Deadlock-freedom-by-design: Multiparty Asynchronous Global Programming

Marco Carbone Fabrizio Montesi

IT University of Copenhagen

{carbonem,fmontesi}@itu.dk

Abstract

Over the last decade, *global descriptions* have been successfully employed for the verification and implementation of communicating systems, respectively as *protocol specifications* and *choreographies*. In this work, we bring these two practices together by proposing a purely-global programming model. We show a novel interpretation of *asynchrony* and *parallelism* in a global setting and develop a typing discipline that verifies choreographies against protocol specifications, based on multiparty sessions. Exploiting the nature of global descriptions, our type system defines a new class of deadlock-free concurrent systems (*deadlock-freedom-by-design*), provides type inference, and supports session mobility. We give a notion of Endpoint Projection (EPP) which generates correct entity code (as π-calculus terms) from a choreography. Finally, we evaluate our approach by providing a prototype implementation for a concrete programming language and by applying it to some examples from multicore and service-oriented programming.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.2 [*Semantics of Programming Languages*]: Process Models

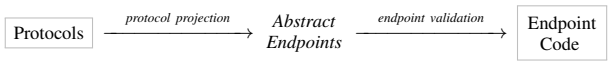
General Terms Design, Languages, Theory

Keywords Concurrency, Choreography, Types, Sessions.

1. Introduction

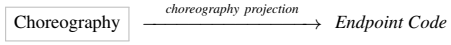
Global descriptions represent a powerful paradigm for designing communicating systems where the programmer gives a global view of how messages are exchanged during execution, instead of separately defining the behaviour of each *endpoint* (entity). Then, the local behaviour of each endpoint can be automatically generated by means of *EndPoint Projection* (EPP). The paradigm has been studied in formal models [10, 15, 22, 26], standards [1, 37], and language implementations [23, 33, 36]. Global descriptions have a great impact on the quality of software, as they represent "formal blueprints" of how communicating systems should behave and offer a concise view of the message flows enacted by a system. In particular, they (i) lower the possibility of introducing programming errors and (ii) ease the task (both manual and automatic) of detecting them. Global descriptions can be used at different levels of abstraction, ranging from abstract descriptions of *protocols* to descriptions of concrete system implementations, dubbed *choreographies*. These different incarnations respectively underpin two recent and successful development methodologies.

In the first methodology, programmers design abstract protocols using global descriptions [22, 23]. These are automatically projected onto *abstract endpoint* specifications which are finally used for the static verification of manually written endpoint code:



The approach above has the benefit of producing very clear protocol specifications. However, it deprives the programmer from a global view of the system when dealing with its implementation. A major consequence is that programming becomes error-prone when dealing with the actual interleaving of different protocol instances. For example, it can easily lead to deadlocked systems [7].

The second methodology deals with system implementations using choreographies [36, 37]. Programmers can write a choreography and then automatically project an executable system from it:



Here, the main advantage is the precise view given by choreographies on the possible system executions. However, choreographies lack in abstraction wrt global protocol descriptions and their programming needs to be disciplined with additional tools. Current disciplines for choreography-driven programming are based on writing abstract endpoint descriptions and then using them to check the behaviour of each endpoint, directly on the choreography or its EPP [15, 36]. Hence, in these models, we lose a global view of the system when describing its protocol specifications.

Inspired by these observations and by private conversations with our industry collaborators [2, 20, 31, 35, 37], we ask:

> *Can we design a unified framework that combines global descriptions of protocols and implementations?*

Clearly, a positive answer would retain the advantages of global descriptions for *both* the writing of protocols and that of implementations. Moreover, a natural following question is whether such a unified framework could offer more than just the sum of the parts: are there other advantages that can arise from the combination of global protocol descriptions with choreographies?

In order to answer the questions above, we build and analyse a model for a fully global framework. In our model, developers design both protocols and implementations from a global viewpoint. Endpoint implementations can then be automatically generated:

[Copyright notice will appear here once 'preprint' option is removed.]

The challenge of reaching our objective is twofold. First, since we aim at designing a model where choreographies can instantiate different protocols multiple times and interleave their execution, the model should ensure 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.

Main contributions. We provide the following contributions:

Multiparty Choreographies. We introduce a choreography model with *multiparty* protocol instances (*sessions*) as first-class elements (§ 3) and provide an EPP that, under simple restrictions, correctly generates endpoint code from a choreography (§ 5).

Asynchrony and Parallelism. Our framework gives a novel and concise interpretation of asynchrony and parallelism, by inferring the implicit concurrent behaviour specified in a choreography (§ 3).

Typing and Type inference. We provide a type system (§ 4) for checking choreographies against protocol specifications given as multiparty session types [22]. Our type analysis plays a major role in ensuring the correctness of EPP code. Interestingly, due to the global nature of choreographies, our framework can generate correct endpoint code that is not allowed by current multiparty session typings (§ 5, Typing expressiveness). We also give a type inference technique supporting the opposite methodology, i.e., extracting the protocols implemented in a choreography (§ 4, Type inference).

Delegation. This is the first work to provide a choreography model supporting *session delegation*, a mobility mechanism for delegating the continuation of a protocol (§ 3). Due to asynchrony and parallelism, typing delegation (§ 4) is nontrivial 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 (§ 5, Corollary 1), a notoriously hard problem in multiparty sessions types [7]. This feature follows from using a choreography as initial design tool.

Implementation and Evaluation. We provide a prototype implementation of our framework (§ 6), featuring a programming language (Chor), an IDE, and an EPP that support the development of concurrent systems using our global methodology. We use Chor to evaluate our programming model against examples of different nature, from multicore to distributed programming (§ 7).

Proofs and full definitions can be found in [14].

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 in a system, whereas a choreography describes a concrete system implementing one or more protocols. We represent protocols with *global types* [22], global descriptions where entities are abstracted as *roles* that communicate following a given conversation structure.

Example 1 (Two-buyer protocol)**.** In this protocol, two buyers B1 and B2 wish to share the purchase of a product from a seller S:

1. B1 -> S : ⟨**string**⟩; S -> B1 : ⟨**int**⟩; S -> B2 : ⟨**int**⟩; B1 -> B2 : ⟨**int**⟩;
2. B2 -> S : { ok : B2 -> S : ⟨**string**⟩; S -> B2 : ⟨**date**⟩, $quit$: end }

Above, B1, B2 and S are called *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 of 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. A session is an instance of a protocol and implements communications between some threads. Sessions can be dynamically created by threads.

Example 2 (Two-buyer choreography)**.** We give a choreography implementing the two-buyer protocol in Example 1.

```
1.  b1[B1], b2[B2] start s[S] : a(k);
2.  b1[B1].book -> s[S].x1 : k;
3.  s[S].quote(x1) -> b1[B1].y1 : k;
4.  s[S].quote(x1) -> b2[B2].z1 : k;
5.  b1[B1].contrib(y1) -> b2[B2].z2 : k;
6.  if (z1 − z2 ≤ 100)@b2
7.  then b2[B2] -> s[S] : k[ok];
8.       b2[B2].addr -> s[S].x2 : k;
9.       s[S].ddate -> b2[B2].z3 : k
10. else b2[B2] -> s[S] : k[quit]
```

In Line 1, threads b_1, b_2 and freshly spawned thread s start a session 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 "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 "quote($book$)". Then, b_1 tells b_2 the amount she wishes to contribute for the purchase, namely "contrib(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 $\mathsf{b}_2[\mathtt{B2}]$ -> s[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. □

Observe that the structure of session k in Example 2 is that of the protocol given in Example 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, and create new threads and sessions. In Line 1 of Example 2, 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 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 with two other sessions, k' and k'', that b_1 and b_2 will respectively use for getting help in the transaction.

```
   ...as Lines 1-5 in Example 2...
6.  b1[B] start h1[H] : b(k');                          }
7.  b1[B].(contrib(y1)/2) -> h1[H].y : k';              }  C1
8.  b1[B] -> h1[H] : k'[done];                          }
9.  b2[B] start h2[H] : b(k'');                         }
10. b2[B].((z1 − z2)/2) -> h2[H].z : k'';               }
11. b2[B] -> h2[H] : k''[del];                          }  C2
12. b2[B].z1 -> h2[H].z' : k'';                         }
13. b2[B] -> h2[H] : k''⟨⟨k[B2]⟩⟩;                       }
14. if ((z/z') ≤ 30%)@h2                                }
15. then h2[B2] -> s[S] : k[ok];                        }
16.      h2[B2].addr -> s[S].x2 : k;                    }  C3
17.      s[S].ddate -> h2[B2].z'' : k                   }
18. else h2[B2] -> s[S] : k[quit]                       }
```

The choreography starts with the first 5 lines of that in Example 2. In block C_1, b_1 starts a new session with a helper h_1, asks it to

contribute for half of its part (Line 7), and informs it that it does not need to do more (Line 8). On the other hand, in block C_2, b_2 does the same with another thread h_2 until Line 11. Differently now, b_2 asks h_2 to continue session k *by taking on its role* (Line 11). Then, it sends the total price received from s to h_2 (Line 12) and *delegates* the session reference (Line 13). Finally, in block C_3, h_2 completes the two-buyer protocol instead of b_2, checking that its own contribution is less than 30% of the total price. Note that h_1 and h_2 are started through the same public channel b, which acts as a reusable shared channel in multiparty session types [22]. □

Our model has two features that interestingly influence choreography interpretation in subtly different ways. Firstly, *parallelism*: thread executions may concurrently proceed without any predetermined ordering unless causal constraints are introduced. Secondly, *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 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, the interpretation of $C_1; C_2$ should be equivalent to that of $C_2; C_1$, and, in general, to that of any interleaving of C_1 and C_2. Furthermore, in Lines 3 and 4 of Example 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. Parallelism and asynchrony are respectively handled by our *swapping* relation and our asynchronous semantics, both formalised in the next section.

In general, we say that our relaxed sequential operator lifts the programmer from expressing the degree of concurrency (asynchrony and parallelism) of a system. Indeed, our framework will automatically infer the latter by looking at the thread identifiers. We made this choice in favour of design minimality and simplicity. In real tools, combining the sequential operator with explicit primitives for, e.g., parallelism, may be preferrable for clarity purposes. We discuss this in § 9, Sequential and Parallel Operators.

3. Choreographies with Multiparty Sessions

We introduce the *Global Calculus (GC)*, a choreography model with multiparty asynchronous sessions.

Syntax. The syntax of our calculus is reported in Figure 1.

$$
\begin{array}{rcll}
C & ::= & \eta; C & \textit{(sequence)} \\
& | & \mathsf{if}\, e@\tau\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2 & \textit{(conditional)} \\
& | & \mathbf{0} & \textit{(inaction)} \\
& | & \mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C'\, \mathsf{in}\, C & \textit{(recursion)} \\
& | & X(\widetilde{e@\tau}, \tilde{k}, \tilde{\tau}) & \textit{(call)} \\
& | & (\boldsymbol{\nu} r)\, C & \textit{(restriction)} \\
\\
\eta & ::= & \tau_1[\mathsf{p}].e \texttt{->} \tau_2[\mathsf{q}].x : k & \textit{(communication)} \\
& | & \tau_1[\mathsf{p}] \texttt{->} \tau_2[\mathsf{q}] : k[l] & \textit{(selection)} \\
& | & \tau_1[\mathsf{p}] \texttt{->} \tau_2[\mathsf{q}] : k\langle\langle k'[\mathsf{p}']\rangle\rangle & \textit{(delegation)} \\
\\
| & \multicolumn{3}{l}{\tau_1[\mathsf{p}_1], ..., \tau_n[\mathsf{p}_n]\ \mathbf{start}\ \tau_{n+1}[\mathsf{p}_{n+1}], ..., \tau_m[\mathsf{p}_m] : a(k)} & \textit{(start)}
\end{array}
$$

Figure 1. Global Calculus with Multiparty Sessions, syntax.

C is a choreography, τ is a thread, and k is a session (identifier). Interactions between threads are specified by the term $\eta; C$ which reads: *the system may execute the interaction η and continue as C*. We distinguish four different kinds of interaction: *(start)*, *(communication)*, *(selection)*, and *(delegation)*. *(start)* denotes session initiation: threads τ_i (for $1 \leq i \leq m$) start a new multiparty session through public channel a and tag it with a fresh identifier k, called session channel. The first n threads, dubbed the *active threads*, are already running, while $\tau_{n+1}, \ldots, \tau_m$, dubbed the *service threads*, are dynamically created and started. 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. *(communication)* denotes a communication where thread τ_1 sends, over session k, the evaluation of a first-order expression e to thread τ_2, which binds it to variable x[1]. In *(selection)*, τ_1 communicates to τ_2 her selection of branch l. Through *(delegation)*, τ_1 delegates to τ_2 over k her role p' in session k'.

GC also offers other standard programming language constructs. In *(conditional)*, expression e is labelled with a thread name, indicating where it is evaluated. *(recursion)* and *(call)* model standard recursive procedures, where each variable in $\tilde{x}$ or expression in $\tilde{e}$, respectively, is located at a thread in $\tilde{\tau}$. In *(call)*, we assume that each expression can only be either a variable or a value. *(restriction)*, which is only used at runtime and cannot be used in programs, models name restriction. r (for restricted name) can be a thread or a session channel. The term $\mathbf{0}$ denotes termination.

In a term $\eta; C$, η can bind session channels, threads and variables. When η is a *(start)*, $\tau_1, \ldots, \tau_n$ are free while k and $\tau_{n+1}, \ldots, \tau_m$ are bound (since they are freshly created). If η is a *(communication)*, variable x is bound. As usual, r is bound in $(\boldsymbol{\nu} r)\ C$. We often omit $\mathbf{0}$, empty vectors, and irrelevant variables. In the remainder, $(\boldsymbol{\nu} r_1, \ldots, r_n)$ is a shortcut for $(\boldsymbol{\nu} r_1) \ldots (\boldsymbol{\nu} r_n)$.

Semantics. Above, we stated that the term $\eta; C$ specifies a system that *may* execute the interaction η and then continue as C. Threads, however, are assumed to run in parallel. As a consequence, some actions in C may be performed before η. For example, blocks C_1 and C_2 from Example 3 describe the behaviour of different threads. Therefore, as discussed in § 2, in an actual system run of these threads, their executions may interleave due to parallelism. To deal with such cases, we define the *swapping* congruence relation $\simeq_{\mathcal{C}}$, which allows permutations of this kind of interaction sequences[2]. $\simeq_{\mathcal{C}}$ is defined as the smallest congruence satisfying the rules in Figure 2. $\simeq_{\mathcal{C}}$ exchanges terms with different threads. The top rule swaps two conditionals: C_1' and C_2 are swapped to preserve the semantics of the term wrt the evaluations of the conditions. The bottom-left rule swaps two interactions η and η' that do not share any thread names (calculated by $\mathsf{thr}(\eta)$). The bottom-right rule swaps an interaction η out of a conditional if it prefixes both branches and does not involve the thread that checks the condition.

Asynchronous messaging can cause situations as the one discussed for Lines 3 and 4 of Example 2 in § 2, where s sends the quote to b_1, then to b_2, and b_2 may receive the quote before b_1. Unlike for parallelism, we address this issue directly in the operational semantics. This is because asynchrony is somehow asymmetric: even though the receiving actions may interleave in a different order wrt that in the choreography, the sending actions instead will surely happen in the specified order, since the thread performing the outputs is the same. This is different from parallelism, where the ordering of *both* receiving and sending actions may change. It is unsafe to manipulate the syntax of the choreography for simulating asynchrony, since when we will generate the code for the sender thread (cf. § 5) remembering the order of outputs will be important.

Figure 3 contains the rules defining the labelled reduction semantics for GC, whose labels λ are defined as:

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

[1] For clarity, we annotate threads with roles. This is necessary only for *(start)* since roles can be inferred from session identifiers in all other terms.

[2] Handling parallelism with a congruence simplifies our development, since swaps in a choreography do not influence the behaviour of its EPP [14].

$$\frac{\tau \neq \tau'}{\mathsf{if}\, e@\tau\, \mathsf{then}\, (\, \mathsf{if}\, e'@\tau'\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2\,)\, \mathsf{else}\, (\, \mathsf{if}\, e'@\tau'\, \mathsf{then}\, C_1'\, \mathsf{else}\, C_2'\,) \;\simeq_{\mathcal{C}}\; \mathsf{if}\, e'@\tau'\, \mathsf{then}\, (\mathsf{if}\, e@\tau\, \mathsf{then}\, C_1\, \mathsf{else}\, C_1')\, \mathsf{else}\, (\mathsf{if}\, e@\tau\, \mathsf{then}\, C_2\, \mathsf{else}\, C_2')}$$

$$\frac{\mathsf{thr}(\eta) \cap \mathsf{thr}(\eta') \;=\; \emptyset}{\eta;\, \eta' \;\simeq_{\mathcal{C}}\; \eta';\, \eta} \qquad \frac{\tau \;\notin\; \mathsf{thr}(\eta)}{\mathsf{if}\, e@\tau\, \mathsf{then}\, (\eta; C_1)\, \mathsf{else}\, (\eta; C_2) \;\simeq_{\mathcal{C}}\; \eta; \mathsf{if}\, e@\tau\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2}$$

Figure 2. Global Calculus with Multiparty Sessions, swap relation $\simeq_{\mathcal{C}}$.

$$\begin{array}{rll}
\lfloor^{\mathrm{C}}|_{\textsc{Act}}\rceil & \eta \in \{\textit{(selection)}, \textit{(delegation)}, \textit{(start)}\} \;\Rightarrow\; \eta; C \xrightarrow{\eta} (\boldsymbol{\nu}\tilde{r})\, C & (\tilde{r} = \mathsf{bn}(\eta)) \\
\lfloor^{\mathrm{C}}|_{\textsc{Com}}\rceil & \eta = \tau_1[\mathtt{p}].e \texttt{->} \tau_2[\mathtt{q}].x : k \;\Rightarrow\; \eta; C \xrightarrow{\eta[v/e]} C[v/x@\tau_2] & (e \downarrow v) \\
\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil & C \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{r})\, C' \;\Rightarrow\; \eta; C \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{r})\, \eta; C' & \left(\begin{array}{lll} \mathsf{snd}(\eta) \in \mathsf{fn}(\lambda) & \tilde{r} = \mathsf{bn}(\lambda) & \eta \neq \textit{(start)} \\ \mathsf{rcv}(\eta) \notin \mathsf{fn}(\lambda) & \tilde{r} \notin \mathsf{fn}(\eta) & \end{array}\right) \\
\lfloor^{\mathrm{C}}|_{\textsc{If}}\rceil & \mathsf{if}\, e@\tau\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2 \xrightarrow{\mathsf{if}@\tau} C_i & (i = 1 \text{ if } e \downarrow \mathsf{true}\,,\; i = 2 \text{ otherwise}) \\
\lfloor^{\mathrm{C}}|_{\textsc{Ctx}}\rceil & C_1 \xrightarrow{\lambda} C_1' \;\Rightarrow\; \mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C_2\, \mathsf{in}\, C_1 \xrightarrow{\lambda} \mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C_2\, \mathsf{in}\, C_1' & \\
\lfloor^{\mathrm{C}}|_{\textsc{Res}}\rceil & C \xrightarrow{\lambda} C' \;\Rightarrow\; (\boldsymbol{\nu}r)\, C \xrightarrow{(\boldsymbol{\nu}r)\,\lambda} (\boldsymbol{\nu}r)\, C' & \\
\lfloor^{\mathrm{C}}|_{\textsc{Eq}}\rceil & C_1\, \mathcal{R}\, C_1' \quad C_1' \xrightarrow{\lambda} C_2' \quad C_2'\, \mathcal{R}\, C_2 \;\Rightarrow\; C_1 \xrightarrow{\lambda} C_2 & \mathcal{R} \in \{\simeq_{\mathcal{C}}, \equiv\}
\end{array}$$

Figure 3. Global Calculus with Multiparty Sessions, semantics.

A label is either an interaction η, an internal action $\mathsf{if}@\tau$ by thread τ (conditional), or another label with restricted name r (when new threads or session channels are created). Rule $\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil$ captures the asynchronous behaviour of endpoint systems, allowing a thread to send a message and then proceed freely before the intended receiver actually receives it. In the rule, the sender of η performs the action λ in the continuation C without waiting for the message in η to be delivered. We check that the receiver of η is not involved in λ since otherwise causality between η and λ would be violated. Finally, η is kept for the later observation of the message delivery. Rule $\lfloor^{\mathrm{C}}|_{\textsc{Act}}\rceil$ models interactions that are not *(communication)*. In the reductum, if η is a *(start)* then $\tilde{r}$ contains the freshly created service threads and the session channel. For all other cases, $\tilde{r}$ is empty. 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) with the *smart substitution* $C[v/x@\tau]$, which substitutes x with v only under the free name τ in C, modelling local variables. In rule $\lfloor^{\mathrm{C}}|_{\textsc{Eq}}\rceil$, the relation $\mathcal{R}$ can be either the swapping relation $\simeq_{\mathcal{C}}$ or *structural congruence* $\equiv$. Structural congruence handles name restriction and recursion unfolding. We just mention the rule for unfolding: $\mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C'\, \mathsf{in}\, X(\widetilde{v@\tau}, \tilde{k}, \tilde{\tau}) \;\equiv\; \mathsf{rec}\, X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) = C'\, \mathsf{in}\, C'[\tilde{v}/\tilde{x@\tau}]$. All other rules are standard and can be found in [14].

Choreographies enjoy deadlock-freedom since every term but $\mathbf{0}$ has a corresponding semantic rule that can reduce it. Formally,

Theorem 1 (Deadlock-Freedom)**.** *Let $C \not\equiv \mathbf{0}$ and contain no free variable names. Then, there exist C', λ such that $C \xrightarrow{\lambda} C'$.*

4. Typing Choreographies

We now present our typing system which allows to specify protocols in terms of global types [7, 22] and then check whether session behaviours in a choreography respect them.

Syntax. Figure 4 contains the syntax of global types. $\mathtt{p}$ -> $\mathtt{q} : \langle U \rangle; G$ abstracts an interaction from role $\mathtt{p}$ to 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 $\mathtt{p}$ -> $\mathtt{q} : \{l_i : G_i\}_{i \in I}$, role $\mathtt{p}$ can select one label l_i and continue as G_i. All other terms are standard.

$$\begin{array}{rll}
G ::= & \mathtt{p} \texttt{->} \mathtt{q} : \langle U \rangle; G & \textit{(communication)} \\
\mid & \mathtt{p} \texttt{->} \mathtt{q} : \{l_i : G_i\}_{i \in I} & \textit{(choice)} \\
\mid & \mathsf{end} & \textit{(inaction)} \\
\mid & \mathsf{rec}\, \mathbf{t}; G & \textit{(recursion)} \\
\mid & \mathbf{t} & \textit{(call)} \\
U ::= & S \mid G@\mathtt{p} & \textit{(values)} \\
S ::= & \mathbf{bool} \mid \mathbf{int} \mid \mathbf{string} \mid \ldots & \textit{(sort)}
\end{array}$$

Figure 4. Global Types, syntax.

Semantics. We give a semantics for global types, which expresses the (abstract) execution of protocols. $G \xrightarrow{\alpha} G'$ is the smallest relation on the recursion-unfolding of global types satisfying the rules given in Figure 5. A label α shows which interaction is consumed. Since our discussion on asynchrony and parallelism applies also to protocols, we need to capture these aspects also in their semantics. Similarly to $\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil$, $\lfloor^{\mathrm{G}}|_{\textsc{Async-Com}}\rceil$ models asynchrony in global types by allowing a sender role to proceed before the corresponding receiver has actually received the message. $\lfloor^{\mathrm{G}}|_{\textsc{Async-Branch}}\rceil$ does the same for branching. Observe that since we are allowing an asynchronous action from inside branch G_j to take place, we restrict the branching to the choice $l_j : G_j$ in order to disable the other branches. In $\lfloor^{\mathrm{G}}|_{\textsc{Swap}}\rceil$, the relation $\simeq_{\mathcal{G}}$ for global types models parallelism and is defined similarly to $\simeq_{\mathcal{C}}$; formally, it is the smallest congruence satisfying the rules in Figure 6. The rules are similar to the ones for $\simeq_{\mathcal{C}}$ where conditional is now replaced by branching.

Type checking. We now introduce our multiparty session typing, which checks that sessions in a program (choreography without restrictions) follow the protocol specifications given as global types. We use three kinds of typing environments:

$$\begin{array}{lll}
\textit{(Service Env)} & \Gamma ::= \Gamma,\; a\langle \tilde{\mathtt{p}} \,||\, \tilde{\mathtt{q}} \rangle : G \;\mid\; \Gamma,\; x@\tau : S \;\mid\; \emptyset & \\
 & \quad \mid \Gamma,\; X : (\Gamma, \Theta, \Delta) & \\
\textit{(Thread Env)} & \Theta ::= \Theta,\; \tau : k[\mathtt{p}] \mid \emptyset & \\
\textit{(Session Env)} & \Delta ::= \Delta,\; k : G \mid \emptyset &
\end{array}$$

A *service environment* Γ carries the global type of each public channel, specifying how a session has to be executed after initialisation. In $a\langle \tilde{\mathtt{p}} \,||\, \tilde{\mathtt{q}} \rangle$, $\tilde{\mathtt{p}}$ and $\tilde{\mathtt{q}}$ are the roles of the active and service

$$\lfloor^{\mathrm{G}}|_{\textsc{Com}}\rceil \quad \mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle U\rangle; G \xrightarrow{\ \mathtt{p}\ \texttt{->}\ \mathtt{q}:\langle U\rangle\ } G \qquad \lfloor^{\mathrm{G}}|_{\textsc{Branch}}\rceil \quad \mathtt{p}\ \texttt{->}\ \mathtt{q} : \{l_i : G_i\}_{i\in I\cup\{j\}} \xrightarrow{\ \mathtt{p}\ \texttt{->}\ \mathtt{q}:l_j\ } G_j$$

$$\lfloor^{\mathrm{G}}|_{\textsc{Rec}}\rceil \quad \frac{G[\mathsf{rec}\ \mathbf{t}; G/\mathbf{t}] \xrightarrow{\alpha} G'}{\mathsf{rec}\ \mathbf{t}; G \xrightarrow{\alpha} G'} \qquad \lfloor^{\mathrm{G}}|_{\textsc{Swap}}\rceil \quad \frac{G_1 \simeq_{\mathcal{G}} G_1' \xrightarrow{\alpha} G_2' \simeq_{\mathcal{G}} G_2}{G_1 \xrightarrow{\alpha} G_2}$$

$$\lfloor^{\mathrm{G}}|_{\textsc{Async-Com}}\rceil \quad \frac{G \xrightarrow{\alpha} G' \quad \mathtt{p} \in \mathsf{roles}(\alpha), \mathtt{q} \notin \mathsf{roles}(\alpha)}{\mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle U\rangle; G \xrightarrow{\alpha} \mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle U\rangle; G'} \qquad \lfloor^{\mathrm{G}}|_{\textsc{Async-Branch}}\rceil \quad \frac{G_j \xrightarrow{\alpha} G_j' \quad \mathtt{p} \in \mathsf{roles}(\alpha), \mathtt{q} \notin \mathsf{roles}(\alpha)}{\mathtt{p}\ \texttt{->}\ \mathtt{q} : \{l_i : G_i\}_{i\in I\cup\{j\}} \xrightarrow{\alpha} \mathtt{p}\ \texttt{->}\ \mathtt{q} : \{l_j : G_j'\}}$$

Figure 5. Global Types, semantics.

$$\lfloor^{\mathrm{Gs}}|_{\textsc{Com-Com}}\rceil \quad \frac{\{\mathtt{p},\mathtt{q}\} \cap \{\mathtt{p}',\mathtt{q}'\} = \emptyset}{\mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle U\rangle;\ \mathtt{p}'\ \texttt{->}\ \mathtt{q}' : \langle U'\rangle \quad \simeq_{\mathcal{G}} \quad \mathtt{p}'\ \texttt{->}\ \mathtt{q}' : \langle U'\rangle;\ \mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle U\rangle}$$

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

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

Figure 6. Global Types, swap relation $\simeq_{\mathcal{G}}$.

threads respectively. Γ also keeps the *sort* types of variables and all environments necessary for typing recursive procedures. A *thread environment* Θ keeps track of which role each thread is playing in a session. Finally, a *session environment* Δ records the type of each running session k. We assume that we can write $\Gamma, a : G$ only if a does not occur in Γ; 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 in multiple sessions playing different roles, but it cannot participate in the same session with more than one role.

Typing judgements have the shape $\Gamma;\ \Theta \vdash C \triangleright \Delta$. Intuitively, C is well-typed provided that public channels are used according to Γ, threads play roles according to Θ, and session channels are used according to Δ. Figure 7 contains the typing rules. Rule $\lfloor^{\mathrm{GT}}|_{\textsc{Start}}\rceil$ types a *(start)* term by checking that, in the subterm C, session k is used according to the type G of public channel a. Also, each thread τ_i is checked to play role $\mathtt{p}_i$ in C when using session k. We require all roles $\mathtt{p}_i$ to occur in G ($\mathsf{roles}(G)$), enforcing that each thread 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 their freshness. In $\lfloor^{\mathrm{GT}}|_{\textsc{Com}}\rceil$ we check that, given an interaction between τ_1 and τ_2 over session channel k, (i) the global type for k in the session environment requires a communication of type S between role $\mathtt{p}$ and role $\mathtt{q}$, (ii) τ_1 plays role $\mathtt{p}$ and τ_2 plays role $\mathtt{q}$ according to Θ, and (iii) expression e and variable x have type S according to Γ. Rule $\lfloor^{\mathrm{GT}}|_{\textsc{Sel}}\rceil$ deals with selection and is similar to $\lfloor^{\mathrm{GT}}|_{\textsc{Com}}\rceil$, although we now check that the chosen label is among the ones allowed by the type. $\lfloor^{\mathrm{GT}}|_{\textsc{Del}}\rceil$ addresses session delegation: it transfers the ownership of role $\mathtt{p}'$ in session k' and checks that the carrying type in the session type of k is the type of the continuation of k'. $\lfloor^{\mathrm{GT}}|_{\textsc{Rec}}\rceil$ types a recursive procedure and the choreography in which it is used. The recursion body is checked using only the types of its parameters (stored in Γ), public channels (Γ_{srv}), and giving the possibility to invoke other procedures (Γ_{rec}). All other rules are standard.

For instance, the protocol in Example 1 types channel a in Example 2.

Runtime typing. For showing that *well-typed programs never go wrong*, we need to extend our typing to runtime choreographies. In particular, we need to deal with two issues: asynchronous delegations and parallelism. Hereby, we give an intuition of these issues and how we deal with them (see [14] for details).

Asynchronous Delegation. We check runtime choreographies with new judgements of the form $\Gamma; \Theta \vdash_{\Sigma} C \triangleright \Delta$. The extra environment Σ, the *delegation environment*, contains information about channels that may have been already delegated by rule $\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil$. E.g., consider the following choreography:

$$C = \underbrace{\tau_1[\mathtt{p}].e\ \texttt{->}\ \tau_2[\mathtt{q}].x : k;}_{\eta_1}\ \underbrace{\tau_1[\mathtt{p}']\ \texttt{->}\ \tau_3[\mathtt{q}'] : k'\langle\!\langle k[\mathtt{p}]\rangle\!\rangle;}_{\eta_2}\ \underbrace{\tau_3[\mathtt{p}].e'\ \texttt{->}\ \tau_4[\mathtt{r}].y : k}_{\eta_3}$$

By $\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil$, C may execute η_2 before η_1 and reduce to $\eta_1;\eta_3$. When typing $\eta_1;\eta_3$, we need to remember the delegation η_2 in Σ, since we cannot construct a thread environment Θ for typing both η_1 and η_3. Therefore, the choreography $\eta_1;\eta_3$ would be well-typed with $\Sigma = \eta_2$ but not with $\Sigma = \emptyset$. Parallelism can actually make things worse, e.g., $\eta_3;\eta_1$, a swap of $\eta_1;\eta_3$, is clearly still safe at endpoint, since the output in η_1 has already been executed and thus τ_1 will no longer use k. Our type system uses Σ also to gracefully handle these cases. To exemplify how we formalise this mechanism, we show the runtime typing version of rule $\lfloor^{\mathrm{GT}}|_{\textsc{Com}}\rceil$:

$$\frac{\begin{array}{c}(\Theta \vdash \tau_1 : k[\mathtt{p}] \ \vee\ \mathsf{delegates}(\Sigma, \tau_1, k[\mathtt{p}]))\\ \Theta \vdash \tau_2 : k[\mathtt{q}] \qquad \neg\mathsf{delegated}(\Sigma, k[\mathtt{q}])\\ \Gamma, x@\tau_2 : S;\ \Theta \vdash_{\Sigma} C \triangleright k : G, \Delta \qquad \Gamma \vdash e@\tau_1 : S\end{array}}{\Gamma;\ \Theta \vdash_{\Sigma} \tau_1[\mathtt{p}].e\ \texttt{->}\ \tau_2[\mathtt{q}].x : k; C \triangleright k : \mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle S\rangle; G, \Delta}$$

The rule above differs from $\lfloor^{\mathrm{GT}}|_{\textsc{Com}}\rceil$ in Figure 7 only in the role checks for the two interacting threads, given in the first two lines. In the first line, we check that thread τ_1 plays role $\mathtt{p}$ in session k, using Θ, *or* that τ_1 has asynchronously delegated $\mathtt{p}$ to another thread, using Σ through the auxiliary predicate $\mathsf{delegates}$. This covers our example above, where C reduces to $\eta_1;\eta_3$. In the second line, we check that τ_2 plays role $\mathtt{q}$, as before in rule $\lfloor^{\mathrm{GT}}|_{\textsc{Com}}\rceil$ for programs. However, now we also need to check that τ_2 has not asynchronously delegated its role to another thread, using another auxiliary predicate $\mathsf{delegated}$ on Σ. This condition is necessary for guaranteeing that $\mathtt{q}$ is played only by one thread, in order to avoid races on the receiving of messages for $\mathtt{q}$ in session k.

Parallelism. Consider the following protocol:

$$G = \mathtt{p}_1\ \texttt{->}\ \mathtt{p}_2 : \langle\mathbf{int}\rangle; \quad \mathtt{p}_3\ \texttt{->}\ \mathtt{p}_4 : \langle\mathbf{string}\rangle$$

$$\lfloor^{\text{GT}}|_{\text{START}}\rceil \quad \frac{\Gamma \vdash a\langle \mathtt{p}_1, \ldots, \mathtt{p}_n \mid\mid \mathtt{p}_{n+1}, \ldots, \mathtt{p}_m\rangle : G \qquad \{\mathtt{p}_1, \ldots, \mathtt{p}_m\} = \mathsf{roles}(G) \quad \Gamma;\ \Theta, \tau_1 : k[\mathtt{p}_1], \ldots, \tau_m : k[\mathtt{p}_m] \vdash C \triangleright \Delta, k : G \qquad \tau_{n+1}, \ldots, \tau_m \notin \Theta}{\Gamma;\ \Theta \vdash \tau_1[\mathtt{p}_1], ..., \tau_n[\mathtt{p}_n]\ \mathbf{start}\ \tau_{n+1}[\mathtt{p}_{n+1}], ..., \tau_m[\mathtt{p}_m] : a(k); C \triangleright \Delta} \qquad \lfloor^{\text{GT}}|_{\text{VAR}}\rceil \quad \frac{\Gamma = \Gamma_{\mathsf{srv}}, \Gamma' \qquad \Delta' \text{ end only}}{\Gamma, X : (\Gamma', \Theta, \Delta);\ \Theta \vdash X(\widetilde{x@\tau}, \tilde{k}, \tilde{\tau}) \triangleright \Delta, \Delta'}$$

$$\lfloor^{\text{GT}}|_{\text{COM}}\rceil \quad \frac{\Gamma \vdash e@\tau_1 : S \qquad \Theta \vdash \tau_1 : k[\mathtt{p}],\ \tau_2 : k[\mathtt{q}] \qquad \Gamma,\ x@\tau_2 : S;\ \Theta \vdash C \triangleright k : G, \Delta}{\Gamma;\ \Theta \vdash \tau_1[\mathtt{p}].e\ \texttt{->}\ \tau_2[\mathtt{q}].x : k; C \triangleright k : \mathtt{p}\ \texttt{->}\ \mathtt{q} : \langle S\rangle; G,\ \Delta} \qquad \lfloor^{\text{GT}}|_{\text{ZERO}}\rceil \quad \frac{\Delta \text{ end only}}{\Gamma;\ \Theta \vdash \mathbf{0} \triangleright \Delta}$$

$$\lfloor^{\text{GT}}|_{\text{SEL}}\rceil \quad \frac{\Theta \vdash \tau_1 : k[\mathtt{p}],\ \tau_2 : k[\mathtt{q}] \qquad \Gamma;\ \Theta \vdash C \triangleright k : G_j,\ \Delta \qquad j \in I}{\Gamma;\ \Theta \vdash \tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}] : k[l_j]; C \triangleright k : \mathtt{p}\ \texttt{->}\ \mathtt{q} : \{l_i : G_i\}_{i \in I}} \qquad \lfloor^{\text{GT}}|_{\text{IF}}\rceil \quad \frac{\Gamma \vdash e@\tau : \mathbf{bool} \qquad \Gamma;\ \Theta \vdash C_1 \triangleright \Delta \qquad \Gamma;\ \Theta \vdash C_2 \triangleright \Delta}{\Gamma;\ \Theta \vdash \mathsf{if}\, e@\tau\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2 \triangleright \Delta}$$

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

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

Figure 7. Global Calculus with Multiparty Sessions, typing rules.

which can correctly type public channel a in the choreography:

$$\begin{array}{rcl} C & = & \tau_1[\mathtt{p}_1], \tau_2[\mathtt{p}_2]\ \mathbf{start}\ \tau_3[\mathtt{p}_3], \tau_4[\mathtt{p}_4] : a(k); \\ & & \tau_1[\mathtt{p}_1].e\ \texttt{->}\ \tau_2[\mathtt{p}_2].x : k; \quad \tau_3[\mathtt{p}_3].e'\ \texttt{->}\ \tau_4[\mathtt{p}_4].y : k \end{array}$$

Since $\{\tau_1, \tau_2\} \cap \{\tau_3, \tau_4\} = \emptyset$, we can swap C with C' such that:

$$\begin{array}{rcl} C' & = & \tau_1[\mathtt{p}_1], \tau_2[\mathtt{p}_2]\ \mathbf{start}\ \tau_3[\mathtt{p}_3], \tau_4[\mathtt{p}_4] : a(k); \\ & & \tau_3[\mathtt{p}_3].e'\ \texttt{->}\ \tau_4[\mathtt{p}_4].y : k; \quad \tau_1[\mathtt{p}_1].e\ \texttt{->}\ \tau_2[\mathtt{p}_2].x : k \end{array}$$

Public channel a in C' does not have type G anymore, but C' is clearly still correct since we can easily follow its swap from C in type G using swapping for global types:

$$G \quad \simeq_{\mathcal{G}} \quad \mathtt{p}_3\ \texttt{->}\ \mathtt{q}_4 : \langle \mathbf{string}\rangle;\ \mathtt{p}_1\ \texttt{->}\ \mathtt{p}_2 : \langle \mathbf{int}\rangle$$

Our type system, however, does not deal with swappings in global types, and would reject C'. We made this choice so that programmers do not need to think about the swap relations when writing programs, which could make error messages confusing in some cases. However, at runtime, we must consider swaps in order to preserve well-typedness wrt reductions (Subject Reduction). Therefore, our runtime type system augments rules $\lfloor^{\text{GT}}|_{\text{START}}\rceil$ and $\lfloor^{\text{GT}}|_{\text{DEL}}\rceil$ for typing up to $\simeq_{\mathcal{G}}$.

Properties. We can now present the expected main properties of our type system. 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 $k : G'$. Also, we write $\Delta \simeq_{\mathcal{G}} \Delta'$ iff $\mathsf{dom}(\Delta) = \mathsf{dom}(\Delta')$ and $\Delta(k) \simeq_{\mathcal{G}} \Delta'(k)$ for all $k \in \mathsf{dom}(\Delta)$.

Theorem 2. *Assume* $\Gamma;\ \Theta \vdash_\Sigma C \triangleright \Delta$. *Then,*

- *(Subject Swap)* $C \simeq_{\mathcal{C}} C'$ *implies* $\Gamma;\ \Theta \vdash_\Sigma C' \triangleright \Delta'$ *where* $\Delta \simeq_{\mathcal{G}} \Delta'$.
- $C \xrightarrow{\lambda} C'$ *implies that there exists* Δ' *such that*
  - *(Subject Reduction)* $\Gamma;\ \Theta' \vdash_{\Sigma'} C' \triangleright \Delta'$ *for some* Θ', Σ'.
  - *(Session Fidelity) if* λ *is a communication on session* k *then* $\Delta \xrightarrow{k:\alpha} \Delta'$ *with* α *and* λ *on the same roles; else,* $\Delta = \Delta'$.

Type inference. We exploit the close correspondence between protocols and choreographies to perform *type inference* of public channels. Thus, we can automatically extract protocols from choreographies. First, we define *subtyping* as set inclusion on branching labels, similarly to the covariant typing of rule $\lfloor^{\text{GT}}|_{\text{SEL}}\rceil$ in Figure 7. Then, we modify our rules to determine the *principal type* of a choreography. Hereby, we discuss this aspect informally (cf. [14] for the formalisation). We change rule $\lfloor^{\text{GT}}|_{\text{SEL}}\rceil$ to require a singleton branching type for the label of interest. Then, $\lfloor^{\text{GT}}|_{\text{IF}}\rceil$ will have to compute the least upper bound (lub) of the session environments Δ_1 and Δ_2 (for C_1 and C_2 in the branches), by *merging* their branching types. Similarly, rule $\lfloor^{\text{GT}}|_{\text{START}}\rceil$ will need to update service types in Γ with the lub of all the global types of each session started through the same public channel a. Recursion is handled in a standard manner [34].

For example, from b in Example 3, we can infer the type: $\mathtt{B}\ \texttt{->}\ \mathtt{H} : \langle \mathbf{int}\rangle; \mathtt{B}\ \texttt{->}\ \mathtt{H} : \{\mathit{done} : \mathsf{end},\ \mathit{del} : \mathtt{B}\ \texttt{->}\ \mathtt{H} : \langle \mathbf{int}\rangle; \mathtt{B}\ \texttt{->}\ \mathtt{H} : \langle (\ldots \text{as Line 2 in Example 1} \ldots)@\mathtt{B2}\rangle\}$.

5. Endpoint Projection and its Properties

We now address endpoint code generation. First, we recall an endpoint model that we shall use as a target language. Then, we show how to generate endpoint code for each thread in a choreography and, finally, how to obtain the code for the entire system. Our code generation, the *EndPoint Projection (EPP)*, will satisfy the *EPP Theorem*, which gives a correspondence between the asynchronous semantics of choreographies and the one of endpoint terms.

Endpoint model. We model endpoint code with the calculus for multiparty sessions [7], whose syntax includes conditional, parallel, the inactive process and recursion plus the following terms, where unboxed terms denote programs:

$$\begin{array}{lll} P, Q & ::= & \ldots \\ & \mid & \overline{a}[\mathtt{p}_1, \ldots, \mathtt{p}_m](k); P \quad \mid a[\mathtt{p}](k); P \quad \mid\ !a[\mathtt{p}](k); P \\ & \mid & c?\mathtt{p}(x); P \quad \mid c?\mathtt{p}((k')); P \quad \mid c?\mathtt{p}\,\&\,\{l_i : P_i\}_{i \in I} \\ & \mid & c!\mathtt{p}\langle e\rangle; P \quad \mid c!\mathtt{p}\langle\!\langle c'\rangle\!\rangle; P \quad \mid c!\mathtt{p} \oplus l; P \\ & \mid & \boxed{k : h} \quad \mid \boxed{(\boldsymbol{\nu} k)\ P} \\ c & ::= & k \mid \boxed{k[\mathtt{q}]} \qquad \boxed{h} ::= \boxed{(\mathtt{p}, \mathtt{q}, \boldsymbol{w}) \cdot h} \mid \boxed{\varnothing} \\ \boxed{\boldsymbol{w}} & ::= & \boxed{v} \mid \boxed{l} \mid \boxed{k[\mathtt{p}]} \end{array}$$

The first row contains the terms for implementing a session **start**: request, accept and replicated accept respectively. Request and accept are used by active threads, while the latter models a process for spawning service threads. The second row concerns in-session inputs of a value, channel (delegation), or label (branching). For $c = k$, $k?\mathtt{p}$ is an input from role $\mathtt{p}$ over session k. Dually, the third row has outputs, where $\mathtt{p}$ in $k!\mathtt{p}$ is the role the message is sent to.

Example 4. $P_{\mathtt{s}} \mid P_{\mathtt{b}_1} \mid P_{\mathtt{b}_2}$ *is an endpoint implementation of Example 2 where:*

$$P_{\mathtt{s}} = \; !a[\mathtt{S}](k);\; k?\mathtt{B1}(x_1);\; k!\mathtt{B1}\langle \mathsf{quote}(x_1)\rangle;\; k!\mathtt{B2}\langle \mathsf{quote}(x_1)\rangle;\; k?\mathtt{B2}\,\&\left\{\begin{array}{ll} ok: & k?\mathtt{B2}(x_2);\\ & k!\mathtt{B2}\langle ddate\rangle,\end{array}\right\}\; quit : \mathbf{0}$$

$$P_{\mathtt{b}_1} = \; \overline{a}[\mathtt{B1},\mathtt{B2},\mathtt{S}](k);\; k!\mathtt{S}\langle book\rangle;\; k?\mathtt{S}(y_1);\; k!\mathtt{B2}\langle \mathsf{contrib}(y_1)\rangle$$

$$P_{\mathtt{b}_2} = \; a[\mathtt{B2}](k);\; k?\mathtt{S}(z_1);\; k?\mathtt{B1}(z_2);\; \mathsf{if}\ (z_1 - z_2 \leq 100)\ \mathsf{then}\ k!\mathtt{S} \oplus ok;\; k!\mathtt{S}\langle addr\rangle;\; k?\mathtt{S}(z_3)\ \mathsf{else}\ k!\mathtt{S} \oplus quit$$

□

Boxed terms are used only at runtime. A session queue $k : h$ is a FIFO queue h for session channel k. A message in h contains the sender and receiver roles, and the carried message $\boldsymbol{w}$ (a value v, a label l or a delegated channel $k[\mathtt{p}]$). Messages with different pairs of roles can be permuted by structural congruence (omitted), simulating a queue per each pair of roles [7]. Since messages in a queue have both a receiver and a sender role, in-session inputs and outputs are annotated with the executing role at runtime. E.g., the term $k[\mathtt{q}]!\mathtt{p} \oplus l; P$ is an executing process which will put the message $(\mathtt{q}, \mathtt{p}, l)$ in the queue k with sender $\mathtt{q}$ and receiver $\mathtt{p}$. Figure 8 contains a selection of the rules defining the semantics of terms, where $R = \prod_{j\in J} !a[\mathtt{p}_j](k_j); Q_j$, $I = \{1,\ldots,n\}$ and $J = \{n+1,\ldots,m\}$. The first rule initiates a session, creating an empty session queue $k : \varnothing$ and substituting 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. Replicated services in R model spawning of new processes. The second rule puts value v, the evaluation of e, in the queue for k. The last rule is about branching: it fetches a label l_j from the queue and then continues as process P_j. Labels, denoted by μ, annotate reductions to make their actions observable. E.g., label $!\mathtt{p}\ \texttt{->}\ \mathtt{q} : k\langle v\rangle$ denotes a communication from role $\mathtt{p}$ to role $\mathtt{q}$ on session channel k carrying v. Similarly, the label for branching starts with a ?, denoting input.

Thread projection. We denote with $C \upharpoonright_\tau$ the projection of the behaviour of a thread in a choreography onto an endpoint term. A selection of the rules defining $C \upharpoonright_\tau$ is reported in Figure 9. Thread projection adds no further ad-hoc communications wrt the originating choreography. In a start, we project the first thread to an endpoint request, threads $\tau_2,\ldots,\tau_n$ to accepts and threads $\tau_{n+1},\ldots,\tau_m$ to replicated accepts. For selection, the sender is projected to an output and the receiver to a branching. Projections of communication and delegation (omitted) follow the same principle. In a choreography conditional, τ is projected to a local conditional, whereas for all other threads we require their projected behaviours to be merged by the *merging* partial operator $\sqcup$ [15]. $P \sqcup Q$ is isomorphic to P and Q up to branching, where all branches of P or Q with distinct labels are also included. As a full example, the thread projections of the choreography in Example 2 (Two-buyer choreography) are the processes reported in Example 4.

Linearity. The expressivity of the *(start)* primitive may introduce races on public channels. For example, the choreography

$$\tau_1[\mathtt{p}], \tau_2[\mathtt{q}]\ \mathbf{start}\ : a(k); \quad \tau_3[\mathtt{p}], \tau_4[\mathtt{q}]\ \mathbf{start}\ : a(k') \qquad (1)$$

features four threads starting two different sessions on the same public channel. If we run their projections 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 choreography.

In the sequel an *interaction node*, denoted by n, is an abstraction of a node in a choreography syntax tree. n can either be $\tau_1,\ldots,\tau_n\ \mathbf{start}\ \tau_{n+1},\ldots,\tau_m : a$ (abstracting a *(start)* node) or $\tau_1\ \texttt{->}\ \tau_2$ (abstracting a session interaction). We write $\mathsf{n}_1 \prec \mathsf{n}_2 \in C$ whenever n_1 precedes n_2 in the choreography C.

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

$\mathsf{n}_1 \prec_\tau \mathsf{n}_2 \in C$ implies that the projection of τ for the originating node of n_2 will not be enabled before that for n_1. We use dependencies to define *linearity*:

Definition 2 (Linearity). *If* $\mathsf{n}_i = \tau^i_1,\ldots,\tau^i_n\ \mathbf{start}\ \tau^i_{n+1},\ldots,\tau^i_m : a$ *($i = 1, 2$, $n \geq 2$) 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 checks that, for all start nodes $\mathsf{n}_1 \prec \mathsf{n}_2 \in C$ on the same a, each active thread in n_2 depends on some thread in n_1, avoiding races between active threads. This is not necessary for service threads, since they will be merged by our EPP. Linearity is preserved by our semantics and is decidable, since a choreography is linear whenever its one-time unfolding of recursions is linear.

Example 5. *In (1), we cannot build any dependency unless, e.g., $\tau_1 = \tau_3$ and $\tau_2 = \tau_4$. Instead, the following choreography is linear with dependencies between τ_1 and τ_3 and between τ_2 and τ_4.*

$$\tau_1[\mathtt{p}], \tau_2[\mathtt{q}]\ \mathbf{start}\ : a(k);\ \tau_1[\mathtt{p}_1]\ \texttt{->}\ \tau_3[\mathtt{p}_3] : k';\ \tau_2[\mathtt{p}_2]\ \texttt{->}\ \tau'_2[\mathtt{p}_4] : k';\ \tau'_2[\mathtt{p}_5]\ \texttt{->}\ \tau_4[\mathtt{p}_6] : k';\ \tau_3[\mathtt{p}], \tau_4[\mathtt{q}]\ \mathbf{start}\ : a(k)$$ □

EPP. Since different service threads may be started on the same public channel and play the same role, we use $\sqcup$ for merging their behaviours into a single replicated process. We identify such threads with $\lfloor C \rfloor^a_{\mathtt{p}}$, the *service grouping operator* [14]. We can finally give the complete definition of EPP:

Definition 3 (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:*

$$C\!\upharpoonright = \; (\boldsymbol{\nu}\tilde{k})\Big(\underbrace{\prod_{\tau\,\in\,\mathsf{fn}(C_f)} C_f\!\upharpoonright_\tau}_{\text{(i)}} \;\mid\; \underbrace{\prod_{k\,\in\,\mathsf{fn}(C_f)} k : \varnothing}_{\text{(ii)}}\Big) \mid \underbrace{\prod_{a,\mathtt{p}}\Big(\bigsqcup_{\tau\in\lfloor C_f\rfloor^a_{\mathtt{p}}} C_f\!\upharpoonright_\tau\Big)}_{\text{(iii)}}$$

The EPP of C is the parallel composition of (i) the projections of all active threads; (ii) the queues for all active sessions; and (iii) the replicated processes obtained by merging the projections of all service threads with same public channel and role. Note how swapping has no influence on EPP, since we now have parallel composition at the endpoint level, i.e., if $C \simeq_{\mathcal{C}} C'$ then $C\!\upharpoonright \equiv C'\!\upharpoonright$.

Example 6. *Let C be the two-buyer-helper choreography from Example 3. Since h_1 and h_2 are grouped under $\lfloor C \rfloor^b_{\mathtt{H}}$, the EPP of C will merge their behaviour into a single process (say, P_{h}). I.e., $C\!\upharpoonright = C\!\upharpoonright_{\mathtt{b}_1} \mid C\!\upharpoonright_{\mathtt{b}_2} \mid C\!\upharpoonright_{\mathtt{s}} \mid P_{\mathsf{h}}$ where P_{h} is:*

$$!b[\mathtt{H}](k');\ k'?\mathtt{B}(z);\ k'?\mathtt{B}\,\&\left\{\begin{array}{ll} done: & \mathbf{0},\\ del: & k'?\mathtt{B}(z');\ k'?\mathtt{B}(\!(k)\!);\\ & \mathsf{if}\ (z - z' \leq 100)\\ & \mathsf{then}\ k!\mathtt{S} \oplus ok; k!\mathtt{S}\langle addr\rangle; k?\mathtt{S}(z'')\\ & \mathsf{else}\ k!\mathtt{S} \oplus quit\end{array}\right\}$$

□

$$\lfloor^{\text{P}}|_{\text{START}}\rceil \quad \begin{array}{l} \overline{a}[\mathtt{p}_1,\ldots,\mathtt{p}_m](k);P \mid \prod_{i\in I} a[\mathtt{p}_i](k_i);P_i \mid R \xrightarrow{\mathtt{p}_1,\ldots,\mathtt{p}_n \ \mathbf{start}\ \mathtt{p}_{n+1},\ldots,\mathtt{p}_m:a(k)} \\ \quad (\boldsymbol{\nu} k)\left(P[k[\mathtt{p}_1]/k] \mid \prod_{i\in I} P_i[k[\mathtt{p}_i]/k_i] \mid \prod_{j\in J} Q_j[k[\mathtt{p}_j]/k_j] \mid k:\varnothing\right) \mid R \end{array}$$

$$\lfloor^{\text{EP}}|_{\text{SEND}}\rceil \quad k[\mathtt{p}]!\mathtt{q}\langle e\rangle;P \mid k:h \xrightarrow{\mathtt{p}\ \texttt{->}\ \mathtt{q}:k\langle v\rangle} P \mid k:h\cdot(\mathtt{p},\mathtt{q},v) \quad (e\downarrow v)$$

$$\lfloor^{\text{EP}}|_{\text{BRANCH}}\rceil \quad k[\mathtt{q}]?\mathtt{p}\,\&\,\{l_i:P_i\}_{i\in I} \mid k:(\mathtt{q},\mathtt{p},l_j)\cdot h \xrightarrow{\mathtt{p}\ \texttt{->}\ \mathtt{q}:k[l_j]} P_j \mid k:h \quad (j\in I)$$

Figure 8. Endpoint calculus, selected reduction rules.

$$\left(\begin{array}{l}\tau_1[\mathtt{p}_1],\ldots,\tau_n[\mathtt{p}_n]\ \mathbf{start} \\ \quad \tau_{n+1}[\mathtt{p}_{n+1}],\ldots,\tau_m[\mathtt{p}_m]:a(k);C\end{array}\right)\lfloor_{\tau_i} = \begin{cases}\overline{a}[\mathtt{p}_1,\ldots,\mathtt{p}_m](k);(C\lfloor_{\tau_i}) & \text{if } i=1\\ a[\mathtt{p}_i](k);(C\lfloor_{\tau_i}) & \text{if } 2\leq i\leq n\\ !a[\mathtt{p}_i](k);(C\lfloor_{\tau_i}) & \text{if } n+1\leq i\leq m\\ C\lfloor_{\tau_i} & \text{otherwise}\end{cases}$$

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

$$(\ \mathsf{if}\ e@\tau\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ )\lfloor_{\tau'} = \begin{cases}\mathsf{if}\ e\ \mathsf{then}\ (C_1\lfloor_{\tau'})\ \mathsf{else}\ (C_2\lfloor_{\tau'}) & \text{if } \tau=\tau'\\ (C_1\lfloor_{\tau'})\sqcup(C_2\lfloor_{\tau'}) & \text{otherwise}\end{cases}$$

Figure 9. Thread projection, selected rules.

EPP Theorem. We now present our *EPP Theorem*, which formalises the relationship between the semantics of a well-typed, linear choreography and the semantics of its EPP. Without loss of generality, we consider only *strict reductions*, denoted by $\rightsquigarrow$, i.e. reductions where restricted names not under a prefix are never renamed. $\rightsquigarrow^*$ denotes the closure of $\rightsquigarrow$. The entailment $\tilde{\lambda}\vdash\tilde{\mu}$ checks that the endpoint actions $\tilde{\mu}$ *implement* the global actions $\tilde{\lambda}$ (cf. [14]).

Theorem 3 (EPP)**.** *Let $C\equiv(\boldsymbol{\nu}\tilde{\tau}\tilde{k})\ C_f$ be linear and well-typed, with C_f restriction-free. 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) $C'\lfloor\prec P'$.*

Above, the *pruning relation* $P\prec Q$ is such that $P\sim Q$ ($\sim$ is bisimilarity) and that P has some unused branches and replicated accepts. Point 1. states that an EPP can mimic (up to pruning) all the reductions of its originating choreography; on the other hand, point 2. says that an EPP always eventually reduces (up to pruning) to the projection of a (possibly reached after multiple reductions) reductum of its originating choreography. Both points ensure that the observables of a choreography and its EPP are correctly related.

By Theorems 1 and 3, we can formalise our *deadlock-freedom-by-design* property. Below, $\xrightarrow{\tilde{\mu}}{}^*$ is the closure of $\xrightarrow{\mu}$.

Corollary 1 (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 $\mathbf{0}\prec P$.*

Moreover, our EPP code enjoys standard communication safety:

Corollary 2 (Safety)**.** *Let C be linear, well-typed and $C\lfloor\xrightarrow{\tilde{\mu}}{}^* P$. Then,*

1. *(Linearity) P has no races on any a or k with same role $\mathtt{p}$;*
2. *(Error-freedom) if P has an enabled input $k[\mathtt{p}]?\mathtt{q}$ that can consume a message $(\mathtt{p},\mathtt{q},\boldsymbol{w})$ from a session queue $k:h$ in P, then $\boldsymbol{w}$ is of the same type of the input (value, label, or delegated channel).*

Typing expressiveness. Previously proposed typing disciplines for session types ensure properties similar to ours, but performing type analysis directly on endpoint programs [7, 15, 22]. Our typing discipline subsumes a larger class of safe deadlock-free systems, by exploiting the extra information that we gain from defining implementations with choreographies. In particular, our typing system allows for two novel features wrt standard multiparty session typing: inter-protocol coherence and partial protocol implementation.

Let us discuss inter-protocol coherence. Consider the protocol:

$$G \;=\; \mathtt{p}\ \texttt{->}\ \mathtt{q}:\{l_1:\mathtt{r}\ \texttt{->}\ \mathtt{p}:\langle\mathbf{int}\rangle,\ l_2:\mathtt{r}\ \texttt{->}\ \mathtt{q}:\langle\mathbf{int}\rangle\}$$

Above, $\mathtt{p}$ communicates to $\mathtt{q}$ a choice between labels l_1 and l_2. In the first case another role, $\mathtt{r}$, is expected to communicate an integer to $\mathtt{p}$. Otherwise, $\mathtt{r}$ will communicate an integer to $\mathtt{q}$. A potential use case for G could be that $\mathtt{r}$ possesses some good, and $\mathtt{p}$ decides where the good should be sent to ($\mathtt{p}$ itself or $\mathtt{q}$). Previous work on global types cannot type any system implementing G, since G cannot be projected onto a correct set of endpoint types [38]. Indeed, in the protocol, $\mathtt{r}$ is not informed of the choice made by $\mathtt{p}$ and thus cannot know whether it should communicate with $\mathtt{p}$ or $\mathtt{q}$ afterwards. We refer to this problem by saying that G is not *coherent* for previous type systems based on global types. In our framework, we do not consider protocol coherence because protocols such as G above can easily be implemented by interleaving them with other ones. For example, consider the following choreography:

1. $\tau_1[\mathtt{p}]\ \mathbf{start}\ \tau_2[\mathtt{q}],\tau_3[\mathtt{r}]:a(k);\quad \tau_2[\mathtt{p}'],\tau_3[\mathtt{q}']\ \mathbf{start}\ :b(k');$
2. $\mathsf{if}\ e@\tau_1\ \mathsf{then}\quad \tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}]:k[l_1];\ \tau_2[\mathtt{p}']\ \texttt{->}\ \tau_3[\mathtt{q}']:k'[l_1];$
3. $\qquad\qquad\qquad\quad \tau_3[\mathtt{r}].\mathit{some\_int}\ \texttt{->}\ \tau_1[\mathtt{p}].x:k$
4. $\qquad\qquad \mathsf{else}\quad \tau_1[\mathtt{p}]\ \texttt{->}\ \tau_2[\mathtt{q}]:k[l_2];\ \tau_2[\mathtt{p}']\ \texttt{->}\ \tau_3[\mathtt{q}']:k'[l_2];$
5. $\qquad\qquad\qquad\quad \tau_3[\mathtt{r}].\mathit{some\_int}\ \texttt{->}\ \tau_2[\mathtt{q}].y:k$

The choreography above can be typed correctly using G as type for a (we omit the typing for b). In order to notify τ_3, playing role $\mathtt{r}$, of the choice performed by τ_1, playing role $\mathtt{p}$, we make use of an additional session between τ_2 and τ_3. We use this session, k', after τ_2 receives the choice from τ_1. Observe that the choreography is typable and can be correctly projected by our EPP. The key aspect of this example is that our framework leaves the task of defining a coherent system to the implementation (the choreography). Hence,

protocols can be designed at a higher level of abstraction. E.g., in G we do not specify how $\mathtt{r}$ is notified of the choice. We call this aspect *inter-protocol coherence*, since it is the composition of protocols in a choreography that is checked for coherence (by checking whether its EPP is defined), and not each protocol by itself.

We now discuss partial protocol implementation with the following choreography:

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

The choreography above is typable in our system (a's type is $\mathtt{p}\ \mathtt{->}\ \mathtt{q} : \{l_1 : \mathsf{end},\ l_2 : \mathsf{end},\ l_3 : \mathsf{end}\}$). However, the endpoint projection for τ' would not be typable with standard contra-variant input typing, which requires that *at least* all the branches in the type are implemented. Again, this is a consequence of using choreographies: since in the choreography we know exactly which outputs will correspond to which inputs, we can ensure that the protocol branches that τ' does not implement will never be used.

6. Implementation

Following our model, we have implemented the Chor programming language [16]. Our implementation is open source and comes with a complete IDE, developed as an Eclipse [19] plugin.

Chor features all GC primitives (Figure 1), an implementation of our typing discipline (Figure 7), and an EPP implementation for projecting choreographies onto executable endpoints (from § 5). Chor also provides some syntax extensions for enhanced usability, e.g., protocols may refer to other protocols and choreographies allow local code for local state manipulation and user interactions.

We give an overview of the development methodology offered by our framework, and detail some aspects of our implementation.

Development methodology. In the development methodology suggested by Chor (see Figure 10), developers can first use our IDE to write protocol specifications and choreographies. The programmer is supported by on-the-fly verification which takes care of checking (i) the syntactic correctness of program terms and (ii) the type compliance of the choreography wrt the protocol specifications, using our typing discipline. Program errors are reported using syntax highlighting, allowing for an interactive programming experience.

Once the global program is completed, developers can automatically project to an endpoint implementation, given in the Jolie programming language (cf. Executable artifacts). Nevertheless, Chor is designed to be extended to multiple endpoint languages: potentially, each thread in a choreography could be implemented with a different endpoint technology. We plan future extensions to support projecting endpoints to e.g., Java, C#, or WS-BPEL [30].

Each Jolie endpoint program comes with its own deployment information, given as a term separated from the code implementing the behaviour of the projected thread. This part can be optionally customised by the programmer, which can be useful for running the programs on some specific network or communication technology.

Finally, the Jolie endpoint programs can be executed. As expected, they will implement the originating choreography.

Executable artifacts. Our EPP implementation targets Jolie, an open-source service-oriented language [3, 29]. Choosing Jolie has several reasons: (i) Jolie offers constructs similar to those of our endpoint calculus, e.g., communications and input choices, making part of our EPP straightforward; (ii) Jolie has a formal semantics and a reference implementation, which we used for implementing some abstract aspects of our model such as message queues and session channels; and (iii) Jolie supports a wide range of compatibility with other technologies as detailed below.

Deployment. By default, our endpoint programs will operate on top of TCP/IP sockets. However, since Jolie also supports other communication technologies – e.g. local memory IPC and Bluetooth – and data formats – e.g. HTTP and SOAP [4] – programmers may customise deployment information of each endpoint. Hence, some endpoints may communicate over, e.g., HTTP, while others, e.g., using fast binary data formats. Additionally, different endpoints may be deployed in different machines and/or networks.

Notably, customising the deployment of an endpoint program does not necessarily require updating the code of the others. Supporting this flexibility has required a careful implementation of session starts (rule $\lfloor {}^{\mathrm{EP}}|_{\mathrm{START}} \rceil$), which are coordinated by special "start services". The (endpoint projections of the) active threads willing to start a session contact the appropriate start service. Then, the start service spawns the (projections of the) service threads by calling the external services that implement them. In every message exchange, each endpoint informs the start service of the *binding information* (e.g., IP address and data format) on which the endpoint can be reached. Finally, the start service informs all participants about all necessary bindings, so that each party can dynamically update its references to the others (e.g., socket connections).

Another key feature concerning different communication technologies is that our queue implementation is based on the latest *correlation sets* implementation in Jolie [28]. Correlation sets allow to program incoming message queueing by correlating some data values inside messages with those inside of a thread local state. Our EPP handles this programming to implement a separate queue for each role in a session as required by our model. Afterwards, the programmer can customise correlation for each deployment artifact. For instance, some threads may identify sessions using HTTP cookies (as in common web applications), while others may use SOAP headers (as in the WS-Addressing specifications [5]).

Delegation. Session delegation is a nontrivial mechanism at the level of endpoint implementation. The main concern lies in updating channel references (bindings). For instance, assume that a session k has some thread participants, say τ and $\tilde{\tau}$. Suppose now that τ delegates its role on k to another thread τ' through a different session. In such a situation, all the threads in $\tilde{\tau}$ need their external references to be updated for reaching τ' instead of τ when communicating with the session/role pair delegated by τ. In our endpoint model this necessity is completely abstracted away by the synchronisations on the centralised message queues (one per session). However, in our implementation, message queues are completely distributed: each thread owns a message queue which must be reached explicitly by other threads. [24] presents a survey of possible solutions to this problem in asynchronous scenarios. The main challenge is that $\tilde{\tau}$ may send messages to τ before getting notified of the delegation, and τ must thus resend these messages to τ', adding extra communications. In our EPP implementation, each thread in $\tilde{\tau}$ knows when the delegation will happen using the information from the choreography. Hence, we are always in the optimal case where no messages are lost.

7. Examples

Choreography-based programming can have several applications, ranging from multicore programming to distributed Web Services. Below, we present and discuss two possible example applications.

Streaming-AVP. In this example, we show how to combine two different protocols for implementing a streaming service for movie files. We start by giving the protocol for streaming:

$$\mathsf{rec}\ \mathtt{t};\ \mathtt{S}\ \mathtt{->}\ \mathtt{C} : \langle \mathbf{bytes} \rangle;\ \mathtt{S}\ \mathtt{->}\ \mathtt{C} : \{\ \ \mathit{again} : \mathtt{t},\ \ \mathit{end} : \mathsf{end}\ \ \}$$

In the protocol above, S is a streaming server sending byte packets to a client C. After each packet, S communicates to C whether there are more bits to be sent or the stream is over (choices again and end). The other protocol (AVP, for Audio-Video Processing) is

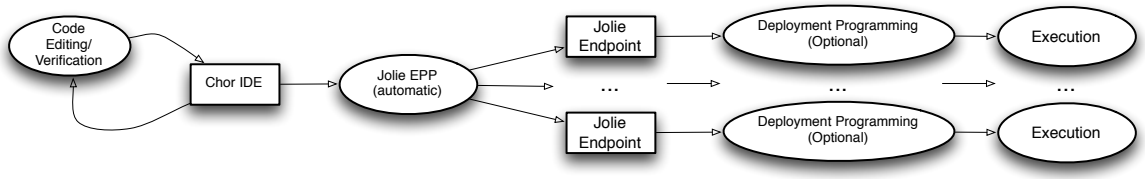


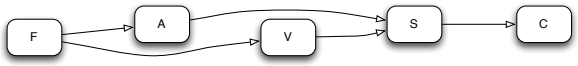
Figure 10. Development methodology with the Chor language.

defined by the following global type:

$$\texttt{rec t}; \texttt{F -> A}:\langle\textbf{bytes}\rangle; \texttt{F -> V}:\langle\textbf{bytes}\rangle; \texttt{A -> S}:\langle\textbf{bytes}\rangle; \texttt{V -> S}:\langle\textbf{bytes}\rangle;$$
$$\texttt{F -> A} : \left\{ \begin{array}{l} again : \texttt{A -> V} : \langle\textbf{bytes}\rangle;\ \texttt{V -> S} : \langle\textbf{bytes}\rangle; \texttt{t}, \\ end : \texttt{end} \end{array} \right\}$$

Four roles participate in this protocol: a filesystem F, an audio decoder A, a video decoder V, and a sink S. The flow of information in this protocol consists of F sending the raw audio information to A and the raw video information to V, read from the movie file. Then both A and V send the processed decoded information to the sink S.

The goal of this example is to show how to interleave the two protocols so that the information produced by an implementation of AVP is forwarded to a client (e.g., a display) as pictured below:



We implement such a system as follows:

$$\texttt{c[C]}\ \textbf{start}\ \texttt{s[S]} : a(stream);\ \texttt{s[S]}\ \textbf{start}\ \texttt{f[F]}, \texttt{a[A]}, \texttt{v[V]} : b(avp);$$
$$\textsf{rec AVPStreaming}(\epsilon, (avp, stream), (f, a, v, s, c)) = C \textsf{ in}$$
$$\quad \textsf{AVPStreaming}(\epsilon, (avp, stream), (f, a, v, s, c))$$

The choreography above starts two sessions (on a and b) corresponding to the two protocols specified above. Note that the streamer in the first protocol and the file system, the audio decoder and the video decoder in the second where chosen to be service threads. The core of the choreography is the term C defined as:

1. $\texttt{f[F]}.audioByteChunk$ -> $\texttt{a[A]}.audioByteChunk : avp;$
2. $\texttt{f[F]}.videoByteChunk$ -> $\texttt{v[V]}.videoByteChunk : avp;$
3. $\texttt{a[A]}.\textsf{decodeA}(audioByteChunk)$ -> $\texttt{s[S]}.audioPkt : avp;$
4. $\texttt{v[V]}.\textsf{decodeV}(videoByteChunk)$ -> $\texttt{s[S]}.videoPkt : avp;$
5. $\texttt{s[S]}.(audioPkt + videoPkt)$ -> $\texttt{c[C]}.pkt : stream;$
6. if $(\textsf{more}())@f$
7. $\quad \texttt{f[F]}$ -> $\texttt{a[A]} : avp[again];\ \texttt{a[A]}$ -> $\texttt{v[V]} : avp[again];$
8. $\quad \texttt{v[V]}$ -> $\texttt{s[S]} : avp[again];\ \texttt{s[S]}$ -> $\texttt{c[C]} : stream[again];$
9. $\quad \textsf{AVPStreaming}(\epsilon, (avp, stream), (f, a, v, s, c))$
10. else
11. $\quad \texttt{f[F]}$ -> $\texttt{a[A]} : avp[end];\ \texttt{a[A]}$ -> $\texttt{v[V]} : avp[end];$
12. $\quad \texttt{v[V]}$ -> $\texttt{s[S]} : avp[end];\ \texttt{s[S]}$ -> $\texttt{c[C]} : stream[end]$

Choreography C can be repeated several times. Lines 1-5 describe how the filesystem sends audio and video information to the threads implementing the audio and video decoders respectively. Then, the audio and video decoder resend the processed information to the thread implementing the sink. The same thread implements the streamer in the other protocol and therefore sends to the client data obtained by composing the audio and video (this is like multiplexing). Lines 7-9 correspond to the if-branch where the file server will communicate to the other threads that there is more data to process. Similarly, termination is communicated to the other threads in the else-branch (Lines 11-12). Clearly, the choreography above is well-typed wrt the two given global types.

As an example, the following process is the EPP for s:

$$!a[\texttt{S}](stream);\ \overline{b}[\texttt{S}, \texttt{F}, \texttt{A}, \texttt{V}](avp);$$
$$\textsf{rec AVPStreaming}(\epsilon, (avp, stream)) =$$
$$\left(\begin{array}{l} avp?\texttt{A}(audiopacket);\ avp?\texttt{V}(videopacket); \\ stream!\texttt{C}\langle audiopacket + videopacket\rangle; \\ avp?\texttt{V}\ \& \left\{ \begin{array}{l} again : stream!\texttt{C} \oplus again; \\ \qquad \textsf{AVPStreaming}(\epsilon, (avp, stream)) \\ end : stream!\texttt{C} \oplus end \end{array} \right\} \end{array} \right)$$
$$\textsf{in AVPStreaming}(\epsilon, (avp, stream))$$

OpenID and Logging. We give an example using a variant of OpenID [32], a protocol where a client (called user) authenticates to a server (called relying party) through a third-party identity provider. We define the protocol with the following global type:

$$\texttt{U -> RP} : \langle\textbf{string}\rangle;\ \texttt{RP -> IP} : \langle\textbf{string}\rangle;\ \texttt{U -> IP} : \langle\textbf{string}\rangle;$$
$$\texttt{IP -> RP} : \left\{ \begin{array}{ll} ok : & \texttt{RP -> U} : \{ok : \texttt{RP -> U} : \langle G@\texttt{C}\rangle; \texttt{end}\}, \\ fail : & \texttt{RP -> U} : \{fail : \texttt{end}\} \end{array} \right\}$$

Above, RP abstracts the relying party, IP the identity provider and U the user. First, U sends her username to RP, which forwards it to IP. Then, U sends her password to IP, which will notify RP of whether the username/password credentials are valid (ok or $fail$). Finally, RP forwards the notification to U. If successful, RP also delegates to U role C in a session of type G, where $G = \texttt{S -> C} : \langle\textbf{string}\rangle$.

This example interleaves OpenID with another protocol where a client C asks a log server S for either a secret or a public log. Finally, S replies with the corresponding log content. Formally,

$$\texttt{C -> S} : \{ secret : \texttt{S -> C} : \langle\textbf{string}\rangle,\ public : \texttt{S -> C} : \langle\textbf{string}\rangle \}$$

Now, we can program our system as follows:

1. $\texttt{rp[RP]}, \texttt{u[U]}\ \textbf{start}\ \texttt{ip[IP]} : publicOpenID(k);$
2. $\texttt{u[U]}.user$ -> $\texttt{rp}.user : k;$
3. $\texttt{rp[RP]}.user$ -> $\texttt{ip[IP]}.username : k;$
4. $\texttt{u[U]}.pwd$ -> $\texttt{ip[IP]}.password : k;$
5. if $(\textsf{check}(username, password))@ip$
6. $\quad \texttt{ip[IP]}$ -> $\texttt{rp[RP]} : k[ok];$
7. $\quad \texttt{rp[RP]}$ -> $\texttt{u[U]} : k[ok];$
8. $\quad$ if $(\textsf{high}(username))@\texttt{rp}$
9. $\quad\quad \texttt{rp[C]}\ \textbf{start}\ \texttt{s[S]} : log(k');$
10. $\quad\quad \texttt{rp[C]}$ -> $\texttt{s[S]} : k'[secret];$
11. $\quad\quad \texttt{rp[RP]}$ -> $\texttt{u[U]} : k\langle\langle k'[\texttt{C}]\rangle\rangle;$
12. $\quad\quad \texttt{s[S]}.secret\_msg$ -> $\texttt{u[C]}.logContent : k'$
13. $\quad$ else
14. $\quad\quad \texttt{rp[C]}\ \textbf{start}\ \texttt{s[S]} : log(k');$
15. $\quad\quad \texttt{rp[C]}$ -> $\texttt{s[S]} : k'[public];$
16. $\quad\quad \texttt{rp[RP]}$ -> $\texttt{u[U]} : k\langle\langle k'[\texttt{C}]\rangle\rangle;$
17. $\quad\quad \texttt{s[S]}.public\_msg$ -> $\texttt{u[C]}.logContent : k'$
18. else
19. $\quad \texttt{ip[IP]}$ -> $\texttt{rp[RP]} : k[fail];$
20. $\quad \texttt{ip[IP]}$ -> $\texttt{u[U]} : k[fail]$

where rp, u, ip, and s are the endpoints of the system. Line 1 describes the initiation of a protocol instance between rp, u and ip, by means of the public name a. In Lines 2-4, u sends its credentials to rp and ip (only username to rp). Then, ip checks the data received

(Line 5) and communicates the outcome to rp (Lines 6 and 19). In both cases, the selection is forwarded to u (Lines 7 and 20). In the if-branch, rp checks the user access level. If high, it starts a new session (log) spawning s (Line 9) and asks for a secret log (Line 10). Consequently, rp will delegate its role in session k' to the user u through session k. Finally, u will get the requested log. The system works similarly in the public log case (Lines 14-17).

We conclude by showing the EPP for service thread s:

$$!log[\mathsf{S}](k');\ k'?\mathsf{C}\ \&\ \left\{ \begin{array}{l} secret : k'!\mathsf{C}\langle private\_msg\rangle, \\ public : k'!\mathsf{C}\langle public\_msg\rangle \end{array} \right\}$$

Note how the thread projections of s for the different branches of the choreography (Lines 9-12 and 14-17) would yield different code which is then merged into the one above.

Other examples. In the Chor website [16] we provide other examples, which can also be directly tested in the Chor IDE. The examples include other applications of OpenID, usage of the choreography language for programming a use case of the Ocean Observatories Initiative [31], and other web services applications.

8. Related Work

Global methods for communicating systems occur in different forms, including MSC [25], security protocols [8, 9, 12] and automata theory [21]. However, these works are not intended as fully-fledged programming languages since they do not deal with, e.g., different layers of abstraction or value passing.

This is the first work proposing an asynchronous semantics for a choreography language based on sessions. To the best of our knowledge, the notion of *delayed input* [27] is the most similar result to the asynchrony modelled by our semantics.

The closest work to ours is [15], which proposes a *synchronous* choreography model without delegation based on *binary* (endpoint) session types. Our framework shows that switching to multiparty asynchronous sessions with delegation introduces more complexity, but also that such complexity can be elegantly hidden from the programmer. Moreover, [15] has implicit threads and deals with a stronger sequential operator, requiring two syntactic restrictions on choreographies, i.e., well-threadedness and connectedness. In contrast, our approach needs no such restrictions because of explicit threads and a more relaxed sequential operator. Finally, [15] ensures EPP correctness based on a type preservation result, while we guarantee the same without the need for an endpoint typing.

Multiparty session types have been previously used for checking endpoint systems [7, 17, 22]. We have shown that they can be adopted for typing choreographies, defining a new class of correct well-typed endpoint systems (through EPP). Our global types as well as our endpoint model are taken by [7]. Other works have given an asynchronous semantics to global types: [22] defines a semantics in terms of that of the projection of global types while [18] interprets global types as asynchronous communication automata. Our linearity notion is inspired by [22].

[7] guarantees *progress* for multiparty sessions by building additional restrictions on top of (endpoint) session typing. Processes satisfying progress do not get stuck provided that they can be run in parallel with other processes that would unlock stuck states. In our work progress, implied by deadlock-freedom, is an immediate consequence of our EPP Theorem, yielding a simpler analysis.

9. Discussion and Future Extensions

We discuss some aspects of choreography-driven programming and future extensions in relation to the work presented in this article.

Approach. It may be unclear how the choreography-driven approach may deal with standard aspects of programming such as *choreography composition* and *endpoint code reuse*.

Choreography composition is fundamental for supporting distributed (team) development. Our framework supports it with (i) service merging and (ii) procedures. (i) Our EPP merges service threads started on the same public channel and role into a single process. This allows two choreographies to be composed into a bigger system, whenever their respective service threads are mergeable. Mergeability can be assured by using a design pattern, i.e. enforcing service threads that need to be merged to start with distinct branches. (ii) Procedures can be written and typed separately, so to create libraries that can be used by other choreographies (code reuse). As future work, we plan to extend Chor with a namespacing system to fully support this methodology.

Endpoint code reuse may be necessary when parts of the system being designed are already implemented. For instance, we may want to reuse an existing identity provider service in § 7, OpenID and Logging. Our model does not currently offer a way of integrating existing endpoint code with the EPP of a choreography. We discuss some potential solutions, which we leave as future work.

Using *bisimulation* techniques, we can verify some existing service code to be bisimilar to the code that would be generated by the EPP [10]. Alternatively, we could use a type system, such as multiparty session typing [7, 22], for guaranteeing that the existing code has a behaviour "compatible" with the choreography.

Both the techniques mentioned above can be adjusted to allow for refinement, i.e. the legacy code may do extra actions as long as they do not interfere with the good behaviour of the choreography. The resulting system would still guarantee communication safety and session fidelity (protocol compliance). Deadlock-freedom would also still be guaranteed, provided that the legacy code has been verified to be deadlock-free. Note that our implementation of delegation (§ 6, Delegation) would need to fall back to the protocol presented in [24] for those sessions whose behaviour is partly implemented by legacy code, since the latter cannot refer to the necessary information provided by the choreography.

GC features. Based on our conversations with our industry collaborators, we discuss some relevant aspects and extensions of GC.

Sequential and Parallel operators. The relaxed semantics of our sequential operator ";" allows our framework to express parallelism with a minimal syntax. However, an explicit parallel operator may make choreographies more readable. We discuss here two possible extensions in this sense, based upon a hypothetical operator $C_1 \mid C_2$ equipped with the classical interleaving semantics of parallel composition. We use the following choreography as reference example:

$$C_{\mathsf{par}} \ = \ \tau_1[\mathtt{p}_1].e \texttt{ -> } \tau_2[\mathtt{p}_2].x : k \ \mid \ \tau_3[\mathtt{p}_3].e' \texttt{ -> } \tau_2[\mathtt{p}_4].y : k'$$

Whenever $\tau_1, \tau_2, \tau_3, \tau_4$ are all different, C_{par} can be encoded using our sequential operator:

$$C_{\mathsf{seq}} \ = \ \tau_1[\mathtt{p}_1].e \texttt{ -> } \tau_2[\mathtt{p}_2].x : k;\ \tau_3[\mathtt{p}_3].e' \texttt{ -> } \tau_2[\mathtt{p}_4].y : k'$$

In fact, thanks to our swap relation $\simeq_{\mathcal{C}}$, C_{par} would behave exactly as C_{seq} . If the two interactions in C_{par} share a thread name, e.g., $\tau_2 = \tau_4$, the parallel operator $\mid$ would not be syntax sugar anymore, since the projection of τ_2 would be a parallel composition of two input actions. This also means that τ_2 would no longer be a simple sequential thread, raising the complexity of our framework and going out of the scope of the present work.

Public channel passing. We leave the treatment of public channel name passing as future work. The main challenge is to statically establish where such channels can be located in the endpoints.

Interactional exceptions. Our language does not offer specific features for error/exception recovery. A possible extension is to include exceptions in our choreography language in the spirit of what is informally suggested by [13] for binary sessions.

Dynamic join. We plan to extend our model to allow threads to dynamically join and leave an existing session, similarly to [11, 17].

We conjecture that asynchrony and parallelism will influence these extensions in a nontrivial way.

Multiple roles. In our model, a thread can play only one role per session. However, there are cases in which multiple roles in a protocol may be implemented by a single thread, which would be useful both for system simplification and resource saving. We plan to extend our typing system to allow for this occurrences. [6, 11] follow a similar direction, but using endpoint implementations.

Global deployment. Chor projects Jolie programs with a default deployment configuration that can be edited afterwards for each endpoint. This may be inconvenient for the programmer, since a choreography may describe many participants. Therefore, we plan to introduce a global deployment language for choreographies, from which the deployment configuration of each endpoint could be automatically generated. We envision that this extension could be relevant for facilitating checks on the correctness of a deployment configuration of a system, e.g., consistent I/O connections.

Security. Global descriptions are particularly suitable for the study of security-related aspects in distributed systems [8, 9, 12]. Our choreographies give a global view of how sessions are interleaved and how their references are transmitted. It would be interesting to see how this aspect may influence the security analysis of a system.

Scaffolding. Since in our model a protocol and a session behaviour in a choreography have similar structures, we could implement a scaffolding tool in our IDE that, given a protocol, would generate a prototype choreography with "dummy" data that implements it. Then, programmers would refine and interleave different prototypes to obtain the desired behaviour.

10. Conclusions

We presented a fully-fledged model for defining asynchronous system implementations as global programs. We developed a type system for checking choreographies against multiparty protocol specifications. Moreover, through type inference, developers can also use choreographies as implementation prototypes to infer new protocol standards. Our EPP generates correct endpoint code, ensuring nontrivial properties such as deadlock freedom and communication safety. Finally, we provided a prototype implementation of our framework and applied it to some realistic examples.

Acknowledgements. We are grateful to N. Yoshida, J. Bengtson, T. Hildebrandt, K.Honda, C. Schürmannn, V. Vasconcelos, and the anonymous reviewers for their comments. The Danish Agency for Science, Technology and Innovation supported Carbone.

References

[1] BPMN. `http://www.omg.org/spec/BPMN/2.0/`.

[2] italianaSoftware. `http://www.italianasoftware.com/`.

[3] Jolie language. `http://www.jolie-lang.org/`.

[4] SOAP Specifications. `http://www.w3.org/TR/soap/`.

[5] WS-Addressing. `http://www.w3.org/TR/ws-addr-core/`.

[6] P. Baltazar, L. Caires, V. T. Vasconcelos, and H. T. Vieira. A Type System for Flexible Role Assignment in Multiparty Communicating Systems. In *Proc. of TGC*, 2012. To appear.

[7] 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.

[8] 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.

[9] S. Briais and U. Nestmann. A formal semantics for protocol narrations. *Theoretical Computer Science*, 389(3):484–511, 2007.

[10] 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.

[11] L. Caires and H. T. Vieira. Conversation types. *Theoretical Computer Science*, 411(51-52):4399–4440, 2010.

[12] C. Caleiro, L. Viganò, and D. A. Basin. On the semantics of Alice&Bob specifications of security protocols. *Theor. Comput. Sci.*, 367(1-2):88–122, 2006.

[13] M. Carbone. Session-based choreography with exceptions. In *Proc. of PLACES*, volume 241, pages 35–55. ENTCS, 2008.

[14] M. Carbone and F. Montesi. Typed multiparty global programming. Technical Report 149, IT University of Copenhagen, 2011. `http://www.itu.dk/people/fabr/papers/multichor`.

[15] 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.

[16] Chor. Programming Language. `http://www.chor-lang.org/`.

[17] P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *Proc. of POPL*, pages 435–446. ACM, 2011.

[18] P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *Proc. of ESOP*, LNCS, pages 194–213. Springer-Verlag, 2012.

[19] Eclipse. The Eclipse IDE. `http://www.eclipse.org/`.

[20] Enea. ENEA: Italian National agency for new technologies, Energy and sustainable economic development. `http://www.enea.it/`.

[21] 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.

[22] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. of POPL*, volume 43(1), pages 273–284. ACM, 2008.

[23] 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.

[24] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in java. In *ECOOP*, pages 516–541, 2008.

[25] International Telecommunication Union. Recommendation Z.120: Message Sequence Chart, 1996.

[26] 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.

[27] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.

[28] F. Montesi and M. Carbone. Programming services with correlation sets. In *ICSOC*, pages 125–141, 2011.

[29] F. Montesi, C. Guidi, and G. Zavattaro. Composing Services with JOLIE. In *Proc. of ECOWS*, pages 13–22, 2007.

[30] OASIS. Web Services Business Process Execution Language. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.

[31] OOI. Ocean Observatories. `http://www.oceanobservatories.org`.

[32] OpenID. Specifications. `http://openid.net/developers/specs/`.

[33] PI4SOA. http://www.pi4soa.org, 2008.

[34] B. C. Pierce. *Types and Programming Languages*. MIT Press, MA, USA, 2002.

[35] Qualit-E. Funded by Cognizant. `http://www.cognizant.com/`.

[36] Savara. JBoss Community. `http://www.jboss.org/savara/`.

[37] W3C WS-CDL Working Group. Web services choreography description language version 1.0. http://www.w3.org/TR/ws-cdl-10/, 2004.

[38] 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.