On Dead Path Elimination in Decentralized Process Executions

Ustun Yildiz

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On Dead Path Elimination in Decentralized Process Executions

Ustun Yildiz

N° ????

February 2007

Thème COG

ISSN 0249-6399 ISRN INRIA/RR--????--FR+ENG

On Dead Path Elimination in Decentralized Process Executions

Ustun Yildiz*

Thème COG — Systèmes cognitifs
Projet ECOO

Rapport de recherche n° ???? — February 2007 — 31 pages

Abstract: There has been a great deal of interest in recent years in the use of service oriented approach and relevant standards to implement business processes. Following the concepts of workflow-based process management, the major focus has been on service composition. Not surprisingly, this default composition approach suffers from the limitations of centralized workflow management. It is well recognized that a decentralized execution setting where composed services can establish P2P interactions, is central to a wide range of ubiquitous, mobile, large-scale and secure business process management. A natural way to enable the decentralized execution is to implement the relevant distributed cooperating processes of a centralized process on composed services. In this way, composed services can establish P2P interactions following the semantics of their processes. In this report, we present a generic approach that enables decentralized executions with such cooperating processes. Precisely, we present our method that derives the latter. We focus on the sophisticated control/data flow, conversational aspects and especially Dead Path Elimination that run counter to naive intuition, most of which, we explain using deeper analysis of the algorithms and data structures that we employed.

Key-words: Workflow, P2P computing, Program Partitioning, Business Process Management, Dead Path Elimination

Ce travail est partiellement supporté par les fonds du Ministère de la Culture, de l'Enseignement supérieur et de la Recherche de Grand-Duché de Luxembourg. Réf: BFR 04/108

* INRIA - LORIA, University of Nancy, ustun.yildiz@loria.fr

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

L’Élimination des Chemins Morts dans des Exécutions Décentralisées des Procédés

Résumé : L’Élimination des Chemins Morts dans des Exécutions Décentralisées des Procédés est une opération spćiale supportée par la plupart des outils de gestion de procédé centralisés. Ce travail porte sur la définition, l’analyze et l’adaptation de la même opération aux exécutions décentralisées des procédés.

Mots-clés : Workflow, Système d’Information Egal-à-Egal, Partitions des Programmes, La Gestion des Procédés Métiers, L’élimination des Chemins Morts

Contents

1 Introduction

Recent advances in service oriented computing and related standardization efforts have made the realization of B2C and B2B applications not only inexpensively feasible but also increasingly popular. The very recent efforts were spawned primarily from a need to express service compositions unambiguously, and to make compositions amenable to formal manipulations as analogous to classical workflow-based process management and component-oriented computing [19]. WS-BPEL (previously BPEL4WS or BPEL for short) [9] which is currently being standardized by OASIS [1] characterizes the current state of art in this field. On the other hand, standard process specifications allow modeled processes to be exchanged between different organizations/tools and executed without major architectural constraints [6].

Typically, a service composition is based on a centralized execution model. This means that composed services do not have direct interconnections with each other. As the relevant research literature on process management confirms [7][5][24], there are several different motivations to enable decentralized execution settings where composed services can establish P2P interactions. We use the terms *decentralized orchestration* and *decentralized workflow (process) management* to describe this mode of operation. Although the latter is not a new idea [2][20], previous propositions have not found their widespread applicability in practical settings. Their slow acceptance is often attributed to, among other things, their architectural assumptions on process participants [12][16] such as additional software layers. These assumptions become more unreasonable in the Web context as they are contradicting to the autonomy of services.

We believe that the secondary outcome of process specification standards, which is the interchangeability issue of processes between organizations, can make decentralized orchestrations inexpensive and reasonable to provide. Intuitively speaking, a process modeled by an organization can be received and executed easily by another one. From this point of view, if organizations execute relevant processes for the purpose of an overall composition, they can establish P2P interconnections following the semantics of the processes that they execute. At middleware level, this can be implemented with services that are invokable with processes rather than simple input values. The above motivations highlight the problems of arriving at a definition of an accurate operation that derives such processes executed by composed services i.e. how a control or data dependency of the centralized specification can be reestablished and preserved in a decentralized orchestration?

In this report, we describe and exemplify such a method that enables the decentralized orchestration. Our proposition does not attempt to discuss the feasibility of the whole issues of a centralized orchestration in a decentralized one. We discuss the paramount aspects such as *Dead Path Elimination (DPE)* and we propose corresponding solutions that run counter to naive intuition. Our proposition relies on well-known concepts of inter-organizational processes [17] and workflow literature [13].

The remainder of this report is organized as follows: Section 2 presents some preliminary concepts. Section 3 presents a formal process specification model. Section 4 presents the core of our contribution. Section 5 reviews the very related works while last section concludes.

2 Background

In this section, we review some general concepts of the workflow literature that are relevant to our work and we overview our approach.

2.1 Dead Path Elimination

Dead Path Elimination (DPE) is a particular procedure that most centralized workflow management systems support. Basically, it can be explained as follows: In some instances of some process models, some control paths may not be executed due to different transition conditions. If these control paths lead to a join activity, the latter may not be executed as it will wait the termination of all incoming branches [13]. DPE consists of the resolution of such blocking situations.

Example 1 (Simple Dead Path Elimination) *Consider the process model depicted in figure 1. The example consists of a claim handling process executed by an insurance company. The first activity is the Inspection that consists of the examination of the claim. According to the outcome of this activity, the claim is reimbursed or refused. If the claim is reimbursed then the Bank Order activity is executed, otherwise this activity is not executed. Prepare Report and Delivery Provisioning activities are executed in both cases. The Prepare Report activity consists of the preparation of inspection results to be archived by the insurance company. The Delivery Provisioning activity prepares a delivery service that will transfer the inspection result to claim's owner. Delivery activity is*

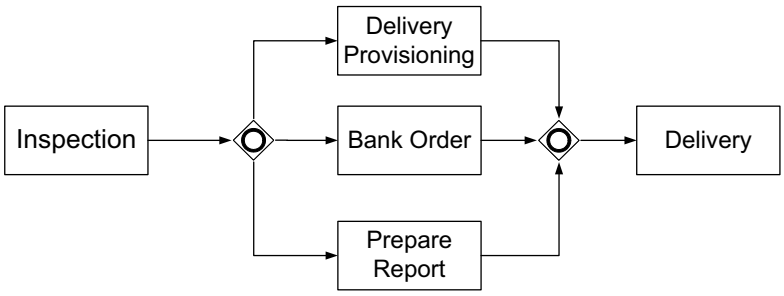

Figure 1: Dead Path Elimination in Centralized Process Execution

executed after the termination of all preceding activities. In this example, Delivery activity is the join activity and the path that includes Bank Order may not be executed. For an instance of the process, if the Bank Order activity is not executed, the corresponding join condition from Bank Order to Delivery will never be evaluated. Consequently, this instance will never terminate. DPE deals with this situation by evaluating all incoming transition of Delivery activity.

DPE computes recursively transitive closures until join activities and treats all transition conditions. Thus, the incoming transitions of a join activity are always evaluated. Interested are referred to [13] for the formal definition of DPE in centralized workflow management. It is fair to say that DPE procedure is extensively studied in centralized workflow management [13][18]. However, it needs reconciliation in decentralized workflow management.

2.2 Decentralized Orchestration

A Decentralized orchestration setting where composed services of a process can establish P2P interconnections can be enabled by several different manners. The most intuitive solution that can enable the decentralization is to implement some additional functionalities on composed services (e.g. [16][22]). Thus, in addition to their advertised behavior, composed services can support necessary explicit coordination features with each other. As we mentioned earlier, this solution can be efficient if composed service support such assumed functionalities. However, it is not reasonable to make architectural assumptions in the Web context as the latter are contradicting to the autonomy of services. Another way to enable decentralized orchestration is to compose services that advertise interfaces that support sophisticated interaction with each other. In this case, the challenge is to check the structural consistency of different service interfaces (e.g. [21]). It should be noted that, this kind of decentralized orchestration assumes the presence of services that can interact with each other. Our last consideration of decentralized orchestration, which is also the approach that we adopt in his report, is to have a centralized specification that composes a number services and to derive corresponding cooperative distributed processes of this centralized specification. Thus, each composed service can execute a process that interacts with other services that execute their corresponding processes. This approach assumes that services can receive and execute relevant processes. In contrast to pre-SOA propositions, this assumption is reasonable. Because, services advertise their capabilities that are already implemented by sophisticated processes beyond their interfaces and consequently, they dispose process execution environments. In addition to this assumption, XML based process specifications such as BPEL allow modeled or instantiated processes to be easily exchanged between organizations (services). To get a better understanding of our decentralization approach, let us consider the process depicted in figure 2. The process which is

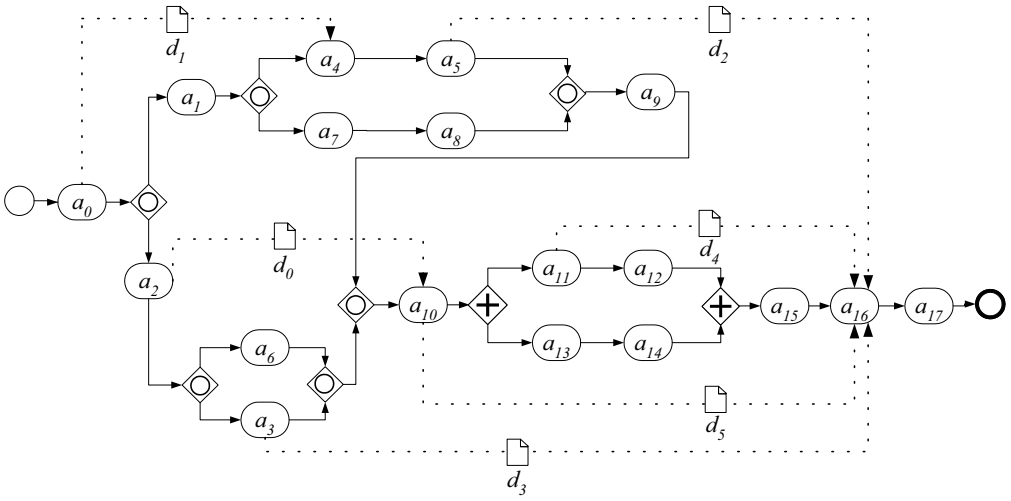

Figure 2: Motivating example: Centralized process

conceived for centralized execution, includes control and data dependencies of process activities. Activities consist of the invocations of different service operations. As we consider conversation-based services, different operations of a service can be invoked with the execution of different activities. Control edges characterize precedence relationships while data edges express input/output data relationships. For example, the output of a_{10} is used as an input of a_{16}. This dependence is expressed with the data edge d_5. This process can be executed by a centralized process execution engine (orchestrator, e.g. BPWS4J). In a centralized execution, services are isolated from each other. They interact only with a single service that orchestrates them. At this point, it may be of interest to provide a decentralized orchestration setting where services establish direct interconnections. For example, let's take a_3 and a_5 activities that consist of the respective invocations of services s_3 and s_5. In a decentralized execution s_3 and s_5 can route their outcomes (d_3 and d_2) directly to s_{16} that is invoked with the execution of a_{16}. Similarly, in order to respect the control flow, s_{15} invoked with a_{15} can inform s_{16} about the termination of the operation invoked by a_{15}. The same P2P interactions can be considered for all data and control edges. In order to enable such interactions, our approach derives processes to be executed by orchestrated services. The latter are called *peer processes*. Peer processes are derived from the centralized process specification. The interactions of peer processes are the decentralized implementation of the control and data dependencies of underlying services of the centralized specification. With this approach, s_3 and s_5 execute respectively their peer processes denoted P_{s_3} and P_{s_5}. They include activities that send d_3 and d_2 to s_{16} after executing respectively a_3 and

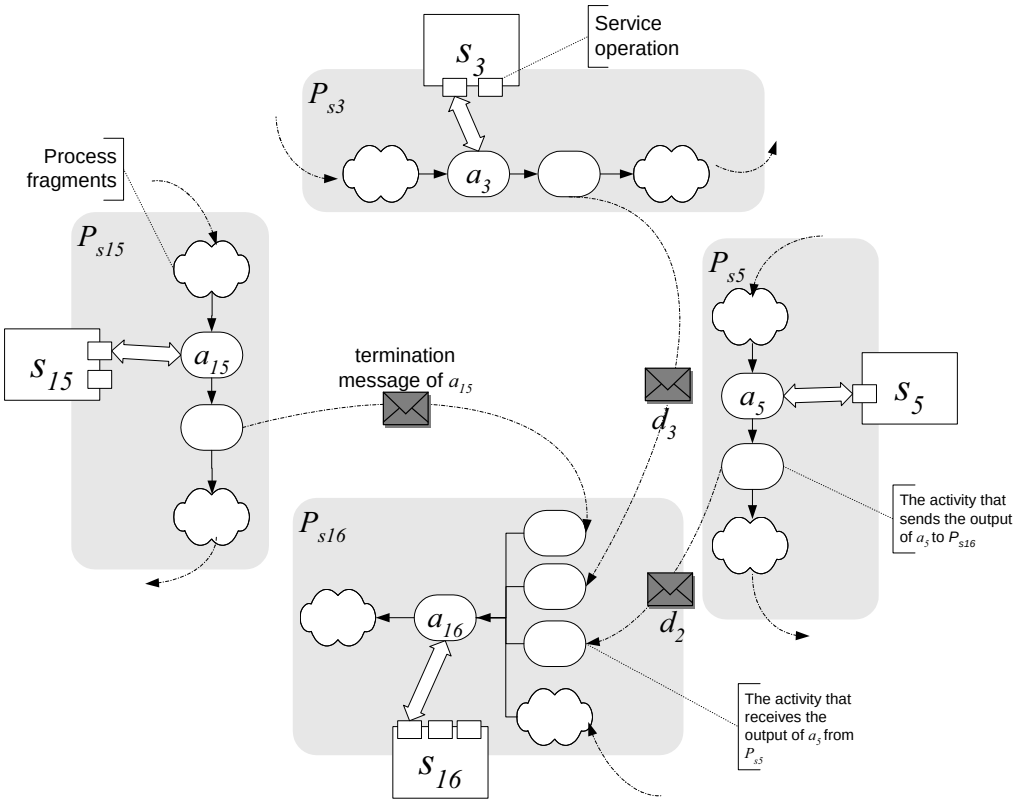

Figure 3: Motivating example: Decentralized process fragments

a_5. s_{15} executes $P_{s_{15}}$ and sends a control message to s_{16} that includes an information about the termination of a_{15}. s_{16} that executes $P_{s_{16}}$ includes receiving activities that collect the sent control messages and the data to use as the input of a_{16} and executes a_{16}. Thus, the semantics of centralized process are preserved with P2P interactions. Figure 3 depicts the decentralized implementation of the above interactions. As we mentioned earlier, the challenge of our work is to derive and deploy peer processes.

Decentralize orchestration must deal with some additional issues that are not extensively studied in the centralized workflow management. As we showed in the above example, the first issue that is taken into account, is different data dependencies of process activities. As centralized workflow management systems are empowered by centralized databases that store and manage process data, the process model does include explicit data dependencies. Data dependencies of the process activities can exist

under different forms. The data dependencies of our modeling include only input and output mappings between different activities. For example, we do not consider concurrent data access of parallel activities. The data dependencies exist along control paths. This means that if there is a data dependency between two activities a_i and a_j, there is also a control path from a_i to a_j.

2.3 Design Choices and Assumptions

Although we claim to provide a decentralized orchestration setting inexpensive to implement, it is necessary to review some of our general assumptions.

- **Asynchronous communication.** The underlying communication among orchestrated services is based on asynchronous communication that is the commonly implemented interaction modality over the Web. In this modality, the writing into a asynchronous communication channel is not blocking while the reading from a channel is blocking if the expected data is not available yet. We assume that asynchronous communication channels are perfect. i.e. messages sent along channels are not lost.
- **Structured processes.** Decentralized processes are structured processes [11]. This means that different activities are structured through control elements such as AND-split, AND-join, OR-split, OR-join and for each split element, there is a corresponding join element of the same type. Additionally, the split-join pairs are properly nested. This is a reasonable assumption as it is possible to transform arbitrary processes to their structured equivalent.

2.4 Global Soundess

Our method that derives peer processes is based on a well-known criterion that must be considered for cooperating processes. In [17], Van Der Aalst has proposed a simple but powerful correctness criterion *(global soundness)* for cooperating inter-organizational processes. Basically, this criterion states that each process involved in an asynchronous composition must terminate and upon termination, its asynchronous communication channels must be empty. If all of the composed processes are known, the global soundness can be automatically verified. If a global view of composed processes is not achievable, the same criterion or similar ones that replace the latter can be verified with alternative approaches such as [21]. When we provide peer processes, we rely on the original global soundness criterion. In contrast to all previous works in this field, our aim is not to verify the global soundness of existing processes but provide peer processes that will satisfy it. So, intuitively speaking our approach is feasible as the complete centralized specification is known and its corresponding peer processes are produced prior to execution.

3 Formal process modeling

The operation that derives peer processes that satisfy the global soundness is fairly sophisticated. The complexity of the former is directly related to the complexity of the decentralized process specification that can include control and data edges and conversational activities in various combinations. This section presents a formal process model that is necessary for a rigorous reasoning.

A process that subjects service composition can be considered as a graph-oriented message passing program [19].

Definition 1 (Process) *A process, P, is a tuple ($\mathcal{A}$, $\mathcal{E}_c$, $\mathcal{E}_d$, $\mathcal{C}$), where $\mathcal{A}$ is a set of activities, $\mathcal{E}_c$ is the set of control edges with $\mathcal{E}_c \subseteq (\mathcal{A} \cup \mathcal{C}) \times (\mathcal{A} \cup \mathcal{C})$, $\mathcal{E}_d$ is a set of data edges that interconnect process activities with $\mathcal{E}_d \subseteq (\mathcal{A} \times \mathcal{A})$ and $\mathcal{C}$ is a set of control connectors that characterize different transition conditions of control edges.*

A process activity $a \in \mathcal{A}$ consists of a one-way or bi-directional interaction with a precise service. So, each activity refers to its interacted service. The set of activities that refer to the same service s_i is noted as $\mathcal{A}_{s_i}$. A process has a unique start activity that has no predecessors and has a unique final activity 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 the completion state. 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. We limit our considerations with concurrent and selective concurrent branchings that are described with AND-split, AND-join, OR-split and OR-join patterns. In order to express control paths of process activities, we use a precedence relationship denoted $<$. It is a partial order defined over $\mathcal{A}\times\mathcal{A}$ that characterizes the existence of a control path between two activities. As a first step to reason with the complexity of the decentralization operation, we need to formally define different dependency types that can exist with regard to the expected interactions of underlying services and their peer processes. The model we use for this purpose, extends traditional $<$ to the notions of *strong/weak forward* and *backward* orders. In the following notations, $\widehat{\text{OR}}$ is used to denote a process fragment that begins with an OR-split and terminates with its corresponding OR-join. For any control connector cc, $\widehat{cc}$ denotes its corresponding split or join connector. Similarly, for an activity a_i that precedes a split point, $\widehat{a_i}$ denotes an activity that succeeds its corresponding join point and vice-a-versa.

Definition 2 (Strong/Weak Forward and Backward Orders) *Let a_i and a_j denote two activities. There are four possible order relations between them:*

- *Strong forward order ($<_s$): The order of a_i and a_j is a strong forward order (denoted $a_i<_s a_j$) if the following hold: $a_i<a_j$, $\nexists cc \in$ {OR-split} s.t. $(a_i<cc) \wedge (cc<a_j) \wedge (a_j<\widehat{cc})$*

- *Weak forward order ($<_w$): The order of a_i and a_j is a weak forward order (denoted $a_i<_w a_j$) if the following hold: $a_i<a_j$, $\exists cc \in$ {OR-split} s.t. ($a_i<cc$) $\wedge$ ($cc<a_j$) $\wedge$ ($a_j<\widehat{cc}$)*
- *Strong backward order ($>_s$): The order of a_i and a_j is a strong backward order (denoted $a_i>_s a_j$) if the following hold: $a_i<a_j$, $\nexists cc \in$ {OR-join} s.t. ($a_i<cc$) $\wedge$ ($\widehat{cc}<a_i$) $\wedge$ ($cc<a_j$)*
- *Weak backward order ($>_w$): The order of a_i and a_j is a weak backward order (denoted $a_i>_w a_j$) if the following hold: $a_i<a_j$, $\exists cc \in$ {OR-join} s.t. ($a_i<cc$) $\wedge$($\widehat{cc}<a_i$) $\wedge$($cc<a_j$)*

The above relationships formalize control dependencies of activities that are not directly connected by control edges with their possible pairwise completion states. For example, the weak forward order $a_0<_w a_4$ means that a_4 may not be executed after a_0. The weak backward order of $a_5>_w a_{16}$ means that at the moment a_{16} is executed, a_5 might not have been executed because of the selective path that it is on. Contrary, $a_{10}<_s a_{16}$ and $a_{10}>_s a_{16}$ mean that the execution of a_{16} always follows the execution a_{10} and vice-a-versa if a_{16} is executed, a_{10} must have been executed previously.

As second step, we project the weak/strong dependencies on data edges that interconnect activities along the control paths. A data edge that interconnects two activities, has two properties out of four that are strong source/strong target ($d^{ss,st}$), strong source/weak target ($d^{ss,wt}$), weak source/strong target ($d^{ws,st}$) and weak source/weak target ($d^{ws,wt}$). Formally, the property of a data edge is defined as below.

Definition 3 (Data edge property) *Let $d \in \mathcal{E}_d$ with $a_i = source(d)$ and $a_j = target(d)$.*

$$d = \begin{cases} d^{ss,st} & \Leftrightarrow \quad (a_i <_s a_j) \wedge (a_i >_s a_j) \\ d^{ss,wt} & \Leftrightarrow \quad (a_i <_w a_j) \wedge (a_i >_s a_j) \\ d^{ws,st} & \Leftrightarrow \quad (a_i <_s a_j) \wedge (a_i >_w a_j) \\ d^{ws,wt} & \Leftrightarrow \quad (a_i <_w a_j) \wedge (a_i >_w a_j) \end{cases}$$

The semantics of these properties can be explained as follows: Let's take $d_2^{ws,st}$ with $source(d_2)$=a_5 and $target(d_2)$=a_{16}. This means that if a_5 is executed, its output is used as one of the inputs of a_{16}. Otherwise, the relevant input of a_{16} is not initialized. Similarly, for $d_1^{ss,wt}$, if a_4 is executed, its input is initialized with the output of a_0. Otherwise, the output of a_0 is not used. The last step of our formal consideration is the generalization of the incoming and outgoing dependencies of an activity. We associate two sets to each activity in order to gather the source of incoming and the target of outgoing edges.

Definition 4 (Preset) *The preset of an activity a is denoted $\bullet a$. $\bullet a$ = {$(a_j, e) \in \mathcal{A} \times (\mathcal{E}_c \cup \mathcal{E}_d)$ s.t. $source(e) = a_j \wedge target(e) = a_i$}.*

The preset activities of an activity a_i consists of activities that have outgoing control/data edges that have a_i as the target. When activities are gathered in presets and postsets, we denote them as tuples with the relevant data or control edge. For example, $\bullet a_{16}$={$(a_3, d_3^{ws,st})$, $(a_5, d_2^{ws,st})$, $(a_{10}, d_5^{ss,st})$, $(a_{11}, d_4^{ss,st})$, (a_{15}, c)}. Note that we consider control edge pairs that include control connectors as single control edges that interconnect the activities that precede and succeed the connectors.

Definition 5 (Postset) *The postset of an activity a is denoted $a\bullet$. $a\bullet$ = {$(a_j, e) \in \mathcal{A}\times(\mathcal{E}_c \cup \mathcal{E}_d)$ s.t. $source(e) = a_i \wedge target(e) = a_j$}*

The postset activities of an activity a_i consist of activities that have incoming control/data dependencies that have a_i as source. For example, $a_{11}\bullet$={$(a_{16}, d_4^{ss,st})$, (a_{12}, c)}. For notational purposes, we use the activity identities to identify preset and postset elements. For example, for $(a_{16}, d_4^{ss,st}) \in a_{11}\bullet$, we denote $a_{16} \in a_{11}\bullet$. The elements of preset and postset are gathered into subsets with respect to the properties of their edges. $\bullet a_{i,ws}$, $\bullet a_{i,ss}$, $\bullet a_{i,wt}$, $\bullet a_{i,st}$, $a_{i,c} \subset \bullet a_i$ denote respectively the elements of $\bullet a_i$ that consist of data edges with weak/strong sources, weak/strong targets and control dependent activities. For example, $\bullet a_{16,ss}$={$(a_{10}, d_5^{ss,st})$, $(a_{11}, d_4^{ss,st})$} and $\bullet a_{16,c}$={(a_{15}, c)}. For postset elements, they can be noted such as $a_{11,c}\bullet$={(a_{12}, c)} and $a_{11,ws}\bullet = \emptyset$.

4 Decentralized Orchestration

4.1 Principles of Decentralized Orchestration

Peer processes can be derived by employing various data structures and algorithms. Our method associates the activities of the centralized specification to the peer processes of the services that they refer to. So, the elements of $\mathcal{A}_{s_i}$ are included in the peer process P_{s_i} that is executed by s_i. In order to preserve the semantics of the centralized specification with P2P interactions, the activities dispatched in different peer processes must be wired. Figure 4 depicts a simple wiring example of two control dependent activities across their corresponding peer processes. a_1 and a_2 consist of the respective invocation of s_1 and s_2. In a decentralized orchestration, a_1 and a_2 are included respectively in P_{s_1} and P_{s_2} that s_1 and s_2 execute. When P_{s_1} executes a_1, it must inform P_{s_2} about the termination of a_1. Thus, P_{s_2} can execute a_2. Consequently, the control dependency of the centralized specification is preserved in a decentralized implementation. In P_{s_1}, the message that must be sent to s_2 is denoted $a_1(t)$ while the activity that sends this message is denoted $a^w_{1\to 2}(a_1(t))$ (superscript is used for *writing*). In fact, $a_1(t)$ characterizes the completion state of a_1 with a "true" message. The sent message is received with the execution of activity $a^r_{1\to 2}(a_1(t))$ in P_{s_2} (superscript is used for *reading*). This intuitive peer process deriving approach works just fine for stateless and simple control interactions. However, the complexity of peer process structuring

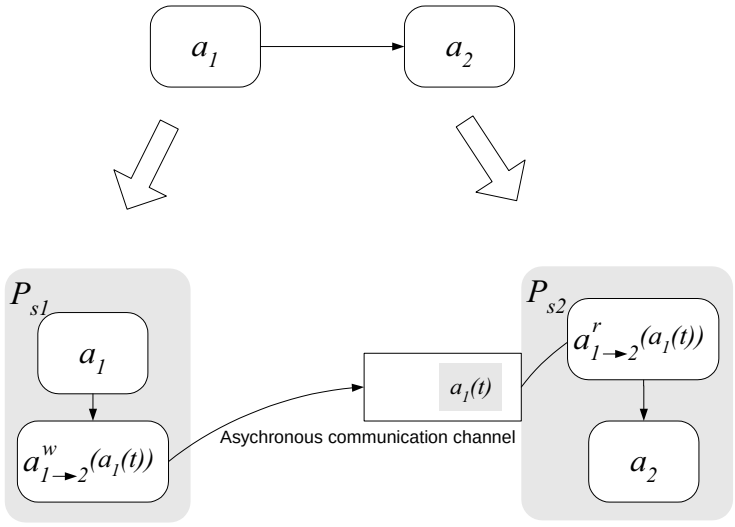

Figure 4: Wiring example

inevitably arises in more sophisticated processes. For example, let's suppose that a_1 and a_2 are interconnected by a data edge d_k and $a_1 >_w a_2$ (cf. figure 5). In P_{s_1}, a_1 may not be executed. In this case, $a^r_{1\to 2}(d_k)$ blocks P_{s_2} as it will never read a data sent by P_{s_1}. In order to prevent these kinds of blockage, $a^r_{1\to 2}(d_k)$ activity must be skipped in P_{s_2} if it is sure that the expected data will not be received. Similarly, if two activities

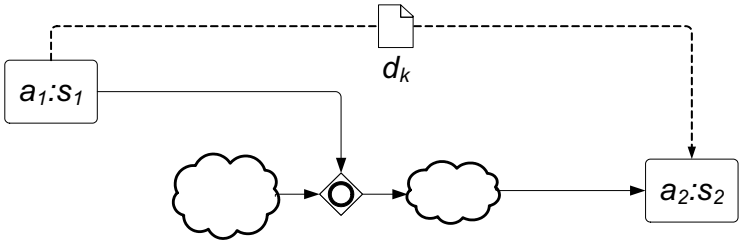

Figure 5: Data dependency along a weak control path

a_1 and a_2 are interconnected by a data edge d_k and $a_1 <_w a_2$, the output data of a_1 may not be used as the input of a_2 if the control path that includes a_2 is not executed (cf. figure 6). When P_{s_1} that executes a_1 sends d_k to P_{s_2} that executes a_2, d_k may not be consumed and may remain in the communication channel. Consequently, P_{s_1} cannot

terminate properly as it disposes an unconsumed data in its communication channel. So, the structuring of peer processes must deal with the particular consequences of different control/data dependencies that can violate the global soundness. More generally, when a data edge with $d^{ws,wt}$ or $d^{ss,wt}$ properties is wired across two peer processes, the sent data may not be consumed if the receiving peer process does not execute the target activity.

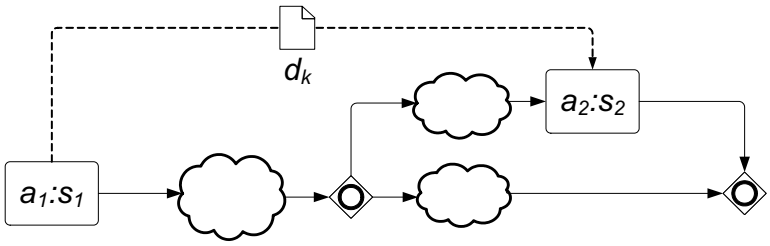

Figure 6: Data dependency along a weak control path

4.2 Wiring Activities Across Peer Processes and DPE

In order to deal with the above situations that can violate the global soundness, we propose a peer process structuring mechanism that propagates the states of activities that are not executed to relevant peer processes to allow to skip activities that can block them. To formalize interactions of peer processes, we make the following considerations: For each activity that precedes an OR-split point, some of outgoing branches can evaluate to "false". At this point, the DPE procedure is started to allow the activation of the activity that follows the corresponding OR-join. Thus, false transition values are propagated along the paths that are not executed. However, DPE does not influence activities that have incoming data edges with their sources on paths taken by DPE. We propose to extend the DPE mechanism toward such activities. Consequently, they can be informed about the state of activities that are at the source of data edges. The activities that ensure the wiring of process activities across different peer processes and their content are noted as follows: The "true" and "false" values corresponding to the state of an activity a_i are denoted respectively $a_i(t)$ and $a_i(f)$. As we mentioned above, an activity that sends (resp. receives) a process data or an activity state to (resp. from) another peer process is characterized with the corresponding identities of source and target activities such as $a_{i\rightarrow j}$ that come before (resp. after) their execution. Receiving activities have ^{r} as superscript while sending activities have ^{w}. The information carried by the latter is denoted such as $a^w_{i\rightarrow j}(a_i(t))$, $a^r_{i\rightarrow j}(a_i(t))$, $a^w_{i\rightarrow j}(a_z(t))$, or simply $a^w_{i\rightarrow j}(d_i)$ if it carries a process data d_i.

Example 2 (Dead path elimination and data dependencies) *Figure 7 depicts the motivating example. Activity a_0 precedes an OR-split point that DPE procedure can be triggered for outgoing paths. If the path that goes through a_1 evaluates to false, the activities between a_0 and a_{10} are not executed as this path is dead. This means that the output of a_5 is not used as one of the inputs of a_{10}. As these activities are executed*

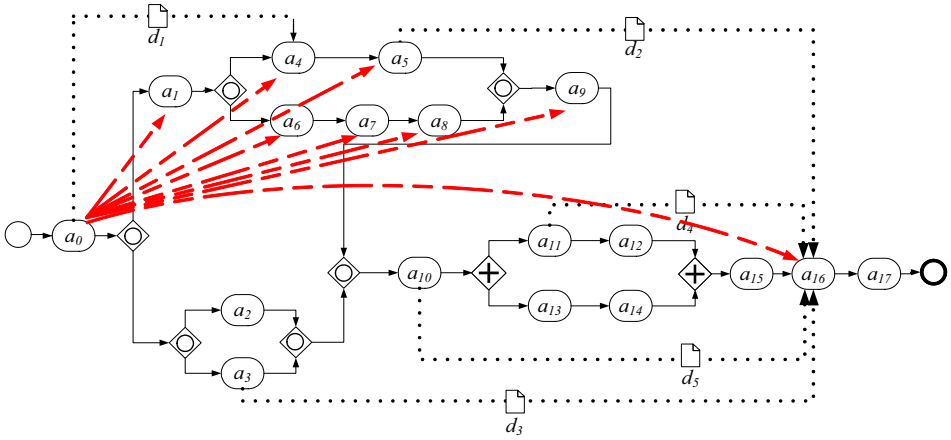

Figure 7: DPE example

within different peer processes, they must be wired together to implement underlying dependencies. The peer process that executes a_{16}, must execute an activity which receives d_2 from the peer process that is expected to execute a_5. However, if a_5 is not executed, the receiving activity of a_{16} blocks as it never reads any data. Naturally, the peer process of a_{16} must be informed about the state of a_5 whether it is executed or not. The question is which peer process informs the peer process of a_{16}.

The peer process that executes a_0 can evaluate the path that goes across a_1 to "false". At this point, the peer processes that expects the output of a_5 can be informed. Moreover, the peer processes including other activities that are on this dead path, must be informed in order to let their peer processes to skip these activities. As the peer process of a_0 evaluates this path, it can inform all of the concerned activities. In figure 7, the red arrows express these dependencies. However, if the transition condition of a_0 and a_1 evaluates to "true", the peer process of a_0 informs only the peer process of a_1 about the evaluation of the transition dependency. Thus, a_1 is executed in its peer process.

Example 3 (Wiring Activities Across Peer Processes and DPE) *Figure 8 shows the same example with the transitions of (a_0 and a_2) and (a_0 and a_3) that evaluate to "false".*

As one of the inputs of a_{16} is initiated by a_3, the peer process of a_{16} must be informed about the state of a_3 which is on a dead path.

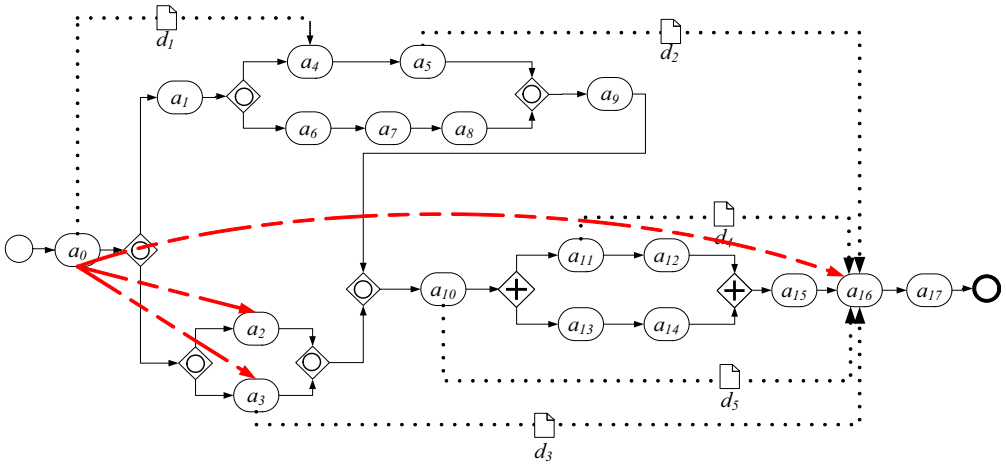

Figure 8: DPE example

As the peer process that evaluates transition condition sends relevant messages to the peer processes that must be informed about dead paths, these peer processes must include activities that execute collect the message that can be sent by other peer processes. If we consider that the centralized specification can include nested OR-split and OR-join points, the DPE can be started by different peer processes that execute the activities that precede OR-split points. Consequently, the peer processes that receive DPE information must include relevant activities that can receive these message from different peer processes.

Example 4 (Wiring Activities Across Peer Processes and DPE) *Figure 9 depicts the same motivating example. One of the inputs of a_{16} is initiated with the output of a_5. Activity a_5 is preceded by two OR-split points. Thus, DPE that takes a_5 can be triggered by two different peer processes that execute a_0 or a_1. Consequently, the peer process that executes a_{16} can be informed these peer processes. The green arrow characterize these dependencies. It should be noted that the receiving activities are exclusively structured as DPE is triggered once.*

Not surprisingly, the challenge of DPE elimination in decentralized process executions consists of structuring peer processes in a such way that they deal with the correct exchange of messages with each other. In order to do so, we examine peer process

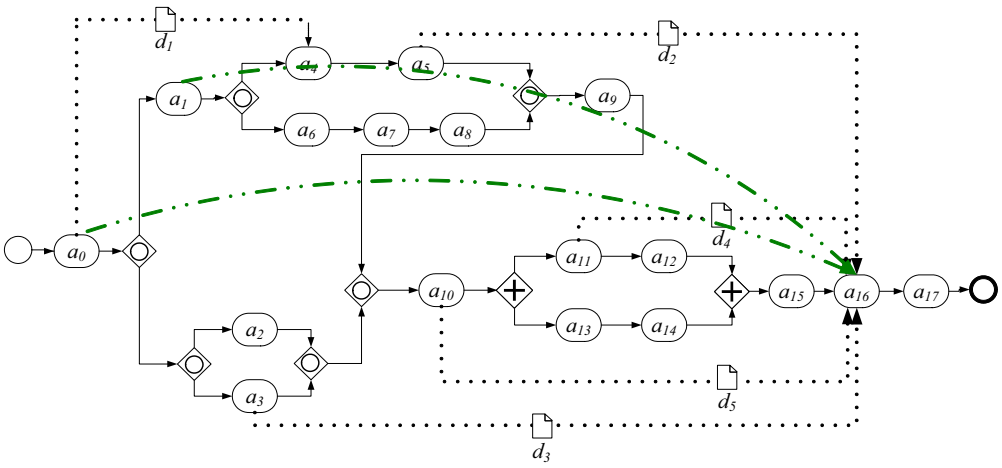

Figure 9: DPE example

deriving and structuring within two subsections. The first consists of structuring activities that send messages to other processes. The second consists of structuring activities that collect messages sent by other processes. The structuring mechanism computes each activity a_i of the centralized specification and derives relevant process fragments that must be executed before and after a_i. The latter are denoted $\bullet\widetilde{a_i}$ and $\widetilde{a_i}\bullet$.

4.3 Wiring with Postset Elements

Here, we describe how a peer process behaves after the execution of one of its activities to wire it with its dependent activities. The structure $\widetilde{a_i}\bullet$ describes a process fragment that the peer process executes following the execution of its activity a_i. a_i can be an activity that precedes a OR-split point or not. If so, it is the starting point of DPE. In our approach to decentralized orchestration, DPE is achieved with P2P interactions. The peer process that executes a_i informs directly the corresponding peer processes of activities that are on the paths taken by DPE. In the same time, the status of activities that are on the paths taken by DPE are propagated to peer processes that expect their outcome. If a_i is not an OR-split activity and its outgoing control edges evaluates to true, its peer process sends "true" control messages to the peer processes of the control dependent postset activities.

Algorithm 1 defines the part of the structuring operation of $\widetilde{a_i}\bullet$. In the notations, $\oplus$, $||$, $+$ denote respectively exclusive, concurrent and sequential append operations to a process fragment.

Algorithm 1 Wiring with postset activities

Require: a_i, *Centralized Process*

Ensure: $\widetilde{a_i\bullet}$

1: **for all** $a_j \in a_{i,c}\bullet$ **do**
2: **for all** $a_n \in \mathcal{A} - \{\widehat{a_i}\}$ s.t. $(a_n < \widehat{a_i}) \wedge (a_j < a_n)$ **do**
3: **for all** $a_k \in \bullet a_{n,c}$ **do**
4: $a_j^{\widetilde{\widetilde{\widetilde{\text{OR/false}}}}} \overset{||}{\leftarrow} a^w_{i\rightarrow n}(a_k(f))$
5: **end for**
6: $a_j^{\widetilde{\text{OR/false}}} \overset{||}{\leftarrow} a_j^{\widetilde{\widetilde{\widetilde{\text{OR/false}}}}}$
7: *empty*$(a_j^{\widetilde{\widetilde{\widetilde{\text{OR/false}}}}})$
8: **end for**
9: **for all** $a_n \in \mathcal{A}$ s.t. $\exists a_k \in \bullet a_{n,ws}, a_j < a_k \wedge a_k < \widehat{a_i}$ **do**
10: **for all** $a_k \in \bullet a_{n,ws}$ s.t. $(a_j < a_k) \wedge (a_k < \widehat{a_i})$ **do**
11: $a_j^{\widetilde{\text{OR/false}}} \overset{||}{\leftarrow} a^w_{i\rightarrow n}(a_k(f))$
12: **end for**
13: **end for**
14: **for all** $a_n \in \mathcal{A}$ s.t. $\exists a_k \in \bullet a_{n,ws}, \nexists cc \in \{\text{OR-split}\}, (a_j < a_k) \wedge (a_k < \widehat{a_i}) \wedge (cc < a_k) \wedge (a_i < cc) \wedge (a_k < \widehat{cc})$ **do**
15: $a_j^{\widetilde{\text{OR/true}}} \overset{||}{\leftarrow} a^w_{i\rightarrow n}(a_k(t))$
16: **end for**
17: $\widetilde{a_i\bullet} \overset{||}{\leftarrow} (a^w_{i\rightarrow j}(a_i(f)) + a_j^{\widetilde{\text{OR/false}}}) \oplus (a^w_{i\rightarrow j}(a_i(t)) + a_j^{\widetilde{\text{OR/true}}})$
18: **end for**
19: **for all** $a_j \in a_{i,st}\bullet \cup a_{i,wt}\bullet$ **do**
20: $\widetilde{a_i\bullet} \overset{||}{\leftarrow} a^w_{i\rightarrow j}(d = a_j.edge)$
21: **end for**
22: $\widetilde{a_i\bullet} \overset{+}{\leftarrow} {}^{\mathsf{F}} a_i$

Example 5 (Notation example) *For example, when* $\widehat{a_i} \overset{||}{\leftarrow} \widehat{\bullet a_n} \oplus a_m$ *is denoted,* a_n *and* a_m *are exclusively structured and added concurrently to* $\widehat{a_i}$ *with existing ones. Figure 10 depicts this notation.*

The algorithm operates on each activity of the centralized specification. It should be noted that $\widetilde{a_i\bullet}$ is executed following the execution of a_i. If a_i is skipped, because it is on a path taken by DPE, the activities of $\widetilde{a_i\bullet}$ are skipped in the same peer process. Informally, the algorithm can be described as follows: For each control dependent activity a_j that follows a_i, two fragments that correspond to "true" and "false" transition conditions of a_j are derived. These are characterized as $a_j^{\widetilde{\text{OR/false}}}$(line 2-8, and line 9-13)

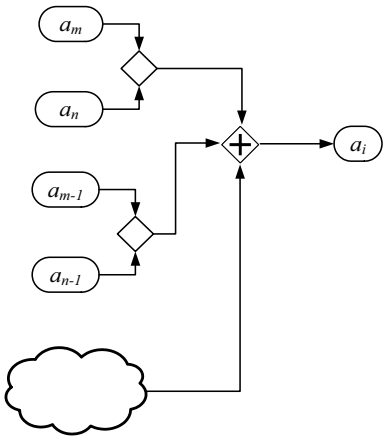

Figure 10: Structuring of preset activities: Notation example

and $\widetilde{a_j^{\text{OR/true}}}$(line 14-16). $\widetilde{a_j^{\text{OR/false}}}$ include activities that send "false" messages to peer processes that execute activities that are on the paths taken by DPE (line 4). Moreover, activities that send "false" messages to peer processes that have incoming data edges from the activities of DPE are included in $\widetilde{a_j^{\text{OR/false}}}$ (line 11). $\widetilde{a_j^{\text{OR/true}}}$ includes activities that send "true" messages to peer processes that have incoming data edges from the activities that are not taken by DPE (line 15). Note that, we take nested OR-split/OR-join fragments into account. These two fragments are structured sequentially after the corresponding "true" and "false" control messages that can be sent to a_j. The latter are exclusively structured (line 17). The last step of $\widetilde{a_i\bullet}$ is sending the outputs of a_i to peer processes that expect them (line 20). An unique activity ${}^{\mathsf{F}}a_i$ characterizes an empty activity added at the end of $\widetilde{a_i\bullet}$. It is wired with the empty start activities of other activities $a \in \mathcal{A}_{s_i}$ that follow a_i with respect to their partial order. This operation is detailed in the next subsection.

Example 6 (Wiring with postset activities) *Let's suppose that activities a_0 and a_{11} depicted in figure 11 consist of the invocations of different service s_1's operations. In P_{s_1}, after the execution of a_0, one of the paths that go to a_1 and a_2 can evaluate to "false". In P_{s_1}, a_0 is followed by two concurrent branches that are followed by two exclusive branches that correspond "false" and "true" control messages to be sent to peer processes that include a_1 and a_2. We can denote $\widetilde{a_0\bullet}$ =* $[(a^w_{0\to1}(a_0(t))$ ***+ 1)*** $\oplus$ $(a^w_{0\to1}(a_0(f))$ ***+ 2)***$]$ $||$ $[(a^w_{0\to2}(a_0(t))$ ***+ 3)*** $\oplus$ $(a^w_{0\to2}(a_0(f))$ ***+ 4)***$]+a^w_{0\to4}(d_1)+{}^{\mathsf{F}}a_0$. *The activities that send control messages to the peer processes that execute the elements of $a_{0,c}\bullet$ are followed either by activities that fulfill DPE or inform peer processes*

about the status of activities that are not on the paths taken by DPE. Thus, ***1*** *=* $\emptyset$*,* ***2*** *=* $[a^w_{0\to4}(a_1(f))||a^w_{0\to5}(a_4(f))||\ a^w_{0\to7}(a_1(f))||a^w_{0\to9}(a_5(f))\ ||a^w_{0\to9}(a_8(f))]$ *+* $[a^w_{0\to16}(a_5(f))]$*,* ***3*** *=* $[a^w_{0\to10}(a_2(t))]$*,* ***4*** *=* $[a^w_{0\to6}(a_2(f))\ ||\ a^w_{0\to3}(a_2(f))\ ||\ a^w_{0\to10}(a_3(f))\ ||\ a^w_{0\to10}(a_6(f))]$ *+* $[a^w_{0\to10}(a_2(f))\ ||\ a^w_{0\to16}(a_3(f))]$*. The activity* ${}^Fa_0$ *is connected to* ${}^Sa_{11}$*.* a_{11} *is the last element in* P_{s_1}*. If there were other* $a_i \in \mathcal{A}_{s_1}$ *that* $a_0<a_i$ *and* $a_i||a_{11}$*,* ${}^Fa_0$ *would be interconnected to their* ${}^Sa_i$*. For* a_{11}*,* $\widetilde{a_{11}\bullet}$ *can be denoted as follows:* $\widetilde{a_{11}\bullet} = a^w_{11\to12}(a_{11}(t)) + a^w_{11\to16}(d_4) + {}^Fa_{11}$*.*

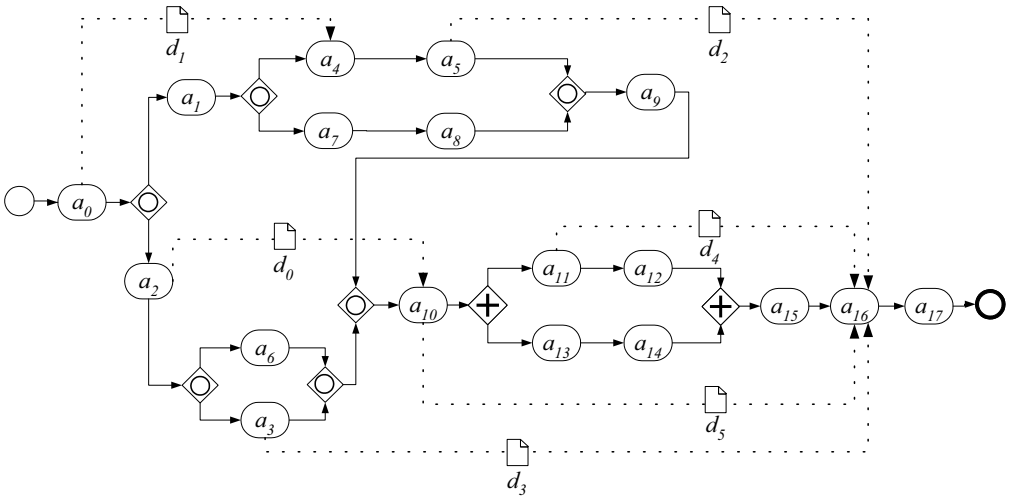

Figure 11: Wiring with postset activities

Example 7 (Structuring peer process) *Figure 12 depicts the messages that the peer processes that executes* a_0 *(*P_{s_1}*), must send the peer processes that execute the target activities pointed by red arrows if the transition of* a_0 *and* a_1 *evaluates to "false".*

In contrast to the above example, figure 13 depicts the example if the transition condition of a_0 and a_2 evaluates to "false". The peer processes that execute a_{10} and a_{16} receive messages as they have incoming data edges coming from the activities that are on the paths taken by DPE.

4.4 Wiring with Preset Elements

Wiring with preset activities consists of identifying the behavior of a peer process before executing or skipping an activity a_i. The structure $\bullet\widetilde{a_i}$ describes the corresponding

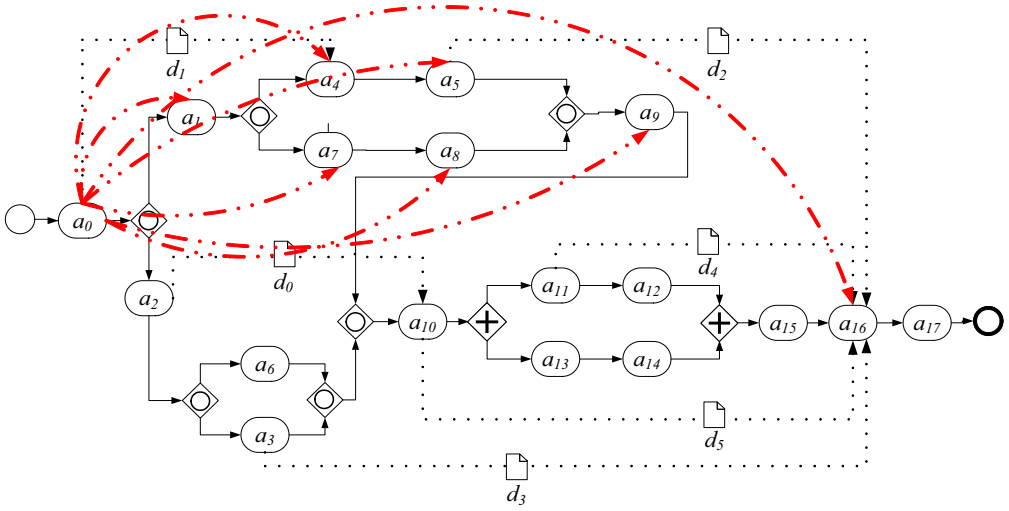

Figure 12: Transition of a_0 and a_1 that evaluates to "false"

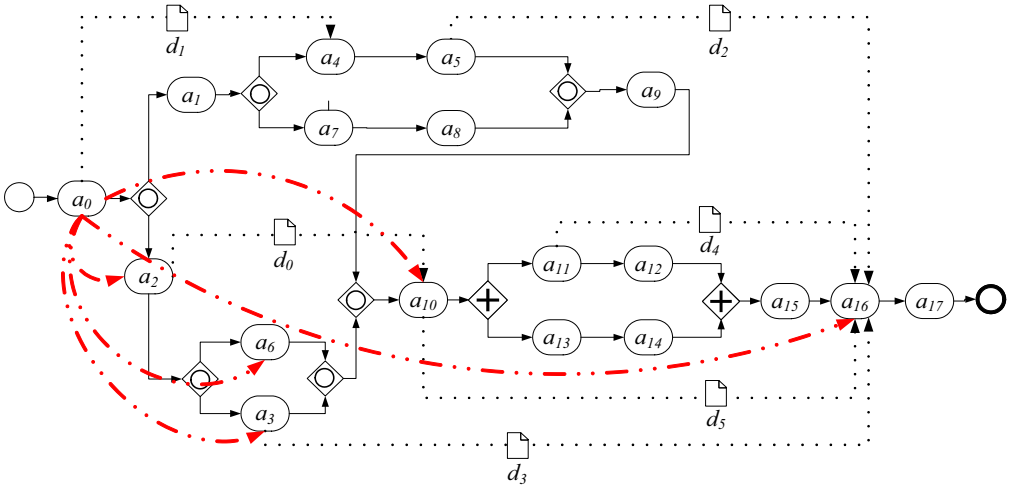

Figure 13: Transition of a_0 and a_2 that evaluates to "false"

process fragment to an activity a_i. First, before executing an activity a_i, its peer process must collect the control messages that correspond to the termination of control dependent preset activities of a_i. It must be noted that control messages about the termination of control dependent preset activities are always received. According to the content of the latter, a_i can be executed or skipped. If a_i is executed (because all of the received messages contain "true" values or one of them contain "true" for a_i that is a OR-join activity), the input values of a_i must be initialized with the outcome of executed preset activities. At this point, the peer process that intends to execute a_i, must execute the activities that collect the sent data. In order to do so, it must test the state of source activities. If the source of activities are executed, the activities that collect the relevant data can be executed. Algorithm 2 resumes the operations that structure the elements of $\bullet\widetilde{a_i}$. ${}^{\mathsf{S}}a_i$ is an empty activity placed at the beginning of $\bullet\widetilde{a_i}$ to connect the latter with previous elements of the same peer process. The algorithm can be explained as follows: If a_i is on paths that can be taken by DPE, the peer processes that execute activities that precede OR-split points can send the messages that contain "false" message for all elements of $\bullet a_{i,c}$ (line 2-8). If the immediate precedent activity is executed, the "true" message is received only from the peer process that executes it (line 6). The activities that collect status of control dependent preset activities are structured exclusively as there is a single peer process that can send the corresponding message (line 4 and line 6). According to the content of control messages, a_i can be executed or skipped. a_i^+ and a_i^- characterize the beginnings of two exclusive branches that respectively execute or skip a_i (line 19). a_i^+ is followed with the execution of fragment $\widetilde{a_i^+}$. $\widetilde{a_i^+}$ includes activities that collect "false" or "true" messages that can come from the peer processes that can start DPE that the activities of $\bullet a_{i,ws}$ are on (line 9-18). The activities that collect "true" messages are followed by activities that read the sent data (line 14). The last step of $\bullet\widetilde{a_i}$ is the collection of data that are sent by peer processes that execute the elements of $\bullet a_{i,ss}$ (line 20-22). It should be noted that the latter do not require the execution of activities that test the status of source activities. a_i^- that corresponds to the start of the exclusive branch that is executed if a_i is taken by DPE, is interconnected to ${}^{\mathsf{F}}a_i$. Consequently, this operation enables the following activities of the same process with respect to their partial order.

Example 8 (Wiring with Preset Activities) *Let's suppose that a_4 and a_{16} depicted in figure 14 consist of the invocations of different s_2's operations. In P_{s_2}, a_4 is preceded by two OR-split points which means that its path can evaluate to "false" at these two different points. Consequently, the peer processes that execute a_0 or a_1 can send "false" control message for the control dependent preset activities of a_4. If the path of a_4 is not taken by DPE, the peer process of a_1 sends "true" control message to P_{s_2}. After the reception of control messages, a_4 can be executed or skipped. If it is executed, its inputs must be collected from peer processes that execute source activities. Thus, a_4^+ is followed with the execution of an activity that receives the data sent by the peer process*

Algorithm 2 Wiring with preset activities

Require: a_i, *Centralized Process*
Ensure: $\widetilde{\bullet a_i}$

1: $\widetilde{\bullet a_i} \overset{+}{\leftarrow}{}^{\mathsf{S}} a_i$
2: **for all** $a_j \in \bullet a_{i,c}$ **do**
3: **for all** $a_k \in \mathcal{A}$ s.t. $\exists cc \in \{\text{OR-split}\}, c \in \mathcal{E}_c, c = (a_k, cc) \wedge a_i < \widehat{cc}\}$ **do**
4: $\widetilde{\bullet a_i^{\mathsf{OR}}} \overset{\oplus}{\leftarrow} a^r_{k\rightarrow i}(a_j(f))$
5: **end for**
6: $\widetilde{\bullet a_i} \overset{||}{\leftarrow} a^r_{j\rightarrow i}(a_j(t)) \oplus \widetilde{\bullet a_i^{\mathsf{OR}}}$
7: $empty(\widetilde{\bullet a_i^{\mathsf{OR}}})$
8: **end for**
9: **for all** $a_j \in \bullet a_{i,ws}$ **do**
10: **for all** $a_m \in \mathcal{A}$ s.t. $\exists cc \in \{\text{OR-split}\}, c \in \mathcal{E}_c, c = (a_m, cc) \wedge (cc < a_j) \wedge (a_j < \widehat{cc})\}$ **do**
11: $\widetilde{a_j^{\mathsf{OR}}} \overset{\oplus}{\leftarrow} a^r_{m\rightarrow i}(a_j(f))$
12: **end for**
13: **for all** $a_n \in \mathcal{A}$ s.t. $\exists cc, \nexists cc' \in \{\text{OR-split}\}, c \in \mathcal{E}_c, c = (a_n, cc) \wedge (cc < a_j) \wedge (a_j < \widehat{cc}) \wedge (cc < cc') \wedge (cc' < a_j)\}$ **do**
14: $\widetilde{a_j^{\mathsf{OR}}} \overset{+}{\leftarrow} \widetilde{a_j^{\mathsf{OR}}} \oplus (a^r_{n\rightarrow i}(a_j(t)) + a^r_{j\rightarrow i}(d = a_j.edge))$
15: **end for**
16: $a_i^+ \overset{||}{\leftarrow} \widetilde{\bullet a_j^{\mathsf{OR}}}$
17: $empty(\widetilde{\bullet a_j^{\mathsf{OR}}})$
18: **end for**
19: $\widetilde{\bullet a_i} \overset{+}{\leftarrow} (a_i^+ + \widetilde{a_i^+}) \oplus (a_i^-)$
20: **for all** $a_j \in \bullet a_{i,ss}$ **do**
21: $\widetilde{\bullet a_i} \overset{||}{\leftarrow} a^r_{j\rightarrow i}(d = a_j.edge)$
22: **end for**

that executes a_0. *As* $\bullet a_{4,ws}$ *is empty, there is no need to test the source of incoming edges. We can denote* $\widetilde{\bullet a_4} = {}^{\mathsf{S}}a_4 + [a^r_{0\rightarrow 4}(a_1(f)) \oplus a^r_{1\rightarrow 4}(a_1(f)) \oplus a^r_{1\rightarrow 4}(a_1(t))] + [(a_4^+ + a^r_{04}(d_1)) \oplus a_4^-]$. *The activity* a_4^- *is interconnected to* ${}^{\mathsf{S}}a_{16}$. *For* a_{16}, *there are four preset activities (*a_3, a_5, a_{10}, a_{11}*) that provide inputs to* a_{16}. *Two of them may not be executed (*a_3, a_5*). Consequently, the "false" status of these activities can be sent to* P_{s_2} *by the peer processes that can start DPE. The "true" status message can be sent only by the peer process that executes the first OR-split point that precedes the source activity of data dependence. Activities* a_{10} *and* a_{11} *are strong source activities. Consequently, there is no necessity to test their status. The activities that collect* d_4 *and* d_5 *are executed before executing* a_{16}. *We can denote* $\widetilde{\bullet a_{16}}$ *as follows* $\widetilde{\bullet a_{16}} = {}^{\mathsf{S}}a_{16} + a^r_{15\rightarrow 16}(a_{15}(t)) + [(a^r_{0\rightarrow 16}(a_3(f)) \oplus a^r_{2\rightarrow 16}(a_3(f)) \oplus (a^r_{2\rightarrow 16}(a_3(t)) + a^r_{3\rightarrow 16}(d_3)))||(a^r_{0\rightarrow 16}(a_5(f)) \oplus a^r_{1\rightarrow 16}(a_5(f)) \oplus (a^r_{1\rightarrow 16}(a_5(t)) + a^r_{5\rightarrow 16}(d_2)))] + [a^r_{10\rightarrow 16}(d_5)||a^r_{11\rightarrow 16}(d_4)]$.

In figure 14, the green arrows depict the control messages that can be received by the peer process that executes a_4 (P_{s_2}) if it is on a path taken by DPE. If the DPE is started by the peer process of a_0, the former can send a control message to P_{s_2} to allow it to skip a_4. Similarly, the peer process that executes a_1 can evaluate the path of a_4 to "false", then the peer process of a_4 must receive a control message from the peer process of a_1 and skip a_4. If the path of a_4 is not taken by DPE, the peer process must wait for a control message to execute a_4. This message can be only sent by the peer process of a_1 as it is the control dependent activity that precede a_4 in the centralized specification.

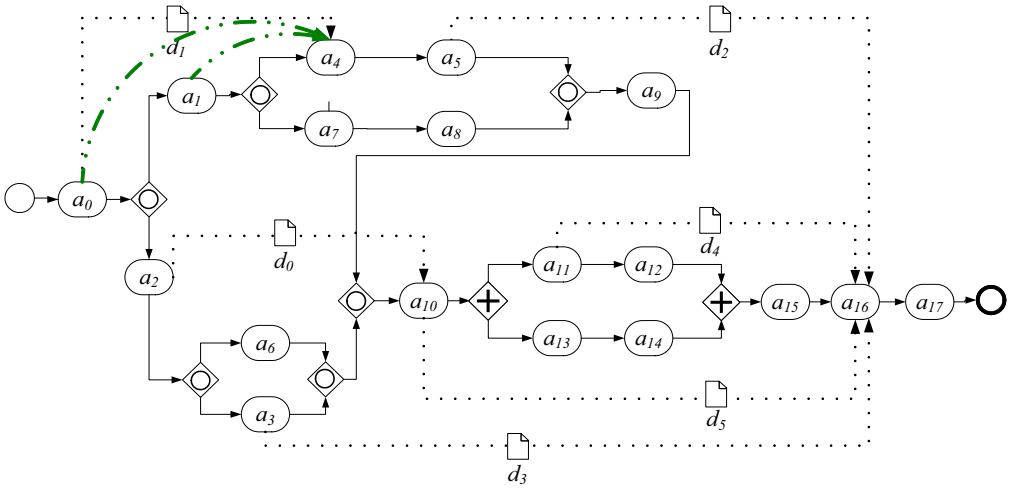

Figure 14: The messages that must be received for a_4 if it is on a path taken by DPE

Again, it should be noted that the activities that collect the messages sent by other peer processes are structured in a way that they do not subject blocking reads. For example, if an activity subject DPE that can be triggered by n different peer processes, the control messages that allow its process to skip this activity, are exclusively structured as there is only one peer process that can send the "false" control messages.

Example 9 (Wiring with preset activities) *In figure 15, the green arrows depict the control messages that must collected by the peer processes of a_{16} if the source of its incoming data edge d_3 is taken by DPE. If the transition condition of a_0 and a_2 evaluates to "false" then the path from a_0 to a_{10} is dead. Consequently, the peer process of a_{16} must be informed about the state of a_3 that will not be executed. Similarly, if the transition of a_0 and a_2 evaluates to "true", then the path that includes a_6 can be dead because of the transition of a_2 and a_6 that can evaluate to "false". Consequently, the*

peer process of a_2 must inform the peer process of a_{16} about the state of a_3. Thus, the peer process of a_{16} can skip the activity that collects d_3.

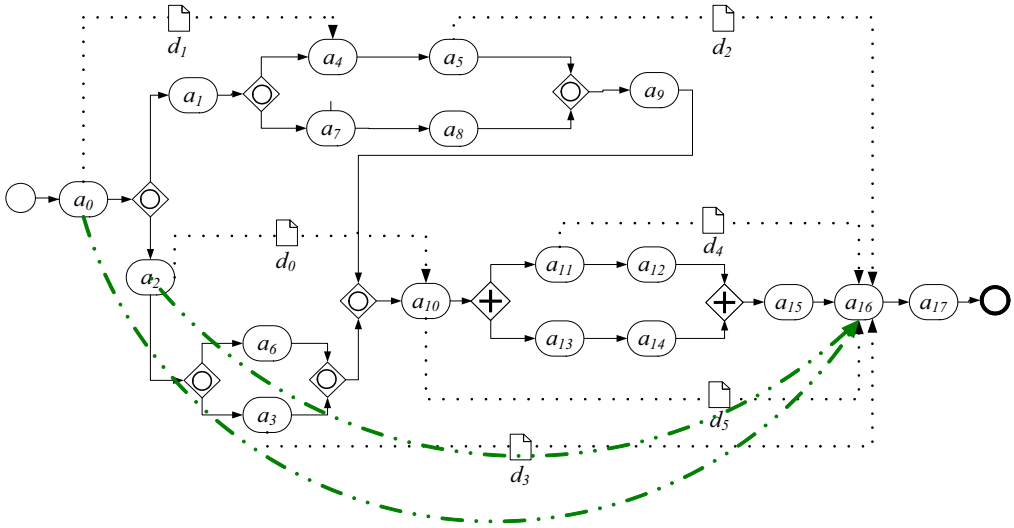

Figure 15: The messges that must be received for a_{16} if the source of incoming data dependencies are on paths taken by DPE

4.5 Structuring peer processes

If the centralized specification subjects the composition of conversational services, the relevant peer processes of that specification must deal with their partial order or exclusivity. The algorithms that we have presented in the previous sections deal with this issue. In this section, we give some concrete examples.

Example 10 (Structuring) *Let's suppose that the activities a_0 and a_{11} of figure 11 consist of the invocations of different operation of the same service. Consequently, they are included in the same peer process that that service will execute. With previous algorithms, the relevant process fragments can be derived. They are denoted $\widetilde{\bullet a_0}$, $\widetilde{\bullet a_{11}}$, $\widetilde{a_{11}\bullet}$ and $\widetilde{a_0\bullet}$. Each derived fragment begins with an empty activity ${}^{S}a_i$ that precedes $\widetilde{\bullet a_i}$ and ends with an empty activity ${}^{F}a_i$ that succeeds $\widetilde{a_i\bullet}$. When the peer process structured, the partial order of activities are taken into account. In the centralized specification, a_0 precedes a_{11}. Consequently, when the process is executed with peer processes, the same order must be preserved. In order to do so, ${}^{F}a_0$ is placed before ${}^{S}a_{11}$ in the peer process. Consequently, the activities that collect messages relevant to*

the activity a_{11} can be executed after the termination of the activities that send wiring information of a_0. Figure 16 depicts the structuring of a_0 and a_{11} within their peer process.

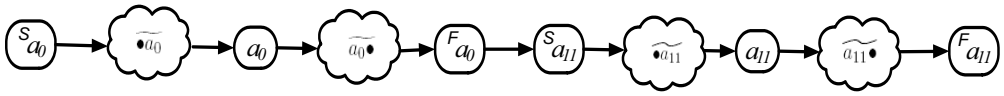

Figure 16: Structuring activities within their peer processes

Let's suppose that a_0, a_{11} and additionally a_{13} must be structured within the same peer process. In order to respect their partial order, ${}^{S}a_{11}$ and ${}^{S}a_{13}$ must succeed ${}^{F}a_0$. However, ${}^{S}a_{11}$ and ${}^{S}a_{13}$ must be concurrent as a_{11} and a_{13} are concurrent in the centralized specification.

In the above example, all of the activities of the peer process are executed as they are not paths that can be taken by dead paths. However, if structured activities are on paths that can be taken by DPE, their peer processes can be able to skip them. The following example describes a peer process where there are two activities that are on the paths that can be taken by DPE.

Example 11 (Structuring peer process) *Let's suppose the activities a_0, a_6 and a_{12} of figure 11 consist of the invocations of different operations that belong to a same service. Activity a_6 is on a path that can be taken by DPE. Consequently, the peer process must skip it if the former is not executed. The peer process of figure 17 depicts the structure of relevant process fragments of a_0, a_6 and a_{12}.*

4.6 On proper termination

he structuring mechanism that wires peer processes with each other prevents blocking reads. However, it does not guarantee the proper termination of peer processes that have unconsumed data in their asynchronous communication channels. In order to ensure the proper termination, we employ an explicit mechanism that sends termination messages to all peer processes after the termination of the final activity. In order to do so, the peer process that executes the final activity of the centralized specification sends a termination message to all peer processes to terminate current peer process instances that belongs to the same composition. Consequently, the peer processes that must terminate include a corresponding receiving activity that is concurrently placed before their final activity. Thus, whenever a termination message is received from the peer process that executes the final activity, they can remove their unconsumed data and terminate correctly.

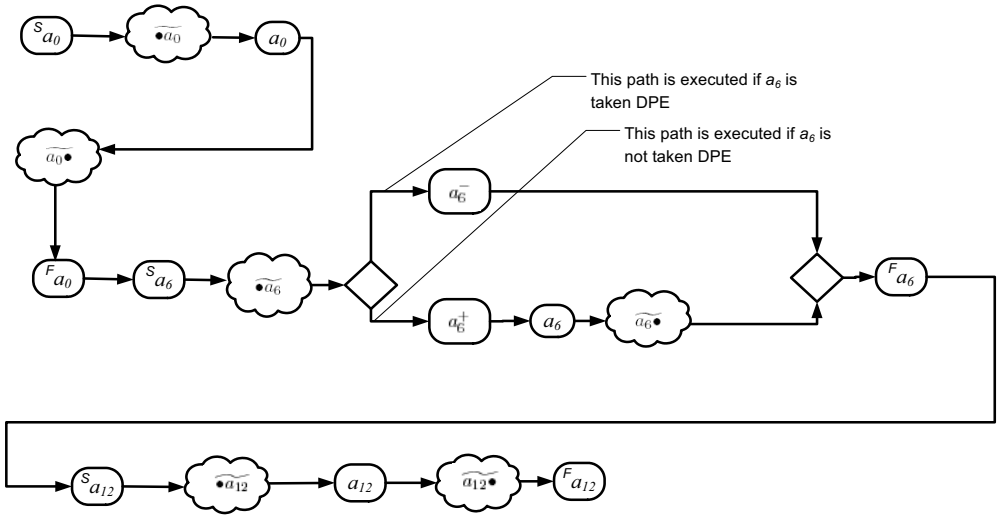

Figure 17: Structuring activities within their peer processes

5 Related work

Considerable amount of work toward decentralized process executions has been done in the context of decentralized workflow management [2][12][16][5][20]. Our approach differs from these propositions mainly because of its focus on the decentralized process and because our approach does not assume the presence of sophisticated software layers on orchestrated services.

Our very related work can, in some sense, be considered as process partitioning. Recently, a number of partitioning methods have been proposed for enabling decentralized execution settings. However, they are in their infancy. Baresi et al. [4] and Sadiq et al. [15] consider the decentralization operation from the point of view of control flow. Although, in [10] Khalaf and Leymann have addressed the partitioning that compute data edges, they do not present the concrete specification of such an operation. Moreover, their partitioning does not aim to establish P2P interconnections. In [14], Nanda et al. propose a partitioning mechanism for centralized BPEL specifications. However, the authors do not detail how they deal with DPE and conversational aspects. They aim to optimize the interactions of distributed processes without considering their P2P interactions. [3] presents a run-time partitioning mechanism that aims to exchange the process between its participants for P2P interconnections. However, the authors assume the presence of a partitioning mechanism implemented by process participants.

Our contribution stands between *Orchestration* and *Choreography* initiatives that govern the implementation of service oriented processes. In contrast to orchestration, choreography (championed by WS-CDL[8] which is used in conjunction with BPEL) describes the public messages exchanges and interaction rules that occur between multiple service endpoints. Choreography is more P2P in nature than orchestration. The most common use of WS-CDL is the derivation of distributed BPEL stubs from a globally agreed WS-CDL description. However, the agreement of a common choreography and the derivation of BPEL stubs are not implicitly defined. Our approach considers a centralized specification that does not require a global agreement between services and naturally, it enforces the global semantics on local implementations [26]. To sum up, we take advantage of the process standardization efforts in another context.

6 Discussion

In this report, we have presented an approach that enables decentralized service orchestrations where services can establish P2P interconnections. Our approach deals with decentralization by returning to the need to derive distributed and cooperating *peer processes* of a centralized specification. We have described a set of formal concepts and algorithms that compute a centralized specification to derive its corresponding peer processes. We focused on sophisticated control/data dependencies and conversational aspects of decentralization that need reconciliation but have not been extensively studied in the relevant literature. Perhaps the most interesting aspect of our work is to benefit from the interchangeability issue of processes that process standardization efforts have brought forward. This in turn strongly argues that the services are capable of receiving and executing peer processes. Although our approach demonstrates how a process can be decentralized, there are many criteria that govern decentralization operation [24][25][23]. We are currently exploring ways to integrate these criteria to our existing contributions. Considerable work of optimization remains on the issues presented in this paper. We intent to implement a more appropriate interaction mechanism that reduces the interactions of peer processes as much as possible while it keeps P2P nature.

References

[1] Organization for the advancement of structured information standards (oasis). http://www.oasis-open.org.

[2] Gustavo Alonso, Roger Günthör, Mohan Kamath, Divyakant Agrawal, Amr El Abbadi, and C. Mohan. Exotica/fmdc: A workflow management system for mobile and disconnected clients. *Distributed and Parallel Databases*, 4(3):229–247, 1996.

[3] Vijayalakshmi Atluri, Soon Ae Chun, and Pietro Mazzoleni. A chinese wall security model for decentralized workflow systems. In *ACM Conference on Computer and Communications Security*, pages 48–57, 2001.

[4] Luciano Baresi, Andrea Maurino, and Stefano Modafferi. Workflow partitioning in mobile information systems. In *Mobile Information Systems, IFIP TC 8 Working Conference on Mobile Information Systems, MOBIS*, pages 93–106, 2004.

[5] Boualem Benatallah, Quan Z. Sheng, Anne H. H. Ngu, and Marlon Dumas. Declarative composition and peer-to-peer provisioning of dynamic web services. In *Proceedings of the* 18^{th} *International Conference on Data Engineering, ICDE*, pages 297–308, 2002.

[6] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing distributed applications with mobile code paradigms. In *Proceedings of the* 19^{th} *International Conference on Software Engineering, ICSE*, pages 22–32. ACM Press, 1997.

[7] Fabio Casati, Ski Ilnicki, Li jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in *flow*. In *the* 12^{th} *International Conference Advanced Information Systems Engineering, CAiSE*, pages 13–31, 2000.

[8] Nickolas Kavantzas et al. *Web Services Choreography Description Language Version 1.0*, 2005.

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

[10] Rania Khalaf and Frank Leymann. Role-based decomposition of business processes using bpel. In *International Conference of Web Services, ICWS*, pages 770–780. IEEE Computer Society, 2006.

[11] Bartek Kiepuszewski, Arthur H. M. ter Hofstede, and Christoph Bussler. On structured workflow modelling. In *Proceedings* 12^{th} *International Conference of Advanced Information Systems Engineering, CAiSE*, pages 431–445, 2000.

[12] Neila Ben Lakhal, Takashi Kobayashi, and Haruo Yokota. Throws: An architecture for highly available distributed execution of web services compositions. In 14^{th} *International Workshop on Research Issues in Data Engineering , RIDE*, pages 103–110, 2004.

[13] Frank Leymann and Dieter Roller. *Production Workflow - Concepts and Techniques*. PTR Prentice Hall, 2000.

[14] Mangala Gowri Nanda, Satish Chandra, and Vivek Sarkar. Decentralizing execution of composite web services. In *Proceedings of the* 19^{th} *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 170–187, 2004.

[15] Wasim Sadiq, Shazia Sadiq, and Karsten Schulz. Model driven distribution of collaborative business processes. In *IEEE International Conference on Services Computing, SCC*, pages 281–284, 2006.

[16] Christoph Schuler, Can Turker, Hans-Jörg Schek, Roger Weber, and Heiko Schuldt. Peer-to-peer execution of (transactional) processes. *International Journal of Cooperative Information Systems*, 14(4):377–405, 2005.

[17] Wil M. P. van der Aalst. Interorganizational workflows: An approach based on message sequence charts and petri nets. *Systems Analysis - Modelling - Simulation*, 3(34):335–367, 1999.

[18] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[19] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Dan Ferguson. *Web Services Platform Architecture*. Prentice Hall, 2005.

[20] Dirk Wodtke, Jeanine Weißenfels, Gerhard Weikum, Angelika Kotz Dittrich, and Peter Muth. The mentor workbench for enterprise-wide workflow management. In *SIGMOD Conference*, pages 576–579, 1997.

[21] Andreas Wombacher. Decentralized consistency checking in cross-organizational workflows. In *The 8th IEEE International Conference on and Enterprise Computing, E-Commerce, and E-Services, CEC/EEE*. IEEE Computer Society, 2006.

[22] Jun Yan, Yun Yang, and G.K. Raikundalia. Swindew - a peer-to-peer based decentralised workflow management system. *IEEE Transactions on Systems, Man and Cybernetics*, page to be published, 2006.

[23] Ustun Yildiz and Claude Godart. Dynamic decentralized service orchestration. In *Proceedings of the* 3^{th} *International Conference on Web Information Systems and Technologies, WEBIST*. INSTICC Press, 2007.

[24] Ustun Yildiz and Claude Godart. Enhancing secured service interoperability with decentralized orchestration. In *Proceedings of the* 23^{nd} *International Conference on Data Engineering Workshops, ICDE Workshops (to appear)*. IEEE Computer Society, 2007.

[25] Ustun Yildiz and Claude Godart. Towards decentralized service orchestrations. In *The 22nd Annual ACM Symposium on Applied Computing, SAC,* pages 1662–1666, 2007.

[26] Johannes Maria Zaha, Alistair P. Barros, Marlon Dumas, and Arthur H. M. ter Hofstede. Let's dance: A language for service behavior modeling. In *OTM Conferences (1),* pages 145–162, 2006.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399