A centralized and a decentralized method to automatically derive choreography-conforming web service systems[☆]

Ismael Rodríguez[a], Gregorio Díaz[b,*], Pablo Rabanal[a], Jose Antonio Mateo[c]

[a]*Dept. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 28040, Madrid, Spain.*
[b]*Universidad de Castilla-La Mancha, Escuela Superior de Ingeniera Informática, 02071, Albacete, Spain.*
[c]*Universidad de Castilla-La Mancha, Instituto de Investigación en Informática, 02071, Albacete, Spain.*

Abstract

We present a formal model to represent orchestrations and choreographies and we define several *conformance* semantic relations allowing to detect whether a set of orchestration models, representing some web services, leads to the overall communications described in a choreography. Given this formal model, we develop automatic methods to derive a set of web services from a given choreography, in such a way that the system consisting of these services necessarily conforms to the choreography. These methods enable the construction of conforming systems of services even in cases where projecting the choreography into each service would lead to a non-conforming system. This issue is addressed by adding some control messages that make services interact as required by the choreography. Two different derivation methods are presented. In the *centralized* method, a new service is responsible of managing these additional control messages. In the *decentralized* method, the responsibility of handling these messages is distributed among all services.

Keywords: Automatic Web service composition, formal specification, web service choreography.

1. Introduction

Web services related technologies are a set of middleware technologies for supporting *Service-Oriented Computing* [23]. The definition of a web service-oriented system involves two complementary views: *Orchestration* and *choreography*. The orchestration concerns the *internal* behavior of a web service in terms of invocations to other services. It is supported, e.g., by WS-BPEL [2] (*Web Services Business Process Execution Language*), which is a language for describing the web service behavior (workflow) in terms of the composition of other web services. On the other hand, the choreography concerns the *observable* interaction among web services. It can be defined, e.g., by using WS-CDL [39] (*Web Services Choreography Description Language*). Roughly speaking, the relation between orchestration and choreography can be

[☆]Research partially supported by projects TIN2009-14312-C02-01 and PEII09-0232-7745.

[*]Corresponding author

Email addresses: isrodrig@sip.ucm.es (Ismael Rodríguez), gregorio.diaz@uclm.es (Gregorio Díaz), prabanal@fdi.ucm.es (Pablo Rabanal), jmateo@dsi.uclm.es (Jose Antonio Mateo)

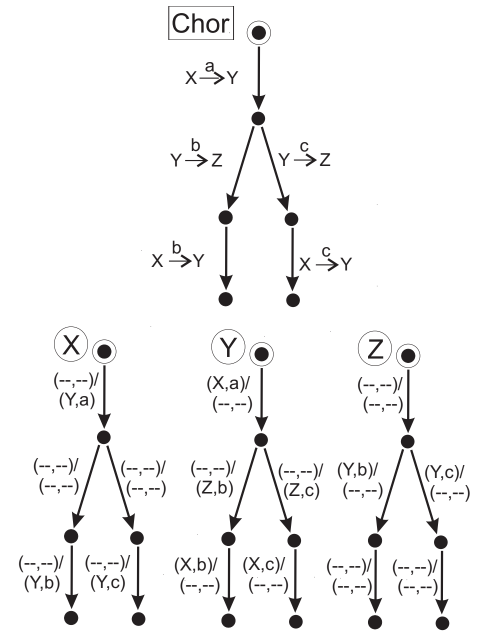

Figure 1: Example of natural projection.

stated as follows: The collaborative behavior, described by the choreography, should be the result of the interaction of the individual behaviors of each involved party, which are defined via the orchestration. Let us note that the communication among services is asynchronous. However, choreographies do not capture the type of communication, but only the communication flow.

In this paper we present some formal frameworks to automatically derive web services (in particular, their orchestration definition) from a given choreography, in such a way that the concurrent behavior of these derived services necessarily *conforms* to the choreography.

The main problem arisen in the derivation of services from a choreography is the fact that the *natural projection* does not necessarily produce a set of services conforming this choreography. We can easily observe this problem in the example depicted in Figure 1. The formalism used here will be fully defined in Section 2, but we can familiarize ourselves with the notation now. The system on the top, $Chor$, represents a choreography. It defines the required communication flow among three services X, Y, and Z. For instance, the first transition denotes that the message a is sent by service X to service Y. In addition, machines X, Y, and Z shown at the bottom of the figure are services derived from the choreography by directly projecting it into each involved service. In order to enable asynchronous communication, we consider that each service is endowed with a buffer to store incoming messages. In the figure, each service transition is labeled by a tag $(S1, m1)/(S2, m2)$ stating that if the service has a message $m1$ from $S1$ stored in its buffer then it can send a message $m2$ to $S2$. Let us note that services X, Y and Z are structural copies of the choreography. In particular, each service transition is labeled with a communication action (*reading* a message from its buffer and/or *sending* a message to another service) where this action is directly taken from the corresponding choreography transition. If the service does not read or send any message in the corresponding choreography transition then we write a null action $(--,--)$ in the service transition.

This example illustrates two problems we are facing in this context. On the one hand, services could take non-deterministic choices of the choreography in a non-consistent way. For instance, let us note that, in our example, the choreography can take two possible paths (its left branch or its right branch) depending on the action taken by service Y (sending b to Z or sending c to Z, respectively). In both branches, service Y sends messages to Z, but neither Y nor Z contacts X afterwards to inform it about the action taken by Y. Since Y and Z communicate, they will follow the same branch. However, since X does not need to have any specific message in its buffer to take any of its two available transitions (both are labeled with $(--,--)/(--,--)$), X could follow the *opposite* branch as the one followed by Y and Z.

On the other hand, we may have *causality* and *race* problems, that is, services can evolve to a successor state by overtaking the rest of services in the choreography, or a sent message can be delayed in such a way that a message from a successor step overtakes it, respectively. Coming back to our example, the causality problem is observed if, for instance, after service X selects the proper branch, this service makes progress to the last transition *before* service Y has taken any of its two available actions. This may happen indeed because service X just *sends* messages and it does not need any message in its buffer to evolve to its final state. This violates the choreography requirement that X must send its message only *after* Y has already sent its own message. Regarding the other type of problem, the races problem, let us replace the first choreography transition (where X sends a to Y) by two transitions, from the same origin to the same destination, where X sends either b or c to Y, respectively. The natural projection would project these two choreography transitions into the first two transitions of services X, Y, and Z (in Z, both transitions would be null because they do not involve Z). If X sends b and *next* it sends c to Y, Y could receive them the other way around due to a long delay of b in the communication medium. Then Y would consume, in its *first* transition, the message it should take in the *third* one, and the other way around.

In this paper we will present two methods to derive a *correct* set of services from a choreography. The first derivation method solves the previous problems by adding an *orchestrator* service, which is a kind of director that is responsible of coordinating services and controlling the system workflow. An alternative method deriving a *decentralized* system, without orchestrator, is presented too. This version varies from the centralized one in the sense that it is not necessary to introduce a new “almost-omniscient” orchestrator service that would be feasible only in some concrete systems. On the contrary, in the decentralized system the *decision making* is shared by all services. In general, this situation is more realistic because web services are independent and do not share all their internal information.

In order to fix the meaning of *conformance* in this context, we define several semantic relations such that, given the orchestration of some web services and a choreography defining how these web services should interact, they decide whether the interaction of these web services necessarily leads to the required observable behavior. The proposed relations allow to assess services either in terms of the times when messages are *sent*, or in terms of the times when messages are *processed* by their destination services (that is, when destination services actually take them from their input buffers to trigger some transition). Models of orchestrations and choreographies are constructed by means of two different formal languages. Languages explicitly consider characteristics such as service identifiers, specific senders/addressees, message buffers for representing *asynchronous* communications, or message types. Besides, two semantical interpretations of asynchrony are considered: One where messages are immediately stored in input buffers of their corresponding addressees (so, the order in which messages are sent is preserved in input buffers of destination services), and another one where there might be a *delay* between the sending and the reception of each message in the input buffer of the addressee (so messages can be mixed up in destination input buffers). Centralized and decentralized derivation algorithms are presented for both interpretations.

This paper makes the following contributions:

- We provide a formal model, based on *finite state machines* (FSMs), for defining choreographies and orchestrations in an asynchronous environment.
- We propose a set of conformance relations allowing to compare systems of orchestration services and choreographies in different cases: (a) considering sending times, processing times, or both; (b) considering all choreography sequences or some of them; and (c) assuming that messages can be mixed up due to unpredictable delays in the communicating medium or not.
- We develop centralized and decentralized derivation algorithms allowing to extract, from *any* choreography (regardless of whether it is nicely defined or not), a set of services such that these services will necessarily produce the behavior required by each proposed conformance relation. Besides, proofs of the correctness of these algorithms are given.

These contributions help to face several problems related to the web service infrastructure. For instance, derivation algorithms allow to automatically extract early prototypes of web services systems from choreographies, and next we can use these models/prototypes to formally/empirically analyze their properties. Moreover, if service orchestrations do not have to be automatically derived but are *given*, then the proposed conformance relations between orchestrations and choreographies also allow developers to select the adequate service that accomplishes the behavior of certain role, thus aiding web service discovery tasks. In fact, according to our conformance relations, the correctness of a service for a given choreography depends on the behavior of the rest of services under consideration, so a kind of global web service discovery criterion is implicitly enabled by the conformance relations. Models defined in our modeling languages can be used to analyze the properties of systems of services, such as stuck-freedom and other problems derived from the concurrent execution. Since conformance relations allow us to check that a system of services conforms to a choreography, we have that, if the choreography is stuck-free, then the relation holds only if the system is so too, so conformance relations implicitly allow to check the stuck-freedom. Moreover, the derivation methods given in the paper guarantee that derived systems are stuck-free as long as the corresponding choreographies are so. Besides, by analyzing the order of exchanged messages we can study whether the information is ready when required, which concerns correlation and compensation issues.

Next we introduce the structure of this paper. The formal model to define orchestrations and choreographies is given in Section 2. The model given in this section assumes that, when a service sends a message, it is immediately stored in the input buffer of destination service. Next, the conformance relations between orchestrations and choreographies are defined in Section 3, where a collection of examples of different nature and complexity is given to show the subtle differences between these relationships. Section 4 introduces the centralized and decentralized methods to derive choreography-compliant sets of services. An alternative operational semantics, where messages that have already been sent can be delayed before they are stored in the input buffers of their addressees, is presented in Section 5. The centralized and decentralized derivation methods are adapted to this case, and their correctness under the new semantics is shown.

Section 6 presents a discussion about features beyond the current model which will be addressed in future works. The related work is presented and compared with our proposal in Section 7. Finally, we state the conclusions in Section 8. Proofs of results are given in the appendix.

2. Formal model

In this section we present our languages to define models of orchestrations and choreographies. Some preliminary notation is presented next.

Definition 2.1. Given a type A and $a_1, \ldots, a_n \in A$ with $n \geq 0$, we denote by $[a_1, \ldots, a_n]$ the *list* of elements $a_1, \ldots, a_n$ of A. We denote the empty list by $[\,]$.

Given two lists $\sigma = [a_1, \ldots, a_n]$ and $\sigma' = [b_1, \ldots, b_m]$ of elements of type A and some $a \in A$, we consider $\sigma \cdot a = [a_1, \ldots, a_n, a]$ and $\sigma \cdot \sigma' = [a_1, \ldots, a_n, b_1, \ldots, b_m]$.

Given a set of lists L, a *path-closure* of L is any subset $V \subseteq L$ such that for all $\sigma \in V$ we have that:

(a) either $\sigma = [\,]$ or $\sigma = \sigma' \cdot a$ for some σ' with $\sigma' \in V$; and

(b) there do not exist $\sigma', \sigma'' \in V$ such that $\sigma \cdot a = \sigma'$ and $\sigma \cdot b = \sigma''$ with $a \neq b$.

We say that a path-closure V of L is *complete* in L if it is *maximal* in L, that is, if there does not exist a path-closure $V' \subseteq L$ such that $V \subset V'$. The set of all complete path-closures of L is denoted by $\texttt{Comp}(L)$. □

Intuitively, a complete path-closure is a set consisting of a (maximal) sequence as well as all of its prefixes.

2.0.1. Web service orchestration model with variables

2.1. Web service orchestration model

Both our orchestration model and our choreography models will be appropriate variants of the notion of *finite state machine* (FSM). Let us note that our conformance relations to compare orchestrations and choreographies will be based on the kind of conformance relations typically appearing in formal testing techniques, where implementations are compared with specifications [26, 28, 10, 31, 30]. Since FSMs and FSM-variant models have been extensively used in this kind of frameworks as underlying models, adopting this kind of model will ease the adaptation of testing conformance notions to our model. Systems of services will be modelled as systems of a suitable variant of FSMs, while choreographies will be modelled by a more direct variant of FSM. The gap between these customized models and the kernel of WS-BPEL and WS-CDL, respectively, is not big in conceptual terms, so they will constitute a suitable (simplified) model of both languages.

We present our model of web service *orchestration*. The internal behavior of a web service in terms of its interaction with other web services is represented by a *finite state machine* where, at each state s, the machine can receive an input i and produce an output o as response before moving to a new state s'. Moreover, each transition explicitly defines which service must send i: A sender identifier snd is attached to the transition denoting that, if i is sent by service snd, then the transition can be triggered. We assume that all web services are identified by a given identifier belonging to a set ID. Moreover, transitions also denote the *addressee* of the output o, which is denoted by an identifier adr. Let us note that web services receive messages asynchronously.

This is represented in the model by considering an *input buffer* where all inputs received and not processed yet are cumulated. Each input has attached the identifier of the sender of the input. A *partition* of the set of possible inputs will be explicitly provided, and each set of the partition will denote a *type of inputs*. If a service transition requires receiving an input i whose type is t, then we will check if the first message of type t appearing in the input buffer is i indeed. If it is so (the predicate `available` given in the next definition will be used later to check this), then we will be able to consume the input from the input buffer and take the transition.

Definition 2.2. Given a set of service identifiers denoted by ID, a *service* for ID is a tuple $(id, S, I, O, s_{in}, T, \psi)$ where $id \in ID$ is the identifier of the service, S is the set of states, I is the set of inputs, O is the set of outputs, $s_{in} \in S$ is the initial state, T is the set of transitions, and ψ is a partition of I, i.e. we have $\bigcup_{p\in\psi} p = I$ and for all $p, p' \in \psi$ we have $p \cap p' = \emptyset$. Each transition $t \in T$ is a tuple (s, i, snd, o, adr, s') where $s, s' \in S$ are the initial and final states respectively, $i \in I$ is an input, $snd \in ID$ is the required sender of i, $o \in O$ is an output, and $adr \in ID$ is the addressee of o, where we require $snd \neq adr$. A transition (s, i, snd, o, adr, s') is also denoted by $s \xrightarrow{(snd,i)/(adr,o)} s'$.

Given a service $M = (id, S, I, O, s_{in}, T, \psi)$, an *input buffer* for the service M is a list $[(id_1, i_1), \ldots, (id_k, i_k)]$ where $id_1, \ldots, id_k \in ID$ and $i_1, \ldots, i_k \in I$. A *configuration* of M is a pair $c = (s, b)$ where $s \in S$ is a state of M and b is an input buffer for M. The set of all input buffers is denoted by $\mathcal{B}$. The *initial configuration* of M is $(s_{in}, [\,])$.

Let us suppose that, given a set R, 2^R denotes the powerset of R. Let us consider that $b = [(id_1, i_1), \ldots, (id_k, i_k)] \in \mathcal{B}$ with $k \geq 0$ is an input buffer, $id \in ID$, $i \in I$, and $S \in 2^I$. The predicate $\texttt{available}(b, id, i, S)$ holds iff, for some $1 \leq j \leq k$, we have $(id_j, i_j) = (id, i)$ and there do not exist $l < j$, $id' \in ID$, and $i' \in S$, such that $(id_l, i_l) = (id', i')$. We also consider $\texttt{insert}(b, id, i) = b \cdot (id, i)$. Finally, we consider $\texttt{remove}(b, id, i) = [(id_1, i_1), \ldots, (id_{j-1}, i_{j-1}), (id_{j+1}, i_{j+1}), \ldots, (id_k, i_k)]$, provided that $j \in \mathbb{N}$ is the minimum value such that $j \in [1..k]$, $id = id_j$, and $i = i_j$. □

Let us note that, alternatively, we may think about message types just as a way to have different input buffers, one for each type. In fact, the behavior of services is the same if this alternative view is adopted. If a service requires a message m from a given type $t \in \psi$ to take a given transition from the current state, then it will be able to do it only if the first message of the buffer storing messages of type t is m indeed. Since message types can be defined by any partition of the inputs set, this model captures, in particular, the case where a single input buffer is used: This is the case where a single message type (embracing all messages) is considered. On the other hand, it also allows us to consider that each kind of message has its own input buffer: This is done by considering that each type has a single message.

Next we compose services into systems of services.

Definition 2.3. Let $ID = \{id_1, \ldots, id_p\}$. For all $1 \leq j \leq p$, let $M_j = (id_j, S_j, I_j, O_j, s_{j,in}, T_j, \psi_j)$ be a service for ID. Then, $\mathcal{S} = (M_1, \ldots, M_p)$ is a *system of services* for ID.

For all $1 \leq j \leq p$, let c_j be a configuration of M_j. We say that $c = (c_1, \ldots, c_p)$ is a *configuration* of $\mathcal{S}$. Let $c'_1, \ldots, c'_p$ be the initial configurations of $M_1, \ldots, M_p$, respectively. Then, $(c'_1, \ldots, c'_p)$ is the *initial configuration* of $\mathcal{S}$. □

We formally define how systems *evolve*, i.e. how a service of the system triggers a transition and how this affects other services in the system. Outputs of services will be considered as inputs of the services these outputs are sent to. Besides, we consider a special case of input/output that will be used to denote a *null* communication. If the input of a transition is $null$ then we are denoting that the service can take this transition without waiting for any previous message from any other service, that is, we denote a *proactive* action of the service. Similarly, a $null$ output denotes that no message is sent to other service after taking the corresponding transition. In both cases, the sender and the addressee of the transition are irrelevant, respectively, so in these cases they will also be denoted by a $null$ symbol. A system evolution will be denoted by a tuple $(c, snd, i, proc, o, adr, c')$ where c and c' are the initial and the final configuration of the system, respectively, i is the input processed in the evolution, o is the output sent as result of the evolution, $proc$ is the service whose transition is taken in the evolution, snd is the sender of i, and adr is the addressee of o. There are two reasons why an evolution can be produced: (a) a service proactively initiates a transition, that is, a transition whose input is $null$ is taken; and (b) a service triggers a transition because there is an available message in its input buffer labeled by the sender identifier and the input required by the transition. In both cases (a) and (b), there are two possibilities regarding whether a new output is sent or not: (1) if the transition denotes a $null$ output then no other input buffer is modified; (2) otherwise, i.e. if the transition denotes an output different from $null$, then this output is stored in the buffer of the addressee as an *input*. By considering any combination of either (a) or (b) with either (1) or (2), four kinds of evolutions arise indeed.

Definition 2.4. Let $ID = \{id_1, \ldots, id_p\}$ be a set of service identifiers and $\mathcal{S} = (M_1, \ldots, M_p)$ be a *system of services* for ID where for all $1 \leq j \leq p$ each service M_j is defined by $M_j = (id_j, S_j, I_j, O_j, s_{j,in}, T_j, \psi_j)$. Let $c = (c_1, \ldots, c_p)$ be a configuration of $\mathcal{S}$ where for all $1 \leq j \leq p$ we have $c_j = (s_j, b_j)$.

An *evolution* of $\mathcal{S}$ from the configuration c is a tuple $(c, snd, i, proc, o, adr, c')$ where $i \in I_1 \cup \ldots \cup I_p \cup \{null\}$ is the input of the evolution, $o \in O_1 \cup \ldots \cup O_p \cup \{null\}$ is the output of the evolution, $c' = ((s'_1, b'_1), \ldots, (s'_p, b'_p))$ is the new configuration of $\mathcal{S}$, and $snd, proc, adr \in ID \cup \{null\}$ are the sender, the processor, and the addressee of the evolution, respectively. All these elements must be defined according to one of the following choices:

(a) *(evolution activated by some service by itself)* For some $1 \leq j \leq p$, let us suppose $s_j \xrightarrow{(null,null)/(adr',o')} s' \in T_j$. Then, $s'_j = s'$ and $b'_j = b_j$. Besides, $snd = null$, $proc = id_j$, $adr = adr'$, $i = null$, $o = o'$;

(b) *(evolution activated by processing a message from the input buffer of some service)* For some $1 \leq j \leq p$, let us suppose that $s_j \xrightarrow{(snd',i')/(adr',o')} s' \in T_j$ and the predicate $\texttt{available}(b_j, snd', i', r)$ holds, where r is the only set belonging to ψ_j such that $i' \in r$. Then, $s'_j = s'$ and $b'_j = \texttt{remove}(b_j, snd', i')$. Besides, $snd = snd'$, $proc = id_j$, $adr = adr'$, $i = i'$, $o = o'$;

where, both in (a) and (b), the new configurations of the rest of services are defined according to one of the following choices:

(1) *(no message is sent to another service)* If $adr' = null$ or $o' = null$ then for all $1 \leq q \leq p$ with $q \neq j$ we have $s'_q = s_q$ and $b'_q = b_q$.

(2) *(a message is sent to another service)* Otherwise, let $id_g = adr'$ for some $1 \leq g \leq p$. Then, we have $s'_g = s_g$ and $b'_g = \texttt{insert}(b_g, id_j, o')$. Besides, for all $1 \leq q \leq p$ with $q \neq j$ and $q \neq g$ we have $s'_q = s_q$ and $b'_q = b_q$.

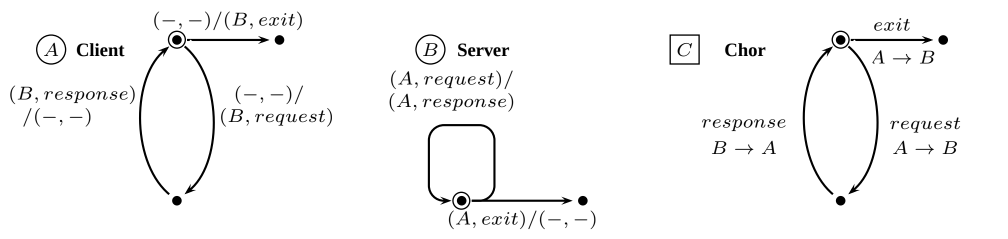

Figure 2: A client/server orchestration (left and center) and a choreography specification (right).

□

Let us note that the previous operational semantics implicitly assume that, when a message is sent, it is immediately stored in the input buffer of the destination service, so messages are stored in the same order as they are sent and they cannot be mixed up. Though this kind of order preservation might be feasible in some cases indeed (e.g. the underlying networking protocol could undertake the responsibility of reordering messages at some lower abstraction level, as it happens, for instance, in live video streaming), this might not be feasible in other cases. In Section 5 we will present an alternative framework where this is not assumed.

Figure 2 (left and center) shows a simple client/server orchestration specification where the client (A) sends requests to the server (B) and the server responds to them, until the client notifies that it leaves the system. Initial states are denoted by a double circle node, and $null$ inputs and outputs are denoted by the dash symbol.

As we will see later, the conformance of a system of service orchestrations with respect to a choreography will be assessed in terms of the behaviors of both machines. We extract the behaviors of systems of services as follows: Given any sequence of consecutive evolutions of the system from its initial configuration, we take the sequence of inputs and outputs labeling each evolution and we remove all $null$ elements from this sequence. The extracted sequence (called *trace*) represents the *effective* behavior of the original sequence. We distinguish two kinds of traces. A *sending trace* is a sequence of outputs ordered as they are *sent* by their corresponding senders. A *processing trace* is a sequence of inputs ordered as they are *processed* by the services which receive them, that is, they are ordered as they are taken from the input buffer of each addressee service to trigger some of its transitions. Both traces attach some information to explicitly denote the services involved in each operation.

Definition 2.5. Let $\mathcal{S}$ be a system, c_1 be the initial configuration of $\mathcal{S}$, and the set of tuples $(c_1, snd_1, i_1, proc_1, o_1, adr_1, c_2)$, $(c_2, snd_2, i_2, proc_2, o_2, adr_2, c_3), \ldots,$ $(c_k, snd_k, i_k, proc_k,$ $o_k, adr_k, c_{k+1})$ be k consecutive evolutions of $\mathcal{S}$.

Let $a_1 \leq \ldots \leq a_r$ denote all indexes of non-null outputs in the previous sequence, i.e. we have $j \in \{a_1, \ldots, a_r\}$ iff $o_j \neq null$. Then, $[(proc_{a_1}, o_{a_1}, adr_{a_1}), \ldots, (proc_{a_r}, o_{a_r}, adr_{a_r})]$ is a *sending trace* of $\mathcal{S}$. In addition, if there do not exist $snd', i', proc', o', adr', c'$ such that $(c_{k+1}, snd', i', proc', o', adr', c')$ is an evolution of $\mathcal{S}$ then we also say that $[(proc_{a_1}, o_{a_1}, adr_{a_1}),$ $\ldots, (proc_{a_r}, o_{a_r}, adr_{a_r}), \texttt{stop}]$ is a sending trace of $\mathcal{S}$. The set of all sending traces of $\mathcal{S}$ is denoted by $\texttt{sndTraces}(\mathcal{S})$.

Let $a_1 \leq \ldots \leq a_r$ denote all indexes of non-null inputs in the previous sequence, i.e. we have $j \in \{a_1, \ldots, a_r\}$ iff $i_j \neq null$. Then, $[(snd_{a_1}, i_{a_1}, proc_{a_1}), \ldots, (snd_{a_r}, i_{a_r}, proc_{a_r})]$

is a *processing trace* of $\mathcal{S}$. In addition, if there do not exist $snd', i', proc', o', adr', c'$ such that $(c_{k+1}, snd', i', proc', o', adr', c')$ is an evolution of $\mathcal{S}$ then we also say that $[(snd_{a_1}, i_{a_1}, proc_{a_1}), \dots, (snd_{a_r}, i_{a_r}, proc_{a_r}), \texttt{stop}]$ is a processing trace of $\mathcal{S}$. The set of all processing traces of $\mathcal{S}$ is denoted by $\texttt{prcTraces}(\mathcal{S})$. $\square$

2.2. *Choreography model*

Next we introduce our formalism to represent choreographies. Contrarily to systems of orchestrations, this formalism focuses on representing the interaction of services as a whole. Thus a single machine, instead of the composition of several machines, is considered. Each choreography transition denotes a *message action* where some service sends a message to another service.

Definition 2.6. A *choreography machine* $\mathcal{C}$ is a tuple $\mathcal{C} = (S, M, ID, s_{in}, T)$ where S denotes the set of states, M is the set of messages, ID is the set of service identifiers, $s_{in} \in S$ is the initial state, and T is the set of transitions. A transition $t \in T$ is a tuple (s, m, snd, adr, s') where $s, s' \in S$ are the initial and final states, respectively, $m \in M$ is the message, and $snd, adr \in ID$ are the sender and the addressee of the message, respectively. A transition (s, m, snd, adr, s') is also denoted by $s \xrightarrow{m/(snd \rightarrow adr)} s'$.

A *configuration* of $\mathcal{C}$ is any state $s \in S$. An *evolution* of $\mathcal{C}$ from the configuration $s \in S$ is any transition $(s, m, snd, adr, s') \in T$ from state s. The *initial configuration* of $\mathcal{C}$ is s_{in}. $\square$

Coming back to our previous example, Figure 2 (right) depicts a choreography C between services A and B, that is, the client and the server. The transitions of this choreography actually denote the same evolutions we can find in a system of services consisting of services A and B.

As we did before for systems of services, next we identify the sequences of messages that can be produced by a choreography machine.

Definition 2.7. Let c_1 be the initial configuration of a choreography machine $\mathcal{C}$. Let the tuples $(c_1, m_1, snd_1, adr_1, c_2), \dots, (c_k, m_k, snd_k, adr_k, c_{k+1})$ be $k \geq 0$ consecutive evolutions of $\mathcal{C}$. We say that $\sigma = [(snd_1, m_1, adr_1), \dots, (snd_k, m_k, adr_k)]$ is a *trace* of $\mathcal{C}$. In addition, if there do not exist m', snd', adr', c' such that $(c_{k+1}, m', snd', adr', c')$ is an evolution of $\mathcal{C}$ then we also say that $[(snd_1, m_1, adr_1), \dots, (snd_k, m_k, adr_k), \texttt{stop}]$ is a trace of $\mathcal{C}$. The set of all traces of $\mathcal{C}$ is denoted by $\texttt{traces}(\mathcal{C})$. $\square$

3. Conformance relations

Now we are provided with all the required formal machinery to define our *conformance relations* between systems of orchestrations and choreographies. We will consider a semantic relation inspired in the *conformance testing* relation given in [33, 34]. This notion is devoted to check whether an *implementation* meets the requirements imposed by a *specification*. In our case, we will check whether the behavior of a system of orchestration services meets the requirement given by the choreography.

However, there are some important differences between the notion proposed in [33, 34] and the notion considered here. Contrarily to those works, the behavior of orchestrations and choreographies will not be compared in terms of their possible interactions with an external entity (i.e. user, observer, external application, etc) but in terms of what both models can/cannot do by their own, because both models are considered as *closed worlds*. Let us also note that non-determinism allows a choreography to provide multiple valid ways to perform the operations it defines. Consequently, we consider that a system of orchestration services conforms to a choreography if it performs *one or more* of these valid ways. For each of these valid ways, care must be taken not to allow the system of services to *incompletely* perform it, i.e. to finish in an intermediate state – provided that the choreography does not allow it either. In order to check these requirements, only complete path-closures will be considered (see Definition 2.1). Moreover, the set of complete path-closures of the system of services is required to be non-empty because the system is required to provide at least *one* (complete) way to perform the requirement given by the choreography. Alternatively, we also consider another relation where the system of services is required to perform *all* execution ways defined by the choreography. This alternative notion will be called *full conformance*.

There are more differences between the conformance relation of [33, 34] and our approach. Let us recall that we consider asynchronous communications in our framework. Thus, the moment when a message is sent does not necessarily coincide with the moment when this message is taken by the receiver from its input buffer and is processed. In fact, we can define a choreography in such a way that defined communications refer to either the former kind of events or the latter (i.e., instants where messages are sent, or instants where messages are processed by their receivers, respectively). Thus, we consider two ways in which a system of services may conform to a choreography: with respect to sending traces, and with respect to processing traces. A similar distinction was proposed in [25], as it is commented in Section 7. The case where both conformance notions simultaneously hold is also identified.

Definition 3.1. Let $\mathcal{S}$ be a system of services and $\mathcal{C}$ be a choreography machine.

We say that $\mathcal{S}$ *conforms to* $\mathcal{C}$ *with respect to sending actions*, denoted by $\mathcal{S}\ \mathtt{conf}_s\ \mathcal{C}$, if either we have that $\emptyset \subset \mathtt{Comp}(\mathtt{sndTraces}(\mathcal{S})) \subseteq \mathtt{Comp}(\mathtt{traces}(\mathcal{C}))$ or we have that $\emptyset = \mathtt{Comp}(\mathtt{sndTraces}(\mathcal{S})) = \mathtt{Comp}(\mathtt{traces}(\mathcal{C}))$.

We say that $\mathcal{S}$ *fully conforms to* $\mathcal{C}$ *with respect to sending actions*, denoted by $\mathcal{S}\ \mathtt{conf}_s^f\ \mathcal{C}$, if $\mathtt{Comp}(\mathtt{sndTraces}(\mathcal{S})) = \mathtt{Comp}(\mathtt{traces}(\mathcal{C}))$.

We say that $\mathcal{S}$ *conforms to* $\mathcal{C}$ *with respect to processing actions*, denoted by $\mathcal{S}\ \mathtt{conf}_p\ \mathcal{C}$, if we have either that $\emptyset \subset \mathtt{Comp}(\mathtt{prcTraces}(\mathcal{S})) \subseteq \mathtt{Comp}(\mathtt{traces}(\mathcal{C}))$ or we have that $\emptyset = \mathtt{Comp}(\mathtt{prcTraces}(\mathcal{S})) = \mathtt{Comp}(\mathtt{traces}(\mathcal{C}))$.

We say that $\mathcal{S}$ *fully conforms to* $\mathcal{C}$ *with respect to processing actions*, denoted by $\mathcal{S}\ \mathtt{conf}_p^f\ \mathcal{C}$, if $\mathtt{Comp}(\mathtt{prcTraces}(\mathcal{S})) = \mathtt{Comp}(\mathtt{traces}(\mathcal{C}))$.

We say that $\mathcal{S}$ *conforms to* $\mathcal{C}$, denoted by $\mathcal{S}\ \mathtt{conf}\ \mathcal{C}$, if $\mathcal{S}\ \mathtt{conf}_s\ \mathcal{C}$ and $\mathcal{S}\ \mathtt{conf}_p\ \mathcal{C}$.

We say that $\mathcal{S}$ *fully conforms to* $\mathcal{C}$, denoted by $\mathcal{S}\ \mathtt{conf}^f\ \mathcal{C}$, if $\mathcal{S}\ \mathtt{conf}_s^f\ \mathcal{C}$ and $\mathcal{S}\ \mathtt{conf}_p^f\ \mathcal{C}$.

□

3.1. Using the conformance relations: Examples

In this section we illustrate the use of the conformance relations given in Definition 3.1 with several simple examples. A small case study introducing a more elaborated system will be given in the next section.

Intuitively, a complete path-closure (see Definition 2.1) is a set consisting of a (maximal) sequence as well as all of its prefixes. Let us note that the longest element of a *finite* complete path-closure of traces necessarily finishes with the `stop` symbol. For the sake of clarity, from now on a complete path-closure will be referred just by its longest element not including

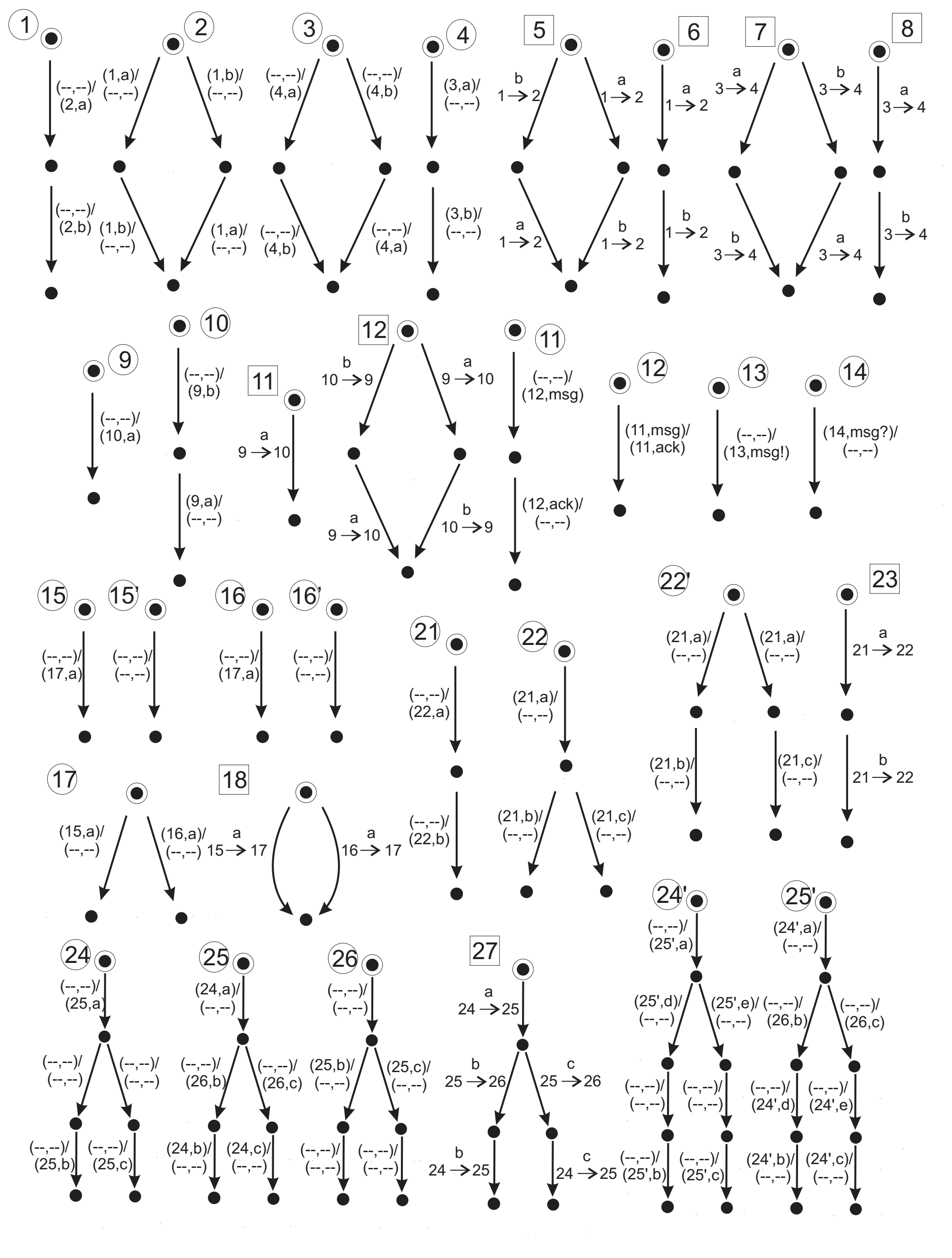

Figure 3: Orchestrations and choreographies.

the `stop` symbol. For instance, the complete path-closure $\{[\ ], [(1,a,2)], [(1,a,2),(1,b,2)], [(1,a,2),(1,b,2), \texttt{stop}]\}$ will be referred just by $[(1,a,2),(1,b,2)]$ (and we will say that $[(1,a,2),(1,b,2)]$ is a *complete trace*). Following a similar idea, an *infinite* complete path-closure $\{[\ ], [(a_1,b_1,c_1)], [(a_1,b_1,c_1),(a_2,b_2,c_2)], [(a_1,b_1,c_1),(a_2,b_2,c_2), (a_3,b_3,c_3)],\ldots\}$ will be referred by the infinite list $[(a_1,b_1,c_1),(a_2,b_2,c_2),(a_3,b_3,c_3),\ldots]$.

Figure 3 presents several orchestration services and choreographies. For all depicted services we will assume that each input belongs to a different type of inputs. Let S_1 be a system of orchestration services consisting of services 1 and 2. We check whether S_1 conforms to choreographies 5 and 6. If we consider the $\texttt{conf}_s$ relation, then we observe that S_1 conforms to both 5 and 6. This is because the only possible complete sending trace of S_1 is $[(1,a,2),(1,b,2)]$, which is included in the set of complete traces of 5 (which is $\{[(1,a,2),(1,b,2)],[(1,b,2),(1,a,2)]\}$) and 6 ($\{[(1,a,2),(1,b,2)]\}$). Concerning *full* conformance, we have that S_1 fully conforms to 6 with respect to sending traces, but not to 5. Regarding processing traces, let us note that S_1 can generate the complete processing traces $[(1,a,2),(1,b,2)]$ and $[(1,b,2),(1,a,2)]$ (note that, after a and b are received in the input buffer of service 2, service 2 can process them in any order). Both complete processing traces are included in the set of complete traces of 5, but not in the corresponding set of 6, which only includes $[(1,a,2),(1,b,2)]$. Thus, if either $\texttt{conf}_p$ or $\texttt{conf}_p^f$ are considered, then S_1 conforms to 5, but not to 6.

Let S_2 be the system consisting of services 3 and 4, and let us compare it with choreographies 7 and 8. In this case, we have the opposite result as before. In particular, if processing traces are considered, then S_2 conforms to both choreographies (if *full* conformance is considered, it only conforms to 8). However, S_2 does not conform to 8 when sending traces are considered, regardless of whether full conformance is considered or not. Let us note that S_2 can perform the sending traces $[(3,a,4),(3,b,4)]$ and $[(3,b,4),(3,a,4)]$. However, the sets of complete traces of choreographies 7 and 8 are $\{[(3,a,4),(3,b,4)],[(3,b,4),(3,a,4)]\}$ and $\{[(3,a,4),(3,b,4)]\}$, respectively. Thus, if $\texttt{conf}_s$ or $\texttt{conf}_s^f$ are considered, then S_2 conforms to choreography 7, but not to choreography 8.

Next, let S_3 be the system consisting of services 9 and 10. We compare S_3 with choreographies 11 and 12. The set of complete sending traces of S_3 is equal to $\{[(9,a,10),(10,b,9)], [(10,b,9),(9,a,10)]\}$, while the set of complete processing traces of S_3 is $\{[(9,a,10)]\}$. On the one hand, the only complete trace of choreography 11 is $[(9,a,10)]$, so S_3 conforms to 11 only if processing traces are considered (with respect to both $\texttt{conf}_p$ and $\texttt{conf}_p^f$). On the other hand, choreography 12 can produce both $[(9,a,10),(10,b,9)]$ and $[(10,b,9),(9,a,10)]$. Since only complete traces are considered, S_3 conforms to 12 only if sending traces are regarded (according to both $\texttt{conf}_s$ and $\texttt{conf}_s^f$).

Despite of the fact that only asynchronous communications are considered in our framework, a kind of synchronous communications can be trivially defined indeed. Let us consider the system S_4 consisting of services 11 and 12. After 11 sends message msg to 12, service 11 will be blocked until 12 performs its unique transition and sends message ack back to 11. So, a synchronous communication between 11 and 12 is actually expressed by this trivial structure. A syntactic sugar to denote a synchronous communication like this is implicitly proposed in pictures of services 13 and 14, which are intended to be equivalent to 11 and 12, respectively. In particular, we denote a synchronous communication on message msg by using new symbols $msg?$ and $msg!$.

Let us recall that the suitability of an orchestration service to fulfill a given choreography depends on the behavior of the rest of involved services. Let us consider that a *travel agency* service requires that either the *air company* service or the *hotel* service (or both) provide a transfer to take the client from the airport to the hotel. A hotel providing a transfer is *good* regardless of whether the air company provides a transfer as well or not. However, a hotel not providing a transfer is valid for the travel agency *only if* the air company does provide the transfer. Choreography 18 denotes the requirement that either the air company (represented by service 15) or the hotel (service 16) must provide the travel agency (service 17) with a message a standing for "*we provide you with a transfer service.*" In fact, we consider two possible air companies, represented by services 15 and $15'$. Service 15 provides service 17 with a transfer service, while $15'$ does not (it does nothing). Similarly, services 16 and $16'$ represent two hotels, where only 16 provides the travel agency with a transfer. Most combinations of, on one hand, either 15 or $15'$ and, on the other hand, either 16 or $16'$, allow 17 to satisfy choreography 18 with respect to (non-full) sending and processing conformance. In fact, only combining $15'$ with $16'$ fails to meet both non-full conformance relations. Thus, either the air company or the hotel must provide the transfer. If full conformance is required, then the only valid combination of air company and hotel consists in taking 15 and 16, respectively.

We show that systems of orchestrations are required to *complete* all started sequences, that is, they are required not to finish a started sequence until the choreography explicitly allows it. Let us consider orchestration services 21, 22, and $22'$, as well as choreography 23. Let S_5 be a system consisting of services 21 and 22. The sequence $[(21, a, 22), (21, b, 22)]$ is both the only complete sending trace and the only complete processing trace of S_5. Thus, S_5 conforms to choreography 23 with respect to both kinds of traces. Let us substitute the definition of service 22 by that given for service $22'$, and let S'_5 be the resulting system. The set of complete sending traces of S'_5 is the same as S_5, so S'_5 also conforms to 23 with respect to sending traces. However, the set of complete processing traces of S'_5 is $\{[(21, a, 22'), (21, b, 22')], [(21, a, 22')]\}$ because $22'$ could take its right path and get stuck after receiving a (more formally, $[(21, a, 22'), \texttt{stop}]$ is a processing trace of S'_5). Since $[(21, a, 22')]$ is not a *complete* processing trace of 23, S'_5 does not conform to 23 with respect to processing traces.

Finally, we consider a case where there are *infinite* complete traces in systems due to the presence of loops. Let us revisit the orchestrations and the choreography previously depicted in Figure 2, and let $\mathcal{S}$ be the composition of A and B. The infinite set of complete traces of choreography C is $T = \{\sigma, \sigma_1, \sigma_2, \sigma_3, \ldots\}$, where σ is the infinite concatenation of the subsequence $\alpha = [(A, request, B), (B, response, A)]$, that is, $\sigma = \alpha \cdot \alpha \cdot \alpha \cdot \ldots$, and for all $i \in \mathbb{N}$ we have $\sigma_i = (\alpha)^i \cdot (A, exit, B)$. In fact, the infinite set of complete sending and processing traces of $\mathcal{S}$ is T as well, so we have that $\mathcal{S}$ conforms to C with respect to all relations $\texttt{conf}_s$, $\texttt{conf}_p$, $\texttt{conf}$, $\texttt{conf}^f_s$, $\texttt{conf}^f_p$, and $\texttt{conf}^f$.

3.2. Case study: Purchase Process

In order to illustrate the application of the proposed notions to a more elaborated system, in this section we present a small case study. This is a typical purchase process that uses Internet as a business context for a transaction. There are three actors in this example: a customer, a seller, and a carrier. The purchase works as follows: "*A customer wants to buy a product by using Internet. There are several sellers that offer different products in web-pages servers. The customer contacts a seller in order to buy the desired product. The seller checks the stock and contacts a carrier. Finally, the carrier delivers the product to the customer.*"

Figures 4 and 5 depict the orchestration of the three actors represented in this purchase process, that is, the customer, the seller, and the carrier. The behavior of each participant is defined as follows:

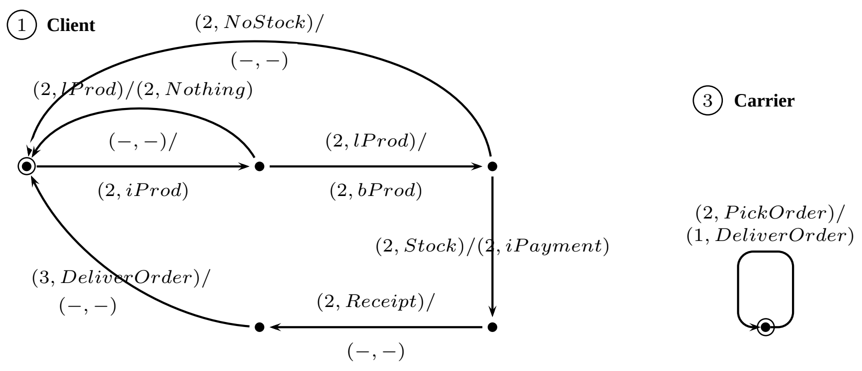

Figure 4: Client and Carrier orchestration specifications.

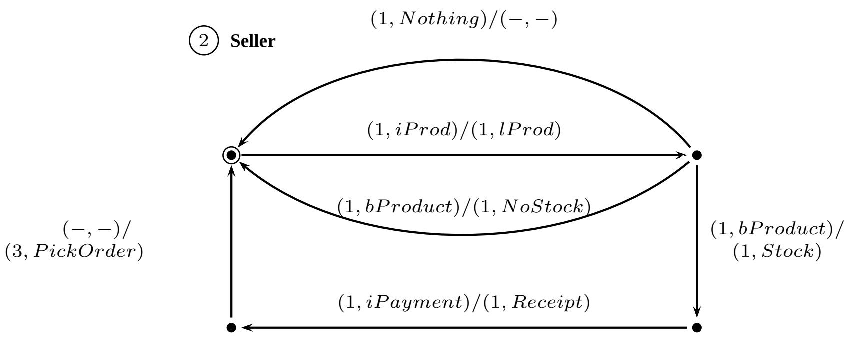

Figure 5: Seller orchestration specification.

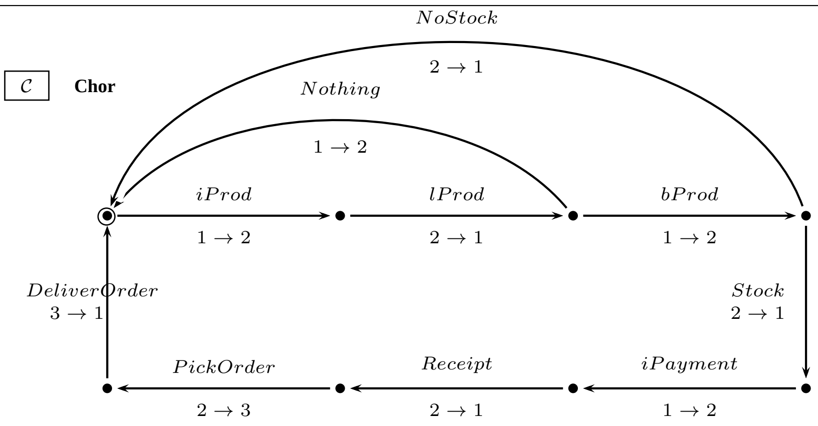

Figure 6: The choreography specification.

sq_1	[(1,iProduct,2), (2,lProduct,1), (1,Nothing,2)]
sq_2	[(1,iProduct,2), (2,lProduct,1), (1,bProduct), (2,NoStock,1)]
sq_3	[(1,iProduct,2), (2,lProduct,1), (1,bProduct), (2,Stock,1), (1,iPayment,2), (2,Receipt,1), (2,PickOrder,3), (3,DeliverOrder,1)]

Table 1: Some sequences of choreography $\mathcal{C}$.

- Customer: It contacts the seller to buy a product. After consulting the product list, it can either order a product or do nothing. If the customer decides to buy a product, then it must send the seller the information about the product and the payment method. After the payment, it waits to receive the product from a carrier.

- Seller: It receives the customer order and the payment method. The seller checks if there is enough stock to deliver the order and sends an acceptance notification to the customer. If there is stock to deliver the order, then it contacts a carrier to deliver the product.

- Carrier: It picks up the order and the customer information in order to deliver the product to the customer.

Figure 6 shows the choreography of this Internet purchase process. Once the three services and the choreography specification are defined, we use the conformance relations given in Definition 3.1 to check if the composition of the proposed orchestration services satisfies the choreography.

Let us consider a system $\mathcal{S} = (1, 2, 3)$, where 1, 2, and 3 represent the client service, the seller service, and the carrier service, respectively. Let $\mathcal{C}$ be the choreography machine depicted in Figure 6, and let sq_1, sq_2, sq_3 be the three sequences depicted in Table 1. For all complete traces σ of $\mathcal{C}$, σ is an infinite concatenation of these subsequences, that is, $\sigma = \alpha_1 \cdot \alpha_2 \cdot \alpha_3 \cdot \alpha_4 \cdot \ldots$ where for all $i \in \mathbb{N}$ we have $\alpha_i \in \{sq_1, sq_2, sq_3\}$. It is easy to see that any complete sending or processing trace σ' of $\mathcal{S}$ must also be an infinite concatenation of subsequences sq_1, sq_2, sq_3. Hence, for all $\sigma' \in \texttt{Comp}(\texttt{sndTraces}(\mathcal{S})) \cup \texttt{Comp}(\texttt{prcTraces}(\mathcal{S}))$ we have $\sigma' \in \texttt{Comp}(\texttt{traces}(\mathcal{C}))$, and thus we have both $\mathcal{S}\texttt{conf}_s\mathcal{C}$ and $\mathcal{S}\texttt{conf}_p\mathcal{C}$, which implies $\mathcal{S}\texttt{conf}\mathcal{C}$. Moreover, in this case we also have that, for all $\sigma \in \texttt{Comp}(\texttt{traces}(\mathcal{C}))$, $\sigma \in \texttt{Comp}(\texttt{sndTraces}(\mathcal{S}))$ and $\sigma \in \texttt{Comp}(\texttt{prcTraces}(\mathcal{S}))$. Therefore, we also have $\mathcal{S}\texttt{conf}^f_s\mathcal{C}$, $\mathcal{S}\texttt{conf}^f_p\mathcal{C}$, and $\mathcal{S}\texttt{conf}^f\mathcal{C}$.

4. Derivation of choreography-compliant sets of services

Once we are provided with appropriate notions to compare sets of orchestration models with choreography models, we study the problem of automatically deriving orchestration services from a given choreography, in such a way that the system consisting of these derived services conforms to the choreography.

Let us reconsider the possibility of deriving services by applying *natural projection* (see Figure 1) to the structure of the choreography into each involved service. Each service copies the form of states and transitions of the choreography, though service transitions are labeled only by actions concerning that service. Unfortunately, as we saw in the introduction, if services are derived in this way then, in general, the resulting set of services does not conform to the choreography with respect to any of the proposed conformance notions. Let us revisit services X, Y, and Z of Figure 1, which are natural projections of choreography $Chor$ (given in the same figure) into each service regarded in the definition of $Chor$. The composition of X, Y, and Z does not necessarily lead to the behavior required by $Chor$ due to the two problems we explained in the introduction: (a) the possible inconsistency of service choices in non-deterministic elections; and (b) the races risk.

In order to enable the comparison of this (wrong) derivation with some forthcoming alternatives in a single figure, services X, Y, Z, and choreography $Chor$ are depicted again in Figure 3 under the new names of 24, 25, 26, and 27, respectively. Let us note that, if *only* messages appearing in choreography 27 are allowed in services then no alternative definition of 24, 25, and 26 allows to meet the requirement imposed by 27: Service 24 cannot decide whether it must send b or c to 25 because it cannot know the message sent by 25 to 26. We will make any choreography realizable by *adding* some control messages to the definition of services. These messages will allow services to know what is required at each time to properly make the next decision, according to the choreography specification.

Next we reconsider our conformance relations under the assumption that these additional messages are allowed indeed. That is, services are allowed to send/receive additional messages not included in the choreography. In order to avoid confusion between standard choreography messages and other messages, the latter messages are required to be different from the former. Regarding the definition of conformance relations, we require traces inclusion/equality again, though we remove additional messages prior to comparing sets of traces.

Definition 4.1. Let $\sigma \in \mathtt{sndTraces}(\mathcal{S}) \cup \mathtt{prcTraces}(\mathcal{S})$ where $\mathcal{S}$ is a system of services. The *constrain* of σ to a set of inputs and outputs Q, denoted by σ^Q, is the result of removing from σ all elements (a, m, b) with $m \notin Q$.

Let $\mathcal{S}$ be a system of services for ID and let $\mathcal{C} = (S, M, ID, s_{in}, T)$ be a choreography. Let $\mathtt{conf}_x \in \{\mathtt{conf}_s, \mathtt{conf}_s^f, \mathtt{conf}_p, \mathtt{conf}_p^f\}$. We have $\mathcal{S}\ \mathtt{conf}'_x\ \mathcal{C}$ if $\mathcal{S}\ \mathtt{conf}_x\ \mathcal{C}$ provided that the occurrences of $\mathtt{sndTraces}(\mathcal{S})$ and $\mathtt{prcTraces}(\mathcal{S})$ appearing in Definition 3.1 are replaced by sets $\{\sigma^M | \sigma \in \mathtt{sndTraces}(\mathcal{S})\}$ and $\{\sigma^M | \sigma \in \mathtt{prcTraces}(\mathcal{S})\}$, respectively. Now, let $\mathtt{conf}_x \in \{\mathtt{conf}, \mathtt{conf}^f\}$. We have $\mathcal{S}\ \mathtt{conf}'_x\ \mathcal{C}$ if $\mathcal{S}\ \mathtt{conf}_x\ \mathcal{C}$ provided that the occurrences of $\mathtt{conf}_s$, $\mathtt{conf}_s^f$, $\mathtt{conf}_p$, $\mathtt{conf}_p^f$ appearing in the definition of $\mathtt{conf}$ and $\mathtt{conf}^f$, given in Definition 3.1, are replaced by $\mathtt{conf}'_s, \mathtt{conf}_s^{f\prime}, \mathtt{conf}'_p, \mathtt{conf}_p^{f\prime}$, respectively. □

We revisit our previous example. Let us modify services 24 and 25 in such a way that, right after 25 sends b or c to service 26, service 25 tells service 24 whether b or c was sent. This is done by sending to service 24 a new message d or e, respectively. Services $24'$ and $25'$ (also depicted in Figure 3) are the resulting new versions of services 24 and 25, respectively. Let us note that the system consisting in $24'$, $25'$, and 26 conforms to 27 with respect to all conformance relations introduced in the previous definition, because all of them ignore messages d and e.

4.1. Centralized derivation method

Let us present our first method to derive a choreography-compliant set of services from a given choreography. Intuitively, a service derivation based on a simple natural projection does not work because it does not guarantee that services will follow the elections and the sequencing of events defined by the choreography. In order to solve this problem, next we consider an alternative way to extract services from the choreography that is inspired on our previous example ($24'$, $25'$, 26). New control messages will be added to make all services follow the same choices at each branching point of the choreography. In particular, we will introduce a new service, called *orchestrator*, which will be responsible of making all choices at choreography branching points, as well as making services follow such choices. For each state s_j of the choreography having several outgoing transitions, the orchestrator will have an equivalent state with the same set of outgoing transitions, which represent all the choices it can make. At that state, the orchestrator will choose any of these transitions, say the p-th available transition. Then, the orchestrator will take several consecutive transitions to *announce* its choice to all services. In each of these transitions, the orchestrator will send a message a_{jp} to another service, meaning that the p-th transition leaving state s_j must be taken by the service. After (a) the orchestrator announces its choice to all services; and (b) the orchestrator receives a message b_f from the *addressee* id_f of the choreography transition (this message denotes that the addressee has processed the message), the orchestrator will reach a state representing the state reached in the choreography after taking the selected transition, and the same process will be followed again. By adding the orchestrator, we make sure that all services take the same branch in each branching point of the choreography. However, it is worth to point out that, since the only message required by the orchestrator to continue is sent by the addressee of the choreography transition, at a given time the orchestrator and the services could have reached different steps of the choreography simulation execution (in general, the orchestrator will be in a *further* step). There is no risk that services break the relative order in which transitions must be taken according to the choreography, because all messages controlling transition choices are introduced in input buffers (as the rest of messages) and they will belong to the same *type*. Thus, they will be processed in the same order as the orchestrator sent each of them. This guarantees that services will be led through the choreography graph by following the orchestrator plan, in the same order as planned. In particular, as we will see after the next definition, a system consisting of the orchestration and the corresponding derived services will *conform* to the choreography with respect to all $\texttt{conf}'_x$ relations given in Definition 4.1. Next we will assume that the identifier of the orchestrator is orc.

Definition 4.2. Let $\mathcal{C} = (S, M, ID, s_{in}, T)$ be a choreography machine where the set of identifiers is $ID = \{id_1, \ldots, id_n\}$ and the set of states is $S = \{s_1, \ldots, s_l\}$. For all $1 \leq i \leq n$, the *controlled service* for $\mathcal{C}$ and id_i, denoted $\texttt{controlled}(\mathcal{C}, id_i)$, is a service

$$M_i = \left(\begin{array}{l} id_i, S \cup \{s_{ij}, s'_{ij} | i,j \in [1..l]\}, \\ M \cup \{a_{ij} | i,j \in [1..l]\}, M \cup \{b_f | f \in [1..l]\}, \\ s_{in}, T_i, \{\{m\} | m \in M\} \cup \{\{a_{ij} | i,j \in [1..l]\}\} \end{array} \right)$$

where for all $s_j \in S$ the following transitions are in T_i:

- Let $t_1, \ldots, t_k$ be the transitions leaving s_j in $\mathcal{C}$. For all $1 \leq p \leq k$ we add the transition $s_j \xrightarrow{(orc, a_{jp})/(null, null)} s_{jp} \in T_i$.
- For all $1 \leq p \leq k$, if $t_p = s_j \xrightarrow{m/(snd \rightarrow adr)} s'_j \in T$ is the p-th transition leaving s_j in $\mathcal{C}$, then we have $s_{jp} \xrightarrow{(snd', i)/(adr', o)} u_{jp} \in T_i$ where
  - (a) if $snd = id_i$ then $snd' = i = null$, $adr' = adr$, $o = m$, and $u_{jp} = s'_j$.
  - (b) else, if $adr = id_i$ then $snd' = snd$, $i = m$, $adr' = o = null$, and $u_{jp} = s'_{jp}$. Besides, we also have $s'_{jp} \xrightarrow{(null, null)/(orc, b_i)} s'_j$ in T_i.
  - (c) else $snd' = i = adr' = o = null$ and $u_{jp} = s'_j$.

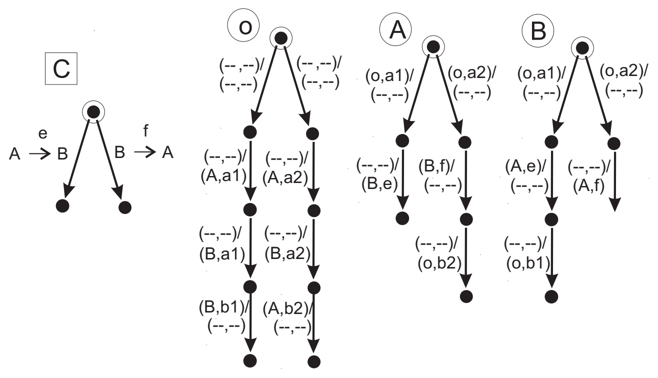

Figure 7: Derivation of services with orchestrator.

The *orchestrator* of $\mathcal{C}$, denoted by $\texttt{orchestrator}(\mathcal{C})$, is a service

$$O = \left(\begin{array}{l} orc, S \cup \{s_{ijk} | i,j \in [1..l], k \in [1..n{+}1]\}, \\ M \cup \{b_f | f \in [1..l]\}, M \cup \{a_{ij} | i,j \in [1..l]\}, \\ s_{in}, T_o, \{\{m\} | m \in M\} \cup \{\{b_f\} | f \in [1..l]\} \end{array} \right)$$

where for all $s_j \in S$ the following transitions are included in T_o:

- Let $t_1, \ldots, t_k$ be the transitions leaving s_j in $\mathcal{C}$. For all $1 \leq p \leq k$ we add the transition $s_j \xrightarrow{(null,null)/(null,null)} s_{jp1} \in T_o$.
- For all $1 \leq p \leq k$, if $t_p = s_j \xrightarrow{m/(snd \to adr)} s'_j \in T$ is the p-th transition leaving s_j in $\mathcal{C}$ and $adr = id_f$, then for all $1 \leq i \leq n$ we have $s_{jpi} \xrightarrow{(null,null)/(id_i,a_{jp})} s_{jp\ i+1} \in T_o$. We also have $s_{jp\ n+1} \xrightarrow{(adr,b_f)/(null,null)} s'_j \in T_o$.

□

Theorem 4.3. Let $\mathcal{C} = (S, M, ID, s_{in}, T)$ be a choreography with $ID = \{id_1, \ldots, id_n\}$. Let $\mathcal{S} = (\texttt{controlled}(\mathcal{C}, id_1), \ldots, \texttt{controlled}(\mathcal{C}, id_n), \texttt{orchestrator}(\mathcal{C}))$. For all conformance relationships $\texttt{conf}_x \in \{\texttt{conf}'_s, \texttt{conf}'_p, \texttt{conf}', \texttt{conf}^{f\prime}_s, \texttt{conf}^{f\prime}_p, \texttt{conf}^{f\prime}\}$ we have $\mathcal{S}\ \texttt{conf}_x\ \mathcal{C}$. □

The proof of the previous result, as well as the proofs of the rest of results, are given in the appendix. Figure 7 shows a choreography C as well as the services derived from C by applying Definition 4.2, including an orchestrator O.

If we do not need to meet the conformance with respect to processing traces, that is, if we only require $\texttt{conf}'_s$ and $\texttt{conf}^{f\prime}_s$, then we do not need to require that addressees of choreography transitions *block* the advance of the orchestrator until they process received messages. This restriction was imposed just to force the processing of messages follow the order required by the choreography. Alternatively, if addressees did not block the orchestrator then, for instance, the service responsible of processing the second message of the execution could process it before the service responsible of processing the first one does so. Even if the orchestrator were not required to wait for the addressees, the order in which messages are *sent* would be correct as long as the orchestrator is required to wait for the *senders*. Actually, if we only consider conformance with respect to sending traces then replacing the restriction of waiting for the addressees by the restriction of waiting for the senders is a good choice in terms of efficiency. This is because, in this case, the orchestrator will not be blocked just waiting for the message to be processed; on the contrary, it will be able to go on even if the message has not been processed yet. Thus, by taking this alternative, the rate of activities the services can actually execute in *parallel* is increased.

Definition 4.4. We have that $\texttt{controlled'}(\mathcal{C}, id_i)$ is defined as $\texttt{controlled}(\mathcal{C}, id_i)$ after replacing cases (a) and (b) of Definition 4.2 by the following expressions:

(a) if $snd = id_i$ then $snd' = i = null$, $adr' = adr$, $o = m$, and $u_{jp} = s'_{jp}$. Besides, we also have $s'_{jp} \xrightarrow{(null,null)/(orc,b_i)} s'_j$ in T_i.

(b) else, if $adr = id_i$ then $snd' = snd$, $i = m$, $adr' = o = null$, and $u_{jp} = s'_j$.

□

Theorem 4.5. Let $\mathcal{C} = (S, M, ID, s_{in}, T)$ be a choreography with $ID = \{id_1, \ldots, id_n\}$. Let $\mathcal{S} = (\texttt{controlled'}(\mathcal{C}, id_1), \ldots, \texttt{controlled'}(\mathcal{C}, id_n), \texttt{orchestrator}(\mathcal{C}))$. For all $\texttt{conf}_x \in \{\texttt{conf}'_s, \texttt{conf}^{f\prime}_s\}$ we have $\mathcal{S}\ \texttt{conf}_x\ \mathcal{C}$. □

4.2. Decentralized derivation method

In this section we introduce our decentralized method to extract a choreography-compliant set of services from a given choreography. Let us note that, in general, the election of which branches are taken at choreography branching points is made by services according to their *local* information. Thus, the centralized solutions considered in the previous section are adequate only as long as we can assume that the orchestrator can access the information that would make each service take each possible choice. This might be the case of systems with strong security measures, as well as some intranet systems. This might also be the case in some specific parts of other larger service systems, which are globally decentralized but contain locally centralized subsystems. For instance, data base providers could be locally centralized in web systems using them, whereas subsystems such as inventory managers and payment gateways could be locally centralized elements of e-commerce service systems. Nevertheless, most of web service systems are mainly decentralized systems, at least at the highest hierarchy level. Therefore, we need an alternative derivation method where we do not assume that a centralized entity could monitor all variables affecting services elections at branching points.

Let us note that we can *remove* the orchestrator and distribute its responsibilities among the services themselves, thus making a decentralized solution. A choreography where, at all non-deterministic points, all available choices involve the decision of a single participant (i.e. all transition branches have the same sender) is a choreography where the decision-making is easy to handle: At each choreography state, the decision responsibility should be given to that service, and next all the other services should be consistent to that choice. The problem arises when a choreography has states where the next choice could be taken by *several* participants (i.e. available transition branches have different senders). Clearly, in this case the natural projection does not work, so a decision-making mechanism involving all services that could make the choice at

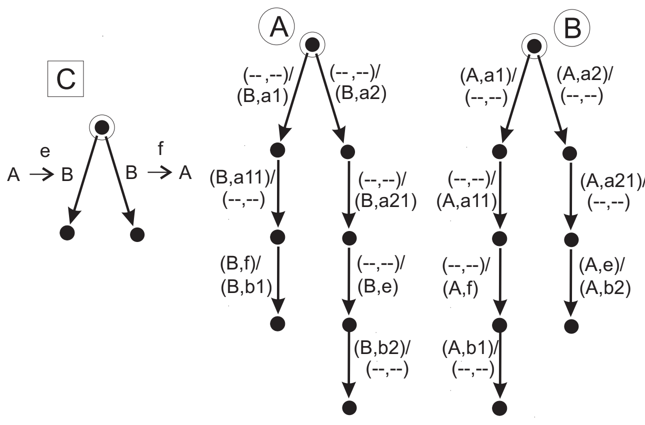

Figure 8: Derivation of services without orchestrator.

the current state must be designed. Let s be a choreography state with several outgoing transitions. Instead of using an orchestrator to choose which transition is taken, we do as follows: We sort all outgoing transitions by any criterion (e.g. by the name of the sender) and we make the first sender choose between (a) taking any of the transitions where it is the sender; or (b) refusing to do so. In case (a) it will announce its choice to the rest of services, thus playing the role of the orchestrator in this step. In case (b) it will notify its rejection to choose a transition to the second service. Then, the second service will choose either (a) or (b) in the same way, and so on up to the last sender, which will be forced to take one of its transitions if all previous senders refused to do so. Let us note that, in this alternative design, a service can receive the request to take a given branch from *several* services, not just from one special service (which was the *orchestrator* in the centralized method). Thus, new transitions have to be carefully added to services, which complicates their design.

An example of decentralized derivation is depicted in Figure 8. For the sake of simplicity, some transitions that would be part of derived services according to the formal derivation method (given next, in Definition 4.6) have been omitted. Choreography C represents a branching point where there are two possibilities: either A sends e to B, or B sends f to A. We derive services A and B from C as we have sketched above. Service A receives the responsibility of either taking one of the transitions where it is the sender (there is only one in this example) or refusing to do so. In the former case, it tells the next service in the list (B) that it will decide the transition indeed (message $a2$) and next it tells all services (i.e. just B) *which* of its transitions it will actually take ($a21$). Then, it sends e to B and waits for a signal indicating that B has processed the message ($b2$). In the latter case, i.e. if it refuses to choose one of its transitions, then it tells its decision to next service B (message $a1$) and waits for the rest of services (just B) to tell it which choice it must take. When B does so ($a11$), it waits for receiving b from B and next it acknowledges the reception $(b1)$. The behavior of B turns out to be dual to the behavior of A.

Let us formally present the derivation of decentralized systems of services from choreographies. As we did in the centralized cases, two alternatives are considered: Making the system conform to the choreography with respect to all proposed $\mathtt{conf}'_x$ conformance relations, and making it conform only with respect to sending traces. Theorems 4.7 and 4.9 show the correctness of both approaches.

Definition 4.6. Let $\mathcal{C} = (S, M, ID, s^1, T)$ be a choreography machine with $ID = \{id_1, \ldots, id_n\}$ and $S = \{s^1, \ldots, s^l\}$. For all $s \in S$ and $id \in ID$, let $T_{s,id} = \{(s, m, id, adr, s') | \exists\ adr, m, s' : (s, m, id, adr, s') \in T\}$ and $m_{s,id} = |T_{s,id}|$. For all $1 \leq j \leq m_{s,id}$, let $t_{s,id,j}$ denote the j-th transition of $T_{s,id}$ according to some arbitrary ordering criterium.

For all $s \in S$, let $[id^s_1, \ldots, id^s_{h_s}, id^s_{h_s+1}, \ldots, id^s_n]$ denote any arbitrary sequence of all identifiers in ID such that the sequence preserves the condition that for all $1 \leq d \leq h_s$ we have $m_{s,id^s_d} \geq 1$, and for all $h_s + 1 \leq d \leq n$ we have $m_{s,id^s_d} = 0$.

For all $1 \leq i \leq n$, the *decentralized service* for $\mathcal{C}$ and id_i, denoted $\mathtt{decentral}(\mathcal{C}, id_i)$, is a service $M_i = (id_i, S'_i, I'_i, O'_i, s_{in}, T_i, \{\{i\} \mid i \in \{I'_i\}\})$, where S'_i, I'_i, O'_i consist of all states, inputs, and outputs appearing in transitions described next and, for all $s^q \in S$, the following transitions are in T_i:

(BASIC CASE) Let $1 \leq k \leq n$ be such that $id_i = id^{s^q}_k$. We assume that $(id_i)^- = id^{s^q}_{k-1}$ and $(id_i)^+ = id^{s^q}_{k+1}$. We have the following transitions in T_i:

(a) $s^q \xrightarrow{((id_i)^-, idontchoose)/(null,null)} s^q_{icanchoose}$ *($(id_i)^-$ tells id_i that it refuses to choose one of its transitions).*

(b) For all $1 \leq y \leq n$ such that we have $id_y \in \{id^{s^q}_1, \ldots, (id_i)^-\} \cap \{id^{s^q}_1, \ldots, id^{s^q}_{h_{s^q}}\}$, we have $s^q \xrightarrow{((id_i)^-, alreadychosen_y)/((id_i)^+, alreadychosen_y)} s^q_{idontchoose}$ *($(id_i)^-$ tells id_i that some service y has already chosen, and id_i propagates the message).*

(c) $s^q_{icanchoose} \xrightarrow{(null,null)/((id_i)^+, idontchoose)} s^q_{idontchoose}$ *(id_i decides not to choose).*

(d) $s^q_{icanchoose} \xrightarrow{(null,null)/((id_i)^+, alreadychosen_i)} s^q_{iwillchoose}$ *(id_i decides to choose).*

(e) For all $1 \leq j \leq |T_{s^q, id_i}|$ we have the following transitions, where we assume $t_{s^q, id_i, j} = (s^q, m, snd, adr, s'^q)$.

(e.1) $s^q_{iwillchoose} \xrightarrow{(id^{s^q}_n, chosencomplete)/(adr, takemychoice_j)} s^q_{ichoose_j}$ *(the last service notifies that all know id_i will choose, and id_i chooses its j-th transition and asks adr to take its choice).*

(e.2) $s^q_{ichoose_j} \xrightarrow{(null,null)/(adr,m)} s^q_{ichoose'_j}$ *(id_i sends the message m denoted by its j-th transition to adr).*

(e.3) Let $G = \{g | g \in [1..n], g \neq i, id_g \neq adr\}$.

(i) If $G \neq \emptyset$ then we have $s^q_{ichoose'_j} \xrightarrow{(adr, ididit)/(null,null)} s^q_{ichoose_j\ min(G)}$ *(id_i waits for a signal from adr indicating that m was processed).* Besides, for all $k \in G$ we have

- $s^q_{ichoose_{jk}} \xrightarrow{(null,null)/(id_k, takemychoice_j)} v$, where $v = s^q_{ichoose_j\ k'}$ if $k \neq max(G)$ and $v = s'^q$ otherwise, and k' is the minimum value in G such that $k' > k$ *(id_i asks everybody to take its choice).*

(ii) Else (that is, if $G = \emptyset$) then we have $s^q_{ichoose'_j} \xrightarrow{(adr,ididit)/(null,null)} s'^q$ (id_i waits for a signal from adr indicating that m was processed, and id_i reaches the destination state without asking anybody else).

(f) For all $j \in [1..n]\backslash\{i\}$ and for all $1 \leq k \leq |T_{s^q,id_j}|$, we have the following transitions, where we assume $t_{s^q,id_j,k} = (s^q, m, snd, adr, s'^q)$.

(f.1) If $adr = id_i$ then we have $s^q_{idontchoose} \xrightarrow{(id_j,takemychoice_k)/(null,null)} s^q_{ifollow_{jk}}$ and $s^q_{ifollow_{jk}} \xrightarrow{(id_j,m)/(id_j,ididit)} s'^q$ *(id_i takes the k-th choice of id_j, which makes id_i receive a message from id_j and next acknowledge it).*

(f.2) Otherwise, we have $s^q_{idontchoose} \xrightarrow{(id_j,takemychoice_k)/(null,null)} s'^q$ *(id_i takes the k-th choice of id_j, which does not concern id_i).*

(OTHER CASES) Transitions listed in the basic case are modified in some specific cases as follows (modifications due to different cases are accumulative):

- If there are transitions leaving s^q in which id_i is the sender and id_i is the first service doing so, that is, if $id_i = id^{s^q}_1$, then transitions given in (a) and (b) of the basic case are replaced by $s^q \xrightarrow{(null,null)/(null,null)} s^q_{icanchoose}$.
- If there are transitions leaving s^q in which id_i is the sender and id_i is the last service doing so, that is, if $id_i = id^{s^q}_{h_{s^q}}$, then the transitions given in (c) of the basic case is deleted.
- If there is no transition leaving s^q in which id_i is the sender, that is if $id_i \neq id^{s^q}_j$ for all $1 \leq j \leq h_{s^q}$, then the transition given in (a) of the basic case is deleted.
- If $id_i = id^{s^q}_n$ (that is, id_i is the last service in the considered sequence of services) then, in any transition labeled by the pair $((id_i)^-, alreadychosen_y)/((id_i)^+, alreadychosen_y)$, this pair is replaced by the pair $((id_i)^-, alreadychosen_y)/(id_y, chosencomplete))$. Besides, in any transition labeled by $(null, null)/((id_i)^+, alreadychosen_i)$, this pair is replaced by $(null, null)/(id_i, chosencomplete)$, and the transition denoted in (c) of the basic case is deleted.

$\square$

Theorem 4.7. Let $\mathcal{C} = (S, M, ID, s_{in}, T)$ be a choreography with $ID = \{id_1, \ldots, id_n\}$. Let $\mathcal{S} = (\mathtt{decentral}(\mathcal{C}, id_1), \ldots, \mathtt{decentral}(\mathcal{C}, id_n))$. For all $\mathtt{conf}_x \in \{\mathtt{conf}'_s, \mathtt{conf}'_p, \mathtt{conf}', \mathtt{conf}^{f\prime}_s, \mathtt{conf}^{f\prime}_p, \mathtt{conf}^{f\prime}\}$ we have $\mathcal{S}\ \mathtt{conf}_x\ \mathcal{C}$. $\square$

Definition 4.8. We have that $\mathtt{decentral'}(\mathcal{C}, id_i)$ is defined as $\mathtt{decentral}(\mathcal{C}, id_i)$ in Definition 4.6 after replacing the first transition appearing in (e.3) (i) by $s^q_{ichoose'_j} \xrightarrow{(null,null)/(null,null)} s^q_{ichoose_{j\ min(G)}}$, the transition of (e.3) (ii) by $s^q_{ichoose'_j} \xrightarrow{(null,null)/(null,null)} s'^q$, and the second transition denoted in (f.1) by $s^q_{ifollow_{jk}} \xrightarrow{(id_j,m)/(null,null)} s'^q$. $\square$

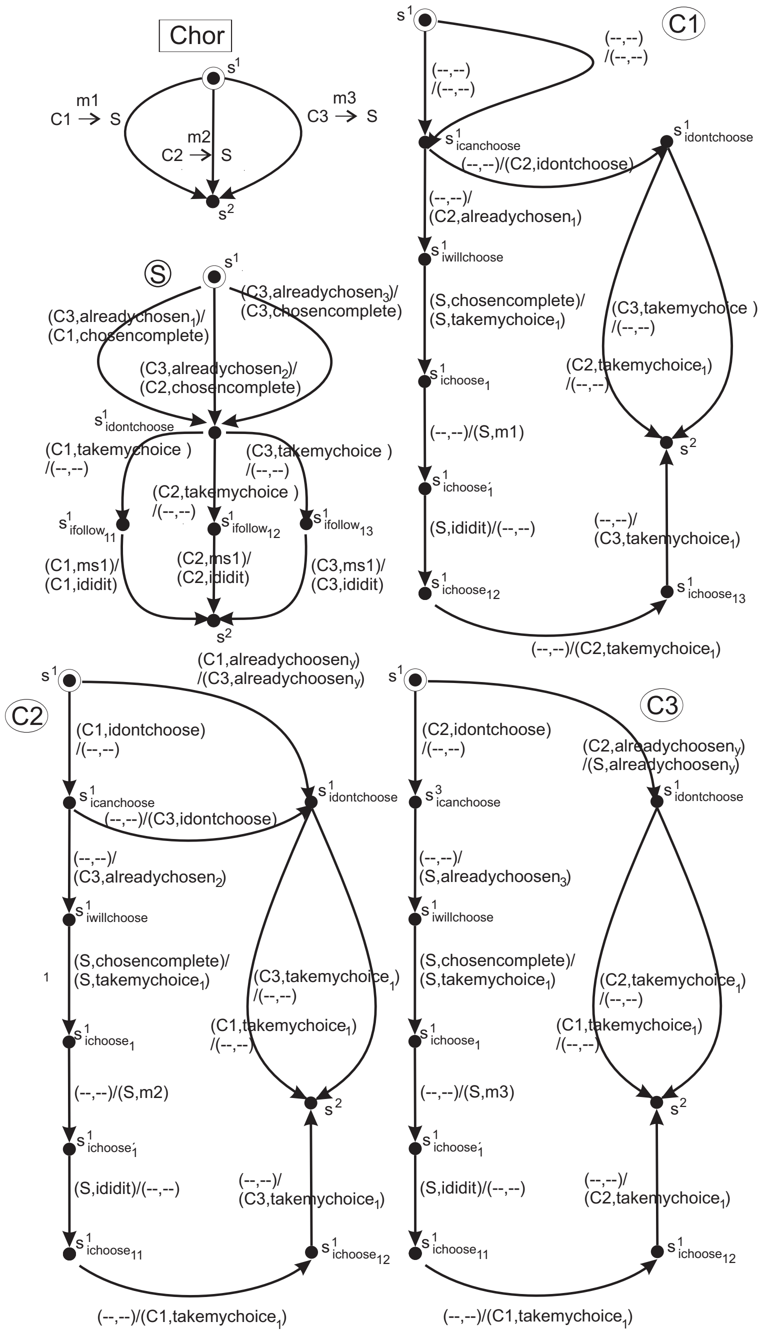
Chor
s1
m1 C1 → S
m2 C2 → S
m3 C3 → S
s2
S
s1
(C3,alreadychosen1)/(C1,chosencomplete)
(C3,alreadychosen2)/(C2,chosencomplete)
(C3,alreadychosen3)/(C3,chosencomplete)
s1 idontchoose
(C1,takemychoice)/(--,--)
(C2,takemychoice)/(--,--)
(C3,takemychoice)/(--,--)
s1 ifollow11
s1 ifollow12
s1 ifollow13
(C1,ms1)/(C1,ididit)
(C2,ms1)/(C2,ididit)
(C3,ms1)/(C3,ididit)
s2
C1
s1
(--,--)/(--,--)
(--,--)/(--,--)
s1 icanchoose
(--,--)/(C2,idontchoose)
s1 idontchoose
(--,--)/(C2,alreadychosen1)
s1 iwillchoose
(S,chosencomplete)/(S,takemychoice1)
(C3,takemychoice)/(--,--)
(C2,takemychoice1)/(--,--)
s1 ichoose1
(--,--)/(S,m1)
s2
s1 ichoose'1
(S,ididit)/(--,--)
(--,--)/(C3,takemychoice1)
s1 ichoose12
s1 ichoose13
(--,--)/(C2,takemychoice1)
(C1,alreadychoosenу)/(C3,alreadychoosenу)
C2
s1
(C1,idontchoose)/(--,--)
s1 icanchoose
(--,--)/(C3,idontchoose)
s1 idontchoose
(--,--)/(C3,alreadychosen2)
s1 iwillchoose
(S,chosencomplete)/(S,takemychoice1)
1
(C3,takemychoice1)/(--,--)
(C1,takemychoice1)/(--,--)
s1 ichoose1
(--,--)/(S,m2)
s2
s1 ichoose'1
(S,ididit)/(--,--)
(--,--)/(C3,takemychoice1)
s1 ichoose11
s1 ichoose12
(--,--)/(C1,takemychoice1)
C3
s1
(C2,idontchoose)/(--,--)
(C2,alreadychoosenу)/(S,alreadychoosenу)
s3 icanchoose
s1 idontchoose
(--,--)/(S,alreadychoosen3)
s1 iwillchoose
(S,chosencomplete)/(S,takemychoice1)
(C2,takemychoice1)/(--,--)
(C1,takemychoice1)/(--,--)
s1 ichoose1
(--,--)/(S,m3)
s2
s1 ichoose'1
(S,ididit)/(--,--)
(--,--)/(C2,takemychoice1)
s1 ichoose11
s1 ichoose12
(--,--)/(C1,takemychoice1)

Figure 9: Example of decentralized derivation.

Theorem 4.9. Let $\mathcal{C} = (S, M, ID, s_{in}, T)$ be a choreography with $ID = \{id_1, \ldots, id_n\}$. Let $\mathcal{S} = (\texttt{decentral'}(\mathcal{C}, id_1), \ldots, \texttt{decentral'}(\mathcal{C}, id_n))$. For all $\texttt{conf}_x \in \{\texttt{conf}'_s, \texttt{conf}^{f'}_s\}$ we have $\mathcal{S}\ \texttt{conf}_x\ \mathcal{C}$. $\square$

In Figure 9 we show an example where the derivation presented in Definition 4.6 is applied literally. Choreography $Chor$ consists of three different branches where a message (m_1, m_2, or m_3) is sent by a client service ($C1$, $C2$, or $C3$, respectively) to a server service S. The derivation of the service client $C2$ corresponds to the general (BASIC CASE) of this definition, whereas $C1$ and $C3$ are obtained by applying the first and the second items of (OTHER CASES), corresponding to the first and the last service sending a message in the choice structure, respectively. Finally, the server service S follows the structure specified for services that do not send messages. Moreover, S plays the role of last service in the service sequence. Thus, the last two items of (OTHER CASES) are applied in this case.

Both the centralized and the decentralized derivation algorithms add a high number of additional messages and constrain the free advance of services for the sake of control. Let us note that, in this paper, our goal is not to provide the optimal solution, that is, the solution where the parallel advance of services is restricted as weakly as possible or the minimum number of additional control messages is added. On the contrary, our goal is providing derivation algorithms to construct sets of services that are correct with respect to the choreography, regardless of whether the designer of the choreography created a nice choreography or, on the contrary, it contains some intrinsic problems. Cutting some additional messages to provide more optimal derivations is out of the scope of this paper and is left as future work.

5. Derivation of services under the presence of delayed messages

In this section we consider an alternative semantic scenario for our framework. Let us note that, according to the operational semantics of our systems of services, given in Definition 2.4, when a service sends a message, this message is immediately stored in the input buffer of the addressee of the message. Let us suppose that a service A sends message m_1 to service B, and next it sends message m_2, also to service B. Since the operational semantics says that messages are stored immediately at the destination service, it is impossible that service B receives message m_2 and next it receives message m_1, which would make m_2 appear before m_1 in the input buffer of service B. Thus, the framework implicitly assumes one of the following hypothesis: Either message delays in the communication medium are always the same (which is unfeasible in practice), or the network protocol implicitly manages, at some lower implementation layer, a proper reordering of messages allowing to keep the order in which messages were sent by each client. For instance, in some cases, time stamps or ordering stamps can be added to messages to enable this implicit ordering [24]. However, in some cases this solution might not be feasible due to e.g. the impossibility to have a global clock.

In this section we consider an alternative scenario where, when a message is sent, it is not immediately stored at the destination service. On the contrary, the message may stay “in the communication medium” for any arbitrarily long time. In order to introduce this alternative scenario, we will assume by default all definitions previously given in Section 2, though some of them will have to be redefined. Next, we redefine the notion of system configuration. A configuration of a system will depend not only on the configuration of each service, but also on the multiset of messages that have already been sent by services but have not reached their destination yet. We will denote this multiset by D. We consider that $(id, m, id') \in D$ denotes that id sent m to id', but id' has not received it yet (so, m is not stored in the input buffer of id' yet, m is still in the communication medium).

Definition 5.1. *(Redefinition of system configuration)* Let $\mathcal{S} = (M_1, \ldots, M_p)$ be a system of services for ID, where for all $1 \leq j \leq p$ we have $M_j = (id_j, S_j, I_j, O_j, s_{j,in}, T_j, \psi_j)$. For all $1 \leq j \leq p$, let c_j be a configuration of M_j. Let D be a multiset of triples belonging to $ID \times (O_1 \cup \ldots \cup O_p) \times ID$. We say that $c = (c_1, \ldots, c_p, D)$ is a *configuration* of $\mathcal{S}$. Let $c'_1, \ldots, c'_p$ be the initial configurations of $M_1, \ldots, M_p$, respectively. Then, $(c'_1, \ldots, c'_p, \emptyset)$ is the *initial configuration* of $\mathcal{S}$. □

Next we redefine the operational semantics of systems of services for the alternative scenario. Now, when a service sends a message, it is not inserted in the input buffer of the addressee, but it is added to the multiset of "not-yet received" messages D. Besides, the operational semantics also allows to take a triple (id, m, id') from D and store (id, m) at the input buffer of id'. Thus, the operational semantics splits any message sending into two separate semantic actions, thus letting other actions happen between both.

Definition 5.2. *(Redefinition of the operational semantics)* Let $ID = \{id_1, \ldots, id_p\}$ be a set of service identifiers and $\mathcal{S} = (M_1, \ldots, M_p)$ be a system of services for ID where for all $1 \leq j \leq p$ we have $M_j = (id_j, S_j, I_j, O_j, s_{j,in}, T_j, \psi_j)$. Let $c = (c_1, \ldots, c_p, D)$ be a configuration of $\mathcal{S}$ where for all $1 \leq j \leq p$ we have $c_j = (s_j, b_j)$.

An *evolution* of $\mathcal{S}$ from the configuration c is a tuple $(c, snd, i, proc, o, adr, c')$ where $i \in I_1 \cup \ldots \cup I_p \cup \{null\}$ is the input of the evolution, $o \in O_1 \cup \ldots \cup O_p \cup \{null\}$ is the output of the evolution, $c' = ((s'_1, b'_1), \ldots, (s'_p, b'_p), D')$ is the new configuration of $\mathcal{S}$, and $snd, proc, adr \in ID \cup \{null\}$ are the sender, the processor, and the addressee of the evolution, respectively. All these elements must be defined according to one of the following choices:

(a) *(evolution activated by some service by itself)* For some $1 \leq j \leq p$, let us suppose $s_j \xrightarrow{(null,null)/(adr',o')} s' \in T_j$. Then, $s'_j = s'$ and $b'_j = b_j$. Besides, $snd = null$, $proc = id_j$, $adr = adr'$, $i = null$, $o = o'$. Moreover, if $adr' \neq null$ then $D' = D \cup (id_j, o', adr')$;

(b) *(evolution activated by processing a message from the input buffer of some service)* For some $1 \leq j \leq p$, let us suppose that $s_j \xrightarrow{(snd',i')/(adr',o')} s' \in T_j$ and the predicate $\texttt{available}(b_j, snd', i', r)$ holds, where r is the only set belonging to ψ_j such that $i' \in r$. Then, $s'_j = s'$ and $b'_j = \texttt{remove}(b_j, snd', i')$. Besides, $snd = snd'$, $proc = id_j$, $adr = adr'$, $i = i'$, $o = o'$. Moreover, if $adr' \neq null$ then $D' = D \cup (id_j, o', adr')$;

(c) *(evolution activated by the reception of some message stored in the multiset of not-yet received messages)* For some $1 \leq j \leq p$ and $1 \leq c \leq p$, if $(id_c, m, id_j) \in D$ then $snd, proc, adr = null$, $s'_j = s_j$, and $b'_j = \texttt{insert}(b_j, id_c, m)$. Moreover, $D' = D \backslash (id_c, m, id_j)$.

where, in any of these cases (a), (b), and (c), for all $1 \leq q \leq p$ with $q \neq j$ we have $s'_q = s_q$ and $b'_q = b_q$. □

Next, let us analyze the (in-)correctness of the centralized and decentralized derivation methods presented in sections 4.1 and 4.2, respectively, under this alternative semantics.

It is easy to see that the centralized version, given in Definition 4.2, does not work under the new semantics. In this derivation, the orchestrator sends messages a_{jp} to all services to indicate that all services must take the p-th available transition at state s_j. When the orchestrator receives the message b_f, indicating that the addressee id_f of the message regarded in the current step of the choreography has processed the message, the orchestrator starts the next step of the choreography. It makes the next election, communicates its choice to all services, and so on. However, nothing guarantees that messages a_{jp} indicating the choice to be followed at the previous step will be received by services *before* messages a_{jp} indicating the choice at the new step. If a message a_{jp} of the next step arrives at a service before the message a_{jp} of the previous step, the service will take a wrong transition at the current step.

This problem can be easily fixed by making the orchestrator receive some new messages from all services, where these messages indicate that the corresponding service has already received the message a_{jp} of the current step. If the orchestrator is forced to receive all of these acknowledgment messages before going on to the next step, then messages a_{jp} of the next step will be sent by the orchestrator *only after* messages a_{jp} of the previous step have been processed by services. Thus, messages a_{jp} of each step will be necessarily processed before that step finishes, that is, it will not be possible that a message a_{jp} from a subsequent step is received before another message a_{jp} from a previous step. In the centralized derivation given in Definition 4.2, the only service forced to send an acknowledgment b_f to the orchestrator was the addressee id_f of the sending. In the next redefined derivation, the orchestrator will be forced to collect these acknowledgment messages from *all* services – and all services will be forced to send these messages to the orchestrator.

Definition 5.3. We have that $\texttt{controlledDelays}(\mathcal{C}, id_i)$ is defined as $\texttt{controlled}(\mathcal{C}, id_i)$ after replacing cases (a), (b), and (c) of Definition 4.2 by the following expressions *(now, messages b_f are sent in all cases)*:

(a) if $snd = id_i$ then $snd' = i = null$, $adr' = adr$, $o = m$, and $u_{jp} = s'_{jp}$. Besides, we also have $s'_{jp} \xrightarrow{(null,null)/(orc,b_i)} s'_j$ in T_i.

(b) else, if $adr = id_i$ then $snd' = snd$, $i = m$, $adr' = o = null$, and $u_{jp} = s'_{jp}$. Besides, we also have $s'_{jp} \xrightarrow{(null,null)/(orc,b_i)} s'_j$ in T_i.

(c) else $snd' = i = adr' = o = null$ and $u_{jp} = s'_{jp}$. Besides, $s'_{jp} \xrightarrow{(null,null)/(orc,b_i)} s'_j$ is in T_i.

Besides, the last term of the tuple defining $\texttt{controlledDelays}(\mathcal{C}, id_i)$, which is equal to $\{\{m\}|m \in M\} \cup \{\{a_{ij}|i, j \in [1..l]\}\}$ in the definition of $\texttt{controlled}(\mathcal{C}, id_i)$ of Definition 4.2, is replaced by $\{\{m\}|m \in M\} \cup \{\{a_{ij}\}|i, j \in [1..l]\}$ *(each message a_{ij} has its own message type)*.

We have that $\texttt{orchestratorDelays}(\mathcal{C}, id_i)$ is defined as $\texttt{orchestrator}(\mathcal{C}, id_i)$ after replacing the second item of the definition of such a term in Definition 4.2 by the following expression *(messages b_f are collected from all services, not just from the addressee)*:

- For all $1 \leq p \leq k$, if $t_p = s_j \xrightarrow{m/(snd \rightarrow adr)} s'_j \in T$ is the p-th transition leaving s_j in $\mathcal{C}$, then for all $1 \leq i \leq n$ we have $s_{jpi} \xrightarrow{(null,null)/(id_i,a_{jp})} s_{jp\,i+1} \in T_o$. Besides, for all

$1 \leq i \leq n-1$ we have $s_{jp\ n+i} \xrightarrow{(id_i,b_i)/(null,null)} s_{jp\ n+i+1} \in T_o$. In addition, we have $s_{jp\ n+n} \xrightarrow{(id_n,b_n)/(null,null)} s'_j \in T_o$.

□

Theorem 5.4. Let us assume that the behavior of systems of services is defined by the operational semantics given in Def. 5.2. Let $\mathcal{C} = (S, M, ID, s_{in}, T)$ be a choreography with $ID = \{id_1, \ldots, id_n\}$. Let $\mathcal{S} = (\texttt{controlledDelays}(\mathcal{C}, id_1), \ldots, \texttt{controlledDelays}(\mathcal{C}, id_n), \texttt{orchestratorDelays}(\mathcal{C}))$. For all conformance relationships $\texttt{conf}_x \in \{\texttt{conf}'_s, \texttt{conf}'_p, \texttt{conf}', \texttt{conf}^{f\prime}_s, \texttt{conf}^{f\prime}_p, \texttt{conf}^{f\prime}\}$ we have $\mathcal{S}\ \texttt{conf}_x\ \mathcal{C}$. □

If we can assume that messages are stored in input buffers in the same order as they are sent, that is, if we can assume the old operational semantics given in Definition 2.4, then the previous solution is unnecessarily inefficient. The previous solution forces the orchestrator to receive acknowledgments from all services before going on, which forces all services to reach the current choreography step before the orchestrator goes on to the next step. This feature reduces the capability of services to evolve independently from the rest of services, and thus reduces the proportion of computations that are actually executed in parallel. However, this high level of control is not necessary if messages cannot be mixed up in input buffers. Thus, if the old operational semantics is assumed, then the old centralized derivation given in Definition 4.2 is a better choice.

Now, let us analyze the correctness of the decentralized derivation, given in Definition 4.6, under the new alternative operational semantics. Let us note that this decentralized derivation *already* imposes a kind of strong control that is similar to the one described in our previous redefined centralized derivation. In particular, let us note that all services are required to know which service will be responsible of taking the current choice *before* that service tells the rest of services which one is its choice (see in Definition 4.6 that, before going on, the service deciding must receive a message $chosencomplete$ from the last service of the sequence). Let us note that this strong level of control is required *even* if messages are not mixed up in input buffers, that is, even if our original operational semantics is assumed. If the service making the decision were not required to be sure that all services know that it will make the choice, then messages denoting which service decides at two consecutive choreography steps could be mixed up: The message announcing which service makes the decision in the step $i+1$ could reach a given service s before the message announcing which service makes the decision of the step i is received by s. Messages denoting which choice is taken by the service making the choice ($takemychoice_j$) are sent by that service one after each other, and they are sent before the corresponding $takemychoice_{j'}$ messages of the next step are sent. So, messages of this kind are sent in the correct order (and stored in the correct order, if the old semantics is assumed). However, messages denoting *which* service decides (i.e. $idontchoose$, $alreadychosen_y$) are sent by *all* services, one service after the other. Thus, the order in which messages of this kind *belonging to different steps* are sent could be mixed up – unless a message like $chosencomplete$ blocks the sending of these messages in the next step until the last message of this kind is sent in the previous step.

The use of the message $chosencomplete$ does not only solve the problem for the old semantics, but also for the new one. On the one hand, $takemychoice_j$ messages from different steps cannot be mixed up, because services must process their $takemychoice_j$ messages before they can participate in the decision about which service chooses in the next step, and this decision must be taken before messages $takemychoice_j$ of the next step are sent. By similar reasons, messages used to decide which service must choose (i.e. $idontchoose$, $alreadychosen_y$) of different steps cannot be mixed up: Messages of this kind belonging to the previous step must be processed before the corresponding messages of the next step are sent. Thus, it turns out that our original decentralized derivation works for the new operational semantics too.

Theorem 5.5. Let us assume that the behavior of systems of services is defined by the operational semantics given in Definition 5.2. Let $\mathcal{C} = (S, M, ID, s_{in}, T)$ be a choreography with $ID = \{id_1, \dots, id_n\}$. Let $\mathcal{S} = (\texttt{decentral}(\mathcal{C}, id_1), \dots, \texttt{decentral}(\mathcal{C}, id_n))$. For all conformance relationships $\texttt{conf}_x \in \{\texttt{conf}'_s, \texttt{conf}'_p, \texttt{conf}', \texttt{conf}^{f\prime}_s, \texttt{conf}^{f\prime}_p, \texttt{conf}^{f\prime}\}$ we have $\mathcal{S}\ \texttt{conf}_x\ \mathcal{C}$. □

6. Discussion: features beyond the current model

In this section we discuss some features of real web services systems that are not explicitly represented in our current model of orchestrations and choreographies. Though introducing these factors in our model is part of our future work plans, in this section we will sketch some ways to take some of these factors into account *without* modifying either the model itself or the derivation algorithms. That is, the approaches described in this section will be conservative with our models. The modification of models to explicitly manage these factors will be developed in our future work.

Several factors affecting the behavior of real web services systems are not explicitly represented in the model, in particular: (a) message parameters and internal variables of services; (b) the effect of *time* on services; (c) the possibility that the information required to make a decision might not be owned by the service that makes the decision; and (d) the presence of external events. Extending the model to include factors (a) and (b) requires replacing our FSM-based model by a more expressive one, such as *extended finite state machines* (EFSM) or *timed automata* (TA), respectively. Changing our FSM-based models by these models will require a similar effort as other similar language extensions developed in other works of the literature, where variables or time were added to previously developed simpler specification languages. The extensions required to include (a) and (b), while somehow standard, will probably be cumbersome in technical terms, so discussing them is out of the scope of this paper.

Regarding (c), that is, the possibility that the information required to make each decision might not be owned by the services that make such decisions, let us mention that the absence of an (explicit) representation of this factor in our model is related to the absence of (b). If models were endowed with internal variables and message parameters, internal variables of services could be used to affect the availability of transitions by enabling/disabling transition guards, and the values of these variables could be transmitted from a service to another one by sending messages with parameters. Thus, the transmission of information affecting decisions, from the services owning this information to the services requiring this information, could be naturally represented by inserting, in the choreography, suitable messages from the former to the latter.

Still, the transmission of such information can be modeled to some extent in our simpler FSM-based model too. For instance, a given service A may be at *different* states, depending on the information we assume is *owns*. Let us assume that this information can be either x or y. Since service A might be in a different state in each case, it can send different messages to services depending on whether this information is x or y. In particular, depending on whether service A sends a *message* x or y to another service B, different choreography states, where service B has different available choices next, can be reached. Thus, the information owned by A and transmitted by A to B (i.e. either message x or message y) can affect the set of choices available later to service B. Following this idea, the choreography designer can include messages from services *owning* the required information to services requiring this information, where each message enables different subsequent choices.

The convenience to keep a low number of states in the choreography could limit in practice the creation of bifurcations to denote the choices of services depending on the received information. For instance, we may explicitly represent the value owned by service A (either x or y) by using two different states of service A or, alternatively, we could *abstract* this information and consider a single state in A. In this case, service A *non-deterministically* communicates to B that its value is x or y. The non-determinism just denotes that both choices are *possible*. In practice, the choreography designer could be forced to introduce some level of abstraction in models for the sake of model simplicity.

This limitation will be overcome when the model is moved from FSMs to EFSMs in our future work. The localization of the information and the dependance on it will be explicitly represented by means of *local* variable values and transition guards potentially depending on variables stored by *other* services. Following the derivation policy proposed in this paper, new adapted algorithms will be able to *automatically* manage the transmission of the information from the services where it is stored to the services where it is needed. If the choreography includes some decisions that depend on some variables that are not stored by the services that will actually make them, the derivation algorithms will automatically add some new control messages in derived services. These control messages will make services owning such information *send it* to services depending on it, before decisions are actually made. In this way, we will be consistent with our derivation policy, where *all* choreographies denote an interaction plan that can be realized – provided that suitable control messages are added to services.

Regarding (d), the presence of external events, let us note that our model considers a closed world assumption, i.e. the system of services does not interact with any external environment and all exchanged messages are produced by services inside the system. Thus, there is no explicit reference to *external* events. In order to represent an *external event source*, capable of producing these events, we could explicitly model it by means of another service in the choreography. Modeling event sources as *services* in choreographies might be a good approach to the notion of event source, but it might not be suitable in terms of the derivation algorithms. The derivation makes all services involved in each decision coordinate with each other, either in terms of the orchestrator (in the centralized derivation) or in terms of the services themselves, which send messages to each other (in the decentralized derivation). Neither of both choices can be applied to an *external event source*, because these sources are not *real* services and thus we cannot *design* them to comply to a given communication protocol established by the derivation algorithm (in particular, an external events source will not participate in the token-ring decision process). An alternative possibility consists in adding a new service representing an *external event interface*. This service receives all external events, delivers them, and coordinates itself with the rest of services as any other service. This solution might be feasible from the point of view of a centralized derivation, because the orchestrator may assume the behavior of this external event interface (note that this role would be consistent with its implicit omniscience). On the contrary, it would not be an appropriate choice for the decentralized derivation method, because it violates the decentralized approach.

A more natural approach to external events consists in assuming that events are just *not* explicitly represented in the model. In fact, the existence of external events in *real* services motivates (part of) the non-deterministic choices of services in the *model*. That is, a state where a real service may take either choice A (if it receives a kind of event) or choice B (if it receives another kind of event, or no event at all) is modeled by the existence of two outgoing transition where the service chooses either of these choices. The reason to take A or B is not represented in the model due to the abstraction of the model: External events may make services take different choices in *real* services, so these choices *exist*, as different possibilities, in the corresponding *models*.

According to this view, an issue must still be addressed. In the decentralized derivation, services involved in each decision have the choice to either take any of the choices where the service chooses, or passing the responsibility of choosing to the next service. In a real service, the decision of whether the responsibility to choosing should be passed to the next service or not could depend, in some cases, on the *absence* of some external event that could be received by the service or not (for instance, the service *passes* the decision token if it has not received the required event to take one of its transitions yet). If *all* services in the decision-making process may pass the responsibility due to the absence of some required event, then the last service in the sequence should also be given the choice to pass. Unfortunately, the derivation forces the last service to take some of its choices if previous services have not chosen yet.

If no service chooses due to the absence of the required external events to do so, then the decision-making should be *repeated* to give all services a new chance to receive the events they require, until some of these events is eventually received by the corresponding service (so it no longer *passes* the decision responsibility to the next service). That is, if a given decision-making process depends on external events that enable/disable services choices, the decision-making processes should have the capability to *loop* among derived services. We have two choices to introduce this change. On the one hand, we could modify the decentralized derivation method to explicitly replace some decision-making *sequences* by decision-making *loops* (in particular, those decision-making processes potentially depending on the *absence* of external events). Let us note that the introduction of loops is motivated by external events, which are not represented in our current models. Since this new derivation would be useful only if factors beyond the current model are considered (external events), we prefer to keep our decentralized derivation method unmodified, and construct the new models with loops by means a new *abstraction layer*. In this way, the (unmodified) decentralized derivation algorithm will still be motivated by the sole goal of achieving correct systems in terms of the model semantics (where external events are not explicitly represented), and details beyond the model will be treated in a different layer.

This new layer works as follows. Given a choreography, we identify those states where decision-making sequences should be converted into decision-making loops (due to the potential absence of external events governing some services choices in *real* services). Let us call these states *loop states*. For each of these states, we identify the service that would be the last one in the decision-making sequence, according to our derivation algorithm. Let us call these services *last services*. Similarly, the first services in the decision-making sequences will be called just *first services*. We construct an *intermediate model*, defined in the same language as the choreography, which has the same states and transitions as the choreography. In addition, for all *loop states* and their respective *last* and *first services*, we add a transition from the loop state to itself where the last service sends a new control message, called *repeat*, to the first service (an example is depicted in Figure 10, where that message is called just r). In this way, one of the choices of the last service of each decision-making sequence consists in *repeating* the decision-making sequence, thus enabling a loop. If the last service takes this new choice then we interpret, in terms of the corresponding *real* service, that the service cannot take any of the other transitions

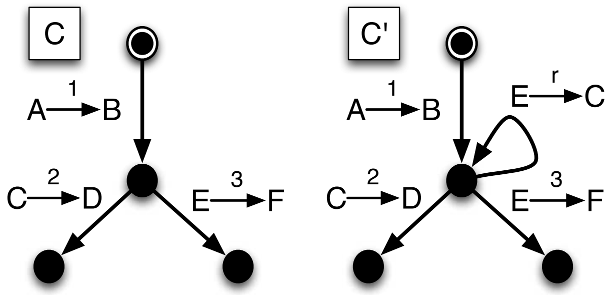

Figure 10: Intermediate model example.

where it is the sender due to the absence of some required external event.

By applying the (unmodified) decentralized derivation to this *intermediate model*, rather than to the choreography, the required loops will be introduced as a result of the new transitions added to the intermediate model. If we compare the systems of services derived from the choreography and the system derived from the intermediate model, it is easy to check that they are *not* equivalent in terms of traces of non-control messages. Let us suppose that state s of the choreography has several outgoing transitions involving different services. No trace of the choreography finishes at state s (recall that we only consider *complete* traces). Consequently, if a system of services is derived from this choreography, then no trace of this system finishes at s. Let us construct an intermediate model, from this choreography, where the possibility of looping in s is added as explained. If a system of services is derived from the intermediate model, then the system will be able to produce a trace where the system reaches s and loops forever. After s is reached, this infinite trace shows only control messages. In particular, control messages where services coordinate the decision-making process are followed by the new control message *repeat*, sent by the last service to the first service, and this process is repeated forever. If we remove control messages from this trace then all messages after s is reached for the first time are removed, and thus the remaining trace of non-control messages *finishes* at s. Hence, the system of services derived from the intermediate model is not equivalent to the system derived from the choreography. Still, the system derived from the intermediate model allows us to reduce the gap between real systems and our models (without modifying the model itself) by introducing the required loops. In the next definition, we consider that a state is *loop-needed* if a loop should be introduced in its decision-making – according to criteria not considered in the model semantics, such as the dependance on external events. We assume that each loop-needed state has at least *two* services with the capability of deciding in its decision-making sequence. Note that, in states where only *one* service decides, no loop is needed to give all deciding services new chances to choose (in particular, this single deciding service can just *not* take any of its choices, until it can choose indeed).

Definition 6.1. Let $\mathcal{C} = (S, M, ID, s_{in}, T)$ be a choreography and $S^* \subseteq S$ be a set of *loop-needed states*. The *intermediate model* of $\mathcal{C}$ and S^*, denoted by $\texttt{intermediate}(\mathcal{C}, S^*)$, is a choreography machine $\mathcal{C}' = (S, M \cup \{repeat\}, ID, s_{in}, T')$ where

$$T' = T \cup \left\{ s \xrightarrow{repeat/(b \to a)} s \;\middle|\; \begin{array}{l} s \in S^* \ \wedge \ a \text{ and } b \text{ are the first and last services in the} \\ \text{decision-making of state } s\text{, respectively, according to} \\ \text{the decentralized derivation } \wedge\, a \neq b \end{array} \right\}$$

□

Definition 6.2. Let $\mathcal{C} = (S, M, ID, s_{in}, T)$ and $\sigma \in \texttt{Traces}(\mathcal{C})$. The *constrain* of σ to a set of messages Q, denoted by σ^Q, is the result of removing from σ all elements (a, m, b) with $m \notin Q$. □

The following straightforward result establishes the relation between choreographies and their corresponding intermediate models (provided that we skip, in traces of the intermediate model, all added $repeat$ messages).

Proposition 6.3. Let $\mathcal{C} = (S, M, ID, s_{in}, T)$ be a choreography and $S^* \subseteq S$. We have that $\texttt{Traces}(\mathcal{C}) \subseteq \{\sigma^M | \sigma \in \texttt{Traces}(\texttt{intermediate}(\mathcal{C}, S^*))$. □

7. Related Work

In this section we compare our proposal with other approaches. Regarding methods to derive services from a given choreography, we can find some related works. In [29], Zongyan et al. identify and face the problems appearing when deriving an implementable projection from a choreography. Authors define the concept of restricted natural choreography, which must fulfill two structural conditions, and show that this kind of choreography is easily implementable. Furthermore, a new concept, the *dominant role* of a choice, is proposed for dealing with projection issues in non-restricted choreographies. At each non-deterministic choice, this dominant role is the one that actually makes the decision. The first difference between this work and our proposal is superficial and lies in the underlying model: [29] uses a process algebraic notation while we use a state machine approach. However, there are also two crucial differences. On the one hand, the orchestration communication style is *synchronous* in that work, while we consider asynchronous communications and delays, which complicates the problem. On the other hand, the solution of the non-deterministic choices problem considered in [29] is based on *explicitly* adding extra information to the choice operator by identifying the dominant role, and this must be given as part of the choreography specification. In particular it is assumed that, for each non-deterministic choice, we can always identify a dominant role that, by design definition, is the one which owns the information to decide what branch the implementation should follow. For instance, in the only branching point appearing in choreography $Chor$ (see Figure 1), the only reasonable candidate to be the dominant role is service Y, because in this choice it is the only service that has the opportunity to send messages to other services. Therefore this service is the only one really involved in the choice. Let us suppose that we modify, in $Chor$, the label of the transition where Y sends c to Z. In particular, let us assume that the service sending c to Z is not Y, but X. Now there are two services involved in the choice and it is not possible to know "a priori" which one is going to play the dominant role, so a kind of coordination between both services must be externally imposed. Our centralized and decentralized approaches to derive services from a choreography face this problem *at orchestration level* (not at the choreography level) by either allowing the orchestration to make the decision (in the centralized version) or by distributing this responsibility among all the services that are actually involved in the non-deterministic choice (in the decentralized version). In addition, we face the non-deterministic choices problem and the races problem in a single integrated framework (both problems are treated separately in [29]), we allow to explicitly distinguish between the times when messages are sent and the times when they are processed in our two approaches (this distinction is possible because we consider asynchronous communication), and we provide correctness proofs of our methods (not given in [29]).

The issue of investigating how we can design asynchronous communicating processes, in such a way that they necessarily produce some behavior or reach some configuration, has been tackled in several ways in the literature. For instance, [21] studies the problem of designing two asynchronous processes in such a way that their progress is guaranteed, whereas [17] studies the pathological situations where we cannot define some communicating processes conforming to a given specification (due to the relevance of the problems identified in this work, the treatment of these problems in our framework is extensively studied at the end of this section). Let us note that, in our approach, we make *any* choreography realizable by *adding* some control messages to the definition of services. These messages allow services to know what is required at each time to properly make the next decision, according to the choreography. In [32], Salaün and Bultan formalize choreographies by means of asynchronous communication with process algebra. However, no solution for non-deterministic choices is provided and no correctness proof is presented. In contrast, authors enhance the proposal by introducing a tool offering the possibility to use bounded buffers and reason about them. Van der Alst et al. [38] present an approach for formalizing compliance and refinement notions, which are applied to service systems specified using open Workflow Nets (a type of Petri Nets) where the communication is asynchronous. Authors show how the contract refinement can be performed independently, and they check whether contracts do not contain cycles. Honda et al. [22] present a generalization of binary session types to multiparty sessions for π-calculus. They provide a new notion of types which can directly abstract the intended conversation structure among n-parties as *global scenarios*, retaining an intuitive type syntax. They also provide a consistency criteria for a conversation structure with respect to the protocol specification (contract), and a type discipline for individual processes by using a *projection*. A similar approach is followed in [15] by Caires and Vieira. They define a formal framework called conversation types and present techniques to ensure progress of systems involving several interleaved conversations/sessions. Bravetti and Zavattaro [9] allow to compare systems of orchestrations and choreographies by means of the *testing* relation given by [6, 18]. Systems are represented by using a process algebraic notation, and operational semantics for this language are defined in terms of labeled transitions systems. On the contrary, our framework uses an extension of *finite state machines* to define orchestrations and choreographies, and a semantic relation based on the *conformance* relation [33, 34] is used to compare both models. In addition, let us note that [9] considers the suitability of a service for a given choreography *regardless* of the actual definition of the rest of services it will interact with, i.e. the service must be valid for the considered role *by its own*. This eases the task of finding a suitable service fitting into a choreography role: Since the rest of services do not have to be considered, we can search for suitable services for each role *in parallel*. However, let us note that sometimes this is not realistic. In some situations, the suitability of a service actually depends on the activities provided by the rest of services. For instance, let us revisit the travel agency example presented before in Section 3.1 (this example involved choreography 18 and services 15, $15'$, 16, $16'$, 17 given in Figure 3). In that example, we assumed that a travel agency service requires that either the air company service or the hotel service (or both) provide a transfer to take the client from the airport

	sync.	async.	send. vs proc.	full vs partial	send+proc	disjoint	immediate vs delayed
Present work	X	✓	✓	✓	✓	X	✓
Lanese et al.	✓	✓	✓	X	✓	✓	X

Table 2: Comparison between Lanese et al. work and our work.

to the hotel. A hotel providing a transfer is good regardless of whether the air company provides a transfer as well or not. However, a hotel not providing a transfer is valid for the travel agency only if the air company provides the transfer. Contrarily to [9], our framework considers that the suitability of a service depends on what the rest of services actually do, so this kind of conditional dependencies is taken into account. Furthermore, we present a method to automatically *derive* services from a choreography in such a way that the system consisting of these services necessarily *conforms* to the choreography. This contrasts with the projection notion given in [9], which does not guarantee that derived services do so. The problems of *the natural projection*, already discussed in the introduction, are also suffered by the method proposed in [9]. In order to avoid these problems, the authors introduce some restrictions on choreographies to state which ones are properly transformed by the projection.

Other works concern the projection and conformance validation between choreography and orchestration with *synchronous* communication. Bravetti and Zavattaro [8] propose a theory of contracts for conformance checking. They define an effective procedure that can be used to verify whether a service with a given contract can correctly play a specific role within a choreography. Carbone et al. [16] study the description of communication behaviors from a global point of view of the communication and end-point behavior levels. Three definitions for proper-structured global description and a theory for projection are developed. Bultan and Fu [12, 11] specify web services as conversations by Finite State Machines to analyze whether UML collaboration diagrams are realizable or not.

In [27], Lucchi and Mazzara provide a formalization of conformance with π-calculus. By means of automata, Schifanella et al. [4] define a conformance notion that checks whether the interoperability is guaranteed. Moreover, Decker et al. [19] show how the Business Process Modeling Notation (BPMN) and the Business Process Execution Language (BPEL) can be used during choreography design. In [37], Van der Aalst et al. also focus on conformance by comparing the observed behavior recorded in logs with some predefined model.

Regarding the definition of conformance relations between choreographies and orchestrations, there are several works related to ours. We begin the comparison by considering the closest work. In [25] Lanese et al. develop a very detailed and broad study to compare these kind of systems. Their objective is bridging the gap between the WS-CDL and BPEL languages by formally defining them and then finding out the *features* systems should have to be equivalent if the natural projection is used. This work is based on the idea of *well formed conditions*, which depend on the properties one wants to preserve in each case. This idea clearly differs from our own objective since we do not try to discover what the conditions allowing the equivalence between systems are, but we define a derivation procedure able to derive a orchestration of services from *any* choreography; thus, since all choreographies enable a correct derivation by adding a suitable set of new control messages, *all* choreographies are well-formed for us. Despite of this difference, our approach follows a similar work line regarding the proposal of conformance relations. We both share the same idea of global and local behaviors for choreographies and orchestrations, respectively, as well as similar asynchrony assumptions.

In Table 2 we present a comparison of the conformance relations proposed in both works. The table compares the presence of different kinds of relations. The first four columns compare relations in terms of (a) different types of communication, i.e., *synchronous* versus *asynchronous* semantics; (b) the focus of the relation on either the times when messages are *produced/sent* or when they are *processed* by the addressee; and (c) the consideration of *full* behaviors or just *part* of them. Concerning the comparison of the type of communications, i.e. synchronous against asynchronous, in [25] both types are considered, but in this paper we do not explicitly consider a synchronous communications framework. Let us note that synchronous communications can be roughly simulated by making all messages be followed by a mandatory acknowledgement message from the addressee to the sender, which must be received by the sender before it can go on (this is illustrated in Section 3.1). Regarding sending and processing traces/behaviors, the same idea is followed in both works, that is, in an asynchronous communications framework, the time when a source generates a message differs from the instant when addressee actually reads it, and the conformance with the respect to each kind of moments differs. On the other hand, we consider not only the possibility of taking all behaviors into account in the comparison, but we also allow to consider only some of them, that is, a system might be correct as long as it implements at least *one* of the paths allowed by the choreography. In the next two columns of the table, we show the presence of a relation considering simultaneously sending and processing behaviors/traces, and the disjoint conformance relationship given in [25]. The last column of the table illustrates the presence, in our framework, of relations under two different asynchrony assumptions: (a) the case where messages are immediately stored in input buffers of destination services; and (b) the case where there might be a delay between the sending and the storage of messages in the corresponding input buffers.

There are other works based in the idea of *well formed conditions* [13, 14, 7]. In [13] Busi et al. present a first version of the formal model followed later by Lanese et al. to describe relations between choreographies and orchestration. In this work authors define the formal machinery to describe these two kinds of systems and a conformance relation based on bisimulation. Here Busi et al. do not deal with coordination or derivation problems. In [14], Busi et al. retake the same formal machinery, but this time they include state variables. The work is focused on the problem of maintaining the data consistency among the participants in the orchestration. In [7] Bravetti et al, following their former works of Lanese et al. [25] and Bravetti et al. [8], study whether it is possible to substitute a service by another one keeping all the properties of the composition.

To a lesser extent, there are also some related works about translating choreography and orchestration languages that use formal models: Valero et al. [35] define a Petri net approach that maps a subset [36] of WS-CDL to a Petri net model for analysis purposes, and Yeung [40] defines a mapping from WS-CDL and BPEL4WS into CSP, providing a formal approach to verifying the behavior of collaborating web services.

Finally, we compare our proposal with works in the domain of communication systems and reactive systems in general that address similar problems and use related formalizations. In [1] and [3], Alur et al. and Baker et al., respectively, study the problem of whether a given model of a distributed system, described as a whole, can be realized or not. On the contrary, we are not concerned about the realizability itself because we are assuming that additional coordination messages can be added to each involved party (in our case, services), and the addition of these messages allows us to realize *any* system described in our choreography formalism. In [3], the authors consider the automatic pathology resolution, but no algorithm or systematic method to solve pathologies is given indeed. Moreover, under the semantics they assume, some pathologies cannot be solved. In [20] Gotzhein and Bochmann present a method to automatically derive the behavior of each party from the model of the distributed system. Authors make some assumptions about the communication medium: Separate input FIFO queues for each source are assumed, and the order of messages is preserved as the protocol states, i.e., message delays are not considered. In [5] local and non-local choices are discussed by Ben-Abdallah and Leue, but only the detection of problems in the system description is concerned, not the synthesis of the behavior of parties in such a way that these problems do not appear. In [21] the synthesis problem is considered by Gouda and Yu, but distributed systems can have only two parties, which strongly eases the task of providing a proper coordination between all existing parties.

In [17] Castejon et al. study system pathologies and informally present some ways to solve them. Though no derivation algorithm or systematic method is given, an interesting contribution here is the classification of realizability problems from the point of view of each composition operator. The operators under consideration are those used in UML 2.0 collaborations, activity, and interaction diagrams, that is: sequential composition, alternative composition, interruption, and parallel composition. Though this model is different from our FSM-based model, most of pathologies identified in [17] apply to any kind of distributed system, so they apply to models defined in our languages too. Next we discuss them.

The *sequential composition* operator is prone to two kind of errors: causality and race conditions. The *causality* is broken when the expected sequence of interactions is not preserved in a system, that is, some action overtakes another one in an undesirable way. This problem is solved in our derivation algorithms by not letting a service evolve to the following step until the whole system has been aligned to do so. In the centralized derivation for the semantics without messages delays, a service may be several steps delayed with respect to the orchestrator, but all services are required to reach the current step when they are involved in the current step indeed. In the centralized derivation for the semantics with delays, services are required to reach the same step as the orchestrator (in particular, the orchestrator does not evolve further until they do). Similarly, in the decentralized derivation, all services are required to coordinately evolve to each new step (under both semantics).

On the other hand, a *race conditions* problem appears if messages are sent to addressees in some order, by they are received by these addressees in a different order. This problem may occur only if the delayed messages semantics is considered. In this case, messages of services derived by our derivation algorithms can reach their destination in a different order as they were sent. However, this does not disrupt the correct ordering in which transitions are actually executed. Let us suppose that service A sends message x to B and next A sends y to B too, but B receives y *before* x. Note that, under the delayed messages semantics, both of our derivation algorithms produce services where each message has its own *type*. This is equivalent to having several buffers, one for each message kind. In our example, service B has a buffer for messages x and another one for messages y. Thus, even if y is received first and x is received next, service B will be able to take x from its buffer before y if it is required. In particular, if service B is in a state where it can *only* process x, it will be able to do so. Next, if it reaches a state where it can only process y, it will be able to do so too. Thus, service B may be designed in such a way that message x will be *processed* (i.e. taken away from its buffer to trigger some transition depending on that message) before y regardless of whether x is received before y or not. If the choreography requires that x is sent and processed *before* y indeed, then the conditions required for achieving sending and processing conformance are preserved in this example. This idea is exploited in the derivation algorithms. By properly defining the messages that can be processed at each state, and not mixing messages involving two consecutive choreography steps, services correctly face the potential reception of messages in a different order as they were sent.

The *alternative composition*, specified by the choice operator, describes alternatives between one or more paths. For this operator, in [17] two sources of problems are identified: the decision-making process and the choice-propagation process. Regarding the *decision-making process*, authors define some choosing components based on the conditions associated with the alternatives. As it was mentioned in the previous section, our FSM-based model does not explicitly represent the localization of local/external information, variables, or guards enabling/disabling transitions according to variable values. In the centralized derivation, the orchestrator has omniscient capabilities, so it centralizes all information affecting decisions in the system and it autonomously decides which alternative is taken next. In the decentralized derivation, the decision-making process is based on a mixture between a classical token ring and a responsibility chain. We assume that the *choosing components* are, in our framework, the *message senders* of each alternative choice. A token ring, where each participant is able to choose either some of the choices where it is the sender, or pass this responsibility to the next service in the ring, is created. Thus it is implicitly assumed that, when each potential sender receives the token, it has the information required to decide whether it will take one of its choices or it will pass the decision to the next potential sender. As mentioned in the previous section, if some information had to be received by these potential senders before choosing, then the choreography designer would have the responsibility of adding some messages before the choice, from services owning this information to these potential senders, to conduct each sender to an appropriate state where it will, or will not, be able make each choice next. Alternatively, the required information to make choices could be *external*, such as external events. In this case, we may adopt the modifications proposed in the previous section: We enable the repetition of decision-making processes in those states where decisions of services might depend on external events that could be delayed. The dependance of models on local information and external events will be *explicitly* represented in our future EFSM-based models.

The second problem with alternative composition identified in [17] is the *choice-propagation process*. In our framework, choices are propagated to services by using control messages generated by the orchestrator (in the centralized derivation) or by the service that eventually took the choice (in the decentralized derivation). Note that there do not exist *orphan* messages (i.e. messages that are sent and never processed) in any of the derivations: At each step, all services are required to process the choice notification messages they receive before continuing (note that there is only *one* choice taken in each step), so they are necessarily processed by the corresponding addressees.

Two operators considered in [17], but not taken into account in our proposal, are the *interruption* and the *parallel composition*. Regarding the interruption, we have not explicitly introduced it in our model because this notion is not specifically identified in the choreography and orchestration languages motivating our models, WS-CDL and WS-BPEL. Still, these languages allow designers to specifically denote that some behaviors are triggered if some exceptional situations are detected. In our models, exceptional possibilities can be denoted as any other possibility, just by adding a new non-deterministic choice to denote this possibility. The reaction to this choice can be defined by attaching an appropriate message to the transition (e.g. an error or alert message) and/or conducting it to an appropriate state. Regarding the parallel composition, there is no explicit operator in our language to denote the parallel execution of several processes inside a given service. Let us note that, in many classical semantical approaches, it is assumed that a parallel execution is equivalent choosing among all possible interleaved executions of all parallel processes. In this specific case, the parallel operation is just a syntactic sugar, so it can be trivially added to our language without modifying the current semantics.

8. Conclusions and future work

In this paper we have presented a formal method to automatically extract a system of services from a given choreography, in such a way that the derived system conforms to the choreography. This method provides web service designers with a way to automatically construct early *prototypes* of services from a given choreography. These prototypes can be used to study their properties, as well as to serve as a kind of early (correct) implementation that can be refined in order to build the final implementation.

Instead on focusing on those choreographies where the natural projection works, our framework produces choreography-compliant sets of services even in cases where the natural projection does not work. This is achieved by adding some additional messaging to control branching and races issues. As we have shown in the examples of the Section 4, elections in choreography branching points may involve *several* services, not just one, so imposing some coordination between these services is required, and this coordination is not provided by the natural projection. Two derivation methods, one of them based on an orchestrator service and the other one yielding a decentralized system, are presented. For each method, we consider two alternatives: Making the system conform with respect to instants where messages are sent, or making it conform with respect to all proposed criteria. This distinction is motivated by the use of *asynchronous* communication, where the times when messages are sent and the times when they are processed may differ. We also consider two possible interpretations of asynchrony: One where the order in which messages are sent is preserved in destination input buffers, and another one where messages can be mixed up in destination input buffers. Centralized and decentralized derivation algorithms are presented for both interpretations.

Languages for defining models of orchestrations and choreographies, based on extensions of finite state machines with buffers, have been presented, and we have defined some formal semantic relations where, in particular, sending traces are distinguished from processing traces, and the suitability of a service for a given choreography may depend on the activities of the rest of services it will be connected with. The proposed framework is illustrated with several toy examples and a small case study.

As future work, we will study methods to reduce the number of additional messaging we have to add to derived services in order to control branching and races issues. Let us note that this goal can be considered at the services level, as we have done in this paper, or, alternatively, at the choreography level. In the latter case, we could rephrase the goal as follows: We wish to study what is the minimum amount of additional messaging we have to add to a given *choreography*, such that a simple natural projection of the augmented choreography would lead to choreography-compliant set of services. Thus, the problem of reducing the additional messaging can be considered at any of these dual levels. Besides, we are currently developing a tool such that, given a choreography defined by (a subset of) WS-CDL, it transforms it into the kind of choreography models considered in this paper, next it automatically extracts service models according to the algorithms proposed in this paper, and finally it transforms these models into WS-BPEL. Let us note that this tool will, in turn, be useful to reduce the number of additional control messages in our derived systems and improve their efficiency, because it will allow to easily experiment with alternative coordination strategies. In particular, we wish to develop a derivation method taking advantage of the main derivation trends, depending on the applicability of each one in each case: a) natural projection if "well-formedness" conditions hold; b) centralized derivation if (a) is not possible but all required information to take a choice could be owned by a single service; and c) decentralized derivation if (b) is not possible, so a set of services must coordinate, according to local information, to determine the branch to choose. Finally, we wish to introduce data variables and time in our modelling languages by using a kind of *extended finite state machines* (EFSMs) with time as core model, instead of FSMs, as previously explained in detail in Section 6.

Acknowledgments

We wish to thank the reviewers for the careful reading of the paper and the detailed comments.

References

[1] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of msc graphs. *Theor. Comput. Sci.*, 331(1):97–114, 2005.

[2] T. Andrews and F. Curbera. Web Service Business Process Execution Language, Working Draft, 2004. Version 2.0, 1.

[3] P. Baker, P. Bristow, C. Jervis, D. J. King, R. Thomson, B. Mitchell, and S. Burton. Detecting and resolving semantic pathologies in UML sequence diagrams. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 50–59, 2005.

[4] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Service selection by choreography-driven matching. In *WEWST*, 2007.

[5] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 259–274. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0035393.

[6] M. Boreale, R. D. Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes. *Inf. Comput.*, 172(2):139–164, 2002.

[7] M. Bravetti, I. Lanese, and G. Zavattaro. Contract-driven implementation of choreographies. In *Trustworthy Global Computing, 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers*, pages 1–18, 2008.

[8] M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *Software Composition*, pages 34–50, 2007.

[9] M. Bravetti and G. Zavattaro. Contract compliance and choreography conformance in the presence of message queues. In *Proc. of 5th International workshop on Web Services and Formal Methods, WS-FM'08, LNCS (in press)*. Springer, 2008.

[10] E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In *4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067*, pages 187–195. Springer, 2001.

[11] T. Bultan and X. Fu. Choreography modeling and analysis with collaboration diagrams. *IEEE Data Eng. Bull.*, 31(3):27–30, 2008.

[12] T. Bultan and X. Fu. Specification of realizable service conversations using collaboration diagrams. *Service Oriented Computing and Applications*, 2(1):27–39, 2008.

[13] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration: A synergic approach for system design. In *Service-Oriented Computing - ICSOC 2005, Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005, Proceedings*, pages 228–240, 2005.

[14] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *Coordination Models and Languages, 8th International Conference, COORDINATION 2006, Bologna, Italy, June 14-16, 2006, Proceedings*, pages 63–81, 2006.

[15] L. Caires and H. T. Vieira. Conversation types. *Theor. Comput. Sci.*, 411(51-52):4399–4440, 2010.

[16] M. Carbone, K. Honda, and N. Yoshida. Theoretical aspects of communication-centred programming. *Electr. Notes Theor. Comput. Sci.*, 209:125–133, 2008.

[17] H. Castejon, R. Braek, and G. Bochmann. Realizability of collaboration-based service specifications. *Asia-Pacific Software Engineering Conference*, 0:73–80, 2007.

[18] I. Castellani and M. Hennessy. Testing theories for asynchronous languages. In *FSTTCS*, pages 90–101, 1998.

[19] G. Decker, O. Kopp, F. Leymann, K. Pfitzner, and M. Weske. Modeling service choreographies using bpmn and bpel4chor. In *CAiSE*, pages 79–93, 2008.

[20] R. Gotzhein and G. von Bochmann. Deriving protocol specifications from service specifications including parameters. *ACM Trans. Comput. Syst.*, 8:255–283, November 1990.

[21] M. Gouda and Y. Yu. Synthesis of communicating finite-state machines with guaranteed progress. *IEEE Transactions on Communications*, 32(7):779–788, 1984.

[22] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008.

[23] M. Huhns and M. Singh. Service-oriented computing: Key concepts and principles. In *IEEE Internet Computing*, pages 75–81. IEEE Computer Society Press, 2005.

[24] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[25] I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*, pages 323–332, 2008.

[26] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines: A survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[27] R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. *J. Log. Algebr. Program.*, 70(1):96–118, 2007.

[28] A. Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In *4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067*, pages 196–205. Springer, 2001.

[29] Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In *WWW*, pages 973–982, 2007.

[30] I. Rodríguez. A general testability theory. In *CONCUR 2009 - Concurrency Theory, 20th International Conference, LNCS 5710*, pages 572–586. Springer, 2009.

[31] I. Rodríguez, M. Merayo, and M. Núñez. $\mathcal{HOTL}$: Hypotheses and observations testing logic. *Journal of Logic and Algebraic Programming*, 74(2):57–93, 2008.

[32] G. Salaün and T. Bultan. Realizability of choreographies using process algebra encodings. In *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings*, pages 167–182, 2009.

[33] J. Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.

[34] J. Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99, LNCS 1664*, pages 46–65. Springer, 1999.

[35] V. Valero, M. E. Cambronero, G. Diaz, and H. Macia. A petri net approach for the design and analysis of web services choreographies. *J. Log. Algebr. Program.*, 2009.

[36] V. Valero, G. Diaz, M. Cambronero, and H. Macia. A barred operational semantics for a subset of ws-cdl with time restrictions. *Journal of Logic and Algebraic Programming*, August 2009.

[37] W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and H. M. W. Verbeek. Choreography conformance checking: An approach based on bpel and petri nets. In *The Role of Business Processes in Service Oriented Architectures*, 2006.

[38] W. M. P. van der Aalst, N. Lohmann, P. Massuthe, C. Stahl, and K. Wolf. From public views to private views - correctness-by-design for services. In *WS-FM*, pages 139–153, 2007.

[39] W3C. Web Services Choreography Description Language, 2004. `http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217`.

[40] W. L. Yeung. Mapping ws-cdl and bpel into csp for behavioural specification and verification of web services. In *ECOWS*, pages 297–305, 2006.

Appendix: Proofs

Proof of Theorem *4.3*

Let us note that if $\mathcal{S}$ $\texttt{conf}^{f\prime}$ $\mathcal{C}$ then for all $\texttt{conf}_x \in \{\texttt{conf}'_s, \texttt{conf}'_p, \texttt{conf}', \texttt{conf}^{f\prime}_s, \texttt{conf}^{f\prime}_p, \texttt{conf}^{f\prime}\}$ we have $\mathcal{S}$ $\texttt{conf}_x$ $\mathcal{C}$, so we will just prove $\mathcal{S}$ $\texttt{conf}^{f\prime}$ $\mathcal{C}$. According to definitions 3.1 and 4.1, this is equivalent to proving that we have $\texttt{Comp}(\{\sigma^M | \sigma \in \texttt{prcTraces}(\mathcal{S})\}) = \texttt{Comp}(\texttt{traces}(\mathcal{C}))$ and $\texttt{Comp}(\{\sigma^M | \sigma \in \texttt{sndTraces}(\mathcal{S})\}) = \texttt{Comp}(\texttt{traces}(\mathcal{C}))$.

Let us begin by proving $\texttt{Comp}(\{\sigma^M | \sigma \in \texttt{prcTraces}(\mathcal{S})\}) = \texttt{Comp}(\texttt{traces}(\mathcal{C}))$. In particular, let us start by proving that if $\sigma \in \texttt{Comp}(\{\sigma^M | \sigma \in \texttt{prcTraces}(\mathcal{S})\})$ then $\sigma \in$

`Comp`(`traces`$(\mathcal{C})$). The path closure σ can be either finite or infinite. Let us consider that it is finite. Let σ consist of $[(snd_1, i_1, proc_1), \ldots, (snd_m, i_m, proc_m), \texttt{stop}]$ as well as all of its prefixes. We will prove that all of these traces are in `traces`$(\mathcal{C})$, which will imply that σ is in fact a complete path closure of `traces`$(\mathcal{C})$.

Let c_1 be the initial configuration of $\mathcal{S}$ and $\alpha = [(snd_1, i_1, proc_1), \ldots, (snd_r, i_r, proc_r)] \in \sigma$. Since this sequence denotes a execution of S, there exist $k \geq r$ consecutive evolutions of $\mathcal{S}$ following the form $(c_1, snd'_1, i'_1, proc'_1, o'_1, adr'_1, c_2)$, $(c_2, snd'_2, i'_2, proc'_2, o'_2, adr'_2, c_3)$, $\ldots$, $(c_k, snd'_k, i'_k, proc'_k, o'_k, adr'_k, c_{k+1})$ such that, if the natural values $a_1 < \ldots < a_r$ denote all consecutive indexes of inputs belonging to M in the previous sequence (that is, $l \in \{a_1, \ldots, a_r\}$ iff $i'_l \in M$) then $[(snd'_{a_1}, i'_{a_1}, proc'_{a_1}), \ldots, (snd'_{a_r}, i'_{a_r}, proc'_{a_r})] = [(snd_1, i_1, proc_1), \ldots, (snd_r, i_r, proc_r)]$.

Let us note that, for all $a_1 \leq a_q \leq a_r$, we have that c_{a_q+1} denotes the configuration of $\mathcal{S}$ after processing the input i'_{a_q} in the previous sequence of consecutive evolutions. Let $c_{a_q+1} = ((u^1_{a_q+1}, b^1_{a_q+1}), \ldots, (u^{n+1}_{a_q+1}, b^{n+1}_{a_q+1}))$. According to the definition of $\mathcal{S}$, the last pair of c_{a_q+1}, i.e. $(u^{n+1}_{a_q+1}, b^{n+1}_{a_q+1})$, denotes the configuration of the orchestrator service. By the construction of $\mathcal{S}$ from $\mathcal{C}$, the names of states of each service in $\mathcal{S}$ are taken exactly from the names of states in $\mathcal{C}$. In particular, for each state s_q of $\mathcal{C}$, in `orchestrator`$(\mathcal{C})$ we have a state s_q, as well as a state s_{qpi} for all $1 \leq i \leq n+1$ and all p less than or equal to the number of transitions leaving s_q in $\mathcal{C}$. Let us rename s_{in} (the initial state of $\mathcal{C}$) by s_1. Let $\mathcal{P}$ denote the property that, for all $a_1 \leq a_q \leq a_r$, there exist $q-1$ states $s_2, \ldots, s_{q+1}$ of $\mathcal{C}$ and q consecutive evolutions $(s_1, i'_{a_1}, snd'_{a_1}, proc'_{a_1}, s_2)$, $\ldots$, $(s_q, i'_{a_q}, snd'_{a_q}, proc'_{a_q}, s_{q+1})$ in $\mathcal{C}$ such that we have:

(a) *(configuration of the orchestrator)* For all $1 \leq h \leq q$, let p_h be the ordinal of the transition $(s_h, i'_{a_h}, snd'_{a_h}, proc'_{a_h}, s_{h+1})$ in the set of all transitions from state s_h in $\mathcal{C}$. Then, $u^{n+1}_{a_q+1} = s_{qp_qj}$ for some $1 \leq j \leq n+1$ and $b^{n+1}_{a_q+1} = [\,]$.

(b) *(configuration of derived services that have already been informed by the orchestrator about where to go in this step)* For all $1 \leq g \leq j-1$ we have $u^g_{a_q+1} \in \{s_e, s_{ep_e} | 1 \leq e \leq q\} \cup \{s'_{qp_q}, s_{q+1}\}$. Moreover, let f be such that $u^g_{a_q+1} = s_f$ or $u^g_{a_q+1} = s_{f-1p_{f-1}}$. Then, $b^g_{a_q+1} = [(orc, a_{f\ p_f}), (orc, a_{f+1\ p_{f+1}}), \ldots, (orc, a_{q\ p_q})]$ (note that if $f = q+1$ then this buffer is empty).

(c) *(configuration of derived services that have not been informed by the orchestrator yet about where to go in this step)* For all $j \leq g \leq n$ we have $u^g_{a_q+1} \in \{s_e, s_{ep_e} | 1 \leq e \leq q\}$. Moreover, let f be such that $u^g_{a_q+1} = s_f$ or $u^g_{a_q+1} = s_{f-1p_{f-1}}$. Then, $b^g_{a_q+1} = [(orc, a_{f\ p_f}), (orc, a_{f+1\ p_{f+1}}), \ldots, (orc, a_{q-1\ p_{q-1}})]$.

Let us note that this property $\mathcal{P}$ would imply, in particular, that $[(snd'_{a_1}, i'_{a_1}, proc'_{a_1}), \ldots, (snd'_{a_r}, i'_{a_r}, proc'_{a_r})] \in$ `traces`$(\mathcal{C})$, that is equal to $[(snd_1, i_1, proc_1), \ldots, (snd_r, i_r, proc_r)] \in$ `traces`$(\mathcal{C})$, as it is required.

We prove $\mathcal{P}$ by induction over q. We take $q = 1$ as anchor case. The first message belonging to M that is processed in $\mathcal{S}$ is i'_{a_1}. This message is sent by snd'_{a_1} and processed by $proc'_{a_1}$. By the construction of $\mathcal{S}$ from $\mathcal{C}$, a service of $\mathcal{S}$ sends a message only after the orchestrator requests to do so. Moreover, the orchestrator requests a service to send a message to another exactly as it is defined in one of the transitions leaving s_1 in $\mathcal{C}$. Thus, there exists a state s_2 of $\mathcal{C}$ such that $(s_1, i'_{a_1}, snd'_{a_1}, proc'_{a_1}, s_2)$ is an evolution of $\mathcal{C}$. Let this evolution be the p-th one leaving s_1 in $\mathcal{C}$. By the construction of $\mathcal{S}$, the state of the orchestrator right after the service $proc'_{a_1}$ processes i'_1

must be s_{1pj} for some $1 \leq j \leq n+1$ (note that the orchestrator cannot go beyond $s_{1p\ n+1}$ before $proc'_{a_1}$ sends a message b_f to it). Moreover, at that moment no service has sent a message to the orchestrator yet, so the input buffer of the orchestrator is [ ]. Thus, we have $\mathcal{P}$ (a). Regarding (b), all services that have already been asked by the orchestrator for taking the p-th transition can be in two configurations: either they have already processed the pair (orc, a_{1p}) from their input buffer, and thus they are in either s_{1p} or s_2, or they have not, and thus they are in s_1. In both cases, condition (b) is preserved. Regarding (c), services that have not been notified to take the p transition are necessarily in s_1, so (c) is fulfilled.

We consider the inductive case. By induction hypothesis, let us suppose that there exist $(s_1, i'_{a_1}, snd'_{a_1}, proc'_{a_1}, s_2)$, $\ldots$, $(s_q, i'_{a_q}, snd'_{a_q}, proc'_{a_q}, s_{q+1})$ transitions in $\mathcal{C}$, $u^{n+1}_{a_q+1} = s_{qpj}$ for some $1 \leq j \leq n+1$, and $b^{n+1}_{a_q+1} = [\ ]$. Also by induction hypothesis, we assume that for all $1 \leq g \leq n+1$ we have that $u^{g}_{a_q+1}$ and $b^{g}_{a_q+1}$ preserve conditions (b) and (c). Let us note that, since we have $u^{n+1}_{a_q+1} = s_{qpj}$, the evolution $(s_q, i'_{a_q}, snd'_{a_q}, proc'_{a_q}, s_{q+1})$ is in fact the p-th transition leaving s_q in $\mathcal{C}$. According to the construction of $\mathcal{S}$ from $\mathcal{C}$, at state s_{qpj} the orchestrator can reach the state $s_{qp\ n+1}$ without requiring any message from any other service. At $s_{qp\ n+1}$, the orchestrator must process a message b_f from $proc'_{a_q}$ to move to state s_{q+1}. Let us note that, right after the service $proc'_{a_q}$ processes its message, it reaches a state s'_{qp} and sends a message b_f to the orchestrator. Thus, the input buffer of the orchestrator will eventually be $[(proc'_{a_q}, b_f)]$, and thus it will be able to move to s_{q+1}. Once the orchestrator reaches state s_{q+1}, its input buffer is empty again. Now we can prove the existence of a subsequent evolution in $\mathcal{C}$ and the preservation of conditions (a), (b), and (c) as we did before in the anchor case, though this time we depart from state s_{q+1} and we process message i'_{a_q+1} (instead of s_1 and i'_{a_1}, respectively). The only significant difference lies in proving conditions (b) and (c). On the one hand, a service that was in case (b) in step q will be able to evolve into state s_{q+1} (if it did not do it before) by processing all pairs stored in its input buffer (note that they are stored in the required order to do so). On the other hand, a service that was in case (c) in step q will receive from the orchestrator an instruction to take the p-th transition in that step, and next it will be able to process it to move to state s_{q+1} by processing all pairs stored in its input buffer, as in (b). Once the orchestrator reaches state s_{q+1}, it will start to tell all services what transition to take in step $q+1$. In particular, the service snd'_{a_q+1} will eventually take the required transition and next it will send the message i'_{a_q+1} to $proc'_{a_q+1}$. Hence, the service $proc'_{a_q+1}$ will eventually be able to process it. Let us note that, at the time when service $proc'_{a_q+1}$ processes that message from a service snd'_{a_q+1}, some services will have already been told by the orchestrator what transition to take next. Thus, any service being in cases (b) or (c) in step q will be again in any of these cases (b) or (c) in step $q+1$.

In this way we have proven property $\mathcal{P}$, and we conclude that $[(snd'_{a_1}, i'_{a_1}, proc'_{a_1}), \ldots, (snd'_{a_r}, i'_{a_r}, proc'_{a_r})] = [(snd_1, i_1, proc_1), \ldots, (snd_r, i_r, proc_r)] \in \texttt{traces}(\mathcal{C})$.

As we consider that the path closure σ is finite, we also have to prove $[(snd_1, i_1, proc_1), \ldots, (snd_m, i_m, proc_m), \texttt{stop}] \in \texttt{traces}(\mathcal{C})$. Let us see that, due to the construction of $\mathcal{S}$ from $\mathcal{C}$, $\mathcal{S}$ can get stuck only if the orchestrator reaches a state s_t such that there is no outgoing transition at s_t in $\mathcal{C}$. Let us note that, if it is not the case, then the orchestrator will be able to select a transition and request all other services to take that transition. All services will add this request to their input buffers, and eventually they will be able to take that transition (note that, according to $\mathcal{P}$ (b) and (c), all services will own all messages required to evolve in their input buffers). Then, the orchestrator will ask a service to send a message to another, the former service will eventually do it, and the latter will eventually process it, thus allowing the orchestrator to continue. We conclude that, if $\mathcal{S}$ can get stuck after executing $[(snd_1, i_1, proc_1), \ldots, (snd_m, i_m, proc_m)]$, then we necessarily have $[(snd_1, i_1, proc_1), \ldots, (snd_m, i_m, proc_m), \texttt{stop}] \in \mathcal{C}$.

Let us suppose that the path closure σ is infinite. The property previously proved by induction over q shows that, *if* a trace of length q can be executed by $\mathcal{S}$, then it can also be executed by $\mathcal{C}$. Since this applies to traces of any size, all traces belonging to the infinite path closure σ can be executed by $\mathcal{C}$, and so we have $\sigma \in \texttt{Comp}(\texttt{traces}(\mathcal{C}))$.

Now we prove the inclusion of sets in the opposite direction, that is, we prove that if $\sigma \in \texttt{Comp}(\texttt{traces}(\mathcal{C}))$ then $\sigma \in \texttt{Comp}(\{\sigma^M | \sigma \in \texttt{prcTraces}(\mathcal{S})\})$. Again, σ can be either finite or infinite. Let us suppose that it is finite, that is, σ consists of a trace $[(snd_1, i_1, proc_1), \ldots, (snd_m, i_m, proc_m), \texttt{stop}]$ and all of its prefixes. We prove that, for all $\alpha \in [(snd_1, i_1, proc_1), \ldots, (snd_r, i_r, proc_r)] \in \sigma$, we have $\alpha \in \{\sigma^M | \sigma \in \texttt{sndTraces}(\mathcal{S})\}$. As before, let us rename s_{in} (the initial state of $\mathcal{C}$) by s_1. Since $\alpha \in \sigma$, we know that there exist $r-1$ states $s_2, \ldots, s_r$ of $\mathcal{C}$ such that $(s_1, i_1, snd_1, proc_1, s_2),\ \ldots,\ (s_q, i_r, snd_r, proc_r, s_{r+1})$ are consecutive evolutions of $\mathcal{C}$. Let $\mathcal{P}'$ be the property that, for all $1 \leq q \leq r$, there exist $b \geq q$ consecutive evolutions $(c_1, i'_1, snd'_1, proc'_1, c_2),\ \ldots,\ (c_b, i'_b, snd'_b, proc'_b, c_{b+1})$ in $\mathcal{S}$ such that, for some natural numbers $a_1 < \ldots < a_q$, we have that $a_1, \ldots, a_q$ are the indexes of inputs in these evolutions belonging to M (i.e. $l \in \{a_1, \ldots, a_q\}$ iff $i'_l \in M$) and for all $1 \leq g \leq q$ we have $i'_{a_g} = i_g$, $snd'_{a_g} = snd_g$, and $proc'_{a_g} = proc_g$. Moreover, let $c_{b+1} = ((u^1_{b+1}, b^1_{b+1}), \ldots, (u^{n+1}_{b+1}, b^{n+1}_{b+1}))$. Then, we have (a), (b), and (c) as stated before in property $\mathcal{P}$ after replacing all appearances of a_q by b (from now on, the resulting conditions will be denoted by (a)', (b)', and (c')). Let us note that the property $\mathcal{P}'$ would imply, in particular, that $\alpha \in \{\sigma^M | \sigma \in \texttt{sndTraces}(\mathcal{S})\}$.

We can prove $\mathcal{P}'$ by induction over q. Let $q = 1$ be the anchor case. Let us suppose that $(s_1, i_1, snd_1, proc_1, s_2)$ is the p-th transition available in $\mathcal{C}$ from s_1. By the construction of $\mathcal{S}$ from $\mathcal{C}$, the orchestrator starts at state s_1 too, and it can move from this state to a state s_{1p1}. From this state, it starts to ask the rest of services for taking the p-th available transition. Eventually, the service snd_1 will be asked for sending i_1 to $proc_1$, it will do it, and $proc_1$ will eventually process it. Let $(c_1, i'_1, snd'_1, proc'_1, c_2),\ \ldots,\ (c_b, i'_b, snd'_b, proc'_b, c_{b+1})$ be the evolutions taken $\mathcal{S}$ until $proc_1$ processes i_1. We have $proc'_b = proc_1$, $i'_b = i_1$, and for all evolutions before $(c_b, i'_b, snd'_b, proc'_b, c_{b+1})$ no message from M is processed. Moreover, by using very similar arguments as before when we considered $\mathcal{P}$, it is easy to see that all conditions (a)', (b)', and (c)' are kept in configuration c_{b+1}.

Let us consider the inductive case. Let us assume that $\mathcal{P}'$ holds for q. After executing the trace $[(snd_1, i_1, proc_1), \ldots, (snd_q, i_q, proc_q)]$, the choreography $\mathcal{C}$ is in state s_{q+1}. Let us suppose that $(s_{q+1}, i_{q+1}, snd_{q+1}, proc_{q+1}, s_{q+2})$ is the p-th available transition from s_{q+1} in $\mathcal{C}$. By the construction of $\mathcal{S}$ from $\mathcal{C}$, from a configuration of $\mathcal{S}$ fulfilling (a)', (b)', and (c)' in the q-th step, the orchestrator of $\mathcal{S}$ can evolve and reach the state s_{q+1}, where it will be able to take its p-th choice to reach $s_{q+1\ p\ 1}$. At this state, the orchestrator will start to ask the rest of services for taking the p-th transition too. On the other hand, regardless of whether the service snd_{q+1} was in case (b)' or (c)' after the q-th step, it will be able to process the messages in its input buffer until it reaches the state s_{q+1} too. Thus, it will eventually send i_{q+1} to $proc_{q+1}$. Similarly, the service $proc_{q+1}$ will eventually process it. Let c_b be the configuration of $\mathcal{S}$ right before $proc_{q+1}$ processes i_{q+1}. There exists a configuration c_{b+1} such that $(c_b, i_{q+1}, snd_{q+1}, proc_{q+1}, c_{b+1})$ is an evolution of $\mathcal{S}$. Now we can use similar arguments as in $\mathcal{P}$ to show that (a)', (b)', and (c)' hold in c_{b+1}.

Proving that $\mathcal{S}$ gets stuck only if $\mathcal{C}$ does so, and proving the inclusion of all traces in σ when σ is infinite, requires similar arguments as well.

Finally, we can prove $\texttt{Comp}(\{\sigma^M|\sigma \in \texttt{sndTraces}(\mathcal{S})\}) = \texttt{Comp}(\texttt{traces}(\mathcal{C}))$ by using very similar arguments as before when proving $\texttt{Comp}(\{\sigma^M|\sigma \in \texttt{prcTraces}(\mathcal{S})\}) = \texttt{Comp}(\texttt{traces}(\mathcal{C}))$.

Proof of Theorem *4.5*

The structure of this proof is very similar to the proof of Theorem 4.3, so we will just point out the differences with that proof. Since $\texttt{conf}^f_s$ implies $\texttt{conf}'_s$, we just have to prove $\texttt{conf}^f_s$, that is, we have to prove $\texttt{Comp}(\{\sigma^M|\sigma \in \texttt{sndTraces}(\mathcal{S})\}) = \texttt{Comp}(\texttt{traces}(\mathcal{C}))$. Let us start by considering $\texttt{Comp}(\{\sigma^M|\sigma \in \texttt{sndTraces}(\mathcal{S})\}) \subseteq \texttt{Comp}(\texttt{traces}(\mathcal{C}))$. Compared to the system $\mathcal{S}$ constructed in Definition 4.2, the only difference of the system $\mathcal{S}$ given in Definition 4.4 is that the acknowledgements of processing actions are substituted by acknowledgements of *sending* actions. In particular, it is the sender of each message, and not the service responsible of processing that message afterwards, the one that sends a message b_f to the orchestrator in order to allow the orchestrator to go on with the next step. In order to prove that all sending traces of $\mathcal{S}$ belong to $\mathcal{C}$, we can use an adaptation of the property $\mathcal{P}$ given in the proof of Theorem 4.3. This adaptation just consists in considering *sending* traces rather than processing traces. Since the system $\mathcal{S}$ of Definition 4.4 deals with sending traces exactly as the system $\mathcal{S}$ of Definition 4.2 deals with processing traces, the adaptation of the property $\mathcal{P}$ and its three statements (a), (b), and (c) to deal with sending traces is straightforward, and so is the adaptation of the proof by induction over q. On the other hand, the adaptation of the property $\mathcal{P}'$ of Theorem 4.3 to prove $\texttt{Comp}(\texttt{traces}(\mathcal{C})) \subseteq \texttt{Comp}(\{\sigma^M|\sigma \in \texttt{sndTraces}(\mathcal{S})\})$ is also direct.

Proof of Theorem *4.7*

The general structure of this proof will be similar to the proof of Theorem 4.3. As we will see, the main difference with that proof will lie in the way we prove that traces of $\mathcal{S}$ and $\mathcal{C}$ belong to each other, which will not be based on the state of the orchestrator (which does not exist in this case) but in the relations between the states of all derived services.

Similarly to that proof, we just prove $\mathcal{S}\ \texttt{conf}^{f\prime}\ \mathcal{C}$ because it implies $\mathcal{S}\ \texttt{conf}_x\ \mathcal{C}$ for all $\texttt{conf}_x \in \{\texttt{conf}'_s, \texttt{conf}'_p, \texttt{conf}', \texttt{conf}^{f\prime}_s, \texttt{conf}^{f\prime}_p, \texttt{conf}^{f\prime}\}$. Again, we prove it by showing that $\texttt{Comp}(\{\sigma^M|\sigma \in \texttt{prcTraces}(\mathcal{S})\}) = \texttt{Comp}(\texttt{traces}(\mathcal{C}))$ and $\texttt{Comp}(\{\sigma^M|\sigma \in \texttt{sndTraces}(\mathcal{S})\}) = \texttt{Comp}(\texttt{traces}(\mathcal{C}))$.

We start by proving $\texttt{Comp}(\{\sigma^M|\sigma \in \texttt{prcTraces}(\mathcal{S})\}) = \texttt{Comp}(\texttt{traces}(\mathcal{C}))$. In particular, let us show that if $\sigma \in \texttt{Comp}(\{\sigma^M|\sigma \in \texttt{prcTraces}(\mathcal{S})\})$ then $\sigma \in \texttt{Comp}(\texttt{traces}(\mathcal{C}))$. Let us assume that the path closure σ is finite. Then, σ consists of some trace $[(snd_1, i_1, proc_1), \ldots, (snd_m, i_m, proc_m), \texttt{stop}]$ as well as all of its prefixes. We prove that all of these traces belong to $\texttt{traces}(\mathcal{C})$, which implies that σ is a complete path closure of $\texttt{traces}(\mathcal{C})$.

Let c_1 be the initial configuration of $\mathcal{S}$ and $\alpha = [(snd_1, i_1, proc_1), \ldots, (snd_r, i_r, proc_r)] \in \sigma$. Let us introduce the same notation as in the proof of Theorem 4.3 to refer the configurations traversed by $\mathcal{S}$ in α. Since α denotes a execution of S, there exist $k \geq r$ consecutive evolutions of $\mathcal{S}$ following the form $(c_1, snd'_1, i'_1, proc'_1, o'_1, adr'_1, c_2)$, $(c_2, snd'_2, i'_2, proc'_2, o'_2, adr'_2, c_3)$, $\ldots$, $(c_k, snd'_k, i'_k, proc'_k, o'_k, adr'_k, c_{k+1})$ such that, if the natural values $a_1 < \ldots < a_r$ denote all consecutive indexes of inputs belonging to M in the previous sequence (that is, $l \in \{a_1, \ldots, a_r\}$ if and only if $i'_l \in M$) then $[(snd'_{a_1}, i'_{a_1}, proc'_{a_1}), \ldots, (snd'_{a_r}, i'_{a_r}, proc'_{a_r})] = [(snd_1, i_1, proc_1), \ldots, (snd_r, i_r, proc_r)]$.

Note that, for all $a_1 \leq a_q \leq a_r$, c_{a_q+1} denotes the configuration of $\mathcal{S}$ after processing the input i'_{a_q} in the previous sequence of consecutive evolutions. Let $c_{a_q+1} = ((u^1_{a_q+1}, b^1_{a_q+1}), \ldots,$

$(u^n_{a_q+1}, b^n_{a_q+1}))$. According to the definition of $\mathcal{S}$, for all $1 \leq g \leq n$ we have that $(u^g_{a_q+1}, b^g_{a_q+1})$ denotes the configuration of the g-th derived service, that is $\mathtt{decentral}(\mathcal{C}, id_g)$. By the construction of $\mathcal{S}$ from $\mathcal{C}$, the names of states of each service in $\mathcal{S}$ are taken from the names of states in $\mathcal{C}$. In particular, for each state s^t of $\mathcal{C}$, we have a state s^t as well as some other states related to t ($s^t_{idontchoose}$, $s^t_{iwillchoose}$, etc) in all derived services. Let us recall that the initial state of $\mathcal{C}$ is s^1. Let $\mathcal{Q}$ denote the property that, for all $a_1 \leq a_q \leq a_r$, there exist $q-1$ states $s^2, \ldots, s^{q+1}$ of $\mathcal{C}$ and q consecutive evolutions $(s^1, i'_{a_1}, snd'_{a_1}, proc'_{a_1}, s^2)$, $\ldots$, $(s^q, i'_{a_q}, snd'_{a_q}, proc'_{a_q}, s^{q+1})$ in $\mathcal{C}$ such that we have:

(a) *(configuration of the service* $\mathtt{decentral}(\mathcal{C}, proc'_{a_q})$*)* Let $id_g = proc'_{a_q}$. We have $u^g_{a_q+1} = s^{q+1}$ and $b^g_{a_q+1} = [\,]$.

(b) *(configuration of the service* $\mathtt{decentral}(\mathcal{C}, snd'_{a_q})$*)* Let $id_g = snd'_{a_q}$. For all $1 \leq h \leq q$, let p_h be the ordinal of the transition $(s^h, i'_{a_h}, snd'_{a_h}, proc'_{a_h}, s^{h+1})$ in the set of all transitions from state s_h in $\mathcal{C}$ where the sender is snd'_{a_h}. We have $u^g_{a_q+1} = s^q_{ichoose'_{p_q}}$ and $b^g_{a_q+1} = [(proc'_{a_q}, ididit)]$.

(c) *(configuration of the rest of services)* Let $1 \leq g \leq n$ be such that $id_g \neq proc'_{a_q}$ and $id_g \neq snd'_{a_q}$. Let d_1 be the index of the last step where either (i) id_g chose the transition to be taken or (ii) id_g was required to make the choice of either choosing one of its transitions or not *before* reaching the service that actually made the decision in that step. Formally, d_1 is the maximum natural value with $1 \leq d_1 \leq q$ such that, for all $d_1 < e \leq q$, we have that there does not exist j such that $id_g = id^{s^e}_j$, or it does but $j > k$ where $snd'_{a_e} = id^{s^e}_k$ (recall the definition of the list $[id^s_1, \ldots, id^s_{h_s}]$ at the beginning of Definition 4.6). Let d_2 be the index of the last step where id_g was the processor of the message. Formally, d_2 is the maximum natural value with $1 \leq d_2 \leq q$ such that, for all $d_2 < e \leq q$, we have $proc'_{a_e} \neq id_g$. Let $d = max(d_1, d_2)$. Then we have $u^g_{a_q+1} = s^t_*$ for some $d \leq t \leq q$, where $s^t_* \in S^t$, and S^t is the set of all states beginning by s^t (that is, s^t, $s^t_{idontchoose}$, $s^t_{iwillchoose}$, etc). For all $1 \leq h \leq q$, let p_h be defined as in (b). We also have $< b^g_{a_q+1} >= [w_{t+1}, \ldots, w_{q-1}]$, where $< b^g_{a_q+1} >$ is the result of removing from σ all pairs where the input does not follow the form $takemychoice_k$ for some k or $ididit$, and for all $t+1 \leq y \leq q-1$ we have $w_y = (snd'_{a_y}, takemychoice_{p_y})$ or $w_y = (proc'_{a_y}, ididit)$.

This property $\mathcal{Q}$ imply, in particular, that $[(snd'_{a_1}, i'_{a_1}, proc'_{a_1}), \ldots, (snd'_{a_r}, i'_{a_r}, proc'_{a_r})] \in \mathtt{traces}(\mathcal{C})$, that is, $[(snd_1, i_1, proc_1), \ldots, (snd_r, i_r, proc_r)] \in \mathtt{traces}(\mathcal{C})$, as it is required.

We prove $\mathcal{Q}$ by induction over q. We take $q = 1$ as anchor case. The first message belonging to M that is processed in $\mathcal{S}$ is i'_{a_1}. This message is sent by snd'_{a_1} and processed by $proc'_{a_1}$. By the definition of $\mathcal{S}$, a service snd'_{a_1} of $\mathcal{S}$ sends a service only after (a) all services before snd'_{a_1} in the list of services capable to sending a message at s^1 explicitly refuse to do so; and (b) all services after snd'_{a_1} have been notified who will make the decision (by propagating the $alreadychosen_{snd'_{a_1}}$ message and receiving $chosencomplete$ from the last service). In particular, by the construction of $\mathcal{S}$ from $\mathcal{C}$, snd'_{a_1} sends i'_{a_1} only if this can be done in $\mathcal{C}$ from state s^1. Thus, there exists a state s^2 of $\mathcal{C}$ such that $(s^1, i'_{a_1}, snd'_{a_1}, proc'_{a_1}, s^2)$ is an evolution of $\mathcal{C}$. Let $id_g = proc'_{a_1}$ be the service that processes i'_{a_1}. Since c_{a_1+1} denotes the configuration of $\mathcal{S}$ right after $proc'_{a_1}$ processes i'_{a_1}, by the definition of $\mathcal{S}$ we have $u^g_{a_1+1} = s^2$ and $b^g_{a_1+1} = [\,]$ so we have (a). Now, let $id_g = snd'_{a_1}$. Since $proc'_{a_1}$ has already processed the message, snd'_{a_1} already sent it, so by the definition of $\mathcal{S}$ we observe that $u^g_{a_1+1} = s^1_{ichoose'_{p_1}}$ and $b^g_{a_1+1} = [(proc'_{a_1}, ididit)]$

and we have (b). Finally, let $1 \leq g \leq n$ be such that $id_g \neq proc'_{a_1}$ and $id_g \neq snd'_{a_1}$. The service snd'_{a_1} has not informed the service id_g about what path it must take at this step (it starts to do so *after* it processes the $ididit$ message), so we have $u^g_{a_q+1} = s^1_*$ for some $s^1_* \in S^1$ and $<b^g_{a_q+1}>= [\,]$ and we have (c).

We consider the inductive case. By induction hypothesis, let us suppose that there exist $(s^1, i'_{a_1}, snd'_{a_1}, proc'_{a_1}, s^2),\ \ldots,\ (s^q, i'_{a_q}, snd'_{a_q}, proc'_{a_q}, s^{q+1})$ transitions in $\mathcal{C}$ and all services at configuration c_{a_q+1} preserve (a), (b), and (c). By the definition of $\mathcal{S}$, at state $s^1_{ichoose'_{p_q}}$ the service snd'_{a_q} starts to ask all the rest of services (but $proc'_{a_q}$, which was already asked) for taking its choice p_q, and it reaches state s^{q+1} when it finishes that task. Consequently, all of these services add a pair $(snd'_{a_q}, takemychoice_{p_q})$ to their input buffers. Note that, by (c), all services can process the messages in their input buffers until they reach s^{q+1} as well; this implies emptying their buffers indeed. Now, we reason similarly as in the anchor case. By the definition of $\mathcal{S}$, a service $proc'_{a_{q+1}}$ processes a message $i'_{a_{q+1}}$ only after the service $snd'_{a_{q+1}}$ sends that message to it. However, by the construction of $\mathcal{S}$ from $\mathcal{C}$, the system $\mathcal{S}$ allows such a behavior only if there exists a state s^{q+2} such that $(s^{q+1}, i'_{a_{q+1}}, snd'_{a_{q+1}}, proc'_{a_{q+1}}, s^{q+2})$ is an evolution of $\mathcal{C}$. Right after $proc'_{a_{q+1}}$ processes $i'_{a_{q+1}}$, by the construction of $\mathcal{S}$ from $\mathcal{C}$ we have that $proc'_{a_{q+1}}$ reaches the destination state of the evolution $(s^{q+1}, i'_{a_{q+1}}, snd'_{a_{q+1}}, proc'_{a_{q+1}}, s^{q+2})$ of $\mathcal{C}$, that is s^{q+2}, and its buffer is empty, so we have (a). Besides, right after $proc'_{a_{q+1}}$ processes the message, the service $snd'_{a_{q+1}}$ is in state $s^{q+1}_{ichoose'_{p_{q+1}}}$ and its input buffer is $[(proc'_{a_{q+1}}, ididit)]$, so we have (b) as well. Regarding (c), each service not being either the processor or the sender of the message is in any of the following cases: (i) in step $q+1$ it had to choose not to take any of its transitions (before giving $proc'_{a_{q+1}}$ the chance to actually take one of its transitions); and (ii) it did not. In case (i), it will be at some state $s^{q+1}_* \in S^{q+1}$ and its input buffer will be empty. In case (ii), let us consider the possible cases of the service in the previous step q. If it was the processor at that step, then its state will be s^{q+1} and its input buffer will be empty, thus fulfilling condition (c) at step $q+1$. If it was the sender at step q, then it could have processed the $ididit$ message from the processor or not; in both cases, it will fulfill condition (c) at step $q+1$. Finally, if it was neither the processor nor the sender, then the sender added a message $(snd'_{a_q}, takemychoice_{p_q})$ to its input buffer and it will fulfill (c) as well.

In this way we prove property $\mathcal{Q}$, and we can conclude that $[(snd'_{a_1}, i'_{a_1}, proc'_{a_1}),\ \ldots,$ $(snd'_{a_r}, i'_{a_r}, proc'_{a_r})] = [(snd_1, i_1, proc_1), \ldots, (snd_r, i_r, proc_r)] \in \mathtt{traces}(\mathcal{C})$.

We are considering that the path closure σ is finite, so we also have to prove $[(snd_1, i_1, proc_1),$ $\ldots, (snd_m, i_m, proc_m), \mathtt{stop}] \in \mathtt{traces}(\mathcal{C})$. Let us see that, due to the construction of $\mathcal{S}$ from $\mathcal{C}$, $\mathcal{S}$ can get stuck only if services reach a state s^t such that there is no outgoing transition at s^t in $\mathcal{C}$. Let us suppose that it is not the case, that is, there is an outgoing transition from s^t in $\mathcal{C}$ where a service snd sends a message m to a service $proc$. By the construction of $\mathcal{S}$ from $\mathcal{C}$, from state s^t there is at least one service capable of sending a message: snd can send m to $proc$ indeed. Let us see that some service of $\mathcal{S}$ will be able to process one of the available transitions. Services can refuse to take one of their transitions until the last service capable to take one of its transitions is reached. This last service is forced to take one of its transitions. So, one of the services will necessarily choose to take one of its transitions (say, one where snd' sends m' to $proc'$). After snd' sends m', it will wait for the acknowledgement from $proc'$. By $\mathcal{Q}$ (a), (b), and (c), the configurations of services guarantee that the service $proc'$ will eventually be able to process m' and send the acknowledgement to snd', which will then ask all the rest of services for taking its choice. The rest of services will eventually be able to take that choice. In order to see this, let us note that there is no risk that a service receives a $takemychoice$ message from the service $serv'$ choosing the transition at step $l+1$ *before* it receives a message $takemychoice$ from a service $serv$ choosing the transition at step l. Note that $serv'$ begins to send $takemychoice$ messages only after (a) all services before $serv'$ have refused to take one of their choices in step $l+1$; and (b) the last service sends a $chosencomplete$ message to $serv'$, which implies that all services after $serv'$ have received an $alreadychosen$ message from $serv'$. In particular, the service $serv$ will be able to do (a) or (b) only after it has sent *all* of its $takemychoice$ messages. So, the first $takemychoice$ message sent by $serv'$ is sent after the last $takemychoice$ message sent by $serv$ is sent. We conclude that all services will be able to process the $takemychoice$ messages in the right order, and thus they will be able to reach the destination of the transition of $\mathcal{C}$ under simulation. We conclude that $\mathcal{S}$ does not get stuck as long as $\mathcal{C}$ allows to execute a subsequent transition. Thus, if $\mathcal{S}$ can get stuck after executing $[(snd_1, i_1, proc_1), \ldots, (snd_m, i_m, proc_m)]$, then we necessarily have $[(snd_1, i_1, proc_1), \ldots, (snd_m, i_m, proc_m), \texttt{stop}] \in \mathcal{C}$.

We consider that σ is infinite. The property $\mathcal{Q}$ shows that, *if* a trace of length q can be executed by $\mathcal{S}$, then it can also be executed by $\mathcal{C}$. Since this applies to traces of any size, all traces belonging to the infinite path closure σ can be executed by $\mathcal{C}$, and so we have $\sigma \in \texttt{Comp}(\texttt{traces}(\mathcal{C}))$.

We consider the inclusion of sets in the opposite direction, that is, we prove that if $\sigma \in \texttt{Comp}(\texttt{traces}(\mathcal{C}))$ then $\sigma \in \texttt{Comp}(\{\sigma^M | \sigma \in \texttt{prcTraces}(\mathcal{S})\})$. Again, σ can be either finite or infinite. Let us suppose that it is finite, that is, σ consists of a trace $[(snd_1, i_1, proc_1), \ldots, (snd_m, i_m, proc_m), \texttt{stop}]$ as well as all of its prefixes. We prove that, for all trace $\alpha \in [(snd_1, i_1, proc_1), \ldots, (snd_r, i_r, proc_r)] \in \sigma$, we have $\alpha \in \{\sigma^M | \sigma \in \texttt{sndTraces}(\mathcal{S})\}$. Let us recall again that the initial state of $\mathcal{C}$ is s^1. Since $\alpha \in \sigma$, we know that there exist $r-1$ states $s^2, \ldots, s^r$ of $\mathcal{C}$ such that $(s^1, i_1, snd_1, proc_1, s^2), \ldots, (s^q, i_r, snd_r, proc_r, s^{r+1})$ are consecutive evolutions of $\mathcal{C}$. Let $\mathcal{Q}'$ be the property that, for all $1 \leq q \leq r$, there exist $b \geq q$ consecutive evolutions $(c_1, i'_1, snd'_1, proc'_1, c_2), \ldots, (c_b, i'_b, snd'_b, proc'_b, c_{b+1})$ in $\mathcal{S}$ such that, for some natural numbers $a_1 < \ldots < a_q$, we have that $a_1, \ldots, a_q$ are the indexes of inputs in these evolutions belonging to M (i.e. $l \in \{a_1, \ldots, a_q\}$ iff $i'_l \in M$) and for all $1 \leq g \leq q$ we have $i'_{a_g} = i_g$, $snd'_{a_g} = snd_g$, and $proc'_{a_g} = proc_g$. Moreover, let $c_{b+1} = ((u^1_{b+1}, b^1_{b+1}), \ldots, (u^{n+1}_{b+1}, b^{n+1}_{b+1}))$. Then, we have (a), (b), and (c) as stated before in property $\mathcal{Q}$ after replacing all appearances of a_q by b (from now on, the resulting conditions will be denoted by (a)', (b)', and (c')). Let us note that the property $\mathcal{Q}'$ would imply, in particular, that $\alpha \in \{\sigma^M | \sigma \in \texttt{sndTraces}(\mathcal{S})\}$.

Let us prove $\mathcal{Q}'$ by induction over q. We take $q = 1$ as anchor case. Let us suppose that $(s^1, i_1, snd_1, proc_1, s^2)$ is the p-th transition available in $\mathcal{C}$ from s_1. By the construction of $\mathcal{S}$ from $\mathcal{C}$, all services start at s^1. The first service capable of sending a message at s^1 decides whether it will take some of its transitions or it will refuse to do so and it will let the next service decide. In this way, all services capable of taking some of their transitions will have the chance to choose one of their transitions until some of them does so. One of these services is snd_1, which can decide to take the transition where it sends i_1 to $proc_1$. After it receives a $chosencomplete$ message from the last service, it sends i_1 to $proc_1$, $proc_1$ processes it and sends an acknowledgement to snd_1. Let $(c_1, i'_1, snd'_1, proc'_1, c_2), \ldots, (c_b, i'_b, snd'_b, proc'_b, c_{b+1})$ be the evolutions taken $\mathcal{S}$ until $proc_1$ processes i_1. We have $proc'_b = proc_1$, $i'_b = i_1$ and, for all evolutions before $(c_b, i'_b, snd'_b, proc'_b, c_{b+1})$, no message from M is processed. Moreover, by using very similar arguments as before when we considered $\mathcal{Q}$, it is easy to see that all conditions (a)', (b)', and (c)' are kept in configuration c_{b+1}.

Let us consider the inductive case. Let us assume that $\mathcal{Q}'$ holds for q. After executing the trace $[(snd_1, i_1, proc_1), \ldots, (snd_q, i_q, proc_q)]$, the choreography $\mathcal{C}$ is in state s^{q+1}. Let us suppose that $(s^{q+1}, i_{q+1}, snd_{q+1}, proc_{q+1}, s^{q+2})$ is the p-th available transition from s^{q+1} in $\mathcal{C}$. By the construction of $\mathcal{S}$ from $\mathcal{C}$, at a configuration of $\mathcal{S}$ fulfilling (a)', (b)', and (c)' in the q-th step, the first service capable of sending a message from s^{q+1} can evolve and reach the state s^{q+1}. Once it reaches s^{q+1}, it can refuse to take any of its transitions or let the next service to decide, which in turn will eventually be able to reach s^{q+1} and decide, and so on until the service responsible to either taking one of its transitions or refusing to do so is snd_{q+1}. This service snd_{q+1} can actually choose to send i_{q+1} to $proc_{q+1}$. The service $proc_{q+1}$, which is also able to eventually reach s^{q+1}, will be able to process i_{q+1}. Let c_b be the configuration of $\mathcal{S}$ right before $proc_{q+1}$ processes i_{q+1}. There exists a configuration c_{b+1} such that $(c_b, i_{q+1}, snd_{q+1}, proc_{q+1}, c_{b+1})$ is an evolution of $\mathcal{S}$. Now, we can use similar arguments as we did in the inductive case of $\mathcal{Q}$ to show that (a)', (b)', and (c)' hold in c_{b+1}.

Proving that $\mathcal{S}$ gets stuck only if $\mathcal{C}$ does so, and proving the inclusion of all traces in σ when σ is infinite, requires similar arguments as well.

Finally, we can prove $\texttt{Comp}(\{\sigma^M | \sigma \in \texttt{sndTraces}(\mathcal{S})\}) = \texttt{Comp}(\texttt{traces}(\mathcal{C}))$ by using very similar arguments as before when proving $\texttt{Comp}(\{\sigma^M | \sigma \in \texttt{prcTraces}(\mathcal{S})\}) = \texttt{Comp}(\texttt{traces}(\mathcal{C}))$.

Proof of Theorem *4.9*

As it happened before with the proofs of theorems 4.3 and 4.5, the structure of this proof is very similar to the proof of Theorem 4.7. Therefore, we will just focus on showing the differences with that proof. Since $\texttt{conf}^f_s$ implies $\texttt{conf}'_s$, we just have to prove $\texttt{conf}^f_s$, that is, we have to prove $\texttt{Comp}(\{\sigma^M | \sigma \in \texttt{sndTraces}(\mathcal{S})\}) = \texttt{Comp}(\texttt{traces}(\mathcal{C}))$. Let us start by considering $\texttt{Comp}(\{\sigma^M | \sigma \in \texttt{sndTraces}(\mathcal{S})\}) \subseteq \texttt{Comp}(\texttt{traces}(\mathcal{C}))$. Compared to the system $\mathcal{S}$ constructed in Definition 4.6, the only difference of the system $\mathcal{S}$ given in Definition 4.8 is that the acknowledgements of processing actions are *deleted*. In particular, no service sends any message $ididit$ to the service that chooses the transition to be taken and next sends a message belonging to M to another service. In order to prove that all sending traces of $\mathcal{S}$ belong to $\mathcal{C}$, we can use an adaptation of the property $\mathcal{Q}$ given in the proof of Theorem 4.3. This adaptation just consists in considering *sending* traces rather than processing traces. It is worth to point out that the sender of each message belonging to M does not begin to ask the rest of services for following its path *until* it has sent its message to the destination service. Let us note that the service that will send a message belonging to M in the *next* step needs to reach the next state to do so, so it is forced to wait until the service of the previous step tells it which transition it must take. Thus, the next sending event will necessarily happen after the previous message has been sent indeed. The adaptation of the property $\mathcal{Q}$ and its three statements (a), (b), and (c) to deal with sending traces is straightforward, and so is the adaptation of the proof by induction over q. On the other hand, the adaptation of the property $\mathcal{Q}'$ of Theorem 4.7 to prove $\texttt{Comp}(\texttt{traces}(\mathcal{C})) \subseteq \texttt{Comp}(\{\sigma^M | \sigma \in \texttt{sndTraces}(\mathcal{S})\})$ is also direct.

Proof of Theorem *5.4*

Due to the similarities between the adapted centralized derivation given in Definition 5.3 and the previous centralized derivation given in Definition 4.2, we can compose this proof as an adaptation of the proof of Theorem 4.3 – taking into account the differences between the original operational semantics, given in Definition 2.4, and the semantics that apply here, given in Definition 5.2. It is easy to adapt the proof of Theorem 4.3 to see that the *new* derivation holds under the *old* operational semantics. In particular, let us note that the use of additional control messages in the new derivation just constraints further the evolution of the system. Let us analyze step by step the behavior of the derived system under the *new* semantics where messages can be delayed – and thus mixed up in input buffers. First, let us consider how the system executes its *first* choreography transition, following some transition available at the first state of the choreography. We distinguish the following points in the execution of this first choreography transition:

(1) The system is in its initial configuration. All services and the orchestrator are in their initial states. Besides, the input buffers of all services are necessarily empty, and the input buffer of the orchestrator is empty too. Moreover, we also have $D = \emptyset$.

(2) From moment (1), the system will eventually reach a configuration where the orchestrator has sent the a_{jp} messages to all services. This occurs after the orchestrator chooses one of the available transitions, and next takes all transitions where it sends a_{jp} to announce its choice to the rest of services (let us note that the orchestrator does not need any message from any other service to take all of these transitions). Let moment (2) denote the execution point where this has just happened. In this point, we can see that all service id_i must be in one of the following cases:

    (2.1) The message (orc, a_{jp}) has not been received by id_i. In this case, $(orc, a_{jp}, id_i) \in D$.

    (2.2) Service id_i has already received a_{jp}, but id_i has not sent the message b_i to orc yet. In this case, the input buffer of id_i must be $[(orc, a_{jp})]$.

    (2.3) Service id_i has already sent message b_i to the orchestrator, but the orchestrator has not received it yet. Then, $(id_i, b_i, orc) \in D$ and the input buffer of id_i is [ ] again.

    (2.4) Service id_i has already sent message to b_i to the orchestrator, and the orchestrator has received it. Then, (id_i, b_i) belongs to the input buffer of the orchestrator, and the input buffer of service id_i is [ ] again.

(3) From moment (2), the system will eventually reach a configuration where all services have sent the messages b_i to the orchestrator. Due to the construction of the derived system, this implies that the service responsible of sending the message at the current choreography transition has done so, and that the service responsible of processing it has done so as well. Let moment (3) denote the moment where all of these events have just happened. Given the four possible cases of moment (2), we can see that the input buffer of all id_i must be [ ]. Besides, either (id_i, b_i) belongs to the input buffer of the orchestrator, or $(id_i, b_i, orc) \in D$.

(4) From moment (3), the system will eventually reach a configuration where the orchestrator has processed the message b_i from all services. Though messages b_i can be received by the orchestrator in any order, let us note that all of these messages $b_1, \ldots, b_n$ belong to different message *types* of the orchestrator, so they do not block each other in the input buffer and the orchestrator can take the message b_i of each required service from the input buffer as long as it has received it, regardless of whether other messages b_j from different services have been received before in the input buffer or not. Let moment (4) denote the execution point where all the aforementioned events have just happened. In this case, the input buffers of all services and the orchestrator are empty, and $D = \emptyset$. Moreover, it is easy to see that all services and the orchestrator must be in a state having the same name

as the destination of the choreography transition that has been executed by the system of services.

At moment (4), the system fulfills the same conditions as in moment (1) regarding input buffers and the state of the set D, though all services and the orchestrator are in the next state of the choreography. It is easy to prove, by induction over the number of choreography transitions taken, that after executing any number of choreography transitions and reaching moment (4), the system will necessarily reach moments (2), (3), and (4) as described above for the next choreography transition, and this can be said for *any* choreography transition that can be taken from the previous choreography state. Thus, the system will be able to make services perform all transitions required by the choreography. By using similar arguments as in the proof of Theorem 4.3, we have that the system of services conforms to the choreography with respect to all proposed relations.

Proof of Theorem *5.5*

Since the derivation used in this result is the derivation presented in Definition 4.6, we can construct this proof as an adaptation of the proof of Theorem 4.7, taking into account the difference between the operational semantics applying in that former case, given in Definition 2.4, and the semantics used here, given in Definition 5.2. Following the same idea as in the proof of Theorem 5.4, let us analyze step by step the behavior of the derived system under the new semantics where messages can be delayed and thus mixed up in input buffers. First, let us consider how the system executes its *first* choreography transition, following some transition available at the first state of the choreography. We distinguish the following points in the execution of this first choreography transition. This time, our analysis step by step will go a little bit further than the execution of the first transition.

(1) The system is in its initial configuration. All services are in their initial states, and their input buffers are necessarily empty. Moreover, we also have $D = \emptyset$.

(2) From moment (1), the system will eventually reach a configuration where some service id_y decides that it will choose some of the transitions where it is the sender. It will do it by sending a message $alreadychosen_y$ to the next service id_z in the decision-making sequence. Let moment (2) denote the execution point where this has just happened. That is, an $alreadychosen_y$ message was sent by id_y to id_z in the last system transition that has been executed before moment (2). At this point, we can see that the input buffers of all services must be [ ], even those of services before id_y in the sequence (which must have already processed their messages regarding the decision-making; otherwise the decision-making would not have reached id_y). The input buffers of id_y, id_z, and the rest of services that have not participated yet in the sequence are also equal to [ ]. Besides, $D = \{(id_y, alreadychosen_y, id_z)\}$.

(3) From moment (2), the system will eventually reach a configuration where the last service of the decision-making sequence, say id_f, sends a message $chosencomplete$ to id_y. Let (3) denote the moment where this just has happened. Let us note that id_y is blocked until it receives that message, so the action of announcing the choice taken by id_y to all services (by sending messages $takemychoice_j$) has not started yet. It is easy to see that, at this point, the input buffers of all services must be [ ], and we must have $D = \{(id_f, chosencomplete, id_y)\}$. Besides, for some choreography state s_q, the state of id_y is $s^q_{iwillchoose}$, and the state of the rest of services is $s^q_{idontchoose}$.

(4) From moment (3), the system will eventually reach a configuration where service id_y has sent the message $takemychoice_j$, as well as the message m required by the selected choreography transition, to its addressee, say id_a, and next id_a has sent $ididit$ to id_y. Let us note that, since messages can be mixed up, id_a could receive m *before* $takemychoice_j$. However, once it has received both messages, it will be able to continue, since m and $takemychoice_j$ are of different types and each one does not block the visibility of the other in the input buffer after both are stored in the buffer. Besides, let us note that no other service has done anything since moment (3) to this point, because only id_y and id_a are involved in the aforementioned messages exchanges and the rest of services are blocked. Let (4) denote the moment where all of this has just happened. It is easy to see that id_y must be in some state $s^q_{ichoose'_j}$ and id_a must be in some state s'_j, and the input buffers of both services are equal to [ ]. Moreover, input buffers of the rest of services are [ ] as well, and we have $D = \{(id_a, ididit, id_y)\}$.

(5) From moment (4), the system will eventually reach a state where id_y has sent the messages $takemychoice_j$ to all services that had not received it yet at moment (4) (these services are all but id_a). Let (5) denote the precise moment where this happens. Some services might have already processed the message $takemychoice_j$, and thus they would have reached a state of the form s'_j, while some other services might not have done so (in this case, their corresponding $takemychoice_j$ messages would belong to the set of not-yet received messages D). Moreover, services that have already reached s'_j could have gone beyond and they could have started to participate in the decision-making of the *next* choreography transition. Moreover, some service $id_{y'}$ could already have taken the decision to take the choice of the next transition, and it could have already propagated its decision of choosing to the next service of the decision-making sequence. However, we know for sure that service $id_{y'}$ has not received the message $chosencomplete$ allowing it to go further, because, as we said before, we are assuming that moment (5) happens right after id_y has sent $takemychoice_j$ to all services. Thus, id_y has *not* participated in the decision-making of the next choreography transition (recall that all services are required to do so before $chosencomplete$ can be sent). Moreover, services that have not processed their $takemychoice_j$ messages from id_y have not been able to participate in that decision-making either. For each of these services, a $takemychoice_j$ message could be mixed up in its input buffer with a message used to participate in the decision making of the next choreography transition ($idontchoose$ or $alreadychosen_{y'}$). However, even if $takemychoice_j$ is received *later* than $idontchoose$ or $alreadychosen_{y'}$, it can be processed by the service when it is received, because both kind of messages belong to different types.

(6) From moment (5), the system will eventually reach a configuration where the last service participating in the decision-making of the next choreography transition, say id_f, sends a message $chosencomplete$ to $id_{y'}$. This must happen because, from moment (5), all services that had not processed their $takemychoice_j$ message at moment (5) will be able to do so, so all services will eventually be able to participate in the decision-making of the new choreography transition. Let (6) denote the precise moment when id_f sends $chosencomplete$ to $id_{y'}$. Similarly to moment (3), let us note that $id_{y'}$ is blocked until it receives that message $chosencomplete$, so the action of announcing the choice taken by $id_{y'}$ in the new step to all services has not started yet. It is easy to see that, at this point, the input buffers of all services must be [ ], and we must have

$D = \{(id_f, chosencomplete, id_y)\}$. Besides, for some choreography state s'_q, the state of $id_{y'}$ is $s^{q'}_{iwillchoose}$, and the state of the rest of services is $s^{q'}_{idontchoose}$.

At moment (6), the system fulfills the same conditions as in moment (3) regarding input buffers and the state of the set D, though all services are in the states $s^{q'}_{iwillchoose}$ and $s^{q'}_{idontchoose}$ concerning the *next* state of the choreography. It is easy to prove, by induction over the number choreography transitions taken, that after executing any number of choreography transitions and reaching moment (6), the system will eventually reach moments (4), (5), and (6) as described above for the next transition, and this can be said for *any* choreography transition that can be taken from the previous choreography state. Thus, the system will be able to make services perform all transitions required by the choreography. By using similar arguments as in the proof of Theorem 4.7, we have that the system of services conforms to the choreography with respect to all proposed relations.