Optimized Decentralization of Composite Web Services

Walid Fdhila
LORIA - INRIA Nancy - Grand Est
F-54500 Vandœuvre-lès-Nancy, France
Email: fdhilawa@loria.fr

Marlon Dumas
University of Tartu
Tartu 50409, Estonia
Email: marlon.dumas@ut.ee

Claude Godart
LORIA - INRIA Nancy - Grand Est
F-54500 Vandœuvre-lès-Nancy, France
Email: godart@loria.fr

Abstract—
Composite services are usually specified by means of orchestration models that capture control and data-flow relations between activities. Concrete services are then assigned to each activity based on various criteria. In mainstream service orchestration platforms, the orchestration 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 service orchestrations in a decentralized manner while fulfilling collocation and separation constraints. However, this and similar methods for decentralized orchestration do not seek to optimize the communication overhead between services participating in the orchestration. This paper presents a method for optimizing the selection of services assigned to activities in a service orchestration in terms of QoS properties and communication overhead. The method takes into account the communication cost between pairs of services, the amount of data that these services need to exchange in the orchestration, and the collocation and separation constraints imposed by the service providers.

I. Introduction

Service-Oriented Architecture (SOA) is a proven collection of principles for structuring large-scale systems in order to improve manageability and to streamline change. One of the pillars of SOA is the ability to rapidly compose multiple services into an added-value business process, and then to expose the resulting business process as a *composite service* [3]. Composite services are generally captured by means of an orchestration model: a process model in which each activity represents either an intermediate step (e.g. a data transformation) or an interaction with one of the services participating in the composition (the *component services*). The process model specifies the control-flow and data-flow relations between activities, using a specialized language such as the Business Process Execution Language (WS-BPEL) or the Business Process Modeling 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 all its output data. The orchestrator then determines which 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 [3].

In previous work, we proposed a method for executing service orchestrations in a decentralized manner [8]. The idea is to group activities into partitions and to assign each partition to a separate orchestrator. 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 partition, or any other partitioning criterion of their choice. Clearly, the performance and robustness of a decentralized service orchestration would benefit from placing each orchestration engine as close as possible to the component services that it manages. But neither the above method nor other similar decentralized orchestration methods [11], [5], [18], [3] 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 communication overhead, while maximizing the QoS expressed in terms of combinations of properties such as time, cost, reliability, etc. The 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 interrelated optimization variables: QoS variables, location variables, collocation and separation constraints. To cope with this complexity, the proposal relies on heuristic optimization techniques [4]. Specifically, we present and analyze a greedy algorithm to build an initial solution, and we outline how Tabu search [9] can be applied to improve the initial solution. The crux of the heuristics is to place services that communicate frequently in the same partition, while fulfilling the collocation and separation constraints given by the designer.

COLLABORATECOM 2010, October 9-12, Chicago, USA

DOI 10.4108/icst.collaboratecom.2010.1

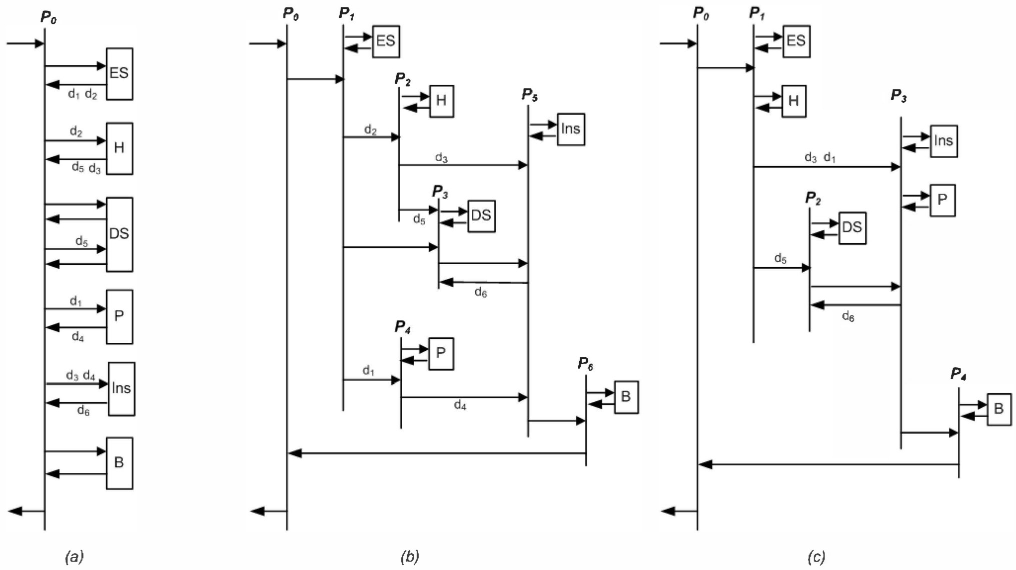

Fig. 1. (a) centralized model (b) First decentralized model (c) Second decentralized model

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. Section 3 describes the details of the proposed method. Section 4 discusses related work and Section 5 summarizes the contribution and outlines future directions.

II. MOTIVATING EXAMPLE

To motivate and illustrate the method presented in this paper, we make use of a sample orchestration taken from [19] (cf. Figure 2). This orchestration is designed to automate a claim handling process at an insurance company IC. The corresponding process model is captured in the BPMN notation, and it includes both control and data dependencies. Task 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.

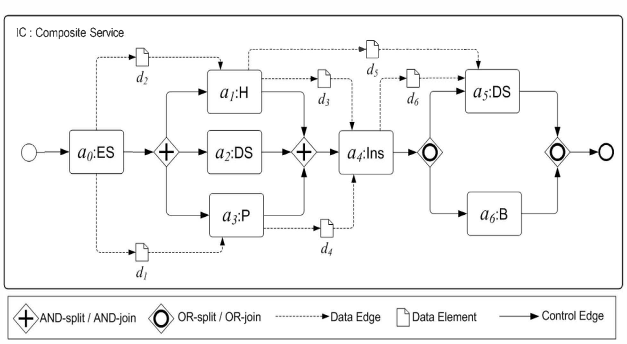

Fig. 2. Motivating example

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 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 2. 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 in the last part of the process: 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 1a. The centralized orchestrator is a bottleneck and may cause performance degradation and availability issues. It also causes additional traffic of messages, since every ac-

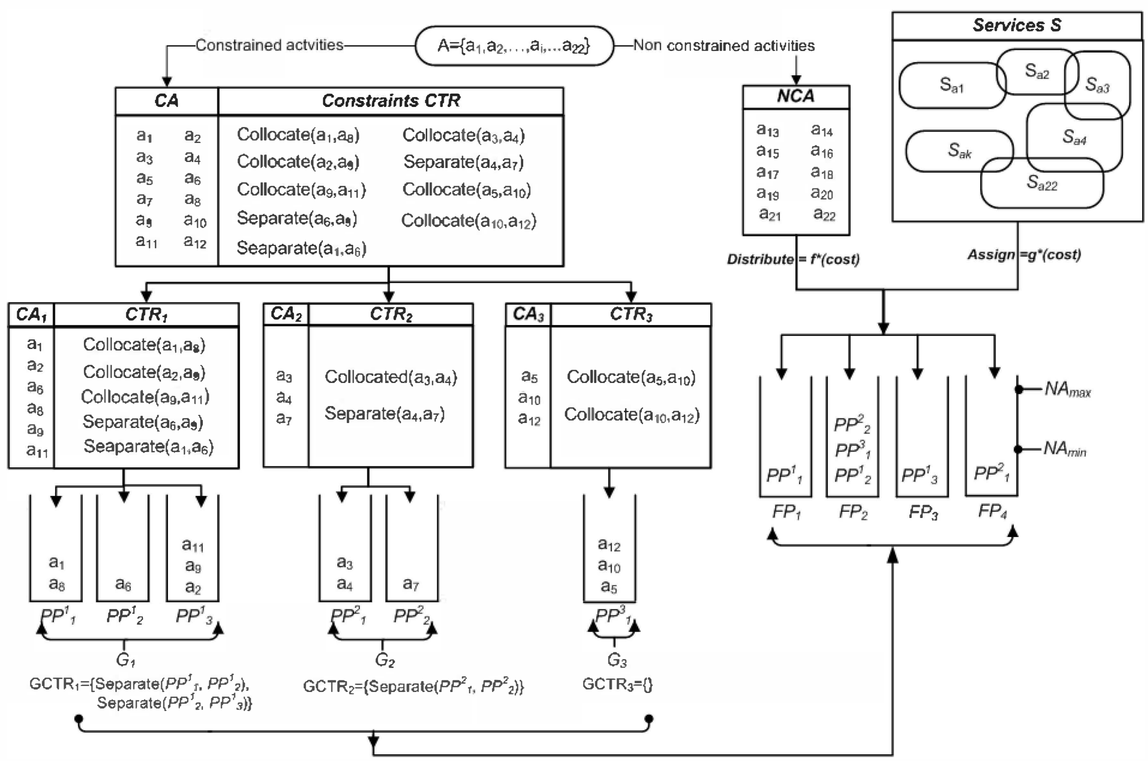

Fig. 3. Partitioning process

tivity 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 1b depicts a possible decentralized execution settings for the same process, where IC is partitioned into seven partitions that are executed by seven distributed orchestrators. Each partition P_i is responsible for all activities that are delegated to a given service. The time needed to exchange messages between a partition and its corresponding orchestrator is assumed to be negligible, since orchestrators are placed close to the services they manage. In this decentralized architecture, the data produced by a service are routed directly to the partitions of the services that consume these data. For example, *hospital* and *police protocols* (d_1 and d_2) generated by ES are routed directly to H and P. If we consider the data exchanged only between services, then the number of data flow messages in figure 1a is 8 (cf. communication links labelled with data items d_i). Meanwhile, in decentralized orchestration depicted in Figure 1b, 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. This paper takes into consideration all these facts in order to obtain optimized partitions for decentralized orchestration.

In addition to seeking to minimize communication overhead, the proposed method also take into account the QoS of each service. Specifically, we consider the case where there are multiple candidate services that can perform each 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.

III. 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 [8], [7] or other techniques discussed in Section IV.

In order to compute an optimized partitioning of an orchestration, we proceed in two steps. 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 III.B), 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.

Before describing the partitioning method, we define the notion of service orchestration and related notions (Section III.A). 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 III.B). We then show how the communication overhead between pairs of activities is computed by analyzing the orchestration models (Section III.C). Finally, using the pre-partitioning and the function for computing communication overhead, we show how the final partitioning is computed (Section III.D).

A. Inputs and Outputs

The method for optimized service selection takes as input a service orchestration consisting of activities related by control, data-flow and distribution constraints. 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 [16]. 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 2 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):

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

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

For example, the BPMN model in Figure 2 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 ($CTR = Collocate \cup Separate$). Also, we write $Act(Orc)$ to refer to the set of activities of an orchestration, $CA(Orc)$ to denote the set of constrained activities and $NCA(Orc)$ 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.

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

- A binding, that is, is a function that maps each activity in the orchestration model to a service;
- 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.

Specifically, 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. We do not impose a particular model for calculating the QoS of a 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 [20] in order to calculate the QoS of each component service, based on a weighted sum of the service's execution time, cost, reliability and availability.

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.

B. 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 3 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 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 3, 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 1. 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 3, 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.

Algorithm 1: Constrained activities partitioning

```
Require: - CTR: set of all constraints
Init: Groups ← {}
begin
    for each CTR_i in ConnectedComponent(CTR) do
        CurGroup ←{}
        for Collocate_i in
        ConnectedComponent(CTR_i ∩ Collocate) do
            NewPartition ← {a|∃a' Collocate_i(a, a')}
            CurGroup ← CurGroup ∪ {NewPartition}
        Groups ← Groups ∪ {CurGroup}
    Return Groups
end
Result: groups of constrained partitions
```

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 2 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 activities partitioning), NP the output of algorithm 2, 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 1 and 2, 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

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

that leads to the best overall tradeoff between communication overhead and QoS according to relative weights given by the user.

$$NP_{min} = \begin{cases} NP \quad if \quad \dfrac{|Act|}{NA_{max}} \leq NP \\ NP + \dfrac{|Act| - (NP * NA_{max})}{NA_{max}} \quad Otherwise \end{cases} \tag{1}$$

$$NP_{max} = \sum_k Size(G_k) + \frac{|NCA|}{NA_{min}} \tag{2}$$

Algorithm 2: 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}**))**

C. 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 2), 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.

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.

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 3 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.

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 4. In this algorithm, $prob(ca)$ denotes the probability associated to a conditional control-flow arc ca.

Having defined functions $numExec$ and $probFollows$ and

Algorithm 3: 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: $(\Pi_{cb \in condBranches} prob(cb) \times (\Pi_{rb \in repeatBlocks} 1/(1 - prob(rb)))$

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:

$$Cons(a_1, a_2) \times numExec(a_1) \times probFollows(a_2) + \Sigma_{(a_1,a_2,d) \in Data} numExec(a_1) * size(d) \quad (3)$$

...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.

Algorithm 4: 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: $\Pi_{ca \in condArcs} prob(ca)$

D. 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 [12] 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 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, which is equivalent to minimizing $1 - QoS_s$.

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

For the sake of conciseness, we hereby assume that all QoS attributes are additive, but the proposed method can be extended to attributes of type "multiplicative" and "critical path" [6]. The problem is quadratic 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 $d_{bind(i),bind(j)}$.

1) Heuristic optimization algorithms overview: Several exact algorithms have been used for solving the QAP problems, like branch and bound, cutting plane and branch and cut algorithms [4]. 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 [4]: Tabu search (TS), Simulated annealing (SA), Genetic algorithms (GA), Greedy randomized adaptive search procedures (GRASP), Ant systems (AS), etc. These methods are also known as local search algorithms. 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. This iterative step is repeated until no further improvement can be found. Improvement methods are local search algorithm which allow only improvements of the current solution in each iteration. 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 Tabu search algorithm to look for an optimal solution to our decentralization problem.

Tabu search [9] 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.

Algorithm 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: $\mathcal{P}_c \leftarrow P_c \cup \{\{a_i\} \backslash a_i \in NCA(Orc)\ \}$
$bestQuality \leftarrow +\infty$, $bestNumber \leftarrow NP_{min}$
$bestPartition \leftarrow \{\}$, $bestBind \leftarrow \{\}$
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**

$$s_{a_i} \leftarrow \underset{s_i \in Cand(a_i)}{\arg\min} \Big[w_q \cdot (1 - QoS(s_i)) + w_{out} \cdot \frac{\sum_{a_j \in FinalPart[FP]} co_{a_i,a_j} . d_{s_i,bind(a_j)}}{|FinalPart[FP]|} + w_{in} \cdot \frac{\sum_{a_j \in FinalPart[FP]} d_{s_i,bind(a_j)}}{|FinalPart[FP]|} \Big]$$

$$CurQual \leftarrow CurQual + \Big[w_q \cdot (1 - QoS(s_i)) + w_{out} \cdot \frac{\sum_{a_j \in FinalPart[FP]} co_{a_i,a_j} . d_{s_i,bind(a_j)}}{|FinalPart[FP]|} + w_{in} \cdot \frac{\sum_{a_j \in FinalPart[FP]} d_{s_i,bind(a_j)}}{|FinalPart[FP]|} \Big]$$

      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

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 [13]. 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.

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.

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

- after a fixed number of iteration
- 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.

Algorithm 6: Tabu search

```
Require: - S_g: greedy solution
Init: S_0 ← s_g
S ← S_0: current solution
S* ← S_0: the best-known solution
f* ← quality(S_0)
T ← {}: Tabu list
begin
    while (¬ StopCondidtion()) do
        S ← argmin_{S'∈Na(S)} [quality(S)]
        if quality(S) < f* then
            f* ← quality(S)
            S* ← S
            record tabu for the current move in T (delete
            oldest entry if necessary)
end
return S*
```

IV. 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 [18]. 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. [17] present another method for decentralized workflow execution based on partitions, but without considering data dependencies. More recently, Khalaf et al [11][10] present a method for decentralized orchestration of BPEL processes, focusing on the derivation of P2P interactions. Meanwhile, Yildiz et al [19] 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. [5] present an approach to partition BPEL processes using program partitioning techniques with the aim of reducing the communication costs between the partitions. However, they do not take into account distribution constraints (Collocate and Separate) so 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.

Other approaches to decentralized orchestration do not require any partitioning. For instance, the Self-Serv system [3][2] 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 1b). Another method for decentralized execution without partitioning is presented in [14][15]. 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.

V. Conclusion

This paper presented a method for optimized constrained decentralization of composite web services. The method seeks to create an activity partitioning and a binding of activities to services that minimizes 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 services in the same partition. The resulting model is richer than previous models for optimizing decentralized service orchestrations. The proposed method also complements existing methods for decentralized orchestration of services that take as input a predetermined partitioning.

Because of the nature of the objective function, we had to formulate the problem as a quadratic assignment problem. A greedy heuristic is used in order to construct an initial solution. The paper also sketched how Tabu search could be used to improve this initial solution. Future work will aim at empirically assessing the quality of the solutions obtained with the greedy algorithm, and the improvements obtained using Tabu search or other meta-heuristics.

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

References

[1] E. Aarts and e. J. K. Lenstra. In *Local Search in Combinatorial Optimization, Wiley, Chichester*, 1997.
[2] 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.
[3] B. Benatallah, Q. Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, 2003.
[4] 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.
[5] 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.
[6] M. Dumas, L. Garca-Bauelos, A. Polyvyanyy, Y. Yong, and L. Zhang. Aggregate quality of service computation for composite services. In *In Proc. of 8th Int. Conf. on Service-Oriented Computing, ICSOC 2010*, San Francisco, CA, Dec. 2010 (To appear).
[7] W. Fdhila and C. Godart. Toward synchronization between decentralized orchestrations of composite web services. In *CollaborateCom 2009, 5th International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 1 –10, 11-14 2009.
[8] 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.
[9] F. Glover and M. Laguna. Tabu search, 1997.
[10] R. Khalaf, O. Kopp, and F. Leymann. Maintaining data dependencies across bpel process fragments. *Int. J. Cooperative Inf. Syst.*, 17(3):259–282, 2008.
[11] R. Khalaf and F. Leymann. E role-based decomposition of business processes using bpel. In *ICWS*, pages 770–780, 2006.
[12] Koopmans and M. J. Beckmann. In *Assignment problems and the location of economic activities*, volume Econometrica, pages 53–76, 1957.
[13] P. Merz and B. Freisleben. Greedy and local search heuristics for unconstrained binary quadratic programming. *J. Heuristics*, 8(2):197–213, 2002.
[14] S. Mitra, R. Kumar, and S. Basu. Optimum decentralized choreography for web services composition. In *IEEE SCC (2)*, pages 395–402, 2008.
[15] S. Mitra, R. Kumar, and S. Basu. A framework for optimal decentralized service-choreography. *Web Services, IEEE International Conference on*, 0:493–500, 2009.
[16] A. Polyvyanyy, L. Garcia-Banuelos, and M. Dumas. Structuring acyclic process models. In *Proceedings of the 8th International Conference on Business Process Management.*
[17] W. Sadiq, S. W. Sadiq, and K. Schulz. Model driven distribution of collaborative business processes. In *IEEE SCC*, pages 281–284, 2006.
[18] 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.
[19] U. Yildiz and C. Godart. Information flow control with decentralized service compositions. In *ICWS*, pages 9–17, 2007.
[20] 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.