# Compositional Choreographies

Fabrizio Montesi[1] and Nobuko Yoshida[2]

[1] IT University of Copenhagen
[2] Imperial College London

**Abstract.** We propose a new programming model that supports a compositionality of choreographies. The key of our approach is the introduction of partial choreographies, which can mix global descriptions with communications among external peers. We prove that if two choreographies are composable, then the endpoints independently generated from each choreography are also composable, preserving their typability and deadlock-freedom. The usability of our framework is demonstrated by modelling an industrial use case implemented in a tool for Web Services, Jolie.

## 1 Introduction

Choreography-based programming is a powerful paradigm for designing communicating systems where the flow of communications is defined from a global viewpoint, instead of separately specifying the behaviour of each *endpoint* (peer). The local behaviour of the endpoints can then be automatically generated by means of *EndPoint Projection* (EPP). This paradigm has been used in standards [21,4] and language implementations [11,19,20,8]. Choreographies impact significantly the quality of software: they lower the chance for programming errors and ease their detection [16,6,7].

Previous works provide models for programming implementations of communicating systems with choreographies [6,7]. These models come with a type discipline for checking choreographies against protocol specifications given as session types [12], which are used to verify that the global behaviour of a choreography implements the expected communication flows. For example, a programmer may express a protocol using a *multiparty session type* [13] (or *global type*) such as the following one:

$$\texttt{B -> C}\colon \langle \texttt{string} \rangle; \texttt{C -> B}\colon \langle \texttt{int} \rangle; \texttt{B -> T}\colon \left\{ \begin{array}{l} ok\colon \texttt{B -> T}\colon \langle \texttt{string} \rangle; \texttt{T -> B}\colon \langle \texttt{date} \rangle, \\ quit\colon \texttt{end} \end{array} \right\}$$

Above, `B`, `C` and `T` are *roles* and abstractly represent endpoints in a system. In the protocol, a buyer `B` sends the name of a product to a catalogue `C`, which replies with the price for that product. Then, `B` notifies the transport role `T` of whether the price is accepted or not. In the first case (label $ok$), `B` sends also a delivery address to `T` and `T` replies with the expected delivery date. Otherwise (label $quit$), the protocol terminates immediately.

To the best of our knowledge, all previous choreography programming models (e.g., [7,6]) require the programmer to implement the behaviour of all roles in a protocol where it is used; e.g., it would not be possible to write the choreography of a system that uses the protocol above but gives the implementations only of roles `C` and `T`, to make

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 425–439, 2013.

those reusable by other programs as software libraries through an API. This seriously hinders the applicability of choreographies in industrial settings, where the interoperability of different systems developed independently is the key. In particular, it is not currently possible to:

- use choreographies to develop software libraries that implement subsets of roles in protocols such that they can be reused from other systems;
- reuse an existing software library that implements subsets of roles in protocols from inside a choreography.

To tackle the issues above, we ask: *Can we design a choreography model in which the EPP of a choreography can be composed with other existing systems?* The main problem is that existing choreography models rely on the complete knowledge of the implementation details of all endpoints to ensure that the systems generated by EPP will behave correctly. This complete knowledge is not available when independently developed implementations of distributed protocols need to be composed. In order to answer our question, we build a model for developing *partial choreographies*. Partial choreographies implement the behaviour of subsets of the roles in the protocols they use. Endpoint implementations are then automatically generated from partial choreographies and composed with other systems, with the guarantee that their overall execution will follow the intended protocols and the behaviour of the originating choreographies.

**Main Contributions.** We provide the following contributions:

***Compositional Choreographies.*** We introduce a new programming model for choreographies in which the implementation of some roles in protocols can be omitted (§ 3). These *partial choreographies* can then be composed with others through message passing. Our model allows to describe both choreographies with many participants or just a single endpoint. We provide a notion of EPP that produces correct endpoint code from a choreography, and we show that the EPP of a choreography preserves its compositional properties (§ 5). Our model introduces *shared channel mobility* to choreographies, which gains a dynamism when two protocols are composed.

***Typing.*** We provide a type system for checking choreographies against protocol specifications given as multiparty session types [13]. The type system ensures that the composition of different programs implements the intended protocols correctly (§ 4), and that our EPP produces code that follows the behaviour of the originating choreographies. Our framework guarantees that the EPP is still typable (§ 5); therefore, the EPP is reusable as a "black box" composable with other systems and the result of the composition can be checked for errors by referring only to types.

***Deadlock-freedom and Progress among Composed Choreographies.*** In the presence of partial choreographies, we prove that we can (i) capture the existing methodologies for deadlock-freedom in complete choreographies as in [6,7] and (ii) extend the notion of progress for incomplete systems investigated in [13] to choreographies (§ 5). Our results demonstrate for the first time that choreographies can be effectively used also as a tool for progress in a compositional setting, offering a new viewpoint for investigating progress and giving a fresh look to the results in [6,7].

Proofs, auxiliary definitions, and other resources are posted at [1], including an implementation of our use case (§ 2) with Jolie [15,18].

## 2 Motivations: A Use Case of Compositional Choreographies

We present motivations for this study by reporting a use case from our industry collaborators [14], and informally introducing our model. For clarity, we discuss only its most relevant parts. An extended version can be found at [1].

In our use case a buyer company needs to purchase a product from one of many available seller companies. The use case has two aspects that previous choreography models cannot handle: (i) the system of the buyer company is developed independently from those of the seller companies, and use the latter as software modules without revealing internal implementation details; (ii) depending on the desired product, the buyer company selects a suitable seller company at runtime. We address these issues with *partial choreographies*. A partial choreography implements a subset of the roles in a protocol, leaving the implementation of the other roles to an external system. External systems can be discovered at runtime. In our case, the buyer company will select a seller and then run the protocol from the introduction by implementing only the buyer role B, and rely on the external seller system to implement the other two roles C and T.

**Buyer Choreography.** We now define a choreography for the buyer company, $C_B$.

$$
C_B = \begin{array}{lll}
1.\ \mathsf{u}[\mathsf{U}]\ \mathbf{starts}\ \mathsf{pd}[\mathsf{PD}] : a(k);\ \mathsf{u}[\mathsf{U}].prod \texttt{ -> } \mathsf{pd}[\mathsf{PD}].x : k; & & \\
2.\ \mathsf{pd}[\mathsf{PD}]\ \mathbf{starts}\ \mathsf{r}[\mathsf{R}] : b(k');\ \mathsf{pd}[\mathsf{PD}].x \texttt{ -> } \mathsf{r}[\mathsf{R}].y : k';\ \mathsf{r}[\mathsf{R}].\mathsf{find}(y) \texttt{ -> } \mathsf{pd}[\mathsf{PD}].z : k'; & & \\
3.\ \mathsf{pd}[\mathsf{B}]\ \mathbf{req}\ \mathsf{C},\mathsf{T} : z(k'');\quad \mathsf{pd}[\mathsf{B}].x \texttt{ -> } \mathsf{C} : k'';\quad \mathsf{C} \texttt{ -> } \mathsf{pd}[\mathsf{B}].price : k'' & & \\
4.\ \mathsf{if}\ \mathsf{check}(price)@\mathsf{pd}\ \mathsf{then} & & \\
5.\quad \mathsf{pd}[\mathsf{B}] \texttt{ -> } \mathsf{T} : k'' \oplus ok; & \mathsf{pd}[\mathsf{PD}] \texttt{ -> } \mathsf{u}[\mathsf{U}] : k[del]; & \mathsf{pd}[\mathsf{PD}] \texttt{ -> } \mathsf{u}[\mathsf{U}] : k\langle k''[\mathsf{B}]\rangle; \\
6.\quad \mathsf{u}[\mathsf{B}].addr \texttt{ -> } \mathsf{T} : k''; & \mathsf{T} \texttt{ -> } \mathsf{u}[\mathsf{B}].ddate : k'' & \\
7.\ \mathsf{else} & & \\
8.\quad \mathsf{pd}[\mathsf{B}] \texttt{ -> } \mathsf{T} : k'' \oplus quit; & \mathsf{pd}[\mathsf{PD}] \texttt{ -> } \mathsf{u}[\mathsf{U}] : k[quit] &
\end{array}
$$

Above, a purchase in the buyer company is initiated by a user process u. In Line 1, process u and the freshly created process pd (for purchasing department) start a session $k$ by synchronising on shared channel $a$. Each process is annotated with the role it plays in the protocol that the session implements. Then, still in Line 1, u sends the product $prod$ the user wishes to buy to pd. In Line 2 pd starts a new session $k'$ with a fresh process r (a service registry) through shared channel $b$. Then, pd forwards the product name to r, which replies with the shared channel of the seller to contact for the purchase.

We refer to statements such as those in Lines 1-2 as complete, since they describe the behaviour of all participants, both sender and receiver(s). On the other hand, the continuation in Lines 3-8 is a partial choreography that relies on the selected external seller to implement the protocol shown in the introduction and perform the purchase.

The partial choreography in Lines 3-8 is depicted as a sequence chart in Fig. 1.a, where dashed lines indicate interactions with external participants. In Line 3 pd *requests* a synchronisation on the shared channel stored in its local variable $z$ to create the new session $k''$, declaring that it will play role B and that it expects the environment to implement roles C and T. Session $k''$ proceeds as specified by the protocol in the introduction. First, pd sends the product name stored in $x$ through session $k''$ to the external process that is playing role C (the product catalogue executed by the seller company). Observe that here we do not specify the process name of the receiver, since that will be established by the external seller system. Then, pd waits to receive the price for the product from the external process playing role C in $k''$. In Line 4, pd checks whether

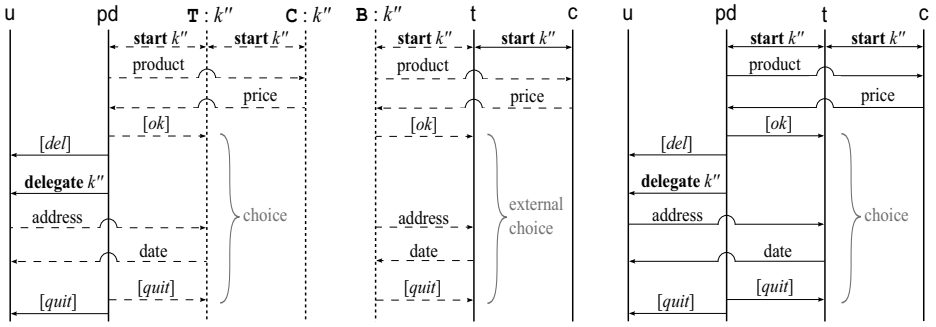


**Fig. 1.** Sequence charts for buyer (a), seller (b), and their composition (c)

the price is acceptable; if so, in Line 5 pd tells the external process playing role T (the transport process executed by the seller company) and user u (which remains internal to the buyer choreography) to proceed with the purchase (labels $ok$ and $del$ respectively). Still in Line 5, pd *delegates* to u the continuation of session $k''$ in its place, as role B. In Line 6, the user sends her address to T and receives a delivery date. If the price is not acceptable, Line 7, then in Line 8 pd informs the others to quit the purchase attempt.

**Seller Choreography and Composition.** We define now a choreography for a seller that can be contacted by $C_B$. Let the find function in $C_B$ return shared channel $c$ for electronic products, and $c'$ for other products; we refer to the choreographies of the respective seller companies as $C_S$ and $C'_S$. Below, we define $C_S$ ($C'_S$, omitted, is similar).

$$C_S = \begin{array}{l} 1.\ \mathbf{acc}\ \mathtt{c[C]}, \mathtt{t[T]} : c(k'');\quad \mathtt{B} \texttt{->} \mathtt{c[C]}.x_2 : k'';\quad \mathtt{c[C]}.\mathsf{price}(x_2) \texttt{->} \mathtt{B} : k'';\\ 2.\ \mathtt{B} \texttt{->} \mathtt{t[T]} : k'' \,\&\, \left\{ \begin{array}{r} ok : \mathtt{B} \texttt{->} \mathtt{t[T]}.daddr : k'';\quad \mathtt{t[T]}.\mathsf{time}(daddr) \texttt{->} \mathtt{B} : k'' \\ quit : \mathbf{0} \end{array} \right\} \end{array}$$

The choreography $C_S$, depicted as a sequence chart in Fig. 1.b, starts by *accepting* the creation of session $k''$ through shared channel $c$, offering to spawn two fresh processes c and t. Choreographies starting with an acceptance act as replicated, modelling typical always-available modules. The acceptance in Line 1 would synchronise with the request made by $C_B$ in the case $z = c$. Afterwards, c expects to receive the product name from the process playing B in session $k''$, and replies with the respective price. In Line 2, t (the process for the transport) waits for either label $ok$ or $quit$. In the first case, t also waits for a delivery address and then sends back the expected time of arrival.

From the code of $C_B$ and $C_S$ and, graphically, from their respective sequence charts we can see that they are *compatible*: sending actions match receiving actions on the other side and vice versa. Our model can recognise this by using roles in protocols as interfaces between partial choreographies (§ 4). The code for buyer and seller companies can be composed in a network with the parallel operator $\mid$ as: $C = C_B \mid C_S \mid C'_S$. Parallel composition allows partial terms in different choreographies to communicate. In (§ 3, Semantics) we formalise a semantics for choreography composition. To give the intuition behind our semantics, let us consider the sequence charts in Fig. 1.a and Fig. 1.b; their composition will behave as the sequence chart in Fig. 1.c.

$$
\begin{array}{lll|lll}
C ::= \eta; C & & \textit{(seq)} & \mid\ C_1 \mid C_2 & & \textit{(par)} \\
\quad \mid\ \mathsf{if}\ e@\mathtt{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 & & \textit{(cond)} & \mid\ (\boldsymbol{\nu} r)\ C & & \textit{(res)} \\
\quad \mid\ \mathsf{rec}\ X(\widetilde{x@\mathtt{p}}, \tilde{k}, \tilde{\mathtt{p}}) = C_2\ \mathsf{in}\ C_1 & & \textit{(rec)} & \mid\ X(\widetilde{x@\mathtt{p}}, \tilde{k}, \tilde{\mathtt{p}}) & & \textit{(call)} \\
\quad \mid\ \mathbf{0} & & \textit{(inact)} & \mid\ \mathtt{A} \texttt{->} q : k\&\{l_i : C_i\}_{i \in I} & & \textit{(branch)} \\
\hline
\eta ::= p\ \mathbf{starts}\ \tilde{q} : a(k) & & \textit{(start)} & \mid\ p.e \texttt{->} q.x : k & & \textit{(com)} \\
\quad \mid\ p \texttt{->} q : k[l] & & \textit{(sel)} & \mid\ p \texttt{->} q : k\langle k'[\mathtt{A}]\rangle & & \textit{(del)} \\
\quad \mid\ p\ \mathbf{req}\ \tilde{\mathtt{B}} : u(k) & & \textit{(req)} & \mid\ \mathbf{acc}\ \tilde{q} : a(k) & & \textit{(acc)} \\
\quad \mid\ p.e \texttt{->} \mathtt{B} : k & & \textit{(com-s)} & \mid\ \mathtt{A} \texttt{->} q.x : k & & \textit{(com-r)} \\
\quad \mid\ p \texttt{->} \mathtt{B} : k\langle k'[\mathtt{C}]\rangle & & \textit{(del-s)} & \mid\ \mathtt{A} \texttt{->} q : k(k'[\mathtt{C}]) & & \textit{(del-r)} \\
\quad \mid\ p \texttt{->} \mathtt{B} : k \oplus l & & \textit{(sel-s)} & & &
\end{array}
$$

$$
p, q ::= \mathtt{p}[\mathtt{A}] \qquad\qquad u ::= x \mid a
$$

**Fig. 2.** Compositional Choreographies.

## 3 Compositional Choreographies

This section introduces our model for compositional choreographies, a calculus where complete and partial actions can be freely interleaved.

**Syntax.** Fig. 2 defines the syntax of our calculus. $C$ is a choreography, $\eta$ is a complete or partial action, $p$ is a typed process identifier made by a process identifier $\mathtt{p}$ and a role annotation $\mathtt{A}$, $k$ is a session identifier, and $a$ is a shared channel. A term $\eta; C$ denotes a choreography that may execute action $\eta$ and then proceed as $C$. In the productions for $\eta$, terms *(start)*, *(com)*, *(sel)* and *(del)* are complete actions, whereas all the others are partial. In the productions for $C$, term *(branch)* is also partial.

***Complete Actions.*** Term *(start)* initiates a session: process $\mathtt{p}$ starts a new multiparty session through shared channel $a$ and tags it with a fresh identifier $k$. $\mathtt{p}$ is already running and dubbed *active process*, while $\tilde{\mathtt{q}}$ (which we assume nonempty) is a set of bound *service processes* that are freshly created. $\mathtt{A}, \tilde{\mathtt{B}}$ represent the respective roles played by the processes in session $k$. Term *(com)* denotes a communication where process $\mathtt{p}$ sends, on session $k$, the evaluation of a first-order expression $e$ to process $\mathtt{q}$, which binds it to its *local variable* $x$. Expressions may be shared channel names, capturing shared channel mobility. In *(sel)*, $\mathtt{p}$ communicates to $\mathtt{q}$ its selection of branch $l$. Term *(del)* models session mobility: process $\mathtt{p}$ delegates to $\mathtt{q}$ through session $k$ its role $\mathtt{C}$ in session $k'$.

***Partial Actions.*** In term *(req)*, process $\mathtt{p}$ is willing to start a new session $k$ by synchronising through shared channel $a$ with some other external processes. $\mathtt{p}$ is willing to play role $\mathtt{A}$ in the session and expects the other processes to play the other roles $\tilde{\mathtt{B}}$. *(req)* terms are supposed to synchronise with always-available *service processes*, modelled by term *(acc)*. In term *(acc)*, processes $\tilde{\mathtt{q}}$ are dynamically spawned whenever requested by a matching *(req)* term on the same shared channel $a$. Term *(com-s)* models the sending of a message from a process $\mathtt{p}$ to an external process playing role $\mathtt{B}$ in session $k$. Dually, in *(com-r)* process $\mathtt{q}$ receives a message intended for $\mathtt{B}$ in session $k$ from the external process playing role $\mathtt{A}$. *(del-s)* and *(del-r)* model, respectively, the sending and receiving of a delegation of role $\mathtt{C}$ in session $k'$. *(sel-s)* models the sending of a selection of label

$l$. *(sel-s)* can synchronise with a *(branch)* term, which offers a choice on multiple labels. Once a label $l_i$ is selected, *(branch)* proceeds by executing its continuation $C_i$.

***Other Terms.*** In term *(cond)*, process $\mathsf{p}$ evaluates condition $e$ to choose the continuation $C_1$ or $C_2$. Term *(res)* restricts the usage of a name $r$ to a choreography $C$. $r$ can be any name, i.e., a process identifier $\mathsf{p}$, a session identifier $k$, or a shared channel $a$. Term *(par)* models the parallel composition of choreographies, allowing partial actions to interact through the network. The other terms are standard: terms *(rec)*, *(call)* and *(inact)* model, respectively, a recursive procedure, a recursive call, and termination.

For clarity, we have annotated process identifiers with roles in all communications. Technically, this is necessary only for terms *(start)*, *(req)* and *(acc)* since roles can be inferred from session identifiers in all other terms (cf. [7]).

**Semantics.** We give semantics to choreographies with a labelled transition system (lts), whose rules are defined in Fig. 3 and whose labels $\lambda$ are defined as:

$$\lambda \quad ::= \quad \eta \quad | \quad \mathtt{A} \to q : k\&l \quad | \quad \mathsf{if@p} \quad | \quad (\boldsymbol{\nu} r)\, \lambda$$

We distinguish between labels representing complete or partial actions with the respective sets $\mathsf{CAct}$ and $\mathsf{PAct}$. $\mathsf{CAct}$ is the smallest set containing all $\eta$ that are complete actions and the labels of the form $\mathsf{if@p}$, closed under restrictions $(\boldsymbol{\nu} r)$. $\mathsf{PAct}$ is the smallest set containing all $\eta$ that are partial actions and the labels of the form $\mathtt{A} \to q : k\&l$, similarly closed under restriction of names. We also use other auxiliary definitions. $\mathsf{fc}(C)$ returns the set of all session/role pairs $k[\mathtt{A}]$ such that $k$ is free in $C$ and there is a process performing an action as role $\mathtt{A}$ in session $k$ in $C$. $\mathsf{rc}(\lambda)$ is defined only for partial labels that are not *(req)* or *(acc)*, and returns the session/role pair of the intended external sender or receiver of $\lambda$; e.g., $\mathsf{rc}(p.e \to \mathtt{B} : k) = k[\mathtt{B}]$. $\mathsf{fn}$ and $\mathsf{bn}$ denote the sets of free and bound names in a label or a term. $\mathsf{snd}(\eta)$ returns the name of the sender process in $\eta$, and is undefined if $\eta$ has no sender process (e.g., when $\eta$ is a *(com-r)*). $\mathsf{rcv}(\eta)$, instead, returns the session/role pair $k[\mathtt{A}]$ where $k$ is the session used in $\eta$ and $\mathtt{A}$ is the role of the receiver (similarly for $\mathsf{rcv}(\lambda)$). $\mathsf{fc}(\lambda)$ is as $\mathsf{fc}(C)$, but applied on labels.

We comment the rules. Rule $\lfloor^{C}|_{\textsc{Act}}\rceil$ handles actions that can be simply consumed. Rule $\lfloor^{C}|_{\textsc{Start}}\rceil$ starts a session with a global action, by restricting the names of the newly created session identifier $k$ and processes $\tilde{\mathsf{q}}$. Rule $\lfloor^{C}|_{\textsc{Com}}\rceil$ handles the communication of a value by substituting, in the continuation $C$, the binding occurrence $x$ under process identifier $\mathsf{q}$ with value $v$ (evaluated from expression $e$). Similarly, rules $\lfloor^{C}|_{\textsc{Com-S}}\rceil$ and $\lfloor^{C}|_{\textsc{Com-R}}\rceil$ implement the respective partial sending and receiving actions of a communication. In rule $\lfloor^{C}|_{\textsc{Branch}}\rceil$, process $\mathsf{q}$ receives a selection on a branching label and proceeds accordingly. Rules $\lfloor^{C}|_{\textsc{Cond}}\rceil$, $\lfloor^{C}|_{\textsc{Res}}\rceil$, and $\lfloor^{C}|_{\textsc{Ctx}}\rceil$ are standard. Rule $\lfloor^{C}|_{\textsc{Par}}\rceil$ makes global actions observable and blocks partial actions if their counterpart is in the parallel branch $C_2$. In rule $\lfloor^{C}|_{\textsc{Eq}}\rceil$, the relation $\mathcal{R}$ can either be the swapping relation $\simeq_C$, which swaps terms that describe the behaviour of different processes [7], or the structural congruence $\equiv$, which handles name restriction and recursion unfolding (see [1]).

Rule $\lfloor^{C}|_{\textsc{Sync}}\rceil$ is the main rule and enables two choreographies to perform compatible sending/receiving partial actions $\lambda$ and $\lambda'$ to interact and realise a global action, defined by $\lambda \circ \lambda'$. Function $\circ : \mathsf{PAct} \times \mathsf{PAct} \to \mathsf{CAct}$ is formally defined by the rules below:

$$\begin{array}{lcl}
\mathsf{p}[\mathtt{A}] \to \mathtt{B} : k\langle v\rangle \;\circ\; \mathtt{A} \to \mathsf{q}[\mathtt{B}] : k\langle v\rangle & = & \mathsf{p}[\mathtt{A}] \to \mathsf{q}[\mathtt{B}] : k\langle v\rangle \\
\mathsf{p}[\mathtt{A}] \to \mathtt{B} : k\langle k'[\mathtt{C}]\rangle \;\circ\; \mathtt{A} \to \mathsf{q}[\mathtt{B}] : k(k'[\mathtt{C}]) & = & \mathsf{p}[\mathtt{A}] \to \mathsf{q}[\mathtt{B}] : k\langle k'[\mathtt{C}]\rangle \\
\mathsf{p}[\mathtt{A}] \to \mathtt{B} : k \oplus l \;\circ\; \mathtt{A} \to \mathsf{q}[\mathtt{B}] : k\&l & = & \mathsf{p}[\mathtt{A}] \to \mathsf{q}[\mathtt{B}] : k[l]
\end{array}$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Act}}\rceil \quad \eta \notin \{(com),(com\text{-}s),(com\text{-}r),(start),(acc)\} \;\Rightarrow\; \eta; C \xrightarrow{\eta} C$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Start}}\rceil \quad \eta = p\ \mathbf{starts}\ \widetilde{\mathsf{q}[\mathsf{B}]} : a(k) \;\Rightarrow\; \eta; C \xrightarrow{\eta} (\boldsymbol{\nu} k, \tilde{\mathsf{q}})\, C$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Com}}\rceil \quad \eta = p.e \texttt{ -> } \mathsf{q}[\mathsf{B}].x : k \;\Rightarrow\; \eta; C \xrightarrow{p \texttt{ -> } \mathsf{q}[\mathsf{B}]:k\langle v\rangle} C[v/x@\mathsf{q}] \qquad (e \downarrow v)$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Com-S}}\rceil \quad p.e \texttt{ -> } \mathtt{B} : k; C \xrightarrow{p \texttt{ -> } \mathtt{B}:k\langle v\rangle} C \qquad (e \downarrow v)$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Com-R}}\rceil \quad \mathtt{A} \texttt{ -> } \mathsf{q}[\mathsf{B}].x : k; C \xrightarrow{\mathtt{A} \texttt{ -> } \mathsf{q}[\mathsf{B}]:k\langle v\rangle} C[v/x@\mathsf{q}]$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Branch}}\rceil \quad \mathtt{A} \texttt{ -> } q : k\&\{l_i : C_i\}_{i\in I} \xrightarrow{\mathtt{A} \texttt{ -> } q:k\&l_j} C_j \qquad (j \in I)$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Cond}}\rceil \quad \mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \xrightarrow{\mathsf{if}@\mathsf{p}} C_i \qquad (i = 1 \text{ if } e \downarrow \mathrm{true},\ i = 2 \text{ otherwise})$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Res}}\rceil \quad C \xrightarrow{\lambda} C' \;\Rightarrow\; (\boldsymbol{\nu} r)\, C \xrightarrow{(\boldsymbol{\nu} r)\,\lambda} (\boldsymbol{\nu} r)\, C'$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Ctx}}\rceil \quad C_1 \xrightarrow{\lambda} C_1' \;\Rightarrow\; \mathsf{rec}\, X(\widetilde{x@\mathsf{p}}, \tilde{k}, \tilde{\mathsf{p}}) = C_2\ \mathsf{in}\ C_1 \xrightarrow{\lambda} \mathsf{rec}\, X(\widetilde{x@\mathsf{p}}, \tilde{k}, \tilde{\mathsf{p}}) = C_2\ \mathsf{in}\ C_1'$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Par}}\rceil \quad C_1 \xrightarrow{\lambda} C_1' \;\Rightarrow\; C_1 \mid C_2 \xrightarrow{\lambda} C_1' \mid C_2 \qquad (\,\lambda \in \mathsf{CAct} \;\vee\; \mathsf{rc}(\lambda) \notin \mathsf{fc}(C_2)\,)$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Eq}}\rceil \quad \mathcal{R} \in \{\equiv, \simeq_C\} \quad C_1\, \mathcal{R}\, C_1' \quad C_1' \xrightarrow{\lambda} C_2' \quad C_2'\, \mathcal{R}\, C_2 \;\Rightarrow\; C_1 \xrightarrow{\lambda} C_2$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Sync}}\rceil \quad C_1 \xrightarrow{\lambda} C_1' \quad C_2 \xrightarrow{\lambda'} C_2' \;\Rightarrow\; C_1 \mid C_2 \xrightarrow{\lambda\circ\lambda'} C_1' \mid C_2'$$

$$\lfloor^{\mathrm{C}}|_{\textsc{P-Start}}\rceil \quad \left.\begin{array}{l} i \in [1,n] \quad \{\tilde{\mathsf{q}}\} = \{\tilde{\mathsf{q}}_1, \ldots, \tilde{\mathsf{q}}_n\} \\ \{\tilde{\mathsf{B}}\} = \{\tilde{\mathsf{B}}_1, \ldots, \tilde{\mathsf{B}}_n\} \quad C'' = \prod_i C_i \\ C \xrightarrow{p\ \mathbf{req}\ \tilde{\mathsf{B}}:u(k)} C' \\ C_i = \mathbf{acc}\ \widetilde{\mathsf{q}[\mathsf{B}]}_i : a(k); C_i' \end{array}\right\} \Rightarrow \begin{array}{l} C \mid C'' \xrightarrow{\lambda} (\boldsymbol{\nu} k, \tilde{\mathsf{q}})\left(C' \mid \prod_i (C_i')\right) \mid C'' \\ (\,\lambda = p\ \mathbf{starts}\ \widetilde{\mathsf{q}[\mathsf{B}]}_1, \ldots, \widetilde{\mathsf{q}[\mathsf{B}]}_n : a(k)\,) \end{array}$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil \quad C \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{r})\, C' \;\Rightarrow\; \eta; C \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{r})\, \eta; C' \quad \left(\begin{array}{cc} \mathsf{snd}(\eta) \in \mathsf{fn}(\lambda) & \tilde{r} = \mathsf{bn}(\lambda) \\ \mathsf{rcv}(\eta) \notin \mathsf{fc}(\lambda) & \tilde{r} \notin \mathsf{fn}(\eta) \\ \eta \notin \{(start),(acc)\} & \end{array}\right)$$

**Fig. 3.** Semantics of Compositional Choreographies

Observe that if $\lambda \circ \lambda'$ is not defined (the actions are incompatible), then the rule cannot be applied. Similarly, $\lfloor^{\mathrm{C}}|_{\textsc{P-Start}}\rceil$ models a session start by synchronising a partial choreography that requests to start a session with other choreographies that can accept the request on the same shared channel. The choreographies accepting the request remain available afterwards, for reuse. Finally, rule $\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil$ models asynchrony, allowing the sender process of an interaction $\eta$ ($\mathsf{snd}(\eta)$) to send a message and then proceed freely before the intended receiver actually receives it. In the rule, we require asynchrony to preserve the message ordering in a session wrt receivers with a causality check ($\mathsf{rcv}(\eta) \notin \mathsf{fc}(\lambda)$).

## 4 Typing Compositional Choreographies

We now present our typing discipline, which ensures that sessions in a choreography follow protocol specifications given as global types [13,3]. The key advances from

previous work [7] are: (i) introduction of the typing rules for partial choreographies and shared channel passing; and (ii) typing endpoints by local types, which offer transparent compositional properties for the behaviour of each process.

**Global and Local Types** from [13,3] are defined below:

$$G ::= \mathtt{A} \to \mathtt{B} : \langle U \rangle ; G \mid \mathtt{A} \to \mathtt{B} : \{l_i : G_i\}_{i \in I} \mid \mu\, \mathbf{t}\,.\, G \mid \mathbf{t} \mid \mathsf{end}$$

$$T ::= !\mathtt{A}\langle U \rangle ; T \mid ?\mathtt{A}\langle U \rangle ; T \mid \oplus \mathtt{A}\{l_i : T_i\}_{i \in I} \mid \&\mathtt{A}\{l_i : T_i\}_{i \in I} \mid \mu\, \mathbf{t}\,.\, T \mid \mathbf{t} \mid \mathsf{end}$$

$$S ::= G \mid \mathtt{int} \mid \mathtt{bool} \cdots \qquad U ::= S \mid T$$

$G$ is a global type. $\mathtt{A} \to \mathtt{B} : \langle U \rangle ; G$ abstracts a communication from role $\mathtt{A}$ to role $\mathtt{B}$ with continuation $G$, where $U$ is the type of the exchanged message. $U$ can either be a sort type $S$ (used for typing values or shared channels), or a local type $T$ (used for typing session delegation). In $\mathtt{A} \to \mathtt{B} : \{l_i : G_i\}_{i \in I}$, role $\mathtt{A}$ selects one label $l_i$ offered by role $\mathtt{B}$ and the global type proceeds as $G_i$. All other terms are standard.

$T$ denotes a local type. $!\mathtt{A}\langle U \rangle ; T$ represents the sending of a message of type $U$ to role $\mathtt{A}$, with continuation $T$. Dually, $?\mathtt{A}\langle U \rangle ; T$ represents the receiving of a message of type $U$ from role $\mathtt{A}$. $\oplus \mathtt{A}\{l_i : T_i\}_{i \in I}$ and $\&\mathtt{A}\{l_i : T_i\}_{i \in I}$ abstract the selection and the offering of some branches. The other terms are standard.

To relate a global type to the behaviour of an endpoint, we project a global type $G$ onto a local type that represents the behaviour of a single role. We write $[\![G]\!]_{\mathtt{A}}$ to denote the projection of $G$ onto the role $\mathtt{A}$, which is defined following [10] (cf. [1]).

**Type Checking.** We now introduce our type checking discipline for checking choreographies against global types. We use two kinds of typing environments, the linear session typing environments $\Delta$ and the unrestricted service environments $\Gamma$:

$$\Delta ::= \Delta, k[\mathtt{A}] : T \mid \emptyset \qquad \Gamma ::= \Gamma, x@\mathtt{p} : S \mid \Gamma, X : (\Gamma, \Delta) \mid \Gamma, \mathtt{p} : k[\mathtt{A}] \mid \Gamma, a : G\langle \mathtt{A} | \tilde{\mathtt{B}} | \tilde{\mathtt{C}} \rangle \mid \emptyset$$

$\Delta$ is standard [3], where $k[\mathtt{A}] : T$ maps a local type $T$ to a role $\mathtt{A}$ in a session $k$. In $\Gamma$, $x@\mathtt{p} : S$ types variable $x$ of process $\mathtt{p}$ with type $S$. $X : (\Gamma, \Delta)$ types recursive procedure $X$. $\mathtt{p} : k[\mathtt{A}]$ establishes that process $\mathtt{p}$ *owns* role $\mathtt{A}$ in session $k$. $a : G\langle \mathtt{A} | \tilde{\mathtt{B}} | \tilde{\mathtt{C}} \rangle$ types a shared channel $a$ with global type $G$: $\mathtt{A}$ is the role of the active process that starts the session through $a$; $\tilde{\mathtt{B}}$ are the roles of the service processes; $\tilde{\mathtt{C}}$ are the roles, in $\tilde{\mathtt{B}}$, that a choreography implements for the shared channel $a$, enabling compositionality of services. Whenever we write $a : G\langle \mathtt{A} | \tilde{\mathtt{B}} | \tilde{\mathtt{C}} \rangle$ in $\Gamma$, we assume that $\tilde{\mathtt{C}} \subseteq \tilde{\mathtt{B}}$, $\mathtt{A} \notin \tilde{\mathtt{B}}$, and that $\mathtt{A}, \tilde{\mathtt{B}} = \mathsf{roles}(G)$. $\mathsf{roles}(G)$ returns the set of roles in a global type $G$.

We can write $\Gamma, \mathtt{p} : k[\mathtt{A}]$ only if $\mathtt{p}$ is not associated to any other role in session $k$ in $\Gamma$ (a process may only play one role per session). A process $\mathtt{p}$ may however appear more than once in a same $\Gamma$, allowing processes to run multiple sessions. As usual, we require all other kinds of occurrences in environments to have disjoint identifiers.

A typing judgement $\Gamma \vdash C \triangleright \Delta$ establishes that a choreography $C$ is well-typed. Intuitively, $C$ is well-typed if shared channels are used according to $\Gamma$ and sessions are used according to $\Delta$. $\Delta$ gives the session types of the free sessions in $C$. Following the design idea that services should always be available, shared by other models [6,7], we assume that all *(acc)* terms in a choreography are not guarded by other actions. A selection of the rules defining our typing judgement is reported in Fig. 4.

We comment the typing rules. Rule $[{}^{\mathrm{T}}|_{\textsc{Start}}]$ types a *(start)*; $a : G\langle \mathtt{A} | \tilde{\mathtt{B}} | \tilde{\mathtt{B}} \rangle$ checks that the choreography should implement all roles in protocol $G$; processes $\tilde{\mathtt{q}}$ are checked to

$$\lfloor^{\mathrm{T}}|_{\textsc{Start}}\rceil\ \frac{\Gamma, a : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle, \Gamma' \vdash C \triangleright \Delta, \Delta' \quad \mathsf{r}[\mathtt{C}] \in \mathsf{p}[\mathtt{A}], \widetilde{\mathsf{q}[\mathtt{B}]} \Leftrightarrow \left(\mathsf{r} : k[\mathtt{C}] \in \Gamma' \wedge k[\mathtt{C}] : [\![G]\!]_{\mathtt{C}} \in \Delta'\right) \quad \tilde{\mathsf{q}} \notin \Gamma}{\Gamma, a : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle \vdash \mathsf{p}[\mathtt{A}]\ \mathbf{starts}\ \widetilde{\mathsf{q}[\mathtt{B}]} : a(k); C \triangleright \Delta}$$

$$\lfloor^{\mathrm{T}}|_{\textsc{Sel}}\rceil\ \frac{j \in I \quad \Gamma \vdash \mathsf{p} : k[\mathtt{A}], \mathsf{q} : k[\mathtt{B}] \quad \Gamma \vdash C \triangleright \Delta, k[\mathtt{A}] : T_j, k[\mathtt{B}] : T'_j}{\Gamma \vdash \mathsf{p}[\mathtt{A}] \texttt{ -> } \mathsf{q}[\mathtt{B}] : k[l_j]; C \triangleright \Delta, k[\mathtt{A}] : \oplus \mathtt{B}\{l_i : T_i\}_{i \in I}, k[\mathtt{B}] : \&\mathtt{A}\{l_i : T'_i\}_{i \in I}}$$

$$\lfloor^{\mathrm{T}}|_{\textsc{Req}}\rceil\ \frac{\Gamma \vdash x@\mathsf{p} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle \quad \Gamma, \mathsf{p} : k[\mathtt{A}] \vdash C \triangleright \Delta, k[\mathtt{A}] : [\![G]\!]_{\mathtt{A}}}{\Gamma \vdash \mathsf{p}[\mathtt{A}]\ \mathbf{req}\ \tilde{\mathtt{B}} : x(k); C \triangleright \Delta} \qquad \lfloor^{\mathrm{T}}|_{\textsc{Par}}\rceil\ \frac{\Gamma, \Gamma_i \vdash C_i \triangleright \Delta_i}{\Gamma, \Gamma_1 \circ \Gamma_2 \vdash C_1 \mid C_2 \triangleright \Delta_1, \Delta_2}$$

$$\lfloor^{\mathrm{T}}|_{\textsc{Acc}}\rceil\ \frac{\Gamma, a : G\langle \mathtt{D}|\tilde{\mathtt{B}}|\emptyset\rangle, \Gamma' \vdash C \triangleright \Delta, \Delta' \quad \mathsf{r}[\mathtt{C}] \in \widetilde{\mathsf{q}[\mathtt{A}]} \Leftrightarrow \left(\mathsf{r} : k[\mathtt{C}] \in \Gamma' \wedge k[\mathtt{C}] : [\![G]\!]_{\mathtt{C}} \in \Delta'\right) \quad \tilde{\mathsf{q}} \notin \Gamma}{\Gamma, a : G\langle \mathtt{D}|\tilde{\mathtt{B}}|\tilde{\mathtt{A}}\rangle \vdash \mathbf{acc}\ \widetilde{\mathsf{q}[\mathtt{A}]} : a(k); C \triangleright \Delta}$$

$$\lfloor^{\mathrm{T}}|_{\textsc{Com-s}}\rceil\ \frac{\Gamma \vdash e@\mathsf{p} : S \quad \Gamma \vdash \mathsf{p} : k[\mathtt{A}] \quad \Gamma \vdash C \triangleright \Delta, k[\mathtt{A}] : T \quad \mathsf{q} : k[\mathtt{B}] \notin \Gamma}{\Gamma \vdash \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B} : k; C \triangleright \Delta, k[\mathtt{A}] : !\mathtt{B}\langle S\rangle; T} \qquad \lfloor^{\mathrm{T}}|_{\textsc{Zero}}\rceil\ \frac{\mathsf{cosha}(\Gamma) \quad \Delta\ \mathsf{end\ only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta}$$

$$\lfloor^{\mathrm{T}}|_{\textsc{Branch}}\rceil\ \frac{i \in I \quad \Gamma \vdash C_i \triangleright \Delta, k[\mathtt{A}] : T_i \quad I \subseteq J \quad \mathsf{p} : k[\mathtt{A}] \notin \Gamma}{\Gamma \vdash \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}] : k\&\{l_i : C_i\}_{i \in I} \triangleright \Delta, k[\mathtt{B}] : \&\mathtt{B}\{l_j : T_j\}_{j \in J}}$$

**Fig. 4.** Typing Rules for Compositional Choreographies (selection)

be fresh ($\tilde{\mathsf{q}} \notin \Gamma$); the continuation $C$ is checked by updating $\Gamma'$ and $\Delta'$ respectively with the process ownerships for their roles in $k$ and the local types for their behaviour in $k$. $\lfloor^{\mathrm{T}}|_{\textsc{Sel}}\rceil$ deals with selection, checking that the selected label $l_j$ is specified in the local types. In rule $\lfloor^{\mathrm{T}}|_{\textsc{Req}}\rceil$, we check that the choreography requesting the services is not responsible for implementing them, to avoid deadlocks due to the lack of services in parallel required by rule $\lfloor^{\mathrm{C}}|_{\textsc{p-Start}}\rceil$, and that the requesting process behaves as expected by its role in the protocol. Conversely, $\lfloor^{\mathrm{T}}|_{\textsc{Acc}}\rceil$ types an *(acc)* term by ensuring that all the roles for which the choreography is responsible are implemented (the other checks are similar to $\lfloor^{\mathrm{T}}|_{\textsc{Start}}\rceil$). This *distribution* of the responsibilities for implementing the different roles in a protocol is handled by rule $\lfloor^{\mathrm{T}}|_{\textsc{Par}}\rceil$, using the role distribution function $\Gamma_1 \circ \Gamma_2$. Formally, $\Gamma_1 \circ \Gamma_2$ is defined as the union of $\Gamma_1$ and $\Gamma_2$ except for the typing of shared channels with the same name, which are merged with the following rule:

$$a : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle = a : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{D}}\rangle \circ a : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{E}}\rangle \qquad (\tilde{\mathtt{C}} = \tilde{\mathtt{D}} \uplus \tilde{\mathtt{E}})$$

In rule $\lfloor^{\mathrm{T}}|_{\textsc{Zero}}\rceil$ we check that all responsibilities have been implemented and that the sessions in $\Delta$ have been executed. Specifically, predicate $\mathsf{cosha}(\Gamma)$ checks that for every $a : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle$ in $\Gamma$ either (i) $\tilde{\mathtt{C}} = \tilde{\mathtt{B}}$, meaning that $a$ was used only internally with *(start)* terms; or (ii) $\tilde{\mathtt{C}} = \emptyset$, meaning that $a$ is used compositionally in collaboration with other choreographies and all roles that the current choreography is responsible for ($\tilde{\mathtt{C}}$) have been implemented correctly with *(acc)* terms. Rules $\lfloor^{\mathrm{T}}|_{\textsc{Com-s}}\rceil$ and $\lfloor^{\mathrm{T}}|_{\textsc{Branch}}\rceil$ type respectively a sending action and a branching. They are very similar to their complete versions since local types allow us to look at the behaviour of processes independently. They also check that the counterpart for the partial action is not in the continuation, by ensuring that there is not process $\mathsf{q}$ such that $\mathsf{q}$ plays the other role for session $k$ in $\Gamma$, which could obviously lead to a deadlock because process $\mathsf{p}$ would not have another process to communicate with in parallel as required by rule $\lfloor^{\mathrm{C}}|_{\textsc{Sync}}\rceil$.

**Typing Expressiveness.** Our typing system exploits the global information given by complete terms and seamlessly falls back to typical session typing when dealing with partial actions. In particular, $\lfloor {}^{\mathrm{T}}|_{\textsc{Sel}} \rceil$ judges that a choice in a protocol is implemented correctly even if only one of the branches is actually followed. This is sound because we are typing a complete term, and therefore we know that the other branches are not used. This expressiveness is typical of choreography-based models [6,7]. However, such a global knowledge is not available in a partial choreography. For example, in rule $\lfloor {}^{\mathrm{T}}|_{\textsc{Branch}} \rceil$ we cannot know which branch will be selected by the sender and we must therefore require that the receiver process supports at least all the branches specified by the corresponding local type, as in standard session typing for endpoints [12,13].

**Properties.** We conclude this section by presenting the expected main properties of our type system. Below, to state session fidelity, we use the transition of local types $\Delta \xrightarrow{\alpha} \Delta'$ (defined as [13] and fully given in [1]), where $\alpha$ types a partial or complete action. $\alpha \vdash \lambda$ judges that the label $\lambda$ is for the same session as $\alpha$ and respects its roles and carried type. We also extend our typing judgement with the extra environment $\Sigma$, for handling session ownerships with asynchronous delegations at runtime (see [1]).

**Theorem 1 (Typing Soundness).** *Let $\Gamma; \Sigma \vdash C \triangleright \Delta$. Then,*

- (Subject Swap) *$C \simeq_C C'$ implies $\Gamma; \Sigma \vdash C' \triangleright \Delta$.*
- *$C \xrightarrow{\lambda} C'$ implies that there exists $\Delta'$ such that*
  - (Subject Reduction) *$\Gamma'; \Sigma' \vdash C' \triangleright \Delta'$ for some $\Gamma', \Sigma'$;*
  - (Session Fidelity) *if $\lambda$ is a communication on session $k$, then $\Delta \xrightarrow{\alpha} \Delta'$ with $\alpha \vdash \lambda$; else, $\Delta = \Delta'$.*

## 5 Properties of Compositional Choreographies

This section states the main properties of our framework wrt the execution of actual systems composed by endpoints.

**Endpoint Projection (EPP)** generates correct endpoint code from a choreography. Formally, by endpoint code we refer to choreographies that do not contain complete actions. To define the complete EPP, we first define how the behaviour of a single process in a choreography can be projected. We denote this *process projection* of a process $\mathsf{p}$ in a choreography $C$ with $[\![C]\!]_{\mathsf{p}}$. Selected rules of process projection are given below:

$$[\![\mathtt{p[A]}\ \mathbf{starts}\ \widetilde{\mathtt{q[B]}} : a(k); C]\!]_{\mathsf{r}} = \begin{cases} \mathtt{p[A]}\ \mathbf{req}\ \tilde{\mathtt{B}} : a(k); [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathbf{acc}\ \mathsf{r}[\mathtt{C}] : a(k); [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r}[\mathtt{C}] \in \widetilde{\mathtt{q[B]}} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$[\![\mathtt{p[A]}.e\ \texttt{->}\ \mathtt{q[B]}.x : k; C]\!]_{\mathsf{r}} = \begin{cases} \mathtt{p[A]}.e\ \texttt{->}\ \mathtt{B} : k; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathtt{A}\ \texttt{->}\ \mathtt{q[B]}.x : k; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$[\![\mathtt{p[A]}.e\ \texttt{->}\ \mathtt{B} : k; C]\!]_{\mathsf{r}} = \begin{cases} \mathtt{p[A]}.e\ \texttt{->}\ \mathtt{B} : k; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$[\![\mathtt{A}\ \texttt{->}\ \mathtt{q[B]}.x : k; C]\!]_{\mathsf{r}} = \begin{cases} \mathtt{A}\ \texttt{->}\ \mathtt{q[B]}.x : k; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!]_{\mathsf{r}} = \begin{cases} \mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ [\![C_1]\!]_{\mathsf{r}}\ \mathsf{else}\ [\![C_2]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C_1]\!]_{\mathsf{r}} \sqcup [\![C_2]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$[\![\mathtt{A}\ \texttt{->}\ \mathtt{q[B]} : k\&\{l_i : C_i\}_{i \in I}]\!]_{\mathsf{r}} = \begin{cases} \mathtt{A}\ \texttt{->}\ \mathtt{q[B]} : k\&\{l_i : [\![C_i]\!]_{\mathsf{r}}\}_{i \in I} & \text{if } \mathsf{r} = \mathsf{q} \\ \bigsqcup_{i \in I} [\![C_i]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

Process projection follows the structure of the originating choreography. In a *(start)*, we project the active process $\mathtt{p}$ to a request and the service processes $\tilde{\mathtt{q}}$ to (always-available) accepts. In a *(com)*, the sender is projected to a partial sending action and the receiver to a partial receiving action. The projections of *(sel)* and *(del)*, omitted, follow the same principle. Above we also report the rule for projecting *(com-s)* and *(com-r)* to exemplify how we treat partial choreographies: these are simply projected as they are for their respective process, following the structure of the choreography. The projections of conditionals and partial branchings are the only special cases. In a conditional, we project it as it is for the process evaluating the condition, but for all other we merge their behaviours with the *merging* partial operator $\sqcup$ [6]. $C \sqcup C'$ is defined only for partial choreographies that define the behaviour of a single process and returns a choreography isomorphic to $C$ and $C'$ up to branching, where all branches with distinct labels are also included. We use $\sqcup$ also in the projection of *(branch)* terms, where we require the behaviour of all processes not receiving the selection to be merged. As an example, the process projection for process $\mathtt{u}$ in the choreography $C_B$ from our example in § 2 is:

$$[\![C_B]\!]_{\mathtt{u}} = \begin{array}{l} \mathtt{u}[\mathtt{U}]\ \mathbf{req}\ \mathtt{PD} : a(k);\ \mathtt{u}[\mathtt{U}].prod\ \texttt{->}\ \mathtt{PD} : k; \\ \mathtt{PD}\ \texttt{->}\ \mathtt{u}[\mathtt{U}] : k\& \left\{ \begin{array}{l} del : \mathtt{PD}\ \texttt{->}\ \mathtt{u}[\mathtt{U}] : k\langle k''[\mathtt{B}]\rangle;\ \mathtt{u}[\mathtt{B}].addr\ \texttt{->}\ \mathtt{T} : k''; \\ \qquad \mathtt{T}\ \texttt{->}\ \mathtt{u}[\mathtt{U}].ddate : k'', \\ quit : \mathbf{0} \end{array} \right\} \end{array}$$

Using process projection, we can now define the EPP of a whole system. Since different service processes may be started through *(start)* terms on the same shared channel and play the same role, we use $\sqcup$ for merging their behaviours into a single service. We identify these processes with the service grouping operator $\lfloor C \rfloor^a_{\mathtt{A}}$, which computes the set of all service process names in a start or a request in $C$ on shared channel $a$ playing role $\mathtt{A}$. Formally, EPP is the endofunction $[\![C]\!]$ defined in the following.

**Definition 1 (Endpoint Projection).** *Let $C \equiv (\boldsymbol{\nu}\tilde{a}, \tilde{k}, \tilde{\mathtt{p}})\ C_f$, where $C_f$ does not contain* (res) *terms. Then, the EPP of $C$ is:*

$$[\![C]\!] = (\boldsymbol{\nu}\tilde{a}) \left( (\boldsymbol{\nu}\tilde{k}, \tilde{\mathtt{p}}) \left( \prod_{\mathtt{p} \in \mathsf{fn}(C_f)} [\![C_f]\!]_{\mathtt{p}} \right) \mid \prod_{a,\mathtt{A}} \left( \bigsqcup_{\mathtt{p} \in \lfloor C_f \rfloor^a_{\mathtt{A}}} [\![C_f]\!]_{\mathtt{p}} \right) \right)$$

The EPP of a choreography $C$ is the parallel composition of (i) the projections of all active processes and (ii) the merged projections of all service processes started under same shared channel and role. EPP respects the following Lemma, which shows that our model can adequately capture not only typical complete choreographies, but also scale down to describing the behaviour of a single endpoint.

**Lemma 1 (Endpoint Choreographies).** *Let $C$ be restriction-free, contain only partial terms, and be well-typed. If one of the following two conditions apply, then $C = [\![C]\!]$.*

1. *$C = \mathbf{acc}\ \mathtt{q}[\mathtt{B}] : a(k); C'$ and $\mathtt{q}$ is the only free process name in $C'$;*
2. *otherwise, $C$ has only one free process name.*

We refer to choreographies that respect one of the two conditions above as *endpoint choreographies*. They implement either the behaviour of a single always-available service process (1), or that of a single free process (2). The EPP for these choreographies is the identity since they already model the behaviour of only one endpoint.

The projection of services may lead to undesirable behaviour if service roles for shared channels are not distributed correctly. For example, if we put the choreography $C_B$ from § 2 in parallel with a choreography with a conflicting service on shared channel $b$ for role R (which is internally implemented in $C_B$) we obtain a race condition, *even if protocols are correctly implemented*. Consider the following choreography:

$$C_R = \mathbf{acc}\ \mathsf{h}[\mathtt{R}] : b(k');\ \mathtt{PD} \texttt{->} \mathsf{h}[\mathtt{R}].x : k';\ \mathsf{h}[\mathtt{R}].c \texttt{->} \mathtt{PD} : k'$$

If we put the projection of $C_B$ in parallel with that of $C_R$, we get a race condition between the service processes r and h for role R on shared channel $b$. Hence, the projection of process pd may synchronise with the service offered by $C_R$ for creating session $k'$, instead of that by the projection of service process r in $C_B$. Consequently, $C_B$ may not follow its intended behaviour. The distribution of service roles performed by our type system avoids this kind of situations. Observe that normal session typing cannot help us in detecting these problems, because the service process h correctly implements the same communication behaviour for session $k'$ as service process r.

**Main Theorems.** We can now present our main theorems. We build our results on the foundation that the EPP of a choreography is still typable. As in previous work [16,6,7], we need to consider that in the projection of complete choreographies, due to merging, some projected processes may still offer branches that the original complete choreography has discarded with a conditional. Therefore, we state our type preservation result below under the *minimal typing* of choreographies $\vdash_{\mathsf{min}}$, in which the branches in rules $\lfloor {}^{\mathrm{T}} |_{\textsc{Sel}} \rceil$ and $\lfloor {}^{\mathrm{T}} |_{\textsc{Branch}} \rceil$ are typed using the respective minimal branch types.

**Theorem 2 (EPP Type Preservation).** *Let $\Gamma \vdash_{\mathsf{min}} C \triangleright \Delta$. Then, $\Gamma \vdash_{\mathsf{min}} [\![C]\!] \triangleright \Delta$.*

By Theorem 2, it follows that Theorem 1 applies also to the EPP of a choreography. We use this result to prove that EPP correctly implements the behaviour of the originating choreography, by establishing a formal relation between their respective semantics.

**Theorem 3 (EPP Theorem).** *Let $C \equiv (\boldsymbol{\nu} \tilde{a}, \tilde{k}, \tilde{\mathsf{p}})\ C_f$, where $C_f$ is restriction-free, be well-typed. Then,*

1. *(Completeness) $C \xrightarrow{\lambda} C'$ implies $[\![C]\!] \xrightarrow{\lambda} \succ [\![C']\!]$.*
2. *(Soundness) $[\![C]\!] \xrightarrow{\lambda} C'$ implies $C \xrightarrow{\lambda} C''$ and $[\![C'']\!] \prec C'$.*

Above, the *pruning relation* $C \prec C'$ is a strong typed bisimilarity [6] such that $C$ has some unused branches and always-available accepts. $\succ$ is a shortcut for $\prec$ interpreted in the opposite direction.

**Deadlock-freedom and Progress.** We introduce our results on deadlock-freedom and progress mentioned in the Introduction. First, we define deadlock-freedom:

**Definition 2 (Deadlock-freedom).** *We say that choreography $C$ is deadlock-free if either (i) $C \equiv \mathbf{0}$ or (ii) there exist $C'$ and $\lambda$ such that $C \xrightarrow{\lambda} C'$ and $C'$ is deadlock-free.*

In our semantics (Fig. 3) complete terms can always be executed; therefore, choreographies that do not contain partial terms, or *complete choreographies*, are deadlock-free:

**Theorem 4 (Deadlock-freedom for Complete Choreographies).** *Let $C$ be a complete choreography and contain no free variable names. Then, $C$ is deadlock-free.*

By Theorems 3 and 4 we can obtain, as a corollary, that the EPP of well-typed complete choreographies never deadlock.

**Corollary 1 (Deadlock-freedom for EPP).** *Let $C$ be a complete choreography, contain no free variable names, and be well-typed. Then, $[\![C]\!]$ is deadlock-free.*

Our model can also be used to talk of deadlock-freedom *compositionally*. In a compositional setting, a choreography may get stuck because of partial actions that need to be executed in parallel composition with other choreographies. We say that a choreography can *progress* if it can be composed with another choreography such that (i) all free names can be restricted and the resulting system is still well-typed, ensuring that protocols are implemented correctly; and (ii) the composition is deadlock-free. Differently from deadlock-freedom for complete choreographies, progress for partial choreographies does not follow directly from the semantics. For example, the following choreography does not have the progress property:

$$\mathtt{A} \texttt{ -> } \mathtt{q}[\mathtt{B}] : k;\ \mathtt{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B} : k$$

Above, $\mathtt{q}$ is waiting for a message on session $k$ from $\mathtt{A}$, but that role is implemented by process $\mathtt{p}$ in the continuation. Thus, the two partial actions will never synchronise. As shown in § 4, our type system takes care of checking that roles in sessions or services are distributed correctly, avoiding cases such as this one and ensuring progress. In general, if a well-typed choreographies does not contain inner *(par)* terms we know that it can progress, since role distribution ensures that there exists a compatible environment.

**Theorem 5 (Progress for Partial Choreographies).** *Let $C$ be a choreography, be well-typed, and contain no* (par) *terms. Then, there exists $C'$ such that $(\boldsymbol{\nu}\tilde{r})\ (C \mid C')$ with $\tilde{r} = \mathsf{fn}(C \mid C')$, is well-typed and deadlock-free.*

By Theorems 2 and 5, it follows as a corollary that also the EPP of a well-typed choreography can progress:

**Corollary 2 (Progress for EPP).** *Let $C$ contain no free variable names, be well-typed, and contain no* (par) *terms. Then, there exists $C'$ such that $(\boldsymbol{\nu}\tilde{r})\ ([\![C]\!] \mid C')$ with $\tilde{r} = \mathsf{fn}(C \mid C')$, is well-typed and deadlock-free.*

**Correctness of Choreography Composition.** We end this section by presenting results that allow to reason about the composition of choreographies.

**Lemma 2 (Compositional EPP).** *Let $C = C_1 \mid C_2$ be well-typed. Then, $[\![C]\!] \equiv [\![C_1]\!] \mid [\![C_2]\!]$.*

By combining Lemma 2 with the Theorems shown so far, we get the following corollary, which summarises the properties for well-typed compositions of choreographies.

**Corollary 3 (Compositional Choreographies).** *Let $C \mid C'$ be well-typed. Then,*

1. *(EPP Type Preservation) $[\![C]\!] \mid [\![C']\!]$ is well-typed.*
2. *(Completeness) $C \mid C' \xrightarrow{\lambda} C''$ implies $[\![C]\!] \mid [\![C']\!] \xrightarrow{\lambda}\succ [\![C'']\!]$.*
3. *(Soundness) $[\![C]\!] \mid [\![C']\!] \xrightarrow{\lambda} C''$ implies $C \xrightarrow{\lambda} C'''$ and $[\![C''']\!] \prec C''$.*

Our corollary above formally addresses the issues mentioned in the Introduction. Choreographies ($C$ and $C'$ in the corollary) can be developed independently and then their respective projections can be composed.

## 6 Related Work

Previous works have tackled the problem of defining a formal model for choreographies and giving a correct EPP [7,6]. The main difference wrt our work is compositionality: previous models can only capture closed systems, and do not treat a methodology for composing choreographies. A major difficulty wrt composition given by the approach in [7] is that the EPP of a choreography could be untypable with known type systems for session types. Typability of EPP is important to achieve composition, since a programmer may need to reuse a choreography *after* it has been projected. [7] is the only previous work providing an asynchronous semantics for multiparty sessions in choreographies; however, asynchrony is modelled in two different ways in the choreography model and the endpoint model, raising complexity. As a consequence, the EPP Theorem in [7] has a more complex formulation with weak transitions and confluence, whereas ours can be formulated in a stronger form where EPP mimics its original choreography step by step. [6] preserves typability of projections but does not handle neither asynchrony nor multiparty sessions; instead, they type choreographies with binary sessions. We have shown that choreographies can be made compositional by introducing partial terms to perform message passing with the environment, and that it is possible to ensure typability of EPP in a multiparty and asynchronous setting. This is the first work introducing a compositional multiparty session typing for choreographies, exploiting the projection of global types onto local types. Finally, neither of [7,6] handles shared channel passing, and does not treat how to handle delegation in a compositional setting, where sessions may be delegated to external or internal processes.

Multiparty session types have been previously used for typing endpoint programs [13,3,9]. In our setting, endpoint programs can be captured as special cases of partial choreographies. Our global types are taken from [3]. Differently from our framework, these works capture asynchronous communications with dedicated processes that model order-preserving message queues. An approach more similar to ours can be found in the notion of *delayed input* presented in [17]. [3] defines a type system for progress by building additional restrictions on top of standard multiparty session typing; our model yields a simpler analysis, since we can rely on the fact that complete terms in a choreography do not get stuck. Nevertheless, [3] can capture sessions started by more than one active thread. We leave an extension of our model in this direction as future work.

In [2] the authors use a concept similar to our partial choreographies for protocol specifications, to allow a single process to implement more than one role in a protocol. Differently from our approach, these are not fully-fledged system implementations but abstract behavioural types, which are then used to type check endpoint code. In our setting, the techniques in [2] can be seen as a more flexible way of handling the projection from global types to local types. An extension of our type system to allow for a process to play more than one role in a session as in [2,9] is an interesting future work.

The relationship between choreographies and endpoints has been explored in, among others, [5,16,13,6,7]. Our work distinguishes itself by adopting the same calculus for describing choreographies and endpoints, simplifying the technical development.

**Acknowledgements.** Yoshida has been partially supported by the Ocean Observatories Initiative and EPSRC EP/K011715/1, EP/K034413/1 and EP/G015635/1.

## References

1. Additional Resources, http://www.itu.dk/people/fabr/papers/compchor/
2. Baltazar, P., Caires, L., Vasconcelos, V.T., Vieira, H.T.: A Type System for Flexible Role Assignment in Multiparty Communicating Systems. In: Proc. of TGC (2012)
3. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008), Long version at http://www.di.unito.it/~dezani/papers/cdy12.pdf
4. Business Process Model and Notation, http://www.omg.org/spec/BPMN/2.0/
5. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration conformance for system design. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 63–81. Springer, Heidelberg (2006)
6. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. 34(2), 8 (2012)
7. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: POPL, pp. 263–274 (2013)
8. Chor. Programming Language, http://www.chor-lang.org/
9. Deniélou, P.-M., Yoshida, N.: Dynamic multirole session types. In: Proc. of POPL, pp. 435–446. ACM (2011)
10. Deniélou, P.-M., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. LMCS 8(4) (2012)
11. Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C., Yoshida, N.: Scribbling interactions with a formal foundation. In: Natarajan, R., Ojo, A. (eds.) ICDCIT 2011. LNCS, vol. 6536, pp. 55–75. Springer, Heidelberg (2011)
12. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
13. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proc. of POPL, vol. 43(1), pp. 273–284. ACM (2008)
14. italianaSoftware. http://www.italianasoftware.com/.
15. Jolie. Java Orchestration Language Interpreter Engine, http://www.jolie-lang.org/
16. Lanese, I., Guidi, C., Montesi, F., Zavattaro, G.: Bridging the gap between interaction- and process-oriented choreographies. In: Proc. of SEFM, pp. 323–332. IEEE (2008)
17. Merro, M., Sangiorgi, D.: On asynchrony in name-passing calculi. Mathematical Structures in Computer Science 14(5), 715–767 (2004)
18. Montesi, F., Guidi, C., Zavattaro, G.: Composing Services with JOLIE. In: Proc. of ECOWS, pp. 13–22 (2007)
19. PI4SOA (2008), http://www.pi4soa.org
20. Savara. JBoss Community, http://www.jboss.org/savara/
21. W3C WS-CDL Working Group. Web services choreography description language version 1.0 (2004), http://www.w3.org/TR/ws-cdl-10/