algorithm for context aware service selection. This algorithm uses a number of context artefacts: user context, service context and composition context. While the latter was introduced in [5], the former two have been discussed in [6, 7]. Here we will provide some insight into this previous work needed to understand the core of this paper. Of particular relevance are 3 ingredients: context information, a ranking mechanism and the selection algorithm.

In order to decide which service is most suitable, we need to consider three context factors. (1) The user context constraints for selecting an individual service for a sub-task, and (2) the services runtime context information. For example a lecturer needs to go to a class in another building but requires some printouts. So the user context contains information about the current position and the classroom, as well as the nature of the print job; some of this information is more static than others, but this does not matter; what matters is that the information is available from a context store. On the other hand a number of printing services are available, each with its own location (static context information), but also the length of the respective print queue (which is of course dynamic) – again the information can be queried from the context store. Further constraints on the selection are imposed by (3) the composition context constraints. This is again stored in the context store, if a service is invoked information regarding the invocation will be logged (did the service succeed? which service was invoked?). The composition context also contains information about preferential treatment between services (use service A and get 20% discount from service B).

The second component required is a method to evaluate the suitability of each service, that is a method to evaluate the match of a service to the user's

current context and needs. Clearly this has to be applied dynamically and shortly before the service is used as the context can change rapidly. While even simple weighted sums can perform this, they have numerous drawbacks (details are beyond the scope of this paper). Logic Scoring for Preference (LSP) [8] is a method to evaluate multi criteria decision problems. It is primarily meant for human use, so decisions as to which evaluation function to use for a specific criteria and how to select some of the parameters needed are manual decisions. The strength of the method is that it allows to express logical bindings between criteria such as the replacability (a good score in one criteria can replace a lower score in another, e.g. a higher price might be acceptable for a very fast service) or simultaneity (independent of the score, both criteria should be considered, e.g. you would never wish to jeopardise privacy no matter how well a service scores in other areas). In [9], we introduced a Type-based LSP Extension (TLE) service selection method, which automates the aspects for which that LSP required human input and hence allows for automatic ranking of services.

TLE is used to automatically rank individual services. In the absence of the composition context it will only consider local criteria. In the presence of the composition context the ranking will also make use of this information to return the best choice of service. However, we face a small problem in that the choice of a specific service does not only depend on the past (that is the services already executed, about which information is available in the composition context), but also on the future (that is the next service(s) to be executed). It is here where the backwards composition context based service selection (BCCbSS) approach comes into the picture. The basic idea is to select (but not execute) a service for a task, then move to the next task and also select a service. At this stage we return to the previous task and evaluate the previous choice again (the composition context now contains a glance at a possible future, in terms of holding data of the next service candidate). We might keep our choice or select a different service but we will then invoke this service. Finally we move to the next task and repeat the process. The pattern of moving one step ahead to get a glance of the future and then returning to finalize the decision will also be apparent in the approach presented in this paper and its working will be explained in more detail later.

At this stage the reader might wonder why we only look ahead one step, and this is a justified question. We will return to this point in the discussion, but one strong motivation is the fact that the context of services and user changes.

3 Workflow notation and encoding

One of the fundamental structures used in this work is a workflow – the workflow describes the service composition in terms of tasks and operators. The presented algorithm will be executing the workflow by selecting and running services for each task. The workflow consists of two types of objects: tasks and operators. These alternate, that is we cannot have two tasks or operators follow each other without an object of the other type inbetween. Tasks are named for easy reference by humans and there is an understanding that tasks represent abstract activities

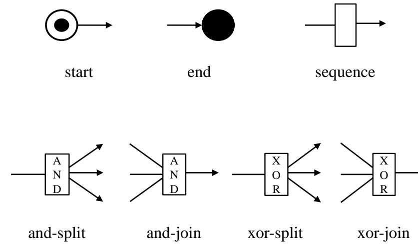
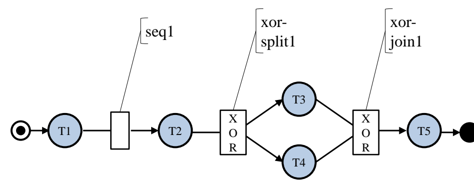


Fig. 2. Workflow Operators

and are executed through services. There are 7 operators: *start*, *end*, *sequence*, *and-split*, *and-join*, *xor-split* and *xor-join*. The operators are depicted in Figure 2 and denote the start and end of a workflow, the sequential connection of tasks, a parallel split and join as well as a conditional split (like if) and join respectively. Note that in each workflow, the first operator is a *start* operator and the last one is an *end* operator. Both of these operators (*start* and *end*) appears only once in a workflow. Operators split and join come as matched pairs; that is e.g. an n-way *and-split* has to be closed by an n-way *and-join*.

Furthermore, each workflow object has two pointers assigned to it, pointing to the previous and next workflow object. In particular operator *start* can always be identified and for a task T we have the following: $T.next$ (the operator following the task), $T.prev$ (the operator preceeding the task), $T.next.next$ (the task following the current task), $T.prev.prev$ (the task preceeding the current task). There is a slight complication to this simple pointer structure in that some operators might be preceeded (joins) or followed (splits) by a number of tasks, hence the the pointers from operators actually point to lists of tasks which might have 1 or more elements depending on the operator. Figure 3 illustrates the pointer structure with a simple example workflow.



T1.prev = start
T1.next = seq1
T1.next.next = [T2]

T2.prev = seq1
T2.prev.prev = [T1]
T2.next = xor-split1
T2.next.next = [T3,T4]

T3.next = xor-join1
T4.next = xor-join1
T3.prev= xor-split1
T4.prev = xor-split1

T5.next = end
T5.prev = xor-join1
T5.prev.prev =[T3,T4]

Fig. 3. Workflow Pointer Structure

4 Context-based Transactional Services Selection Algorithm

We propose an approach to resolve the service selection problem by choosing the most appropriate service for a task respecting the local and global QoS and transactional properties at runtime. The inputs of our approach are a workflow and a risk value, the latter captures the desired transactional property. The result is that the workflow is executed using suitable services and the achieved transactional property is returned. We make the assumptions that one WS invocation completes only one task of the workflow and that the registry contains at least one service per transactional property for each functionality. The approach is presented by the `BackwardsTransactionalServiceSelection (BTSS)` algorithm divided into 6 parts (methods) described below.

`BackwardsTransactionalServiceSelection (BTSS)` (Algorithm 1) represents our context-based transactional service selection and has two inputs: the workflow template WF, as described in Section 3, and the global transactional property $Risk$ as described in Section 2.1. Depending on the $Risk$ value, the set $TSet$ of the allowed transactional properties for a service to achieve overall transactional behaviour is computed (lines 4 and 5). Then, the algorithm selects and invokes at run-time a WS for each task of the workflow WF by calling function `BTSS-Impl` and returns the transactional property (TP) of the executed TCWS. The main method of our selection process is described in Algorithm 2.

Algorithm 1: BackwardsTransactionalServiceSelection(WF,Risk)

Input: WF, $Risk$ */* The workflow template and the desired overall aggr. TP or global transactional property */*
Output: TP */* The achieved aggr. TP of the invoked services*/*

```
1 begin
2     /* Initialize algorithm */ T ← start.next /* Select the first task */;
3     /* TSet is the allowed transactional properties for a service to achieve the overall
      transactional behavior */
4     if Risk = 0 then TSet ← {c, cr};
5     else TSet ← {p, pr, a⃗, a⃗r, c, cr};
6     /* Start working through workflow */
7     (TP, T) ← BTSS-Impl(WF, T, TSet, Risk) /* See Algorithm 2 */;
8     return(TP);
9 end
```

`BTSS-Impl` (Algorithm 2) implements the backwards composition context based service selection approach. The idea is to go back one step to check if the selected service is the best choice regarding the currently selected service and the composition context information and invoke the selected service as soon as possible. The method is composed of 5 steps. The first step discovers all candidate services from the registry for the current task in the composition workflow (lines 1 to 9). This step will return a set of services S_c providing the functionality of the current task. If S_c is an empty set, then the composition fails, otherwise we proceed with the next step. The second step reduces the set S_c to services with a transactional property regarding to $TSet$ and calls the ranking function in `FindBestServ` method (Algorithm 4) to give a fixed evaluation value to each candidate service by considering the selected service

for the previous task and the aggr. TP (lines 11 to 15). The currently selected service for the current task is then the first service in S_c. The next two steps depend respectively on the operators before and after the current task. The third step checks the operator preceeding the current task. If the operator is a *seq* or an *xor-join* then go back one step and call the ranking function over the set of services selected for the previous task, S_p, to check if the selected service for the previous task is still the best choice regarding the composition context information, the currently selected service for the current task and the aggr. TP. The result of this ranking is, either the selected service for the previous task is still the best choice (first service in S_p) then it will be re-selected and executed, or another service in S_p will be selected and executed. This is done by calling `InvokingService` method (Algorithm 6). After the execution of a service, the aggr. TP is updated following the `AggregateTransactionalProperties` method (Algorithm 5) (lines 16 to 22). The fourth step checks the operator following the current task. If the operator is an *xor-split*, an *and-split*, an *and-join* or the *end* operator then a service for the current task must be executed before proceeding. To do so, the ranking function is called over S_c to check if the currently selected service for the current task is the still the best choice regarding the composition context information and the aggr. TP. Depending on the result of the ranking, either the currently selected service is still the best choice (first service in S_c) then it will re-selected and executed, or another service in S_c will be selected and executed. As for the previous step, after the execution of a service for the current task, the aggr. TP is updated following the `AggregateTransactionalProperties` method (Algorithm 5) (lines 23 to 29).The last step moves the *next* pointer to a new task in the workflow depending on the operator following the current task. If the operator is an *and-split* then the `AND-Split` method (Algorithm 3) is called in order to select services for each branch of the pattern independently. If the operator is an *and-join* then *next* points to the task after the *and* pattern and we return the transactional result of the pattern. If the operator is an *xor-split* then *next* points to the first task in the branch chosen based on the split condition. Finally, if the operator is a *seq* or an *xor-join* then *next* points to the next task in the workflow and the set of service candidates is stored in S_p (lines 30 to 46).

`AND-Split` (Algorithm 3) manages the selection in an *and-split* pattern. It consists in selecting and invoking at run-time the services needed for each branch of the pattern independently (lines 3 to 8), by calling method `BTSS-Impl` (Algorithm 2). Because each branch is managed independently for the others, we restrict the set of allowed transactional properties to retriable properties ($\{pr, \vec{a}r, cr\}$) when the desired overall aggregate TP is $Risk1$. Theses restrictions allow for example to avoid the selection of a pivot (p) or an atomic ($\vec{a}$) service for several branches or the selection of a compensatable service (c) in one branch and of non compensatable ones ($\{p, pr, \vec{a}, \vec{a}r, \}$) for the other branches. Indeed, as shown by Figure 1, a pivot (p) or atomic service ($\vec{a}$) can only be executed in parallel with a compensatable retriable one (cr) and a compensatable service (c) can only be executed in parallel with compensatable (c) or compensatable

retriable (cr) ones. After the synchronization of the execution of all branches (line 9), the transactional property of each branch (TP_i) is aggregated and the achieved aggr. TP $TPAND$ of the services invoked in the pattern is computed and returned by the algorithm.

Algorithm 3: AND-Split(WF, T, $TSet$, TP, $Risk$)

Input: WF, T, $TSet$, TP, $Risk$ /* *The workflow template as described earlier, the first task to be considered, the set of allowed transactional properties, the transactional property of activated services and, the desired overall aggr. TP* */
Output: TP, T /* *The achieved aggr. TP of the invoked services and the next task after the WF segment completed by the specific call to the algorithm* */

```
1  begin
2      TPAND ← I /* The aggr. TP of the pattern */;
3      /* If Risk=1 then we don't care about compensation Else we need compensatable
       services but TSet is already fixed properly in Algo. 1*/
4      if Risk = 1 then TSet ← TSet ∩ {pr, ar⃗, cr} ;
5      NBBranches ← T.next.next.length() /* Number of branches in the pattern */ ;
6      /* Call method BTSS-Impl for each branches independently */
7      for (i = 1; i < NBBranches; i++) do
8          fork((TP_i, T) ← BTSS-Impl(WF, T.next.next[i], TSet, Risk));
9      sync /* Wait for all forks to finish */;
10     /* Compute the aggr. TP of the pattern from the aggr. TP TP_i of each branches */
11     for (i = 1; i < NBBranches; i++) do
12         TPAND ← AggregateTransactionalProperties(TPAND, TP_i);
13     /* Compute the achieved aggr. TP of the invoked services */
14     TP ← AggregateTransactionalProperties(TPAND, TP);
15     return (TP,T); /* The forks update T; see and-join in Algo. 2 */
16 end
```

`FindBestServ` (Algorithm 4) is essentially concerned with the invocation of the TLE ranking function (see Section 2.2), overviewed in the background section. To recap, this function evaluates the scores for the relevant non-functional properties and aggregates these to determine an overall score for each service. Services are then ranked by their respective scores. The evaluation of the individual criteria takes into account the composition context (so anything that is known about the execution so far as well as explicitly stated relationships between services such as discounts), the context of the user as well as the context of the service. While this algorithm is of course key to selecting the right service, it can be seen as a variation point in the overall method. Different ranking methods can be used to achieved desired outcomes.

For example, we have two types of properties to consider: transactional and non-functional (i.e. QoS) properties, and hence we can immediately think of several variants of this algorithm. (1) We could apply the ranking function within subsets of services that group services by transactional properties. This approach is probably not desirable as we might miss out on good opportunities (maybe we find that the best service in the most desirable TP set is scoring very low, while making a slight trade-off in the TP could provide a high scoring service). (2) An alternative is to rank by non-functional properties but use the transactional property as a tiebreaker: so if two services score were similarly (the distance would be defined as a threshold) their transactional properties are considered, and if making a trade-off on the overall non-functional property score gives a better transactional property the ranking order would be changed. Apart from the difficulty of identifying a good threshold value this could be a very attractive

solution. (3) Finally, one could consider the transactional properties as another non-functional property and deeply embed transactional properties as a factor in the TLE method – this would work as we have already removed services that have unsuitable transactional properties before invoking the ranking but of course makes the role the transactional property in the selection less apparent.

Algorithm 4: FindBestService(S_c,s_p, s_n, T_i, TP)

```
  Input: S, s_p, s_n, T_i, /* The set of candidate services, the service used for previous task,
         the service candidate for the next task and information on the current task*/
  Output: S /* The ranked list of services for the current task */
1 begin
2   if S ≠ ∅ then
3     for (i = 1; i <= Size(S); i++) do
4       r_i = LSP(s_i, s_p, s_n) s.t. s_i ∈ S;
5       /* LSP ranking function; if s_p and/or s_n are empty then the ranking will be
        based on the remaining available information*/
6     return S;
7 end
```

`AggregateTransactionalProperties` (Algorithm 5) implements the calculation of transactional properties for composed services, as explained in the background section (see Figure 1), with variable TP corresponding to the states of the state diagram and variable t corresponding to the label of the transitions of the diagram.

Algorithm 5: AggregateTransactionalProperties(TP,t)

```
   Input: TP,t /* The transactional property of activated services and the transactional
          property of a selected service*/
   Output: TP /* The aggregation of transactional properties computed from its previous
           value and the value of t, using the state diagram of Figure 1 */
 1 begin
 2   if TP ∈ {I, cr} then
 3     if t ∈ {p, ã} then TP ← ã;
 4     else
 5       if t ∈ {pr, ãr} then TP ← ãr;
 6       else
 7         if t = c then TP ← c;
 8         else TP ← cr;
 9   else
10     if TP = c and t ∈ {p, pr, ã, ãr} then TP ← ã;
11   return TP;
12 end
```

`InvokeService` (Algorithm 6) is concerned with executing the best service for a specific task. It will essentially try to invoke the best service in the ranking list, however if this fails it will try the next best option. This continues until one service executes successfully or that we run out of service candidates. There is one exception to this behaviour: if we are trying to invoke a service that is retriable we have a guarantee that it will eventually succeed, so we are simply trying this candidate until we gain the wanted success. The method returns information on the actually executed service and logs relevant information regarding the executed service in the composition context.

Algorithm 6: InvokeService(S)

```
    Input: S /* A ranked list of Service candidates */
    Output: s /* The actually executed service */
 1  begin
 2      /* Assume S is indexed from 1 to n */;
 3      /* success reflects that the service completed execution successfully */;
 4      i ← 1;
 5      repeat
 6          s ← invoke S[i];
 7          /* Execute the service at the position i in the ranked list S */;
 8          if ¬success and TP of S[i] ∈ r then
 9              repeat
10                  s ← invoke S[i] /* retry until success */;
11              until success ;
12          else
13              i ← i + 1;
14      until (success or i > n) ;
15      Update_Composition_Context;
16      return s;
17  end
```

In order to illustrate the execution of all the above presented methods of our approach, let us consider the following example: a registry with a set of Web services each with its execution price (called cost) as a non-functional property, the workflow of Figure 3 and a global transactional property $Risk1$. Suppose that two candidates services are discovered for the task T_1: a service s_{11} whose cost is €2 and transactional property p, and a service s_{12} whose cost is €5 and transactional property c. At the beginning of the Algorithm, there is composition context is empty, so s_{11} is selected because of its cost. Because T_1 is the first task, we go back to the `repeat` loop of Algorithm 2 to discover services for task T_2 ($s_p = s_{11}$). Let us suppose that two candidate services are discovered for T_2: a service s_{21} with a transactional property pr which requires €5 to be composed with s_{11} and €1 to be composed with s_{12} and a service s_{22} with a transactional property cr which requires €8 to be composed with s_{11} and €2 to be composed with s_{12}. All compositions between these services are transactionally correct. Based on the first selected service $s_p = s_{11}$, s_{21} is selected ($s_c = s_{21}$). Because $T_2.prev = seq$, s_p is re-selected for T_1, based on the selected service for the next task s_{21} to make sure the best possible combination between T_1 and T_2. As result, s_{12} is ranked as a better service than s_{11} in the process and s_{12} is invoked ($TP = c$). Because $T_2.next \neq seq$ and $T_2.next \neq xor - join$, then $s_c = s_{21}$ is invoked. Depending on the context and on the condition of the *xor* pattern, let us consider that the branch chosen is T_3. Candidates services are selected for task T_3 taking into account the composition context and the achieved aggr. TP of the invoked services for the two previous tasks. Then, a process equivalent for T_3 and T_5 is done, as it has been done for tasks T_1 and T_2. Let us now suppose that the pattern in the workflow of Figure 3 is an *and* pattern rather than an *xor* pattern. In this case, Algorithm 3 is executed. Candidate services are selected for tasks T_3 and T_4 independently, using the composition context information and the achieved aggr. TP of the invoked services for the two previous tasks (both are invoked because they are followed by *and-join*). Then, candidate services are selected for task T_5 and invoked because T_5 is followed by an *end* operator.

5 Related Work

Service selection approaches for business process instantiations are either aiming at local or global optimal selection. *Local optimization based service selection* refers to selection methods which only consider selection constraints related to the current activity in the workflow without specifying and considering the constraints implied by the workflow context and the consequences that the choice will have on later tasks. For example, a policy based BPEL workflow Web service selection method is presented in [10]. It extends BPEL for run-time adaptation of service by adding the policy reference to each node. The policy documents provide the local optimization rules which are independent from each other. The service selection process is applied at each node separately. A similar approach was also presented in the earlier e-Flow project [11]. The biggest advantage of the local optimization methods is efficiency in selection time - the worst case can be solved in polynomial time. However, it does not necessarily select the optimal or even close to optimal service in the global composition context. *Global optimization based service selection*, on the other hand, considers the global selection constraints to select a group of services to instantiate the whole workflow. [12, 13] are two example approaches. By studying these approaches, we find they surely narrow the disadvantages pointed out for local optimization. However, they introduce their own problems: (1) The problem is inherently NP-complete [14] and hence scalability is not guaranteed; (2) if a service is not available or fails by the time it is invoked a new solution needs to be computed and (3) they are not able to determine appropriate choices in the presence of dynamic context information, which will inevitably not be available at the time a solution is computed or will have changed when a service is actually needed.

The presented approach makes the selection decisions task by task based on the current local and global composition context. The composition context is growing as we proceed through the tasks. Based on these context constraints, we may select the best service according to real-time knowledge for the next task. As we continue to select services, the composition context grows allowing for more fine-grained selection. Also going one step ahead before reviewing the last decision allows to consider relationships between closely related services.

As explained by [15], todays WS applications require advanced transactional models to guarantee integrity and continuity of business processes. The transactional WS composition problem has been extensively treated in the literature by using a predefined control structure or by automatically discovering the services and their control flow. A predefined control structure, such as workflows [16, 17] and Advanced Transactional Models (ATM) [18, 19], is comprised of abstract processes to meet the functional user request, and the order in which they must be evaluated. In workflows, the execution control is defined by the structure of the workflow while, in ATM approaches, it is explicitly defined within the application logic. As we do, in these approaches the problem is to identify resources or concrete services for each of the abstract processes. As far as we know, only few approaches consider QoS and transactional properties and none of them at run-time. [20] propose a composition model in design-time which

captures both aspects in order to evaluate the QoS of a composite WS with various transactional requirements. However, the authors do not consider the automatic selection step and only analyze the impact of the transactional requirements on the QoS of the composite WS. [17] propose a selection algorithm based on QoS by integrating the failure risk impact of each selected WS to reduce the average losses caused by execution failures of WSs. In a previous work [4], we proposed a selection algorithm for automatic WS composition integrating QoS and transactional properties but at design-time. We defined transactional properties definitions of WSs and composite WSs that we use in this work.

6 Conclusions and Future Work

This paper presented an approach that allows to select the most appropriate service for a task inside a workflow at runtime. To decide which service is most appropriate the context of the user and service are taken into consideration; furthermore the transactional property of the service candidates is used to decide on the best choice. In the presented approach selection decisions are made task by task based on the current local and global composition context, decisions might not be "perfect" in the sense of finding a global optimum but they are as good as available information allows for. They do consider a wider environment than just the current task and the relationships between closely related services for subsequent tasks is considered through the backwards checking mechanism. Executed solutions are guaranted to achieve the overall desired transactional property. As said earlier, we could debate why we only reconsider services one step back. The main reason is that we are assuming a dynamic environment where user and service context changes rapidly, and due to this what is "most relevant" becomes dynamic in a way which means that we want to make timely and as local as possible decisions.

The current approach does simply lead to a failure of the process when no suitable service is found. In future work we plan to investigate the use of the transactional properties of the chosen services to either undo the process entirely, undo parts and try alternatives or at least inform the user which service is compensatable. Another aspect for future investigation is more localised desired transactional properties. Currently, the user can express that the overall workflow should be compensatable, but in many applications it would be nice to tag crucial tasks (maybe expensive ones) as having to be compensatable without the overall process being so.

References

1. Liu, A., Li, Q., Huang, L., Xiao, M.: FACTS: A Framework for Fault-Tolerant Composition of Transactional Web Services. IEEE Trans. Serv. Comput. **3**(1) (2010) 46–59
2. Cardinale, Y., El Haddad, J., Manouvrier, M., Rukoz, M.: Web service selection for transactional composition. Procedia Computer Science (Elsevier) **1**(1) (2010) 2683 – 2692 ICCS 2010.

3. Mehrotra, S., Rastogi, R., Korth, H., , Silberschatz., A.: A transaction model for multidatabase systems. In: Int Conf. on Distributed Computing Sys. (1992) 56–63
4. El Haddad, J., Manouvrier, M., Rukoz, M.: TQoS: Transactional and QoS-aware selection algorithm for automatic Web service composition. IEEE Trans. on Serv. Comp. **3**(1) (2010) 73–85
5. Yu, H.Q., Reiff-Marganiec, S.: A backwards composition context based service selection approach for service composition. In: IEEE Int. Conf. on Serv. Comp. (2009) 419–426
6. Yu, H.Q., Reiff-Marganiec, S.: Automated context-aware service selection for collaborative systems. In: Proc. of The 21st Int. Conf. on Advanced Information Sys. Volume 5565 of LNCS. (2009) 193–200
7. Reiff-Marganiec, S., Truong, H.L., Casella, G., Dorn, C., Dustdar, S., Moretzky, S.: The incontext pervasive collaboration services architecture. In: ServiceWave. Volume 5377 of LNCS., Madrid, Spain (2008) 134–146
8. Dujmovic, J.: Continuous preference logic for system evaluation. IEEE Trans. on Fuzzy System **15**(6) (2007)
9. Yu, H.Q., Reiff-Marganiec, S.: A Method for Automated Web Service Selection. In: Proc. of the 2008 IEEE Congress on Services - Part I. (2008) 513–520
10. Karastoyanova, D., Houspanossian, A., Cilia, M., Leymann, F., Buchmann, A.: Extending BPEL for Run Time Adaptability. In: Proc. of the 9th IEEE Int. EDOC Enterprise Computing Conf. (2005) 15–26
11. Casati, F., Ilnicki, S., Jin, L., krishnamoorthy, V., Shan, M.C.: Adptive and Dynamic Service Compostion in eFlow. HP Laboratories Technique report (2000)
12. Canfora, G., PentaRaffaele, M.D., Esposito, R., Villani, M.L.: An approach for QoS-aware service composition based on genetic algorithms. In: Conf. on Genetic and Evolutionary Computation, Washington DC, USA (2005) 1069–1075
13. Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Trans. on Software Eng. (2005) 311–327
14. Yu, T., Lin, K.: Service Selection Algorithms for Composing Complex Services with Multiple QoS Constrains. In: Proc. of the Int. Conf. on Service-oriented Computing ICSOC, Springer (2005) 130–143
15. Badr, Y., Benslimane, D., Maamar, Z., Liu, L.: Guest Editorial: Special Section on Transactional Web Services. IEEE Trans. Serv. Comput. **3**(1) (2010) 30–31
16. Montagut, F., Molva, R., Golega, S.T.: Automating the Composition of Transactional Web Services. Int. J. Web Service Res. **5**(1) (2008) 24–41
17. Liu, H., Zhang, W., Ren, K., Zhang, Z., Liu, C.: A risk-driven selection approach for transactional web service composition. In: GCC '09: Proc. of the 2009 Eighth Int. Conf. on Grid and Cooperative Computing. (2009) 391–397
18. Lakhal, N.B., Kobayashi, T., Yokota, H.: FENECIA: failure endurable nested-transaction based execution of composite Web services with incorporated state analysis. In: The VLDB Journal. Volume 18. (2009) 1–56
19. El Haddad, J., Manouvrier, M., Rukoz, M.: A Hierarchical Model for TransactionalWeb Service Composition in P2P Networks. In: Proc. of the IEEE Int. Conf. on Web Services (ICWS). (2007) 346–353
20. Liu, A., Huang, L., Li, Q.: QoS-Aware Web Services Composition Using Transactional Composition Operator. 7th Int. Conf. Advances in Web-Age Information Management (WAIM), LNCS 4016 (June 2006) 217–228

Algorithm 2: BTSS-Impl(WF, T, TSet, Risk)

Input: WF, T, $TSet$, $Risk$ /* *The workflow template as described earlier, the first task to be considered, the set of allowed transactional properties and the desired overall aggr. TP* */

Output: TP, T /* *The achieved aggr. TP of the invoked services and the next task after the WF segment completed by the specific call to the algorithm* */

1 **begin**
2 /* *S is a set of candidate services, index by c(urrent) or p(revious)**/;
3 /* *s is a candidate services, index by c(urrent) and p(revious)**/;
4 $s_p \leftarrow null$;
5 /* *TP is the aggregated transactional property of invoked services, t_p is the transactional property of service s_p* */;
6 $TP \leftarrow I$;
7 $t_p \leftarrow I$;
8 **repeat**
9 $S_c \leftarrow$ DiscoverServices(T) /* *the set of candidate services for task T* */;
10 **if** $S_c \neq \emptyset$ **then**
11 $S_c \leftarrow S_c \cap$ {s| transactional property of $s \in TSet$} /* *Reduce the set S_c to services with transactional property in $TSet$; this operation maintains the order of S_c* */;
12 $S_c \leftarrow$ FindBestService(S_c, s_p, $null$, T, AggregateTransactionalProperties(TP, t_p));
13 /* *Find best services for T and rank them, see Algo. 4 and Algo. 5**/;
14 $s_c \leftarrow First(S_c)$;
15 $t_c \leftarrow$ Transactional Property of s_c;
16 **if** *(T.prev ='seq' OR T.prev ='xor-join')* /* *There was a previous choice that has not run* */ **then**
17 $S_p \leftarrow$ FindBestService(S_p, $null$, s_c, $T.prev.prev$, TP);
18 /* *Find the best service for T.prev.prev, see Algo. 4**/;
19 $s_p \leftarrow InvokeService(S_p)$ /* *See Algo. 6**/;
20 $t_p \leftarrow$ Transactional Property of s_p;
21 $TP \leftarrow$ AggregateTransactionalProperties(TP,t_p);
22 **if** $TP \in \{\vec{a}, \vec{a}r\}$ **then** $TSet \leftarrow TSet \cap \{pr, \vec{a}r, cr\}$;
23 **if** *(T.next ≠'seq' AND T.next ≠'xor-join')* /* *Commit the current choice before an operator* */ **then**
24 $S_c \leftarrow S_c \cap$ {s| transactional property of $s \in TSet$} /* *educe the set S_c to services with transactional property in $TSet$; this operation maintains the order of S_c* */;
25 $S_c \leftarrow$ FindBestService(S_c, $null$, $null$, T, TP);
26 /* *Find the best service for T, see Algo. 4**/;
27 $s_c \leftarrow InvokeService(S_c)$ /* *Find a service for task T, see Algo. 6**/;
28 $TP \leftarrow$ AggregateTransactionalProperties(TP,t_c);
29 **if** $TP \in \{\vec{a}, \vec{a}r\}$ **then** $TSet \leftarrow TSet \cap \{pr, \vec{a}r, cr\}$;
30 **switch** *(T.next)* **do**
31 **case** *'and-split'*:
32 $(TP, T) \leftarrow$ AND-Split(WF, T, $TSet$, TP, $Risk$);
33 **if** $TP = \vec{a}r\}$ **then** $TSet \leftarrow TSet \cap \{pr, \vec{a}r, cr\}$;
34 break;
35 **case** *'and-join'*:
36 $T \leftarrow T.next.next$;
37 return(TP,T);
38 **case** *'xor-split'*:
39 $T \leftarrow T.next.next[i]$ /* *Index i referring to the branch chosen based on condition in split* */;
40 break;
41 **case** *'seq', 'xor-join'*:
42 $S_p \leftarrow S_c$;
43 $S_p \leftarrow S_p \cap$ {s| transactional property of $s \in TSet$} /* *Reduce the set S_p to services with transactional property in $TSet$; this operation maintains the order of S_p* */;
44 $s_p \leftarrow First(S_p)$;
45 $t_p \leftarrow$ Transactional Property of s_p ;
46 $T \leftarrow T.next.next$ /* Select next task */;
47 **else**
48 return(fail) /* No service found for task */;
49 **until** *T.next = 'end'* ;
50 return(TP, null);
51 **end**