DYNAMIC DECENTRALIZED SERVICE ORCHESTRATIONS

Ustun Yildiz[1,2] and Claude Godart[1]

[1]*INRIA - LORIA*
*BP 239 Campus Scientifique*11
F-54500 Vandœuvre-lès-Nancy, France

[2]*Gabriel Lippmann Research Center*
41, rue du Brill
L-4422 Belvaux, Luxembourg

Keywords: Web Services, Decentralized Orchestration, Peer-to-Peer Computing, Workflow, Dynamic Service Composition.

Abstract: This paper reports a new approach to decentralized orchestration of service compositions. Our contribution is motivated by the recent advances of service orchestration standardizations that enable the exchange of modeled processes among different tools and organizations. Precisely, we provide an efficient process transformation technique that converts a process conceived for centralized orchestration to a set of nested *peer processes*. Each peer process is conceived to be executed by a dynamically orchestrated service. Assuming that services are invokable with peer processes, we provide a decentralized orchestration setting where services can establish direct interconnections via the peer processes that they execute. Our proposition considers conversation-based services and dynamic service binding.

1 INTRODUCTION

Web services are emerging as a new standardized way to design and implement business processes within and across enterprise boundaries. Basic XML-based service technologies provide simple but powerful means to model, discover and access software applications over the Web. The term *orchestration* is used to describe the design and implementation issues of the combination of distinct services into a coherent whole in order to fulfill sophisticated tasks that cannot be fulfilled by a single service. Service orchestration and workflow management have many similarities for both design and execution aspects. A service orchestration environment implicitly assumes a centralized execution setting where orchestrated services interact with a centralized orchestrator service as analogous to traditional workflow management. As the relevant workflow literature (Chen and Hsu, 2001)(Alonso et al., 1995) confirms, this approach falls short of supporting a wide range of ubiquitous, mobile , large-scale and secure process management. In view of these challenges, it is desirable to implement decentralized orchestration settings where orchestrated services can establish direct interconnections following the principles of peer-to-peer computing. The key question of this approach is how composed services that work following simple ”request-response” and ”one-way” communication modalities can be able to establish direct interconnections with services other than the orchestrator that invokes them for the purpose of the same process that they are involved in.

In addition to basic service technologies, service orchestration is the subject of various specification and standardization efforts. Among others WS-BPEL(IBM et al., 2005) (BPEL for short) has emerged as the de-facto standard for implementing service orchestration by means of processes(Weerawarana et al., 2005). A major outcome of process standardization is the common comprehension and execution semantics that they provide. With respect to commonly agreed process specifications, processes can be moved between distinct tools and organizations and then it is very reasonable to assume that orchestrated services can execute processes. Consequently, a service that can behave as an orchestrator can manage sophisticated interactions with other services following the application logic of the process that it executes. Following the motiva-

Yildiz U. and Godart C. (2007).
DYNAMIC DECENTRALIZED SERVICE ORCHESTRATIONS.
In *Proceedings of the Third International Conference on Web Information Systems and Technologies - Internet Technology*, pages 36-45
DOI: 10.5220/0001273000360045

tion that services can be able to execute processes, this paper takes on the challenge of designing a technique that enables decentralized orchestration. Our proposition introduces the concept of *peer process*. Peer processes are partitions of centralized orchestration specification. Each peer process is executed by a dynamically orchestrated service. Peer processes behave as cooperating workflows during the execution and enable truly peer-to-peer interactions of orchestrated services. Thus, our precise contribution is to provide such peer processes using initial peer process specification. As we mentioned above, our approach assumes that services can execute processes which can be also considered as they are invokable with processes rather than their regular input. The proposed approach is not an attempt to solve all of the problems of decentralized process executions, our aim is to provide a level of automation for the distribution of relevant peer processes on autonomous services. In the remainder of this paper, we present our model that enables decentralized orchestration with peer processes. The proposed technique that provides peer processes deals with many different aspects of decentralized orchestrations. First, it supports dynamic binding of services and introduces a new synchronization technique for services that are likely to be concurrently bound. We consider the orchestration of conversation-based services where service interfaces externalize the business protocol supported by the latter. Moreover, with the modeled processes, we are taking sophisticated control and data dependencies of process activities into account. Next subsection presents a motivating example and overviews our approach. Section 2 describes a language independent process model that we use to reason with processes. Section 4 presents the core of our contribution. Section 5 discusses the current status of our architectural consideration while the last section reviews very related works and concludes.

1.1 Motivating Example and Overview

As example scenario, we consider the manner in which the claim of a policyholder is handled by an Insurance Company (IC) service. Figure 1 outlines the process executed by IC. It depicts different activities that consist of service interactions with their control/data dependencies and interacting service identities. A control dependency expresses a precedence relationship of two activities while a data edge means that the data provided by an activity is used as the input of another. In the process, first, IC contacts Emergency Service(ES) that the policyholder contacted at the time of accident in order to receive the information relevant to Hospital(H) and Police(P) services that hold accident reports. After the invocation of ES, P and H are invoked with the outcome provided from ES and a Delivery Service (DS) is provisioned. It should be noted that DS provisioning activity (a_3) does not require any data provided by ES. However, DS requires the outcome of H when it is invoked a second time by activity a_6. After the invocations of P and H, concurrently received reports are used to invoke an independent Inspection(Ins) service to decide whether the claim must be reimbursed. The outcome of Ins and H are sent to the policyholder using DS (a_6). In this example, DS is a conversation-based service as it is invoked twice during the same process.

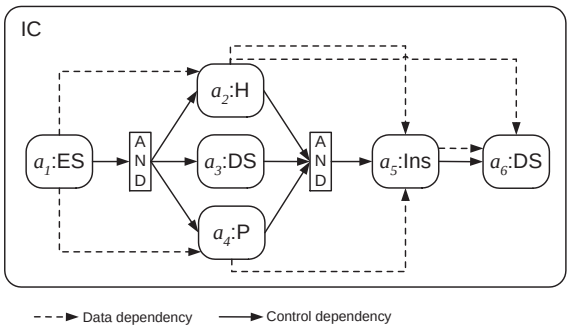

Figure 1: Process Example for Insurance Case.

Centralized Orchestration. In a centralized orchestration, all of the exchanged messages flow back and forth through IC that is the centralized orchestrator service. For example, when IC invokes ES, the former receives the outcome of ES and then uses it to invoke H and P. Similarly, the outcomes of H and P are received by IC and sent to Ins from IC. In the same time, the outcome received from H is also sent to DS from IC in a successive stage of the process.

Decentralized Orchestration. If a decentralized orchestration setting is required, IC can invoke ES and ES that terminates its execution can send its outcome to H and P. In the same time, as the first activity that interacts with DS(a_3) has a incoming control dependence with ES, ES can invoke DS. Similarly, H and P can send their outcomes directly to Ins. It should be noted that in order to execute a_5 of Ins, a_3 must be terminated. Consequently, DS must inform Ins about the termination of a_3. When H sends its outcome to Ins as the input of a_5, it can send the same outcome to DS as it is used by a_6 too.

Dynamic Service Binding. One important point that must be considered is the dynamic binding of services rather than their binding at design time. Following the same decentralized approach, ES must be able to dynamically bind H, P and DS before it sends its out-

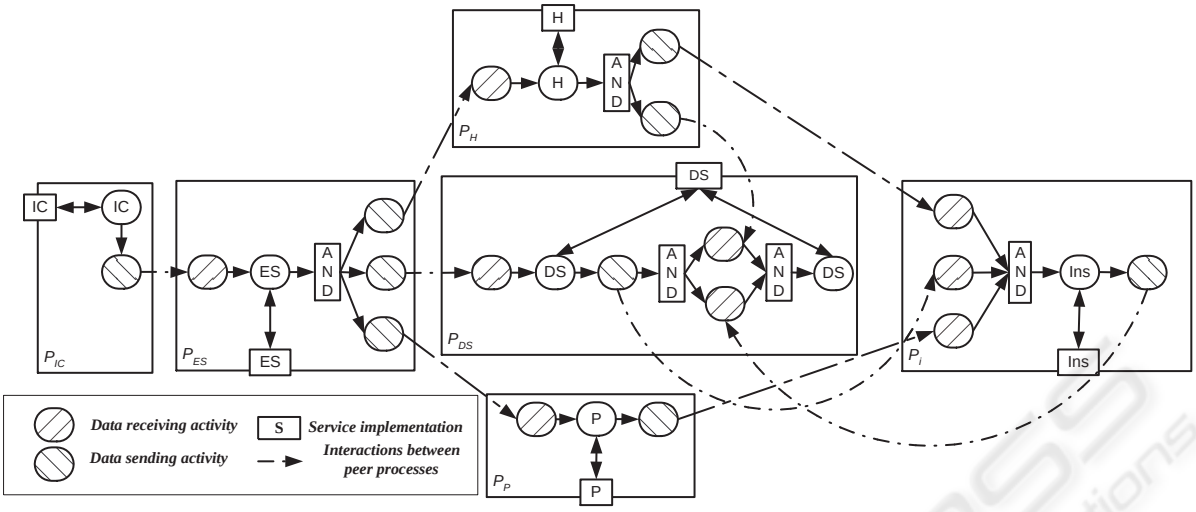

Figure 2: Decentralized Orchestration of Insurance Case.

come to them. In their turn, H, P and DS can bind commonly an Ins that they can invoke. It should be noted that Ins must not bind the service DS to which it can send its outcome, because DS has been bound in its previous step by ES. However, Ins must know the service DS to which it must route its outcome.

Decentralized orchestration can be enabled if the service interfaces support sophisticated P2P interactions. For example, in the centralized orchestration H must have some operations that receive the identity of the service Ins to which it must route its outcome instead of responding to IC that invokes it. Thus, in centralized execution, with its regular input IC can inform Ins about where to send its relevant outcome. However, if all of the service interfaces do not include the relevant operations to support decentralized orchestration, it is not possible to have the above setting.

With our approach, we aim to enable sophisticated P2P interactions via the peer processes that are executed by orchestrated services. A peer process that a service S executed is noted P_{S}. Let's take the service H. With respect to its simple interface, we aim to allow it to route its outcome to Ins. We make it execute a peer process P_{H} that receives the data sent by ES, invokes the core operations of H characterized by a_2, and routes the outcome of a_2 to Ins which executes in fact P_{Ins}. In the same way, P_{H} send the same outcome to P_{DS}. It should be noted that ES is supposed to execute its peer process denoted by P_{ES} that receives the invocation message of P_{IC}, executes a_1 and routes the outcome of a_1 to P_{H} and P_{P}. The same setting is considered for other services that have to route their outcome to other services directly. Figure 2 depicts a decentralized setting of the motivating example with peer processes. As readers will notice immediately, peer processes are not direct partitions of the original process specification. They include additional activities that enable interactions with other processes. If the dynamic binding is supported, peer processes must include activities that accomplish service binding operations. For example, if DS, H and P are required to be dynamically bound, the first peer process that must interact with them, which is P_{ES}, must bind them. Moreover, their corresponding peer processes that are P_{DS}, P_{H}, P_{P} must be deployed by P_{ES} in order to involve them in the overall decentralzied orchestration. Consequently, P_{ES} must include the peer processes that it must deploy.

2 PROCESS MODEL

This section describes the formal underpinnings of our approach that requires rigorous semantics on modeled processes. It is important to clarify what we do not intent to propose a new process language or extends an existing one. The presented concepts are applicable to a variety of process modeling languages and execution engines. A process can be represented as a message passing program modeled by a directed acyclic graph (Leymann and Roller, 2000)

Definition 1 (Process) *A process P is a tuple ($\mathcal{A}$, $\mathcal{C}$, $\mathcal{E}_c$, $\mathcal{E}_d$, $\mathcal{D}$) where $\mathcal{A}$ is the set of activities, $\mathcal{C}$ is the set of control connectors, $\mathcal{E}_c \subseteq \mathcal{A} \times \mathcal{C} \times \mathcal{A}$ is the set of control edges, $\mathcal{E}_d \subseteq (\mathcal{A} \times \mathcal{A}) \times \mathcal{D}$ is the set of data edges and $\mathcal{D}$ is the set of data elements.*

A process P has a unique start activity, denoted with a_s, that has no predecessors, and has a unique final ac-

tivity, denoted with a_f, with no successors. A control edge from an activity a_i to another activity a_j means that a_j can not be executed until a_i has reached a certain execution state (termination by default). A data edge exists between two activities if a data is defined by an activity and referenced by another without interleaving the definition of the same data. For example, a data received from a service s_i with the execution of an activity a_i can be used as the input value of an activity a_j that consists of an interaction with a service s_j. In this case, there is a data edge between a_i and a_j. It should be noted that our process modeling does not consider control edges between control connectors. We associate $source, target$: $\mathcal{E}_c \cup \mathcal{E}_d \rightarrow \mathcal{A}$ functions return the target and source activities of control and data edges. With respect to control flow we can also define a partial order, denoted by $\succ$, over $\mathcal{A}$, with $\succ \subseteq \mathcal{A} \times \mathcal{A}$ such that $a_i \succ a_j$ means there is a control path from a_i to a_j.

A service oriented process consists of synchronous/asynchronous interactions with services and locally executed activities such as variable assignments. We can define a function *ser*: $\mathcal{A} \rightarrow \mathcal{S}$ that returns a service interacted with an activity. As the dynamic service binding is supported, the return value of function *ser* can be a real service instance if there is already a bound service and if not it can be another value that identifies the service that is supposed to interacted with that activity. Thus, we can define a process activity as below.

Definition 2 (Activity) *An activity a is a tuple (in, out, s, $type$) where in, $out \in \mathcal{D}$ that are input and output values of a, $s \in \mathcal{S}$ is the interacting service with $\mathcal{S}$ is the set of services, and $type \in \Sigma_{\mathcal{A}}$ is the type of activity with $\Sigma_{\mathcal{A}}$ is the set of activity types.*

The set of activity types can be defined as follows $\Sigma_{\mathcal{A}}$ = {*read, write, write/read, local*} where *read* activities receive data from services, *write* activities send data to services, *write/read* activities consists of synchronous interactions that sends a data and waits the reply and finally, *local* refers to activities that do not consist of service interactions. During the execution of sophisticated processes, some services can be used with conversations. This means that different operations of the same service can be interacted several times with distinct activities. The activities that characterize these conversation-based interactions with the same service are *correlated activities*[1]. In our example, activities that refer to interactions with the same DS service are correlated. The set of correlated activities belonging to a service s_i is denoted by $\mathcal{A}_{s_i}$.

[1]Note that the term of correlation is literally different from the usual correlation term used in BPEL

Activities of a process are structured using control connectors (or control flow patterns) (van der Aalst et al., 2003). A control flow connector can be seen as an abstract description of a recurrent class of interactions based on partial ordering of activities. For example, the AND-split connector describes a choreography by activation dependencies as following: *an activity is activated after the completion of several other activities.* Following the works presented in (van der Aalst et al., 2003) that discusses the concrete implementation of patterns, we can consider a set of seven relevant control connectors to use to express different activation dependencies of activities. We gather them into three categories as follows: Sequence, Split = {AND-split, OR-split, XOR-split} and Join = {AND-join, OR-join, XOR-join}. For each activity, with respect to its incoming and outgoing control and data edges, we can identify two sets that include corresponding source and target activities pointed by these edges. The preset of an activity a is $\bullet a = \{a_i \in \mathcal{A} | \exists d \in \mathcal{E}_d \vee \exists c \in \mathcal{E}_c, (source(d) = a_i \wedge target(d) = a) \wedge (source(c) = a_i \wedge target(c) = a)\}$. The postset of an activity a is $a\bullet = \{a_i \in \mathcal{A} | \exists d \in \mathcal{E}_d \vee \exists c \in \mathcal{E}_c, (source(d) = a_i \wedge target(d) = a) \wedge (source(c) = a_i \wedge target(c) = a)\}$.

3 PEER PROCESSES

In this section, we describe the core of our decentralization approach. First, we focus on issues related to the dynamic service binding. Next, we present our process transformation technique that produces relevant peer processes.

3.1 Service Binding

Service binding is the identification of a concrete service for the fulfillment of a single or set of correlated activities. Both in centralized and decentralized execution settings, service binding operations can be done at different stages of process life cycle. Basically, a service instance (or endpoint) can be identified prior to execution or it can be dynamically bound during the process execution when it must be interacted. The difference of decentralized orchestration is that there is no centralized entity that does this operation. So, if dynamic service binding is supported, orchestrated services are also responsible for service binding operation. It is intuitive that if a peer process must send a coordination message or a process data to another peer process, executed by another service, it must know that process. If the corresponding service end-point is not bound yet, the peer process that

wants to interact with it, can bind it and deploy its peer process. Consequently, the outcome can be sent to the bound service. It should be noted that the bound service identity must be propagated to other peer processes that must interact with the same peer process. At this point, we assume that control and data messages exchanged among peer processes include also bound service identities.

Example 1 (Service Binding) *In the insurance orchestration example, P_{ES} must send the outcome of ES to H and P. In the same time, DS must be invoked following the termination of activity a_1. If at the moment P_{ES} must send its outcome, if there are no bound H, P and DS services, P_{ES} can bind H, P, DS and deploy their peer processes (this can be explained also bound services can be invoked with their respective peer processes that are P_H, P_{DS}, P_H). Thus, H and P can receive data sent by P_{ES} by executing the first activities of their peer processes and accomplish the rest of their role following their deployed peer processes. In this case, P_{ES} must include relevant activities that bind services and also deploy corresponding peer processes. In their turn, P_H, P_P and P_{DS} must interact with P_{Ins}. If Ins is expected to be dynamically bound, P_H, P_P, P_{DS} can bind a service Ins and send their outcome. However, the concurrent binding can cause a synchronization problem as the former may bind different Ins service instances.*

As explained in the above example, when peer processes are produced, service binding operation must be taken into account such that only one peer process binds a precise service and other dependent peer processes learn the bound service. In order to do so, we use *service binding dependencies*. A service binding dependency interconnects two activities. It characterizes service binding and peer process deployment activities that must be executed by the peer process that executes the source activity of the binding dependency. With service binding dependencies we aim to prevent concurrent service binding problem by privileging a single peer process such that other dependent peer processes can learn bound services. It should be noted that during run-time binding operation is done if there is no service yet bound. Moreover, if an activity that belongs to a set of correlated activities is considered, the binding operation is done for the very first activity and correlated activities are initiated with the identity of bound service. In the example, DS is bound only once for the first activity that it is interacted with. The binding policy that we adopt postpones service binding operation as late as possible such that a service is bound in a stage that every peer process that must interact with bound service's process can learn its identity correctly. The identification of a binding dependency that targets an activity is governed by the control and data dependencies that target it.

Below, we make a classification of the situations that define binding dependencies among activities. With respect to the different dependencies of activities in the centralized specification and to the underlying services, we identify two cases.

Case 1: In the preset activities of an activity a for which its fulfillment a service is supposed to be dynamically bound, there can be one or several activities. The respective peer processes that execute these preset activities will have to send messages to the peer process of a. In the first case that identify binding dependencies, if there is a unique preset activity a_b that precedes all of the rest, there is a binding dependency between this activity and a. This means that the peer process that executes a_b must bind the service for the fulfillment of the activity a and deploy its peer process. It should be noted that a service can be dynamically bound to an activity if there is no correlated activity that precedes the former. More formally this binding dependency can be explained as follows:

$|\bullet a| \geq 1$, $\exists! a_b \in \bullet a$ such that $\forall a_i \in \bullet a$, $a_b \succ a_i \Rightarrow a_b \xrightarrow{b} a$.

The relation $\xrightarrow{b}$ characterizes the binding dependency of two activities. Activity a_b is executed by the first peer process that must interact with the peer process of a. Naturally, the service invoked by a must be bound by the peer process of a_b. However, a_b must be unique and it must precede all preset activities, because the bound service identity is expected to be propagated to other peer processes that execute other peer processes that include preset activities. For example, if we consider the activity a_2 that interacts with H, in its preset, there is only a_1 that interacts with ES. As H is expected to be dynamically bound and there is no other preceding activity that interacts with H, there is a binding dependency between a_1 and a_2 that invoke respectively ES and H.

Case 2: The preset of the activity a include several activities. In contrast to Case 1, there is not an a_b in the preset that precedes all of the preset activities. Thus, an activity outsite of preset activity is considered as the source of the binding dependency that targets a. Intuitively speaking, an activity that precedes all of the preset activities is chosen as the source of binding dependency. Formally this situation can be described as follows:

$|\bullet a| > 1$, $\nexists a_b \in \bullet a$, $\exists! a_{bb} \in \mathcal{A}$ such that $\forall a_i \in \bullet a$, $a_b \succ a_i$, and $a_{bb} \succ a_i \Rightarrow a_{bb} \xrightarrow{b} a$.

For example, in contrast to case 1, a_5 has three preset activities. Consequently, if Ins is expected to be dynamically bound, three peer processes of these activities can not concurrently bind a service Ins as they might not bind the same service. The privilege can not be given to one of them as they have no interactions before the invocation of Ins. In this situation, we identify an activity that precedes all preset activities as the source of binding dependency. In the example, this activity is a_1 that interacts with ES. With this dependency, after a_1 is executed, P_{ES} binds a service Ins. Furthermore, as P_{ES} has interactions with P_{H}, P_{DS} and P_{P}, it can inform them about the relevant identity of Ins service that allows them to send their outcome to the common Ins.

By definition, there can be more that one activity a_{bb} that precedes all of the preset activities. If this is the case, one of them must be privileged for the binding dependency, thus the peer process that includes it can bind a service to a and peer processes that execute preset activities can learn bound service correctly. As we mentioned above, we aim to bind a service as late as possible. Consequently, among others a_{bb} must be an activity that precedes preset activities but succeeds other activities that can be the source of binding dependency. The sketch of the algorithm that defines the activity a_{bb} that is the source of the binding dependency is informally described above:

Var:
$\widehat{a_{bb}}$: *the set of all* a_{bb} *that* $\forall a_i \in \bullet a$, $a_i \prec a_{bb}$.
begin
if $\exists! a_{bb\_min} \in \widehat{a_{bb}}$ *that succeeds all activities* **then** a_{bb_min} *is the source of binding dependency to* a.
if a_{bb_min} *is not unique* **then** *one of them is chosen as the source of binding dependency (e.g. the activity with the smallest identification)*
end

It should be noted the above algorithm is one of the possible solutions for privileging an activity as the source of binding dependency. The important point that governs the operation is the precedence of the source activity comparing to preset activities. With service binding dependencies, each activity can have at most one incoming binding dependency. However, it can have several outgoing binding dependencies. In order to characterize the activities for which an activity a is the source of their incoming binding dependencies, we use $a(b)$. For each element a_i in $a(b)$, a is the source of the binding dependency that targets a_i.

4 PEER PROCESS

The definition of binding dependencies is the very first step of our decentralized orchestration approach. The essential part of decentralization is the characterization of peer processes. We call this step the production of peer processes as they are produced using a transformation that operates on centralized specification. Naturally, this step uses binding dependencies. Peer process production is accomplished in several steps. First, let us resume the content of peer processes. A peer process that is to be executed by a service s_i is denoted by P_{s_i}. As essential elements, P_{s_i} includes the elements of $\mathcal{A}_{s_i}$. Each element a_i of $\mathcal{A}_{s_i}$ is preceded and succeeded by activities that exchange coordination and data messages with relevant peer processes. Activities of $\mathcal{A}_{s_i}$ can have outgoing binding dependencies towards other activities which means that the execution of source activities must be followed by corresponding binding operations and the deployment of corresponding peer processes. Consequently, P_{s_i} must include the peer processes that must be deployed to the services that it has bound. However, the peer processes included within others are like regular data elements to the latter that include them. Another point that must be considered is the propagation of service identities that are bound following binding dependencies. When peer processes exchange coordination and data messages, they are expected to exchange the relevant information about the bound service identities that they received from others and bound themselves. Instead of proposing an additional coordination information that carry bound service identities, we prefer to integrate these information to exchanged coordination and data messages. For bound service identities propagation, the peer processes must be able to update messages with the service identities that they must send to other peer processes. Consequently, peer processes must include locally executed activities for update operations. In this way, when a peer process learns the identity of a bound service, it can update messages that it sends to other processes.

We identify two main steps to examine the production of peer processes. The first step operates on single activity level and consists of adding relevant activities that enable the interaction with other peer processes concerning that activity. The second step deals with two issues: the first is the structuring of correlated activities in peer processes. The second is the nesting of peer processes within each other.

Step 1: Adding relevant activities for coordination purposes. Each peer process P_{s_i} includes activities

of $\mathcal{A}_{s_i}$. Naturally, in order to be able to execute each element a_i of $\mathcal{A}_{s_i}$, P_{s_i} must collect all of the input data of a_i. Moreover, all of the control dependent preset activities of a_i must be terminated. This implies that P_{s_i} must collect also control messages of the preset activities of a_i. Thus, in P_{s_i}, each activity a_i is preceded by a number of activities that collect coordination and data messages coming from other peer processes that include preset activities of a_i. When all of the activities that collect messages related to a_i are executed, a_i can be executed by P_{s_i}. Peer processes that send messages related to a_i to P_{s_i} can be executed concurrently, however the synchronized reception of sent messages is required to execute a_i. Due to exclusive choice and merging situations (XOR-split and XOR-join) all of the preset activities might not be executed. Consequently, the exclusivity of preset activities must be taken into account when they are collected. The execution of a_i by P_{s_i} can produce an outcome to be sent to other activities to be executed in their proper peer processes. Moreover, control messages corresponding to the termination of a_i must be sent accordingly. As the reception of messages from peer processes that execute preset activities, the sending of coordination and data messages can be accomplished concurrently.

Example 2 (Step 1) *Step 1 is applied separately for each activity of the centralized specification. Let's take a_3 interacting with DS. The execution of a_3 requires the termination of a_1 interacting with ES. Consequently, P_{DS} must receive a control message about the termination of a_1 from P_{ES}. The same activity has an outgoing control dependency with a_5 interacting with Ins. Thus, its execution must be followed by an activity that sends a control message to P_{Ins}. If we take the second activity a_6 that interacts with DS, it has two preset activities (a_2, a_5) interacting respectively with Ins and H. Consequently, P_{DS} must collect the outcome of a_2 and a_5 from their corresponding peer processes that are respectively P_H and P_{Ins}.*

In addition to control and data edges that interconnect activities, there can be also binding edges that characterize binding dependencies and relevant binding activities. As we mentioned earlier, collected coordination and data messages of a_i contain also identities of previously bound services. These information and services identities bound following the execution of a_i must be propagated to interacted peer processes. Consequently, P_{s_i} updates the messages that it must send to other peer processes respectively.

Example 3 (Step 1 bis) *In our example, the activities a_1 and a_5 have a binding dependency which means that P_{ES} must bind a service Ins and deploy P_{Ins}. In the same time P_{ES} must bind H, P and DS.*

Algorithm 1 Adding relevant activities for coordination purposes

Input: $a_i \in \mathcal{A}$
Output: $\widetilde{a_i}$

1: AND-split
2: **for all** $a_j \in \bullet a_i$ **do**
3: *read($ser(a_j)$, $a_i.coor$) //concurrent reception of coordination messages from peer processes that execute preset activities*
4: **end for**
5: AND-join
6: *local(extract) //extraction of the input values of a_i and service identities from received coordination messages*
7: a_i //execution of regular activity
8: AND-split
9: **for all** $a_j \in a(b)$ **do**
10: *local(bind) // binding operations are executed*
11: **end for**
12: AND-join
13: *local(update) // coordination messages and peer processes are updated with service identities collected from other processes and bound by the current peer process*
14: AND-split
15: **for all** $ser(a_j)$ of $a_j \in a(b)$ **do**
16: *write($ser(a_j)$, $P_{ser(a_j)}$) //peer process deployment for bound services*
17: **end for**
18: AND-join
19: AND-split
20: **for all** $a_j \in a_i\bullet$ **do**
21: *write($ser(a_j)$, $a_j.coor$) // concurrent sending of coordination messages to peer processes that execute postset activities*
22: **end for**
23: AND-join

Furthermore, P_{Ins} must interact with P_{DS}. In order to let P_{Ins} to interact with P_{DS} (it must send a data message for the execution of a_6 in P_{DS}), P_{ES} must include the identity of DS in the P_{Ins} that it deploys. This operation is accomplished as follows: When P_{ES} is deployed to ES, it includes P_H, P_{DS}, P_P, P_{Ins} as data elements. When P_{ES} executes the service binding operating, it must update peer processes with bound service identities. This means that P_H, P_P and P_{DS} must be updated with the identity of Ins while P_{Ins} must be updated with DS's identity. Consequently, P_{ES} must include relevant local activities that implement update operations after the binding operations.

Algorithm 1 describes the pseudo-code of the mechanism that consists of the first step of transformation. We note the output of the algorithm that takes an activity a_i as input with $\widetilde{a_i}$. Besides the iterations, each instruction of the algorithm characterizes an activity or a control connector that must be added to $\widetilde{a_i}$. Step 1 operates on each activity of the initial process speci-

fication. It adds the relevant activities that ensures the coordination of peer process that includes this activity with other peer processes that include control and data dependent activities of the same activity. Coordination messages related to an activity a_i are denoted by $a_i.coor$. They can include either process data or light-weight coordination messages with bound services identities. Moreover, following the outgoing binding dependencies, they include also the identities of bound services. Locally executed activities let peer processes manage the content of the coordination messages exchanged with other peer processes.

Step 2: Structuring Correlated Activities and Iterative Nesting. The operation of step 2 is to gather correlated activities manipulated by step 1 within their corresponding peer processes and to nest peer processes within each other to support dynamic service binding and peer process deployment. Both operations are conducted together because when a peer process is produced, the peer processes that it includes, must have been produced in previous iterations. Consequently, the production of peer processes is started from the ones that do not include activities that have outgoing binding dependencies. This operation is followed by the processes that include produced peer processes.

The operation that gathers correlated activities is expected to preserve the partial order between them within their peer processes. Basically, two correlated activities $a_i \succ a_j$ of the centralized specification must preserve the same order $\widetilde{a_i} \succ \widetilde{a_j}$ in their peer processes after being transformed in step 1. If their partial order is total, $\widetilde{a_i}$ must precede $\widetilde{a_j}$. If a_i and a_j are on concurrent paths, the concurrency of $\widetilde{a_i}$ and $\widetilde{a_j}$ must be preserved in the peer process that include them. At this point, their exclusivity must be considered. This means that if $\widetilde{a_i}$ and $\widetilde{a_j}$ are exclusive activities and only one of them is executed then their peer process must structure them on exclusive paths. Similarly, if they are expected to be concurrently executed and synchronized before executing another correlated activity, they must be concurrently structured within their peer processes.

Example 4 (Step 2) *Figure 3 depicts the structuring of the correlated activities a_3 and a_6 that refer to DS. As we mentioned in the previous example, step 1 produces $\widetilde{a_3}$ and $\widetilde{a_6}$. When they are gathered in P_{DS}, $\widetilde{a_3}$ and $\widetilde{a_6}$ must preserve the same order of a_3 and a_6. Consequently, step 2 puts $\widetilde{a_6}$ after $\widetilde{a_3}$ with a sequential dependency.*

It should be noted that peer processes must be executable processes that satisfy properties such as deadlock freeness, soundness etc. In order to do so,

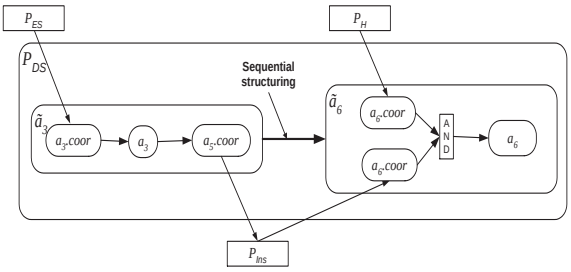

Figure 3: Structuring correlated activities in peer processes.

dummy activities and control connectors are added to peer processes.

Algorithm 2 depicts the mechanism that operates on the outcome of step 1. The operation starts with peer processes that have no activities with outgoing binding dependencies and iterates with ones that bind the latter. We can identify a set $\mathcal{P} = (P, \succ)$ with $P = \{P_{s_0}, , P_{s_n}\}$ being the set of peer processes. If there are two processes P_{s_i} and P_{s_j} with $P_{s_j} \succ P_{s_i}$, this means that s_i is bound by P_{s_j} and P_{s_i} is deployed by P_{s_j}. Consequently, P_{s_i} must be produced before P_{s_j}. In order to describe peer processes that have no outgoing binding dependencies, we use the set **min($\mathcal{P}$)**. Similarly, the partial order $\succ$ of centralized specification is the same for the correlated activities of a service s_i included in $\mathcal{A}_{s_i}$. For both sets, the function **prec()** returns the preceding element with respect to the partial order. As correlated activities cannot be always ordered, it is possible to have some subset of activities that are incomparable. We note activities that are incomparable by a set denoted as **Inc**$_i$. All of the activities of an incomparable set **Inc**$_i$ are included in a common split and a join point in the centralized process specification. In the peer processes, after being transformed with step 1, these activities must be structured within common split and join connectors. The common split and join points of the activities of an **Inc**$_i$ are defined as below:

Definition 3 (Split-point) *Let **Inc**$_i$ be a set of unordered concurrent activities. A split-point of **Inc**$_i$ is the first backward control connector c such that its preceding activity a_{split} precedes all elements of **Inc**$_i$. More formally, $\forall a_i \in$ **Inc**$_i$, $a_i \succ a_{split}$.*

Definition 4 (Join-point) *The joint point of **Inc**$_i$ is the first forward control connector c such that its successive activity a_{join} follows all elements of **Inc**$_i$. More formally, $\forall a_i \in$ **Inc**$_i$, $a_{join} \succ a_i$.*

The type of split or join connectors is defined according to activation conditions of the elements of **Inc**$_i$. For example, if there are two activities $\widetilde{a_i}$ and $\widetilde{a_j}$ that are not activated because of their exclusivity, they are

Algorithm 2 Structuring and nesting peer processes

Input: $\mathcal{A}_{s_i} \subset \mathcal{A}$ //The algorithm is executed for correlated activities modified with the step 1
Output: P_{s_0} //The peer process to be executed by the initial service
1: $P_{s_i} \leftarrow min(\mathcal{P})$ //initialization of peer processes with a process that does not include any activity with outgoing binding dependencies
2: $a_i \leftarrow a_f$ //initialization of the current activity with the final activity of the produced process
3: **for all** $P_{s_i} \in P$ **do**
4: **for all** $\widetilde{a_i}, a_i \in \mathcal{A}_{s_i}$ **do**
5: **if** $\exists$**Inc**$_i \subset \mathcal{A}, a_i \in$ **Inc**$_i$ /* if a_i is a set of concurrent activities, split and join point detection procedures are started */ **then**
6: **call**$_{split}$**(Inc**$_i$**)**
7: **call**$_{join}$**(Inc**$_i$**)**
8: $\widetilde{a_i} \leftarrow$ **prec(**$\widetilde{a_i}$**)** //the structuring is repeated with preceding activities that are not in **Inc**$_i$
9: **else**
10: Sequence //If there is no **Inc**$_i$ that a_i belongs to, sequential structuring is considered
11: $a_i \leftarrow$ **prec(**a_i**)** //the structuring is repeated with preceding activities
12: **end if**
13: **end for**
14: $P_{s_i} \leftarrow$ **prec(**P_{s_i}**)** //the operation is repeated for peer processes that include produced processes
15: **end for**

split by an XOR-split. Similarly, if activities are precisely executed, they are structured after an AND-split. Due to lack of space, we don't give the full details of mechanism that identifies control connectors of an **Inc**$_i$. In the algorithm, we call the operation that identifies the split and join points of concurrent correlated activities with **call**$_{split}$**(Inc**$_i$**)**and **call**$_{join}$**(Inc**$_i$**)**. The structuring algorithm begins with peer processes that do not deploy other peer processes(**min(**$\mathcal{P}$**)**) and continues with those that include them. When a peer process is structured, the structuring operation is started with the final activity and continues toward the beginning.

5 ARCHITECTURAL CONSIDERATIONS

The abstract concepts described in this paper are applicable to a variety of process modeling languages and execution engines. However, it is important to clarify some of the implementation details. In this section, we present the current state of our implementation.

We use BPEL that characterizes the current state of art in process modeling standardization efforts, as

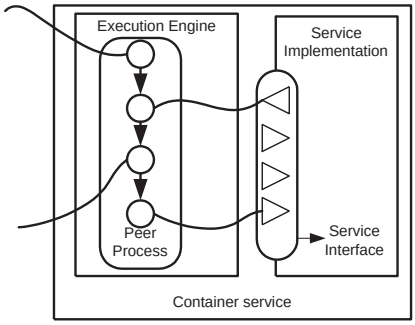

Figure 4: Implementation overview.

process specification. In contrast to centralized execution settings, decentralized executions require additional assumptions on composed services. As we assume services to execute relevant peer processes, the core services must be implemented inside services that have process execution engines such as BPWS4J(cf. Figure 4). The container services must be invokable with peer processes. When they are invoked with peer processes, they must also implement corresponding WSDL interfaces with corresponding `portTypes` and `partnerLinkTypes`. The basic method to support the dynamic service binding can be supported with the reassignments of `partnerlinks`. At this point, we assume a local method that interacts with core service implementation reassigns required partnerlinks for dynamic service binding purposes.

The essential operation of decentralization that provides the nested peer processes is accomplished on a single site which can be the first service where the overall process starts. BPEL allows specifying a process in different ways. Typically, there are two essential ways, additionally a third style that combines both can be considered. The first is the block-oriented approach whereby there process is modeled through nested use of structured activities (sequence, flow, while, switch). The second is the explicit use of link constructs that allow the expression of arbitrary precedences and related join/transition conditions. We use block-oriented modeling that makes the data and control dependence analysis of processes easier. The core module that produces peer processes is a Java application that uses XPath to analyze BPEL specification. Xpath expressions query BPEL process for control and data dependencies of basic activities that consist of service interactions. Analyzed dependencies are stored to define binding dependencies. We prefer to not manipulate BPEL process that is analyzed as the concurrent analyze and manipulation of tree structures have an important complexity.

6 RELATED WORK AND CONCLUDING REMARKS

The idea of decentralized execution setting is not new. Issues related to decentralized execution of workflows has been considered as very important in the relevant research literature a decade ago. If we consider previous works that are not relevant to service oriented paradigms and technologies, much of them are conceived to prevent the overloading of a central coordinator component. Their basic proposition is to migrate process instances from an overloaded coordinator to a more available according to some metrics (e.g. (Bauer et al., 2003)) or execute sub-partitions of a centralized specification on distinct coordinators (e.g. (Wodtke et al., 1997)). Although efficient, these approaches do not consider P2P interactions of workflow elements.

The implementation of decentralization is relatively new for service oriented applications. A simple but crucial observation is that some of the recent approaches deal with decentralization by using centralized services or additional software layers what can be a strong hypothesis in the Web context (Schuler et al., 2005)(Benatallah et al., 2002) rather than reasoning decentralization with decentralized processes. In contrast to the latter, the unique assumption of our approach for orchestrated services is the capability of executing processes. This is a reasonable assumption in the context of modern virtual organizations that already implement internally their published interfaces as processes.

(Nanda et al., 2004) proposes a process partitioning technique similar to ours. However, the dynamic service binding and conversations are not considered. The authors focus on the partitioning of variable assignment activities of a BPEL processes. The works presented in (Maurino and Modafferi, 2005) and (Sadiq et al., 2006) are similar.

This paper has taken on the challenge of designing a technique for translating a centralized orchestration specification to a set of dynamically deployable peer processes. The proposed approach aims to allow orchestrated services to have P2P interactions through the peer processes that they execute. When coupled with an efficient process transformation technique, we believe that the common process representation brought forward by standards can deal with architectural concerns of decentralized execution settings without making unreasonable assumptions about composed services. Our contribution stands between *Orchestration* and *Choreography* initiatives that govern the implementation of service oriented applications. We demonstrated how an orchestration description can be effective with a decentralized execution as analogous to choreography that requires much more sophisticated design and run-time reasoning.

REFERENCES

Alonso, G., Kamath, M., Agrawal, D., Abbadi, A. E., Gunthor, R., and Mohan, C. (1995). Exotica/ fmqm: A persistent messagebased architecture for distributed workflow management. In *IFIP Working Conference on Information System Development for Decentralised Organisations, Trondheim*.

Bauer, T., Reichert, M., and Dadam, P. (2003). Intra-subnet load balancing in distributed workflow management systems. *Int. J. Cooperative Inf. Syst.*, 12(3):295–324.

Benatallah, B., Sheng, Q. Z., Ngu, A. H. H., and Dumas, M. (2002). Declarative composition and peer-to-peer provisioning of dynamic web services. In *ICDE*, pages 297–308.

Chen, Q. and Hsu, M. (2001). Inter-enterprise collaborative business process management. In *Proceedings of the 17th International Conference on Data Engineering, ICDE*, pages 253–260.

IBM, BEA Systems, Microsoft, SAP AG, and Siebel Systems (2005). Business Process Execution Language for Web Services, version 1.1 (updated 01 feb 2005). http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.

Leymann, F. and Roller, D. (2000). *Production Workflow - Concepts and Techniques*. PTR Prentice Hall.

Maurino, A. and Modafferi, S. (2005). Partitioning rules for orchestrating mobile information systems. *Personal and Ubiquitous Computing*, 9(5):291–300.

Nanda, M. G., Chandra, S., and Sarkar, V. (2004). Decentralizing execution of composite web services. In *OOPSLA*, pages 170–187.

Sadiq, W., Sadiq, S., and Schulz, K. (2006). Model driven distribution of collaborative business processes. In *IEEE International Conference on Services Computing, SCC*.

Schuler, C., Turker, C., Schek, H.-J., Weber, R., and Schuldt, H. (2005). Peer-to-peer execution of (transactional) processes. *International Journal of Cooperative Information Systems*, 14(4):377–405.

van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. (2003). Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51.

Weerawarana, S., Curbera, P., Leymann, F., Storey, T., and Ferguson:, D. (2005). *Web Services Platform Architecture*. Prentice Hall PTR.

Wodtke, D., Weißenfels, J., Weikum, G., Dittrich, A. K., and Muth, P. (1997). The mentor workbench for enterprise-wide workflow management. In *SIGMOD Conference*, pages 576–579.