Multiparty Classical Choreographies

Marco Carbone[1], Luís Cruz-Filipe[2], Fabrizio Montesi[2], and Agata Murawska[1]

[1]IT University of Copenhagen
{carbonem,agmu}@itu.dk

[2]University of Southern Denmark
{lcf,fmontesi}@imada.sdu.dk

Abstract. We present Multiparty Classical Choreographies (MCC), a language model where global descriptions of communicating systems (choreographies) implement typed multiparty sessions. Typing is achieved by generalising classical linear logic to judgements that explicitly record parallelism by means of hypersequents. Our approach unifies different lines of work on choreographies and processes with multiparty sessions, as well as their connection to linear logic. Thus, results developed in one context are carried over to the others. Key novelties of MCC include support for server invocation in choreographies, as well as logic-driven compilation of choreographies with replicated processes.

1 Introduction

Choreographic Programming [17] is a programming paradigm where programs, called *choreographies*, define the intended communication behaviour of a system based on message passing, using an "Alice and Bob" notation, rather than the behaviour of each endpoint. Choreographies are useful for several reasons: they give a succinct description, or *blueprint*, of the intended behaviour of a whole system, making the implementation less error-prone. Then, correct-by-construction distributed implementations can be synthesised automatically by means of *projection*, a compilation algorithm that generates the code for each endpoint described in the choreography [6, 8]. Reversely, it is often possible to obtain a choreography from an endpoint implementation by means of *extraction*, providing a precise blueprint of a distributed system.

Choreographic programming has a deep relationship with the proof theory of linear logic [9]. Specifically, choreographic programs can be seen as terms describing the reduction steps of cut elimination in linear logic (choreographies as cut reductions). The key advantage of this result is that it provides a logical reconstruction of two useful translations, one from choreographies to processes (*projection*, or synthesis) and another from processes to choreographies (*extraction*) – this is obtained by exploiting the correspondence between intuitionistic linear logic and a variant of the π-calculus [4]. These translations can be used to keep process implementations aligned with the desired communication flows given as choreographies, whenever code changes are applied to any of the two. This kind of alignment is a desirable property in practice, e.g., it is the basis of the Testable Architecture development lifecycle for web services [14].

Unfortunately, the logical reconstruction of choreographies in [9] covers only the multiplicative-additive fragment of intuitionistic linear logic, limiting its

practical applicability to simple scenarios. The aim of this paper is to push the boundaries of this approach towards more realistic scenarios with sophisticated features. In this article, we define a model, strictly related to classical linear logic, that allows for *replicated services*, and *multiparty sessions*.

Reaching our aim is challenging for both design and technical reasons. In the multiplicative-additive fragment of linear logic considered in [9], all reductions intuitively match choreographic terms explored in previous works on choreographies, i.e., communication of a channel and branch selection [8]. This is not the case for the exponential fragment, which yields *reductions never considered before in choreographies*, e.g., explicit garbage collection of services and server cloning (see kill and clone operations). To bridge this gap, we exploit the fact that these operations occur naturally in the process language and, through the logic, can be reflected to choreographic primitives for management of services as explicit resources that can be duplicated, used, or destroyed. We show that the reductions for these terms correspond to the principal cut reductions for exponentials in classical linear logic. Typing guarantees that resource management is safe, e.g., no destroyed resource is ever used again.

In [9], all sessions (protocols) have exactly two participants. This works well in intuitionistic linear logic, where sequents are two-sided: two processes can be connected if one "provides" a behaviour and the other "needs" it. This is verified by checking identity of types, respectively between a type on the right-hand side of the sequent of the first process and a type of the left-hand side of the sequent for the second. To date, it is still unclear how identity for two-sided sequents can be generalised to multiparty sessions, where a session can have multiple participants and thus we need to check compatibility of multiple types. Instead, this topic has been investigated in the setting of classical linear logic, where multiparty compatibility is captured by coherence, a generalisation of duality [10]. Therefore, our formulation of Multiparty Classical Choreographies (MCC) is based on classical linear logic. In order to bridge choreographies to multiparty sessions, we introduce a new session environment, which records the types of multiparty communications performed by a choreography as *global types* [13]. The manipulation of the session environment reveals that typing a choreography with multiparty sessions corresponds to *building the coherence proofs for typing its sessions*. Since a proof of coherence is the type compatibility check required by the multiparty version of cut in classical linear logic, our result generalises the choreographies as cut reductions approach to the multiparty case as one would expect, providing further evidence of the robustness of this idea. The final result of our efforts is an expressive calculus for programming choreographies with multiparty sessions and services, which supports both projection and extraction operations *for all typable programs*.

2 Preview

We start by introducing MCC informally, focusing on modelling a protocol inspired by OpenID [20], where a client authenticates through a third-party iden-

tity provider. MCC offers a way of specifying protocols in terms of global types. For example, our variant of OpenID can be specified by the global type G:

$$u \to rp(\mathsf{String}); u \to ip(\mathsf{String}); u \to ip(\mathsf{PWD}); ip \to rp.\mathsf{case}(\ u \to rp(\mathsf{String}); G_1,\ G_2)$$

This protocol concerns three endpoints (often called roles in literature) denoted by u (user), rp (relaying party) and ip (identity provider). The user starts by sending its login string to both rp and ip. Then, it sends its password to ip which will either confirm or reject u's authentication to rp. If the authentication is successful then the user will send an evaluation of the authentication service to rp, and then complete as the unspecified protocol G_1. Otherwise, if the password is wrong, then the protocol continues as G_2. The specification given by the global type G can be used by a programmer during an implementation. In MCC, we could give an implementation in terms of the choreography:

$$\begin{array}{ll}
u\,\mathsf{starts}\,rp, ip; & \texttt{// } u \texttt{ starts protocol with rp and ip} \\
u(user_u) \to rp(user_{rp}); & \texttt{// } u \texttt{ sends its login to } rp \\
u(login_u) \to ip(login_{ip}); & \texttt{// } u \texttt{ sends its login to } ip \\
u(pwd_u) \ \to \ ip(pwd_{ip}); & \texttt{// } u \texttt{ authenticates with } ip \\
ip \ \to rp. \begin{cases} \mathsf{inl} : u'\,\mathsf{starts}\,s; & \texttt{// } u' \texttt{ starts protocol with } s \\ \quad u'(rep_{u'}) \to s(rep_s); & \texttt{// } u' \texttt{ sends report to } s \\ \quad s(ack_s) \to u'(ack_{u'}); & \texttt{// } s \texttt{ acknowledges to } u' \\ \quad u(rep_u) \to rp(rep_{rp}); P, & \texttt{// } u \texttt{ sends report to } rp \\ \mathsf{inr} : Q & \texttt{// authentication fails} \end{cases}
\end{array}$$

Each line is commented with an explanation of the performed action. We observe that two different protocols are started. The first line starts the OpenID protocol between u, rp and ip described above. Moreover, after branching, the choreography starts another session between the user (named u') and a server s that is used for reviewing the authentication service given by ip. In this case, the protocol used is $G' = u' \to s(\mathsf{String}); s \to u'(\mathsf{String}); G_3$, for some unspecified G_3. We leave undefined the case in which the identity provider receives a wrong password (term Q).

In this work, we show how a choreography that follows a protocol such as G can be expressed as a proof in a proof theory strictly related to classical linear logic. Moreover, thanks to proof transformations, the choreography above can be projected into a parallel composition of endpoint processes, each running a different endpoint. As an example, the endpoint process for the user would correspond to the process P_u, defined as

$$\mathsf{use}\,u; \overline{u}(user_u); \overline{u}(login_u); \overline{u}(pwd_u); \mathsf{use}\,u'; \overline{u'}(rep_{u'}); u'(ack_{u'}); \overline{u}(rep_u); R$$

which mimics the behaviour of u and u' specified in the choreography. Operator use is used to start a session, while the other two operators utilised above are for in-session communication. Similarly, we can have the endpoint processes for rp, ip and s:

$$P_{rp} = \mathsf{srv}\,rp; rp(user_{rp}); rp.\mathsf{case}(rp(rep_{rp}); R_1, Q_1)$$
$$P_{ip} = \mathsf{srv}\,ip; ip(login_{ip}); ip(pwd_{ip}); R_2 \qquad P_s = \mathsf{srv}\,s; s(rep_s); \overline{s}(ack_s); R_3$$

3 GCP with Hypersequents

In this section, we present the *action fragment* of MCC, where we only consider local actions, e.g., inputs or outputs. The action fragment is a variant of Globally-governed Classical Processes (GCP) [7] whose typing rules use hypersequents. In the remainder, we denote a vector of endpoints $x_1, \ldots, x_n$ as $\tilde{x}$ or $(x_i)_i$.

Syntax. The action fragment is a generalisation of Classical Processes [22] that supports multiparty session types. As hinted in §2, when writing a program in our language, we do not identify sessions via channel names, but rather we name sessions' *endpoints*. Each process owns a single endpoint of a session it participates in. The complete syntax is given by the following grammar:

$P ::=$	$x^A \to y$	link	$\mid \mathsf{use}\, x; P$	client
	$\mid P \mid Q$	parallel	$\mid \mathsf{srv}\, y; P$	server
	$\mid (\boldsymbol{\nu} \tilde{x} : G)\, P$	restriction	$\mid \mathsf{kill}\, x \mid P$	server kill
	$\mid \overline{x}(x'); (P \mid Q)$	send	$\mid \mathsf{clone}\, x(x'); P$	server clone
	$\mid y(y'); P$	receive		
	$\mid \mathsf{close}[x]$	close session		
	$\mid \mathsf{wait}[y]; P$	receive close	$\mid y.\mathsf{case}(P, Q)$	branching
	$\mid x.\mathsf{inl}; P$	left selection	$\mid x.(\mathsf{inl} : P, \mathsf{inr} : Q)$	general selection
	$\mid x.\mathsf{inr}; Q$	right selection	$\mid x^{\tilde{u}}.\mathsf{case}()$	empty choice

With a few exceptions, the terms above are identical to those of GCP. *For space restriction reasons, we only discuss the key differences.* Parallel and restriction constructs form a single term $(\nu \tilde{x} : G)(P \mid Q)$ in the original GCP. The link process $x^A \to y$ is a forwarder from x to y. We further allow the general selection $x.(\mathsf{inl} : P, \mathsf{inr} : Q)$, denoting a process that non-deterministically selects a left or a right branch. For services, an endpoint x may kill all servers by executing the action $\mathsf{kill}\, x \mid P$, or duplicate them by means of $\mathsf{clone}\, x(x'); P$ – these operations were silent in the original GCP. In cloning, the new server copies are replicated at fresh endpoints, ready to engage in a session with new endpoint x'. More generally, we follow the convention of [22], denoting the result of refreshing names in Q by Q' (changing each $x \in \mathsf{fv}(Q)$ into a fresh x').

Types. Types, used to ensure proper behaviour of endpoints, are defined as:

$A ::=$	$A \otimes B$	output	$\mid A \parr B$	input	$G ::=$	$\tilde{x} \to y(G); H$	$(\otimes \parr)$
	$\mid A \oplus B$	selection	$\mid A \,\&\, B$	choice		$\mid x \to \tilde{y}.\mathsf{case}(G, H)$	$(\oplus \,\&)$
	$\mid\, !A$	server	$\mid\, ?A$	client		$\mid\, !x \to \tilde{y}(G)$	$(!?)$
	$\mid \mathbf{1}$	close	$\mid \bot$	wait		$\mid \tilde{x} \to y$	$(\mathbf{1}\bot)$
	$\mid 0$	false	$\mid \top$	empty		$\mid x \to \tilde{y}.\mathsf{case}()$	$(0\top)$
	$\mid X$	variable	$\mid X^{\bot}$	dual variable		$\mid x^A \to y$	(AXIOM)

In the multiparty setting, types can be split into *local types* A, which specify behaviours of a single process, and *global types* G, which describe interaction within sessions (and choreography actions). Again, most global types correspond to pairs of local types, the exception being the global axiom type, describing a linking session (restricted by typing to type variables and their duals). Local type operators are based on connectives from classical linear logic – thus, $A \otimes B$

is the type of a process that outputs an endpoint of type A and continues with type B, whereas $A \parr B$ is the type of a process that receives endpoints of type A and is itself ready to continue as B. The corresponding global type $\tilde{x} \to y(G); H$ types the interaction where each of the processes owning an endpoint x_i sends their new endpoint to y. Type 0 is justified by the necessity of having a type dual to $\top$, while the rule $0\top$ is essential for the definition of coherence. Type variables are used to represent concrete datatypes. It is worth noting that the logic formulas in our type system enjoy the usual notion of duality, where a formula's dual is obtained by recursively replacing each connective by the other one in the same row in the table above. For example, the dual of $!(A \otimes 0)$ is $?(A^{\perp} \parr \top)$, where $A^{\perp}$ is the dual of formula A.

Typing. We type our terms in judgements of the form $\Sigma \Vdash P \vdots \Psi$, where: (i) Σ is a set of session typings of the form $(x_i)_i : G$; (ii) P is a process; and, (iii) Ψ is a hypersequent, a set of classical linear logic sequents. Intuitively, $\Sigma \Vdash P \vdots \Psi$ reads as *"Ψ types P under the session protocols described in Σ."*
Given a judgment $\Sigma \Vdash P \vdots \Psi \mid \vdash \Gamma, x : A$, checking whether x is *available* – not engaged in a session – is implicitly done by verifying that x does not occur in the domain of Σ. Note that names cannot occur more than once in Σ: each endpoint x may only belong to (at most) one session G. Hypersequents Ψ_1, Ψ_2 and sets of sessions Σ_1, Σ_2 can only be joined if their domains do not intersect. Moreover, we use indexing in different ways: $\left(\Sigma_i \Vdash P_i \vdots \Psi_i\right)_i$ denotes several judgements $\Sigma_1 \Vdash P_1 \vdots \Psi_1, \ldots, \Sigma_n \Vdash P_n \vdots \Psi_n$; indexed pairs $(x_i : A_i)_i$ are a set of pairs $x_1 : A_1, \ldots, x_n : A_n$; and, finally, $(\vdash \Gamma_i)_i$ denotes the hypersequent $\vdash \Gamma_1 \mid \ldots \mid \vdash \Gamma_n$.

In order to separate restriction and parallel (reasons for this separation will be explained in § 4), we split the classical linear logic Cut rule into two:

$$\frac{(\Sigma_i \Vdash P_i \vdots \Psi_i \mid \vdash \Gamma_i, x_i : A_i)_i \quad G \vDash (x_i : A_i)_i}{(\Sigma_i)_i, (x_i)_i : G \Vdash (P_i)_i \vdots (\Psi_i \mid \vdash \Gamma_i, x_i : A_i)_i}\ \mathsf{Conn} \qquad \frac{\Sigma, (x_i)_i : G \Vdash P \vdots \Psi \mid (\vdash \Gamma_i, x_i : A_i)_i}{\Sigma \Vdash (\boldsymbol{\nu} \tilde{x} : G)\, P \vdots \Psi \mid \vdash (\Gamma_i)_i}\ \mathsf{Scope}$$

Rule Conn is used for merging proofs that provide *coherent* types (we address coherence below), but without removing them from the environment. Since such types need to remain in the conclusion of the rule, we need to use hypersequents. The sequents involved in a session get merged once a Scope rule is applied. This hypersequent presentation is similar to a classical linear logic variant of [9] with sessions explicitly remembered in a separate context Σ.

Coherence is a generalisation of duality [7] to more than two parties: when describing a multiparty session, simple duality of types does not suffice to talk about their compatibility. In Fig. 1, we report the rules defining the coherence relation $\vDash$. We do not describe these here in detail, as they remain unchanged compared to the original GCP presentation, with the exception of the axiom rule which is only applicable to atomic types in our system.

The remaining typing rules for the action fragment, presented in Fig. 2 are identical to those of GCP with the exception that a context in GCP may be distributed among several sequents here. For example, rule $\otimes$ takes two sequents $\vdash \Gamma_1, x' : A$ and $\vdash \Gamma_2, x : B$ from two different hypersequents, and merges them

$$\frac{G \vDash (x_i : A_i)_i,\ y : C \qquad H \vDash \Gamma,\ (x_i : B_i)_i,\ y : D}{\tilde{x} \to y(G); H \vDash \Gamma,\ (x_i : A_i \otimes B_i)_i,\ y : C \parr D}\ \otimes\parr \qquad \frac{}{\tilde{x} \to y \vDash (x_i : \mathbf{1})_i, y : \bot}\ \mathbf{1}\bot$$

$$\frac{G_1 \vDash \Gamma, x : A, (y_i : C_i)_i \qquad G_2 \vDash \Gamma, x : B, (y_i : D_i)_i}{x \to \tilde{y}.\mathsf{case}(G_1, G_2) \vDash \Gamma, x : A \oplus B, (y_i : C_i \ \&\ D_i)_i}\ \oplus\& \qquad \frac{G \vDash x : A, (y_i : B_i)_i}{!x \to \tilde{y}(G) \vDash x : ?A, (y_i : !B_i)_i}\ !?$$

$$\frac{}{x \to \tilde{y}.\mathsf{case}() \vDash \Gamma, x : 0, (y_i : \top)_i}\ 0\top \qquad \frac{A^\top = X \text{ or } A = X^\perp}{x^A \to y \vDash x : A, y : A^\perp}\ \textsc{Axiom}$$

Fig. 1. Coherence rules.

$$\frac{A = X \text{ or } A = X^\perp}{\cdot \Vdash x^A \to y \mathrel{\wr} \vdash x : A, y : A^\perp}\ \mathsf{Ax} \qquad \frac{\Sigma_1 \Vdash P \mathrel{\wr} \Psi_1 \mid \vdash \Gamma_1, x' : A \qquad \Sigma_2 \Vdash Q \mathrel{\wr} \Psi_2 \mid \vdash \Gamma_2, x : B}{\Sigma_1, \Sigma_2 \Vdash \overline{x}(x'); (P \mid Q) \mathrel{\wr} \Psi_1 \mid \Psi_2 \mid \vdash \Gamma_1, \Gamma_2, x : A \otimes B}\ \otimes$$

$$\frac{\Sigma \Vdash P \mathrel{\wr} \Psi \mid \vdash \Gamma, y' : A, y : B}{\Sigma \Vdash y(y'); P \mathrel{\wr} \Psi \mid \vdash \Gamma, y : A \parr B}\ \parr \qquad \frac{\Sigma \Vdash P \mathrel{\wr} \Psi \mid \vdash \Gamma}{\Sigma \Vdash \mathsf{wait}[y]; P \mathrel{\wr} \Psi \mid \vdash \Gamma, y : \bot}\ \bot$$

$$\frac{}{\cdot \Vdash \mathsf{close}[x] \mathrel{\wr} \vdash x : \mathbf{1}}\ \mathbf{1} \qquad \text{(no rule for } \mathbf{0}\text{)} \qquad \frac{\mathsf{vars}(\Gamma) = \tilde{u}}{\cdot \Vdash x^{\tilde{u}}.\mathsf{case}() \mathrel{\wr} \vdash \Gamma, x : \top}\ \top$$

$$\frac{\Sigma \Vdash P \mathrel{\wr} \Psi \mid \vdash \Gamma, x : A}{\Sigma \Vdash x.\mathsf{inl}; P \mathrel{\wr} \Psi \mid \vdash \Gamma, x : A \oplus B}\ \oplus_1 \qquad \frac{\Sigma \Vdash Q \mathrel{\wr} \Psi \mid \vdash \Gamma, x : B}{\Sigma \Vdash x.\mathsf{inr}; Q \mathrel{\wr} \Psi \mid \vdash \Gamma, x : A \oplus B}\ \oplus_2$$

$$\frac{\Sigma \Vdash P \mathrel{\wr} \Psi \mid \vdash \Gamma, x : A \qquad \Sigma \Vdash Q \mathrel{\wr} \Psi \mid \vdash \Gamma, x : B}{\Sigma \Vdash x.(\mathsf{inl} : P, \mathsf{inr} : Q) \mathrel{\wr} \Psi \mid \vdash \Gamma, x : A \oplus B}\ \oplus \qquad \frac{\cdot \Vdash P \mathrel{\wr} \vdash ?\Gamma, y : A}{\cdot \Vdash \mathsf{srv}\, y; P \mathrel{\wr} \vdash ?\Gamma, y : !A}\ !$$

$$\frac{\Sigma \Vdash P \mathrel{\wr} \Psi \mid \vdash \Gamma, y : A \qquad \Sigma \Vdash Q \mathrel{\wr} \Psi \mid \vdash \Gamma, y : B}{\Sigma \Vdash y.\mathsf{case}(P, Q) \mathrel{\wr} \Psi \mid \vdash \Gamma, y : A \ \&\ B}\ \& \qquad \frac{\Sigma \Vdash P \mathrel{\wr} \Psi \mid \vdash \Gamma, x : A}{\Sigma \Vdash \mathsf{use}\, x; P \mathrel{\wr} \Psi \mid \vdash \Gamma, x : ?A}\ ?$$

$$\frac{\Sigma \Vdash P \mathrel{\wr} \Psi \mid \vdash \Gamma}{\Sigma \Vdash \mathsf{kill}\, x \mid P \mathrel{\wr} \Psi \mid \vdash \Gamma, x : ?A}\ \mathsf{Weaken} \qquad \frac{\Sigma \Vdash P \mathrel{\wr} \Psi \mid \vdash \Gamma, x : ?A, x' : ?A}{\Sigma \Vdash \mathsf{clone}\, x(x'); P \mathrel{\wr} \Psi \mid \vdash \Gamma, x : ?A}\ \mathsf{Contract}$$

Fig. 2. Rules for the action fragment.

into $\vdash \Gamma_1, \Gamma_2, x : A \otimes B$, as in classical linear logic. However, elements of Γ_1 and Γ_2 may be connected through Σ_1 and Σ_2 to other parts of Ψ_1 and Ψ_2 respectively (as a result of previously applied Conn). Note that the rules of this fragment work only with processes not engaged in any session, since the endpoints explicitly mentioned in proof terms cannot occur in the domain of Σ: this is an implicit check in all rules of Fig. 2. Rule $\top$ introduces a single sequent $\Gamma, x : \top$, allowing for any Γ. The proof term $x^{\tilde{u}}.\mathsf{case}()$ keeps track of the endpoints introduced in Γ: it ensures that all endpoints in the typing are mentioned in the proof term, which is useful when defining semantics. In this article, we restrict the axiom to only type variables (see §6).

Semantics. The semantics of the action fragment is almost identical to that of standard GCP. It is obtained from cases of the proof of cut elimination: the principal cases describe reductions ($\longrightarrow$), while the permutations of rule applications give rise to the rules for structural equivalence ($\equiv$), reported in Fig. 3. Note that as we are interested only in commuting conversions of typable programs, there are certain cases where the correct equivalence can be found only by looking at the typing derivation which contains information that is not part of the process term. Under $\equiv$, parallel distributes safely over case (because

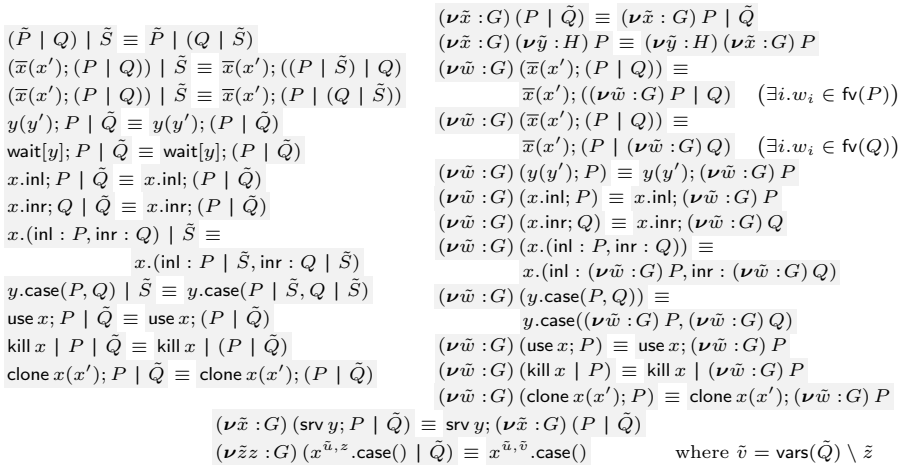

Fig. 3. Equivalences for commuting the action fragment with Conn and Scope. All rules assume that both sides of the equation are typable in the same context.

only the actions of one branch are going to be executed). A similar mechanism can be found in the original presentation of Classical Processes [22], and was later demonstrated to correspond to a bisimulation law in [1]. The semantics of the action fragment of our calculus is presented in Fig. 4. Notice that the β-reductions are coordinated by a global type, as they correspond to multiple parties communicating.[1] The reduction rules for server killing and cloning may look strange because both kill and clone remain in the proof term after reduction. This is because of the corresponding reduction in classical linear logic, where it is necessary to use weakening and contraction (corresponding to kill and clone respectively) also after reduction. As a consequence, we get them as proof terms.

4 Extending GCP with Choreographies

In order to obtain full MCC, we extend the action fragment presented in the previous section with choreography terms (interactions).

Syntax. Unlike a process in the action fragment, a choreography, which describes a global view of the communications of a process, will own all of the endpoints of the sessions it describes. We call the fragment of MCC with choreography terms

[1] It may be surprising that some of the rules also include a restriction to a vector $\tilde{z}$, and a session using a vector of processes $\tilde{S}$, whose shape we do not inspect. This follows from the shape of coherence rules: rules such as $\otimes\, ⅋$, $\oplus\, \&$ and $0\, \top$ contain an additional context Γ, captured here by $\tilde{z}$.

$$(\boldsymbol{\nu}\tilde{x}, y, \tilde{z} : \tilde{x} \to y(G); H)\ \big((\overline{x_i}(x_i'); (P_i \mid Q_i))_i \mid y(y'); R \mid \tilde{S}\big) \longrightarrow (\boldsymbol{\nu}\tilde{x}', y' : G\{\tilde{x}'/\tilde{x}, y'/y\})\ \big(\tilde{P} \mid (\boldsymbol{\nu}\tilde{x}, y, \tilde{z} : H)\ (\tilde{Q} \mid R \mid \tilde{S})\big)$$

$$(\boldsymbol{\nu}\tilde{x}, y : \tilde{x} \to y)\ \big((\mathsf{close}[x_i])_i \mid \mathsf{wait}[y]; P\big) \longrightarrow P$$

$$(\boldsymbol{\nu}x, \tilde{y}, \tilde{z} : x \to \tilde{y}.\mathsf{case}(G, H))\ \big(x.\mathsf{inl}; P \mid (y_i.\mathsf{case}(Q_i, R_i))_i \mid \tilde{S}\big) \longrightarrow (\boldsymbol{\nu}x, \tilde{y}, \tilde{z} : G)\ \big(P \mid \tilde{Q} \mid \tilde{S}\big)$$

$$(\boldsymbol{\nu}x, \tilde{y}, \tilde{z} : x \to \tilde{y}.\mathsf{case}(G, H))\ \big(x.\mathsf{inr}; P \mid (y_i.\mathsf{case}(Q_i, R_i))_i \mid \tilde{S}\big) \longrightarrow (\boldsymbol{\nu}x, \tilde{y}, \tilde{z} : H)\ \big(P \mid \tilde{R} \mid \tilde{S}\big)$$

$$(\boldsymbol{\nu}x, \tilde{y}, \tilde{z} : x \to \tilde{y}.\mathsf{case}(G, H))\ \big(x.(\mathsf{inl} : P, \mathsf{inr} : Q) \mid (y_i.\mathsf{case}(R_i, S_i))_i \mid \tilde{T}\big) \longrightarrow (\boldsymbol{\nu}x, \tilde{y}, \tilde{z} : G)\ \big(P \mid \tilde{R} \mid \tilde{T}\big)$$

$$(\boldsymbol{\nu}x, \tilde{y}, \tilde{z} : x \to \tilde{y}.\mathsf{case}(G, H))\ \big(x.(\mathsf{inl} : P, \mathsf{inr} : Q) \mid (y_i.\mathsf{case}(R_i, S_i))_i \mid \tilde{T}\big) \longrightarrow (\boldsymbol{\nu}x, \tilde{y}, \tilde{z} : H)\ \big(Q \mid \tilde{S} \mid \tilde{T}\big)$$

$$(\boldsymbol{\nu}x, \tilde{y} : !x \to \tilde{y}(G))\ \big(\mathsf{use}\, x; P \mid (\mathsf{srv}\, y_i; Q_i)_i\big) \longrightarrow (\boldsymbol{\nu}x, \tilde{y} : G)\ \big(P \mid \tilde{Q}\big)$$

$$(\boldsymbol{\nu}x, \tilde{y} : !x \to \tilde{y}(G))\ \big(\mathsf{kill}\, x \mid P \mid (\mathsf{srv}\, y_i; Q_i)_i\big) \longrightarrow (\mathsf{kill}\, u_j)_j \mid P \quad \text{where } \forall i. \forall v_i \in \mathsf{fv}(Q_i). v_i \neq y_i \Rightarrow \exists j. v_i = u_j$$

$$(\boldsymbol{\nu}x, \tilde{y} : !x \to \tilde{y}(G))\ \big(\mathsf{clone}\, x(x'); P \mid (\mathsf{srv}\, y_i; Q_i)_i\big) \longrightarrow \Big(\mathsf{clone}\, u_j(u_j')\Big)_j ; (\boldsymbol{\nu}x, \tilde{y} : !x \to \tilde{y}(G))\ (\boldsymbol{\nu}x', \tilde{y}' : !x' \to \tilde{y}'(G\{x'/x, \tilde{y}'/\tilde{y}\}))\ \big(P \mid (\mathsf{srv}\, y_i; Q_i)_i \mid (\mathsf{srv}\, y_i'; Q_i')_i\big)$$
$$\text{where } \forall i. \forall v_i \in \mathsf{fv}(Q_i). v_i \neq y_i \Rightarrow \exists j. v_i = u_j$$

$$(\boldsymbol{\nu}x, y : x^X \to y)\ \big(x^X \to w \mid P\big) \longrightarrow P\{w/y\}$$

$$(\boldsymbol{\nu}x, y : x^{X^\perp} \to y)\ \big(w^X \to x \mid P\big) \longrightarrow P\{w/y\}$$

Fig. 4. Semantics for the action fragment.

the *interaction fragment*. Formally, MCC syntax is extended as follows:

$P ::=$	... as in the action fragment ...		$\mid x\,\mathsf{starts}\,\tilde{y}; P$	server accept/request
$\mid z \leftarrow y^B \to x; P$	link		$\mid x\,\mathsf{kills}\,\widetilde{y(Q)}; P$	server kill
$\mid \tilde{x}(\tilde{x}') \to y(y'); P$	communication		$\mid x\,\mathsf{clones}\,\tilde{y}(x', \tilde{y}'); P$	server clone
$\mid \tilde{x}\ \mathsf{closes}\ y; P$	session close			
$\mid x \to \tilde{y}.\mathsf{inl}(P; Q_1, \ldots, Q_n)$	left selection		$\mid x \to \tilde{y}.(\mathsf{inl} : P, \mathsf{inr} : Q)$	general selection
$\mid x \to \tilde{y}.\mathsf{inr}(P_1, \ldots, P_n; Q)$	right selection			

The link term $z \leftarrow y^B \to x; P$ gives the choreographic view of an axiom connected to some other process P through endpoints x and y. A linear interaction $\tilde{x}(\tilde{x}') \to y(y'); P$ denotes a communication from endpoints $\tilde{x}$ to the endpoint y, where a new session with endpoints $\tilde{x}'$,y' is created. The choreography $\tilde{x}\ \mathsf{closes}\ y; P$ closes a session between endpoints $\tilde{x}$, y. When it comes to branching, we have two choreographic terms denoting left and right selection: $x \to \tilde{y}.\mathsf{inl}(P; Q_1, \ldots, Q_n)$ and $x \to \tilde{y}.\mathsf{inr}(P_1, \ldots, P_n; Q)$. A third term, $x \to \tilde{y}.(\mathsf{inl} : P, \mathsf{inr} : Q)$, is used for non-deterministic choice. In MCC, we can model non-linear behaviour: this is done with the terms $x\,\mathsf{starts}\,\tilde{y}; P$, $x\,\mathsf{kills}\,\widetilde{y(Q)}; P$ and $x\,\mathsf{clones}\,\tilde{y}(x', \tilde{y}'); P$. The first term features a client x starting a new session with servers $\tilde{y}$, while the second term is used by endpoint x to shut down servers $\tilde{y}$. Finally, we have a term for cloning servers so that they can be used by different clients in different sessions.

Typing. Fig. 5 details the rules for typing choreography terms. Each of these rules combines two rules from the action fragment simulating their reduction,

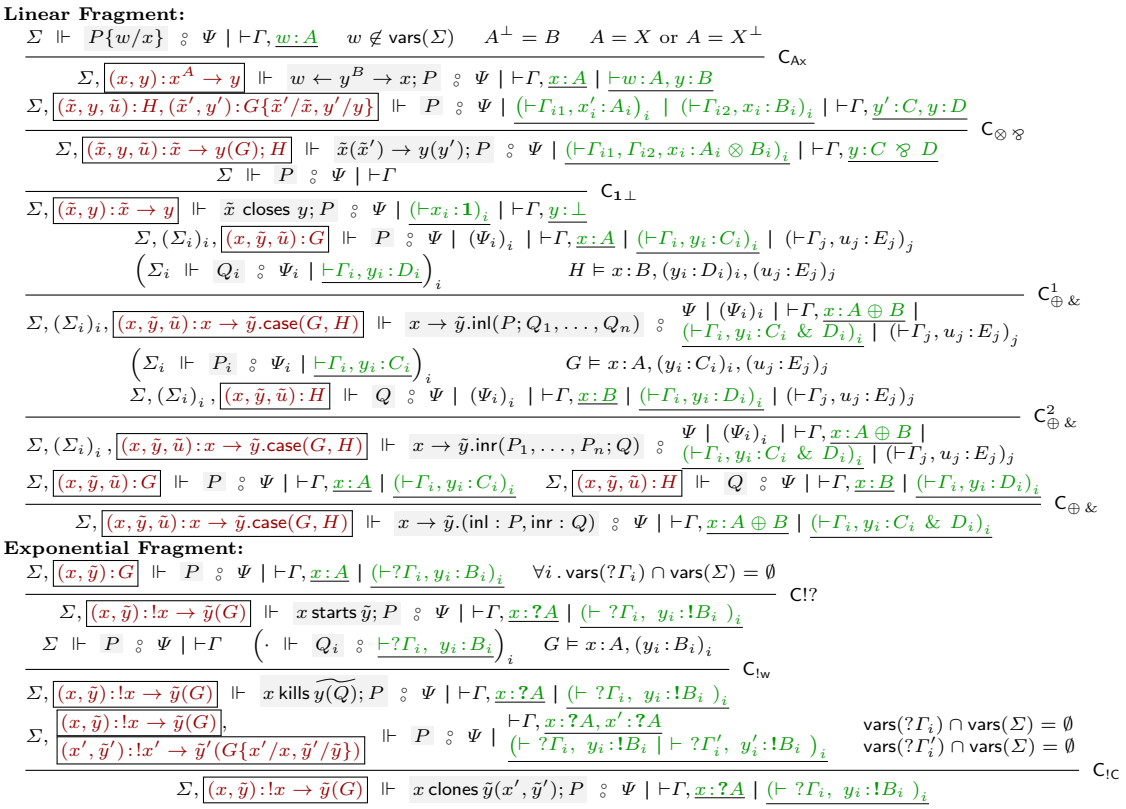

Fig. 5. Rules for the interaction fragment.

where the conclusion of a rule corresponds to the redex and the premise to the reductum. Unlike process rules, the choreography rules now also look at Σ to check that the interactions described conform to the types of the ongoing sessions. In rule $\mathsf{C}_{\mathbf{1}\perp}$, we close a session (removed from Σ) and terminate all processes involved in it. Rule $\mathsf{C}_{\otimes\parr}$ types the creation of a new session with protocol G, created among endpoints $\tilde{z}$ and w; this session is stored in Σ, while the process types are updated as in rules $\otimes$ and $\parr$ above. The remaining rules in the linear fragment are similarly understood. Exponentials give rise to three rules, all of them combining **!** with another rule. In rule C!?, process x invokes the services provided by $\tilde{y}$, creating a new session among these processes with type G. Rule $\mathsf{C}_{!\mathsf{w}}$ combines **!** with Weaken: here the processes providing the service are simply removed from the context. Finally, rule $\mathsf{C}_{!\mathsf{C}}$ combines **!** with Contract, allowing a service to be duplicated.

Reduction Semantics. Fig. 6 gives the reductions for the interaction fragment. From a proof-theoretical perspective, these reductions correspond to proof trans-

$$
\begin{array}{rcll}
(\boldsymbol{\nu} x, y : x^{X} \to y)\,(w \leftarrow y^{X^{\perp}} \to x; P) & \longrightarrow & P\{w/x\} & \\
(\boldsymbol{\nu} x, y : x^{X^{\perp}} \to y)\,(w \leftarrow y^{X} \to x; P) & \longrightarrow & P\{w/x\} & \\
(\boldsymbol{\nu} \tilde{x}, y, \tilde{z} : \tilde{x} \to y(G); H)\,(\tilde{x}(\tilde{x}') \to y(y'); P) & \longrightarrow & (\boldsymbol{\nu} \tilde{x}', y' : G\{\tilde{x}'/\tilde{x}, y'/y\})\,(\boldsymbol{\nu} \tilde{x}, y, \tilde{z} : H)\,P & \\
(\boldsymbol{\nu} \tilde{x}, y : \tilde{x} \to y)\,(\tilde{x}\ \mathsf{closes}\ y; P) & \longrightarrow & P & \\
(\boldsymbol{\nu} x, \tilde{y}, \tilde{z} : x \to \tilde{y}.\mathsf{case}(G, H))\,(x \to \tilde{y}.\mathsf{inl}(P; Q_1, \ldots, Q_n)) & \longrightarrow & (\boldsymbol{\nu} x, \tilde{y}, \tilde{z} : G)\,P & \\
(\boldsymbol{\nu} x, \tilde{y}, \tilde{z} : x \to \tilde{y}.\mathsf{case}(G, H))\,(x \to \tilde{y}.\mathsf{inr}(P_1, \ldots, P_n; Q)) & \longrightarrow & (\boldsymbol{\nu} x, \tilde{y}, \tilde{z} : H)\,Q & \\
(\boldsymbol{\nu} x, \tilde{y}, \tilde{z} : x \to \tilde{y}.\mathsf{case}(G, H))\,(x \to \tilde{y}.(\mathsf{inl} : P, \mathsf{inr} : Q)) & \longrightarrow & (\boldsymbol{\nu} x, \tilde{y}, \tilde{z} : G)\,P & \\
(\boldsymbol{\nu} x, \tilde{y}, \tilde{z} : x \to \tilde{y}.\mathsf{case}(G, H))\,(x \to \tilde{y}.(\mathsf{inl} : P, \mathsf{inr} : Q)) & \longrightarrow & (\boldsymbol{\nu} x, \tilde{y}, \tilde{z} : H)\,Q & \\
(\boldsymbol{\nu} x, \tilde{y} : !x \to \tilde{y}(G))\,(x\ \mathsf{starts}\ \tilde{y}; P) & \longrightarrow & (\boldsymbol{\nu} x, \tilde{y} : G)\,P & \\
(\boldsymbol{\nu} x, \tilde{y} : !x \to \tilde{y}(G))\,\big(x\ \mathsf{kills}\ \widetilde{y(Q)}; P\big) & \longrightarrow & (\mathsf{kill}\,u_j)_j \mid P & (\forall v_i \in \mathsf{fv}(Q_i).v_i \neq y_i \Rightarrow \exists j.v_i = u_j) \\
(\boldsymbol{\nu} x, \tilde{y} : !x \to \tilde{y}(G))\,(x\ \mathsf{clones}\ \tilde{y}(x', \tilde{y}'); P) & \longrightarrow & & \\
(\mathsf{clone}\,u_j(u'_j))_j; (\boldsymbol{\nu} x, \tilde{y} : !x \to \tilde{y}(G))\,(\boldsymbol{\nu} x', \tilde{y}' : !x' \to \tilde{y}'(G\{x'/x, \tilde{y}'/\tilde{y}\}))\,P & & & \text{(see Remark 1)}
\end{array}
$$

Fig. 6. Semantics for the interaction fragment.

formations of C rules from Fig. 5 followed by a structural Scope rule; the transformation removes the C rule and pushes Scope higher up in the proof tree.

Remark 1 (Server Cloning). The reduction rule for a server cloning choreography must clone all of the doubled endpoints. Looking at the typing rule $\mathsf{C}_{!\mathsf{C}}$ on Figure 5, cloned variables u_j are all of the endpoints mentioned in $(?\Gamma_i)_i$, and u'_j are corresponding endpoints from $(?\Gamma'_i)_i$. To make the search for these variables syntactic, one could do an endpoint projection, as described in the next section, and look at the appropriate subterm of the Conn rule which connects y_i and x. The u_j are then the free variables of this subterm, excluding y_i.

Structural equivalence. The reductions given earlier require that programs are written in the very specific form given in their left-hand side. Formally, this is achieved by closing $\longrightarrow$ under structural equivalence: if $P \equiv P'$, $P' \longrightarrow Q'$ and $Q' \equiv Q$, then $P \longrightarrow Q$. The equivalences for the interaction part are given in Fig. 7. As in the action fragment, we are only interested in commuting conversions of typable programs, and therefore rely on typing derivations for finding the correct equivalence. Besides the commuting conversions, we also have the usual structural equivalence rules where parallel composition under restriction, linking process and global type for linking sessions are all symmetric. Furthermore, the order of restrictions can be swapped.

$$
\begin{array}{rcl}
x^{A} \to y & \equiv & y^{A^{\perp}} \to x \\
(\boldsymbol{\nu} \tilde{w}, y, x, \tilde{z} : G)\,\tilde{P} \mid R \mid Q \mid \tilde{S} & \equiv & (\boldsymbol{\nu} \tilde{w}, x, y, \tilde{z} : G)\,\tilde{P} \mid Q \mid R \mid \tilde{S} \\
(\boldsymbol{\nu} z, \tilde{w} : H)\,(\boldsymbol{\nu} x, \tilde{y} : G)\,P \mid \tilde{R} \mid \tilde{Q} & \equiv & (\boldsymbol{\nu} x, \tilde{y} : G)\,(\boldsymbol{\nu} z, \tilde{w} : H)\,P \mid \tilde{Q} \mid \tilde{R}
\end{array}
$$

Properties. We finish the presentation of MCC by establishing the expected meta-theoretic properties of the system. As structural congruence is typing-based, subject congruence is a property holding by construction:

$$
\begin{array}{l}
w \leftarrow y^B \to x; P \mid \tilde{Q} \equiv w \leftarrow y^B \to x; (P \mid \tilde{Q}) \\
\tilde{x}(\tilde{x}') \to y(y'); P \mid \tilde{Q} \equiv \tilde{x}(\tilde{x}') \to y(y'); (P \mid \tilde{Q}) \\
\tilde{x} \text{ closes } y; P \mid \tilde{Q} \equiv \tilde{x} \text{ closes } y; (P \mid \tilde{Q}) \\
x \to \tilde{y}.\mathsf{inl}(P; Q_1, \ldots, Q_n) \mid \tilde{S} \equiv x \to \tilde{y}.\mathsf{inl}((P \mid \tilde{S}); Q_1, \ldots, Q_n) \\
x \to \tilde{y}.\mathsf{inl}(P; Q_1, \ldots, Q_n) \mid \tilde{S} \equiv x \to \tilde{y}.\mathsf{inl}((P \mid \tilde{S}); (Q_1, \ldots, (Q_i \mid \tilde{S}), \ldots, Q_n)) \\
x \to \tilde{y}.\mathsf{inr}(P_1, \ldots, P_n; Q) \mid \tilde{S} \equiv x \to \tilde{y}.\mathsf{inr}(P_1, \ldots, P_n; (Q \mid \tilde{S})) \\
x \to \tilde{y}.\mathsf{inr}(P_1, \ldots, P_n; Q) \mid \tilde{S} \equiv x \to \tilde{y}.\mathsf{inr}((P_1, \ldots, (P_i \mid \tilde{S}), \ldots, P_n); (Q \mid \tilde{S})) \\
x \text{ starts } \tilde{y}; P \mid \tilde{Q} \equiv x \text{ starts } \tilde{y}; (P \mid \tilde{Q}) \\
x \text{ kills } \widetilde{y(Q)}; P \mid \tilde{S} \equiv x \text{ kills } \widetilde{y(Q)}; (P \mid \tilde{S}) \\
x \text{ clones } \tilde{y}(x', \tilde{y}'); P \mid \tilde{Q} \equiv x \text{ clones } \tilde{y}(x', \tilde{y}'); (P \mid \tilde{Q}) \\
(\boldsymbol{\nu} \tilde{w} : G)\, (\tilde{x}(\tilde{x}') \to y(y'); P) \equiv \tilde{x}(\tilde{x}') \to y(y'); (\boldsymbol{\nu} \tilde{w} : G)\, P \\
(\boldsymbol{\nu} \tilde{w} : G)\, (\tilde{x} \text{ closes } y; P) \equiv \tilde{x} \text{ closes } y; (\boldsymbol{\nu} \tilde{w} : G)\, P \\
(\boldsymbol{\nu} \tilde{w} : G)\, (x \to \tilde{y}.\mathsf{inl}(P; Q_1, \ldots, Q_n)) \equiv x \to \tilde{y}.\mathsf{inl}((\boldsymbol{\nu} \tilde{w} : G)\, P; Q_1, \ldots, Q_n) \\
(\boldsymbol{\nu} \tilde{w} : G)\, (x \to \tilde{y}.\mathsf{inl}(P; Q_1, \ldots, Q_n)) \equiv x \to \tilde{y}.\mathsf{inl}((\boldsymbol{\nu} \tilde{w} : G)\, P; Q_1, \ldots, (\boldsymbol{\nu} \tilde{w} : G)\, Q_i, \ldots, Q_n) \\
(\boldsymbol{\nu} \tilde{w} : G)\, (x \to \tilde{y}.\mathsf{inr}(P_1, \ldots, P_n; Q)) \equiv x \to \tilde{y}.\mathsf{inr}(P_1, \ldots, P_n; (\boldsymbol{\nu} \tilde{w} : G)\, Q) \\
(\boldsymbol{\nu} \tilde{w} : G)\, (x \to \tilde{y}.\mathsf{inr}(P_1, \ldots, P_n; Q)) \equiv x \to \tilde{y}.\mathsf{inr}(P_1, \ldots, (\boldsymbol{\nu} \tilde{w} : G)\, P_i, \ldots, P_n; (\boldsymbol{\nu} \tilde{w} : G)\, Q) \\
(\boldsymbol{\nu} \tilde{w} : G)\, (x.(\mathsf{inl} : P, \mathsf{inr} : Q)) \equiv x.(\mathsf{inl} : (\boldsymbol{\nu} \tilde{w} : G)\, P, \mathsf{inr} : (\boldsymbol{\nu} \tilde{w} : G)\, Q) \\
(\boldsymbol{\nu} \tilde{w} : G)\, (x \text{ starts } \tilde{y}; P) \equiv x \text{ starts } \tilde{y}; (\boldsymbol{\nu} \tilde{w} : G)\, P \\
(\boldsymbol{\nu} \tilde{w} : G)\, (x \text{ kills } \widetilde{y(Q)}; P) \equiv x \text{ kills } \widetilde{y_i(Q_i)}; (\boldsymbol{\nu} \tilde{w} : G)\, P \\
(\boldsymbol{\nu} \tilde{w} : G)\, (x \text{ clones } \tilde{y}(x', \tilde{y}'); P) \equiv x \text{ clones } \tilde{y}(x', \tilde{y}'); (\boldsymbol{\nu} \tilde{w} : G)\, P \\
(\boldsymbol{\nu} \tilde{z} : G)\, (x \text{ clones } \tilde{y}(x', \tilde{y}'); P \mid \tilde{Q}) \equiv x \text{ clones } \tilde{y}(x', \tilde{y}'); (\boldsymbol{\nu} \tilde{z} : G)\, (P \mid \tilde{Q})
\end{array}
$$

Fig. 7. Equivalences for commuting C-rules with Conn and Scope. All rules assume that both sides are typable in the same context.

Theorem 1 (Subject Congruence). *$\Sigma \Vdash P \mathrel{:\!:} \Psi$ and $P \equiv Q$ implies that $\Sigma \Vdash Q \mathrel{:\!:} \Psi$.*

Proof. By induction on the proof that $P \equiv Q$. In [5], it is explained how the rules for structural equivalence were derived, making this proof straightforward.

Moreover, our reductions preserve typing since they are proof transformations.

Theorem 2 (Subject Reduction). *$\Sigma \Vdash P \mathrel{:\!:} \Psi$ and $P \longrightarrow Q$ implies $\Sigma \Vdash Q \mathrel{:\!:} \Psi$.*

Proof. By induction on the proof that $P \longrightarrow Q$. In [5], it is explained how the semantics of MCC were designed in order to make this proof straightforward.

Finally, we can show that MCC is deadlock-free, since the top-level Scope application can be pushed up the derivation. In case the top-level Scope application is next to an application of Conn, either the choreography can reduce directly or both rules can be pushed up. Proof-theoretically, this procedure can be viewed as MCC's equivalent of the Principal Lemma of Cut Elimination.

Theorem 3 (Deadlock-freedom). *If P begins with a restriction and $\Sigma \Vdash P \mathrel{:\!:} \Psi$, then there exists Q such that $P \longrightarrow Q$.*

Proof (Sketch). Our proof idea is similar to that of Theorem 3 in [9]. We apply induction on the size of the proof of $\Sigma \Vdash P \mathrel{:\!:} \Psi$. If a rule from Fig. 4 or

$$
\begin{array}{rcll}
P \mid x^X \to y & \rightleftharpoons & x \leftarrow y^{X^\perp} \to w; P & (w,y) \in \mathsf{dom}(\Sigma) \\
P \mid y^X \to x & \rightleftharpoons & x \leftarrow y^X \to w; P & (w,y) \in \mathsf{dom}(\Sigma) \\
\left(\overline{x_i}(x_i'); (P_i \mid Q_i)\right)_i \mid y(y'); R \mid \tilde{S} & \rightleftharpoons & \tilde{x}(\tilde{x}') \to y(y'); \left(\tilde{P} \mid \tilde{Q} \mid R \mid \tilde{S}\right) & \\
(\mathsf{close}[x_i])_i \mid \mathsf{wait}[y]; P & \rightleftharpoons & \tilde{x} \ \mathsf{closes}\ y; P & \\
x.\mathsf{inl}; P \mid (y_i.\mathsf{case}(Q_i, R_i))_i \mid \tilde{S} & \rightleftharpoons & x \to \tilde{y}.\mathsf{inl}(P \mid \tilde{Q} \mid \tilde{S}; R_1, \ldots, R_n) & \\
x.\mathsf{inr}; P \mid (y_i.\mathsf{case}(Q_i, R_i))_i \mid \tilde{S} & \rightleftharpoons & x \to \tilde{y}.\mathsf{inr}(Q_1, \ldots, Q_n; P \mid \tilde{R} \mid \tilde{S}) & \\
x.(\mathsf{inl} : P, \mathsf{inr} : Q) \mid (y_i.\mathsf{case}(R_i, S_i))_i \mid \tilde{T} & \rightleftharpoons & x \to \tilde{y}.(\mathsf{inl} : P \mid \tilde{R} \mid \tilde{T}, \mathsf{inr} : Q \mid \tilde{S} \mid \tilde{T}) & \\
\mathsf{use}\, x; P \mid (\mathsf{srv}\, y_i; Q_i)_i & \rightleftharpoons & x \ \mathsf{starts}\ \tilde{y}; (P \mid \tilde{Q}) & \\
\mathsf{kill}\, x \mid P \mid (\mathsf{srv}\, y_i; Q_i)_i & \rightleftharpoons & x \ \mathsf{kills}\ \widetilde{y(Q)}; P & \\
\mathsf{clone}\, x(x'); P \mid (\mathsf{srv}\, y_i; Q_i)_i & \rightleftharpoons & x \ \mathsf{clones}\ \tilde{y}(x', \tilde{y}'); \left(P \mid (\mathsf{srv}\, y_i; Q_i)_i \mid (\mathsf{srv}\, y_i'; Q_i')_i\right) &
\end{array}
$$

Fig. 8. Extraction ($\rightharpoonup$) and projection ($\leftharpoondown$).

Fig. 6 is applicable (corresponding to a proof where an application of Conn and an application of Scope meet), then the thesis immediately holds.

Otherwise, we apply commuting conversions from Fig. 3 or Fig. 7, "pushing" the top-level Scope application up in the derivation (and, if it is preceded by an application of Conn, "pushing" also that application). This results in a smaller proof of $\Sigma \Vdash P \mathbin{\circ} \Psi$, to which the induction hypothesis can be applied.

5 Projection and Extraction

As suggested by the previous sections, interactions can be implemented in two ways: as a single choreography term, or as multiple process terms appearing in different behaviours composed in parallel. In this section, we formally show that choreography interactions can be projected to process implementations, and symmetrically, process implementations can be extracted to choreographies. We do this by transforming proofs (derivations in the typing system), similarly to the way we defined equivalences and reductions for MCC.

We start by defining the principal transformations for projection and extraction, a set of equivalences that require proof terms to have a special shape. We report such transformations in Fig. 8: they perform extraction if read from left to right, while they perform projection if read from right to left. The extraction relation requires access to the list of open sessions Σ to ensure that we have all the endpoints participating in the session to extract a choreography from. The first two rules deal with axioms: the parallel composition (rule Conn) of an axiom with a process P can be expressed by rule C_{Ax} and vice-versa. On the third line, we show how to transform the parallel composition of an output ($\otimes$) and an input ($\parr$) into a $\mathsf{C}_{\otimes\parr}$. Similarly, $\tilde{x}$ closes $y; P$ is the choreographic representation of the term $(\mathsf{close}[x_i])_i \mid \mathsf{wait}[y]; P$. Each branching operation (left, right, non-deterministic) has a representative in both fragments with straightforward transformations. A server $\mathsf{srv}\, y; Q$ can either be used by a client, killed

or cloned. In the first two cases, such interactions trivially correspond to the choreographic terms $x\,\mathsf{starts}\,\tilde{y};(P \mid \tilde{Q})$ and $x\,\mathsf{kills}\,\widetilde{y(Q)};P$. In the case of cloning, we create the interaction term $x\,\mathsf{clones}\,\tilde{y}(x',\tilde{y}');(P \mid (\mathsf{srv}\,y_i;Q_i)_i \mid (\mathsf{srv}\,y_i';Q_i')_i)$, which shows how the choreographic cloning $x\,\mathsf{clones}\,\tilde{y}(x',\tilde{y}');$ must be followed by two instances of the server that is cloned. Note that these transformations are derived by applying similar techniques as those of cut elimination. Concrete derivations, here omitted, are straightforward: an example can be found in [5].

Remark 2. In order to project/extract an arbitrary well-typed term, given the strict format required by the transformations in Fig. 8, we will sometimes have to perform rewriting of terms in accordance with the commuting conversions to reach an expected shape. In particular, we note that when projecting, we must first project the subterms (we start from the leaves of a proof), step by step moving down to the main term. In contrast, when extracting, we must proceed from the root of the proof towards the leaves.

Note that our example in §2 does not provide an exact projection: in order to improve readability, we have removed all parallels that follow output operations, which would be introduced by the translation presented above. This is not problematic, since the outputs in the example are just basic types.

Properties. In the sequel, we write $P \xrightarrow{\tilde{x}}_{\mathsf{extr}} P'$ whenever it is possible to apply one of the transformations in Fig. 8 to (a term equivalent to) term P from left to right, where $\tilde{x}$ are the endpoints involved in the transformation. Similarly, we write $P \xrightarrow{\tilde{x}}_{\mathsf{proj}} P'$ whenever it is possible to apply a transformation from Fig. 8 to (a term equivalent to) term P from right to left. We also write $P \Longrightarrow_{\mathsf{extr}} P'$ ($P \Longrightarrow_{\mathsf{proj}} P'$) if there is a finite sequence of applications of $\longrightarrow_{\mathsf{extr}}$ ($\longrightarrow_{\mathsf{proj}}$) and P' cannot be further transformed. We then have the following results:

Theorem 4 (Type Preservation). *If $P \xrightarrow{\tilde{x}}_{\mathsf{extr}} Q$ and $\Sigma \Vdash P :: \Psi$, then $\Sigma \Vdash Q :: \Psi$, and if $Q \xrightarrow{\tilde{x}}_{\mathsf{proj}} P$ and $\Sigma \Vdash Q :: \Psi$, then $\Sigma \Vdash P :: \Psi$.*

Proof. By induction on the proof that $P \xrightarrow{\tilde{x}}_{\mathsf{extr}} Q$ or $Q \xrightarrow{\tilde{x}}_{\mathsf{proj}} P$. In [5], we explain how the rules for projection and extraction were derived from the typing rules to ensure that the proof of this result is straightforward.

Theorem 5 (Admissibility of Conn and C-rules). *Let P be a proof term such that $\Vdash P :: \vdash \Gamma$. Then,*

- *there exists P' such that $P \Longrightarrow_{\mathsf{extr}} P'$ and P' is Conn-free;*
- *there exists P' such that $P \Longrightarrow_{\mathsf{proj}} P'$ and P' is free from C-rules.*

Proof (Sketch). The idea is similar to the proof of Theorem 4.4.1 in [17]: by applying commuting conversions we can always rewrite P such that one of the rules in Fig. 8 is applicable, thus eliminating the outermost application of Conn (in the case of extraction) or the innermost application of a C-rule (in the case of projection). See also Remarks 2 and 3.

Remark 3. The theorem above is only applicable to judgments of the form $\Vdash P \mathbin{::} \vdash \Gamma$. This is because of the commuting conversion of the server rule

$$(\boldsymbol{\nu}\tilde{x}x : G)\,(\mathsf{srv}\,y; P \mid \tilde{Q}) \equiv \mathsf{srv}\,y; (\boldsymbol{\nu}\tilde{x}x : G)\,(P \mid \tilde{Q})$$

where we can only permute Conn and Scope together. This conversion is needed to rearrange certain proofs into the format required by the transformations in Fig. 8. Note that any judgement $\Sigma \Vdash P \mathbin{::} \Psi$ can always be transformed into this format, by repeatedly applying rule Scope to all elements in Σ.

As a consequence of the admissibility of Conn, every program can be rewritten into a (non-unique) process containing only process terms by applying the rules in Fig. 8 from right to left until no longer possible. Conversely, because of admissibility of C-rules, every program can be rewritten into a maximal choreographic form by applying the same rules from left to right until no longer possible.

We conclude this section with our main theorem that shows the correspondence between the two fragments with respect to their semantics. In order to do that, we annotate our semantics with the endpoints where the reduction takes place. This is denoted by $P \longrightarrow^{\tilde{x}} Q$ and $P \longrightarrow^{\bullet\tilde{x}} Q$ where the first relation is a reduction in the action fragment, while the second is a reduction in the interaction fragment. The sequence $\mathsf{rev}(\tilde{x})$ is obtained by reversing $\tilde{x}$.

Theorem 6 (Correspondence). *Let P be a proof term such that $\Sigma \Vdash P \mathbin{::} \Psi$. Then,*

- *$P \longrightarrow^{\tilde{x}} Q$ implies that there exists P' s.t. $P \xrightarrow{\tilde{x}}_{\mathsf{extr}} P'$ and $P' \longrightarrow^{\bullet\tilde{x}} Q$;*
- *$P \longrightarrow^{\bullet\tilde{x}} Q$ implies that there exists P' s.t. $P \xrightarrow{\mathsf{rev}(\tilde{x})}_{\mathsf{proj}} P'$ and $P' \longrightarrow^{\tilde{x}} Q$.*

Proof. This proof follows the same strategy as that of Theorem 6 in [9].

6 Related Work and Discussion

Related Work. The principle of choreographies as cut reductions was introduced in [9]. As discussed in §1, that system cannot capture services or multiparty sessions. Another difference is that it is based on intuitionistic linear logic, whereas ours on classical linear logic – in particular, on Classical Processes [22].

Switching to classical linear logic is not a mere change of appearance. It is what allows us to reuse the logical understanding of multiparty sessions in linear logic as *coherence proofs*, introduced in [10] and later extended to polymorphism in [7]. These works did not consider choreographic programs, and thus do not offer a global view on how different sessions are composed, as we do in this paper.

Extracting choreographies from compositions of process code is well-known to be a hard problem. In [15], choreographies that abstract from the exchanged values and computation are extracted from communicating finite-state machines. The authors of [11] present an efficient algorithm for extracting concrete choreographic programs with asynchronous messaging. These works do not consider the

composition of multiple sessions, multiparty sessions, and services, as in MCC. However, they can both deal with infinite behaviour (through loops or recursion), which we do not address. An interesting direction for this feature would be to integrate structural recursion for classical linear logic [16].

Our approach can be seen as a principled reconstruction of previous works on choreographic programming. The first work that typed choreographies using multiparty session types is [8]. The idea of mixing choreographies with processes using multiparty session types is from [19]. None of these consider extraction.

Discussion. For the sake of clarity, our presentation of MCC adopts simplifications that may limit the model expressivity. Below, we discuss some key points as well as possible extensions based on certain developments in this research line.
Non-determinism. We introduced non-determinism in a straightforward way, i.e., our non-deterministic rules in both action and interaction fragments require for each branch to have the same type, as done for standard session typing. However, this solution breaks the property of confluence that we commonly have in logics. In order to preserve confluence, we would have to extend MCC with the non-deterministic linear types from [3].
η-expansion. GCP in [7] allows for the axiom to be of any type A. This requires heavily using η-expansions for transforming axioms into processes with communication actions. It is straightforward to do this in the action fragment of MCC. However, given the way choreographies work, we can only define an axiom for binary sessions in the interaction fragment. As a consequence, in order to apply extraction to a process where an axiom is engaged in a multiparty session with several endpoints, we would need to first use η-expansions to transform such axiom into an ordinary process. In the opposite direction, we would never be able to project a process containing an axiom from a choreography, unless it is part of a binary session. We leave further investigation of this as future work.
Annotated Types. The original version of GCP [7] comes with an extension called MCP, where an endpoint type A is annotated with names of endpoints which it will be in a session with. In this way, endpoint types become more expressive, since it is possible to specify with whom each endpoint has to communicate, without having to use a global type (coherence proof) during execution. We claim that this extension is straightforward for our presentation of MCC.
Polymorphism. As in GCP [7], we can easily add polymorphic types to MCC. However, for simplifying the presentation of this work, we have decided to leave it out, even though adding the GCP rules to the action fragment is straightforward. In the case of the interaction fragment, we obtain the following rule:

$$\frac{\begin{array}{l} X \notin \mathsf{fv}(\Psi, \Gamma, (\Gamma_i)_i) \\ \Sigma, \boxed{(x, \tilde{y}, \tilde{u}) : G\{A/X\}} \Vdash P\{A/X\} \ \circ \ \Psi \mid \vdash \Gamma, \underline{x : B\{A/X\}} \mid \underline{(\vdash \Gamma_i, y_i : B_i\{A/X\})_i} \end{array}}{\Sigma, \boxed{(x, \tilde{y}, \tilde{u}) : x \to \tilde{y}.(X)G} \Vdash x[A] \to \tilde{y}(X); P \ \circ \ \Psi \mid \vdash \Gamma, \underline{x : \exists X.B} \mid \underline{(\vdash \Gamma_i, y_i : \forall X.B_i)_i}} \ \mathsf{C}_{\exists\forall}$$

Above we have added to the syntax of global types the term $x \to \tilde{y}.(X)G$, denoting a session where an endpoint x is supposed to send a type to endpoints $\tilde{y}$. At choreography level, endpoint x realises the abstraction of the global type

sending the actual type A. When it comes to extraction and projection, we would have to add the following transformation:

$$x[A]; P \mid (y_i(X); Q_i)_i \mid \tilde{S} \quad \rightleftharpoons \quad x[A] \to \tilde{y}(X); \left(P \mid \tilde{Q} \mid \tilde{S}\right)$$

where $x[A]; P$ and $y_i(X); Q_i$ are action fragment terms (as those of GCP).

Other Extensions. By importing the functional stratification from [21], we could obtain a monadic integration of choreographies with functions. The calculus of classical higher-order processes [18] could be of inspiration for adding code mobility to MCC, by adding higher-order types. Types for manifest sharing in [2] may lead us to global specifications of sharing in choreographies. And the asynchronous interpretation of cut reductions in [12] might give us an asynchronous implementation of choreographies in MCC. We leave an exploration of these extensions to future work. Hopefully, the shared foundations of linear logic will make it possible to build on these pre-existing technical developments following the same idea of choreographies as cut reductions.

References

1. R. Atkey. Observed communication semantics for classical processes. In H. Yang, editor, *Proc. of ESOP 2017*, volume 10201 of *Lecture Notes in Computer Science*, pages 56–82. Springer, 2017.
2. S. Balzer and F. Pfenning. Manifest sharing with session types. *PACMPL*, 1(ICFP):37:1–37:29, 2017.
3. L. Caires and J. A. Pérez. Linearity, control effects, and behavioral types. In *ESOP*, volume 10201 of *Lecture Notes in Computer Science*, pages 229–259. Springer, 2017.
4. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236, 2010.
5. M. Carbone, L. Cruz-Filipe, F. Montesi, and A. Murawska. Multiparty classical choreographies. *CoRR*, abs/1808.05088, 2018.
6. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM TOPLAS*, 34(2):8, 2012.
7. M. Carbone, S. Lindley, F. Montesi, C. Schürmann, and P. Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
8. M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.
9. M. Carbone, F. Montesi, and C. Schürmann. Choreographies, logically. *Distributed Computing*, 31(1):51–67, 2018.
10. M. Carbone, F. Montesi, C. Schürmann, and N. Yoshida. Multiparty session types as coherence proofs. *Acta Inf.*, 54(3):243–269, 2017. Also: CONCUR 2015.
11. L. Cruz-Filipe, K. S. Larsen, and F. Montesi. The paths to choreography extraction. In *FoSSaCS*, volume 10203 of *Lecture Notes in Computer Science*, pages 424–440, 2017.
12. H. DeYoung, L. Caires, F. Pfenning, and B. Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In *CSL*, volume 16 of *LIPIcs*, pages 228–242. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

13. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.
14. JBoss Community and Red Hat. Testable Architecture. http://www.jboss.org/savara/.
15. J. Lange, E. Tuosto, and N. Yoshida. From communicating machines to graphical choreographies. In *POPL*, pages 221–232. ACM, 2015.
16. S. Lindley and J. G. Morris. Talking bananas: structural recursion for session types. In *ICFP*, pages 434–447. ACM, 2016.
17. F. Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. http://www.itu.dk/people/fabr/papers/phd/thesis.pdf.
18. F. Montesi. Classical higher-order processes - (short paper). In *FORTE*, volume 10321 of *Lecture Notes in Computer Science*, pages 171–178. Springer, 2017.
19. F. Montesi and N. Yoshida. Compositional choreographies. In *CONCUR*, pages 425–439, 2013.
20. OpenID. OpenID specifications. http://openid.net/developers/specs/.
21. B. Toninho, L. Caires, and F. Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, pages 350–369, 2013.
22. P. Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.