Deadlock-Freedom-by-design: Multiparty Asynchronous Global Programming *

Marco Carbone** and Fabrizio Montesi

IT University of Copenhagen, Denmark
{carbonem,fmontesi}@itu.dk

Abstract. We present a choreography model for *globally* programming communicating systems, using multiparty sessions as a central design element. For the first time, we provide an interpretation of *communication asynchrony* and *execution parallelism* directly at the global level, allowing for their reasoning without the need to look at endpoint implementations. We develop a typing discipline for checking our choreographies against multiparty protocol specifications. Our type system improves on the state of the art by (i) defining a new class of type-safe deadlock-free programs some of which are not typable by previous multiparty session typings, (ii) performing type inference on choreographies, and (iii) introducing support for session delegation in choreographies. We give a notion of Endpoint Projection (EPP) which generates correct entity implementations (given as π-calculus terms) from a choreography. Our framework effortlessly guarantees deadlock-freedom of system executions, yielding what we call *deadlock-freedom-by-design*.

Contents

* The authors contributed equally to this work.

** Research supported by the Danish Agency for Science, Technology and Innovation.

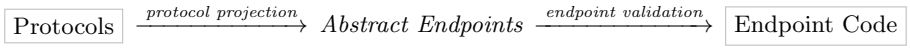
1 Introduction

Global descriptions represent a powerful paradigm for designing communicating system where the programmer gives a global view of how messages are exchanged during execution, instead of separately defining the behaviour of each *endpoint* (entity). Then, the local behaviour of each endpoint can be automatically generated by means of *EndPoint Projection* (EPP). Global descriptions have been studied as models [9,5,17,15], as standards [28,1], and as language implementations [14,24], and can be used at different levels of abstraction. For instance, a *choreography* is a global description of a concrete system implementation. Or, abstract global descriptions can specify the protocols that implementations must follow. Unfortunately, no current framework supports global programming of both implementations and protocols. Therefore, we pose the following question:

> *Can we design a choreography-based language whose programs implement protocols also specified as global descriptions?*

A positive answer would allow software architects to exploit the advantages of global descriptions when dealing with the design of both protocols and implementations. For instance, choreographies naturally guarantee deadlock-freedom of correct EPPs, since they pair i/o communications correctly by construction.

Currently, protocol-based communication-centred programming consists of *globally* designing multiparty protocols and then manually implementing each *endpoint* by referring to *abstract endpoint* descriptions automatically projected from the protocols [15,6,14]:



The approach above deprives the programmer from a global view of the system when writing its implementation. This is error-prone in several aspects; e.g., it can easily lead to deadlocked systems due to protocol interleaving [2]. Instead, in a fully global framework, a programmer would define both protocols and implementations from a global viewpoint, and then deadlock-free endpoint implementations would be automatically generated:

The challenge of reaching such an objective is twofold. First, since we aim at designing a model where choreographies can instantiate different *multiparty* protocols multiple times and interleave their execution, the model should support this methodology by ensuring that these interleavings will not lead to bad behaviour. Second, it is not clear how common aspects of concurrent systems such as *asynchrony* (communications are asynchronous) and *parallelism* (parallel executions) should influence the interpretation of a choreography: choreographies describe communications as atomic actions, making concurrency less explicit. This also raises the question of whether we could use choreographies for simplifying the design of distributed systems without losing in efficiency.

Contributions. The present work provides the following main contributions:

Multiparty Choreographies. We introduce a choreography model with *multiparty* protocol instances as first-class elements (§ 4). We provide an EPP that correctly generates concrete endpoint code from a choreography (§ 6).

Asynchrony and Parallelism. We give a novel interpretation of asynchrony and parallelism directly in the semantics of choreographies, without introducing explicit syntactic primitives (§ 4, Semantics). As a result, we *lift from the programmer the burden of reasoning about asynchrony and parallelism*, without losing efficiency at endpoint level, where executions are still carried out as usual.

Typing and Type Inference. We provide a type system (§ 4.2) for checking choreographies against protocol specifications given as multiparty session types [15]. Our type analysis plays a major rôle in ensuring communication safety of EPP code. Interestingly, our framework can generate correct endpoint code that is not allowed by current multiparty session typings (§ 6, Typing Expressiveness). We also define a type inference technique for supporting the opposite methodology, i.e., developing just choreographies and then automatically extracting the protocols that they implement (§ 4.3).

Delegation. This is the first work to provide a choreography model supporting *session delegation*, a mobility mechanism for delegating the continuation of a protocol (§ 4). Typing delegation (§ 4.2) becomes nontrivial due to asynchrony and parallelism, since messages prior to and after delegation may be interleaved, making it difficult to check that channel ownerships are consistently respected.

Deadlock-Freedom-by-Design. Our framework seamlessly guarantees deadlock freedom (§ 6, Corollary 6.23), a notoriously hard problem in multiparty sessions types [2]. This feature follows from using a choreography as initial design tool.

2 Model Preview

In this section we give an informal description of our model, whose key elements are *protocols* and *choreographies*. A protocol is an abstract specification of the structure of some communications of a system, whereas a choreography describes a concrete system implementing one or more protocols. We represent protocols with *global types* [15], a global description where entities are abstracted as *roles* that communicate following a given conversation structure.

Example 2.1 (Two-buyer protocol). We define a two-buyer protocol between two buyers (`B1` and `B2`) and a seller (`S`) for the payment and delivery of some product:

1. $\texttt{B1} \texttt{ -> } \texttt{S} : \langle \textbf{string} \rangle; \quad \texttt{S} \texttt{ -> } \texttt{B1} : \langle \textbf{int} \rangle; \quad \texttt{S} \texttt{ -> } \texttt{B2} : \langle \textbf{int} \rangle; \quad \texttt{B1} \texttt{ -> } \texttt{B2} : \langle \textbf{int} \rangle;$
2. $\texttt{B2} \texttt{ -> } \texttt{S} : \{\ \mathit{ok} : \texttt{B2} \texttt{ -> } \texttt{S} : \langle \textbf{string} \rangle;\ \texttt{S} \texttt{ -> } \texttt{B2} : \langle \textbf{date} \rangle;\ \texttt{end}, \qquad \mathit{quit} : \texttt{end}\ \}$

Above, `B1`, `B2` and `S` are *roles*. Buyer `B1` sends to a seller `S` a purchase request of type **string**. Then, `S` sends a quote to `B1` and another potential buyer `B2`. Thereafter, `B1` tells `B2` the amount she wishes to contribute with. Afterwards, `B2` notifies `S` whether she has accepted (*ok* or *quit*). If so, `B2` sends to `S` a string (address) and, finally, `S` replies with a delivery date of type **date**. □

In this paper, we introduce a choreography model for globally implementing protocols such as the one above. Its core elements are *threads* and *sessions*. A thread represents a (logical) processing unit that executes a sequence of instructions. Each thread has its own local variables, and can exchange messages with other threads by performing i/o communications. Threads can be programmed to be already active or dynamically created at runtime when needed. A session is an instantiation of a protocol and implements communications between some threads. Sessions can be dynamically created by threads.

Example 2.2 (Two-buyer choreography). We give a choreography for Example 2.1.

1. $\mathsf{b}_1[\mathtt{B1}], \mathsf{b}_2[\mathtt{B2}]$ **start** $\mathsf{s}[\mathtt{S}] : a(k)$;
2. $\mathsf{b}_1[\mathtt{B1}].book$ -> $\mathsf{s}[\mathtt{S}].x_1 : k$;
3. $\mathsf{s}[\mathtt{S}].\mathsf{quote}(x_1)$ -> $\mathsf{b}_1[\mathtt{B1}].y_1 : k$;
4. $\mathsf{s}[\mathtt{S}].\mathsf{quote}(x_1)$ -> $\mathsf{b}_2[\mathtt{B2}].z_1 : k$;
5. $\mathsf{b}_1[\mathtt{B1}].\mathsf{contrib}(y_1)$ -> $\mathsf{b}_2[\mathtt{B2}].z_2 : k$;
6. if $(z_1 - z_2 \leq 100)@\mathsf{b}_2$
7. then $\mathsf{b}_2[\mathtt{B2}]$ -> $\mathsf{s}[\mathtt{S}] : k[ok]$;
8. $\mathsf{b}_2[\mathtt{B2}].addr$ -> $\mathsf{s}[\mathtt{S}].x_2 : k$;
9. $\mathsf{s}[\mathtt{S}].ddate$ -> $\mathsf{b}_2[\mathtt{B2}].z_3 : k$
10. else $\mathsf{b}_2[\mathtt{B2}]$ -> $\mathsf{s}[\mathtt{S}] : k[quit]$

In Line 1, threads b_1, b_2 and s start (create) a session, identified by k, through public channel a playing roles B1, B2 and S respectively. In Lines 2-5, b_1 asks s for book "*book*" and gets back the quote "$\mathsf{quote}(book)$" which is also sent to b_2. Note that, e.g., b_1 uses its local variable y_1 to receive the evaluation of "$\mathsf{quote}(book)$". Then, b_1 tells b_2 the amount she wishes to contribute for the purchase, namely "$\mathsf{contrib}(\mathsf{quote}(book))$". In Line 6, b_2 evaluates the offer received by b_1 in the guard $(z_1 - z_2 \leq 100)@\mathsf{b}_2$. If positive, b_2 communicates her decision with the selection $\mathsf{b}_2[\mathtt{B2}]$ -> $\mathsf{s}[\mathtt{S}] : k[ok]$, sends her address *addr* and receives the delivery date *ddate* (Lines 7-9). Otherwise, b_2 aborts by selecting *quit* in Line 10. □

We can immediately observe that the structure of session k in Example 2.2 is exactly that of the protocol specification given in Example 2.1. The only differences are that data has become explicit and that we introduced the **start** primitive. The latter allows threads to synchronise on a public name, e.g., a in Line 1 above, and dynamically create new threads and sessions. In Line 1 of our example, b_1 and b_2 are already active threads while s is a service thread, i.e., a dynamically spawned thread. Active threads appear on the left-hand side of the **start** keyword, whereas (fresh) service threads appear on its right-hand side. Role annotations, e.g., $\mathsf{b}_1[\mathtt{B1}]$, relate each thread to the role it plays in a session.

Example 2.3 (Two-buyer-helper choreography). Choreographies can also describe multiple, interleaved instances of multiple protocols. Hereafter, we extend the two-buyer choreography from Example 2.2 with two other sessions k' and k'', that b_1 and b_2 will respectively use for asking help in the transaction.

1. ... as Lines 1-5 in Example 2.2 ...

C_1:
2. $\mathsf{b}_1[\mathtt{B}]$ **start** $\mathsf{h}_1[\mathtt{H}] : b(k')$;
3. $\mathsf{b}_1[\mathtt{B}].(y_1/2)$ -> $\mathsf{h}_1[\mathtt{H}].y : k'$;
4. $\mathsf{b}_1[\mathtt{B}]$ -> $\mathsf{h}_1[\mathtt{H}] : k'[done]$;

C_2:
5. $\mathsf{b}_2[\mathtt{B}]$ **start** $\mathsf{h}_2[\mathtt{H}] : b(k'')$;
6. $\mathsf{b}_2[\mathtt{B}].(z_1/2)$ -> $\mathsf{h}_2[\mathtt{H}].z : k''$;
7. $\mathsf{b}_2[\mathtt{B}]$ -> $\mathsf{h}_2[\mathtt{H}] : k''[del]$;
8. $\mathsf{b}_2[\mathtt{B}].z_2$ -> $\mathsf{h}_2[\mathtt{H}].z' : k''$;
9. $\mathsf{b}_2[\mathtt{B}]$ -> $\mathsf{h}_2[\mathtt{H}] : k''\langle\langle k[\mathtt{B2}]\rangle\rangle$;

C_3:
10. if $(y - z' \leq 100)@\mathsf{h}_2$
11. then $\mathsf{h}_2[\mathtt{B2}]$ -> $\mathsf{s}[\mathtt{S}] : k[ok]$;
12. $\mathsf{h}_2[\mathtt{B2}].addr$ -> $\mathsf{s}[\mathtt{S}].x_2 : k$;
13. $\mathsf{s}[\mathtt{S}].ddate$ -> $\mathsf{h}_2[\mathtt{B2}].z'' : k$
14. else $\mathsf{h}_2[\mathtt{B2}]$ -> $\mathsf{s}[\mathtt{S}] : k[quit]$

Line 1 recalls part of the two-buyer choreography. In block C_1, b_1 starts a new session with a helper h_1, asks it to contribute for half of the price (Line 3), and informs it that it does not need to do more (Line 4). On the other hand, in C_2, b_2 does the same with another thread h_2 until Line 7. Differently now, b_2 asks h_2 to continue session k *by taking on its role* (Line 7). Then, it sends the contribution received from b_1 to h_2 (Line 8) and *delegates* the actual session reference (Line 9). Finally, in C_3 thread h_2 follows the two-buyer protocol in the place of b_2. Note that h_1 and h_2 are started through the same public channel b, which acts as a reusable shared channel in standard multiparty session types [15]. □

Our model has two features that interestingly influence a choreography interpretation in subtly different ways. The first is *parallelism*: thread executions may concurrently proceed without any predetermined ordering unless causal constraints are introduced. The second is *asynchrony*: communications are asynchronous, so a thread may send a message to another thread and then immediately proceed before the message has actually been delivered by the network.

For instance, in Example 2.3, the threads whose behaviour is described in blocks C_1 and C_2 are different (b_1 and h_1 for C_1, b_2 and h_2 for C_2). Therefore, their executions may interleave due to parallelism. E.g., b_2 and h_2 may start session k'' *before* b_1 and h_1 start session k'. Even more, k'' may be completely executed before k' is started. Hence, $C_1; C_2$ should be somehow interpreted as $C_2; C_1$ or, more generally, as any interleaving of C_1 and C_2. This suggests a notion of equivalence, *swapping*, formalised in the next section.

Furthermore, in Lines 3 and 4 of Example 2.2, where s sends the quote to b_1 and then b_2, it may happen that b_2 (Line 4) receives the quote before b_1 due to asynchronous messaging. The asynchronous semantics for choreography proposed by this work takes into account situations like this one.

Example 2.4 (OpenID). As a further example, we show how we can model a system that uses OpenID [23], a well-known distributed protocol, where a client (called user) authenticates to a server (called relying party) through a third-party identity provider. First, we give a description of the protocol:

$$\texttt{RP -> IP} : \langle \mathbf{string} \rangle;\ \texttt{U -> IP} : \langle \mathbf{string} \rangle;\ \texttt{IP -> RP} : \{\, \mathit{ok} : \mathsf{end},\quad \mathit{quit} : \mathsf{end} \,\} \tag{1}$$

Above, RP is the relying party, IP is the identity provider and U is the user. According to the protocol description, RP must send to IP a string, namely the username she wishes to authenticate. Then, U must send her credentials to IP, which will finally notify RP of whether the credentials are valid (*ok* or *quit*). The following choreography describes an implementation of the protocol:

$$\begin{array}{ll}
1. & \mathsf{rp}[\texttt{RP}], \mathsf{u}[\texttt{U}]\ \mathbf{start}\ \mathsf{ip}[\texttt{IP}] : a(k); \\
2. & \mathsf{rp}[\texttt{RP}].\mathit{username}\ \texttt{->}\ \mathsf{ip}[\texttt{IP}].u_1 : k;\quad \mathsf{u}[\texttt{U}].\mathsf{authStr}(\mathit{user}_1, \mathit{pwd}_1)\ \texttt{->}\ \mathsf{ip}[\texttt{IP}].y_1 : k; \\
3. & \mathsf{if}\ \mathsf{check}(u_1, y_1)@\mathsf{ip}\ \ \mathsf{then}\ \mathsf{ip}[\texttt{IP}]\ \texttt{->}\ \mathsf{rp}[\texttt{RP}] : k[\mathit{ok}] \\
4. & \qquad\qquad\qquad\ \ \mathsf{else}\ \ \mathsf{ip}[\texttt{IP}]\ \texttt{->}\ \mathsf{rp}[\texttt{RP}] : k[\mathit{quit}]
\end{array} \tag{2}$$

where threads rp, u and ip denote the endpoints of our system. Line 1 is the initiation of the protocol instance k between the three threads, by means of the public name a. Lines 2-4 implement the protocol. □

3 Global Descriptions for Protocols

Syntax of Global Types. As mentioned in the introduction, we define global descriptions of protocols in terms of global types [2,15]. Their syntax follows.

$$
\begin{array}{lll}
G ::= \mathtt{p} \to \mathtt{q} : \langle U \rangle ; G & & \textit{(com)} \\
\quad | \quad \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} & & \textit{(choice)} \\
\quad | \quad \mathsf{end} & & \textit{(end)} \\
\quad | \quad \mathsf{rec}\ \mathbf{t}; G & & \textit{(rec)} \\
\quad | \quad \mathbf{t} & & \textit{(recVar)}
\end{array}
\qquad
\begin{array}{lll}
U ::= S \mid G@\mathtt{p} & & \textit{(value)} \\
S ::= \mathbf{bool} \mid \mathbf{int} \mid \ldots & & \textit{(sort)}
\end{array}
$$

$\mathtt{p} \to \mathtt{q} : \langle U \rangle ; G$ abstracts an interaction from a role $\mathtt{p}$ to a role $\mathtt{q}$ with continuation G, where U, referred to as the *carrying type*, is the type of the exchanged message. U can either be a basic type S or $G@\mathtt{p}$. Communicating $G@\mathtt{p}$ means that a sender role *delegates* to another role her role $\mathtt{p}$ in protocol G. In $\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}$, role $\mathtt{p}$ can select one label l_i and continue as G_i. All other terms [1] are standard.

Semantics of Global Types. Hereby, we introduce a operational semantics for global types that captures precisely the asynchrony of protocols. This is given by a labelled reduction relation:

Definition 3.1 (Global Type Reduction). *Let* global type labels *be defined as*

$$\alpha ::= \ \mathtt{p} \to \mathtt{q} : \langle U \rangle \quad | \quad \mathtt{p} \to \mathtt{q} : l_j$$

Global type reduction *is the minimal labelled relation* $\to$ *on global types such that:*

$$
\begin{array}{rl}
\lfloor {}^{\mathrm{G}} |_{\textsc{Com}} \rceil & \mathtt{p} \to \mathtt{q} : \langle U \rangle ; G \xrightarrow{\mathtt{p} \to \mathtt{q} : \langle U \rangle} G \\
\lfloor {}^{\mathrm{G}} |_{\textsc{Branch}} \rceil & \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} \xrightarrow{\mathtt{p} \to \mathtt{q} : l_j} G_j \quad (j \in I) \\
\lfloor {}^{\mathrm{G}} |_{\textsc{Rec}} \rceil & G[\mathsf{rec}\ \mathbf{t}; G/\mathbf{t}] \xrightarrow{\alpha} G' \quad \Rightarrow \quad \mathsf{rec}\ \mathbf{t}; G \xrightarrow{\alpha} G' \\
\lfloor {}^{\mathrm{G}} |_{\textsc{Swap}} \rceil & G_1 \simeq_{\mathcal{G}} G_1' \quad G_1' \xrightarrow{\alpha} G_2' \quad G_2' \simeq_{\mathcal{G}} G_2 \quad \Rightarrow \quad G_1 \xrightarrow{\alpha} G_2 \\
\lfloor {}^{\mathrm{G}} |_{\textsc{Async-Com}} \rceil & G \xrightarrow{\alpha} G' \quad \mathtt{p} \in \mathsf{roles}(\alpha), \mathtt{q} \notin \mathsf{roles}(\alpha) \quad \Rightarrow \quad \mathtt{p} \to \mathtt{q} : \langle U \rangle ; G \xrightarrow{\alpha} \mathtt{p} \to \mathtt{q} : \langle U \rangle ; G' \\
\lfloor {}^{\mathrm{G}} |_{\textsc{Async-Branch}} \rceil & \begin{pmatrix} G_j \xrightarrow{\alpha} G_j' \\ \mathtt{p} \in \mathsf{roles}(\alpha), \mathtt{q} \notin \mathsf{roles}(\alpha) \\ j \in I \end{pmatrix} \quad \Rightarrow \quad \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} \xrightarrow{\alpha} \mathtt{p} \to \mathtt{q} : \{l_j : G_j'\}
\end{array}
$$

The rules are standard, apart from $\lfloor {}^{\mathrm{G}} |_{\textsc{Swap}} \rceil$, $\lfloor {}^{\mathrm{G}} |_{\textsc{Async-Com}} \rceil$, and $\lfloor {}^{\mathrm{G}} |_{\textsc{Async-Branch}} \rceil$.

In $\lfloor {}^{\mathrm{G}} |_{\textsc{Swap}} \rceil$, the swap relation $\simeq_{\mathcal{G}}$ for global types allows unrelated interactions in a protocol (on different roles) to proceed following an arbitrary order. Formally, $\simeq_{\mathcal{G}}$ is the smallest congruence satisfying the rules reported in Table 1. The swap relation models the fact that different roles in a protocol are implemented by different threads running in parallel. Consequently, $\simeq_{\mathcal{G}}$ makes sequences and choices in a protocol to have a relaxed notion of precedence that is based on *causality* instead of the usual strict syntactic precedence. Briefly, $\lfloor {}^{\mathrm{Gs}} |_{\textsc{Com-Com}} \rceil$ allows for two communications to be swapped if they do not share any roles. Rule

[1] We regard $\mathsf{rec}\ \mathbf{t}; G$ and $\mathbf{t}$ in the standard way [25], i.e., taking an equi-recursive view such that type variables only appear under prefixes.

$$\lfloor^{\text{Gs}}|_{\text{Com-Com}}\rceil \quad \frac{\{\mathtt{p}_1,\mathtt{q}_1\} \cap \{\mathtt{p}_2,\mathtt{q}_2\} = \emptyset}{\mathtt{p}_1\ \texttt{->}\ \mathtt{q}_1 : \langle U_1\rangle;\ \mathtt{p}_2\ \texttt{->}\ \mathtt{q}_2 : \langle U_2\rangle \quad \simeq_{\mathcal{G}} \quad \mathtt{p}_2\ \texttt{->}\ \mathtt{q}_2 : \langle U_2\rangle;\ \mathtt{p}_1\ \texttt{->}\ \mathtt{q}_1 : \langle U_1\rangle}$$

$$\lfloor^{\text{Gs}}|_{\text{Com-Choice}}\rceil \quad \frac{\{\mathtt{p},\mathtt{q}\} \cap \{\mathtt{p}',\mathtt{q}'\} = \emptyset}{\mathtt{p}\ \texttt{->}\ \mathtt{q} : \{l_i : \mathtt{p}'\ \texttt{->}\ \mathtt{q}' : \langle U\rangle;\ G_i\}_{i\in I} \quad \simeq_{\mathcal{G}} \quad \mathtt{p}'\ \texttt{->}\ \mathtt{q}' : \langle U\rangle;\ \mathtt{p}\ \texttt{->}\ \mathtt{q} : \{l_i : G_i\}_{i\in I}}$$

$$\lfloor^{\text{Gs}}|_{\text{Choice-Choice}}\rceil \quad \frac{\{\mathtt{p},\mathtt{q}\} \cap \{\mathtt{p}',\mathtt{q}'\} = \emptyset}{\mathtt{p}\ \texttt{->}\ \mathtt{q} : \{l_i : \mathtt{p}'\ \texttt{->}\ \mathtt{q}' : \{l_j : G_j\}_{j\in J}\}_{i\in I} \quad \simeq_{\mathcal{G}} \quad \mathtt{p}'\ \texttt{->}\ \mathtt{q}' : \{l_j : \mathtt{p}\ \texttt{->}\ \mathtt{q} : \{l_i : G_i\}_{i\in I}\}_{j\in J}}$$

Table 1. Swap Relation Rules for Global Types

$\lfloor^{\text{Gs}}|_{\text{Com-Choice}}\rceil$ allows for a communication to be swapped out (or in, reading the conclusion from right to left) of a choice if it prefixes all branches and does not share any role with the choice. Rule $\lfloor^{\text{Gs}}|_{\text{Choice-Choice}}\rceil$ swaps two choices if they share no roles.

Let us see rule $\lfloor^{\text{G}}|_{\text{Async-Com}}\rceil$ now. It allows actions in the *(com)*-prefixed type G to take place, provided that they do not involve the receiving role $\mathtt{q}$ in the prefix. This captures the asynchronous semantics that we will give for our endpoint processes: as far as communications go, the ordering we are interested in is only that in which messages are received. Rule $\lfloor^{\text{G}}|_{\text{Async-Branch}}\rceil$ is similar to $\lfloor^{\text{G}}|_{\text{Async-Com}}\rceil$, but allows asynchronous actions in a branch to take place by ensuring that the branch will be selected afterwards.

4 A Choreography Model with Multiparty Protocols

4.1 Syntax and Semantics

Syntax. We model our choreography language with the *global calculus*, whose syntax follows:

$$\begin{array}{lll}
C ::= \eta; C & & \textit{(seq)}\\
\quad | \quad \mathsf{if}\, e@\tau\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2 & & \textit{(cond)}\\
\quad | \quad \mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C'\, \mathsf{in}\, C & & \textit{(rec)}\\
\quad | \quad X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) & & \textit{(recVar)}\\
\quad | \quad \mathbf{0} & & \textit{(inact)}\\
\\
\eta ::= \tau_1[\mathtt{p}_1], \ldots, \tau_n[\mathtt{p}_n]\ \mathbf{start}\ \tau_{n+1}[\mathtt{p}_{n+1}], \ldots, \tau_m[\mathtt{p}_m] : a(k) & & \textit{(start)}\\
\quad | \quad \tau_1[\mathtt{p}].e\ \texttt{->}\ \tau_2[\mathtt{q}].x : k & & \textit{(com)}\\
\quad | \quad \tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}] : k[l] & & \textit{(sel)}\\
\quad | \quad \tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}] : k\langle\!\langle k'[\mathtt{p}']\rangle\!\rangle & & \textit{(del)}
\end{array}$$

A term C is called *choreography*; τ is a thread (running process); $\mathtt{p}$ is a role; a is a public channel; k is a session channel; v is a generic data value, e.g., an

integer; x is a placeholder for values; and l is a *label* for branching. e denotes a first-order expression, whose syntax we leave unspecified.

Interactions between threads are denoted by η. Term *(start)* denotes the initiation of a multiparty session: threads τ_i (for $1 \leq i \leq m$) wish to start multiparty session a and tag it with a fresh session channel (identifier) k. The first n threads are ordinary threads running in parallel, while threads $\tau_{n+1}, \ldots, \tau_m$ denote replicated services. We assume that $m \geq 2$ (a session has at least two participants) and $n \geq 1$ (a session is started by at least one running thread). The p_i's denote the roles played by the threads in the session. In-session communication is denoted by the term $\tau_1[\mathsf{p}].e$ -> $\tau_2[\mathsf{q}].x : k$ where thread τ_1 sends the evaluation of expression e to thread τ_2 which binds it to variable x. We will omit irrelevant variables in our examples. In term *(sel)*, τ_1 communicates to τ_2 her wish to select branch l. Through *(del)*, thread τ_1 can delegate to τ_2 through k her role in session k'. In *(cond)*, expression e is labelled with a thread name in order to specify where the evaluation of e is made. In *(rec)*, we assume that each expression is located at a thread in $\tilde{\tau}$. All other terms are standard. Session channels and threads can be bound. In *(start)*, $\tau_1, \ldots, \tau_n$ are free while k and $\tau_{n+1}, \ldots, \tau_m$ bind in the subterm C (since services are replicated entities that spawn a new thread upon invocation). In (*com*) x is a binder. The free and bound session channels, term variables and threads are defined as usual. We often omit $\mathbf{0}$ and empty vectors.

Remark 4.1. In *(com)*, *(sel)*, and *(del)* threads are deliberately annotated with roles to make programming a choreography clearer. Technically, this makes endpoint projection in § 6 simpler. Additionally, it allows for a stronger semantics and, consequently, stronger results about our typing system and endpoint projection.

Example 4.2 (Three-Buyer Protocol). We implement an example from [2], the *three-buyer* protocol, where a session identical to the one in the two-buyer protocol is interleaved with another session where b_2 asks a third buyer, b_3, to contribute to her share for the purchase. The choreography is given as follows:

1. ... as Lines 1-9 in Example 2.2 ...
2. else $\mathsf{b}_2[\mathtt{B1}]$ **start** $\mathsf{b}_3[\mathtt{B2}] : b(k')$;
3. $\quad \mathsf{b}_2[\mathtt{B1}].(z_1 - z_2 - 100)$ -> $\mathsf{b}_3[\mathtt{B2}].w_1 : k'$;
4. $\quad \mathsf{b}_2[\mathtt{B1}]$ -> $\mathsf{b}_3[\mathtt{B2}] : k' \langle\!\langle k[\mathtt{B2}] \rangle\!\rangle$;
5. $\quad$ if $(w_1 \leq 100)@\mathsf{b}_3$
6. $\quad$ then $\mathsf{b}_3[\mathtt{B2}]$ -> $\mathsf{s}[\mathtt{S}] : k[ok]$;
7. $\quad\quad \mathsf{b}_3[\mathtt{B2}].addr$ -> $\mathsf{s}[\mathtt{S}] : k$;
8. $\quad\quad \mathsf{s}[\mathtt{B2}].ddate$ -> $\mathsf{b}_3[\mathtt{B2}] : k$
9. $\quad$ else $\mathsf{b}_3[\mathtt{B2}]$ -> $\mathsf{s}[\mathtt{S}] : k[quit]$

Line 1 recalls the beginning of the two-buyer protocol. However, the else-branch in Line 2 differs from the previous example since b_2 initiates a new session rather than terminating. This session is started with another buyer, b_3, playing role $\mathtt{B2}$. Once the session is started, b_2 tells b_3 how much she wants him to contribute (Line 3). Subsequently, she also delegates the termination of the session k in

Line 4. Now, all the decisions will be made by b_3 who, based on the value w_1, will communicate to s his decision. □

Runtime syntax. In order to define the semantics of the global calculus, we need to extend the syntax presented above with standard name restriction. In the sequel, we use r (for restricted name) for referring to both threads and channels.

$$C ::= \ldots \mid (\boldsymbol{\nu} r)\, C \qquad \textit{(restriction)}$$

In the remainder of the paper, $(\boldsymbol{\nu} r_1, \ldots, r_n)$ is a shortcut for $(\boldsymbol{\nu} r_1) \ldots (\boldsymbol{\nu} r_n)$.

Structural Congruence. Structural congruence $\equiv$ for C is the smallest congruence supporting α-conversion and satisfying the following rules:

$$(\boldsymbol{\nu} r)\, \mathbf{0} \equiv \mathbf{0} \qquad (\boldsymbol{\nu} r)\, (\boldsymbol{\nu} r')\, C \equiv (\boldsymbol{\nu} r')\, (\boldsymbol{\nu} r)\, C \qquad \mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C \,\mathsf{in}\, \mathbf{0} \;\equiv\; \mathbf{0}$$

$$\mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C' \,\mathsf{in}\, (\boldsymbol{\nu} r)\, C \;\equiv\; (\boldsymbol{\nu} r)\, \mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C' \,\mathsf{in}\, C \quad \text{if} \quad r \notin \mathsf{fn}(C')$$

$$\mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C' \,\mathsf{in}\, X(\widetilde{v@\tau}, \tilde{k}, \tilde{\tau}) \;\equiv\; \mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C' \,\mathsf{in}\, C'[\tilde{v}/\tilde{x}@\tilde{\tau}]$$

Above, $C[v/x@\tau]$ is a *smart substitution*, described in the following paragraph.

Reduction Semantics. We equip our global calculus with a labelled reduction semantics whose labels, denoted by λ, are formally defined as follows:

$$\lambda \quad ::= \quad \eta \quad \mid \quad \mathsf{if}@\tau \quad \mid \quad (\boldsymbol{\nu} r)\, \lambda$$

The rules defining the operational semantics are reported in Table 2. We briefly comment the most relevant ones. Rule $\lfloor^{\mathrm{C}}|_{\textsc{Start}}\rceil$ gives meaning to session initiation. In the reductum, the restrictions $(\boldsymbol{\nu} k, \tau_{n+1}, \ldots, \tau_m)$ denote that a new session k has been created and that threads $\tau_{n+1}, \ldots, \tau_m$ have been spawned. In $\lfloor^{\mathrm{C}}|_{\textsc{Com}}\rceil$, we substitute variable x with value v (the evaluation of the expression e in a system that we leave unspecified). This is done by a *smart substitution*: $C[v/x@\tau]$ substitutes x with v only under the free name τ in C in order to simulate separate states between sessions. Session delegation is treated by rule $\lfloor^{\mathrm{C}}|_{\textsc{Del}}\rceil$. Observe that, since we are at the choreography level, nothing happens to the subterm C. Apart from $\lfloor^{\mathrm{C}}|_{\textsc{Swap}}\rceil$, the remaining rules are standard.

In rule $\lfloor^{\mathrm{C}}|_{\textsc{Swap}}\rceil$ the *swap relation* $\simeq_{\mathcal{C}}$ is defined as the smallest congruence satisfying the rules in Table 3. Similarly to the semantics of global types, $\simeq_{\mathcal{C}}$ allows for terms with different threads as actors to exchange places in a choreography. Rule $\lfloor^{\mathrm{Cs}}|_{\textsc{Inter}}\rceil$ allows for two interactions η and η' to be swapped if they do not share any thread names (calculated by the function $\mathsf{threads}$). Rule $\lfloor^{\mathrm{Cs}}|_{\textsc{Cond-Inter}}\rceil$ allows for an interaction η to be swapped out (or in, reading the conclusion from right to left) of an if-then-else if it prefixes both branches of the choice and does not involve the thread that checks the condition. Rule $\lfloor^{\mathrm{Cs}}|_{\textsc{Cond-Cond}}\rceil$ allows to swap two if-then-else constructs, if the threads checking the two conditions are different. Branches C_1' and C_2 are swapped to preserve the semantic meaning of the term wrt the evaluations of the conditions.

Remark 4.3. Note that such a relation allows for the global modelling of processes running in parallel although our choreographies lack a parallel or mixed

$$\lfloor^{\mathrm{C}}|_{\textsc{Start}}\rceil\ \eta = \tau_1[\mathtt{p}_1], \ldots, \tau_n[\mathtt{p}_n]\ \mathbf{start}\ \tau_{n+1}[\mathtt{p}_{n+1}], \ldots, \tau_m[\mathtt{p}_m] : a(k) \quad \Rightarrow \quad \eta; C \xrightarrow{\eta} (\boldsymbol{\nu} k, \tau_{n+1}, \ldots, \tau_m)\, C$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Com}}\rceil\ \eta = \tau_1[\mathtt{p}].e \texttt{ -> } \tau_2[\mathtt{q}].x : k \quad \Rightarrow \quad \eta; C \xrightarrow{\eta[v/e]} C[v/x@\tau_2] \quad (e \downarrow v)$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Sel}}\rceil\ \eta = \tau_1[\mathtt{p}] \texttt{ -> } \tau_2[\mathtt{q}] : k[l] \quad \Rightarrow \quad \eta; C \xrightarrow{\eta} C$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Del}}\rceil\ \eta = \tau_1[\mathtt{p}] \texttt{ -> } \tau_2[\mathtt{q}] : k\langle\!\langle k'[\mathtt{r}] \rangle\!\rangle \quad \Rightarrow \quad \eta; C \xrightarrow{\eta} C$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil\ C \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{r})\, C' \quad \Rightarrow \quad \eta; C \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{r})\, \eta; C' \qquad \begin{pmatrix} \mathsf{snd}(\eta) \in \mathsf{fn}(\lambda) & \eta \neq \textit{(start)} \\ \mathsf{rcv}(\eta) \notin \mathsf{fn}(\lambda) & \tilde{r} \notin \mathsf{fn}(\eta) \end{pmatrix}$$

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

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

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

$$\lfloor^{\mathrm{C}}|_{\textsc{Swap}}\rceil\ C_1 \simeq_{\mathcal{C}} C_1' \quad C_1' \xrightarrow{\lambda} C_2' \quad C_2' \simeq_{\mathcal{C}} C_2 \quad \Rightarrow \quad C_1 \xrightarrow{\lambda} C_2$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Struct}}\rceil\ C_1 \equiv C_1' \quad C_1' \xrightarrow{\lambda} C_2' \quad C_2' \equiv C_2 \quad \Rightarrow \quad C_1 \xrightarrow{\lambda} C_2$$

Table 2. Semantics of the Global Calculus

choice operator in their syntax. For example, the fact that choreography

$$\tau_1[\mathtt{p}].e \texttt{ -> } \tau_2[\mathtt{q}].x : k; \tau_3[\mathtt{p}'].e' \texttt{ -> } \tau_4[\mathtt{q}'].x' : k'$$

can be swapped to

$$\tau_3[\mathtt{p}'].e' \texttt{ -> } \tau_4[\mathtt{q}'].x' : k'; \tau_1[\mathtt{p}].e \texttt{ -> } \tau_2[\mathtt{q}].x : k$$

shows that the two interactions between τ_1 and τ_2 and between τ_3 and τ_4 occur in parallel. □

Example 4.4 (Semantics of the Two-Buyer Choreography). We give the semantics of Lines 1-3 of the choreography given in Example 2.2, where $\tilde{r} = k\ \mathsf{s}$:

$$\text{Lines 1-3 from Example 2.2} \quad \to \quad (\boldsymbol{\nu}\tilde{r}) \begin{pmatrix} \mathsf{b}_1[\mathtt{B1}].\mathit{book} \texttt{ -> } \mathsf{s}[\mathtt{S}].x : k; \\ \mathsf{s}[\mathtt{S}].\mathsf{quote}(x) \texttt{ -> } \mathsf{b}_1[\mathtt{B1}].y : k \end{pmatrix} \to$$

$$\to (\boldsymbol{\nu}\tilde{r})\ (\mathsf{s}[\mathtt{S}].\mathsf{quote}(\mathit{book}) \texttt{ -> } \mathsf{b}_1[\mathtt{B1}].y : k) \quad \to \quad (\boldsymbol{\nu}\tilde{r})\, \mathbf{0}$$ □

Choreographies enjoy the deadlock-freedom property. Intuitively, this holds because every term but **0** in the syntax for choreographies has a corresponding semantic rule that can reduce it. The only restriction that we need to make is that there are no free variables under any thread, which would prevent expressions to be evaluated in $\lfloor^{\mathrm{C}}|_{\textsc{Com}}\rceil$. Formally,

Theorem 4.5 (Deadlock-Freedom). *Let $C \not\equiv \mathbf{0}$ and assume that there are no free variable names in C. Then there exist C', λ such that $C \xrightarrow{\lambda} C'$.*

$$\lfloor^{\text{Cs}}|_{\text{Inter}}\rceil \quad \frac{\mathsf{thr}(\eta) \ \cap \ \mathsf{thr}(\eta') \ = \ \emptyset}{\eta;\ \eta' \quad \simeq_{\mathcal{C}} \quad \eta';\ \eta}$$

$$\lfloor^{\text{Cs}}|_{\text{Cond-Inter}}\rceil \quad \frac{\tau \ \notin \ \mathsf{thr}(\eta)}{\mathsf{if}\, e@\tau\, \mathsf{then}\, (\eta; C_1)\, \mathsf{else}\, (\eta; C_2) \quad \simeq_{\mathcal{C}} \quad \eta;\ \mathsf{if}\, e@\tau\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2}$$

$$\lfloor^{\text{Cs}}|_{\text{Cond-Cond}}\rceil \quad \frac{\tau \neq \tau'}{\begin{array}{c}\mathsf{if}\, e@\tau\, \mathsf{then}\, (\, \mathsf{if}\, e'@\tau'\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2\,)\, \mathsf{else}\, (\, \mathsf{if}\, e'@\tau'\, \mathsf{then}\, C_1'\, \mathsf{else}\, C_2'\,) \\ \simeq_{\mathcal{C}} \\ \mathsf{if}\, e'@\tau'\, \mathsf{then}\, (\mathsf{if}\, e@\tau\, \mathsf{then}\, C_1\, \mathsf{else}\, C_1')\, \mathsf{else}\, (\mathsf{if}\, e@\tau\, \mathsf{then}\, C_2\, \mathsf{else}\, C_2')\end{array}}$$

Table 3. Swap Relation Rules for Choreographies

4.2 Global Typing

We now introduce our multiparty session typing discipline which guarantees that protocol instances follow consistently the specification given with global types.

Environments. We use four kinds of typing environments, namely the *service environment*, the *thread environment*, the *delegation environment*, and the *session environment*. Formally, their syntax is given as follows:

(Service Env)	$\Gamma \ ::= \Gamma,\ a\langle \tilde{\mathtt{p}} \parallel \tilde{\mathtt{q}} \rangle : G \ \mid \ \Gamma,\ x@\tau : S \ \mid \ \Gamma,\ X : (\Gamma, \Theta, \Sigma, \Delta) \ \mid \ \emptyset$	
(Thread Env)	$\Theta \ ::= \Theta,\ \tau : k[\mathtt{p}] \ \mid \ \emptyset$	
(Delegation Env)	$\Sigma \ ::= \Sigma, k[\mathtt{p}] : \tilde{\eta} \ \mid \ \emptyset$	where each η in $\tilde{\eta}$ is a *(del)*
(Session Env)	$\Delta \ ::= \Delta,\ k : G \ \mid \ \emptyset$	

A service environment carries the global type of each public channel, which specifies how a session has to be executed after initialisation. In $a\langle \tilde{\mathtt{p}} \parallel \tilde{\mathtt{q}} \rangle$, $\tilde{\mathtt{p}}$ specifies the initiator roles while $\tilde{\mathtt{q}}$ the roles of the replicated services. Γ also keeps the *sort* type of each variable and all environments for recursion usage. A thread environment keeps track of which role each thread is playing in a session. A delegation environment tracks the asynchronous delegations performed at runtime. Finally, a session environment records the (global) type of each running session k. Since a delegation environment is used only for typing runtime choreographies, we assume it to be always empty when typing programs.

We assume that we can write $\Gamma, a : G$ only if a does not occur in Γ, briefly $a \notin \mathsf{dom}(\Gamma)$; the same holds for Δ wrt sessions k. Furthermore, we can write $\Theta, \tau : k[\mathtt{p}]$ only if τ is not associated to any other role in the same session k in Θ. Consequently, a thread can participate to multiple sessions playing different roles, but it can not participate to the same session with more than one role. Finally, we write $\Sigma, k[\mathtt{p}] : \tilde{\eta}$ only if $k[\mathtt{p}]$ does not appear in Σ.

We define some auxiliary functions for handling thread and delegation environments. $\mathsf{delegates}(\Sigma, \tau, k[\mathtt{p}])$ is a predicate true iff $\Sigma = \Sigma', k[\mathtt{p}] : \tilde{\eta}$ and there is a delegation in $\tilde{\eta}$ with τ as sender. $\mathsf{delegated}(\Sigma, k[\mathtt{p}])$ is a predicate true iff $\Sigma = \Sigma', k[\mathtt{p}] : \tilde{\eta}$ and $\tilde{\eta}$ is not empty. $\mathsf{rem}(\Sigma, \tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}] : k\langle\!\langle k'[\mathtt{p}'] \rangle\!\rangle)$ returns Σ after removing $\tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}] : k\langle\!\langle k'[\mathtt{p}'] \rangle\!\rangle$ from the delegations associated to $k'[\mathtt{p}']$.

$\mathsf{add}(\Sigma, \tau_1[\mathtt{p}] \texttt{->} \tau_2[\mathtt{q}] : k\langle\langle k'[\mathtt{p}']\rangle\rangle)$ returns Σ after adding $\tau_1[\mathtt{p}] \texttt{->} \tau_2[\mathtt{q}] : k\langle\langle k'[\mathtt{p}']\rangle\rangle$ to the delegations associated to $k'[\mathtt{p}']$. Finally, $\Theta[\tau \mapsto k[\mathtt{p}]]$ returns $\Theta', \tau : k[\mathtt{p}]$, where Θ' is obtained from Θ by removing, for all τ', $\tau' : k[\mathtt{p}]$ in it.

Global Typing. Typing judgements have the shape $\Gamma;\ \Theta \vdash_\Sigma C \triangleright \Delta$. Intuitively, C is well-typed provided that public channels are used according to Γ, threads own channels according to Θ and Σ, and session channels are used according to Δ. Relation $\vdash$ is defined by the rules in Table 4.

We comment the most relevant rules. Rule $\lfloor^{\text{GT}}|_{\text{START}}\rceil$ types the term *(start)* by checking that, in the subterm C, session k is used according to the type G of public channel a global types. In there, the swap relation $\simeq_{\mathcal{G}}$ for global types is used to accept choreographies up to admissible swaps due to asynchrony. Also, each thread τ_i is checked to play role $\mathtt{p}_i$ in C when using session k. We require that all $\mathtt{p}_i$'s must occur in G ($\mathsf{roles}(G)$) enforcing that each role communicates at least once in the session. We abuse notation $\tau_{n+1}, \ldots, \tau_m \notin \Theta$ for checking that threads $\tau_{n+1}, \ldots, \tau_m$ are not (associated to any session) in Θ, ensuring that the bound thread names are fresh. In $\lfloor^{\text{GT}}|_{\text{COM}}\rceil$, we check that, given an interaction between τ_1 and τ_2 over session channel k, the global type for k in the session environment requires a communication of type S between role $\mathtt{p}$ and role $\mathtt{q}$ such that τ_1 plays role $\mathtt{p}$ and τ_2 plays role $\mathtt{q}$ according to the thread environment and expression e has type S according to Γ. Rule $\lfloor^{\text{GT}}|_{\text{SEL}}\rceil$ deals with selection and is similar to $\lfloor^{\text{GT}}|_{\text{COM}}\rceil$, although we now check that the chosen label is among the ones allowed by the type. $\lfloor^{\text{GT}}|_{\text{DEL}}\rceil$ deals with session delegation and checks (in the session environment) that the carrying type specified in the type of k is that of the continuation of k', up-to swapping.

Properties of the Type System. We conclude this section with the results on our typing system. Below, we report the substitution Lemma. Note that we do not require substitution for session channels, threads and recursion variables [27].

Lemma 4.6 (Substitution). *Assume* $\Gamma;\ \Theta \vdash_\Sigma C \triangleright \Delta$. *Then* $\Gamma \vdash x@\tau : S$, $v : S$ *implies* $\Gamma;\ \Theta \vdash_\Sigma C[v/x@\tau] \triangleright \Delta$;

Proof. Immediate by induction on the typing rules. □

Typing is preserved by structural congruence.

Lemma 4.7 (Subject Congruence). *Assume* $\Gamma;\ \Theta \vdash_\Sigma C \triangleright \Delta$. *Then* $C \equiv C'$ *implies* $\Gamma;\ \Theta \vdash_\Sigma C' \triangleright \Delta$ *(up to* α*-renaming).*

Proof. The proof is standard by induction on the rules defining the structural congruence relation reported in § 4.1. □

The following Lemma formalises the relationship between thread environments and delegation environments.

Lemma 4.8 (Asynchronous delegation). *Let* $\lambda = \tau_1[\mathtt{p}] \texttt{->} \tau_2[\mathtt{q}] : k\langle\langle k'[\mathtt{p}']\rangle\rangle$. *Then,* $\Gamma;\ \Theta[\tau_2 \mapsto k'[\mathtt{p}']] \vdash_{\mathsf{rem}(\Sigma,\lambda)} C \triangleright \Delta$ *implies* $\Gamma;\ \Theta \vdash_{\mathsf{add}(\Sigma,\lambda)} C \triangleright \Delta$.

$$\lfloor^{\text{GT}}|_{\text{START}}\rceil\ \frac{\Gamma\ \vdash\ a\langle \mathtt{p}_1,\ldots,\mathtt{p}_n\ ||\ \mathtt{p}_{n+1},\ldots,\mathtt{p}_m\rangle : G \qquad \{\mathtt{p}_1,\ldots,\mathtt{p}_m\} = \mathsf{roles}(G) \qquad G \simeq_{\mathcal{G}} G' \qquad \Gamma;\ \Theta, \tau_1 : k[\mathtt{p}_1],\ldots,\tau_m : k[\mathtt{p}_m]\ \vdash_\Sigma\ C\ \triangleright\ \Delta, k : G' \qquad \tau_{n+1},\ldots,\tau_m \notin \Theta}{\Gamma;\ \Theta\ \vdash_\Sigma\ \tau_1[\mathtt{p}_1],\ldots,\tau_n[\mathtt{p}_n]\ \mathbf{start}\ \tau_{n+1}[\mathtt{p}_{n+1}],\ldots,\tau_m[\mathtt{p}_m] : a(k); C\ \triangleright\ \Delta}$$

$$\lfloor^{\text{GT}}|_{\text{COM}}\rceil\ \frac{\big(\Theta\ \vdash\ \tau_1 : k[\mathtt{p}]\ \vee\ \mathsf{delegates}(\Sigma, \tau_1, k[\mathtt{p}])\big) \qquad \Theta\ \vdash\ \tau_2 : k[\mathtt{q}] \qquad \neg\mathsf{delegated}(\Sigma, k[\mathtt{q}]) \qquad \Gamma;\ \Theta\ \vdash_\Sigma\ C\ \triangleright\ k : G, \Delta \qquad \Gamma\ \vdash\ e@\tau : S \qquad \Gamma\ \vdash\ x@\tau : S}{\Gamma;\ \Theta\ \vdash_\Sigma\ \tau_1[\mathtt{p}].e\ \texttt{->}\ \tau_2[\mathtt{q}].x : k; C\ \triangleright\ k\ :\ \mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle S\rangle; G, \Delta}$$

$$\lfloor^{\text{GT}}|_{\text{SEL}}\rceil\ \frac{\big(\Theta\ \vdash\ \tau_1 : k[\mathtt{p}]\ \vee\ \mathsf{delegates}(\Sigma, \tau_1, k[\mathtt{p}])\big) \qquad \Theta\ \vdash\ \tau_2 : k[\mathtt{q}] \qquad \neg\mathsf{delegated}(\Sigma, k[\mathtt{q}]) \qquad \Gamma;\ \Theta\ \vdash_\Sigma\ C\ \triangleright\ k : G_j, \Delta \qquad j \in I}{\Gamma;\ \Theta\ \vdash_\Sigma\ \tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}] : k[l_j]; C\ \triangleright\ k\ :\ \mathtt{p}\ \texttt{->}\ \mathtt{q} : \{l_i : G_i\}_{i\in I}}$$

$$\lfloor^{\text{GT}}|_{\text{DEL}}\rceil\ \frac{\big(\Theta\ \vdash\ \tau_1 : k[\mathtt{p}]\ \vee\ \mathsf{delegates}(\Sigma, \tau_1, k[\mathtt{p}])\big) \qquad \Theta\ \vdash\ \tau_2 : k[\mathtt{q}] \qquad \neg\mathsf{delegated}(\Sigma, k[\mathtt{q}]) \qquad \Theta\ \vdash\ \tau_1 : k'[\mathtt{p}'] \qquad G' \simeq_{\mathcal{G}} G'' \qquad \neg\mathsf{delegates}(\Sigma, \tau_1, k'[\mathtt{p}']) \qquad \Gamma;\ \Theta[\tau_2 \mapsto k'[\mathtt{p}']]\ \vdash_{\mathsf{rem}(\Sigma, \tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}] : k\langle\!\langle k'[\mathtt{p}']\rangle\!\rangle)}\ C\ \triangleright\ k : G, k' : G', \Delta}{\Gamma;\ \Theta\ \vdash_\Sigma\ \tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}] : k\langle\!\langle k'[\mathtt{p}']\rangle\!\rangle; C\ \triangleright\ k\ :\ \mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle G''@\mathtt{p}'\rangle; G,\ k'\ : G',\ \Delta}$$

$$\lfloor^{\text{GT}}|_{\text{IF}}\rceil\ \frac{\Gamma\ \vdash\ e@\tau : \mathbf{bool} \quad \Gamma;\ \Theta\ \vdash_\Sigma\ C_1\ \triangleright\ \Delta \quad \Gamma;\ \Theta\ \vdash_\Sigma\ C_2\ \triangleright\ \Delta}{\Gamma;\ \Theta\ \vdash_\Sigma\ \mathsf{if}\, e@\tau\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2\ \triangleright\ \Delta}$$

$$\lfloor^{\text{GT}}|_{\text{ZERO}}\rceil\ \frac{\Delta\ \mathsf{end\ only}}{\Gamma;\ \Theta\ \vdash_\Sigma\ \mathbf{0}\ \triangleright\ \Delta} \qquad \lfloor^{\text{GT}}|_{\text{VAR}}\rceil\ \frac{\Gamma = \Gamma_{\mathsf{srv}}, \Gamma' \qquad \Delta'\ \mathsf{end\ only}}{\Gamma, X : (\Gamma', \Theta, \Sigma, \Delta);\ \Theta\ \vdash_\Sigma\ X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau})\ \triangleright\ \Delta, \Delta'}$$

$$\lfloor^{\text{GT}}|_{\text{REC}}\rceil\ \frac{\Gamma, X : (\Gamma|_{\widetilde{x@\tau}}, \Theta|_{\tilde{\tau}}, \Sigma|_{\tilde{k},\tilde{\tau}}, \Delta|_{\tilde{k}});\ \Theta\ \vdash_\Sigma\ C\ \triangleright\ \Delta \qquad \Gamma_{\mathsf{rec}}, \Gamma_{\mathsf{srv}}, \Gamma|_{\widetilde{x@\tau}}; \Theta|_{\tilde{\tau}}\ \vdash_{\Sigma|_{\tilde{k},\tilde{\tau}}}\ C'\ \triangleright\ \Delta|_{\tilde{k}}}{\Gamma;\ \Theta\ \vdash_\Sigma\ \mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C'\, \mathsf{in}\, C\ \triangleright\ \Delta}$$

where
- Γ_{rec} is the portion of Γ with only recursion variables plus $X : (\Gamma|_{\widetilde{x@\tau}}, \Theta|_{\tilde{\tau}}, \Sigma|_{\tilde{k},\tilde{\tau}}, \Delta|_{\tilde{k}})$,
- Γ_{srv} is the portion of Γ with only services,
- $\Gamma|_{\widetilde{x@\tau}}$ is the portion of Γ restricted to $\widetilde{x@\tau}$,
- $\Theta|_{\tilde{\tau}}$ is the portion of Θ restricted to $\tilde{\tau}$,
- $\Sigma|_{\tilde{k},\tilde{\tau}}$ is the portion of Σ restricted to $\tilde{k}$ and $\tilde{\tau}$,
- $\Delta|_{\tilde{k}}$ is the portion of Δ restricted to $\tilde{k}$.

$$\lfloor^{\text{GT}}|_{R^{\text{CRES}}}\rceil\ \frac{\Gamma;\ \Theta, \Theta'\ \vdash_\Sigma\ C\ \triangleright\ \Delta, k : G \quad k \notin \Theta}{\Gamma;\ \Theta\ \vdash_\Sigma\ (\boldsymbol{\nu} k)\, C\ \triangleright\ \Delta} \qquad \lfloor^{\text{GT}}|_{R^{\text{TRES}}}\rceil\ \frac{\Gamma, \Gamma';\ \Theta, \Theta'\ \vdash_\Sigma\ C\ \triangleright\ \Delta \quad \tau \notin \Gamma \quad \tau \notin \Theta}{\Gamma;\ \Theta\ \vdash_\Sigma\ (\boldsymbol{\nu}\tau)\, C\ \triangleright\ \Delta}$$

Table 4. Typing Rules for the Global Calculus

Proof. Immediate from the typing rules. □

The following Lemma establishes that the swap relations $\simeq_{\mathcal{G}}$ and $\simeq_{\mathcal{C}}$ are in harmony.

Lemma 4.9 (Subject Swapping). *Assume* $\Gamma;\ \Theta\ \vdash_{\Sigma}\ C\ \triangleright\ \Delta$, *where* $\Delta = \{k_i : G_i \mid i \in I\}$. *Then* $C \simeq_{\mathcal{C}} C'$ *implies that there exists* $\Delta' = \{k_i : G_i' \mid G_i \simeq_{\mathcal{G}} G_i' \text{ and } i \in I\}$ *such that* $\Gamma;\ \Theta\ \vdash_{\Sigma}\ C'\ \triangleright\ \Delta'$.

Proof. Immediate, by induction on the rules for swapping in choreographies (Table 3), showing that for each rule there is a corresponding rule for performing the appropriate swapping in Δ. □

We are now going to prove subject reduction for global typing, giving a correspondence between the reductions of a choreography and its typing. First, we establish a relationship between the reduction labels of global types and choreographies (fidelity).

Definition 4.10 (Global Label Judgements). *The relation* $k : \alpha \vdash_{\Gamma;\,\Theta;\,\Sigma} \lambda$ *is the minimal relation satisfying the following rules.*

$$\lfloor^{\mathrm{L}}|_{\textsc{Com}}\rceil\ \frac{\big(\Theta\ \vdash\ \tau_1 : k[\mathtt{p}]\ \vee\ \mathsf{delegates}(\Sigma, \tau_1, k[\mathtt{p}])\big) \qquad \neg\mathsf{delegated}(\Sigma, k[\mathtt{q}]) \qquad \Theta\ \vdash\ \tau_2 : k[\mathtt{q}] \qquad \Gamma\ \vdash\ v : S \qquad \Gamma\ \vdash\ x : S}{k : \mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle S\rangle \quad \vdash_{\Gamma;\,\Theta;\,\Sigma} \quad \tau_1[\mathtt{p}].v\ \texttt{->}\ \tau_2[\mathtt{q}].x : k}$$

$$\lfloor^{\mathrm{L}}|_{\textsc{Sel}}\rceil\ \frac{\big(\Theta\ \vdash\ \tau_1 : k[\mathtt{p}]\ \vee\ \mathsf{delegates}(\Sigma, \tau_1, k[\mathtt{p}])\big) \qquad \neg\mathsf{delegated}(\Sigma, k[\mathtt{q}]) \qquad \Theta\ \vdash\ \tau_2 : k[\mathtt{q}]j \in I}{k : \mathtt{p}\ \texttt{->}\ \mathtt{q} : l \quad \vdash_{\Gamma;\,\Theta;\,\Sigma} \quad \tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}] : k[l]}$$

$$\lfloor^{\mathrm{L}}|_{\textsc{Del}}\rceil\ \frac{\big(\Theta\ \vdash\ \tau_1 : k[\mathtt{p}]\ \vee\ \mathsf{delegates}(\Sigma, \tau_1, k[\mathtt{p}])\big) \qquad \Theta\ \vdash\ \tau_2 : k[\mathtt{q}] \qquad \neg\mathsf{delegated}(\Sigma, k[\mathtt{q}]) \qquad \Theta\ \vdash\ \tau_1 : k'[\mathtt{p}'] \qquad \neg\mathsf{delegates}(\Sigma, \tau_1, k'[\mathtt{p}'])}{k : \mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle T\rangle \quad \vdash_{\Gamma;\,\Theta;\,\Sigma} \quad \tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}] : k\langle\!\langle k'[\mathtt{p}']\rangle\!\rangle}$$

$$\lfloor^{\mathrm{L}}|_{\textsc{Res}}\rceil\ \frac{k : \alpha \quad \vdash_{\Gamma;\,\Theta;\,\Sigma} \quad \lambda}{k : \alpha \quad \vdash_{\Gamma;\,\Theta;\,\Sigma} \quad (\boldsymbol{\nu}\tilde{r})\ \lambda}$$

$$\lfloor^{\mathrm{L}}|_{\textsc{If}}\rceil\ \frac{}{\mathsf{if}@\tau \quad \vdash_{\Gamma;\,\Theta;\,\Sigma} \quad \mathsf{if}}$$

In the sequel, we say that $\Delta \xrightarrow{k:\alpha} \Delta'$ whenever $k : G$ is in Δ such that $G \xrightarrow{\alpha} G'$ and Δ' is the result of substituting $k : G$ in Δ with $k : G'$.

We can now state subject reduction.

Theorem 4.11 (Subject Reduction). *Let* $\Gamma;\ \Theta\ \vdash_{\Sigma}\ C\ \triangleright\ \Delta$ *and* $C \xrightarrow{\lambda} C'$. *Then,* $\Gamma;\ \Theta'\ \vdash_{\Sigma'}\ C'\ \triangleright\ \Delta'$ *for some* Θ', Σ' *and* Δ' *such that*

(i) *if* $\lambda \in \{\mathsf{if}@\tau, \textit{(start)}\}$ *then* $\Theta = \Theta'$, $\Sigma = \Sigma'$ *and* $\Delta = \Delta'$;

(ii) *if* $\lambda = \tau_3[\mathtt{p}_3]$ `->` $\tau_4[\mathtt{p}_4] : k''\langle\langle k'[\mathtt{p}']\rangle\rangle$ *and* $C \xrightarrow{\lambda} C'$ *is by* $\lfloor^{\mathrm{C}}|_{\textsc{Del}}\rceil$ *then* $\Theta' = \Theta[\tau_4 \mapsto k'[\mathtt{p}']]$, $\Sigma' = \Sigma$ *and* $\Delta \xrightarrow{k:\alpha} \Delta'$ *such that* $k : \alpha \vdash_{\Gamma;\, \Theta;\, \Sigma} \lambda$;

(iii) *if* $\lambda = \tau_3[\mathtt{p}_3]$ `->` $\tau_4[\mathtt{p}_4] : k''\langle\langle k'[\mathtt{p}']\rangle\rangle$ *and* $C \xrightarrow{\lambda} C'$ *is by* $\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil$ *then* $\Theta' = \Theta$, $\Sigma' = \mathsf{add}(\Sigma, \lambda)$ *and* $\Delta \xrightarrow{k:\alpha} \Delta'$ *such that* $k : \alpha \vdash_{\Gamma;\, \Theta;\, \Sigma} \lambda$;

(iv) *otherwise,* $\Theta = \Theta'$, $\Sigma = \Sigma'$ *and* $\Delta \xrightarrow{k:\alpha} \Delta'$ *such that* $k : \alpha \vdash_{\Gamma;\, \Theta;\, \Sigma} \lambda$.

Proof. We proceed by induction on the relation $\rightarrow \subseteq (C, \lambda, C)$. Let us assume $\Gamma;\ \Theta \ \vdash_{\Sigma}\ C \ \triangleright\ \Delta$. Then,

– **Case** $\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil$. By such rule, we know that

$$\boxed{C_1 \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{r})\, C_1' \quad \Rightarrow \quad \eta; C_1 \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{r})\, \eta; C_1'}$$

such that the sender in η is a free name in λ, the receiver in η is not a free name in λ, η is not a *(start)*, and $\tilde{r} \notin \mathsf{fn}(\eta)$.

In this case, we wish to show that:

$$\boxed{\Gamma; \Theta \ \vdash_{\mathsf{add}(\Sigma,\lambda)}\ (\boldsymbol{\nu}\tilde{r})\, \eta; C_1' \ \triangleright\ \Delta'} \tag{3}$$

where $\Delta \xrightarrow{k:\alpha} \Delta'$ such that $k : \alpha \vdash_{\Gamma;\, \Theta;\, \Sigma} \lambda$.
We proceed by cases on η.

- $\boxed{\eta = \tau_1[\mathtt{p}].e$ `->` $\tau_2[\mathtt{q}].x : k}$

  Clearly, since η is a communication, we are forced to type it by applying $\lfloor^{\mathrm{GT}}|_{\textsc{Com}}\rceil$. Hence:

$$\boxed{\begin{array}{l} 1.\ \ \Gamma; \Theta \ \vdash_{\Sigma}\ C_1 \ \triangleright\ \Delta_1, k : G \\ 2.\ \ \Gamma \vdash e@\tau_1 : S \ \wedge\ \Gamma \vdash x@\tau_2 : S \\ 3.\ \ \Theta \vdash \tau_1 : k[\mathtt{p}] \ \vee\ \mathsf{delegates}(\Sigma, \tau_1, k[\mathtt{p}]) \\ 4.\ \ \Theta \vdash \tau_2 : k[\mathtt{q}] \\ 5.\ \ \neg\mathsf{delegated}(\Sigma, k[\mathtt{q}]) \end{array}} \tag{4}$$

  Now, by *induction hypothesis*, since $C_1 \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{r})\, C_1'$, we have that

$$\boxed{\Gamma;\ \Theta_1' \ \vdash_{\Sigma_1'}\ (\boldsymbol{\nu}\tilde{r})\, C_1' \ \triangleright\ \Delta_1'}$$

  According to the induction hypothesis, we have four cases, depending on the value of λ and on the semantic rule we have applied:

(i) if $\lambda \in \{\mathsf{if}@\tau, \textit{(start)}\}$ then, by induction hypothesis, $\Theta'_1 = \Theta$, $\Sigma'_1 = \Sigma$ and $\Delta_1, k : G = \Delta'_1$. Hence, we can apply $\lfloor^{\text{GT}}|_{\text{COM}}\rceil$ (after applying the restriction rules for removing $(\boldsymbol{\nu}\tilde{r})$ knowing k is not in $\tilde{r}$), prefix G with $\mathtt{p} \to \mathtt{q} : \langle U \rangle$ and conclude by applying the rules for restriction (exploiting $\tilde{r} \notin \mathsf{fn}(\eta)$ in the side condition of $\lfloor^{\text{C}}|_{\text{ASYNC}}\rceil$) $\Gamma; \Theta \vdash_\Sigma (\boldsymbol{\nu}\tilde{r})\, \eta; C'_1 \triangleright \Delta$.

(ii) if $\lambda = \tau_3[\mathtt{p}_3] \to \tau_4[\mathtt{p}_4] : k''\langle\!\langle k'[\mathtt{p}'] \rangle\!\rangle$ and we have applied $\lfloor^{\text{C}}|_{\text{DEL}}\rceil$, we notice that $\tilde{r}$ is empty. Moreover, also by $\lfloor^{\text{GT}}|_{\text{DEL}}\rceil$,

$$
\begin{array}{|l|}
\hline
1.\ C_1 = \lambda; C_2 \\
2.\ k'' : \alpha \vdash_{\Gamma;\, \Theta;\, \Sigma} \lambda \text{ where } \alpha = \mathtt{p}_3 \to \mathtt{p}_4 : \langle G''@\mathtt{p}' \rangle \\
3.\ \Theta \vdash \tau_3 : k'[\mathtt{p}'] \\
4.\ \neg\mathsf{delegates}(\Sigma, \tau_3, k'[\mathtt{p}']) \\
5.\ \Theta \vdash \tau_3 : k''[\mathtt{p}_3] \ \vee\ \mathsf{delegates}(\Sigma, \tau_3, k''[\mathtt{p}_3]) \\
6.\ \Theta \vdash \tau_4 : k''[\mathtt{p}_4] \\
7.\ \neg\mathsf{delegated}(\Sigma, k''[\mathtt{p}_4]) \\
\hline
\end{array}
\tag{5}
$$

Now, we distinguish subcases depending on k, k' and k'':

- $k' = k''$. This case is not allowed by rule $\lfloor^{\text{GT}}|_{\text{DEL}}\rceil$ since it is not possible to delegate a channel over itself.

- $k \neq k''$ and $k = k'$. By having applied $\lfloor^{\text{GT}}|_{\text{DEL}}\rceil$ we also observe:

$$\Gamma;\ \Theta[\tau_4 \mapsto k[\mathtt{p}']] \vdash_{\mathsf{rem}(\Sigma,\lambda)} C_2 \triangleright \Delta_0, k : G, k'' : G''$$

  Since we wish to prove (3), we need to show that we can apply $\lfloor^{\text{GT}}|_{\text{COM}}\rceil$, hence, we need to prove the following points (for $S = U$):

  1. $\Gamma; \Theta \vdash_{\mathsf{add}(\Sigma,\lambda)} \eta; C_2 \triangleright \Delta_0, k : \mathtt{p} \to \mathtt{q} : \langle U \rangle; G, k'' : G''$
  2. $\Gamma \vdash e@\tau_1 : S \ \wedge\ \Gamma \vdash x@\tau_2 : S$
  3. $\Theta \vdash \tau_1 : k[\mathtt{p}] \ \vee\ \mathsf{delegates}(\mathsf{add}(\Sigma, \lambda), \tau_1, k[\mathtt{p}])$
  4. $\Theta \vdash \tau_2 : k[\mathtt{q}]$
  5. $\neg\mathsf{delegated}(\mathsf{add}(\Sigma, \lambda), k[\mathtt{q}])$

  Now, points 2. and 4. follow directly from points 2. and 4. in (4) above. As for point 5., using point 5. in (4), we need to show that $\mathtt{p}' \neq \mathtt{q}$. This follows from point 3. in (5) since $\mathtt{p} = \mathtt{q}$ would imply $\tau_3 = \tau_2$ violating the side condition of $\lfloor^{\text{C}}|_{\text{ASYNC}}\rceil$. In point 3, we are safe if $\Theta \vdash \tau_1 : k[\mathtt{p}]$ by point 3 in (4). Otherwise, we know, again by point 3 in (4), that $\mathsf{delegates}(\Sigma, \tau_1, k[\mathtt{p}])$. If $\mathtt{p}' \neq \mathtt{p}$, clearly, $\mathsf{delegates}(\Sigma, \tau_1, k[\mathtt{p}])$ implies $\mathsf{delegates}(\mathsf{add}(\Sigma, \lambda), \tau_1, k[\mathtt{p}])$. Otherwise, by point 3. in (5), it must be the case that $\tau_1 = \tau_3$ which

is a contradiction between point 3. in (4) and point 4. in (5). Finally point 1 follows immediately by now applying $\lfloor^{\text{GT}}|_{\text{Com}}\rceil$ exploiting Lemma 4.8. We need to show that $\Delta \xrightarrow{k'':\alpha} \Delta'$, which trivially follows by the semantics of global types.

- $k \neq k'$ and $k = k''$. By having applied $\lfloor^{\text{GT}}|_{\text{Del}}\rceil$ we also observe:

$$\Gamma;\ \Theta[\tau_4 \mapsto k'[\mathtt{p}']]\ \vdash_{\mathsf{upd}(\Sigma,\lambda)}\ C_2\ \triangleright\ \Delta_0, k : G, k' : G'$$

Since we wish to prove (3), we need to show that we can apply $\lfloor^{\text{GT}}|_{\text{Com}}\rceil$, hence, we need to prove the following points (for $S = U$):

1. $\Gamma;\Theta\ \vdash_{\Sigma[k\mapsto\mathsf{add}(\lambda,\Sigma(k))]}\ \eta; C_2\ \triangleright\ \Delta_0, k : \mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle U\rangle; G, k' : G'$
2. $\Gamma \vdash e@\tau_1 : S\ \wedge\ \Gamma \vdash x@\tau_2 : S$
3. $\Theta \vdash \tau_1 : k[\mathtt{p}]\ \vee\ \mathsf{delegates}(\Sigma[k \mapsto \mathsf{add}(\lambda, \Sigma(k))], \tau_1, k[\mathtt{p}])$
4. $\Theta \vdash \tau_2 : k[\mathtt{q}]$
5. $\neg\mathsf{delegated}(\Sigma[k \mapsto \mathsf{add}(\lambda, \Sigma(k))], k[\mathtt{q}])$

The points above can be trivially proven with a simple argument (on the line of the previous one) since we delegate a different channel, i.e., k'.
The only interesting difference is in showing the reduction of the session environment Δ. In particular, we wish to show (for $U = S$) that

$$\mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle U\rangle; \mathtt{p_3}\ \texttt{->}\ \mathtt{p_4} : \langle G''@\mathtt{p}'\rangle; G\ \xrightarrow{\alpha}\ \mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle U\rangle; G$$

We first show that $\mathtt{q} \notin \{\mathtt{p_3}, \mathtt{p_4}\}$. If $\mathtt{q} = \mathtt{p_3}$ then, observing that $\tau_2 \neq \tau_3$ by $\lfloor^{\text{C}}|_{\text{Async}}\rceil$, by point 5. in (5), it must be the case that either $\Theta \vdash \tau_3 : k[\mathtt{q}]$ or $\mathsf{delegates}(\Sigma, \tau_3, k[\mathtt{q}])$. The first is impossible since $\Theta \vdash \tau_2 : k[\mathtt{q}]$. The second is also impossible since it would contradict point 5. above. If $\mathtt{q} = \mathtt{p_4}$ then, observing that $\tau_2 \neq \tau_4$ by $\lfloor^{\text{C}}|_{\text{Async}}\rceil$, we get a contradiction between point 4 above and point 6. in (5). This concludes the case since now we can apply the semantics of global types.

- $k \neq k'$, $k \neq k''$ and $k' \neq k''$. This case follows the previous proofs although is easier since there is no overlapping of channels.

(iii) if $\lambda = \tau_3[\mathtt{p_3}]\ \texttt{->}\ \tau_4[\mathtt{p_4}] : k''\langle\!\langle k'[\mathtt{p}']\rangle\!\rangle$ and we have applied $\lfloor^{\text{C}}|_{\text{Async}}\rceil$, we notice that $\tilde{r}$ is empty. Moreover, by induction hypothesis,

$$\boxed{\begin{array}{l}1.\ \ k'' : \alpha \vdash_{\Gamma;\,\Theta;\,\Sigma} \lambda \text{ where } \alpha = \mathtt{p_3}\ \texttt{->}\ \mathtt{p_4} : \langle G''@\mathtt{p}'\rangle \\ 2.\ \ \Theta_1' = \Theta \\ 3.\ \ \Sigma_1' = \Sigma[k \mapsto \mathsf{add}(\lambda, \Sigma(k''))]\end{array}} \tag{6}$$

From point 1. in 6 and $\lfloor^{\text{L}}|_{\text{DEL}}\rceil$ we deduce:

$$\boxed{\begin{array}{l}1.\ \Theta \vdash \tau_3 : k'[\mathtt{p}'] \\ 2.\ \neg\mathsf{delegates}(\Sigma, \tau_3, k'[\mathtt{p}']) \\ 3.\ \Theta \vdash \tau_3 : k''[\mathtt{p}_3] \ \vee\ \mathsf{delegates}(\Sigma, \tau_3, k''[\mathtt{p}_3]) \\ 4.\ \Theta \vdash \tau_4 : k''[\mathtt{p}_4] \\ 5.\ \neg\mathsf{delegated}(\Sigma, k''[\mathtt{p}_4])\end{array}} \tag{7}$$

Now, we distinguish subcases depending on k, k' and k'':

- $k' = k''$. This case is not allowed by rule $\lfloor^{\text{GT}}|_{\text{DEL}}\rceil$ since it is not possible to delegate a channel over itself.

- $k \neq k''$ and $k = k'$. Since we wish to prove (3), we need to show that we can apply $\lfloor^{\text{GT}}|_{\text{COM}}\rceil$, hence, we need to prove the following points (for $S = U$):

  1. $\Gamma; \Theta \vdash_{\Sigma[k \mapsto \mathsf{add}(\lambda, \Sigma(k))]} \eta; C_1' \triangleright \Delta_0, k : \mathtt{p} \texttt{->} \mathtt{q} : \langle U \rangle; G, k'' : G''$
  2. $\Gamma \vdash e@\tau_1 : S \ \wedge\ \Gamma \vdash x@\tau_2 : S$
  3. $\Theta \vdash \tau_1 : k[\mathtt{p}] \ \vee\ \mathsf{delegates}(\Sigma[k \mapsto \mathsf{add}(\lambda, \Sigma(k))], \tau_1, k[\mathtt{p}])$
  4. $\Theta \vdash \tau_2 : k[\mathtt{q}]$
  5. $\neg\mathsf{delegated}(\Sigma[k \mapsto \mathsf{add}(\lambda, \Sigma(k))], k[\mathtt{q}])$

  Now, points 2. and 4. follow directly from points 2. and 4. in (4) above. As for point 5., using point 5. in (4), we need to show that $\mathtt{p}' \neq \mathtt{q}$. This follows from point 3. in (7) since $\mathtt{p} = \mathtt{q}$ would imply $\tau_3 = \tau_2$ violating the side condition of $\lfloor^{\text{C}}|_{\text{ASYNC}}\rceil$. In point 3, we are safe if $\Theta \vdash \tau_1 : k[\mathtt{p}]$ by point 3 in (4). Otherwise, we know, again by point 3 in (4), that $\mathsf{delegates}(\Sigma, \tau_1, k[\mathtt{p}])$. If $\mathtt{p}' \neq \mathtt{p}$, clearly, $\mathsf{delegates}(\Sigma, \tau_1, k[\mathtt{p}])$ implies $\mathsf{delegates}(\Sigma[k \mapsto \mathsf{add}(\lambda, \Sigma(k))], \tau_1, k[\mathtt{p}])$. Otherwise, by point 1. in (7), it must be the case that $\tau_1 = \tau_3$ which is a contradiction between point 3. in (4) and point 2. in (7). Finally point 1 follows immediately by induction hypothesis. We need to show that $\Delta \xrightarrow{k'':\alpha} \Delta'$ which trivially follows by the semantics of global types.

- $k \neq k'$ and $k = k''$. Since we wish to prove (3), we need to show that we can apply $\lfloor^{\text{GT}}|_{\text{COM}}\rceil$, hence, we need to prove the following points (for $S = U$):

  1. $\Gamma; \Theta \vdash_{\Sigma[k \mapsto \mathsf{add}(\lambda, \Sigma(k))]} \eta; C_2 \triangleright \Delta_0, k : \mathtt{p} \texttt{->} \mathtt{q} : \langle U \rangle; G, k' : G'$
  2. $\Gamma \vdash e@\tau_1 : S \ \wedge\ \Gamma \vdash x@\tau_2 : S$
  3. $\Theta \vdash \tau_1 : k[\mathtt{p}] \ \vee\ \mathsf{delegates}(\Sigma[k \mapsto \mathsf{add}(\lambda, \Sigma(k))], \tau_1, k[\mathtt{p}])$
  4. $\Theta \vdash \tau_2 : k[\mathtt{q}]$
  5. $\neg\mathsf{delegated}(\Sigma[k \mapsto \mathsf{add}(\lambda, \Sigma(k))], k[\mathtt{q}])$

The points above can be trivially proven with a simple argument (on the line of the previous one) since we delegate a different channel, i.e., k'.
As in ii), the only interesting difference is in showing the reduction of the session environment Δ. In particular, we wish to show (for $U = S$) that

$$\mathtt{p} \texttt{ -> } \mathtt{q} : \langle U \rangle; G \xrightarrow{\mathtt{p_3} \texttt{ -> } \mathtt{p_4}:\langle G''@\mathtt{p}' \rangle} \mathtt{p} \texttt{ -> } \mathtt{q} : \langle U \rangle; G''$$

knowing that $G \xrightarrow{\alpha} G''$. We first show that $\mathtt{q} \notin \{\mathtt{p}_3, \mathtt{p}_4\}$. If $\mathtt{q} = \mathtt{p}_3$ then, observing that $\tau_2 \neq \tau_3$ by $\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil$, by point 3. in (7), it must be the case that either $\Theta \vdash \tau_3 : k[\mathtt{q}]$ or $\mathsf{delegates}(\Sigma, \tau_3, k[\mathtt{q}])$. The first is impossible since $\Theta \vdash \tau_2 : k[\mathtt{q}]$. The second is also impossible since it would contradict point 5. above. If $\mathtt{q} = \mathtt{p}_4$ then, observing that $\tau_2 \neq \tau_4$ by $\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil$, we get a contradiction between point 4 above and point 4. in (7). This concludes the case since now we can apply the semantics of global types.

· $k \neq k'$, $k \neq k''$ and $k' \neq k''$. Similar to previous cases.

(iv) This case follows from the induction hypothesis, having $\Theta' = \Theta = \Theta'_1$ and $\Sigma' = \Sigma = \Sigma'_1$.

- $\boxed{\eta = \tau_1[\mathtt{p}] \texttt{ -> } \tau_2[\mathtt{q}] : k\langle\langle k'[\mathtt{p}'] \rangle\rangle}$

  This case is similar to the one in which η is a communication.

- $\boxed{\eta = \tau_1[\mathtt{p}] \texttt{ -> } \tau_2[\mathtt{q}] : k[l]}$

  This case is similar to the one in which η is a communication.

- $\boxed{\eta = \tau_1[\mathtt{p}_1], \ldots, \tau_n[\mathtt{p}_n] \textbf{ start } \tau_{n+1}[\mathtt{p}_{n+1}], \ldots, \tau_m[\mathtt{p}_m] : a(k)}$

  This case is not allowed by rule $\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil$.

– **Case** $\lfloor^{\mathrm{C}}|_{\textsc{Start}}\rceil$.

Let $\eta = \tau_1[\mathtt{p}_1], \ldots, \tau_n[\mathtt{p}_n] \textbf{ start } \tau_{n+1}[\mathtt{p}_{n+1}], \ldots, \tau_m[\mathtt{p}_m] : a(k)$. We know that

$$\boxed{\eta; C_1 \xrightarrow{\eta} (\boldsymbol{\nu} k, \tau_{n+1}, \ldots, \tau_m)\, C_1}$$

From $\lfloor {}^{\text{GT}}|_{\text{START}} \rceil$ we know:

1. $\Gamma;\ \Theta\ \vdash_\Sigma\ \eta; C_1\ \triangleright\ \Delta$
2. $\Gamma;\ \Theta, \tau_1 : k[\mathtt{p}_1], \ldots, \tau_m : k[\mathtt{p}_\mathtt{m}]\ \vdash_\Sigma\ C_1\ \triangleright\ \Delta, k : G'$
3. $\Gamma\ \vdash\ a\langle \mathtt{p}_1, \ldots, \mathtt{p}_n\ ||\ \mathtt{p}_{n+1}, \ldots, \mathtt{p}_m\rangle : G$
4. $\Gamma\ \vdash\ G$
5. $G \simeq_\mathcal{G} G'$
6. $\{\mathtt{p}_1, \ldots, \mathtt{p}_m\} = \mathsf{roles}(G)$
7. $\tau_{n+1}, \ldots, \tau_m \notin \Theta$

Hence, we can obtain the thesis by applying $\lfloor {}^{\text{GT}}|_{R^{\text{TRES}}} \rceil$ $m - n$ times (one for each thread name restriction) and, finally, $\lfloor {}^{\text{GT}}|_{R^{\text{CRES}}} \rceil$.

– **Case** $\lfloor {}^{\text{C}}|_{\text{COM}} \rceil$.
Let $\eta = \tau_1[\mathtt{p}].e$ `->` $\tau_2[\mathtt{q}].x : k$. We know that

$$\tau_1[\mathtt{p}].e \texttt{ -> } \tau_2[\mathtt{q}].x : k; C'' \xrightarrow{\tau_1[\mathtt{p}].v \texttt{ -> } \tau_2[\mathtt{q}].x:k} C''[v/x@\tau_2]\ =\ C' \quad (e \downarrow v)$$

By $\lfloor {}^{\text{GT}}|_{\text{COM}} \rceil$, for $\Delta = k\ :\ \mathtt{p}$ `->` $\mathtt{q} : \langle S\rangle; G, \Delta'$,

$$\Gamma;\ \Theta\ \vdash_\Sigma\ C''\ \triangleright\ k : G, \Delta'$$

with $\Gamma\ \vdash\ e@\tau_1 : S$, $\Gamma\ \vdash\ x@\tau_2 : S$, and, $\Theta\ \vdash\ \tau_1 : k[\mathtt{p}],\ \tau_2 : k[\mathtt{q}]$. Finally, by Substition Lemma (Lemma 4.6),

$$\Gamma;\ \Theta\ \vdash_\Sigma\ C'\ \triangleright\ k : G, \Delta'$$

Clearly, by $\lfloor {}^{\text{L}}|_{\text{COM}} \rceil$,

$$k : \mathtt{p} \texttt{ -> } \mathtt{q} : \langle S\rangle \vdash_{\Gamma;\,\Theta} \tau_1[\mathtt{p}].v \texttt{ -> } \tau_2[\mathtt{q}].x : k$$

since $\Delta \xrightarrow{\mathtt{p} \texttt{ -> } \mathtt{q}:\langle S\rangle} \Delta'$ (assuming $\Gamma\ \vdash\ e@\tau_1 : S$ and $e \downarrow v$ imply $\Gamma\ \vdash\ v@\tau_1 : S$).

– **Case** $\lfloor {}^{\text{C}}|_{\text{SEL}} \rceil$. Immediately, by $\lfloor {}^{\text{GT}}|_{\text{SEL}} \rceil$, for $\Delta = k\ :\ \mathtt{p}$ `->` $\mathtt{q} : \{l_i : G_i\}_{i\in I}$,

$$\Gamma;\ \Theta\ \vdash_\Sigma\ C'\ \triangleright\ k : G_j, \Delta'$$

such that $\Theta \vdash \tau_1 : k[\mathtt{p}],\ \tau_2 : k[\mathtt{q}]$ and $j \in I$. Moreover, by $\lfloor {}^{\text{L}}|_{\text{SEL}} \rceil$, we can conclude that

$$k : \mathtt{p} \texttt{ -> } \mathtt{q} : l_j \vdash_{\Gamma;\,\Theta} \tau_1[\mathtt{p}] \texttt{ -> } \tau_2[\mathtt{q}] : k[l_j]$$

since $\Delta \xrightarrow{\mathtt{p} \texttt{ -> } \mathtt{q}:l_j} \Delta'$.

– **Case** $\lfloor {}^{\text{C}}|_{\text{DEL}} \rceil$. By $\lfloor {}^{\text{GT}}|_{\text{DEL}} \rceil$, for $\Theta = \Theta'', \tau_1 : k'[p']$ and $\Delta = k\ :\ \mathtt{p}$ `->` $\mathtt{q} : \langle G''@\mathtt{p}'\rangle; G,\ k'\ :\ G',\ \Delta'$, it follows that

$$\Gamma;\ \Theta'', \tau_2 : k'[\mathtt{p}']\ \vdash_\Sigma\ C'\ \triangleright\ k\ :\ G,\ k'\ :\ G',\ \Delta'$$

such that $\Theta'' \vdash \tau_1 : k[\mathtt{p}], \tau_2 : k[\mathtt{q}]$. Moreover, by $\lfloor {}^{\text{L}}|_{\text{DEL}} \rceil$ we can conclude that

$$k : \mathtt{p} \texttt{ -> } \mathtt{q} : \langle G''@\mathtt{p}'\rangle \vdash_{\Gamma;\,\Theta;\,\Sigma} \tau_1[\mathtt{p}] \texttt{ -> } \tau_2[\mathtt{q}] : k\langle\!\langle k'[\mathtt{p}']\rangle\!\rangle$$

– **Case** $\lfloor^{\text{C}}|_{\text{IF}}\rceil$. We treat only the case for $e \downarrow \text{true}$. The other case follows by equivalent reasoning. We have that:

$$\mathsf{if}\, e@\tau\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2 \xrightarrow{\mathsf{if}@\tau} C_1$$

The thesis follows immediately by $\lfloor^{\text{GT}}|_{\text{IF}}\rceil$:

$$\Gamma;\ \Theta\ \vdash_{\Sigma}\ C_1\ \triangleright\ \Delta$$

– **Case** $\lfloor^{\text{C}}|_{\text{SWAP}}\rceil$. Trivial, by Lemma 4.9.

All other cases are standard. □

4.3 Type Inference

We exploit the close correspondence between protocol implementations (sessions) and specifications (global types) to perform *type inference* of public channels in a choreography. First, we define *subtyping* as set inclusion on branching labels, similarly to the covariant typing of rule $\lfloor^{\text{GT}}|_{\text{SEL}}\rceil$ in Table 4. Then, we modify our rules to determine the *principal type* of a choreography. In the remainder of this section, we shall consider only typing of programs, therefore consider judgements with Σ empty. Moreover, we only refer to the services of Γ and ignore normal variables and recursion variables when talking about principal typing.

Definition 4.12 (Subtyping). *The subtyping $\ll$ is the smallest relation over closed unfolded global types satisfying the following rules:*

$$\frac{I \subseteq J \qquad \forall i \in I \cap J.\, G_i \ll G'_i}{\mathtt{p}\ \mathtt{->}\ \mathtt{q} : \{l_i : G_i\}_{i\in I} \ \ll\ \mathtt{p}\ \mathtt{->}\ \mathtt{q} : \{l_i : G'_i\}_{i\in J}} \qquad \frac{\begin{array}{c} G \approx G' \\ or \\ G \simeq_{\mathcal{G}} G' \end{array} \qquad G \ll G''}{G' \ll G''}$$

$$\frac{G_1 \ll G'_1 \qquad G_2 \ll G'_2}{\mathtt{p}\ \mathtt{->}\ \mathtt{q} : \langle G_1\rangle; G_2 \ll \mathtt{p}\ \mathtt{->}\ \mathtt{q} : \langle G'_1\rangle; G'_2} \qquad \frac{\mathsf{end} \approx G}{\mathsf{end} \ll G}$$

We extend $\ll$ to set inclusion and point-wise to service and session typing as usual. Given $\ll$, we denote its least upper bound with $\triangledown$.

Above, observe that it is trivial to show that, if two types have an upper bound, then their lub exists.

Remark 4.13 (Algorithmic Subtyping). We must observe that subtyping is algorithmically checkable. Given the simplicity of global types, checking the subtyping relation is decidable since (i) swapping is decidable (one-time unfolding of recursion is enough given that global types are regular trees) and (ii) global types are regular trees hence recursive types can be standardly dealt with as shown in [13]. In particular, our swapping relation $\simeq_{\mathcal{G}}$ plays a similar rôle as the swap of outputs in [21] and deciding our subtyping is just an instance. □

Proposition 4.14 (Subsumption). *Let $\Gamma \ll \Gamma'$ and $\Delta \ll \Delta'$. Then, $\Gamma; \Theta \vdash C \triangleright \Delta$ implies $\Gamma'; \Theta \vdash C \triangleright \Delta'$.*

Proof. Immediate from $\lfloor^{\text{GT}}|_{\text{SEL}}\rceil$. □

We are now able to show that our type system has principal typing: this will follow by subsumption and minimal typing wrt $\ll$.

Proposition 4.15 (Existence of Minimal Typing). *Let $\Gamma; \Theta \vdash C \triangleright \emptyset$. Then, there exists Γ_0 such that $\Gamma_0; \Theta \vdash C \triangleright \emptyset$ and whenever $\Gamma'; \Theta \vdash C \triangleright \emptyset$, we have $\Gamma_0 \ll \Gamma'$. The service environment Γ_0 can be algorithmically calculated from C and is called the* minimum typing *of G.*

Proof. The proof is standard and is done by constructing the minimal typing system defining $\Gamma;\ \Theta \vdash_{\mathsf{min}} G \triangleright \emptyset$, whose rules are reported in Table 5. In the rules $\lfloor^{\text{MIN}}|_{\text{VAR}}\rceil$ and $\lfloor^{\text{MIN}}|_{\text{ZERO}}\rceil$ we used some *auxiliary* information: $\tilde{S} \in \mathsf{vars}$ makes sure that $\tilde{S}$ is the type for variables $\widetilde{x@\tau}$; $\Theta \in \mathsf{ownership}$ makes sure that Θ is the ownership typing of the body of procedure X; and, $k \in \mathsf{sessions}$ establishes those session channels needed for constructing the session environment of the choreography whose type is being inferred. All of them can be easily constructed through a preliminary top-down visit of the choreography syntax tree. Finally, $\mathsf{solve}(\mathbf{t}, \Delta)$ solves the equation $\mathbf{t} = G$ for all global types contained in Δ.

□

Corollary 4.16 (Principal Typing). *Minimal typing is also principal typing.*

5 The Multiparty Endpoint Calculus

Syntax. We report the syntax of the endpoint calculus [2], an extension of the π-calculus [19], that we use for generating target code from choreographies.

$P, Q, R ::= \overline{a}[\mathtt{p}_1, \ldots, \mathtt{p}_m](k); P$	*(start)*	$\mid k?\mathtt{p}((k')); P$	*(recS)*
$\mid\ a[\mathtt{p}](k); P$	*(join)*	$\mid k!\mathtt{p}\langle\langle k' \rangle\rangle; P$	*(deleg)*
$\mid\ !a[\mathtt{p}](k); P$	*(serv)*	$\mid \mathsf{if}\, e\, \mathsf{then}\, P\, \mathsf{else}\, Q$	*(cond)*
$\mid\ k?\mathtt{p}(x); P$	*(in)*	$\mid P \mid Q$	*(par)*
$\mid\ k!\mathtt{p}\langle e \rangle; P$	*(out)*	$\mid \mathbf{0}$	*(inact)*
$\mid\ k!\mathtt{p} \oplus l; P$	*(select)*	$\mid \mathsf{rec}\, X(\tilde{x}, \tilde{k}) = P' \,\mathsf{in}\, P$	*(rec)*
$\mid\ k?\mathtt{p}\,\&\,\{l_i : P_i\}_{i \in I}; P$	*(branch)*	$\mid X(\tilde{x}, \tilde{k})$	*(recVar)*

All terms are standard. Note that roles are explicit in the syntax, e.g., the input $k!\mathtt{p}\langle e \rangle; P$ also specifies the role the message must be sent to. Comparing to [2], our syntax features an extra operation, namely the replicated service $!a[\mathtt{p}](k); P$. Technically, this can be implemented by using recursion. However, having it as an explicit construct helps in giving a simpler definition of EPP (cf. § 6).

Example 5.1 (Implementation of the Two-Buyer Protocol). The endpoint implementation of the two-buyer protocol from Example **??** can be given by the

$$\lfloor^{\text{MIN}}|\textsc{Start1}\rceil\quad \frac{\Gamma' = a\langle \mathtt{p}_1,\ldots,\mathtt{p}_n \mid\mid \mathtt{p}_{n+1},\ldots,\mathtt{p}_m\rangle \;:\; G \qquad \{\mathtt{p}_1,\ldots,\mathtt{p}_m\} = \mathsf{roles}(G) \qquad a \notin \mathsf{dom}(\Gamma) \qquad \Gamma;\ \Theta, \tau_1 : k[\mathtt{p}_1],\ldots,\tau_m : k[\mathtt{p}_m] \ \vdash_{\mathsf{min}}\ C \ \triangleright\ \Delta, k : G \qquad \tau_{n+1},\ldots,\tau_m \notin \Theta}{\Gamma,\Gamma';\ \Theta \ \vdash_{\mathsf{min}}\ \tau_1[\mathtt{p}_1],\ldots,\tau_n[\mathtt{p}_n]\ \mathbf{start}\ \tau_{n+1}[\mathtt{p}_{n+1}],\ldots,\tau_m[\mathtt{p}_m] : a(k); C \ \triangleright\ \Delta}$$

$$\lfloor^{\text{MIN}}|\textsc{Start2}\rceil\quad \frac{\Gamma' = a\langle \mathtt{p}_1,\ldots,\mathtt{p}_n \mid\mid \mathtt{p}_{n+1},\ldots,\mathtt{p}_m\rangle \;:\; G \qquad \{\mathtt{p}_1,\ldots,\mathtt{p}_m\} = \mathsf{roles}(G) \qquad \Gamma,\Gamma';\ \Theta, \tau_1 : k[\mathtt{p}_1],\ldots,\tau_m : k[\mathtt{p}_m] \ \vdash_{\mathsf{min}}\ C \ \triangleright\ \Delta, k : G' \qquad \tau_{n+1},\ldots,\tau_m \notin \Theta \qquad \Gamma'' = a\langle \mathtt{p}_1,\ldots,\mathtt{p}_n \mid\mid \mathtt{p}_{n+1},\ldots,\mathtt{p}_m\rangle \;:\; G \triangledown G'}{\Gamma,\Gamma'';\ \Theta \ \vdash_{\mathsf{min}}\ \tau_1[\mathtt{p}_1],\ldots,\tau_n[\mathtt{p}_n]\ \mathbf{start}\ \tau_{n+1}[\mathtt{p}_{n+1}],\ldots,\tau_m[\mathtt{p}_m] : a(k); C \ \triangleright\ \Delta}$$

$$\lfloor^{\text{MIN}}|\textsc{Com}\rceil\quad \frac{\Gamma \ \vdash\ e@\tau : S \qquad \Theta \ \vdash\ \tau_1 : k[\mathtt{p}],\ \tau_2 : k[\mathtt{q}] \qquad \Gamma, x@\tau : S;\ \Theta \ \vdash_{\mathsf{min}}\ C \ \triangleright\ k : G, \Delta}{\Gamma;\ \Theta \ \vdash_{\mathsf{min}}\ \tau_1[\mathtt{p}].e\ \texttt{->}\ \tau_2[\mathtt{q}].x : k; C \ \triangleright\ k : \mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle S\rangle; G, \Delta}$$

$$\lfloor^{\text{MIN}}|\textsc{Sel}\rceil\quad \frac{\Theta \ \vdash\ \tau_1 : k[\mathtt{p}] \qquad \Theta \ \vdash\ \tau_2 : k[\mathtt{q}] \qquad \Gamma;\ \Theta \ \vdash_{\mathsf{min}}\ C \ \triangleright\ k : G, \Delta}{\Gamma;\ \Theta \ \vdash_{\mathsf{min}}\ \tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}] : k[l]; C \ \triangleright\ k \ :\ \mathtt{p}\ \texttt{->}\ \mathtt{q} : \{l : G\}}$$

$$\lfloor^{\text{MIN}}|\textsc{Del}\rceil\quad \frac{\Theta \vdash \tau_1 : k[\mathtt{p}],\ \tau_2 : k[\mathtt{q}] \qquad \Gamma;\ \Theta, \tau_2 : k'[\mathtt{p}'] \ \vdash_{\mathsf{min}}\ C \ \triangleright\ k : G,\ k' : G',\ \Delta}{\Gamma;\ \Theta, \tau_1 : k'[\mathtt{p}'] \ \vdash_{\mathsf{min}}\ \tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}] : k\langle\!\langle k'[\mathtt{p}']\rangle\!\rangle; C \ \triangleright\ k : \mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle G'@\mathtt{p}'\rangle; G,\ k' : G',\ \Delta}$$

$$\lfloor^{\text{MIN}}|\textsc{If}\rceil\quad \frac{\Gamma \ \vdash\ e@\tau : \mathbf{bool} \quad \Gamma;\ \Theta \ \vdash_{\mathsf{min}}\ C_1 \ \triangleright\ \Delta_1 \quad \Gamma;\ \Theta \ \vdash_{\mathsf{min}}\ C_2 \ \triangleright\ \Delta_2}{\Gamma;\ \Theta \ \vdash_{\mathsf{min}}\ \mathsf{if}\, e@\tau\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2 \ \triangleright\ \Delta_1 \triangledown \Delta_2}$$

$$\lfloor^{\text{MIN}}|\textsc{Var}\rceil\quad \frac{\tilde{S} \in \mathsf{vars} \qquad \Theta \in \mathsf{ownership} \qquad \mathbf{t}\ \text{fresh}}{X : (\widetilde{x@\tau} : \tilde{S}, \Theta, \mathbf{t});\ \Theta \ \vdash_{\mathsf{min}}\ X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) \ \triangleright\ \bigcup_{k \in \mathsf{sessions}} k : \mathbf{t}}$$

$$\lfloor^{\text{MIN}}|\textsc{Rec}\rceil\quad \frac{X \in \mathsf{dom}(\Gamma) \Rightarrow \Gamma \vdash X : (\widetilde{x@\tau} : \tilde{S}, \Theta', \mathbf{t}) \qquad X \in \mathsf{dom}(\Gamma'_{\mathsf{rec}}) \Rightarrow \Gamma'_{\mathsf{rec}} \vdash X : (\widetilde{x@\tau} : \tilde{S}, \Theta', \mathbf{t}') \qquad \Gamma;\ \Theta \ \vdash_{\mathsf{min}}\ C \ \triangleright\ \Delta \qquad \Gamma'_{\mathsf{rec}}, \Gamma'_{\mathsf{srv}}, \widetilde{x@\tau} : \tilde{S}; \Theta' \ \vdash_{\mathsf{min}}\ C' \ \triangleright\ \Delta'}{\Gamma_{\mathsf{srv}} \triangledown \Gamma'_{\mathsf{srv}},\ \Gamma_{\mathsf{var}},\ \Gamma_{\mathsf{rec}} \setminus X;\ \Theta \ \vdash_{\mathsf{min}}\ \mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C' \,\mathsf{in}\, C \ \triangleright\ \mathsf{solve}(\mathbf{t}, \Delta \triangledown \Delta'[\mathbf{t}/\mathbf{t}'])}$$

$$\lfloor^{\text{MIN}}|\textsc{Zero}\rceil\quad \frac{\Theta \in \mathsf{ownership}}{\emptyset;\ \Theta \ \vdash_{\mathsf{min}}\ \mathbf{0} \ \triangleright\ \bigcup_{k \in \mathsf{sessions}} k : \mathsf{end}}$$

Table 5. Rules for Minimal Typing of Programs

process $P_{\mathsf{buyer}_1} \mid P_{\mathsf{buyer}_2} \mid P_{\mathsf{seller}}$ where:

$$\begin{aligned}
P_{\mathsf{buyer}_1} &= \overline{a}[\mathtt{B1}, \mathtt{B2}, \mathtt{S}](k);\ k!\mathtt{S}\langle book\rangle;\ k?\mathtt{S}(y);\ k!\mathtt{B2}\langle \mathsf{contrib}(y)\rangle \\
P_{\mathsf{buyer}_2} &= a[\mathtt{B2}](k);\ k?\mathtt{S}(z_1);\ k?\mathtt{B1}(z_2);\ \mathsf{if}\ (z_1 - z_2 \leq 100) \\
&\qquad\qquad \mathsf{then}\ k!\mathtt{S} \oplus ok;\ k!\mathtt{S}\langle addr\rangle;\ k?\mathtt{S}(w) \\
&\qquad\qquad \mathsf{else}\ k!\mathtt{S} \oplus quit \\
P_{\mathsf{seller}} &= !a[\mathtt{S}](k);\ k?\mathtt{B1}(x);\ k!\mathtt{B1}\langle \mathsf{quote}(x)\rangle;\ k!\mathtt{B2}\langle \mathsf{quote}(x)\rangle; \\
&\qquad k?\mathtt{B2}\,\&\left\{\begin{array}{l} ok : k?\mathtt{B2}(x');\ k!\mathtt{B2}\langle ddate\rangle \\ quit : \mathbf{0} \end{array}\right\}
\end{aligned}$$

□

Runtime Syntax. We extend our syntax to runtime processes by adding communication queues, restriction, and local roles to channels. In the sequel, the

letter $\boldsymbol{w}$ can be a value, a label or a session channel, namely $\boldsymbol{w} ::= v \mid l \mid k[\mathtt{p}]$.

$$
\begin{array}{llll}
P, Q, R ::= \ldots & \mid k[\mathtt{q}]?\mathtt{p}(x); P & \mid k[\mathtt{q}]?\mathtt{p}((k')); P & \mid k[\mathtt{q}]?\mathtt{p} \,\&\, \{l_i : P_i\}_{i \in I}; P \\
& \mid k[\mathtt{q}]!\mathtt{p}\langle e \rangle; P & \mid k[\mathtt{q}]!\mathtt{p}\langle\langle k' \rangle\rangle; P & \mid k[\mathtt{q}]!\mathtt{p} \oplus l; P \\
& \mid (\boldsymbol{\nu} k)\ P & \mid k[\mathtt{q}]!\mathtt{p}\langle\langle k'[\mathtt{p}'] \rangle\rangle; P & \mid k : h
\end{array}
$$

$$
h ::= m \cdot h \mid \varnothing \qquad\qquad m ::= (\mathtt{p}, \mathtt{q}, \boldsymbol{w})
$$

A session queue $k : h$ is identified by the session channel k and the queue h. The latter is a FIFO queue where each message specifies the sender and receiver role and the carried message, namely a value v, a label l or a delegated channel k.

Reduction Semantics. Table 6 reports the rules defining the labelled reduction semantics for the endpoint calculus, whose labels, ranged over by μ, are defined as:

$$
\begin{array}{ll}
\mu ::= \mathtt{p}_1, \ldots, \mathtt{p}_n \ \mathbf{start}\ \mathtt{p}_{n+1}, \ldots, \mathtt{p}_m : a(k) & \textit{(start)} \\
\quad \mid\ !\mathtt{p}\ \texttt{->}\ \mathtt{q} : k\langle v \rangle & \textit{(send-com)} \\
\quad \mid\ ?\mathtt{p}\ \texttt{->}\ \mathtt{q} : k\langle v \rangle & \textit{(recv-com)} \\
\quad \mid\ !\mathtt{p}\ \texttt{->}\ \mathtt{q} : k\langle\langle k' \rangle\rangle & \textit{(send-del)} \\
\quad \mid\ ?\mathtt{p}\ \texttt{->}\ \mathtt{q} : k\langle\langle k' \rangle\rangle & \textit{(recv-del)} \\
\quad \mid\ !\mathtt{p}\ \texttt{->}\ \mathtt{q} : k[l] & \textit{(send-sel)} \\
\quad \mid\ ?\mathtt{p}\ \texttt{->}\ \mathtt{q} : k[l] & \textit{(recv-sel)} \\
\quad \mid\ \mathsf{if} & \textit{(cond)}
\end{array}
$$

Rule $\lfloor {}^{\mathrm{P}} |_{\textsc{Start}} \rceil$ initiates a multiparty session. The rule will substitute every occurrence of session channel k (which will be restricted) with $k[\mathtt{p}_j]$ where $\mathtt{p}_j$ is the role that must be played by process P_j. Additionally, the empty session queue $k : \varnothing$ is created. Rules $\lfloor {}^{\mathrm{P}} |_{\textsc{Send}} \rceil$, $\lfloor {}^{\mathrm{P}} |_{\textsc{Del}} \rceil$ and $\lfloor {}^{\mathrm{P}} |_{\textsc{Sel}} \rceil$ put a value v, a session channel k' and label l in the queue respectively. Symmetrically, $\lfloor {}^{\mathrm{P}} |_{\textsc{Recv}} \rceil$, $\lfloor {}^{\mathrm{P}} |_{\textsc{Recs}} \rceil$ extract a value and a session channel from the queue. Rule $\lfloor {}^{\mathrm{P}} |_{\textsc{Branch}} \rceil$ fetches a label l_j from the queue and then continues as process P_j. Rule $\lfloor {}^{\mathrm{P}} |_{\textsc{Struct}} \rceil$ uses structural congruence. Structural congruence $\equiv$ for P is the smallest congruence supporting α-conversion and satisfying the following standard rules:

$$
(\boldsymbol{\nu} k)\ \mathbf{0} \equiv \mathbf{0} \qquad P \mid Q \equiv Q \mid P \qquad (\boldsymbol{\nu} k)\ (\boldsymbol{\nu} k')\ P \equiv (\boldsymbol{\nu} k')\ (\boldsymbol{\nu} k)\ P
$$

$$
((\boldsymbol{\nu} k)\ P) \mid Q \equiv (\boldsymbol{\nu} k)\ (P \mid Q) \text{ if } k \notin \mathsf{fn}(Q)
$$

$$
k : h_1 \cdot (\mathtt{p}_1, \mathtt{q}_1, \boldsymbol{w}_1) \cdot (\mathtt{p}_2, \mathtt{q}_2, \boldsymbol{w}_2) \cdot h_2 \equiv k : h_1 \cdot (\mathtt{p}_2, \mathtt{q}_2, \boldsymbol{w}_2) \cdot (\mathtt{p}_1, \mathtt{q}_1, \boldsymbol{w}_1) \cdot h_2 \quad (\text{for } \mathtt{p}_1 \neq \mathtt{p}_2 \text{ or } \mathtt{q}_1 \neq \mathtt{q}_2)
$$

$$
\mathsf{rec}\, X(\tilde{x}, \tilde{k}) = P' \,\mathsf{in}\, (P \mid Q) \ \equiv\ (\mathsf{rec}\, X(\tilde{x}, \tilde{k}) = P' \,\mathsf{in}\, P) \mid Q \qquad \text{if } X \notin \mathsf{fvar}(Q)
$$

$$
\mathsf{rec}\, X(\tilde{x}, \tilde{k}) = P' \,\mathsf{in}\, \mathbf{0} \ \equiv\ \mathbf{0}
$$

$$
\mathsf{rec}\, X(\tilde{x}, \tilde{k}) = P' \,\mathsf{in}\, X(\tilde{v}, \tilde{k}') \ \equiv\ \mathsf{rec}\, X(\tilde{x}, \tilde{k}) = P \,\mathsf{in}\, P[\tilde{v}/\tilde{x}]
$$

Structural congruence allows allows to permute messages with different pairs of roles in a queue. Technically, this is equivalent to having a (full-duplex) queue per each pair of roles [2].

$$
\begin{array}{rl}
 & \overline{a}[\mathtt{p}_1,\ldots,\mathtt{p}_m](k);P \mid \prod_{j\in J} a[\mathtt{p}_j](k_j);P_j \mid R \xrightarrow{\mathtt{p}_1,\ldots,\mathtt{p}_n\ \mathbf{start}\ \mathtt{p}_{n+1},\ldots,\mathtt{p}_m:a(k)} \\
\lfloor^{\mathrm{P}}|_{\textsc{Start}}\rceil & (\boldsymbol{\nu} k)\left(P[k[\mathtt{p}_1]/k] \mid \prod_{j\in J} P_j[k[\mathtt{p}_j]/k_j] \mid \prod_{j\in J'} P_j[k[\mathtt{p}_j]/k_j] \mid k:\emptyset\right) \mid R \\
 & (\text{where } J=\{2,\ldots,n\} \wedge J'=\{n+1,\ldots,m\} \wedge R=\textstyle\prod_{j\in J'} !a[\mathtt{p}_j](k_j);P_j) \\
\lfloor^{\mathrm{P}}|_{\textsc{Send}}\rceil & k[\mathtt{p}]!\mathtt{q}\langle e\rangle;P \mid k:h \xrightarrow{\mathtt{!p\ ->\ q}:k\langle v\rangle} P \mid k:h\cdot(\mathtt{p},\mathtt{q},v) \quad (e\Downarrow v) \\
\lfloor^{\mathrm{P}}|_{\textsc{Recv}}\rceil & k[\mathtt{p}]?\mathtt{q}(x);P \mid k:(\mathtt{q},\mathtt{p},v)\cdot h \xrightarrow{\mathtt{?p\ ->\ q}:k\langle v\rangle} P[v/x] \mid k:h \\
\lfloor^{\mathrm{P}}|_{\textsc{SendDel}}\rceil & k[\mathtt{p}]!\mathtt{q}\langle\!\langle k'[\mathtt{p}']\rangle\!\rangle;P \mid k:h \xrightarrow{\mathtt{!p\ ->\ q}:k\langle\!\langle k'\rangle\!\rangle} P \mid k:h\cdot(\mathtt{p},\mathtt{q},k'[\mathtt{p}']) \quad (e\Downarrow v) \\
\lfloor^{\mathrm{P}}|_{\textsc{RecvDel}}\rceil & k[\mathtt{p}]?\mathtt{q}(\!(k'')\!);P \mid k:(\mathtt{q},\mathtt{p},k'[\mathtt{p}'])\cdot h \xrightarrow{\mathtt{?p\ ->\ q}:k\langle\!\langle k'\rangle\!\rangle} P[k'[\mathtt{p}']/k''] \mid k:h \\
\lfloor^{\mathrm{P}}|_{\textsc{Sel}}\rceil & k[\mathtt{p}]!\mathtt{q}\oplus l;P \mid k:h \xrightarrow{\mathtt{!p\ ->\ q}:k[l]} P \mid k:h\cdot(\mathtt{p},\mathtt{q},l) \\
\lfloor^{\mathrm{P}}|_{\textsc{Branch}}\rceil & k?\mathtt{p}\,\&\,\{l_i:P_i\}_{i\in I} \mid k:(\mathtt{q},\mathtt{p},l_j)\cdot h \xrightarrow{\mathtt{?p\ ->\ q}:k[l_j]} P_j \mid k:h \quad (j\in I) \\
\lfloor^{\mathrm{P}}|_{\textsc{RecCtx}}\rceil & P \xrightarrow{\mu} P' \quad\Rightarrow\quad \mathsf{rec}\,X(\tilde{x},\tilde{k})=Q\,\mathsf{in}\,P \xrightarrow{\mu} \mathsf{rec}\,X(\tilde{x},\tilde{k})=Q\,\mathsf{in}\,P' \\
\lfloor^{\mathrm{P}}|_{\textsc{If}}\rceil & \mathsf{if}\,e\,\mathsf{then}\,P_1\,\mathsf{else}\,P_2 \xrightarrow{\mathsf{if}} P_i \quad (i=1 \text{ if } e\Downarrow \mathrm{true}\,,\ i=2 \text{ otherwise}) \\
\lfloor^{\mathrm{P}}|_{\textsc{Par}}\rceil & P \xrightarrow{\mu} P' \quad\Rightarrow\quad P \mid Q \xrightarrow{\mu} P' \mid Q \\
\lfloor^{\mathrm{P}}|_{\textsc{Res}}\rceil & P \xrightarrow{\mu} P' \quad\Rightarrow\quad (\boldsymbol{\nu} k)\,P \xrightarrow{\mu} (\boldsymbol{\nu} k)\,P' \\
\lfloor^{\mathrm{P}}|_{\textsc{Struct}}\rceil & P\equiv P' \quad P'\xrightarrow{\mu} Q' \quad Q'\equiv Q \quad\Rightarrow\quad P\xrightarrow{\mu} Q
\end{array}
$$

Table 6. Labelled Semantics Rules for the Endpoint Calculus

Example 5.2 (Two-Buyer Protocol Endpoint Semantics). Applying the semantics given above, we can prove the reduction

$$P_{\mathsf{buyer}_1} \mid P_{\mathsf{buyer}_2} \mid P_{\mathsf{seller}} \xrightarrow{\mu_1} (\boldsymbol{\nu} k)\,(Q_{\mathsf{buyer}_1} \mid Q_{\mathsf{buyer}_2} \mid Q_{\mathsf{seller}}) \mid P_{\mathsf{seller}} \xrightarrow{\mu_2} \ldots \xrightarrow{\mu'} P_{\mathsf{seller}}$$

where

$$
\begin{array}{rcl}
Q_{\mathsf{buyer}_1} & = & k[\mathtt{B1}]!\mathtt{S}\langle book\rangle;\ k[\mathtt{B1}]?\mathtt{S}(y);\ k[\mathtt{B1}]!\mathtt{B2}\langle \mathsf{contrib}(y)\rangle \\
Q_{\mathsf{buyer}_2} & = & k[\mathtt{B2}]?\mathtt{S}(z_1);\ k[\mathtt{B2}]?\mathtt{B1}(z_2);\ \mathsf{if}\ (z_1-z_2\le 100) \\
 & & \qquad\qquad \mathsf{then}\ k[\mathtt{B2}]!\mathtt{S}\oplus ok;\ k[\mathtt{B2}]!\mathtt{S}\langle addr\rangle;\ k[\mathtt{B2}]?\mathtt{S}(w) \\
 & & \qquad\qquad \mathsf{else}\ \ k[\mathtt{B2}]!\mathtt{S}\oplus quit \\
Q_{\mathsf{seller}} & = & k[\mathtt{S}]?\mathtt{B1}(x);\ k[\mathtt{S}]!\mathtt{B1}\langle \mathsf{quote}(x)\rangle;\ k[\mathtt{S}]!\mathtt{B2}\langle \mathsf{quote}(x)\rangle; \\
 & & k[\mathtt{S}]?\mathtt{B2}\,\&\left\{\begin{array}{l} ok : k[\mathtt{S}]?\mathtt{B2}(x');\ k[\mathtt{S}]!\mathtt{B2}\langle ddate\rangle \\ quit : \mathbf{0}\end{array}\right\}
\end{array}
$$

and, assuming that $z_1 - z_2 > 100$, $\mu_1 = \mathtt{B1},\mathtt{B2}\ \mathbf{start}\ \mathtt{S} : a(k)$, $\mu_2 = \mathtt{!B1\ ->\ S} : k\langle book\rangle$, $\mu' = \mathtt{?B2\ ->\ S} : k[quit]$. Note that P_{seller} will never reduce to $\mathbf{0}$ since it is a replicated service. Moreover, we observe that the process is confluent, i.e., it always reduces to P_{Seller}. □

6 Endpoint Projection and its Properties

We formalise here our notion of EPP, which correctly generates endpoint code from a choreography.

6.1 Thread Projection

We first establish how we can project the behaviour of a single thread. For this purpose, we have to consider that the same thread may appear in different branches of a choreography. For example, consider the following choreography:

$$\mathsf{if}\, e@\tau_1\, \mathsf{then}\ \ \tau_1[\mathtt{p}] \texttt{->} \tau_2[\mathtt{q}] : k[l_1]\ \ \mathsf{else}\ \ \tau_1[\mathtt{p}] \texttt{->} \tau_2[\mathtt{q}] : k[l_2]$$

where τ_1 performs an internal choice and then, depending on the chosen branch, sends to τ_2 either label l_1 or label l_2. Observe that τ_2 does not know which branch has been chosen by τ_1 until it receives a label from the latter. Thus, the projection of τ_2 should be able to react to both labels. We address this issue with *merging* [9]. Whenever the code for a thread τ is scattered in different branches of a choreography, we will try to obtain an endpoint code that implements the behaviours of τ in the different branches. Hence, in our example, the projection of τ_2 should be $k[\mathtt{q}]?\mathtt{p}\,\&\,\{l_1 : \mathbf{0},\ l_2 : \mathbf{0}\}$. Formally, we define merging $\sqcup$ as:

Definition 6.1. *$\sqcup$ is a partial commutative operator on processes such that:*

1. $(\boldsymbol{k}?\mathtt{p}\,\&\,\{l_i : P_i\}_{i\in I})\ \sqcup\ (\boldsymbol{k}?\mathtt{p}\,\&\,\{l_j : Q_j\}_{j\in J})\ =$
   $\boldsymbol{k}?\mathtt{p}\,\&\,(\ \{l_i : P_i\}_{i\in I\setminus J} \cup \{l_i : Q_i\}_{i\in J\setminus I} \cup \{l_i : (P_i \sqcup Q_i)\}_{i\in I\cap J}\ )$
   where $\boldsymbol{k}$ is a channel possibly annotated with a role, i.e., $\boldsymbol{k} ::= k \mid k[\mathtt{p}]$
2. *otherwise, $P \sqcup Q$ is defined congruently up to $\equiv$.*

We can now define the projection of a single thread onto an endpoint term.

Definition 6.2 (Thread Projection). *$C\!\upharpoonright_\tau^{\tilde{k}}$ is a partial operation from choreographies to endpoint processes, inductively defined on the structure of C by the rules in Table 7.*

We comment the reported rules. Session start projects the first thread to an endpoint start, threads $\tau_2, \ldots, \tau_n$ to joins and threads $\tau_{n+1}, \ldots, \tau_m$ to services. In the case of interaction on a channel k, the projection of τ_1 is an output on k. Symmetrically, τ_2 is projected onto an input. Selection is similar. The projection of $\mathsf{if}\, e@\tau\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2$ is a local conditional if we are projecting τ, otherwise it is the merging of the projections of the two branches, reflecting that the other threads should behave accordingly to both branches. In $C\!\upharpoonright_\tau^{\tilde{k}}$, the vector $\tilde{k}$ tracks bound channel names since they need to be projected as static terms. For example, in selection, thread τ_1 is projected to $k!\mathtt{q}\langle e\rangle; (C\!\upharpoonright_\tau^{\tilde{k}})$ if it is a runtime term and to $k[\mathtt{p}]!\mathtt{q}\langle e\rangle; (C\!\upharpoonright_\tau^{\tilde{k}})$ otherwise. In the sequel, $C\!\upharpoonright_\tau$ is a shortcut for $C\!\upharpoonright_\tau^{\emptyset}$.

Example 6.3. The thread projections of the choreography in Example **??** (Two-Buyer Protocol Choreography) are the processes reported in Example 5.1. □

$$(C)\lfloor^{\tilde{k}}_{\tau_i} = \begin{cases} \overline{a}[\mathtt{p}_1,\ldots,\mathtt{p}_m](k);(C'\lfloor^{\tilde{k},k}_{\tau_i}) & \text{if } i=1 \\ a[\mathtt{p}_i](k);(C'\lfloor^{\tilde{k},k}_{\tau_i}) & \text{if } 2\le i\le n \\ !a[\mathtt{p}_i](k);(C'\lfloor^{\tilde{k},k}_{\tau_i}) & \text{if } n+1\le i\le m \\ C'\lfloor^{\tilde{k}}_{\tau_i} & \text{otherwise} \end{cases}$$

where $C = \tau_1[\mathtt{p}_1],\ldots,\tau_n[\mathtt{p}_n]\ \ \mathbf{start}\ \tau_{n+1}[\mathtt{p}_{n+1}],\ldots,\tau_m[\mathtt{p}_m] : a(k);C'$

$$(\tau_1[\mathtt{p}].e \texttt{ -> } \tau_2[\mathtt{q}].x : k;C)\lfloor^{\tilde{k}}_{\tau} = \begin{cases} k!\mathtt{q}\langle e\rangle;(C\lfloor^{\tilde{k}}_{\tau}) & \text{if } \tau=\tau_1 \text{ and } k\in\tilde{k} \\ k[\mathtt{p}]!\mathtt{q}\langle e\rangle;(C\lfloor^{\tilde{k}}_{\tau}) & \text{if } \tau=\tau_1 \text{ and } k\notin\tilde{k} \\ k?\mathtt{p}(x);(C\lfloor^{\tilde{k}}_{\tau}) & \text{if } \tau=\tau_2 \text{ and } k\in\tilde{k} \\ k[\mathtt{q}]?\mathtt{p}(x);(C\lfloor^{\tilde{k}}_{\tau}) & \text{if } \tau=\tau_2 \text{ and } k\notin\tilde{k} \\ C\lfloor^{\tilde{k}}_{\tau} & \text{otherwise} \end{cases}$$

$$(\tau_1[\mathtt{p}] \texttt{ -> } \tau_2[\mathtt{q}] : k[l];C)\lfloor^{\tilde{k}}_{\tau} = \begin{cases} k!\mathtt{q}\oplus l;(C\lfloor^{\tilde{k}}_{\tau}) & \text{if } \tau=\tau_1 \text{ and } k\in\tilde{k} \\ k[\mathtt{p}]!\mathtt{q}\oplus l;(C\lfloor^{\tilde{k}}_{\tau}) & \text{if } \tau=\tau_1 \text{ and } k\notin\tilde{k} \\ k?\mathtt{p}\,\&\,\{l:(C\lfloor^{\tilde{k}}_{\tau})\} & \text{if } \tau=\tau_2 \text{ and } k\in\tilde{k} \\ k[\mathtt{q}]?\mathtt{p}\,\&\,\{l:(C\lfloor^{\tilde{k}}_{\tau})\} & \text{if } \tau=\tau_2 \text{ and } k\notin\tilde{k} \\ C\lfloor^{\tilde{k}}_{\tau} & \text{otherwise} \end{cases}$$

$$(\tau_1[\mathtt{p}] \texttt{ -> } \tau_2[\mathtt{q}] : k\langle\!\langle k'[\mathtt{p}']\rangle\!\rangle;C)\lfloor^{\tilde{k}}_{\tau} = \begin{cases} k!\mathtt{q}\langle\!\langle k'\rangle\!\rangle;(C\lfloor^{\tilde{k}}_{\tau}) & \text{if } \tau=\tau_1 \text{ and } k\in\tilde{k} \text{ and } k'\in\tilde{k} \\ k[\mathtt{p}]!\mathtt{q}\langle\!\langle k'\rangle\!\rangle;(C\lfloor^{\tilde{k}}_{\tau}) & \text{if } \tau=\tau_1 \text{ and } k\notin\tilde{k} \text{ and } k'\in\tilde{k} \\ k!\mathtt{q}\langle\!\langle k'[\mathtt{p}']\rangle\!\rangle;(C\lfloor^{\tilde{k}}_{\tau}) & \text{if } \tau=\tau_1 \text{ and } k\in\tilde{k} \text{ and } k'\notin\tilde{k} \\ k[\mathtt{p}]!\mathtt{q}\langle\!\langle k'[\mathtt{p}']\rangle\!\rangle;(C\lfloor^{\tilde{k}}_{\tau}) & \text{if } \tau=\tau_1 \text{ and } k\notin\tilde{k} \text{ and } k'\notin\tilde{k} \\ k?\mathtt{p}((k'));(C\lfloor^{\tilde{k},k'}_{\tau}) & \text{if } \tau=\tau_2 \text{ and } k\in\tilde{k} \\ k[\mathtt{q}]?\mathtt{p}((k'));(C\lfloor^{\tilde{k},k'}_{\tau}) & \text{if } \tau=\tau_2 \text{ and } k\notin\tilde{k} \\ C\lfloor^{\tilde{k}}_{\tau} & \text{otherwise} \end{cases}$$

$$(\mathsf{if}\, e@\tau\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2)\lfloor^{\tilde{k}}_{\tau'} = \begin{cases} \mathsf{if}\, e\, \mathsf{then}\, (C_1\lfloor^{\tilde{k}}_{\tau'})\, \mathsf{else}\, (C_2\lfloor^{\tilde{k}}_{\tau'}) & \text{if } \tau=\tau' \\ (C_1\lfloor^{\tilde{k}}_{\tau'}) \sqcup (C_2\lfloor^{\tilde{k}}_{\tau'}) & \text{otherwise} \end{cases}$$

$$((\boldsymbol{\nu} k)\, C)\lfloor^{\tilde{k}}_{\tau} = (\boldsymbol{\nu} k)\, (C\lfloor^{\tilde{k}}_{\tau}) \qquad ((\boldsymbol{\nu} \tau')\, C)\lfloor^{\tilde{k}}_{\tau} = C\lfloor^{\tilde{k}}_{\tau}$$

$$(\mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C'\, \mathsf{in}\, C)\lfloor^{\tilde{k'}}_{\tau} = \mathsf{rec}\, X(\tilde{x}', \tilde{k}'') = (C' \lfloor_{\tau}\, \tilde{k}')\, \mathsf{in}\, (C\lfloor^{\tilde{k'}}_{\tau})$$

$$X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau})\lfloor^{\tilde{k'}}_{\tau} = X(\tilde{x}, \tilde{k})$$

$$\mathbf{0}\lfloor^{\tilde{k}}_{\tau} = \mathbf{0}$$

Table 7. Thread Projection

6.2 Linearity

We now introduce a condition for avoiding races between threads wishing to do a start with the same role and on the same session. For example, the choreography

$$\tau_1[\mathsf{p}], \tau_2[\mathsf{q}] \ \mathbf{start} \ : a(k);\ \tau_3[\mathsf{p}], \tau_4[\mathsf{q}] \ \mathbf{start} \ : a(k) \tag{8}$$

features four threads willing to engage into two different sessions started through public channel a. If we run the projections of the four threads in parallel, we have a race between τ_1 and τ_3 and another between τ_2 and τ_4 for synchronising on a. This may result in τ_1 starting a session with τ_4 and τ_2 starting a session with τ_3, violating the specification given by the choreography.

In the sequel, an *interaction node*, denoted by n, is either τ_1 `->` τ_2 or $\tau_1, \ldots, \tau_n$ **start** $\tau_{n+1}, \ldots, \tau_m : a$. Interaction nodes are abstractions of nodes in a choreography syntax tree: we associate $\tau_1, \ldots, \tau_n$ **start** $\tau_{n+1}, \ldots, \tau_m : a$ to a start node and τ_1 `->` τ_2 to any other node involving an in-session interaction. We write $\mathsf{n}_1 \prec \mathsf{n}_2 \in C$ whenever n_1 occurs before n_2 in the choreography C. We use interaction nodes for establishing dependencies between thread actions.

Definition 6.4 (Dependency). *We write* $\mathsf{n}_1 \prec_\tau \mathsf{n}_2 \in C$ *if* $\mathsf{n}_1 \prec \mathsf{n}_2 \in C$ *and either*

1. $\mathsf{n}_1 = \tau_1, \ldots, \tau_n$ **start** $\tau_{n+1}, \ldots, \tau_m : a$ *and* $\mathsf{n}_2 = \tau$ `->` τ' *if* $\tau = \tau_i$, $1 \leq i \leq m$*; or,*
2. $\mathsf{n}_1 = \tau_1, \ldots, \tau_n$ **start** $\tau_{n+1}, \ldots, \tau_m : a$ *and* $\mathsf{n}_2 = \tau'_1, \ldots, \tau'_n$ **start** $\tau'_{n+1}, \ldots, \tau'_m : a$ *where* $\tau = \tau_i$ *for* $1 \leq i \leq m$ *and* $\tau \in \mathsf{fn}(\mathsf{n}_2)$*; or,*
3. $\mathsf{n}_1 = \tau'$ `->` τ *and* $\tau \in \mathsf{fn}(\mathsf{n}_2)$.

Intuitively, $\mathsf{n}_1 \prec_\tau \mathsf{n}_2$ means that the projection of τ for (the original syntax node of) n_2 will not be enabled before that for n_1. We can finally exploit this dependency for ensuring that there are no races on a public channel a.

Definition 6.5 (Linearity). *Whenever* $\mathsf{n}_i = \tau^i_1, \ldots, \tau^i_n$ **start** $\tau^i_{n+1}, \ldots, \tau^i_m : a$ *($i = 1, 2$, $n \geq 2$) are in C and are not in different branches of a (cond), we say C is* linear *if either* $\forall j \in \{1, \ldots, n\}. \exists j' \in \{1, \ldots, m\}.\ \mathsf{n}_1 \prec_{\tau^1_{j'}} \ldots \prec_{\tau^2_j} \mathsf{n}_2$ *or* $\forall j \in \{1, \ldots, n\}. \exists j' \in \{1, \ldots, m\}.\ \mathsf{n}_2 \prec_{\tau^2_{j'}} \ldots \prec_{\tau^1_j} \mathsf{n}_1$.

Linearity makes sure that, given two start nodes n_1 and n_2 on the same a such that $\mathsf{n}_1 \prec \mathsf{n}_2$, each running thread in n_2 depends on some thread in n_1. We only care about races between two or more running threads ($n \geq 2$); races between service threads can be avoided by merging their behaviour. Observe that linearity is decidable since a choreography is linear whenever its one-time unfolding of recursions is linear [15].

Example 6.6. In the choreography (8) we cannot build any dependency unless, e.g., $\tau_1 = \tau_3$ and $\tau_2 = \tau_4$. However, the following choreography is linear as there are dependencies between τ_1 and τ_3 and between τ_2 and τ_4.

$$\begin{array}{l} \tau_1[\mathsf{p}], \tau_2[\mathsf{q}] \ \mathbf{start} \ : a(k); \quad \tau_1 \texttt{ -> } \tau_3 : k'; \\ \tau_2 \texttt{ -> } \tau'_2 : k'; \quad \tau'_2 \texttt{ -> } \tau_4 : k'; \quad \tau_3[\mathsf{p}], \tau_4[\mathsf{q}] \ \mathbf{start} \ : a(k) \end{array} \qquad \square$$

Linearity is preserved by swapping.

Lemma 6.7 (Linearity preservation under swapping). *Let C be linear. Then, for all C', $C \simeq_{\mathcal{C}} C'$ implies that C' is linear.*

6.3 Endpoint Projection

Since different service threads may be started on the same public channel and play the same role, we can use the operator $\sqcup$ for merging their behaviours into a single service process. We identify such threads with the following function.

Definition 6.8 (Service Thread Grouping). *The operator $\lfloor C \rfloor^a_{\mathtt{p}}$ is inductively defined as follows:*

$$\lfloor C \rfloor^a_{\mathtt{p}_i} = \begin{cases} \{\tau_i\} \cup \lfloor C' \rfloor^a_{\mathtt{p}_i} & \text{if } n+1 \leq i \leq m \\ \lfloor C' \rfloor^a_{\mathtt{p}_i} & \text{otherwise} \end{cases}$$
$$\text{where } C = \tau_1[\mathtt{p}_1], \ldots, \tau_n[\mathtt{p}_n] \ \mathbf{start} \ \tau_{n+1}[\mathtt{p}_{n+1}], \ldots, \tau_m[\mathtt{p}_m] : a(k); C'$$

$$\lfloor \mathsf{if}\, e@\tau \,\mathsf{then}\, C_1 \,\mathsf{else}\, C_2 \rfloor^a_{\mathtt{p}} = \lfloor C_1 \rfloor^a_{\mathtt{p}} \cup \lfloor C_2 \rfloor^a_{\mathtt{p}} \qquad \lfloor X \rfloor^a_{\mathtt{p}} = \lfloor \mathbf{0} \rfloor^a_{\mathtt{p}} = \emptyset$$
$$\lfloor C \rfloor^a_{\mathtt{p}} = \lfloor C' \rfloor^a_{\mathtt{p}} \text{ if } C \text{ is an in-session interaction with continuation } C'$$

$\lfloor C \rfloor^a_p$ recursively visit the structure of a choreography for finding all the service threads started on channel a that play role $\mathtt{p}$.

We can finally give the complete definition of EPP.

Definition 6.9 (Endpoint Projection). *Let $C \equiv (\boldsymbol{\nu} \tilde{\tau} \tilde{k})\ C_f$ where C_f is restriction-free, i.e., there are no subterms $(\boldsymbol{\nu} r)\ C'$ in C_f. Then, the EPP of C is:*

$$C \downharpoonright = (\boldsymbol{\nu} \tilde{k}) \left(\prod_{\tau \,\in\, \mathsf{fn}(C_f)} C_f \downharpoonright_\tau \ \mid \prod_{k \,\in\, \mathsf{fn}(C_f)} k : \varnothing \right) \mid \prod_{a,\mathtt{p}} \left(\bigsqcup_{\tau \in \lfloor C_f \rfloor^a_{\mathtt{p}}} C_f \downharpoonright_\tau \right)$$

Intuitively, the EPP of a choreography is the parallel composition of (i) the projections of all the active threads; (ii) the services obtained by merging the projections of all the threads in C that can be started on the same public channel playing the same role; and (iii) the message queues for all the active sessions.

Example 6.10. Consider the following choreography where two clients, c_1 and c_2, open two separate sessions for printing some texts. c_1 prints a single text, whereas c_2 prints five.

$$\begin{array}{l} \mathsf{c}_1[\mathtt{C}] \ \mathbf{start} \ \mathsf{s}_1[\mathtt{S}] : a(k); \\ \mathsf{c}_1[\mathtt{C}] \ \texttt{->} \ \mathsf{s}_1[\mathtt{S}] : k[one]; \\ \mathsf{c}_1[\mathtt{C}].text_1 \ \texttt{->} \ \mathsf{s}_1[\mathtt{S}].x_1 : k \end{array} \quad \mid \quad \begin{array}{l} \mathsf{c}_2[\mathtt{C}] \ \mathbf{start} \ \mathsf{s}_2[\mathtt{S}] : a(k); \\ \mathsf{c}_2[\mathtt{C}] \ \texttt{->} \ \mathsf{s}_2[\mathtt{S}] : k[multi]; \\ \mathsf{rec}\ X(texts_2@\mathsf{c}_2, n@\mathsf{c}_2, k, \mathsf{c}_2, \mathsf{s}_2) = \\ \quad \mathsf{c}_2[\mathtt{C}].\mathsf{get}(texts_2, n) \ \texttt{->} \ \mathsf{s}_2[\mathtt{S}].x_2 : k; \\ \quad \mathsf{if}\ (n>1)@\mathsf{c}_2 \ \mathsf{then}\ \mathsf{c}_2[\mathtt{C}] \ \texttt{->} \ \mathsf{s}_2[\mathtt{S}] : k[next]; \\ \qquad\qquad\qquad X(texts_2@\mathsf{c}_2, (n-1)@\mathsf{c}_2, k, \mathsf{c}_2, \mathsf{s}_2) \\ \quad\ \ \mathsf{else}\ \mathsf{c}_2[\mathtt{C}] \ \texttt{->} \ \mathsf{s}_2[\mathtt{S}] : k[done] \\ \mathsf{in}\ X(texts_2@\mathsf{c}_2, 5@\mathsf{c}_2, k, \mathsf{c}_2, \mathsf{s}_2) \end{array}$$

Since s_1 and s_2 are grouped under $\lfloor C \rfloor^a_{\mathtt{S}}$, the EPP of C will merge their behaviour into a single process. I.e., $C \downharpoonright = C \downharpoonright_{\mathsf{c}_1} \mid C \downharpoonright_{\mathsf{c}_2} \mid P_{\mathsf{s}}$ where:

$$P_{\mathsf{s}} = !a[\mathtt{S}](k);\ k?\mathtt{S} \,\&\, \left\{ \begin{array}{c} one : k?\mathtt{S}(x) \\ multi : \mathsf{rec}\, X = k?\mathtt{S}(x); k?\mathtt{S} \,\&\, \{next : X, done : \mathbf{0}\} \,\mathsf{in}\, X \end{array} \right\}$$

□

Observe that EPP is not influenced by the swap relation $\simeq_{\mathcal{C}}$ for choreographies. Intuitively, the role played by swapping in choreographies will be played by the usual semantics for parallel composition in the generated endpoint system.

Lemma 6.11 (EPP invariance under swapping). *Let $C \simeq_{\mathcal{C}} C'$. Then, $C \upharpoonright = C' \upharpoonright$.*

Proof (sketch). The main part of the proof is to show that thread projection is invariant under the rules that define the swapping relation for choreographies (Table 3). $\lfloor {}^{\text{Sw}}|_{\text{Inter}} \rceil$ is a trivial case. For $\lfloor {}^{\text{Sw}}|_{\text{Cond-Inter}} \rceil$, we have to check that the projections of the threads in the swapped interaction η do not change. We simply need to observe that, in the definition of thread projection for if-then-else choices, for each τ in η we have that $\eta \upharpoonright_\tau = \eta \upharpoonright_\tau \sqcup \eta \upharpoonright_\tau$. The last case is $\lfloor {}^{\text{Sw}}|_{\text{Cond-Cond}} \rceil$. Here, we simply need to check the definition of merging for observing that the projection of the choice and its swapped version is the same for τ, τ', and for all other threads. □

EPP respects some basic expected properties about substitution.

Lemma 6.12 (EPP substitution lemma).

$$(C[v/x@\tau]) \upharpoonright_\tau = (C \upharpoonright_\tau)[v/x]$$

Proof. Immediate from the definition of thread projection. □

Lemma 6.13 (EPP substitution locality). *Let τ' be a free thread name in C', i.e. $\tau' \in \mathsf{fn}(C')$, and*

$$C \upharpoonright = (\boldsymbol{\nu} \tilde{k}) \left(\prod_{\tau \in \mathsf{fn}(C')} C' \upharpoonright_\tau \mid \prod_{k \in \mathsf{fn}(C')} k : \varnothing \right) \mid \prod_{a,\mathtt{p}} \left(\bigsqcup_{\tau \in \lfloor C' \rfloor^a_{\mathtt{p}}} C' \upharpoonright_\tau \right)$$

Then

$$(C[v/x@\tau']) \upharpoonright = (\boldsymbol{\nu} \tilde{k}) \left((C[v/x@\tau']) \upharpoonright_{\tau'} \mid \prod_{\tau \in \mathsf{fn}(C) \setminus \{\tau'\}} C \upharpoonright_\tau \mid \prod_{k \in \mathsf{fn}(C)} k : \varnothing \right) \mid \prod_{a,\mathtt{p}} \left(\bigsqcup_{\tau \in \lfloor C \rfloor^a_{\mathtt{p}}} C \upharpoonright_\tau \right)$$

Proof. Immediate from the definition of EPP and Lemma 6.12. □

Lemma 6.13 says that a substitution under a free thread τ' in a choreography C affects its projection $C \upharpoonright$ only in the thread projection of τ'.

6.4 Properties

Before showing the main theorem for our EPP we have to introduce some auxiliary concepts for establishing the relationship between a choreography, its projection, and their respective reduction labels.

In order to correlate the actions of a choreography to those of its projection, we consider only *strict reductions* of terms, denoted by $\rightsquigarrow$. Strict reductions are

reductions where $(\boldsymbol{\nu})$-restricted names that are active, i.e. not under a prefix, are never renamed. Observe that we do not lose generality, since for every reduction there is always a corresponding strict reduction. Referring only to strict reductions allows us to observe the actions performed by restricted names. Formally this is obtained by changing the rules for handling restriction in the semantics of the global and endpoint calculi. The updated rules are the ones concerning restriction:

Definition 6.14 (New Restriction Rules).

$$\lfloor {}^{\mathrm{C}}|_{\mathrm{RES}} \rceil\ C \overset{\lambda}{\rightsquigarrow} C' \quad \Rightarrow \quad (\boldsymbol{\nu} r)\, C \overset{\lambda}{\rightsquigarrow} (\boldsymbol{\nu} r)\, C'$$

$$\lfloor {}^{\mathrm{P}}|_{\mathrm{RES}} \rceil\ P \overset{\mu}{\rightsquigarrow} P' \quad \Rightarrow \quad (\boldsymbol{\nu} k)\, P \overset{\mu}{\rightsquigarrow} (\boldsymbol{\nu} k)\, P'$$

In the following, we denote a finite strict reduction chain with $C \overset{\tilde{\lambda}}{\rightsquigarrow}{}^{*} C'$, i.e. $\tilde{\lambda} = \lambda_1, \ldots, \lambda_n$ for $C \overset{\lambda_1}{\rightsquigarrow} \ldots \overset{\lambda_n}{\rightsquigarrow} C'$. We adopt the same notation for strict reduction chains in the endpoint calculus.

We introduce two results about usage of channels from the point of view of endpoint systems, which will be useful later. The first result simply formulates the meaning of linearity of public channels at the endpoint level. The second result, instead, deals with the linearity of session channels, guaranteeing that if a thread is capable of putting a message on the queue for a session channel k with some role $\mathtt{p}$, then no other thread will do the same until such action is done.

Lemma 6.15 (Linearity Lemma). *Let C be linear, and $C \lfloor \overset{\tilde{\mu}}{\rightsquigarrow}{}^{*} P$ for some P. If P has a redex at a, then $a[\mathtt{p}]$ is enabled at most once in P for any $\mathtt{p}$.*

Proof. By induction on the length n of the reduction chain. □

Lemma 6.16 (In-session Linearity). *Let C be well-typed and*

$$C \lfloor \equiv (\boldsymbol{\nu} \tilde{k}) \left(P \mid Q \mid k : h \right)$$

where $Q = k[\mathtt{p}]!\mathtt{q}; Q'$ and $(\mathtt{p}, \mathtt{r}, \boldsymbol{w}) \notin h$ for any $\mathtt{r}$ and $\boldsymbol{w}$. Then, $C \lfloor \overset{\tilde{\mu}}{\rightsquigarrow}{}^{*} P'$ for some P' such that:

$$P' = (\boldsymbol{\nu} \tilde{k'}) \left(P'' \mid Q \mid k : h' \right)$$

implies that $(\mathtt{p}, \mathtt{r}', \boldsymbol{w}') \notin h'$ for any $\mathtt{r}'$ and $\boldsymbol{w}'$.

Proof (sketch). The only way for putting a message in the queue for k with role $\mathtt{p}$ is by executing a corresponding output. From the well-typedness of C, we can derive that P does not have any output (or input) on the free channel k with role $\mathtt{p}$ (this check is performed by the thread environment Θ). Therefore, the only possibility for P to eventually perform the output is to reduce to a process that comes to contain k as a free name through a substitution. This can happen only if P receives k through a delegation. But this is impossible, since the only process that can delegate k is Q, and Q does not perform any action in the n reductions. □

We introduce pruning, a relation that allows us to ignore unused endpoint services and branches.

Definition 6.17 (Pruning). *Let $Q \equiv Q_0 \mid R$, where $R = \prod_{i \in I} !a_i[\mathtt{p}_i](k_i); R_i$. If furthermore we have that:*

(i) $a_i \notin \mathsf{fn}(Q_0)$ *for every* $i \in I$*;*
(ii) $P \sqcup Q_0 = Q_0$*;*
(iii) $Q_0 \xrightarrow{\mu} Q_0'$ *implies that there exists* P' *such that* $P \xrightarrow{\mu} P'$ *and* $P' \prec Q_0'$*.*

then we write $P \prec Q$ *and say that* P prunes Q*.*

Intuitively, pruning establishes that P is equal to Q apart from the following differences:

- some unused services are removed *(i)*;
- some input branches may be removed *(ii)*, but they were not going to be used *(iii)*.

From the definition of merging it follows directly that pruning is followed also in the other direction.

Proposition 6.18 (Pruning preservation). *Let $P \prec Q$. Then, $P \xrightarrow{\mu} P'$ implies that there exists Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \prec Q'$.*

We can observe that pruning is transitive. We can also show that it is preserved by all the thread projections that are not involved in the reduction of a choreography, as formalised by the following Lemma.

Lemma 6.19 (Passive threads pruning invariance). *Let C be restriction-free. Then $C \overset{\lambda}{\rightsquigarrow} C'$ implies that for every $\tau \in \mathsf{fn}(C) \setminus \mathsf{fn}(\lambda)$*

$$C' \upharpoonright_\tau \quad \prec \quad C \upharpoonright_\tau$$

Proof (sketch). The proof is by cases on the global calculus semantics. From the definition of EPP we have that the only difference between the two projections is in if-then-else choices, on the receiver side. The thesis follows by definition of pruning. Observe that we can safely ignore potential swaps in C performed in its reduction, thanks to Lemma 6.11. □

Lemma 6.20 (Pruning Lemma). *Let C be well-typed and $C \upharpoonright = ((\boldsymbol{\nu} \tilde{k})\, P) \mid R$ where*

$$R = \prod_{a,\mathtt{p}} \Big(\bigsqcup_{\tau \in \lfloor C \rfloor^a_{\mathtt{p}}} C \upharpoonright_\tau \Big)$$

Let now R' be the process obtained from R by (i) adding some new services on new public channels and (ii) merging the services in R with some other services:

$$R' = \prod_{a,\mathtt{p}} \Big(\big(\bigsqcup_{\tau \in \lfloor C \rfloor^a_{\mathtt{p}}} C \upharpoonright_\tau \big) \sqcup R^a_{\mathtt{p}} \Big) \mid \prod_{i \in I} !a_i[\mathtt{p}_i](k_i); P_i$$

where for all $i \in I$, $a_i \notin \mathsf{fn}(C)$. Then, R' defined implies: $C \upharpoonright \prec ((\boldsymbol{\nu} \tilde{k})\, P) \mid R'$

Proof (sketch). From the well-typedness of C, we can derive that P does not use any of the new public channels a_i. The same reasoning can be used for proving that all the new branches introduced in R' are never going to be used. □

We can now formally establish how to relate the observable labels of a choreography with those of an endpoint system.

Definition 6.21 (EPP trace judgements). *The relation $\tilde{\lambda} \vdash \tilde{\mu}$ is the minimal relation satisfying the rules reported in Table 8.*

$$\lfloor^{\text{J}}|_{\textsc{Start}}\rceil \frac{\tilde{\lambda} \vdash \tilde{\mu}}{\tau_1[\mathtt{p}_1], \ldots, \tau_n[\mathtt{p}_n] \textbf{ start } \tau_{n+1}[\mathtt{p}_{n+1}], \ldots, \tau_m[\mathtt{p}_m] : a(k), \tilde{\lambda} \vdash \mathtt{p}_1, \ldots, \mathtt{p}_n \textbf{ start } \mathtt{p}_{n+1}, \ldots, \mathtt{p}_m : a(k), \tilde{\mu}}$$

$$\lfloor^{\text{J}}|_{\textsc{Com}}\rceil \frac{?\mathtt{p} \texttt{->} \mathtt{q} : k \notin \tilde{\mu_1} \qquad \tilde{\lambda} \vdash \tilde{\mu_1}, \tilde{\mu_2}}{\tau_1[\mathtt{p}].v \texttt{->} \tau_2[\mathtt{q}].x : k, \tilde{\lambda} \vdash !\mathtt{p} \texttt{->} \mathtt{q} : k\langle v \rangle, \tilde{\mu_1}, ?\mathtt{p} \texttt{->} \mathtt{q} : k\langle v \rangle, \tilde{\mu_2}}$$

$$\lfloor^{\text{J}}|_{\textsc{Sel}}\rceil \frac{?\mathtt{p} \texttt{->} \mathtt{q} : k \notin \tilde{\mu_1} \qquad \tilde{\lambda} \vdash \tilde{\mu_1}, \tilde{\mu_2}}{\tau_1[\mathtt{p}] \texttt{->} \tau_2[\mathtt{q}] : k[l], \tilde{\lambda} \vdash !\mathtt{p} \texttt{->} \mathtt{q} : k[l], \tilde{\mu_1}, ?\mathtt{p} \texttt{->} \mathtt{q} : k[l], \tilde{\mu_2}}$$

$$\lfloor^{\text{J}}|_{\textsc{Del}}\rceil \frac{?\mathtt{p} \texttt{->} \mathtt{q} : k \notin \tilde{\mu_1} \qquad \tilde{\lambda} \vdash \tilde{\mu_1}, \tilde{\mu_2}}{\tau_1[\mathtt{p}] \texttt{->} \tau_2[\mathtt{q}] : k\langle\!\langle k'[\mathtt{r}] \rangle\!\rangle, \tilde{\lambda} \vdash !\mathtt{p} \texttt{->} \mathtt{q} : k\langle\!\langle k' \rangle\!\rangle, \tilde{\mu_1}, ?\mathtt{p} \texttt{->} \mathtt{q} : k\langle\!\langle k' \rangle\!\rangle, \tilde{\mu_2}}$$

$$\lfloor^{\text{J}}|_{\textsc{Cond}}\rceil \frac{\tilde{\lambda} \vdash \tilde{\mu}}{\mathsf{if}@\tau, \tilde{\lambda} \vdash \mathsf{if}, \tilde{\mu}}$$

$$\lfloor^{\text{J}}|_{\textsc{Empty}}\rceil \frac{}{\emptyset \vdash \emptyset}$$

Table 8. Trace judgements for endpoint processes

We briefly comment the rules for trace judgements. Rule $\lfloor^{\text{J}}|_{\textsc{Start}}\rceil$ checks that a start performed in a choreography corresponds to an equivalent start in the endpoint system. Rule $\lfloor^{\text{J}}|_{\textsc{Com}}\rceil$ checks that a communication in a choreography trace is performed at the enpoint level by check that both the corresponding input and output are made. The rule allows for some interleaving of unrelated message-receiving labels, given the asynchrony of endpoint systems. Rules $\lfloor^{\text{J}}|_{\textsc{Sel}}\rceil$ and $\lfloor^{\text{J}}|_{\textsc{Del}}\rceil$ behave in the same way, for selections and delegations respectively. Rules $\lfloor^{\text{J}}|_{\textsc{Internal}}\rceil$ and $\lfloor^{\text{J}}|_{\textsc{Cond}}\rceil$ handle internal choices ϵ and empty traces.

We can now present our main theorem.

Theorem 6.22 (EPP). *Let $C \equiv (\boldsymbol{\nu}\tilde{\tau}\tilde{k})\, C_f$, where C_f is restriction-free, be linear and well-typed. Then,*

1. *(Completeness) $C \overset{\lambda}{\rightsquigarrow} C'$ implies there exists P such that (i) $C' \lfloor \prec P$ and either (ii) $C \lfloor \overset{\mu}{\rightsquigarrow} P$ where $\lambda \vdash \mu$ or (iii) $C \lfloor \overset{\mu_1}{\rightsquigarrow} \overset{\mu_2}{\rightsquigarrow} P$ where $\lambda \vdash \mu_1, \mu_2$.*
2. *(Soundness) $C \lfloor \overset{\tilde{\mu}}{\rightsquigarrow}{}^* P$ implies there exist P' and C' such that (i) $P \overset{\tilde{\mu}'}{\rightsquigarrow}{}^* P'$; (ii) $C \overset{\tilde{\lambda}}{\rightsquigarrow}{}^* C'$ and $\tilde{\lambda} \vdash \tilde{\mu}, \tilde{\mu}'$; and (iii) $C' \lfloor \prec P'$.*

We split the proof in two parts, for the sake of presentation. We first prove the completeness part and then the soundness part of the Theorem.

Proof (Compleness). The proof is by induction on the relation $\rightsquigarrow \subseteq (C, \lambda, C)$. We report the most interesting cases.

– **Case** $\lfloor {}^{\text{C}} \rvert_{\textsc{Com}} \rceil$. We know that

$$C = \tau_1[\mathtt{p}].e \texttt{ -> } \tau_2[\mathtt{q}].x : k; C'' \quad \overset{\tau_1[\mathtt{p}].v \texttt{ -> } \tau_2[\mathtt{q}].x:k}{\rightsquigarrow} \quad C''[v/x@\tau_2] = C' \quad (e \downarrow v) \tag{9}$$

From the definition of EPP we have:

$$C \lfloor \equiv (\boldsymbol{\nu}k, \tilde{k}) \left(k[\mathtt{p}]!\mathtt{q}\langle e\rangle; (C''_f \lfloor_{\tau_1}) \quad \mid \quad k[\mathtt{q}]?\mathtt{p}(x); (C''_f \lfloor_{\tau_2}) \quad \mid \quad k : \varnothing \right.$$
$$\left. \mid \prod_{\tau \in \mathsf{fn}(C_f) \setminus \{\tau_1, \tau_2\}} C_f \lfloor_\tau \mid \prod_{k \in \tilde{k}} k' : \varnothing \right) \mid \prod_{a, \mathtt{p}} \Big(\bigsqcup_{\tau \in \lfloor C_f \rfloor^a_{\mathtt{p}}} C_f \lfloor_\tau \Big) \tag{10}$$

where $C''_f \equiv C''$ and $\tilde{k}$ are the free session channel names in C_f excluding k. Using $\lfloor {}^{\text{P}} \rvert_{\textsc{Send}} \rceil$ we can derive:

$$C \lfloor \overset{!\mathtt{p} \texttt{ -> } \mathtt{q}:k\langle v\rangle}{\rightsquigarrow} (\boldsymbol{\nu}k, \tilde{k}) \left((C''_f \lfloor_{\tau_1}) \mid k[\mathtt{q}]?\mathtt{p}(x); (C''_f \lfloor_{\tau_2}) \mid k : (\mathtt{p}, \mathtt{q}, v) \right.$$
$$\left. \mid \prod_{\tau \in \mathsf{fn}(C_f) \setminus \{\tau_1, \tau_2\}} C_f \lfloor_\tau \mid \prod_{k' \in \tilde{k}} k' : \varnothing \right) \mid \prod_{a, \mathtt{p}} \Big(\bigsqcup_{\tau \in \lfloor C_f \rfloor^a_{\mathtt{p}}} C_f \lfloor_\tau \Big) = Q \tag{11}$$

By $\lfloor {}^{\text{P}} \rvert_{\textsc{Recv}} \rceil$:

$$Q \quad \overset{?\mathtt{p} \texttt{ -> } \mathtt{q}:k\langle v\rangle}{\rightsquigarrow} \quad (\boldsymbol{\nu}k, \tilde{k}) \left((C''_f \lfloor_{\tau_1}) \mid (C''_f \lfloor_{\tau_2})[v/x] \mid k : \varnothing \right.$$
$$\left. \mid \prod_{\tau \in \mathsf{fn}(C_f) \setminus \{\tau_1, \tau_2\}} C_f \lfloor_\tau \mid \prod_{k' \in \tilde{k}} k' : \varnothing \right) \mid \prod_{a, \mathtt{p}} \Big(\bigsqcup_{\tau \in \lfloor C_f \rfloor^a_{\mathtt{p}}} C_f \lfloor_\tau \Big) = P \tag{12}$$

which proves (iii).
Let us see now the projection of C'. From (9) and Lemma 6.13 we have:

$$C'\lfloor \equiv (\boldsymbol{\nu}\tilde{k'})\left((C''_f\lfloor_{\tau_1}) \mid (C''_f[v/x@\tau_2]\lfloor_{\tau_2}) \mid \prod_{k'\in\tilde{k'}} k' : \varnothing \mid \prod_{\tau\in \mathsf{fn}(C''_f)\setminus\{\tau_1,\tau_2\}} C''_f\lfloor_{\tau}\right) \mid \prod_{a,\mathtt{p}}\Big(\bigsqcup_{\tau\in\lfloor C''_f\rfloor^a_{\mathtt{p}}} C''_f\lfloor_{\tau}\Big) \tag{13}$$

where $\tilde{k'}$ are the free session channels in C''_f. From Lemma 6.12 we obtain:

$$C'\lfloor \equiv (\boldsymbol{\nu}\tilde{k'})\left((C''_f\lfloor_{\tau_1}) \mid (C''_f\lfloor_{\tau_2})[v/x] \mid \prod_{k'\in\tilde{k'}} k' : \varnothing \mid \prod_{\tau\in \mathsf{fn}(C''_f)\setminus\{\tau_1,\tau_2\}} C''_f\lfloor_{\tau}\right) \mid \prod_{a,\mathtt{p}}\Big(\bigsqcup_{\tau\in\lfloor C''_f\rfloor^a_{\mathtt{p}}} C''_f\lfloor_{\tau}\Big) \tag{14}$$

We observe now that $\mathsf{fn}(C') \subseteq \mathsf{fn}(C)$, because the reduction from C to C' has not added any free name. From this observation and Lemma 6.19 we get:

$$C'\lfloor \prec (\boldsymbol{\nu}\tilde{k'})\left((C''_f\lfloor_{\tau_1}) \mid (C''_f\lfloor_{\tau_2})[v/x] \mid \prod_{k'\in\tilde{k'}} k' : \varnothing \mid \prod_{\tau\in \mathsf{fn}(C_f)\setminus\{\tau_1,\tau_2\}} C_f\lfloor_{\tau}\right) \mid \prod_{a,\mathtt{p}}\Big(\bigsqcup_{\tau\in\lfloor C''_f\rfloor^a_{\mathtt{p}}} C''_f\lfloor_{\tau}\Big) \tag{15}$$

Using the same observation again ($\mathsf{fn}(C') \subseteq \mathsf{fn}(C)$), it follows from the definition of EPP (and merging) that:

$$C'\lfloor \prec (\boldsymbol{\nu}\tilde{k'})\left((C''_f\lfloor_{\tau_1}) \mid (C''_f\lfloor_{\tau_2})[v/x] \mid \prod_{k'\in\tilde{k'}} k' : \varnothing \mid \prod_{\tau\in \mathsf{fn}(C_f)\setminus\{\tau_1,\tau_2\}} C_f\lfloor_{\tau}\right) \mid \prod_{a,\mathtt{p}}\Big(\bigsqcup_{\tau\in\lfloor C_f\rfloor^a_{\mathtt{p}}} C_f\lfloor_{\tau}\Big) \prec P \tag{16}$$

which proves (i).

– **Case** $\lfloor^{\mathrm{C}}|_{\textsc{Start}}\rceil$. We know that

$$C = \tau_1[\mathtt{p}_1],\ldots,\tau_n[\mathtt{p}_n]\ \mathbf{start}\ \tau_{n+1}[\mathtt{p}_{n+1}],\ldots,\tau_m[\mathtt{p}_m] : a(k); C'' \quad \tau_1[\mathtt{p}_1],\ldots,\tau_n[\mathtt{p}_n]\ \mathbf{start}\ \tau_{n+1}[\mathtt{p}_{n+1}],\ldots,\tau_m[\mathtt{p}_m]:a(k) \rightsquigarrow \quad (\boldsymbol{\nu}k,\tau_{n+1},\ldots,\tau_m)\, C'' = C' \tag{17}$$

Let now $[\tau] = \{\tau_1, \ldots, \tau_m\}$. From the definition of EPP we have

$$C\lfloor \equiv (\boldsymbol{\nu} \tilde{k'}) \left(\overline{a}[\mathtt{p}_1, \ldots, \mathtt{p}_m](k); (C''_f \lfloor_{\tau_1}) \mid \prod_{i \in \{2,\ldots,n\}} a[\mathtt{p}_i](k); (C''_f \lfloor_{\tau_i}) \mid \prod_{k' \in \tilde{k'}} k' : \varnothing \right.$$
$$\left. \mid \prod_{\tau \in \mathsf{fn}(C_f) \setminus [\tau]} C_f \lfloor_\tau \right) \mid \prod_{i \in \{n+1,\ldots,m\}} !a[\mathtt{p}_i](k); (\bigsqcup_{\tau \in \lfloor C_f \rfloor^a_{\mathtt{p}_i}} C''_f \lfloor_\tau)$$
$$\mid \prod_{a' \neq a, \mathtt{p}} (\bigsqcup_{\tau \in \lfloor C_f \rfloor^{a'}_{\mathtt{p}}} C_f \lfloor_\tau) = Q \tag{18}$$

where $\tilde{k}$ are the free session channel names in C_f. We base our partitioning of the services in the EPP on the fact that, thanks to the well-typedness of C, we are sure that the roles $\{\mathtt{p}_{n+1}, \ldots, \mathtt{p}_m\}$ are the only ones that are played by the service threads in C.

Let $\mu = \mathtt{p}_1, \ldots, \mathtt{p}_n$ **start** $\mathtt{p}_{n+1}, \ldots, \mathtt{p}_m : a(k)$. Applying $\lfloor^{\mathrm{P}} \rceil_{\textsc{start}}$ we can derive

$$Q \overset{\mu}{\rightsquigarrow} (\boldsymbol{\nu} \tilde{k'}, k) \left(\prod_{i \in \{1,\ldots,n\}} (C''_f \lfloor_{\tau_i}) \mid \prod_{k' \in \tilde{k'}, k} k' : \varnothing \mid \prod_{\tau \in \mathsf{fn}(C_f) \setminus [\tau]} C_f \lfloor_\tau \right)$$
$$\mid \prod_{i \in \{n+1,\ldots,m\}} !a[\mathtt{p}_i](k); (\bigsqcup_{\tau \in \lfloor C_f \rfloor^a_{\mathtt{p}_i}} C''_f \lfloor_\tau) \mid \prod_{a' \neq a, \mathtt{p}} (\bigsqcup_{\tau \in \lfloor C_f \rfloor^{a'}_{\mathtt{p}}} C_f \lfloor_\tau) = P \tag{19}$$

which proves (ii). (i) follows from similar reasoning as in the case for $\lfloor^{\mathrm{C}} \rceil_{\textsc{com}}$.

– **Case** $\lfloor^{\mathrm{C}} \rceil_{\textsc{swap}}$. Follows directly from the inductive hypothesis and Lemma 6.11.

□

Proof (Soundness). The proof is by induction on the structure of C_f. We report the most interesting cases.

– **Case** $C_f = \tau_1[\mathtt{p}].e \texttt{ -> } \tau_2[\mathtt{q}].x : k; C_b$. The inductive hypothesis is:

$$\text{for all } \tilde{\mu_1} \quad (\boldsymbol{\nu} \tilde{\tau} \tilde{k}) \, C_b \lfloor \overset{\tilde{\mu_1}}{\rightsquigarrow}{}^* \; P_1 \;\Rightarrow P_1 \overset{\tilde{\mu'_1}}{\rightsquigarrow}{}^* \; P'_1$$
$$(\boldsymbol{\nu} \tilde{\tau} \tilde{k}) \, C_b \overset{\tilde{\lambda'}}{\rightsquigarrow}{}^* \; C'_1 \tag{20}$$
$$\tilde{\lambda'} \vdash \tilde{\mu_1}, \tilde{\mu'_1}$$
$$C'_1 \lfloor \prec P'_1$$

From the definition of EPP we have

$$C\lfloor \equiv (\boldsymbol{\nu}\tilde{k})\left(\prod_{\tau\in \mathsf{fn}(C_b)\setminus\{\tau_1,\tau_2\}} C_b\lfloor_\tau \mid k[\mathtt{p}]!\mathtt{q}\langle e\rangle;(C_b\lfloor_{\tau_1}) \mid k[\mathtt{q}]?\mathtt{p}(x);(C_b\lfloor_{\tau_2}) \mid \prod_{k'\in \mathsf{fn}(C_b)\setminus\{k\}} k':\varnothing \mid k:\varnothing\right) \mid \prod_{a,\mathtt{p}}\left(\bigsqcup_{\tau\in\lfloor C_b\rfloor^a_{\mathtt{p}}} C_b\lfloor_\tau\right) \tag{21}$$

The projection of τ_2 is stuck on an input and the projection of τ_1 can perform an output. Lemma 6.11 allows us not to consider potential swaps in C when looking at its EPP, since it is always the same. From the inductive hypothesis and the semantics for the global calculus we get that

$$C \overset{\tilde{\lambda}}{\rightsquigarrow}{}^* \ (\boldsymbol{\nu}\tilde{\tau}\tilde{k})\, C_1'[v/x@\tau_2] \ = \ C' \qquad (e\downarrow v) \tag{22}$$

where $\tilde{\lambda}=\tilde{\lambda_1'},\tau_1[\mathtt{p}].v\ \texttt{->}\ \tau_2[\mathtt{q}].x:k,\tilde{\lambda_2'}$ and $\tilde{\lambda'}=\tilde{\lambda_1'},\tilde{\lambda_2'}$.
Now we have two cases, depending on the actions performed in the first reduction chain from $C\lfloor$ to P. If the projection of τ_1 does not perform its output in these reductions, then we apply the inductive hypothesis choosing $\tilde{\mu}=\tilde{\mu_1}$ and we obtain

$$C\lfloor \overset{\tilde{\mu_1}}{\rightsquigarrow}{}^* (\boldsymbol{\nu}\tilde{k''})\left(P_b \mid k[\mathtt{p}]!\mathtt{q}\langle e\rangle;(C_b\lfloor_{\tau_1}) \mid k[\mathtt{q}]?\mathtt{p}(x);(C_b\lfloor_{\tau_2}) \mid k:h\right) \mid \prod_{a,\mathtt{p}}\left(\bigsqcup_{\tau\in\lfloor C_b\rfloor^a_{\mathtt{p}}} C_b\lfloor_\tau\right) \ = \ P \tag{23}$$

where P_b may contain some new processes spawned by starting some services and some session queues. From Lemma 6.16 we also know that $(\mathtt{p},\mathtt{q},\boldsymbol{w})\notin h$ for any $\boldsymbol{w}$, since all the other threads can not play the same roles of τ_1 and τ_2 in k. Now we need to reduce P to P' as per the theorem statement. Applying rules $\lfloor^{\mathrm{P}}\vert_{\textsc{Send}}\rceil$ and $\lfloor^{\mathrm{P}}\vert_{\textsc{Recv}}\rceil$ to (23) we obtain

$$P \ \overset{!\mathtt{p}\ \texttt{->}\ \mathtt{q}:k\langle v\rangle}{\rightsquigarrow} \ \overset{?\mathtt{p}\ \texttt{->}\ \mathtt{q}:k\langle v\rangle}{\rightsquigarrow} \ (\boldsymbol{\nu}\tilde{k''})\left(P_b \mid (C_b\lfloor_{\tau_1}) \mid (C_b\lfloor_{\tau_2}\,[v/x]) \mid k:h\right) \mid \prod_{a,\mathtt{p}}\left(\bigsqcup_{\tau\in\lfloor C_b\rfloor^a_{\mathtt{p}}} C_b\lfloor_\tau\right) \ = \ Q \tag{24}$$

where $e\downarrow v$. From well-typedness we know that $C_b\lfloor_{\tau_2}$ is the only process using name x. Therefore $Q=P_1[v/x]$. The thesis follows now from the inductive hypothesis, having $\tilde{\mu'}=!\mathtt{p}\ \texttt{->}\ \mathtt{q}:k\langle v\rangle,?\mathtt{p}\ \texttt{->}\ \mathtt{q}:k\langle v\rangle,\tilde{\mu_1'}$.

Let us now consider the case in which the projection of τ_1 performs its output in the first reduction chain of $C\lfloor$. By applying similar reasoning to the previous case, we can split the reduction chain as follows. First we have:

$$C\lfloor \overset{\tilde{\mu_{1_1}}}{\rightsquigarrow}{}^* \ (\boldsymbol{\nu}\tilde{k''}) \left(P_b \mid k[\mathtt{p}]!\mathtt{q}\langle e\rangle; (C_b\lfloor_{\tau_1}) \mid k[\mathtt{q}]?\mathtt{p}(x); (C_b\lfloor_{\tau_2}) \mid k : h \right) \mid \prod_{a,\mathtt{p}} \left(\bigsqcup_{\tau \in \lfloor C_b \rfloor^a_{\mathtt{p}}} C_b\lfloor_{\tau} \right) \ = \ P_{1_1} \tag{25}$$

where $(\mathtt{p}, \mathtt{q}, \boldsymbol{w}) \notin h$ for any $\boldsymbol{w}$. When τ_1 performs the output we obtain:

$$P_{1_1} \ \overset{!\mathtt{p}\ \texttt{->}\ \mathtt{q}:k\langle v\rangle}{\rightsquigarrow} \ (\boldsymbol{\nu}\tilde{k''}) \left(P_b \mid (C_b\lfloor_{\tau_1}) \mid k[\mathtt{q}]?\mathtt{p}(x); (C_b\lfloor_{\tau_2}) \mid k : h \cdot (\mathtt{p}, \mathtt{q}, v) \right) \mid \prod_{a,\mathtt{p}} \left(\bigsqcup_{\tau \in \lfloor C_b \rfloor^a_{\mathtt{p}}} C_b\lfloor_{\tau} \right) \ = P_{1_2} \tag{26}$$

where $e \downarrow v$. Now there are two sub-cases: the projection of τ_2 may perform its input in the reduction chain from $C\lfloor$ to P or not. Let us see the first sub-case. We have:

$$P_{1_2} \ \overset{?\mathtt{p}\ \texttt{->}\ \mathtt{q}:k\langle v\rangle}{\rightsquigarrow} \ (\boldsymbol{\nu}\tilde{k''}) \left(P_b \mid (C_b\lfloor_{\tau_1}) \mid (C_b\lfloor_{\tau_2}\,[v/x]) \mid k : h \right) \mid \prod_{a,\mathtt{p}} \left(\bigsqcup_{\tau \in \lfloor C_b \rfloor^a_{\mathtt{p}}} C_b\lfloor_{\tau} \right) \ = P_{1_3} \tag{27}$$

Therefore we can split the reduction chain from $C\lfloor$ to P as:

$$C\lfloor \overset{\tilde{\mu_{1_1}}}{\rightsquigarrow}{}^* \ P_{1_1} \ \overset{!\mathtt{p}\ \texttt{->}\ \mathtt{q}:k\langle v\rangle}{\rightsquigarrow} \ P_{1_2} \ \overset{?\mathtt{p}\ \texttt{->}\ \mathtt{q}:k\langle v\rangle}{\rightsquigarrow} \ P_{1_3} \ \overset{\tilde{\mu_{1_2}}}{\rightsquigarrow}{}^* \ P \ = P_1[v/x] \tag{28}$$

The thesis now follows by inductive hypothesis.
Let us see the other sub-case, in which τ_2 does not perform the input in the reduction chain from $C\lfloor$ to P. By similar reasoning we can split the reduction chain as:

$$C\lfloor \overset{\tilde{\mu_{1_1}}}{\rightsquigarrow}{}^* \ P_{1_1} \ \overset{!\mathtt{p}\ \texttt{->}\ \mathtt{q}:k\langle v\rangle}{\rightsquigarrow} \ P_{1_2} \ \overset{\tilde{\mu_{1_2}}}{\rightsquigarrow}{}^* \ P \tag{29}$$

Now we can choose to perform the input in the reduction chain from P to P':

$$P \ \overset{?\mathtt{p}\ \texttt{->}\ \mathtt{q}:k\langle v\rangle}{\rightsquigarrow} \ P'' \ \overset{\tilde{\mu'}}{\rightsquigarrow}{}^* \ P' \tag{30}$$

The thesis now follows from similar reasoning to the above.

– **Case** $C_f = \tau_1[\mathtt{p}_1], \ldots, \tau_n[\mathtt{p}_n]$ **start** $\tau_{n+1}[\mathtt{p}_{n+1}], \ldots, \tau_m[\mathtt{p}_m] : a(k); C_b$. The inductive hypothesis is:

$$\begin{array}{lll} \text{for all } \tilde{\mu_1} \quad (\boldsymbol{\nu}\tilde{\tau}\tilde{k})\, C_b \lfloor \overset{\tilde{\mu_1}}{\rightsquigarrow}{}^* \; P_1 \;\Rightarrow & P_1 \overset{\tilde{\mu_1'}}{\rightsquigarrow}{}^* \; P_1' & \\ & (\boldsymbol{\nu}\tilde{\tau}\tilde{k})\, C_b \overset{\tilde{\lambda_1}}{\rightsquigarrow}{}^* \; C_1' & (31) \\ & \tilde{\lambda_1} \;\vdash\; \tilde{\mu_1}, \tilde{\mu_1'} & \\ & C_1' \lfloor \prec P_1' & \end{array}$$

The thesis can now be proven with similar reasoning as for the case of communications. The only important difference is that we have to check that the projection of

$$\tau_1[\mathtt{p}_1], \ldots, \tau_n[\mathtt{p}_n] \textbf{ start } \tau_{n+1}[\mathtt{p}_{n+1}], \ldots, \tau_m[\mathtt{p}_m] : a(k)$$

should not introduce any race on a. This is guaranteed by the Linearity Lemma (Lemma 6.15).

□

The completeness part of the EPP Theorem states that an EPP is capable of reproducing (up to pruning) all the reductions of its originating choreography; on the other hand, the soundness result says that the EPP always eventually reduces (up to pruning) to the projection of a (possibly reached after multiple reductions) reductum of its originating choreography. Both results check that the observables of a choreography and its projection are correctly related.

An important consequence of the completeness result for EPP is that we can combine it with Theorem 4.5 (deadlock-freedom for choreographies) for ensuring deadlock-freedom of endpoint projections. This formalises our *deadlock-freedom-by-design* property, mentioned in the introduction.

Corollary 6.23 (Deadlock-Freedom-by-design). *Let C be linear and well-typed. Then, for any P such that $C \lfloor \xrightarrow{\tilde{\mu}}{}^* P$, we have that either $P \xrightarrow{\mu'} P'$ for some P', μ' or $\mathbf{0} \prec P$.*

Typing Expressiveness. In this work we do not provide a typing discipline for endpoint processes, since we can actually write choreographies that would not be typable in other session typing systems such as [9,2,15]. For example, the choreography,

$$\begin{array}{l} \tau[\mathtt{p}], \tau'[\mathtt{q}] \textbf{ start } : a(k);\; \tau[\mathtt{p}] \texttt{ -> } \tau'[\mathtt{q}] : k[l_1]; \\ \tau[\mathtt{p}], \tau'[\mathtt{q}] \textbf{ start } : a(k');\; \mathsf{if}\, e@\tau\, \mathsf{then}\, \tau[\mathtt{p}] \texttt{ -> } \tau'[\mathtt{q}] : k'[l_2]\, \mathsf{else}\, \tau \texttt{ -> } \tau' : k'[l_3] \end{array}$$

is linear and typable (a's type is $\mathtt{p} \texttt{ -> } \mathtt{q} : \{l_1 : \mathsf{end},\; l_2 : \mathsf{end},\; l_3 : \mathsf{end}\}$). However, the endpoint for τ' would not be typable with contra-variant input typing.

Communication safety. Even without an endpoint typing discipline, our EPP generates code which enjoys standard communication safety. Below, we write

$P\langle\!\langle k[\mathtt{p}]!\mathtt{q}\langle \boldsymbol{w}\rangle\rangle\!\rangle$ if an output on k from role $\mathtt{p}$ to role $\mathtt{q}$ is enabled in P (not under any prefix). We write $P\langle\!\langle k[\mathtt{p}]?\mathtt{q}(x)\rangle\!\rangle$ for enabled inputs of values, $P\langle\!\langle k[\mathtt{p}]?\mathtt{q}(l)\rangle\!\rangle$ for labels, and $P\langle\!\langle k[\mathtt{p}]?\mathtt{q}(k')\rangle\!\rangle$ for delegated channels. We omit parameters when they do not matter. In the sequel, a reduction context $\mathcal{E}$ is defined as

$$\mathcal{E} ::= \mathcal{E}|P \quad | \quad (\boldsymbol{\nu} k)\, P \quad | \quad \mathsf{rec}\, X(\tilde{x},\tilde{k}) = P' \,\mathsf{in}\, \mathcal{E}$$

Corollary 6.24 (Communication safety). *Let C be linear and well-typed such that $C \downarrow \rightarrow^* P$. Then,*

1. *(Linearity 1) if P has a redex at k then $P \equiv \mathcal{E}[k : h]$ and either*
   (a) *$P\langle\!\langle k[\mathtt{p}]!\mathtt{q}\rangle\!\rangle$ and k is enabled exactly once in $\mathcal{E}$; or*
   (b) *$P\langle\!\langle k[\mathtt{p}]?\mathtt{q}\rangle\!\rangle$, k is enabled exactly once in $\mathcal{E}$ and $h = h_1 \cdot (\mathtt{q}, \mathtt{p}, \boldsymbol{w}) \cdot h_2$; or*
   (c) *$P\langle\!\langle k[\mathtt{p}]?\mathtt{q}\rangle\!\rangle$, $P\langle\!\langle k[\mathtt{p}]!\mathtt{q}\rangle\!\rangle$ and k is enabled exactly twice in $\mathcal{E}$.*
2. *(Linearity 2) if P has a redex at a then $a[\mathtt{p}]$ is enabled exactly once in P.*
3. *(Error-freedom) If $P \equiv \mathcal{E}[k : (\mathtt{q}, \mathtt{p}, \boldsymbol{w}) \cdot h]$ then*
   (a) *$P\langle\!\langle k[\mathtt{p}]?\mathtt{q}(x)\rangle\!\rangle$ implies $\boldsymbol{w} = v$ for some v;*
   (b) *$P\langle\!\langle k[\mathtt{p}]?\mathtt{q}(l)\rangle\!\rangle$ implies $\boldsymbol{w} = l$ for some l; and*
   (c) *$P\langle\!\langle k[\mathtt{p}]?\mathtt{q}(k')\rangle\!\rangle$ implies $\boldsymbol{w} = k''[\mathtt{p}']$ for some k'',$\mathtt{p}'$.*

7 Example

We report here an example where two peers want to exchange a file. Before exchanging the file, however, each peer wants to ensure that the other is run by a user possessing valid credentials in a system (authentication). We first define appropriate global types for the protocols that we intend to use. Then, we present a choreography that uses the global types for implementing our scenario. Finally, we show the endpoint system generated by the EPP of the choreography.

Types. The global type for the file exchange, G_{file}, follows:

$$G_{\mathsf{file}} = \mathtt{P1} \to \mathtt{P2} : \langle \mathbf{string}\rangle;\ \mathtt{P2} \to \mathtt{P1} : \left\{ \begin{array}{l} ok : \mathtt{P2} \to \mathtt{P1} : \langle \mathbf{string}\rangle; \\ \qquad\qquad \mathtt{P1} \to \mathtt{P2} : \left\{ \begin{array}{l} ok : \mathtt{P1} \to \mathtt{P2} : \langle \mathbf{int}\rangle; \\ \qquad \mathtt{P1} \to \mathtt{P2} : \langle \mathbf{file}\rangle; \\ \qquad \mathsf{end}, \\ quit : \mathsf{end} \end{array} \right\}, \\ quit : \mathsf{end} \end{array} \right\}$$

In G_{file}, the two peers are respectively identified by roles P1 and P2. The protocol starts with P1 sending her username (her identity) to P2. P2 then chooses if she wants to proceed or not. If she does, she sends her own username to P1, who will perform the same choice. If also P1 chooses to proceed, she will first send the size of the file (as an integer) to P2, and then the file itself.

The type modelling user authentication, G_{auth}, is the same global type reported in the introduction for the OpenID protocol [23]. Observe that G_{file} does not depend on G_{auth}, as it can be used in combination with different authentication protocols (though we do not exploit this aspect here).

We introduce also a third global type, G_{fs}, that will enable P2 to use an external file server for storing the received file if the latter is too big for its local storage. The type allows for the delegation of a channel that will receive a file:

$$G_{\mathsf{fs}} \;=\; \mathtt{C} \texttt{ -> } \mathtt{FS} : \langle\, \mathtt{P1}?(\mathbf{file});\ \mathsf{end} \,\rangle$$

Choreography. The following choreography, named C, implements our scenario. In C, the two peers are represented, respectively, by threads p_1 and p_2:

1. $\mathsf{p}_1[\mathtt{P1}], \mathsf{p}_2[\mathtt{P2}]$ **start** $: a(k);\ \mathsf{p}_1[\mathtt{P1}].user_1$ -> $\mathsf{p}_2[\mathtt{P2}].x_2 : k;$
2. $\mathsf{p}_2[\mathtt{RP}], \mathsf{p}_1[\mathtt{U}]$ **start** $\mathsf{ip}[\mathtt{IP}] : b(k');\ \mathsf{p}_2[\mathtt{RP}].x_2$ -> $\mathsf{ip}[\mathtt{IP}].u_1 : k';$
3. $\mathsf{p}_1[\mathtt{U}].\mathsf{authStr}(user_1, pwd_1)$ -> $\mathsf{ip}[\mathtt{IP}].y_1 : k';$
4. if $\mathsf{check}(u_1, y_1)@\mathsf{ip}$
5. then $\mathsf{ip}[\mathtt{IP}]$ -> $\mathsf{p}_2[\mathtt{RP}] : k'[ok];\ \mathsf{p}_2[\mathtt{P2}]$ -> $\mathsf{p}_1[\mathtt{P1}] : k[ok];$
6. $\quad \mathsf{p}_2[\mathtt{P2}].user_2$ -> $\mathsf{p}_1[\mathtt{P1}].x_1 : k;$
7. $\quad \mathsf{p}_1[\mathtt{RP}], \mathsf{p}_2[\mathtt{U}]$ **start** $\mathsf{ip}_2[\mathtt{IP}] : b(k'');$
8. $\quad \mathsf{p}_1[\mathtt{RP}].x_1$ -> $\mathsf{ip}_2[\mathtt{IP}].u_2 : k'';$
9. $\quad \mathsf{p}_2[\mathtt{U}].\mathsf{authStr}(user_2, pwd_2)$ -> $\mathsf{ip}_2[\mathtt{IP}].y_2 : k'';$
10. $\quad$ if $\mathsf{check}(u_2, y_2)@\mathsf{ip}_2$
11. $\quad$ then $\mathsf{ip}_2[\mathtt{IP}]$ -> $\mathsf{p}_1[\mathtt{RP}] : k''[ok];\ \mathsf{p}_1[\mathtt{P1}]$ -> $\mathsf{p}_2[\mathtt{P2}] : k[ok];$
12. $\quad\quad \mathsf{p}_1[\mathtt{P1}].size_1$ -> $\mathsf{p}_2[\mathtt{P2}].size_2 : k;$
13. $\quad\quad$ if $(size_2 < limit)@\mathsf{p}_2$
14. $\quad\quad\quad$ then $\mathsf{p}_1[\mathtt{P1}].file_1$ -> $\mathsf{p}_2[\mathtt{P2}].file_2 : k$
15. $\quad\quad\quad$ else $\mathsf{p}_2[\mathtt{C}]$ **start** $\mathsf{fs}[\mathtt{FS}] : c(k''');$
16. $\quad\quad\quad\quad \mathsf{p}_2[\mathtt{C}]$ -> $\mathsf{fs}[\mathtt{FS}] : k'''\langle\!\langle k[\mathtt{P2}] \rangle\!\rangle;$
17. $\quad\quad\quad\quad \mathsf{p}_1[\mathtt{P1}].file_1$ -> $\mathsf{fs}[\mathtt{P2}].file_3 : k$
18. $\quad\quad$ else $\mathsf{ip}_2[\mathtt{IP}]$ -> $\mathsf{p}_1[\mathtt{RP}] : k''[quit];\ \mathsf{p}_1[\mathtt{P1}]$ -> $\mathsf{p}_2[\mathtt{P2}] : k[quit]$
19. else $\mathsf{ip}[\mathtt{IP}]$ -> $\mathsf{p}_2[\mathtt{RP}] : k'[quit];\ \mathsf{p}_2[\mathtt{P1}]$ -> $\mathsf{p}_1[\mathtt{P1}] : k[quit]$

First, a session k (implementing G_{file}) for the file exchange is opened through public channel a. Following G_{file}, p_1 sends her username to p_2. Then, p_1 and p_2 start an authentication session k' (implementing G_{auth}), spawning a new thread ip, for authenticating p_1. In Line 4, ip checks the credentials (encoded as a string by $\mathsf{authStr}$) given by p_1. If they are not valid, all the sessions are closed by sending label *quit* first to p_2 and then to p_1. Otherwise, p_1 is authenticated and now a new session is started for authenticating p_2. The same protocol is repeated, with a new spawned thread ip_2. Observe that threads ip and ip_2 have been spawned through the same public channel b. This will be reflected in the EPP of the choreography. Line 10 implements the same choice performed before by ip, but in this case it is on the credentials of p_2. If they are, p_1 sends the file size to p_2. In Line 13, p_2 decides if she wants to receive the file or to delegate the receiving to another party because of excessive size. Lines 15-17 define the delegation. There, p_2 opens a session with a file server fs, delegates the file exchange session k and, finally, p_1 sends the file to fs, concluding the protocol.

EPP. The EPP of the choreography C above is $C \downarrow = P_{\mathsf{p}_1} \mid P_{\mathsf{p}_2} \mid P_{\mathsf{ip}} \mid P_{\mathsf{fs}}$, where

$$P_{\mathsf{p}_1} = \overline{a}[\mathtt{P1},\mathtt{P2}](k);\ k!\mathtt{P2}\langle user_1\rangle;\ b[\mathtt{U}](k');\ k'!\mathtt{IP}\langle \mathsf{authStr}(user_1, pwd_1)\rangle;$$
$$k?\mathtt{P2}\,\&\left\{\begin{array}{l} ok : k?\mathtt{P2}(x_1);\ \overline{b}[\mathtt{RP},\mathtt{U},\mathtt{IP}](k'');\ k''!\mathtt{IP}\langle x_1\rangle; \\ \qquad k''?\mathtt{IP}\,\&\left\{\begin{array}{l} ok : k!\mathtt{P2} \oplus ok;\ k!\mathtt{P2}\langle size_1\rangle;\ k!\mathtt{P2}\langle file_1\rangle \\ quit : k!\mathtt{P2} \oplus quit \end{array}\right\} \\ quit : \mathbf{0} \end{array}\right\}$$

$$P_{\mathsf{p}_2} = a[\mathtt{P2}](k);\ k?\mathtt{P1}(x_2);\ \overline{b}[\mathtt{RP},\mathtt{U},\mathtt{IP}](k');\ k'!\mathtt{IP}\langle x_2\rangle;$$
$$k'?\mathtt{IP}\,\&\left\{\begin{array}{l} ok : k!\mathtt{P1} \oplus ok;\ k!\mathtt{P1}\langle user_2\rangle;\ b[\mathtt{U}](k'');\ k''!\mathtt{IP}\langle \mathsf{authStr}(user_2, pwd_2)\rangle; \\ \qquad k?\mathtt{P1}\,\&\left\{\begin{array}{l} ok : k!\mathtt{P1}\langle size_2\rangle;\ \mathsf{if}\ (size_2 < limit) \\ \qquad\qquad\qquad\qquad \mathsf{then}\ k?\mathtt{P1}(file_2) \\ \qquad\qquad\qquad\qquad \mathsf{else}\ \overline{c}[\mathtt{C},\mathtt{FS}](k'''); k!\mathtt{FS}\langle\langle k\rangle\rangle \\ quit : \mathbf{0} \end{array}\right\} \\ quit : k!\mathtt{P1} \oplus quit \end{array}\right\}$$

$$P_{\mathsf{ip}} = !b[\mathtt{IP}](k_{\mathsf{ip}});\ k_{\mathsf{ip}}?\mathtt{RP}(u_1);\ k_{\mathsf{ip}}?\mathtt{U}(y_1);\ \mathsf{if}\,(\mathsf{check}(u_1, y_1))\,\mathsf{then}\ k_{\mathsf{ip}}!\mathtt{RP} \oplus ok\ \mathsf{else}\ k_{\mathsf{ip}}!\mathtt{RP} \oplus quit$$

$$P_{\mathsf{fs}} = !c[\mathtt{FS}](k_{\mathsf{fs}});\ k_{\mathsf{fs}}?\mathtt{C}(\!(k)\!);\ k?\mathtt{P2}(file_3)$$

Note that EPP merged ip and ip_2 into a single reusable service P_{ip}.

Using our type system, it can be verified that C is well-typed. Furthermore, P enjoys the progress property and is communication-safe.

8 Related Work, Future Work and Conclusions

Related Work. Global methods for describing communication have been practised in many different forms, including Message Sequence Charts [16], Petri Nets [26], UML [22], security protocols [3] and automata theory [12]. These works offer a useful basis for design/specification/analysis. However, they do not address different layers of abstraction as in this work. Moreover, they are not intended as fully-fledged programming languages since they lack in, e.g., general control structures and constructs for value passing and state change.

Our global types with no explicit session channels are inspired by [2]. We did not consider several extensions – such as global types with exceptions [7], parameterised global types [29], assertions [4], and multirole global types [11] – since they do not influence our main results. Previous works check systems against the specifications of multiparty protocols. However, the systems are not defined by means of choreographies, but as endpoint programs. Previous work handled progress in multiparty sessions by building additional restrictions on top of standard session typing [2]. In our work, instead, it is an immediate consequence of the correctness of our model and does not need any additional check. Our type system is also more relaxed, since we can type systems that are not typable in [2]. Our notion of linearity of shared channels in choreographies is inspired by the one used in [15] on global types.

[9] proposes a choreography model based on binary session types. Our model differs in several aspects: we made threads explicit, we decoupled selection from

communication and we have added multiparty sessions, delegation and asynchronous messaging. Moreover, we replaced state variables with binders. [9] bases its EPP theorem on three conditions, namely connectedness, well-threadedness and coherence. Since we have chosen to relax dependencies between threads, e.g., we consider $\tau_1[\mathtt{p}] \to \tau_2[\mathtt{q}] : k.\tau_3[\mathtt{p}'] \to \tau_4[\mathtt{q}'] : k$ as a parallel, we need no connectedness and, partly, no well-threadedness. The latter has been replaced with explicit threads, thread typing and linearity. Finally, the EPP in [9] has type preservation, while we need no endpoint typing since we can ensure communication safety directly by EPP.

The swap relation $\simeq_{\mathcal{C}}$ that we use for representing concurrent endpoint behaviour in our global programs is similar to the notion of *delayed input* presented in [18]. Our $\simeq_{\mathcal{C}}$ is a more syntactic notion that checks the thread names of th e terms instead of all the names in a transition label as in delayed input. This aspect, made possible by the fact that in our global calculus threads are explicit, simplifies our result of EPP invariance under swapping. Moreover, $\simeq_{\mathcal{C}}$ handles branching (if-then-else), which is not handled in [18].

Future Work. Our model does not treat public channel passing (and restriction). Although it may be a natural feature to have, e.g., in scenarios where entities need to exchange links, we have decided to leave it as future work. The main challenge is to statically establish where such channels can be located in the endpoints. Another possible extension is to include exceptions in our choreography language in the spirit of what is suggested in [8]. This would require a different endpoint calculus capable of dealing with exceptions such as [7]. The work presented in this paper is a step towards the implementation of a fully-fledged and strongly typed choreography language. We are currently implementing our model using the JOLIE framework [20] as the target of our EPP.

Conclusions. This work presented a fully-fledged framework where both protocols and systems can be defined globally in a consistent manner. We have shown how to automatically generate endpoint systems with our correct EPP. We can now globally design multiparty asynchronous systems and automatically generate an implementation which satisfies the progress property.

References

1. Business Process Model and Notation. `http://www.omg.org/spec/BPMN/2.0/`.
2. L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
3. K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *Proc. of CSF*, pages 124–140, 2009.
4. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.
5. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *Proc. of Coordination*, volume 4038 of *LNCS*, pages 63–81. Springer-Verlag, 2006.

6. L. Caires and H. T. Vieira. Conversation types. *Theoretical Computer Science*, 411(51-52):4399–4440, 2010.
7. S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty sessions. In *FSTTCS*, volume 8, pages 338–351, 2010.
8. M. Carbone. Session-based choreography with exceptions. In *Proc. of PLACES*, volume 241, pages 35–55. ENTCS, 2008.
9. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *Proc. of ESOP*, volume 4421 of *LNCS*, pages 2–17. Springer-Verlag, 2007.
10. M. Carbone and F. Montesi. Typed multiparty global programming, Technical Report, 2011. `http://www.itu.dk/people/fabr/multichor`.
11. P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *Proc. of POPL*, pages 435–446. ACM, 2011.
12. X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. *International Journal on Web Service Res.*, 2(4):68–93, 2005.
13. S. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, Nov. 2005.
14. K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In *Proc. of ICDCIT*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.
15. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. of POPL*, volume 43(1), pages 273–284. ACM, 2008.
16. International Telecommunication Union. Recommendation Z.120: Message sequence chart, 1996.
17. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *Proc. of SEFM*, pages 323–332. IEEE, 2008.
18. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.
19. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, Sept. 1992.
20. F. Montesi, C. Guidi, and G. Zavattaro. Composing Services with JOLIE. In *Proc. of ECOWS*, pages 13–22, 2007.
21. D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP*, LNCS, pages 316–332. Springer-Verlag, 2009.
22. OMG. Unified modelling language, version 2.0, 2004.
23. OpenID. OpenID specifications. `http://openid.net/developers/specs/`.
24. PI4SOA. http://www.pi4soa.org, 2008.
25. B. C. Pierce. *Types and Programming Languages*. MIT Press, MA, USA, 2002.
26. W. van der Aalst. Inheritance of Interorganizational Workflows: How to agree to disagree without loosing control? *Info. Tech. and Manag. J.*, 2(3):195–231, 2002.
27. V. Vasconcelos and N. Yoshida. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.
28. W3C WS-CDL Working Group. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/, 2004.
29. N. Yoshida, P.-M. Deniélou, A. Bejleri, and R. Hu. Parameterised multiparty session types. In *FOSSACS'10*, volume 6014 of *LNCS*, pages 128–145, 2010.