Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Heuristics for Composite Web Service Decentralization

Walid Fdhila[1], **Marlon Dumas**[2], **Claude Godart**[1], **Luciano García-Bañuelos**[2]

[1]LORIA-INRIA-UMR 7503, France
[2]University of Tartu, Estonia

The date of receipt and acceptance will be inserted by the editor

Abstract A composite service is usually specified by means of a process model that captures control-flow and data-flow relations between activities that are bound to underlying component services. In mainstream service orchestration platforms, this process model is executed by a centralized orchestrator through which all interactions are channeled. This architecture is not optimal in terms of communication overhead and has the usual problems of a single point of failure. In previous work, we proposed a method for executing composite services in a decentralized manner. However, this and similar methods for decentralized composite service execution do not optimize the communication overhead between the services participating in the composition. This paper studies the problem of optimizing the selection of services assigned to activities in a decentralized composite service, both in terms of communication overhead and overall Quality of Service (QoS), and taking into account collocation and separation constraints that may exist between activities in the composite service. This optimization problem is formulated as a Quadratic Assignment Problem (QAP). The paper puts forward a greedy algorithm to compute an initial solution as well as a tabu search heuristic to identify improved solutions. An experimental evaluation shows that the tabu search heuristic achieves significant improvements over the initial greedy solution. It is also shown that the greedy algorithm combined with the tabu search heuristic scale up to models of realistic size.

1 Introduction

One of the pillars of Service-Oriented Architecture (SOA) is the ability to rapidly compose multiple services into a value-added business process, and to expose the resulting business process as a *composite service* [6]. Composite services are generally captured by means of an orchestration model, at the heart of which is a process model that specifies control-flow and data-flow relations between activities, where each activity represents either an internal step (e.g. a data transformation) or an interaction with one of the services participating in the composition (the *component services*). Such process models are captured using languages such as the Business Process Execution Language (BPEL) or the Business Process Model and Notation (BPMN).

In mainstream service composition platforms, the responsibility for coordinating the execution of a composite service lies on a single entity, namely the *orchestrator*. The orchestrator handles incoming requests for the composite service and interacts with the component services in order to fulfill these requests. Every time a component service completes an activity, it sends a message back to the orchestrator with its output data. The orchestrator then determines which component services need to be invoked next and forwards them the required input data. This architecture is not optimal in terms of communication overhead and has the usual problems of a single point of failure [6].

In previous work, we proposed a method for executing composite services in a decentralized manner [14]. The idea is to group activities into partitions and to assign each partition to a separate orchestrator. In [14], partitions are chosen manually by service designers. Designers may opt, for example, to put all activities invoking the same service into a partition, or to put all activities invoking services in a given organizational domain into a single partition, or any other partitioning criterion of their choice. Clearly, the performance and robustness of a decentralized composite service would benefit from the services in a given partition being close to one another and to the partition's orchestrator. But neither the above manual partitioning method nor other decentralized orchestration methods [23,11,36,6] help designers to optimize the communication overhead between component services.

This paper presents a method for partitioning activities in an orchestration and assigning services to activities, in such a way as to minimize the overall communication overhead, while at the same maximizing the Quality of Service (QoS) of the composite service. The proposed method also allows designers to keep control over the placement of activities. Specifically, designers may specify *collocation* and *separation* constraints between pairs of activities. A collocation constraint states that two activities must be placed in the same partition (e.g. because they are performed by services from the same company), while a separation constraint imposes that two activities must be in different partitions.

The proposed method needs to deal with an optimization problem involving different types of constraints and inter-related optimization variables: QoS variables, location variables, collocation and separation constraints. Various approaches have been proposed that deal with communication optimization in distributed and parallel systems [7,33,35,16]. These techniques could be adapted in order to partition the activities in a service orchestration in such a way as to minimize inter-partition communication. However, the existence of collocation and separation constraints, and the need to trade-off communication overhead and QoS when mapping activities to concrete services, makes the problem at hand different from previously studied ones.

In this paper, we formulate the above optimization problem as a Quadratic Optimization Problem (QAP) [4]. Given the inherent complexity of the QAP problem, we apply well-known heuristic optimization techniques [8] to explore the search space. Specifically, we present and evaluate a greedy algorithm to build an initial solution, followed by a Tabu search [15] to improve over the initial solution. The crux of these heuristics is to place activities that communicate frequently in the same partition (taking into account the collocation and separation constraints) and to assign services to activities in such a way as to optimize a function that captures intra-partition communication cost, inter-partition communication cost, and aggregate QoS.

The rest of this paper is structured as follows. Section 2 introduces a motivating example and uses it to illustrate the importance of choosing the right partitioning for decentralized orchestration. Next, sections 3-5 describe the details of the proposed method while Section 6 presents an experimental evaluation of the method. Section 7 discusses related work and Section 8 summarizes the contribution and outlines future directions.

2 Illustration

2.1 Motivating example

To motivate and illustrate the method presented in this paper, we make use of a sample orchestration model taken from [38] (cf. Figure 1). This orchestration model encodes a claims handling process at an insurance company IC. The orchestration model is captured in the BPMN notation, and it includes both control and data dependencies. Activity nodes have labels of the form a_i:S where the a_i is the activity identifier and S is the identifier of the invoked service. We assume for the time being that each activity has already been assigned to a component service. We will discuss later how this assignment is done in an optimized manner.

Before this process starts, it is assumed that the policyholder has contacted the Emergency Service (ES) to report an accident. ES provides emergency call answering service to policyholders and liaises

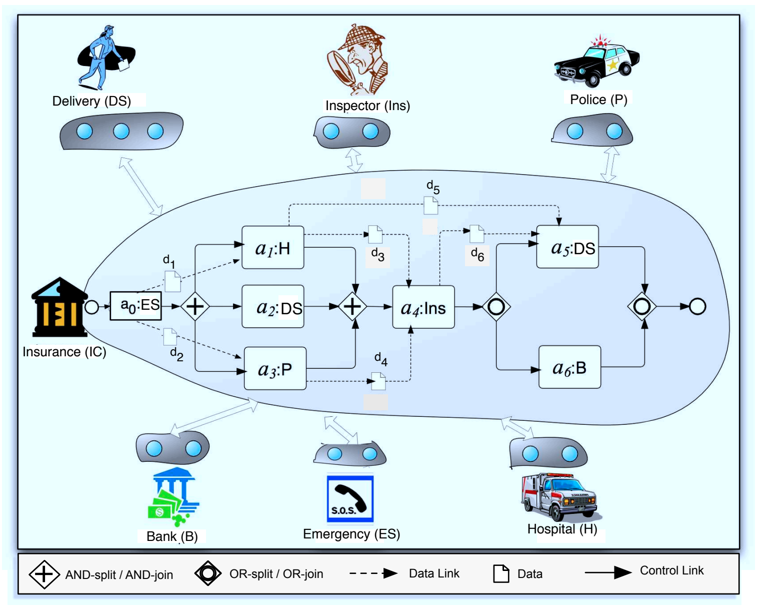


Fig. 1 Motivating example

with the hospital (Hospital) and the traffic patrol (Police). Some time after the accident, the policyholder contacts IC for reimbursement. In order to handle the claim, IC executes the orchestration depicted in Figure 1. First, IC invokes ES to obtain details about the incident (activity a_0). ES provides the protocol numbers that are required by Hospital (H) and Police (P) services, in order to release the respective incident reports. These dependencies are denoted d_1 and d_2. With the details provided by ES, IC invokes P and H concurrently. Additionally, Delivery Service (DS) is invoked in order to pick up the physical claim documents from the customer (activity a_2). Note that a_2 is executed after a_0 but it does not have a data dependency with it, while there are data dependencies between a_0 and a_1 and a_0 and a_3. IC uses the output obtained from P and H in order to invoke the Inspection Service (Ins) (activity a_4). Again note that, there are data dependencies between a_1 and a_4, a_3 and a_4 but not between a_2 and a_4. Service Ins decides whether the claim must be reimbursed or not. If so, the report provided by H (data dependency d_5) and the results of inspection (d_6) are sent to the policyholder by invoking DS (activity a_5). Moreover, a Bank (B) service is invoked for the reimbursement. If the claim is not reimbursable, B is not invoked. This is why an OR-split/OR-join is used: sometimes both DS and B are invoked, and other times only DS is invoked.

In existing service orchestration platforms (e.g. BPMN or BPEL engines), control and data dependencies between services are managed centrally by IC. The resulting interactions between IC and the component services are hence as depicted in Figure 2a. The centralized orchestrator is a bottleneck and may cause performance degradation and availability issues. It also causes additional traffic of messages, since every activity execution involves a back-and-forth message exchange between IC and a service, which may be located arbitrarily apart and in a different organizational domain. An alternative is to execute the orchestration in a decentralized manner.

Figure 2b depicts a possible decentralized execution settings for the same orchestration, where IC is partitioned into seven partitions that are executed by seven distributed orchestrators. The orchestrator assigned to a partition P_i is responsible for coordinating the services assigned to the activities of this partition. In this decentralized architecture, the data produced by a service are routed directly to the partitions of the services that consume these data (meaning to the orchestrator of that partition). For example, *hospital* and *police protocols* (d_1 and d_2) generated by ES are routed directly to the partitions of services H and P. If we consider the data exchanged only between services, then the number of data

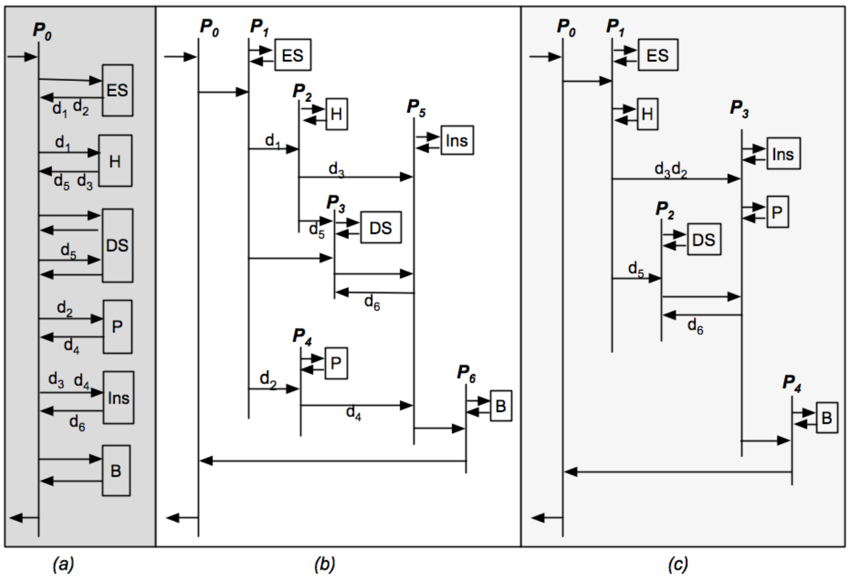


Fig. 2 (a) centralized model (b) First decentralized model (c) Second decentralized model

flow messages in figure 2a is 8 (cf. communication links labelled with data items d_i). Meanwhile, in decentralized orchestration depicted in Figure 2b, the number of messages is reduced to 6 since data are transferred directly from their sources to their points of consumption.

Now consider the case where ES and H are geographically close to each other, and the same holds for P and Ins. Then, it is preferable to create a single partition for ES and H, and same for P and Ins. This arrangement reduces the number of data flows exchanged between partitions to only 3 messages.
The example shows that that the communication overhead varies depending on the number of partitions, the placement of activities into partitions, the distance between services, and the number of message exchanges.

2.2 Problem statement

Consider the decentralized orchestration example depicted in figure 3. The example shows three geographically distributed partitions. The orchestrators corresponding to these partitions communicate through an asynchronous messages exchange mechanism. Messages represent control or data informations. Each partition includes a set of activities each of which has at least one candidate service which can perform this activity. For instance, partition $\mathcal{P}_1$ contains activities a_1 and a_2 which communicate with other partitions' activities a_4 and a_7 respectively. The activity a_1 may invoke either s_{5a} or s_{5b}. Each service has a location and a QoS determined by its response time, availability and reliability. The bold lines represent intense communication between pairs of activities and thin lines describe a poor communication. Arrows represent services invocations. In this example, activities a_1 and a_4 belong to different partitions but communicate rather intensely. While a_1 has a low communication with a_3 in the same partition $\mathcal{P}_1$, a_4 has an intense communication with a_3 in the same partition $\mathcal{P}_3$. This example allows us to illustrate the tradeoffs involved, for instance:

- is it better to keep a_1 in partition $\mathcal{P}_1$ near the service it invokes?
- or is it better to put a_1 in the same partition as a_4 since they communicate a lot? in this case, should we preserve the assignment of a_1 to s_{5a} or reassign a_1 to the service s_{5b} which is nearer to $\mathcal{P}_3$? and what if s_{5a} has a better quality than s_{5b}?
- is it more judicious to put a_4 in $\mathcal{P}_1$ since s_7 is more nearer to $\mathcal{P}_1$ than to $\mathcal{P}_3$ and a_4 has an intense communication with a_3 in $\mathcal{P}_3$?

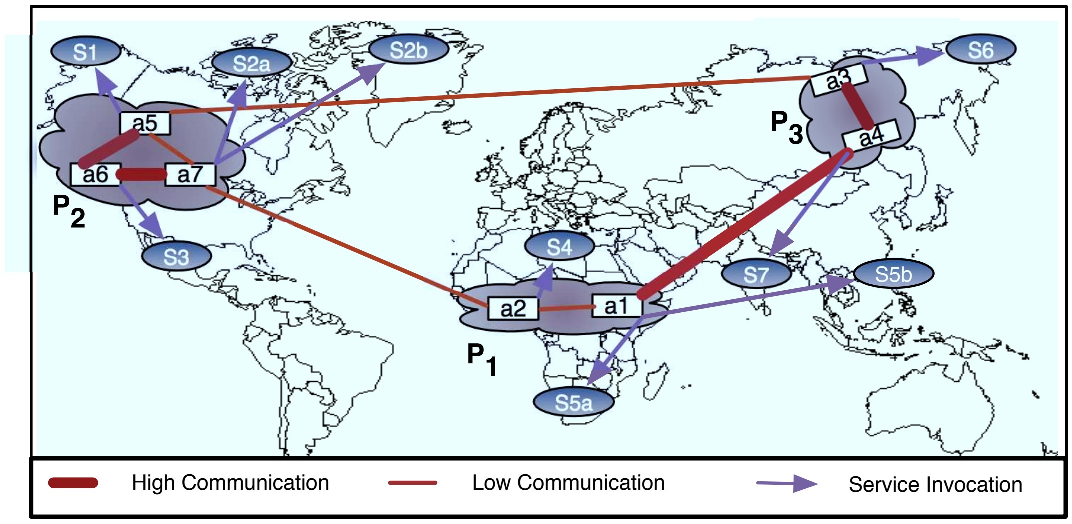


Fig. 3 Decentralized orchestration model

– in partition $\mathcal{P}_2$ should a_4 be assigned to s_{2a} which is located near to it or the service s_{2b} which has better quality?
– does this number of partitions is optimal? does the placement of partitions is good? is there any risk of overload if we increase the activities number of a partition?
– if we move an activity to another partition, is there any risk of security constraints violation?

This paper takes into consideration the above kinds of tradeoffs in order to compute an optimized partition for decentralized orchestration of a composite service. Essentially, the idea is that the services in the same partition should be close to one another. In this way, the orchestrator of the partition can be placed close to all the services that it orchestrates and the communication overhead created by messages exchanged between services in the same partition is minimal or (ideally) negligible. On the other hand, services in different partitions may be far from one another, so the communication volume between services in different partitions should be as low as possible.

In addition to seeking to minimize communication overhead, the proposed method also takes into account the QoS of each service. Specifically, we consider the case where there are multiple candidate services that can perform a given activity. Each of these services offers a QoS and has a location. The method seeks to assign services to activities and to place activities in partitions in such a way as to strike a tradeoff between minimizing the communication overhead and maximizing overall QoS. Relative weights are assigned to each factor in order to capture their relative importance. Before describing the optimized partitioning method, we define the notion of service orchestration model and related notions (Section 3). We then show how the communication overhead between pairs of activities is computed by analyzing the orchestration models (Section 4).

3 Preliminaries

The method for optimized service selection takes as input a service orchestration consisting of activities related by control, data-flow and distribution constraints.

3.1 Orchestration model

In order to precisely define the notion of service orchestration, we need to adopt a model for representing control-flow relations between activities. In this paper, we adopt a structured representation of process models. In essence a process model is represented as a tree whose leaves represent activities and whose internal nodes represent either sequence (SEQ), parallel (PAR), choice (CHC) or repeat loop (RPT) constructs. Structured process models are very close to BPEL, and they have the advantage of being simpler to analyze. And while it is possible to write unstructured models both in BPEL and in BPMN, recent work has shown that most unstructured process models can be automatically translated into structured ones [29]. Note that for the purpose of the proposed method, we do not need to capture concrete branching expressions. Instead, it is sufficient to know the probability of taking each conditional branch in a choice and the probability of taking the "repeat" branch in a loop. Also, we do not need to capture OR-split/OR-join pairs, because when a process is structured, OR-split/OR-join can be trivially translated into a combination of choice and parallel blocks. For example, the OR-split/OR-join pair in Figure 1 can be transformed into a choice between executing a_5 only or executing both a_5 and a_6 in parallel. Formally, we capture structured process models as follows.

Definition 1 *[(Structured) Process Model] A process model is a tree with the following structure (here we use the type definition syntax of the ML language [26]):*

$$\begin{aligned}
Process &::= \mathit{ProcNode} \\
ProcNode &::= \mathit{Activity} \mid \mathit{ControlNode} \\
ControlNode &::= \mathit{SEQ}([\mathit{ProcNode}]) \\
&\quad \mid \mathit{CHC}([\mathit{CondBranch}]) \mid \\
&\quad \mid \mathit{PAR}(\{\mathit{ProcNode}\}) \\
&\quad \mid RPT(\mathit{ProcNode} \times P) \\
\mathit{CondBranch} &::= \mathit{COND}\ (P \times \mathit{ProcNode})
\end{aligned}$$

where P is the range of real numbers from 0.0 *to* 1.0*, denoting probabilities.*

For example, the BPMN model in Figure 1 is represented by the following expression:
SEQ(a_0, PAR(a_1, a_2, a_3), a_4, CHC(COND(p_1, a_5), COND(p_2, PAR(a_5, a_6)))).

An activity in a service orchestration represents a one-way or a bidirectional interaction with a service via the invocation of one of its operations. Each activity has a non-empty set of candidate services that it can be bound with. In addition, activities may be related by means of two types of distribution constraints: collocation (activities must be placed in the same partition), and separation (activities must be placed in different partitions). Formally, a service orchestration is defined as follows:

Definition 2 *[Service Orchestration] A* service orchestration *is a tuple* (Proc, Data. Cand, Collocate, Separate)*, where:*

– Proc *is a process model capturing control-flow dependencies between a set of activities;*
– Data *is a ternary relation consisting of tuples of the form Data(a_i, a_j,d_k) stating that, upon completion of activity a_i, data item d_k needs to be transferred to activity a_j*
– Cand *is a function that maps each activity to a set of candidate services that are able to perform that activity.*
– Collocate *is a relation consisting of facts of the form Collocate(a_i, a_j) stating that the activities a_1 and a_2 must be placed together;*
– Separate *is a relation consisting of facts of the form Separate(a_i, a_j) stating that the activities a_1 and a_2 must be placed in different partitions.*

For consistency, we impose that $\forall a_1,\ a_2\ \neg(Collocate^{+}(a_1,\ a_2) \wedge Separate(a_1, a_2))$ where $Collocate^{+}$ is the transitive closure of relation Collocate. This means that if we declare that two activities must be collocated, we cannot state additionally that these activities must be separated.

An activity that is not related with any other activity by a collocate or separate constraint is called an *unconstrained activity.* In the sequel, we write CTR to denote the set of all distribution constraints defined in an orchestration ($CTR = \mathit{Collocate} \cup \mathit{Separate}$). Also, we write $\mathit{Act}(\mathit{Orc})$ to refer to the set of activities of an orchestration, $CA(\mathit{Orc})$ to denote the set of constrained activities and $NCA(\mathit{Orc})$ to denote the set of unconstrained activities. Unconstrained activities are also called *flexible activities* since we can place them in any partition. When it is clear to which orchestration we are referring to, we will simply write Act, CA and NCA.

3.2 QoS Model

Given a service orchestration defined as above, the purpose of the method is to construct:

- A binding $\mathit{bind}(a)$, that is, is a function that maps each activity a in the orchestration model to a service s;
- A partitioning of activities, that is, a function that maps each activity in an orchestration to a partition. This partition function is needed for decentralized service orchestration.

The method seeks to bind candidate services to activities in such a way as to minimize the communication overhead and to maximize the QoS of the services in the binding. Composite service designers are able to influence the relative importance given to the minimization of the communication overhead versus the maximization of the quality by setting two weights: $w_c \in [0..1]$ is the weight given to the communication overhead and $w_q \in [0..1]$ is the weight given to the quality of service.

We do not impose a particular model for calculating the QoS of a given service. Instead, we assume that there is a function $QoS(s)$ that returns the QoS of a service s. For example, we could use the QoS model presented in [39] in order to determine the QoS of each component service based on a weighted sum of a pre-determined set of QoS attributes such as:

- **Execution cost.** The execution cost $QoS^{cost}(s)$ of a service s is the fee that a service requester has to pay for invoking the service. Web service providers either advertise the execution cost of their services, or provide means for potential requesters to inquire about it.
- **Execution time.** The execution time $QoS^{time}(s)$ of a service s measures the expected delay in seconds between the moment when a request is sent and the moment when the results are received. Services advertise their processing time or provide methods to inquire about it.
- **Reputation.** The reputation $QoS^{rep}(s)$ of a service s is a measure of its trustworthiness. It mainly depends on end user's experiences of using the service s. Different end users may have different opinions on the same service, hence, the average of the raking given by end users is to be considered.
- **Reliability.** The reliability $QoS^{rel}(s)$ of a service s is the probability that a request is correctly responded (i.e., the operation is completed and a message indicating that the execution has been successfully completed is received by service requestor) within the maximum expected time frame indicated in the Web service description. The reliability is a measure related to hardware and/or software configuration of Web services and the network connections between the service requesters and providers.
- **Availability.** The availability $QoS^{av}(s)$ of a service s is the probability that the service is accessible for immediate use. Availability is also described as the uptime of a service in a pre-determined period.

Given the QoS of each service – QoS_s – computed for example as a weighted sum of the above QoS attributes, the proposed method seeks to bind services to activities in a service orchestration in such a way as to minimize the *aggregate QoS* of the service orchestration, where the aggregate QoS is the weighted sum of the QoS of the services bound to each activity, taking as weights the frequency of execution of the activity in question. In other words, given a service orchestration involving activities a_1, ..., a_n, and given a binding *bind* that maps each activity to a service, we seek to maximize:

$$\sum_{i=1}^{n} QoS_{bind(a_i)}$$

while at the same time maximizing the communication cost of the binding as discussed later. Note that the above aggregation function is not the only possible one. A classification of other types of aggregation function together with methods to efficiently compute those aggregation functions can be found in [37].

4 Communication Overhead

One of the aims of the optimized partitioning approach is to produce partitions such that the communication overhead (i.e. the amount of communication) between activities inside a partition is as large as possible and, conversely, the communication overhead across partitions is as small as possible. To construct such optimized partitions, we need to estimate the communication overhead between pairs of activities. Two activities a_1 and a_2 need to communicate if:

– Activities a_1 and a_2 are consecutive. If we take the representation of a process model as a graph consisting of activities and gateways (as in Figure 1), two activities are consecutive if there is a control-flow arc directly from a_1 to a_2, or there is a path from a_1 to a_2 that does not traverse any other activity (i.e. only gateways are traversed). In this case, every time an instance of activity a_1 completes, if activity a_2 needs to be executed next, the service assigned to a_1 must send a control-flow notification to the service attached to a_2.
– There exists a data-flow from activity a_1 to activity a_2 $(a_1, a_2, d) \in Data$. The presence of such a data flow implies that every time activity a_1 completes, the service assigned to a_1 must send a message containing a datum of type d to the service assigned to a_2.

Without loss of generality, we measure communication overhead in bytes. We assume that control-flow notification has a size of one byte. We also assume that the average size in bytes of a message of type d is known, and we write $size(d)$ to denote this size. In order to determine how many bytes will be exchanged between the service assigned to a_1 and the service assigned to a_2 during one execution of an orchestration, we need to determine two things:

– How many times a given activity will be executed (for a given execution of the orchestration)? We write $numExec(a)$ to denote this amount.
– Given two consecutive activities a_1 and a_2, what is the probability that one execution of activity a_1 is immediately followed by an execution of activity a_2. We write $probFollows(a_1, a_2)$ to denote this probability.

4.1 Computation of numExec(a)

To compute the number of times that a given activity is executed we reason on the structured process model (as defined in Definition 1), and make the following observations:

– If a process node PN is a direct child of a sequence (SEQ) node, then each execution of the SEQ node entails one execution of PN
– If a process node PN is a direct child of a parallel (PAR) node, then each execution of the PAR node entails one execution of PN
– If a process node PN is a direct child of a conditionalBranch (COND) node that has a branching probability of p, then each evaluation of node COND entails p executions of PN.
– If a process node PN is a direct child of a Repeat (RPT) node that has a repeat probability of p, then each execution of the node RPT entails $1/(1-p)$ executions of PN.

Algorithme 1 : Algorithm $numExec(a)$

```
Input: orc // an Orchestration
       a // an activity in Act(orc)
path ← the path from the root of Proc(orc) to a
condBranches ← the list of COND nodes in path
repeatBlocks ← the list of RPT nodes in path
Output: $(\Pi_{cb \in condBranches} prob(cb) \times (\Pi_{rb \in repeatBlocks} 1/(1 - prob(rb)))$
```

Based on these observations, we conclude that the number of times an activity a needs to be executed (for a given execution of an orchestration) is determined by the probabilities of the conditional branch and repeat nodes that appear in the path from the root of the process model to a. Starting from one execution of the entire process, each time a COND node with probability p is traversed, the number of executions of its child node is multiplied by p, while every time a RPT node is traversed the number of executions is multiplied by $1/(1 - p)$. This observation leads us to Algorithm 1 that calculates the average number of times that a given activity is executed for each execution of an orchestration. In this algorithm, $prob(cb)$ and $prob(rb)$ denote the probability attached to conditional branch cb or a repeat block rb respectively.

4.2 Computation of probFollow(a_1, a_2)

Next, we have to compute $probFollows(a_1, a_2)$: the probability that the completion of an instance of activity a_1 triggers the execution of another activity a_2 – assuming that a_1 and a_2 are consecutive activities. For this, it is more convenient to take the representation of the process model as a graph consisting of activities and gateways, and to retrieve the conditional control-flow arcs traversed on the path from a_1 to a_2. Here, a conditional control-flow arc is an arc in the process graph whose source is an XOR gateway. For each traversed conditional control-flow arc, the $probFollows(a_1, a_2)$ is multiplied by the probability attached to the control-flow arc. This leads to the Algorithm 2. In this algorithm, *prob*(*ca*) denotes the probability associated to a conditional control-flow arc *ca*.

Algorithme 2 : Algorithm $probFollows(a_1, a_2)$

```
Input: orc // an Orchestration
       $a_1, a_2$ // two consecutive activities in Act(orc)
path ← the path in the process graph from $a_1$ to $a_2$
condArcs ← the list of conditional control-flow arcs in path
Output: $\Pi_{ca \in condArcs} prob(ca)$
```

4.3 Communication cost co(a_1, a_2)

Having defined functions $numExec$ and $probFollows$ and given the above observations, the communication overhead between two activities a_1 and a_2 – namely $co(a_1, a_2)$ – is computed as follows:

$$\begin{aligned} co(a_1, a_2) = {} & Cons(a_1, a_2) \times numExec(a_1) \times probFollows(a1, a_2) \\ & + \Sigma_{(a_1, a_2, d) \in Data} numExec(a_1) * size(d) \end{aligned} \qquad (1)$$

where $Cons(a_1, a_2)$ is a function equal to one if a_1 and a_2 are consecutive activities, and zero otherwise. The first term in this formula corresponds to the communication overhead induced by control-flow notifications, while the second term corresponds to the communication overhead induced by data-flows. Note that $probFollows$ does not appear in the second term, because a data-flow dependency implies that the source activity will send the corresponding datum to the target activity, regardless of whether or not the target activity is performed.

5 Partitioning Approach

Given a centralized process specification, our decentralized orchestration is composed of two parts. The first step consists in determining an optimized partitioning of activities and an optimized assignment of services to activities in order to reduce communication overhead and maximize QoS. This is the subject of this paper. The second part consists in wiring the activities in the same partition and across partitions in order to preserve the semantics of the process model. This wiring means that data and control dependencies need to be realized by means of message exchanges between services and distributed orchestrators assigned to each partition. For this part, we can use a technique we presented in previous work [14] or other techniques discussed in Section IV.

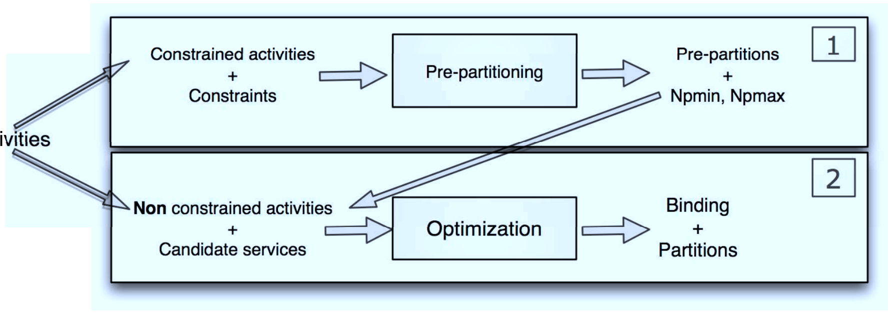

Activities
Constrained activities + Constraints
Pre-partitioning
Pre-partitions + Npmin, Npmax
1
Non constrained activities + Candidate services
Optimization
Binding + Partitions
2

Fig. 4 Partitioning steps

In order to compute an optimized partitioning of an orchestration, we proceed in two steps (c.f. figure 4). First, we perform a pre-partitioning in which activities that are related through Collocate relations are put in the same partitions. In this pre-partitioning phase (Section 5.1), we also construct "groups of partitions" such that activities across different groups are not related neither by Separate nor by Collocate constraints. This pre-partitioning is useful since we can then easily identify which activities must be collocated, and which sets of activities must be kept separated. In the second step, we use this pre-partitioning in order to form final partitions using a Greedy algorithm. We also sketch how the initial solution computed by the Greedy algorithm can be improved using Tabu search.

Next, we introduce the pre-partitioning algorithm as well as an algorithm for calculating the minimum and maximum amount of final partitions to be created (Section 5.1). Finally, using the pre-partitioning and the function for computing communication overhead, we show how the final partitioning is computed (Section 5.2).

5.1 Pre-partitioning of Constrained Activities

The purpose of the pre-partitioning phase is to partition the set of constrained activities CA so that we can later easily identify which activities should be collocated and which activities should be separated. To this end, we decompose the set of activities into groups $\{CA_1 \ldots CA_n\}$, so that elements in two groups are not related neither by a *Separate* nor by a *Collocate* constraint. In other words, if we view the relation $CTR = Separate \cup Collocate$ as a graph, a group consists of all activities in one of the connected components of this graph. Figure 5 shows an example involving 12 activities $CA = \{a_1, .., a_{12}\}$ linked through *Separate* and *Collocate* relations. Looking at the corresponding CTR relation, we can see that there are three connected components in the induced graph, and thus three groups are created, namely CA_1, CA_2 and CA_3. If we restrict the relation CTR to the activities in each of these groups, we obtain three restricted CTR relations, namely CTR_1, CTR_2 and CTR_3 respectively.[1] The rationale for this

[1] We note that $\forall i, j, i \neq j$, $CA_i \cap CA_j = \{\emptyset\}$ and $CTR_i \cap CTR_j = \{\emptyset\}$.

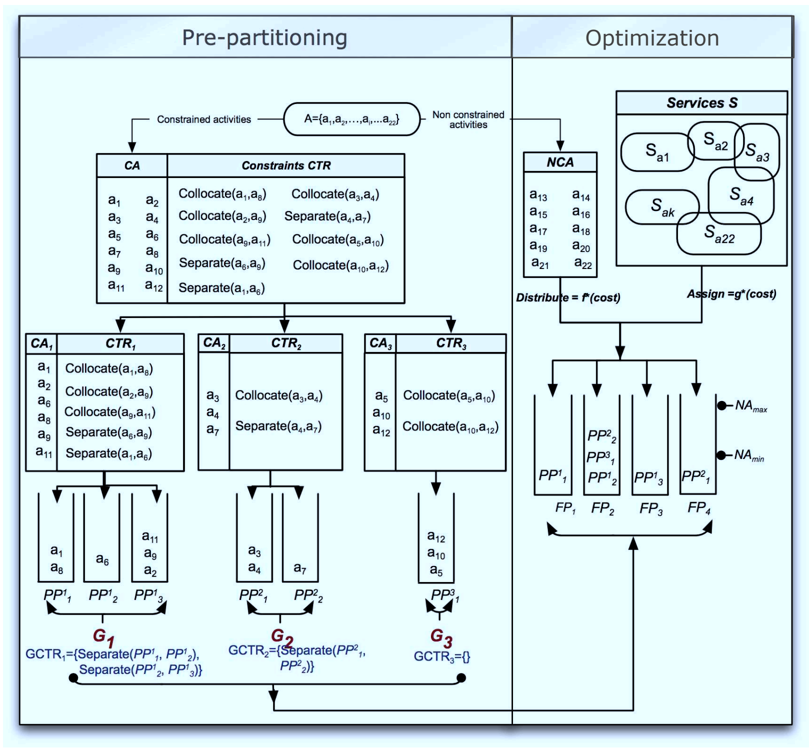


Fig. 5 Partitioning process

initial grouping is that activities belonging to different groups can be freely combined with one another in a final partition (or they can be left in separate final partitions), because no constraint links them.

Next, each group is further partitioned into a number of *pre-partitions* by looking at the relation *Collocate* only. The idea is that each of these partitions is a maximal set of activities that must be collocated. In other words, if we view the relation *Collocate* as a graph, a partition in a group CA_k consists of all activities in CA_k that belong to one of the connected components of this graph. The pre-partitioning of each group CA_k is a set of *pre-partitions* such that $G_k{=}\bigcup PP_k^j$. For example, in Figure 5, CA_1 is decomposed into three *pre-partitions*: $PP_1^1{=}\{a_1{,}a_8\}$, $PP_1^2{=}\{a_6\}$ and $PP_1^3{=}\{a_9{,}a_{11}{,}a_2\}$. After the pre-partitoning phase, we know that all activities in a pre-partitions should be manipulated as a single package and put together in one final partition.

This pre-partitioning is operationalized by algorithm 3. This algorithm first computes the groups by calculating the connected components CTR_i of CTR. Each CTR_i leads to one group. Next, the algorithm computes the partitions within each group by computing the connected components of the *Collocate* relation restricted to the connected component CTR_i. For convenience, we lift the relation *Separate* so that it can be applied to partitions as follows:

$$Separate(P_i, P_j) \Leftrightarrow \exists a_i \in P_i, a_j \in P_j : Separate(a_i, a_j)$$

For example, with respect to Figure 5, it holds that $Separate(PP_1^1, PP_2^1) \wedge Separate(PP_2^1, PP_3^1)$. This implies that PP_2^1 should not be combined neither with PP_1^1 nor with PP_3^1 in the same final partition.

The final partitioning algorithm presented later tries to compute partitions of different sizes. To this end, we need to know the approximate minimum and maximum number of possible final partitions FP_j. Algorithm 4 describes how to compute the minimum required final partitions that can be obtained by merging pre-partitions from different groups, while respecting the constraints that link *pre-partitions* of the same group. However, this number does not take into consideration *non-constrained* activities NCA. So, to have the exact number, consider $|Act|$ the total number of activities, NA_{max} (NA_{min}) the maximum (minimum) number of allowed activities by partition (fixed by user after constrained

Algorithme 3 : Constrained activities partitioning

```
Require: - $CTR$: set of all constraints
Init: $Groups \leftarrow \{\}$
begin
    for each $CTR_i$ in ConnectedComponent($CTR$) do
        $CurGroup \leftarrow \{\}$
        for $Collocate_i$ in $ConnectedComponent(CTR_i \cap Collocate)$ do
            $NewPartition \leftarrow \{a | \exists a' \; Collocate_i(a, a')\}$
            $CurGroup \leftarrow CurGroup \cup \{NewPartition\}$
        $Groups \leftarrow Groups \cup \{CurGroup\}$
    Return Groups
end
Result: groups of constrained partitions
```

activities partitioning), NP the output of algorithm 4, and $|CA|$ ($|NCA|$) the number of *constrained (Non-constrained)* activities. Then the minimum and maximum number of final partitions NP_{min} and NP_{max} are computed by equations 2 and 3, respectively. In Section 3.4, we will vary the number of partitions from NP_{min} to NP_{max} and try to distribute the flexible activities FA and the groups G_k over those partitions in such a way as to minimize the communication overhead and maximize the QoS. We will then choose the partitioning that leads to the best overall tradeoff between communication overhead and QoS according to relative weights given by the user.

$$NP_{min} = \begin{cases} NP \qquad if \ \dfrac{|Act|}{NA_{max}} \leq NP \\ \\ NP + \dfrac{|Act| - (NP * NA_{max})}{NA_{max}} \ Otherwise \end{cases} \tag{2}$$

$$NP_{max} = \sum_{k} Size(G_k) + \frac{|NCA|}{NA_{min}} \tag{3}$$

5.2 Optimized partitioning process

In the previous sections, we presented algorithms to partition constrained activities into a set of independent partition groups G_k (*pre-partitions*), while respecting constraints defined by user. We also introduced algorithms to compute the minimal and maximum number of final partitions FP_j. In the following, we will present our solution to optimally distribute the *pre-partitions* and unconstrained activities over final partitions, and assign activities to web services. The problem can be considered as a quadratic assignment problem (QAP) introduced by Koopmans and Beckmann [4] in 1957, as a mathematical model for the location of a set of indivisible economical activities. Using the QAP formulation of Koopmans-Beckman, we are given a cost matrix $C = [co_{ij}]$, where co_{ij} is the communication overhead between activity a_i and activity a_j. We are also given a distance matrix between partitions $D^p = [d^p_{ij}]$, where d^p_{ij} represents the distance between partition P_i and partition P_j, a distance matrix between services $D^s = [d^s_{ij}]$ where d^s_{ij} represents the distance between service s_i and service s_j and a quality matrix Q=$[q_{ij}]$, where q_{ij} is the contribution to overall QoS obtained by assigning activity a_i to service s_j.

Given the above matrices, if activity i is assigned to service $bind(i)$, the contribution of this assignment to the overall QoS is equal to the QoS of service $bind(i)$ multiplied by the average number of times that a_i is executed per execution of the orchestration, i.e. $numExec(a_i)$ as defined above. Meanwhile, if activity i is assigned to $P(i)$, and activity j is assigned to $P(j)$, the inter-partition communication cost associated with this assignment is $co_{ij} \cdot d^p_{P(i),P(j)}$. Finally, if activity i is assigned to $bind(i)$, and activity j is assigned to $bind(j)$, the intra-partition distance cost associated with this assignment is $co_{ij} \cdot d^s_{bind(i),bind(j)}$. Note

Algorithme 4 : Computing approximative minimum number of partitions after groups merging

```
Require: - $Groups = \cup G_k$ // The set of all partition groups
- $NA_{max}$ // The maximum number of activities by partition
Init: $Ng \leftarrow |Groups|$
$Ng_{max} \leftarrow Max(|G_k|),\ k \in [1..Ng]$
Recursive($Groups$, $Ng_{max}$)
begin
    if $(G_k = \{\},\ \forall\ k \neq Ng_{max})$ then
    return $Groups$
    for $(G_k\ in\ Groups,\ k \neq Ng_{max})$ do
        for $(P_i^k\ in\ G_k)$ do
            $min \leftarrow Min(|P_l^{Ng_{max}}|)\ l \in [1..|G^{Ng_{max}}|]$
            if $(|P_i^k| + |P_{min}^{Ng_{max}}| > NA_{max})$ then
            $Add(P_i^k,\ G_{Ng_{max}})$
             $Delete(P_i^k,\ G_k)$
    repeat
        $P_{max} \leftarrow Max(P_i^k)\ st\ \neg constrained(Max(P_i^k),\ P_{min}^{Ng_{max}})\ \forall k \neq Ng_{max},\ \forall i \in [1..|G_k|]$
        $Add(P_{max},\ P_{min}^{Ngmax})$
        $Delete(P_{max})$
    until $((G_k = \{\} \forall k \neq Ng_{max}) \vee (|P_{max}| + |P_{min}^{Ng_{max}}| > NA_{max})$
    Recursive($Groups$, $Ng_{max}$)
end
Result: $NP$=Size(Recursive($Groups$, $Ng_{max}$))
```

that $bind(i)$ and $bind(j)$ are subject to the constraints $bind(i) \in Cand(i)$ and $bind(j) \in Cand(j)$, meaning that an activity can only be bound to one of its candidate services.

The optimization problem has three components: we have to maximize the quality of service, minimize the inter-partition communication cost – because it implies communication between orchestrators possibly located far from one another – and we have to minimize the distance between services placed in the same partition – given that such services need to interact with a local orchestrator. Because we wish to strike a tradeoff between three factors, we introduce three parameters w_q, w_{out} and w_{in}, where w_q is the relative weight given to maximizing QoS, w_{out} is the weight given to minimizing inter-partition communication cost, and w_{in} is the weight given to minimizing the distance between services assigned to activities in the same partition.

Given these weights, the total cost of a solution to this assignment problem is given by equation 4. An optimal solution to the problem consists of an assignment of activities to partitions and a binding of activities to services such that this total cost is minimal. Solutions are only admissible if they respect the binding constraints (a service can only be assigned to an activity if it is one of the candidates of this activity), and the collocation and separation constraints for assigning activities to partitions. In equation 4 we write $1 - QoS_s$ because we seek to maximize the sum of QoS of the services in the binding, which is equivalent to minimizing $1 - QoS_s$. This discussion leads us to the following *objective function*:

$$w_q \sum_{i=1}^{n} (1 - QoS_{bind(i)}) \times numExec(i)$$
$$+ w_{out} \sum_{i=1}^{n} \sum_{j=1}^{m} co_{ij} \times d_{P(i)P(j)} + w_{in} \sum_{i=1}^{n} \sum_{j=1}^{n} co_{ij} \times d_{bind(i),bind(j)} \tag{4}$$

We note that in this objective function, we try to optimize the weighted sum of QoS, taking into account the number of times that each service will be invoked for a given execution of the orchestration. In certain cases, one may wish to optimize a different QoS aggregation function (e.g. the product of the

QoS of the services) as discussed in [12]. If this is the case, one can replace the QoS term in the above formula and replace it with the desired aggregation function. The proposed optimization technique would need to be adapted accordingly, but the basic principles remain applicable.

It should be noted that the equation 4 includes three main terms: (i) the first concerns the quality of service, (ii) the second describes the inter-partitions communication and (iii) the third is related to the intra-partitions communication. In order to correctly evaluate the communication cost, these terms should be in the same values domain. For this purpose we consider the normalization process defined in the equations 5 as follows:

$$\textbf{(1)} \qquad w_q \cdot \frac{\sum_{i=1}^{n}(1 - QdS_{bind(i)}) \times numExcec(i)}{\sum_{i=1}^{n} numExec(i)} \tag{5}$$

$$\textbf{(2)} \quad w_{out} \cdot \frac{\sum_{i=1}^{n}\sum_{j=1}^{n} co'_{ij} \times d_{P(i)P(j)}}{\sum_{i=1}^{n}\sum_{j=1}^{n} d_{P(i)P(j)}} \quad with \begin{cases} \text{N: activities number} \\ co'_{ij} = \dfrac{co_{ij}}{M}, \quad M = Max_{i,j=1..n} co_{ij} \end{cases}$$

$$\textbf{(3)} \quad w_{in} \cdot \frac{1}{m \times M_d} \times \sum_{k=1}^{m} InDistances(P_k) \quad with \begin{cases} \text{m: partitions number} \\ M_d = Max_{k=1,m} InDistances(P_k) \end{cases}$$

$$\text{and } InDistances(P_k) = \frac{\sum_{i=1}^{size(P_k)}\sum_{j=1}^{size(P_k)} co_{ij} \times d_{bind(i)bind(j)}}{\sum_{i=1}^{size(P_k)}\sum_{j=1}^{size(P_k)} co_{ij}}$$

In the first term (1), we assume that the quality of service varies between 0 and 1 and therefore 1-$QoS_{bind(i)} \in [0..1]$. The execution number of a given activity varies between 0 and ∞, so we divide it by the sum of execution numbers of all the activities. In the second term (2) (inter-partitions communication), we divide the communication cost between each couple of activities by the highest communication cost found. We also divide each inter-partitions distance $d_{P(i)P(j)}$ by the sum of inter-partitions distances between all couples of activities. Finally to normalize the intra-partitions communication (3), we divide each intra-partitions distance by the partitions number multiplied by the maximal internal distance. The intra-partitions communication cost is divided by the sum of communication costs between all pairs of activities of the same partition. Each of the terms is multiplied by the correspondent weight.

5.2.1 Heuristic optimization algorithms overview The problem of optimizing equation 4 is a QAP because $d_{P(i)P(j)}$ depends on the partitions to which a_i is assigned and the one to which a_j is assigned. If we use a boolean (0-1) variable to encode to which partition a given activity is assigned, this term would involve a product of two variables. A similar remark applies to $d_{bind(i),bind(j)}$.

Several exact algorithms have been used for solving QAP, like branch and bound, cutting plane and branch and cut algorithms [8]. Although substantial improvements have been done in the development of exact algorithms for the QAP, they remain inefficient to solve problems with size n>20 in reasonable computational time (there are $n!$ distinct permutations). This makes the development of heuristic algorithms essential to provide good quality solutions in a reasonable time. Many research have been devoted to the development of such approaches. We distinguish the following heuristic algorithms [8]: Hill-climbing (HC), Tabu search (TS), Simulated annealing (SA), Genetic algorithms (GA), Greedy randomized adaptive search procedures (GRASP), Ant systems (AS), etc. The simpler and faster methods are based on local search (e.g. HC and TS). A local search procedure starts with an initial feasible solution and iteratively tries to improve the current solution. This is done by substituting the latter with a (better) feasible solution from its neighborhood. In HC, this iterative step is repeated until no further improvement can be found. For a comprehensive discussion of theoretical and practical aspects of local search in combinatorial optimization the reader is referred to [1]. In this paper we adopt the TS to look for an optimal solution to our decentralization problem. Tabu search [15] is a local search method where the basic idea is to remember which solutions have been already visited by the algorithm, in order to derive the promising directions for further search. A generic procedure starts with an initial feasible solution and selects a best-quality solution S among (a part of) the neighbors of S obtained by non-tabu moves. Then the current solution is updated by the selected solution. If there are no improving moves, tabu search chooses one that least degrades the objective function. The search stops when a stop criterion (running time limit, limited number of iterations) is fulfilled.

Algorithme 5 : Greedy algorithm: initial elite solution computation

```
Require: - $NCA(Orc)$, $NP_{min}$, $NP_{max}$
- $\mathcal{P}_c$: Constrained partitions ($pre-partitions$)
- $\{Cand(a_i), \forall a_i \in Act(Orc)\}$
Init: $\mathcal{P}_c \leftarrow P_c \cup \{\{a_i\} | a_i \in NCA(Orc)\ \}$
$bestQuality \leftarrow +\infty$, $bestNumber \leftarrow NP_{min}$
$bestPartition \leftarrow \{\}$, $bestBind \leftarrow \{\}$
Begin
for ($NP \leftarrow NP_{min}$ To $NP_{max}$) do
    $FinalPart \leftarrow$ a set of size $NP$ of empty sets
    for (each $PP$ in $\mathcal{P}_c$) do
        $Quality^* \leftarrow +\infty$
        for (each $FP \in [1..NP]$ where $\neg Separate(PP, FinalPart[FP])$) do
            $CurQual \leftarrow 0$
            for (each $a_i$ in $PP$) do
                $s_{a_i} \leftarrow \underset{s_i \in Cand(a_i)}{\arg\min} \Big[ w_q \cdot (1 - QoS(s_i)) \cdot numExec(s_i) + w_{out} \cdot \frac{\sum_{a_j \in FinalPart[FP]} co_{a_i,a_j} . d_{P(a_i),P(a_j))}}{|FinalPart[FP]|} + w_{in} \cdot \frac{\sum_{a_j \in FinalPart[FP]} d_{s_i,bind(a_j)}}{|FinalPart[FP]|} \Big]$
                $CurQual \leftarrow CurQual + \Big[ w_q \cdot (1 - QoS(s_i)) \cdot numExec(s_i) + w_{out} \cdot \frac{\sum_{a_j \in FinalPart[FP]} co_{a_i,a_j} . d_{P(a_i),P(a_j)}}{|FinalPart[FP]|} + w_{in} \cdot \frac{\sum_{a_j \in FinalPart[FP]} d_{s_i,bind(a_j)}}{|FinalPart[FP]|} \Big]$
            if $CurQual < Quality^*$ then
                $FP^* \leftarrow FP$
                $Quality^* \leftarrow CurQual$
                for ($a_i$ in $PP$) do $bind(a_i) \leftarrow s_{a_i}$
        $FinalPart[FP^*] \leftarrow FinalPart[FP^*] \cup PP$
        $qualSolution \leftarrow qualSolution + Quality^*$
    if ($qualSolution < bestQuality$) then
        $bestQuality \leftarrow qualSolution$
        $bestPartition \leftarrow FinalPart$
        $bestBind \leftarrow bind$
Return($bestPartition$, $bestBind$, $bestQuality$))
End
```

5.2.2 Greedy algorithm The first part of the Tabu Search TS algorithm is the construction of a feasible initial solution in order to find better solutions by stepwise transformations. The simplest way to do this, is to generate a random solution by randomly assigning activities to partitions and services to activities. However, the obtained results proved to be not sufficient. In this sense, many recent researches in TS deals with various techniques for making the search more effective. These include methods for creating better starting points called elite solutions. For this purpose, we adopt Greedy algorithm to generate a good initial solution. Greedy algorithms are intuitive heuristics in which greedy choices are made to achieve a certain goal [25]. Greedy heuristics are constructive heuristics since they construct feasible solutions for optimization problems from scratch by making the most favorable choice in each step of construction. By adding an element to the (partial) solution which promises to deliver the highest gain, the heuristic acts as a greedy constructor.

Algorithm 5 presents a method that computes a good feasible solution to activity placement and service selection. It takes as input *pre-partitions*, unconstrained activities and service candidates of each activity. Then, according to a fixed final partitions number, try to place at each step an activity (or *pre-partition*) to a final partition, and assign a service (or a set of service) to it. Both assignment and placement are based on cost estimation. The cost of assigning an activity to a service among its candidate services depends of the latter quality. Then the cost of placing an activity in each final partition depends of the communication overhead as well as the average distance between the activity to place and all activities of the partition. The most favorable choice among final partitions costs is selected. For pre-partitions placement, the same procedure is used except the fact that we take into consideration the constraints, and a global cost of assigning it to a final partition since it includes a set of activities. Once all activities and pre-partitions are assigned, we compute the global cost, and then change final partitions number and iterate. After each iteration we compare the quality of the current solution to the previous one and save the best. The output of the algorithm is an optimized feasible solution.

To analyze the complexity of Algorithm 5 , we first analyze the complexity of one iteration of the outer loop. In one such iteration, we consider every possible binding of an activity (that has not yet been bound) to a service. If we write *MaxCand* to denote the maximum number of candidate services that any activity has, we have to consider *MaxCand* possible bindings per activity and thus at most $MaxCand \times |Act|$ bindings in total. Each such binding is then compared against all activities that have already been bound in order to compute the distances (again, there are at most $|Act|$ such bound activities). We also have to evaluate the QoS of each service binding, but we assume this is a constant-time operation. Thus, the complexity of one iteration of the outer loop is $O(MaxCand \times |Act|^2)$. Also, during each iteration of the outer loop, we have to test NP times whether or not two partitions are linked through any *Separate* constraint. Each such test takes at most $|A|^2$ operations. Next, we note that the outer loop is executed $NP_{max} - NP_{min}$ times, with NP ranging between these two values. Thus overall, the complexity is $O((NP_{max} - NP_{min}) \times MaxCand \times |Act|^2 + (NP_{max} - NP_{min})^2 \times |A|^2)$. Thus we can say that the complexity of the algorithm is a polynomial of order four, but one of the variables in this polynomial is $NP_{max} - NP_{min}$, which can be made smaller if needed since we do not need to consider all possible numbers of partitions.

5.2.3 Tabu search algorithm In the following we will describe a solution that combine the greedy algorithm to the Tabu search algorithm in order to optimize the previously presented solution. As we mentioned before, the key idea is to start the Tabu search with an initial good solution. For this purpose we use the greedy solution. Then, for each iteration, possible moves will be calculated and the move leading to the highest benefit will be performed. If the highest benefit is negative, the move will be performed anyway, unless this move is forbidden by the tabu list. In order to guide the moves, we utilize some heuristics that can be employed (in conjunction with the tabu search algorithm) to improve the solution. The heuristics are described as follows:

- Put together activities which exchange lot of data to reduce inter-partitions interactions.
- Put together activities whose invoked services are geographically close.

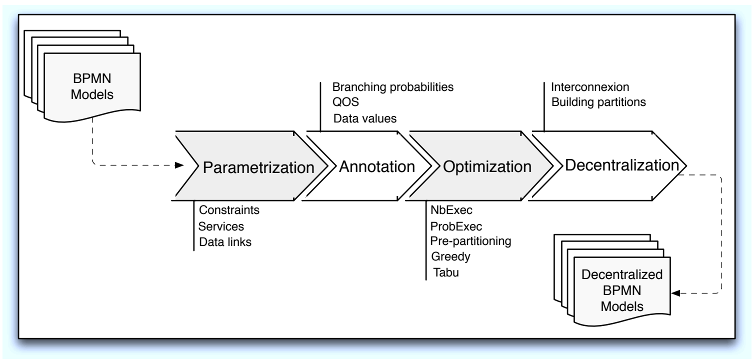


Fig. 6 Implementation architecture

Algorithm 6 presents a pseudo code for the tabu search where stop condition represents:

- after a fixed number of iterations.
- after number of iterations without an improvement in the objective function value.
- when the objective reaches a pre-specified threshold value.

The function *quality* is evaluated as described in equation 4. A move is described by an activity assignment to another partition or service with respect to the constraints. At each step, the quality function compute the cost of a set of possible moves, choses the best move and evaluates the quality of the new partitioning. The object function is the minimization of the communication cost and the maximization of the overall QoS. We also use an aspiration criteria to determine if a tabu move can be accepted or not. Indeed, some of the solutions that must be avoided (tabu list) could be of excellent quality and might not have been visited. To mitigate this problem, the aspiration criteria allow to override a solution's tabu state, thereby including the otherwise-excluded solution in the allowed set.

Algorithme 6 : Tabu search

```
Require: - $S_g$: greedy solution($w_q$, $w_{in}$, $w_{out}$)
Init: $S_0 \leftarrow s_g$
$S \leftarrow S_0$: current solution
$S^* \leftarrow S_0$: the best-known solution
$f^* \leftarrow quality(S_0)$
$T \leftarrow \{\}$: Tabu list
begin
    while ($\neg$ StopCondidtion()) do
        $$S \leftarrow \arg\min_{S' \in Na(S)} [quality(S)]$$
        if $quality(S) < f^*$ then
            $f^* \leftarrow quality(S)$
            $S^* \leftarrow S$
            record tabu for the current move in T (delete oldest entry if necessary)
end
return $S^*$
```

6 Implementation and Experimental Evaluation

6.1 Overview

We have implemented the proposed techniques as a prototype and used this prototype to empirically study the performance and effectiveness of our approach in different cases by a series of experiments. The experiments were conducted using abstract process models (BPMN) defined in XML format. These models are available on *Oryx* [2] and *IBM research* web site [3]. We also made use of the open source *BPstruct* [4] mainly developed to transform non structured process models into structured ones. Particularly, we used the packages for transforming the XML process models to graphs and for loops identification. The implementation architecture is described in figure 6. Since the BPMN models we used include only control flow, the first step consists in completing the models by a data flow structure. In this step, we also assign to each activity a set of candidate services and we define random collocate and separate constraints with respect to the consistency property. In the second step, we assign values to the services (QoS), to branching patterns (branching probabilities), to data (size) and to identified loops (loop exit and entry probabilities). We also assign to each service a random GPS position in order to compute the distances between them. A partition position is computed according to the positions of the the services assigned to it (e.g. barycenter). The third step is related to the optimization methods including the Greedy and Tabu algorithms. The output of this step is a set of partitions and a binding. The final step concerns the decentralization and deployment process which is out of scope of this paper.

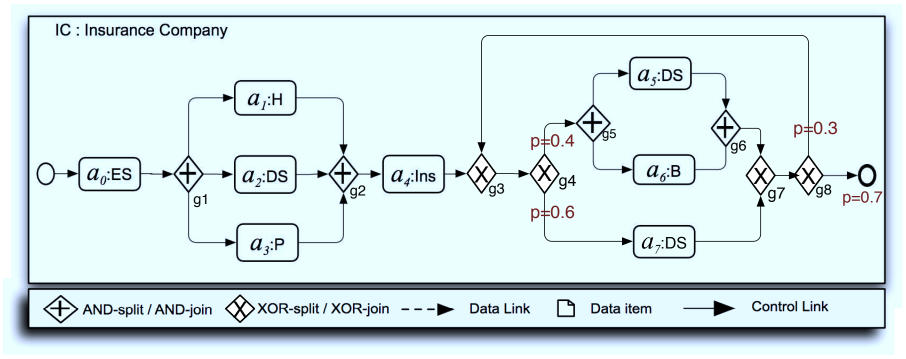


Fig. 7 Extended insurance process example

Next, we use an extended version of the insurance process example 7. Specifically, we replaced the OR construct by in association of XOR and AND (which is semantically equivalent), and we added a *loop* on the delivery service and the bank. We also annotated the model with the transition probabilities. The objective of these modifications is to better explain some implementation details.

The table 1 resumes the execution number of each activity of the insurance example ($NumExec$). For instance, the execution number of the activity $a_7 : DS$ is equal to $0.6 \times 1/(1-0.3)$ (where 0.3 is the probability to stay in the loop).

	$\mathbf{a_0}$: **ES**	$\mathbf{a_1}$: **H**	$\mathbf{a_2}$: **DS**	$\mathbf{a_3}$: **P**	$\mathbf{a_4}$: **Ins**	$\mathbf{a_5}$: **DS**	$\mathbf{a_6}$: **B**	$\mathbf{a_7}$: **DS**
NbExec	1	1	1	1	1	0.57	0.57	0.85

Table 1 Execution number of the insurance process activities

[2] http://oryx-project.org/backend/poem/repository
[3] http://www.zurich.ibm.com/csc/bit/downloads.html
[4] http://code.google.com/p/bpstruct/

The table 2 resumes for each activity the probability that it follows immediately another activity (ProbFollows). We note that $a_5 : SL$ may be executed immediately after its first execution since it is in a loop ($0.3 \times 0.4 = 0.12$). The probFollows of to non-consecutive activities is equal to zero.

	$\mathbf{a_0 : ES}$	$\mathbf{a_1 : H}$	$\mathbf{a_2 : DS}$	$\mathbf{a_3 : P}$	$\mathbf{a_4 : Ins}$	$\mathbf{a_5 : DS}$	$\mathbf{a_6 : B}$	$\mathbf{a_7 : DS}$
$\mathbf{a_0 : ES}$	0	1	1	1	0	0	0	0
$\mathbf{a_1 : H}$	0	0	0	0	1	0	0	0
$\mathbf{a_2 : DS}$	0	0	0	0	1	0	0	0
$\mathbf{a_3 : P}$	0	0	0	0	1	0	0	0
$\mathbf{a_4 : Ins}$	0	0	0	0	0	0.4	0.4	0.6
$\mathbf{a_5 : DS}$	0	0	0	0	0	0.12	0.12	0.18
$\mathbf{a_6 : B}$	0	0	0	0	0	0.12	0.12	0.18
$\mathbf{a_7 : DS}$	0	0	0	0	0	0.12	0.12	0.18

Table 2 ProbFollows values of the insurance process example

6.2 DJ Graphs

In a process model, data links should respect the control flow constraints to avoid some use cases (e.g. an activity waiting for a data that would never come or deadlocks). For instance, in the insurance process example we can not add a data link between the hospital H and the police P since they are in parallel. For this purpose, and in order to add random and correct data links to the tested models, we use The DJ-graphs [34]. The latter is used to check if the add is allowed or not. Moreover, DJ-Graphs help identifying each loop in a process model, its entry, its exit and all activities it encapsulates. This is useful to compute the communication cost and the partitioning quality. We briefly introduce DJ-Graphs below.

In a flow-graph with a single start (source) node, a node n_1 is a dominator of a node n_2 if every path from the start node to n_2 goes through node n_1. It is well-know [30] that this dominance relation is a tree, where the source node is the root and every node in this tree dominates its children. This tree is called the *dominator tree.* Dominator trees allow us to identify some types of loops (single-entry loops) since the entry node dominates the others, but it does not provide a sufficient level of granularity to identify other loops. The DJ Graph lifts this limitation by combining the dominator tree with the original flow-graph. The DJ graph of a flow-graph consists of the same set of nodes as in the flow-graph and two types of edges, namely D edges and J edges. D edges are the dominator tree edges. A "J" edge represents either a "cross-edge" which intuitively means an edge going from one branch of the graph to another, or a "back-edge", which intuitively is an edge that makes the flow of control go back to an "earlier" edge.

An example is depicted in Figure 8 and refers to the DJ-graph of the extended insurance example. The DJ-graph consists of the edges of the dominator tree (dashed), backedges, and the remaining edges of the control flow called cross edges (solid). Loops in the DJ-Graph can then be found starting from the lowest dominator level. If a backedge exists at the current level, then nodes corresponding to its "natural loop" are collapsed into one node. In this example, $a_0 : SU$ dominates $a_4 : INS$ but $a_6 : B$ does not dominate $a_7 : SL$. This means that we can add a data form a_0 to a_4 but not from a_6 to a_7 (this is logical since they are in parallel). It should be noted that a_7 is not dominated by g_7 since they are in a loop and then there is a case where g_7 could be executed before a_7 (execute g_5, g_7 and then a_7). Therefore if we consider g_7 as an activity we can not add a data item from g_7 to a_7.

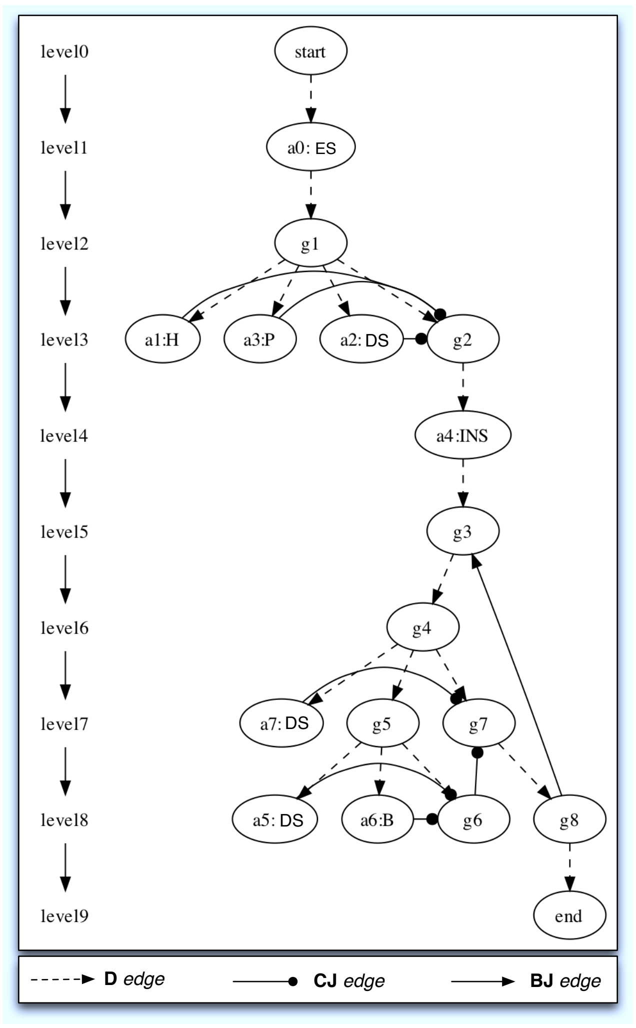


Fig. 8 DJ Graph of running example

6.3 Results and analysis

Figure 9 represents a comparison between the results obtained by Greedy search and those obtained by the Tabu algorithm. The tested were conducted on 105 models with various sizes (between 12 and 88 nodes with an average size of 24 nodes). The y-axis represents the quality of the corresponding process partitioning. This includes intra and inter-partitions communication, and the QoS of the overall split process. The x-axis represents the tested models. In the table 3, we show only a subset of the tested models. For each process model, we apply the tests many times and we compute the average of the corresponding results. We remind that we look to minimize the cost. Experiments shows that in most cases, Tabu search improves the results obtained by the greedy algorithm. This is due to the fact that Tabu algorithm explores more search space and therefore has more probability to to get a better solution. In addition, by accepting some solutions which are worst than the current one, the Tabu algorithm may jump to another local optimum and in some cases to the global optimum. It should be noted that, in many cases, the Tabu algorithm gives similar results as the Greedy search (e.g. models M7 and M9).

The figure 10 describes an evaluation of the communication cost with respect to the number of partitions in the case where the weights of QoS, intra and inter-partitions communication are equal ($w_q = w_{in} = w_{out}$). We remark that the communication cost of the split process increases when the number of partitions is very high. Indeed, when the number of partitions increases, the communication

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11
Greedy	0,195	0,937	0,370	0,798	0,661	0,318	0,373	0,795	0,499	0,587	0,469
Tabu	0,160	0,167	0,343	0,133	0,554	0,184	0,373	0,103	0,472	0,475	0,112

Table 3 Comparaison between Greedy and Tabu algorithms: communication costs

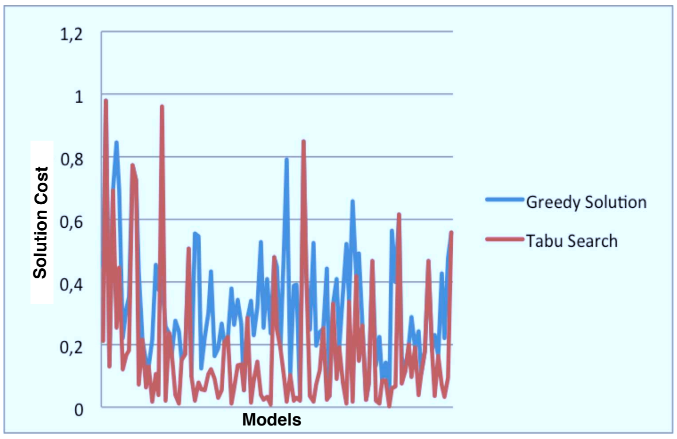


Fig. 9 Tabu search vs Greedy search: solution quality

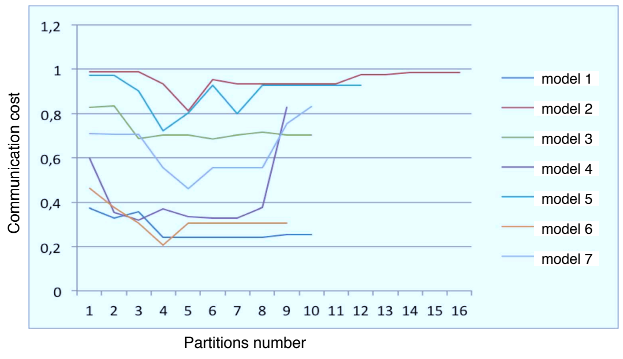


Fig. 10 Results: communication cost versus the partitions number

inter-partitions increases and becomes more significant compared to the QoS. Since $w_q = w_{out}$, the overall cost increases.

Similarly, when the number of partitions is small, the number of activities by partition increases as well as the number of services assigned to it. This reduces the flexibility in terms of the partition position according to the services it invokes, since we look to put the partition near the services it controls.In our experiments, we assume that each partition should be put in the barycenter of the services locations it is assigned to, with respect to the execution number of each service. Thereby, the more the number of services assigned to the same partition and which are geographically dispersed increases, the more the probability of putting the partition far from these services increases. This affect directly the intra-partition communication cost and consequently the overall cost.

Our experimental setup for testing optimized orchestration is achieved using an Intel processor Core 2 Duo, 2.5 GHz, a 4 GB memory and Eclipse Galileo environment. Figure 11 presents an evaluation of the execution time of the tabu search algorithm according to the models sizes. The size of the model corresponds to the number of nodes (tasks + gateways) it includes. The y-axis is presented in a logarithmic scale and represents the execution time (ms). The x-axis represents the models sizes. As can seen in the figure, the time grows exponentially with respect to the size of the model. The average size is 24 nodes

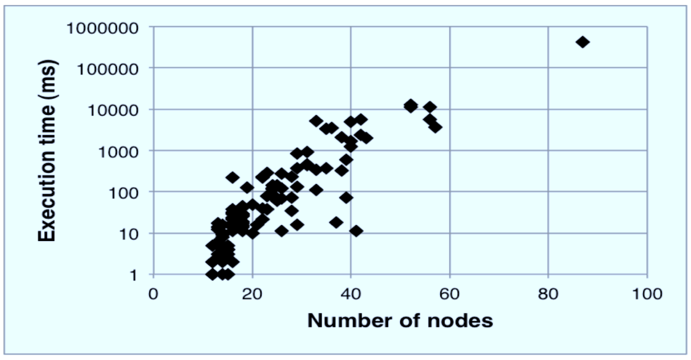


Fig. 11 Computation time of our Tabu algorithm for different size of process models

and the average execution time is $4.24s$. In worst case, the execution time is 7 minutes for 88 nodes. This does not affect the quality of the optimization solution since it is executed once in design time.

6.4 Tool support

The figure 12 presents a screenshot of the optimization tool. The user should specify the number of separation and colocation constraints, the weights for the QoS, the intra and the inter-partitions communication, and the data links to add between process activities. Then, to execute the optimization methods, the user should specify the tabu list size and the iteration number. The results presented in the figure concerns the optimized partitioning of the insurance example. For each model it generates the partitioning, the binding, the best quality and partitions positions (GPS positions) for both the tabu and the greedy algorithms. With respect to this partitioning, the final decentralized partitions are depicted in figure 13. The result is three partitions communicating through messages exchange. In the figure we represent only control messages. This model is generated manually.

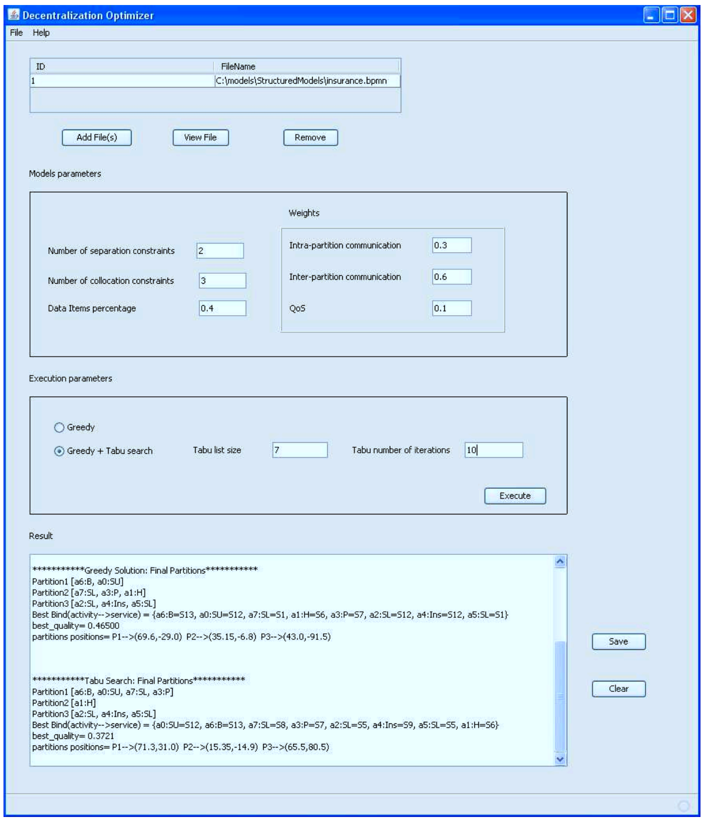
In future work, we further to complete the graphical support of the tool and give more control to the user to allow direct specification of the collocation and separation constraints as well as the models.

7 Related Work

In recent years, several methods and systems for decentralized business process execution have been proposed. One of the earliest work in the area is the Mentor project [36]. In Mentor, workflows are modeled using state-charts that are partitioned so that each partition is delegated to a separate *processing entitiy* (PE). Each PE-specific state-chart is executed locally on the PE workstation. Their approach takes into account both control and data-flow dependencies. Sadiq et al. [31] present another method for decentralized workflow execution based on partitions, but without considering data dependencies. Meanwhile, Yildiz et al [38] consider the decentralization of processes from an abstract perspective by extending the dead path elimination algorithm used in BPEL process execution engines. Their contribution focuses on preserving the control-flow constraints in the centralized specification, while preventing deadlocks when services interact with one another.

The above approaches do not consider communication overhead when splitting the process into partitions. Instead, they assume that the split is given by the designer or inferred from the roles specified in the process model. Importantly, our partitioning approach could be used on top of any of the above decentralized orchestration approaches. Thus, our work is complementary to the above ones.

Nanda et al. [11] present an approach to partition BPEL processes using program partitioning techniques with the aim of reducing the communication costs between the partitions. More recently, Khalaf et al [23][22] presented a method for decentralized orchestration of BPEL processes, focusing on the derivation of P2P interactions. Both proposals do not take into account distribution constraints (Collocate and



ID	FileName
1	C:\models\StructuredModels\insurance.bpmn

Models parameters

Number of separation constraints: 2
Number of collocation constraints: 3
Data Items percentage: 0.4

Weights
Intra-partition communication: 0.3
Inter-partition communication: 0.6
QoS: 0.1

Execution parameters

Greedy [✓ Greedy + Tabu search]
Tabu list size: 7
Tabu number of iterations: 10

Result

```
***********Greedy Solution: Final Partitions***********
Partition1 [a6:B, a0:SU]
Partition2 [a7:SL, a3:P, a1:H]
Partition3 [a2:SL, a4:Ins, a5:SL]
Best Bind(activity-->service) = {a6:B=S13, a0:SU=S12, a7:SL=S1, a1:H=S6, a3:P=S7, a2:SL=S12, a4:Ins=S12, a5:SL=S1}
best_quality= 0.46500
partitions positions= P1-->(69.6,-29.0)  P2-->(35.15,-6.8)  P3-->(43.0,-91.5)


***********Tabu Search: Final Partitions***********
Partition1 [a6:B, a0:SU, a7:SL, a3:P]
Partition2 [a1:H]
Partition3 [a2:SL, a4:Ins, a5:SL]
Best Bind(activity-->service) = {a0:SU=S12, a6:B=S13, a7:SL=S8, a3:P=S7, a2:SL=S5, a4:Ins=S9, a5:SL=S5, a1:H=S6}
best_quality= 0.3721
partitions positions= P1-->(71.3,31.0)  P2-->(15.35,-14.9)  P3-->(65.5,80.5)
```

Fig. 12 Prototype screenshot

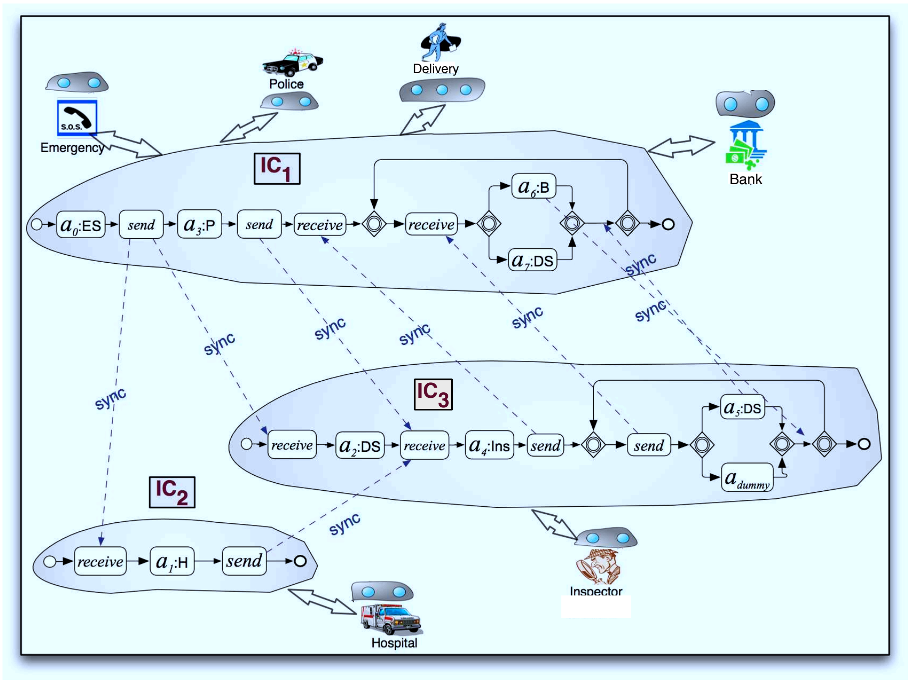


Fig. 13 Decentralized Insurance process after Tabu search

Separate) and therefore the designer cannot control the partitioning. Also, they do not take into account the possibility of an activity having multiple candidate services, each with a different location and a different QoS. The common limitation of the current decentralization approaches is their dependency on the underlying process specification (i.e. BPEL).

Recently, Lifeng and al. [2] proposed an extension to the BPEL partitioning approach introduced in [11]. The extension represents a penalty-based genetic approach to optimize the partitioning process. First, they create an initial partitioning topology, then they generate new topologies by stepwise transformations using genetic operators (i.e. selection, crossover and mutation) and evaluate the fitness of each topology. The procedure is repeated until a fixed maximal generation number is reached. The mentioned approach is similar to our proposal, but it deals only with BPEL processes and do not consider collocation and separation constraints.

Safi Esfahani et al. [32] present a method for adaptable decentralization of BPEL processes. The process of decentralization may be configured so that the produced fragments become adaptable with different aspects of runtime environment (dynamic criteria). The decentralization is achieved in two steps: (i) at design time, where the initial centralized process is split into a set of *fragments* according to the designer criteria and, (ii) at run-time, where each of the derived fragment is in turn split into *adaptable fragments* according to runtime requirements. In sharp contrast to our approach, this work addresses only BPEL processes. Moreover, each derived partition is a strong connected component of the initial graph. In our work, nodes composing each derived partition may not have direct connection in the original graph. Besides, their approach do not consider collocation and separation constraints as well as communication overhead.

Hildebrandt et al. [17] present an approach to decompose a declarative *global* process model described in Dynamic Condition Respone Graphs (DCR Graphs) into declarative *local* sub-processes. They use a projection technique to project DCR Graphs according to a subset of labels and events. This projection do not take into consideration neither the QoS nor the optimization of communication overhead between the derived partitions. Besides, the authors do not consider collocation and separation constraints which are relevant for privacy or security concerns. Also they do not mention how they deal with repeated blocs of activities (*loops*).

Other approaches to decentralized orchestration do not require any partitioning. For instance, the Self-Serv system [6][5] is able to execute web service compositions in an entirely peer-to-peer fashion: services send messages to one another after completing each activity in the orchestration. This approach is equivalent to assigning each activity (service) to a separate partition (as illustrated in Figure 2b). Another method for decentralized execution without partitioning is presented in [27][28]. The authors developed a formal approach that takes as input the existing services, the goal service and the costs, and produces a set of decentralized choreographers that optimally realize the goal service using the existing services. However, the authors do not explain how they deal with Repeat blocks (i.e. loops), which have a significant impact on communication overhead.

The graph partitioning problem is ubiquitous in many fields of computer science and engineering [7, 33, 35, 16]. It has important applications in areas ranging from work-load balancing in parallel programming, to database storage. The graph partitioning problem is NP- complete. Therefore many heuristic methods are proposed to find high quality partitions. Graph partitioning algorithms aim at identifying partitions of a graph such that the weights of the edges inside a partition are high, while the weight of the edges across partitions are low. In other words, connections inside a partition are strong, while connections across partitions are weak. This is akin to our goal of partitioning the activities in a composite service and assigning activities to services in such a way as to minimize inter-partition communication. However, mainstream graph partitioning approaches are not designed to take into account co-location and separation constraints, which we argue are relevant in the context of cloud deployments. Additionally, in the context of decentralized service orchestration, graph partitioning methods are applicable once the activities in the composite service are bound to concrete services, while part of the problem we address in this paper is precisely that of assigning services to activities in such a way as to optimize not only communication cost, but also overall QoS. This combination of factors make graph partitioning techniques not directly applicable to the problem at hand.

Several previous studies have addressed the problem of aggregating QoS of composite services based on orchestration models. Jaeger *et al.* [20, 21] discuss the QoS aggregation problem for orchestration models consisting of sequence, choice and parallel flow blocks. This approach does not deal with loops. This restriction is lifted by Cardoso *et al.* [10] who proposed a Stochastic Workflow Reduction (SWR) algorithm that takes as input a process graph and computes the expected QoS by repeatedly applying a set of reduction rules for well-structured sequential, parallel, choice and repeat loops. In a similar vein, Hwang *et al.* [18, 19] represent composite services using a tree structure and compute the aggregate QoS of composite services by traversing the tree using breadth-first search. Canfora *et al.* [9] propose the same set of QoS aggregation functions and use them to tackle the problem of binding and re-binding component services to an orchestration model in order to maximize the QoS of the final binding. One common limitation in the works above is that all of them require that the orchestration model is well structured. Such limitation is partially lift in a previous work reported in [37]. Also related to the problem of QoS-aware composite service decentralization is that of QoS-aware service composition [3, 24, 39]. The goal is to find a binding that optimizes a given objective function while satisfying a given set of constraints. The input is an orchestration model and a set of service candidates for each task in the orchestration model. Zeng *et al.* [39] study a local and a global optimization approach to this problem using Simple Additive Weighting (SAW) and Integer Programming (IP), respectively. Meanwhile, Liu *et al.* [24] propose a dynamic QoS computation model for web services selection in order to deal with runtime QoS selection. The authors construct a QoS matrix and compute QoS of a composite service via normalization and then multiplication with weights given by a user. A combination of local optimization and global optimization approaches is studied in Alrifai *et al.* [3]. This latter work considers three types of QoS aggregation functions: summation, multiplication and minimum relation. Both QoS aggregation and QoS aware service composition problems are orthogonal to the problem of service decentralization. In this paper, we have explicitly described the integration of additive QoS attributes. The extension of the method to multiplicative and critical path QoS attributes is straightforward.

This paper is an extended and revised version of a previous conference paper [13]. With respect to the conference version, the extensions include the normalization method to compute the communication cost (which is needed in order to combine them with other costs), the detailed definition of a Tabu heuristics, as well as the empirical evaluation of both the Greedy and the Tabu heuristics.

8 Conclusion

This paper presented a method for optimized constrained decentralization of composite web services. The method seeks to compute a partitioning of activities and a binding of activities to services in such a way as to minimize communication costs while maximizing QoS. In doing so, the method takes into account the expected communication volume between partitions, the distance between partitions and the distance between the services assigned to activities in the same partition. Because of the nature of the objective function, the underlying optimization problem is formulated as a Quadratic Assignment Problem (QAP). A greedy heuristic is used in order to construct an initial solution, while a Tabu search is employed to improve over this initial solution. The experimental evaluation reported in this paper shows that the Tabu search consistently improves over the initial solution produced by the Greedy algorithm, and that the combined heuristic can deal with models of realistic size.

Although previous work have studied the problem of partitioning service orchestrations, the method proposed in this paper is arguably richer insofar as it takes into account the need to optimize QoS during the assignment of services to activities, and it is able to handle separation and colocation constraints.

A possible direction for future work is to apply other meta-heuristics to the optimization problem formulated in this paper, such as simulated annealing or genetic algorithms. A genetic algorithm with a suitable crossover operator to smartly combine partitions from different solutions is a particularly appealing approach to achieve higher-quality solutions.

Acknowledgments This work was conducted while the second author was Visiting Professor at LORIA – INRIA Nancy. The second and fourth authors were also supported by the ERDF through the Estonian Centre of Excellence in Computer Science.

References

1. E. Aarts and J. Lenstra, editors. *Local Search in Combinatorial Optimization.* Discrete Mathematics and Optimization. Wiley, Chichester, UK, June 1997.
2. L. Ai, M. Tang, and C. J. Fidge. Partitioning composite web services for decentralized execution using a genetic algorithm. *Future Generation Comp. Syst.*, 27(2):157–172, 2011.
3. M. Alrifai and T. Risse. Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition. In *Proc. 18th Int. Conf. on World Wide Web*, pages 881–890. ACM, 2009.
4. M. Beckman and T. Koopmans. Assignment problems and the location of economic activities. *Econometrica*, 25:53–76, 1957.
5. B. Benatallah, M. Dumas, and Q. Z. Sheng. Facilitating the rapid development and scalable orchestration of composite web services. In *Distributed and Parallel Databases*, 2005.
6. B. Benatallah, Q. Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, 2003.
7. S. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. *Computers, IEEE Transactions on*, 37(1):48 –57, jan 1988.
8. R. E. Burkard, E. Çela, G. Rote, and G. J. Woeginger. The quadratic assignment problem with a monotone anti-monge and a symmetric toeplitz matrix: Easy and hard cases. In *IPCO*, pages 204–218, 1996.
9. G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. A framework for QoS-aware binding and re-binding of composite web services. *Systems and Software*, 81(10), 2008.
10. J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Quality of Service for Workflows and Web Service Processes. *Web Semantics*, 1(3):281–308, 2004.
11. G. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized orchestration of composite web services. In *WWW (Alternate Track Papers & Posters)*, pages 134–143, 2004.
12. M. Dumas, L. García-Bañuelos, A. Polyvyanyy, Y. Yang, and L. Zhang. Aggregate quality of service computation for composite services. In *Proc. of the 8th International Conference on Service-Oriented Computing (ICSOC), San Francisco, CA, USA*, pages 213–227, December 2010.
13. W. Fdhila and C. Godart. Toward synchronization between decentralized orchestrations of composite web services. In *Proc. of the 5th International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom 2009, Washington DC, USA*, pages 1–10. IEEE, November 2009.
14. W. Fdhila, U. Yildiz, and C. Godart. A flexible approach for automatic process decentralization using dependency tables. In *ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services*, pages 847–855, Los Angeles, CA, USA, 2009. IEEE Computer Society.
15. F. Glover and M. Laguna. Tabu search, 1997.
16. B. Hendrickson. Load balancing fictions, falsehoods and fallacies. *Applied Mathematical Modelling*, 25(2):99–108, 2000.
17. T. Hildebrandt, R. R. Mukkamala, and T. Slaats. Safe distribution of declarative processes. In *Proceedings of the 9th international conference on Software engineering and formal methods*, SEFM'11, pages 237–252, Berlin, Heidelberg, 2011. Springer-Verlag.
18. S.-Y. Hwang, H. Wang, J. Srivastava, and R. A. Paul. A Probabilistic QoS Model and Computation Framework for Web Services-Based Workflows. In *Proc. 23rd Int. Conf. on Conceptual Modeling*, pages 596–609. Springer, 2004.
19. S.-Y. Hwang, H. Wang, J. Tang, and J. Srivastava. A probabilistic approach to modeling and estimating the QoS of web-services-based workflows. *Information Sciences*, 177(23):5484–5503, 2007.
20. M. C. Jaeger, G. Rojec-Goldmann, and G. Muhl. QoS Aggregation for Web Service Composition using Workflow Patterns. In *Proc. of the Int. Conf. on Enterprise Distributed Object Computing (EDOC)*, pages 149–159. IEEE, 2004.
21. M. C. Jaeger, G. Rojec-Goldmann, and G. Muhl. QoS Aggregation in Web Service Compositions. In *Proc. of the IEEE Conf. on E-Commece, E-Services and E-Government (EEE)*, pages 181–185, 2005.
22. R. Khalaf, O. Kopp, and F. Leymann. Maintaining data dependencies across bpel process fragments. *Int. J. Cooperative Inf. Syst.*, 17(3):259–282, 2008.

23. R. Khalaf and F. Leymann. E role-based decomposition of business processes using bpel. In *ICWS*, pages 770–780, 2006.
24. Y. Liu, A. H. Ngu, and L. Z. Zeng. QoS Computation and Policing in Dynamic Web Service Selection. In *WWW Alt.*, pages 66–73, 2004.
25. P. Merz and B. Freisleben. Greedy and local search heuristics for unconstrained binary quadratic programming. *J. Heuristics*, 8(2):197–213, 2002.
26. R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.
27. S. Mitra, R. Kumar, and S. Basu. Optimum decentralized choreography for web services composition. In *IEEE SCC (2)*, pages 395–402, 2008.
28. S. Mitra, R. Kumar, and S. Basu. A framework for optimal decentralized service-choreography. *Web Services, IEEE International Conference on*, 0:493–500, 2009.
29. A. Polyvyanyy, L. García-Bañuelos, and M. Dumas. Structuring acyclic process models. *Information Systems (IS)*, 37(6):518–538, 2012.
30. G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.*, 24(5):455–490, 2002.
31. W. Sadiq, S. W. Sadiq, and K. Schulz. Model driven distribution of collaborative business processes. In *IEEE SCC*, pages 281–284, 2006.
32. F. Safi Esfahani, M. A. Azmi Murad, M. N. B. Sulaiman, and N. I. Udzir. Adaptable decentralized service oriented architecture. *J. Syst. Softw.*, 84(10):1591–1617, Oct. 2011.
33. V. Sarkar. *Partitioning and scheduling parallel programs for multiprocessors.* Research monographs in parallel and distributed computing. Pitman, 1989.
34. V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using dj graphs. *ACM Trans. Program. Lang. Syst.*, 18(6):649–658, 1996.
35. H. Trandac and V. Duong. A constraint-programming formulation for dynamic airspace sectorization. In *Digital Avionics Systems Conference, 2002. Proceedings. The 21st*, volume 1, pages 1C5–1 – 1C5–11 vol.1, 2002.
36. D. Wodtke, J. Weißenfels, G. Weikum, and A. K. Dittrich. The mentor project: Steps toward enterprise-wide workflow management. In *ICDE*, pages 556–565, 1996.
37. Y. Yang, M. Dumas, L. García-Bañuelos, A. Polyvyanyy, and L. Zhang. Generalized aggregate quality of service computation for composite services. *Journal of Systems and Software*, 2012.
38. U. Yildiz and C. Godart. Information flow control with decentralized service compositions. In *ICWS*, pages 9–17, 2007.
39. L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.