A Core Model for Choreographic Programming

Luís Cruz-Filipe and Fabrizio Montesi

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

Abstract. Choreographic Programming is a paradigm for developing concurrent programs that are deadlock-free by construction, by programming communications declaratively and then synthesising process implementations automatically. Despite strong interest on choreographies, a foundational model that explains which computations can be performed with the hallmark constructs of choreographies is still missing.
In this work, we introduce Core Choreographies (CC), a model that includes only the core primitives of choreographic programming. Every computable function can be implemented as a choreography in CC, from which we can synthesise a process implementation where independent computations run in parallel. We discuss the design of CC and argue that it constitutes a canonical model for choreographic programming.

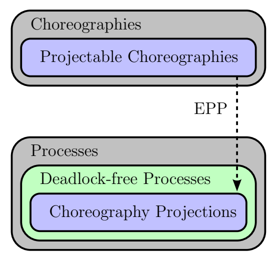
1 Introduction

Programming concurrent and distributed systems is hard, because it is challenging to predict how programs executed at the same time in different computers will interact. Empirical studies reveal two important lessons: (i) while programmers have clear intentions about the order in which communication actions should be performed, tools do not adequately support them in translating these wishes to code [21]; (ii) combining different communication protocols in a single application is a major source of mistakes [20].

The paradigm of Choreographic Programming [22] was introduced to address these problems. In this paradigm, programmers declaratively write the communications that they wish to take place, as programs called *choreographies*. Choreographies are descriptions of concurrent systems that syntactically disallow writing mismatched I/O actions, inspired by the "Alice and Bob" notation of security protocols. An EndPoint Projection (EPP) can then be used to synthesise implementations in process models, which faithfully realise the communications given in the choreography and are guaranteed to be deadlock-free by construction even in the presence of arbitrary protocol compositions [6, 25].

So far, work on choreographic programming focused on features of practical value – including web services [5], multiparty sessions [6, 8], modularity [24], and runtime adaptation [12]. The models proposed all come with differing domain-specific syntaxes, semantics and EPP definitions (e.g., for channel mobility or runtime adaptation), and cannot be considered minimal. Another problem, arguably a consequence of the former, is that choreographic programming is meant for implementation, but we still know little of what can be computed with the code obtained from choreographies (*choreography projections*). The expressivity of the aforementioned models is evaluated just by showing some examples.

In this paper, we propose a canonical model for choreographic programming, called Core Choreographies (CC). CC includes only the core primitives that can be found in most choreography languages, restricted to the minimal requirements to achieve the computational power of Turing machines. In particular, local computation at processes is severely restricted, and therefore nontrivial computations must be implemented by using communications. Therefore, CC is both representative of the paradigm and simple enough to analyse from a theoretical perspective. Our technical development is based on a natural notion of function implementation, and the proof of Turing completeness yields an algorithm for constructing a choreography that implements any given computable function. Since choreographies describe concurrent systems, it is also natural to ask what degree of parallelism choreographies exhibit. CC helps us in formally defining parallelism in choreographies; we exemplify how to use this notion to reason about the concurrent implementation of functions.



However, analysing the expressivity of choreographies is not enough. What we are ultimately interested in is what can be computed with choreography projections, since those are the terms that represent executable code. However, the expressivity of choreographies does not translate directly to expressivity of projections, because EPP is typically an incomplete procedure: it must guarantee deadlock-freedom, which in previous models is obtained by complex requirements, e.g., type systems [5, 6]. Therefore, only a subset of choreographies (projectable choreographies) can be used to synthesise process implementations. The EPPs of such projectable choreographies form the set of choreography projections, which are deadlock-free processes (see figure on the right).

The main technical contribution of this paper is showing that the set of projectable choreographies in CC is still Turing complete. Therefore, by EPP, the set of corresponding choreography projections is also Turing complete, leading us to a characterisation of a Turing complete and deadlock-free fragment of a process calculus (which follows the same minimal design of CC). Furthermore, the parallel behaviour observed in CC choreographies for function implementations translates directly to parallel execution of the projected processes.

More importantly, the practical consequence of our results is the identification of CC as a simple common setting for the formal study of foundational questions in choreographies. This makes CC an appropriate foundational model for choreographic programming, akin to λ-calculus for functional programming and π-calculus for mobile processes. As an example of such foundational questions, we describe how the standard communication primitive of label selection can be removed from CC without altering its computational power, yielding a truly minimal choreography language wrt computation called Minimal Choreographies (MC). However, doing so eliminates the clean separation between data and behaviour in message exchanges, which makes the resulting choreography hard to read. Thus, in a practical application of our work, CC would be the better candidate as frontend language for programmers, and MC could be used as an intermediate step in a compiler. A key technical advantage of this methodology is that it bypasses the need for the standard notion of merging [5], which is typically one of the most complicated steps in EPP. Our EPP for MC enjoys an elegant definition.

Structure of the paper. CC is defined in § 2. In § 3, we introduce Stateful Processes (SP), our target process model, and an EPP procedure from CC to SP. We show that CC and its set of choreography projections are Turing complete in § 4. In § 5, we show that all primitives of CC except for label selections are necessary to achieve Turing completeness; we then introduce MC (the fragment of CC without label selections) and prove both that it is Turing complete and that removing or weakening any of its primitives breaks this property. In § 6, we discuss the implications of our work for other choreography languages. Related work and discussion are given in § 7. Full definitions and proofs are given in the Technical Report at http://www.fabriziomontesi.com/files/cc-full.pdf.

2 Core Choreographies

We introduce Core Choreographies (CC), define function implementation and parallel execution of choreographies, and prove some key properties of CC.

Syntax. The syntax of CC is as follows, where C ranges over choreographies.

$$C ::= \eta; C \mid \mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1 \mid X \mid \mathbf{0}$$
$$\eta ::= \mathsf{p}.e\ \texttt{->}\ \mathsf{q} \mid \mathsf{p}\ \texttt{->}\ \mathsf{q}[l] \qquad e ::= \varepsilon \mid \mathsf{c} \mid \mathsf{s} \cdot \mathsf{c} \qquad l ::= \textsc{l} \mid \textsc{r}$$

We use two (infinite) disjoint sets of names: processes ($\mathsf{p}, \mathsf{q}, \ldots$) and procedures ($X, \ldots$). Processes run in parallel, and each process stores a value – a string of the form $\mathsf{s} \cdots \mathsf{s} \cdot \varepsilon$ – in a local memory cell. Each process can access its own value, but it cannot read the contents of another process (no data sharing). Term $\eta; C$ is an interaction between two processes, read "the system may execute η and proceed as C". An interaction η is either a value communication – $\mathsf{p}.e\ \texttt{->}\ \mathsf{q}$ – or a label selection – $\mathsf{p}\ \texttt{->}\ \mathsf{q}[l]$. In $\mathsf{p}.e\ \texttt{->}\ \mathsf{q}$, p sends its local evaluation of expression e to q, which stores the received value. Expressions are either the constant ε, the value of the sender (written as c), or an application of the successor operator to c. In $\mathsf{p}\ \texttt{->}\ \mathsf{q}[l]$, p communicates label l (either $\textsc{l}$ or $\textsc{r}$) to q. In a conditional $\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, q sends its value to p, which checks if the received value is equal to its own; the choreography proceeds as C_1, if that is the case, or as C_2, otherwise. In value communications, selections and conditionals, the two interacting processes must be different (no self-communications). Definitions and invocations of recursive procedures are standard. The term $\mathbf{0}$, also called *exit point*, is the terminated choreography.

Semantics. The semantics of CC uses reductions of the form $C, \sigma \to C', \sigma'$. The total state function σ maps each process name to its value. We use v, w, ... to range over values: $v, w, \ldots ::= \varepsilon \mid \mathsf{s} \cdot v$. Values are isomorphic to natural numbers via $\ulcorner n \urcorner = \mathsf{s}^n \cdot \varepsilon$. The reduction relation $\to$ is defined by the rules given below and closed under structural precongruence $\preceq$.

$$\frac{v = e[\sigma(\mathsf{p})/\mathsf{c}]}{\mathsf{p}.e \texttt{ -> } \mathsf{q}; C, \sigma \to C, \sigma[\mathsf{q} \mapsto v]} \lfloor\text{C}|\text{Com}\rceil \qquad \frac{i = 1 \text{ if } \sigma(\mathsf{p}) = \sigma(\mathsf{q}),\ i = 2 \text{ o.w.}}{\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \sigma \to C_i, \sigma} \lfloor\text{C}|\text{Cond}\rceil$$

$$\frac{}{\mathsf{p} \texttt{ -> } \mathsf{q}[l]; C, \sigma \to C, \sigma} \lfloor\text{C}|\text{Sel}\rceil \qquad \frac{C_1, \sigma \ \to\ C_1', \sigma'}{\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1, \sigma \ \to\ \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1', \sigma'} \lfloor\text{C}|\text{Ctx}\rceil$$

These rules formalise the intuition presented earlier. In the premise of $\lfloor\text{C}|\text{Com}\rceil$, we write $e[\sigma(\mathsf{p})/\mathsf{c}]$ for the result of replacing c with $\sigma(\mathsf{p})$ in e. In the reductum, $\sigma[\mathsf{q} \mapsto v]$ denotes the updated state function σ where q now maps to v. The key rule defining the structural precongruence is $\lfloor\text{C}|\text{Eta-Eta}\rceil$, allowing non-interfering actions to be executed in any order.

$$\lfloor\text{C}|\text{Eta-Eta}\rceil \qquad \text{if} \quad \mathsf{pn}(\eta) \cap \mathsf{pn}(\eta') = \emptyset \quad \text{then} \quad \eta; \eta' \ \equiv\ \eta'; \eta$$

Function $\mathsf{pn}(C)$ returns the set of all process names occurring in C, and $C \equiv C'$ stands for $C \preceq C'$ and $C' \preceq C$. The other rules for $\preceq$ are standard, and support recursion unfolding and garbage collection of unused definitions.

Remark 1 (Label Selection). To the reader unfamiliar with choreographies, the role of selection – $\mathsf{p} \texttt{ -> } \mathsf{q}[l]$ – may be unclear at this point. They are crucial in making choreographies projectable, as we anticipate with the choreography $\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ (\mathsf{p.c} \texttt{ -> } \mathsf{r}; \mathbf{0})\ \mathsf{else}\ (\mathsf{r.c} \texttt{ -> } \mathsf{p}; \mathbf{0})$. Here, p checks whether its value is the same as that of q. If so, p communicates its value to r; otherwise, it is r that communicates its value to p. Recall that processes are assumed to run independently and share no data. Here, p is the only process that knows which branch of the conditional should be executed. However, r also needs to know this information, since it must behave differently. Intuitively, we need to propagate p's decision to r, which is achieved with selections: $\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ (\mathsf{p} \texttt{ -> } \mathsf{r}[\text{L}]; \mathsf{p.c} \texttt{ -> } \mathsf{r}; \mathbf{0})\ \mathsf{else}\ (\mathsf{p} \texttt{ -> } \mathsf{r}[\text{R}]; \mathsf{r.c} \texttt{ -> } \mathsf{p}; \mathbf{0})$. Now, p tells r about its choice by sending a different label. This intuition will be formalised in our definition of EndPoint Projection in § 3. The first choreography we presented (without label selections) is not projectable, whereas the second one is.

CC enjoys the usual deadlock-freedom-by-design property of choreographies.

Theorem 1. *If C is a choreography, then either $C \preceq \mathbf{0}$ (C has terminated) or, for all σ, $C, \sigma \to C', \sigma'$ for some C' and σ' (C can reduce).*

The semantics of CC suggests a natural definition of computation. We write $\to^*$ for the transitive closure of $\to$ and $C, \sigma \not\to^* \mathbf{0}$ for $C, \sigma \not\to^* \mathbf{0}, \sigma'$ for any σ'.

Definition 1. *A choreography C* implements *a function $f : \mathbb{N}^n \to \mathbb{N}$ with input processes $\mathsf{p}_1, \ldots, \mathsf{p}_n$ and output process q if, for all $x_1, \ldots, x_n \in \mathbb{N}$ and for every state σ s.t. $\sigma(\mathsf{p}_i) = \ulcorner x_i \urcorner$:*

– *if* $f(\tilde{x})$ *is defined, then* $C, \sigma \to^* \mathbf{0}, \sigma'$ *where* $\sigma'(\mathsf{q}) = \ulcorner f(\tilde{x}) \urcorner$*;*
– *if* $f(\tilde{x})$ *is undefined, then* $C, \sigma \not\to^* \mathbf{0}$*.*

By Theorem 1, in the second case C, σ must reduce infinitely (diverge).

Sequential composition and parallelism. We are mostly interested in choreographies with only one exit point ($\mathbf{0}$). When a choreography C has a single exit point, we write $C \fatsemi C'$ for the choreography obtained by replacing $\mathbf{0}$ in C with C'. Then, $C \fatsemi C'$ behaves as a "sequential composition" of C and C'.

Lemma 1. *Let* C *have one exit point,* C' *be a choreography,* $\sigma, \sigma', \sigma''$ *be states.*

1. *If* $C, \sigma \to^* \mathbf{0}, \sigma'$ *and* $C', \sigma' \to^* \mathbf{0}, \sigma''$*, then* $C \fatsemi C', \sigma \to^* \mathbf{0}, \sigma''$*.*
2. *If* $C, \sigma \not\to^* \mathbf{0}$*, then* $C \fatsemi C', \sigma \not\to^* \mathbf{0}$*.*
3. *If* $C, \sigma \to^* \mathbf{0}, \sigma'$ *and* $C', \sigma' \not\to^* \mathbf{0}$*, then* $C \fatsemi C', \sigma \not\to^* \mathbf{0}$*.*

Structural precongruence gives $C \fatsemi C'$ fully parallel behaviour in some cases. Intuitively, C_1 and C_2 run in parallel in $C_1 \fatsemi C_2$ if their reduction paths to $\mathbf{0}$ can be interleaved in any possible way. Below, we write $C \xrightarrow{\tilde{\sigma}}^* \mathbf{0}$ for $C, \sigma_1 \to C_2, \sigma_2 \to \cdots \to \mathbf{0}, \sigma_n$, where $\tilde{\sigma} = \sigma_1, \ldots, \sigma_n$, and $\widetilde{\sigma(\mathsf{p})}$ for the sequence $\sigma_1(\mathsf{p}), \ldots, \sigma_n(\mathsf{p})$.

Definition 2. *Let* $\tilde{\mathsf{p}}$ *and* $\tilde{\mathsf{q}}$ *be disjoint. Then,* $\tilde{\sigma}$ *is an* interleaving *of* $\widetilde{\sigma_1}$ *and* $\widetilde{\sigma_2}$ *wrt* $\tilde{\mathsf{p}}$ *and* $\tilde{\mathsf{q}}$ *if* $\tilde{\sigma}$ *contains two subsequences* $\widetilde{\sigma'_1}$ *and* $\widetilde{\sigma'_2}$ *such that:*

– $\widetilde{\sigma'_2} = \tilde{\sigma} \setminus \widetilde{\sigma'_1}$*;*
– $\widetilde{\sigma'_1(\mathsf{p})} = \widetilde{\sigma_1(\mathsf{p})}$ *for all* $\mathsf{p} \in \tilde{\mathsf{p}}$*, and* $\widetilde{\sigma'_2(\mathsf{q})} = \widetilde{\sigma_2(\mathsf{q})}$ *for all* $\mathsf{q} \in \tilde{\mathsf{q}}$*;*
– $\widetilde{\sigma(\mathsf{r})}$ *is constant for all* $\mathsf{r} \notin \tilde{\mathsf{p}} \cup \tilde{\mathsf{q}}$*.*

Definition 3. *Let* C_1 *and* C_2 *be choreographies such that* $\mathsf{pn}(C_1) \cap \mathsf{pn}(C_2) = \emptyset$ *and* C_1 *has only one exit point. We say that* C_1 *and* C_2 run in parallel *in* $C_1 \fatsemi C_2$ *if: whenever* $C_i \xrightarrow{\tilde{\sigma_i}}^* \mathbf{0}$*, then* $C_1 \fatsemi C_2 \xrightarrow{\tilde{\sigma}}^* \mathbf{0}$ *for every interleaving* $\tilde{\sigma}$ *of* $\tilde{\sigma_1}$ *and* $\tilde{\sigma_2}$ *wrt* $\mathsf{pn}(C_1)$ *and* $\mathsf{pn}(C_2)$*.*

Theorem 2. *Let* C_1 *and* C_2 *be choreographies such that* $\mathsf{pn}(C_1) \cap \mathsf{pn}(C_2) = \emptyset$ *and* C_1 *has only one exit point. Then* C_1 *and* C_2 *run in parallel in* $C_1 \fatsemi C_2$*.*

Example 1. We present examples of choreographies in CC, writing them as macros (syntax shortcuts). We use the notation $\textsc{m}(\textit{params}) \triangleq C$, where $\textsc{m}$ is the name of the macro, *params* its parameters, and C its body.

The macro $\textsc{inc}(\mathsf{p}, \mathsf{t})$ increments the value of p using an auxiliary process t.

$$\textsc{inc}(\mathsf{p}, \mathsf{t}) \quad \triangleq \quad \mathsf{p.c \text{ -> } t;\ t.(s \cdot c) \text{ -> } p;}\ \mathbf{0}$$

Using $\textsc{inc}$, we write a macro $\textsc{add}(\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{t}_1, \mathsf{t}_2)$ that adds the values of p and q and stores the result in p, using auxiliary processes r, t_1 and t_2. We follow the intuition as in low-level abstract register machines. First, t_1 sets the value of r to zero, and then calls procedure X, which increments the value of p as many times as the value in q. In the body of X, r checks whether its value is the same as q's.

If so, it informs the other processes that the recursion will terminate (selection of L); otherwise, it asks them to do another step (selection of R). In each step, the values of p and r are incremented using t_1 and t_2. The compositional usage of INC is allowed, as it has exactly one exit point.

$$
\begin{aligned}
&\textsc{add}(\mathsf{p},\mathsf{q},\mathsf{r},\mathsf{t}_1,\mathsf{t}_2) \stackrel{\Delta}{=} \\
&\quad \mathsf{def}\, X = \mathsf{if}\ \mathsf{r} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ \mathsf{r} \texttt{ -> } \mathsf{p}[\mathrm{L}]; \mathsf{r} \texttt{ -> } \mathsf{q}[\mathrm{L}]; \mathsf{r} \texttt{ -> } \mathsf{t}_1[\mathrm{L}]; \mathsf{r} \texttt{ -> } \mathsf{t}_2[\mathrm{L}]; \mathbf{0} \\
&\qquad\qquad \mathsf{else}\ \mathsf{r} \texttt{ -> } \mathsf{p}[\mathrm{R}]; \mathsf{r} \texttt{ -> } \mathsf{q}[\mathrm{R}]; \mathsf{r} \texttt{ -> } \mathsf{t}_1[\mathrm{R}]; \mathsf{r} \texttt{ -> } \mathsf{t}_2[\mathrm{R}]; \textsc{inc}(\mathsf{p},\mathsf{t}_1) \fatsemi \textsc{inc}(\mathsf{r},\mathsf{t}_2) \fatsemi X \\
&\quad \mathsf{in}\ \mathsf{t}_1.\varepsilon \texttt{ -> } \mathsf{r}; X
\end{aligned}
$$

By Theorem 2, the calls to $\textsc{inc}(\mathsf{p},\mathsf{t}_1)$ and $\textsc{inc}(\mathsf{r},\mathsf{t}_2)$ can be executed in parallel. Indeed, applying rule ⌊C|Eta-Eta⌉ for $\preceq$ repeatedly we can check that:

$$
\begin{aligned}
&\underbrace{\mathsf{p.c} \texttt{ -> } \mathsf{t}_1;\ \mathsf{t}_1.(\mathsf{s}\cdot\mathsf{c}) \texttt{ -> } \mathsf{p};}_{\text{expansion of } \textsc{inc}(\mathsf{p},\mathsf{t}_1)} \underbrace{\mathsf{r.c} \texttt{ -> } \mathsf{t}_2;\ \mathsf{t}_2.(\mathsf{s}\cdot\mathsf{c}) \texttt{ -> } \mathsf{r};}_{\text{expansion of } \textsc{inc}(\mathsf{r},\mathsf{t}_2)} X \\
&\qquad\qquad\qquad \preceq\ \underbrace{\mathsf{r.c} \texttt{ -> } \mathsf{t}_2;\ \mathsf{t}_2.(\mathsf{s}\cdot\mathsf{c}) \texttt{ -> } \mathsf{r};}_{\text{expansion of } \textsc{inc}(\mathsf{r},\mathsf{t}_2)} \underbrace{\mathsf{p.c} \texttt{ -> } \mathsf{t}_1;\ \mathsf{t}_1.(\mathsf{s}\cdot\mathsf{c}) \texttt{ -> } \mathsf{p};}_{\text{expansion of } \textsc{inc}(\mathsf{p},\mathsf{t}_1)} X
\end{aligned}
$$

3 Stateful Processes and EndPoint Projection

We present Stateful Processes (SP), our target process model, and show how to synthesise process implementations from choreographies in CC.

Syntax. The syntax of SP is reported below. Networks (N, M) are either the inactive network $\mathbf{0}$ or parallel compositions of processes $\mathsf{p} \triangleright_v B$, where p is the name of the process, v its stored value, and B its behaviour.

$$
\begin{aligned}
B ::= {} & \mathsf{q}!\langle e\rangle; B \ \mid\ \mathsf{p}?; B \ \mid\ \mathsf{q} \oplus l; B \ \mid \mathsf{p}\&\{l_i : B_i\}_{i\in I} \ \mid \qquad N, M ::= \mathsf{p} \triangleright_v B \ \mid\ \mathbf{0} \ \mid\ N \mid M \\
& \mid \mathbf{0} \mid \mathsf{if}\ \mathsf{c} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2 \ \mid\ \mathsf{def}\ X = B_2\ \mathsf{in}\ B_1 \ \mid\ X
\end{aligned}
$$

Expressions and labels are as in CC. A send term $\mathsf{q}!\langle e\rangle; B$ sends the evaluation of expression e to q, proceeding as B. Term $\mathsf{p}?; B$, the dual receiving action, stores the value received from p in the process executing the behaviour, proceeding as B. A selection term $\mathsf{q} \oplus l; B$ sends l to q. Dually, a branching term $\mathsf{p}\&\{l_i : B_i\}_{i\in I}$ receives one of the labels l_i and proceeds as B_i. A process offers either: a single branch (labeled L or R); or two branches (with distinct labels). In a conditional $\mathsf{if}\ \mathsf{c} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2$, the process receives a value from process q and compares it with its own value to choose the continuation B_1 or B_2. The other terms (definition/invocation of recursive procedures, termination) are standard.

Semantics. The reduction rules for SP are mostly standard, from process calculi. The key difference from CC is that execution is now distributed over processes.

We report the key rules for synchronisation:

$$\frac{u = e[v/\mathsf{c}]}{\mathsf{p} \triangleright_v \mathsf{q}!\langle e\rangle; B_1 \;\mid\; \mathsf{q} \triangleright_w \mathsf{p}?; B_2 \;\to\; \mathsf{p} \triangleright_v B_1 \;\mid\; \mathsf{q} \triangleright_u B_2} \;\lfloor\text{S}|\text{Com}\rceil$$

$$\frac{j \in I}{\mathsf{p} \triangleright_v \mathsf{q} \oplus l_j; B \;\mid\; \mathsf{q} \triangleright_w \mathsf{p}\&\{l_i : B_i\}_{i\in I} \;\to\; \mathsf{p} \triangleright_v B \;\mid\; \mathsf{q} \triangleright_w B_j} \;\lfloor\text{S}|\text{Sel}\rceil$$

$$\frac{i = 1 \text{ if } v = e[w/\mathsf{c}], \quad i = 2 \text{ otherwise}}{\mathsf{p} \triangleright_v \mathsf{if}\ \mathsf{c} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2 \;\mid\; \mathsf{q} \triangleright_w \mathsf{p}!\langle e\rangle; B' \;\to\; \mathsf{p} \triangleright_v B_i \;\mid\; \mathsf{q} \triangleright_w B'} \;\lfloor\text{S}|\text{Cond}\rceil$$

Rule $\lfloor$S|Com$\rceil$ follows the standard communication rule in process calculi. A process p executing a send action towards a process q can synchronise with a receive-from-p action at q; in the reduct, q's value is updated with the value sent by p, obtained by replacing the placeholder c in e with the value of p. Rule $\lfloor$S|Sel$\rceil$ is selection from session types [15], with the sender selecting one of the branches offered by the receiver. In rule $\lfloor$S|Cond$\rceil$, p (executing the conditional) acts as a receiver for the value sent by the process whose value it wants to read (q). All other rules are standard (see the Technical Report), and use a structural precongruence that supports: recursion unfolding, garbage collection of terminated processes and unused definitions, and associativity and commutativity of parallel composition.

As for CC, we can define function implementation in SP.

Definition 4. *A network N* implements *a function $f : \mathbb{N}^n \to \mathbb{N}$ with input processes $\mathsf{p}_1, \ldots, \mathsf{p}_n$ and output process q if $N \preceq (\prod_{i\in[1,n]} \mathsf{p}_i \triangleright_{v_i} B_i) \mid \mathsf{q} \triangleright_w B' \mid N'$ and, for all $x_1, \ldots, x_n \in \mathbb{N}$:*

- *if $f(\tilde{x})$ is defined, then $N(\tilde{x}) \to^* \mathsf{q} \triangleright_{\ulcorner f(\tilde{x})\urcorner} \mathbf{0}$;*
- *if $f(\tilde{x})$ is not defined, then $N(\tilde{x}) \not\to^* \mathbf{0}$.*

where $N(\tilde{x})$ is a shorthand for $N[\widetilde{\ulcorner x_i \urcorner / v_i}]$, the network obtained by replacing in N the values of the input processes with the arguments of the function.

Projection. We now define an EndPoint Projection (EPP) from CC to SP.

We first discuss the rules for projecting the behaviour of a single process p, a partial function $[\![C]\!]_{\mathsf{p}}$ defined by the rules in Figure 1. All rules follow the intuition of projecting, for each choreography term, the local action performed by the process that we are projecting. For example, for a communication term $\mathsf{p}.e$ `->` q, we project a send action for the sender p, a receive action for the receiver q, or just the continuation otherwise. The rule for selection is similar. The rules for projecting recursive definitions and calls assume that procedure names have been annotated with the process names appearing inside the body of the procedure, in order to avoid projecting unnecessary procedure code (see [5]).

The rule for projecting a conditional is more involved, using the partial merging operator $\sqcup$ to merge the possible behaviours of a process that does not know which branch will be chosen. Merging is a homomorphic binary operator; for all terms but branchings it requires isomorphism, e.g.: $\mathsf{q}!\langle e\rangle; B \sqcup \mathsf{q}!\langle e\rangle; B' =$

$$[\![\mathsf{p}.e \text{ -> } \mathsf{q}; C]\!]_{\mathsf{r}} = \begin{cases} \mathsf{q}!\langle e\rangle; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}?; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_{\mathsf{r}} & \text{o.w.} \end{cases} \qquad [\![\mathsf{p} \text{ -> } \mathsf{q}[l]; C]\!]_{\mathsf{r}} = \begin{cases} \mathsf{q} \oplus l; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}\&\{l : [\![C]\!]_{\mathsf{r}}\} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_{\mathsf{r}} & \text{o.w.} \end{cases}$$

$$[\![\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!]_{\mathsf{r}} = \begin{cases} \mathsf{if}\ \mathsf{c} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ [\![C_1]\!]_{\mathsf{r}}\ \mathsf{else}\ [\![C_2]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}!\langle \mathsf{c}\rangle; ([\![C_1]\!]_{\mathsf{r}} \sqcup [\![C_2]\!]_{\mathsf{r}}) & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C_1]\!]_{\mathsf{r}} \sqcup [\![C_2]\!]_{\mathsf{r}} & \text{o.w.} \end{cases} \qquad [\![\mathbf{0}]\!]_{\mathsf{r}} = \mathbf{0}$$

$$[\![\mathsf{def}\ X^{\tilde{\mathsf{p}}} = C_2\ \mathsf{in}\ C_1]\!]_{\mathsf{r}} = \begin{cases} \mathsf{def}\ X = [\![C_2]\!]_{\mathsf{r}}\ \mathsf{in}\ [\![C_1]\!]_{\mathsf{r}} & \text{if } \mathsf{r} \in \tilde{\mathsf{p}} \\ [\![C_1]\!]_{\mathsf{r}} & \text{o.w.} \end{cases} \qquad [\![X^{\tilde{\mathsf{p}}}]\!]_{\mathsf{r}} = \begin{cases} X & \text{if } \mathsf{r} \in \tilde{\mathsf{p}} \\ \mathbf{0} & \text{o.w.} \end{cases}$$

Fig. 1. Minimal Choreographies, Behaviour Projection.

$\mathsf{q}!\langle e\rangle; (B \sqcup B')$. Branching terms can have unmergeable continuations, as long as they are guarded by distinct labels. In this case, merge returns a larger branching including all options (merging branches with the same label):

$$\mathsf{p}\&\{l_i : B_i\}_{i\in J} \sqcup \mathsf{p}\&\{l_i : B'_i\}_{i\in K} = \\ \mathsf{p}\&\big(\{l_i : (B_i \sqcup B'_i)\}_{i\in J\cap K} \cup \{l_i : B_i\}_{i\in J\setminus K} \cup \{l_i : B'_i\}_{i\in K\setminus J}\big)$$

Merging explains the role of selections in CC, common in choreography models [2, 5, 6, 16, 12, 25]. Recall the choreographies from Remark 1. In the first one, the behaviour of r cannot be projected because we cannot merge its different behaviours in the two branches of the conditional (a send with a receive). The second one is projectable, and the behaviour of r is $[\![C]\!]_{\mathsf{r}} = \mathsf{p}\&\{\textsc{l} : \mathsf{p}?; \mathbf{0},\ \textsc{r} : \mathsf{p}!\langle \mathsf{c}\rangle; \mathbf{0}\}$.

Definition 5. *Given a choreography C and a state σ, the* endpoint projection *of C and σ is the parallel composition of the projections of the processes in C: $[\![C, \sigma]\!] = \prod_{\mathsf{p}\in\mathsf{pn}(C)} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} [\![C]\!]_{\mathsf{p}}$.*

Since the σs are total, $[\![C, \sigma]\!]$ is defined for some σ iff $[\![C, \sigma']\!]$ is defined for all other σ'. In this case, we say that C is *projectable.*

EPP guarantees the following operational correspondence.

Theorem 3. *Let C be a projectable choreography. Then, for all σ:*

Completeness: *If $C, \sigma \to C', \sigma'$, then $[\![C, \sigma]\!] \to\succ [\![C', \sigma']\!]$;*
Soundness: *If $[\![C, \sigma]\!] \to N$, then $C, \sigma \to C', \sigma'$ for some σ', with $[\![C', \sigma']\!] \prec N$.*

The *pruning relation* $\prec$ [5, 6] deletes branches introduced by merging when no longer needed; $N \succ N'$ means $N' \prec N$. Pruning does not alter the behaviour of a network: eliminated branches are never selected, as shown in [5, 18, 12]. As a consequence of Theorems 1 and 3, choreography projections never deadlock.

Theorem 4. *Let $N = [\![C, \sigma]\!]$ for some C and σ. Then, either $N \preceq \mathbf{0}$ (N has terminated), or $N \to N'$ for some N' (N can reduce).*

Choreography Amendment. An important property of CC is that all unprojectable choreographies can be made projectable by adding some selections. We annotate recursion variables as for EPP, assuming that $\mathsf{pn}(X^{\tilde{\mathsf{p}}}) = \{\tilde{\mathsf{p}}\}$.

Definition 6. *Given C in CC, the transformation $\mathsf{Amend}(C)$ repeatedly applies the following procedure until no longer possible, starting from the innermost subterms in C. For each conditional subterm $\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ in C, let $\tilde{\mathsf{r}} \subseteq (\mathsf{pn}(C_1) \cup \mathsf{pn}(C_2))$ be the largest set such that $[\![C_1]\!]_{\mathsf{r}} \sqcup [\![C_2]\!]_{\mathsf{r}}$ is undefined for all $\mathsf{r} \in \tilde{\mathsf{r}}$; then $\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ in C is replaced with:*

$$\mathsf{if}\ (\mathsf{p} \overset{\leftarrow}{=} \mathsf{q})\ \mathsf{then}\ (\mathsf{p}\ \texttt{->}\ \mathsf{r}_1[\mathrm{L}]; \cdots ; \mathsf{p}\ \texttt{->}\ \mathsf{r}_n[\mathrm{L}]; C_1)\ \mathsf{else}\ (\mathsf{p}\ \texttt{->}\ \mathsf{r}_1[\mathrm{R}]; \cdots ; \mathsf{p}\ \texttt{->}\ \mathsf{r}_n[\mathrm{R}]; C_2)$$

From the definitions of Amend, EPP and the semantics of CC, we get:

Lemma 2. *For every choreography C:*

Completeness: $\mathsf{Amend}(C)$ *is defined;*
Projectability: *for all σ, $[\![\mathsf{Amend}(C), \sigma]\!]$ is defined;*
Correspondence: *for all σ, $C, \sigma \to^* C', \sigma'$ iff $\mathsf{Amend}(C), \sigma \to^* \mathsf{Amend}(C'), \sigma'$.*

Example 2. Applying Amend to the first choreography in Remark 1 yields the second choreography in the same remark. Thanks to merging, amendment can also recognise some situations where additional selections are not needed. For example, in the choreography $C = \mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ (\mathsf{p}.(\mathsf{s} \cdot \mathsf{c})\ \texttt{->}\ \mathsf{r}; \mathbf{0})\ \mathsf{else}\ (\mathsf{p}.(\mathsf{c})\ \texttt{->}\ \mathsf{r}; \mathbf{0})$, r does not need to know the choice made by p, as it always performs the same input action. Here, C is projectable and $\mathsf{Amend}(C) = C$.

4 Turing completeness of CC and SP

We now move to our main result: the set of choreography projections of CC (the processes synthesised by EPP) is not only deadlock-free, but also capable of computing all partial recursive functions, as defined by Kleene [17], and hence Turing complete. To this aim, the design and properties of CC give us a considerable pay off. First, by Theorem 3, the problem reduces to establishing that a projectable fragment of CC is Turing complete. Second, by Lemma 2, this simpler problem is reduced to establishing that CC is Turing complete regardless of projectability, since any unprojectable choreography can be amended to one that is projectable and computes the same values. We also exploit the concurrent semantics of CC and Theorem 2 to parallelise independent sub-computations (Theorem 6).

Establishing that CC is Turing complete is long, but not difficult. Our proof is in line with other traditional proofs of computational completeness [11, 17, 27], where data and programs are distinct. This differs from other proofs of similar results for process calculi (e.g., π-calculus [26]) or λ-calculus [1], which encode data as particular programs. The advantages are: our proof can be used to build choreographies that compute particular functions; and we can parallelise independent sub-computations in functions (Theorem 6).

Partial Recursive Functions. Our definition of the class of partial recursive functions $\mathcal{R}$ is slightly simplified, but equivalent to, that in [17], where it is also shown that $\mathcal{R}$ is the class of functions computable by a Turing machine. $\mathcal{R}$ is defined inductively as follows.

Unary zero: $Z \in \mathcal{R}$, where $Z : \mathbb{N} \to \mathbb{N}$ is s.t. $Z(x) = 0$ for all $x \in \mathbb{N}$.
Unary successor: $S \in \mathcal{R}$, where $S : \mathbb{N} \to \mathbb{N}$ is s.t. $S(x) = x + 1$ for all $x \in \mathbb{N}$.
Projections: If $n \geq 1$ and $1 \leq m \leq n$, then $P^n_m \in \mathcal{R}$, where $P^n_m : \mathbb{N}^n \to \mathbb{N}$ is s.t. $P^n_m(x_1, \ldots, x_n) = x_m$ for all $x_1, \ldots, x_n \in \mathbb{N}$.
Composition: if $f, g_i \in \mathcal{R}$ for $1 \leq i \leq k$, with each $g_i : \mathbb{N}^n \to \mathbb{N}$ and $f : \mathbb{N}^k \to \mathbb{N}$, then $h = C(f, \tilde{g}) \in \mathcal{R}$, where $h : \mathbb{N}^n \to \mathbb{N}$ is defined by composition from f and $g_1, \ldots, g_k$ as: $h(\tilde{x}) = f(g_1(\tilde{x}), \ldots, g_k(\tilde{x}))$.
Primitive recursion: if $f, g \in \mathcal{R}$, with $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$, then $h = R(f, g) \in \mathcal{R}$, where $h : \mathbb{N}^{n+1} \to \mathbb{N}$ is defined by primitive recursion from f and g as: $h(0, \tilde{x}) = f(\tilde{x})$ and $h(x_0 + 1, \tilde{x}) = g(x_0, h(x_0, \tilde{x}), \tilde{x})$.
Minimization: If $f \in \mathcal{R}$, with $f : \mathbb{N}^{n+1} \to \mathbb{N}$, then $h = M(f) \in \mathcal{R}$, where $h : \mathbb{N}^n \to \mathbb{N}$ is defined by minimization from f as: $h(\tilde{x}) = y$ iff (1) $f(\tilde{x}, y) = 0$ and (2) $f(\tilde{x}, y)$ is defined and different from 0 for all $z < y$.

Encoding Partial Recursive Functions in CC. All functions in $\mathcal{R}$ can be implemented in CC, in the sense of Definition 1. Given $f : \mathbb{N}^n \to \mathbb{N}$, we denote its implementation by $[\![f]\!]^{\tilde{\mathsf{p}} \mapsto \mathsf{q}}$, where $\tilde{\mathsf{p}}$ and q are parameters. All choreographies we build have a single exit point, and we combine them using the sequential composition operator $\fatsemi$ from § 2. We use auxiliary processes $(\mathsf{r}_0, \mathsf{r}_1, \ldots)$ for intermediate computation, and annotate the encoding with the index ℓ of the first free auxiliary process name ($[\![f]\!]^{\tilde{\mathsf{p}} \mapsto \mathsf{q}}_\ell$). To alleviate the notation, the encoding assigns mnemonic names to these processes and their correspondence to the actual process names is formalised in the text using $\pi(f)$ for the number of auxiliary processes needed for encoding $f : \mathbb{N}^n \to \mathbb{N}$, defined by

$$\pi(S) = \pi(Z) = \pi\left(P^n_m\right) = 0 \qquad \pi(R(f,g)) = \pi(f) + \pi(g) + 3$$
$$\pi\left(C(f, g_1, \ldots, g_k)\right) = \pi(f) + \textstyle\sum_{i=1}^k \pi(g_i) + k \qquad \pi(M(f)) = \pi(f) + 3$$

For simplicity, we write $\tilde{\mathsf{p}}$ for $\mathsf{p}_1, \ldots, \mathsf{p}_n$ (when n is known) and $\{A_i\}_{i=1}^n$ for $A_1 \fatsemi \ldots \fatsemi A_n$. We omit the selections needed for projectability, as they can be inferred by amendment; we will discuss this aspect formally later.

The encoding of the base cases is straightforward.

$$[\![Z]\!]^{\mathsf{p} \mapsto \mathsf{q}}_\ell = \mathsf{p}.\varepsilon \; \texttt{->} \; \mathsf{q} \qquad [\![S]\!]^{\mathsf{p} \mapsto \mathsf{q}}_\ell = \mathsf{p}.(\mathsf{s} \cdot \mathsf{c}) \; \texttt{->} \; \mathsf{q} \qquad [\![P^n_m]\!]^{\tilde{\mathsf{p}} \mapsto \mathsf{q}}_\ell = \mathsf{p}_m.\mathsf{c} \; \texttt{->} \; \mathsf{q}$$

Composition is also simple. Let $h = C(f, g_1, \ldots, g_k) : \mathbb{N}^n \to \mathbb{N}$. Then:

$$[\![h]\!]^{\tilde{\mathsf{p}} \mapsto \mathsf{q}}_\ell = \left\{ [\![g_i]\!]^{\tilde{\mathsf{p}} \mapsto \mathsf{r}'_i}_{\ell_i} \right\}_{i=1}^k \fatsemi [\![f]\!]^{\mathsf{r}'_1, \ldots, \mathsf{r}'_k \mapsto \mathsf{q}}_{\ell_{k+1}}$$

where $\mathsf{r}'_i = \mathsf{r}_{\ell+i-1}$, $\ell_1 = \ell + k$ and $\ell_{i+1} = \ell_i + \pi(g_i)$. Each auxiliary process r'_i connects the output of g_i to the corresponding input of f. Choreographies obtained inductively use these process names as parameters; name clashes are prevented by increasing ℓ. By definition of $\fatsemi$ $[\![g_{i+1}]\!]$ is substituted for the (unique) exit point of $[\![g_i]\!]$, and $[\![f]\!]$ is substituted for the exit point of $[\![g_k]\!]$. The resulting choreography also has only one exit point (that of $[\![f]\!]$). Below we discuss how to modify this construction slightly so that the g_is are computed in parallel.

For the recursion operator, we need to use recursive procedures. Let $h = R(f,g) : \mathbb{N}^{n+1} \to \mathbb{N}$. Then, using the macro $\textsc{inc}$ from Example 1 for brevity:

$$
\begin{aligned}
[\![h]\!]_\ell^{\mathsf{p}_0,\dots,\mathsf{p}_n\mapsto\mathsf{q}} = \quad & \mathsf{def}\ T = \mathsf{if}\ (\mathsf{r}_c \overset{\leftarrow}{=} \mathsf{p}_0)\ \mathsf{then}\ (\mathsf{q}'.\mathsf{c} \text{ -> } \mathsf{q};\ \mathbf{0}) \\
& \qquad\qquad \mathsf{else}\ [\![g]\!]_{\ell_g}^{\mathsf{r}_c,\mathsf{q}',\mathsf{p}_1,\dots,\mathsf{p}_n\mapsto\mathsf{r}_t} \fatsemi \mathsf{r}_t.\mathsf{c} \text{ -> } \mathsf{q}';\ \textsc{inc}(\mathsf{r}_c,\mathsf{r}_t) \fatsemi T \\
& \mathsf{in}\ [\![f]\!]_{\ell_f}^{\mathsf{p}_1,\dots,\mathsf{p}_n\mapsto\mathsf{q}'} \fatsemi \mathsf{r}_t.\varepsilon \text{ -> } \mathsf{r}_c;\ T
\end{aligned}
$$

where $\mathsf{q}' = \mathsf{r}_\ell$, $\mathsf{r}_c = \mathsf{r}_{\ell+1}$, $\mathsf{r}_t = \mathsf{r}_{\ell+2}$, $\ell_f = \ell + 3$ and $\ell_g = \ell_f + \pi(f)$. Process r_c is a counter, q' stores intermediate results, and r_t is temporary storage; T checks the value of r_c and either outputs the result or recurs. Note that $[\![h]\!]$ has only one exit point (after the communication from r to q), as the exit points of $[\![f]\!]$ and $[\![g]\!]$ are replaced by code ending with calls to T.

The strategy for minimization is similar, but simpler. Let $h = M(f) : \mathbb{N}^n \to \mathbb{N}$. Again we use a counter r_c and compute successive values of f, stored in q', until a zero is found. This procedure may loop forever, either because $f(\tilde{x}, x_{n+1})$ is never 0 or because one of the evaluations itself never terminates.

$$
\begin{aligned}
[\![h]\!]_\ell^{\mathsf{p}_1,\dots,\mathsf{p}_{n+1}\mapsto\mathsf{q}} = \ & \mathsf{def}\ T = [\![f]\!]_{\ell_f}^{\mathsf{p}_1,\dots,\mathsf{p}_n,\mathsf{r}_c\mapsto\mathsf{q}'} \fatsemi \mathsf{r}_c.\varepsilon \text{ -> } \mathsf{r}_z; \\
& \qquad\qquad \mathsf{if}\ (\mathsf{r}_z \overset{\leftarrow}{=} \mathsf{q}')\ \mathsf{then}\ (\mathsf{r}_c.\mathsf{c} \text{ -> } \mathsf{q};\ \mathbf{0})\ \mathsf{else}\ (\textsc{inc}(\mathsf{r}_c,\mathsf{r}_z) \fatsemi T) \\
& \mathsf{in}\ \mathsf{r}_z.\varepsilon \text{ -> } \mathsf{r}_c;\ T
\end{aligned}
$$

where $\mathsf{q}' = \mathsf{r}_\ell$, $\mathsf{r}_c = \mathsf{r}_{\ell+1}$, $\mathsf{r}_z = \mathsf{r}_{\ell+2}$, $\ell_f = \ell + 3$ and $\ell_g = \ell_f + \pi(f)$. In this case, the whole if-then-else is inserted at the exit point of $[\![f]\!]$; the only exit point of this choreography is again after communicating the result to q.

Definition 7. *Let $f \in \mathcal{R}$. The* encoding *of f in CC is $[\![f]\!]^{\tilde{\mathsf{p}}\mapsto\mathsf{q}} = [\![f]\!]_0^{\tilde{\mathsf{p}}\mapsto\mathsf{q}}$.*

Main Results. We prove that our construction is sound by induction.

Theorem 5. *If $f : \mathbb{N}^n \to \mathbb{N}$ and $f \in \mathcal{R}$, then, for every k, $[\![f]\!]_k^{\tilde{\mathsf{p}}\mapsto\mathsf{q}}$ implements f with input processes $\tilde{\mathsf{p}} = \mathsf{p}_1, \dots, \mathsf{p}_n$ and output process q.*

Let $\mathrm{SP}^{\mathrm{CC}} = \{[\![C,\sigma]\!] \mid [\![C,\sigma]\!] \text{ is defined}\}$ be the set of the projections of all projectable choreographies in CC. By Theorem 4, all terms in $\mathrm{SP}^{\mathrm{CC}}$ are deadlock-free. By Lemma 2, for every function f we can amend $[\![f]\!]$ to an equivalent projectable choreography. Then $\mathrm{SP}^{\mathrm{CC}}$ is Turing complete by Theorems 3 and 5.

Corollary 1. *Every partial recursive function is implementable in SP^{CC}.*

We finish this section by showing how to optimize our encoding and obtain parallel process implementations of independent computations. If h is defined by composition from f and $g_1, \ldots, g_k$, then in principle the computation of the g_is could be completely parallelised. However, $[\![\,]\!]$ does not fully achieve this, as $[\![g_1]\!]$,…,$[\![g_k]\!]$ share the processes containing the input. We define a modified variant $\{\!\{\,\}\!\}$ of $[\![\,]\!]$ where, for $h = C(f, g_1, \ldots, g_k)$, $\{\!\{h\}\!\}^{\tilde{\mathsf{p}}\mapsto\mathsf{q}}_{\ell}$ is

$$\left\{\mathsf{p}_j.\mathsf{c}\ \texttt{->}\ \mathsf{p}^i_j\right\}_{1\le i\le k, 1\le j\le n} \fatsemi \left\{\{\!\{g_i\}\!\}^{\tilde{\mathsf{p}^i}\mapsto\mathsf{r}'_i}_{\ell_i}\right\}^{k}_{i=1} \fatsemi \{\!\{f\}\!\}^{\mathsf{r}'_1,\ldots,\mathsf{r}'_k\mapsto\mathsf{q}}_{\ell_{k+1}}$$

with a suitably adapted label function ℓ. Now Theorem 2 applies, yielding:

Theorem 6. *Let $h = C(f, g_1, \ldots, g_k)$. For all $\tilde{\mathsf{p}}$ and q, if $h(\tilde{x})$ is defined and σ is such that $\sigma(\mathsf{p}_i) = \ulcorner x_i \urcorner$, then all the $\{\!\{g_i\}\!\}^{\tilde{\mathsf{p}^i}\mapsto\mathsf{r}'_i}_{\ell_i}$ run in parallel in $\{\!\{h\}\!\}^{\tilde{\mathsf{p}}\mapsto\mathsf{q}}$.*

This parallelism is preserved by EPP, through Theorem 3.

5 Minimality in Choreography Languages

We now discuss our choice of primitives for CC, showing it to be a good candidate core language for choreographic programming. We analyse each primitive of CC. Recall that Turing completeness of CC is a pre-requisite for the Turing completeness of choreography projections. In many cases, simplifying CC yields a decidable termination problem (thus breaking Turing completeness). We discuss these cases first, and then proceed to a discussion on label selection.

Minimality in CC. Removing or simplifying the following primitives makes termination decidable.

- Exit point – $\mathbf{0}$: without it, no choreography terminates.
- Value communication – $\mathsf{p}.e$ -> q: without it, values of processes cannot be changed, and termination becomes decidable. The syntax of expressions is also minimal: ε (zero) is the only terminal; without c values become statically defined, while without s no new values can be computed; in either case, termination is decidable.
- Recursion – $\mathsf{def}\, X = C_2 \,\mathsf{in}\, C_1$ and X: without it, all choreographies trivially terminate. The terms are minimal: they support only tail recursion and definitions are not parameterised.

Theorem 7. *Let C be a choreography with no conditionals. Then, termination of C is decidable and independent of the initial state.*

More interestingly, limiting processes to evaluating only their own local values in conditions makes termination decidable. Intuitively, this is because a process can only hold a value at a time, and thus no process can compare its current value to that of another process anymore.

$$\llparenthesis \mathbf{0} \rrparenthesis = \mathbf{0} \quad \llparenthesis \mathsf{p}.e \texttt{ -> } \mathsf{q}; C \rrparenthesis = \mathsf{p}.e \texttt{ -> } \mathsf{q}; \llparenthesis C \rrparenthesis \quad \llparenthesis \mathsf{def}\ X = C_2 \mathsf{\ in\ } C_1 \rrparenthesis = \mathsf{def}\ X = \llparenthesis C_2 \rrparenthesis \mathsf{\ in\ } \llparenthesis C_1 \rrparenthesis$$

$$\llparenthesis X \rrparenthesis = X \qquad \llparenthesis \mathsf{if\ p} \stackrel{\leftarrow}{=} \mathsf{q\ then\ } C_1 \mathsf{\ else\ } C_2 \rrparenthesis = \mathsf{if\ p} \stackrel{\leftarrow}{=} \mathsf{q\ then\ } \llparenthesis C_1, C_2 \rrparenthesis_1 \mathsf{\ else\ } \llparenthesis C_1, C_2 \rrparenthesis_2$$

$\llparenthesis C_1, C_2 \rrparenthesis = \langle \llparenthesis C_1 \rrparenthesis, \llparenthesis C_2 \rrparenthesis \rangle$ if C_1 and C_2 do not begin with a selection

$\llparenthesis \mathsf{p} \texttt{ -> } \mathsf{q}[\textsc{l}];\ C_1, \mathsf{p} \texttt{ -> } \mathsf{q}[\textsc{r}];\ C_2 \rrparenthesis =$

$$\Big\langle \mathsf{q.c} \texttt{ -> } \mathsf{q}^\bullet;\ \mathsf{p}.\varepsilon \texttt{ -> } \mathsf{q};\ \mathsf{if\ q} \stackrel{\leftarrow}{=} \mathsf{z\ then\ q}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{q};\ \llparenthesis C_1, C_2 \rrparenthesis_1 \mathsf{\ else\ q}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{q};\ \llparenthesis C_1, C_2 \rrparenthesis_2,$$

$$\mathsf{q.c} \texttt{ -> } \mathsf{q}^\bullet;\ \mathsf{p.sc} \texttt{ -> } \mathsf{q};\ \mathsf{if\ q} \stackrel{\leftarrow}{=} \mathsf{z\ then\ q}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{q};\ \llparenthesis C_1, C_2 \rrparenthesis_1 \mathsf{\ else\ q}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{q};\ \llparenthesis C_1, C_2 \rrparenthesis_2 \Big\rangle$$

Fig. 2. Elimination of selections from amended choreographies.

Theorem 8. *If the conditional is replaced by* $\mathsf{if\ p.c} = v \mathsf{\ then\ } C_1 \mathsf{\ else\ } C_2$*, where* v *is a value, and rule* $\lfloor C|Cond \rceil$ *by* $\dfrac{i = 1 \text{ if } \sigma(\mathsf{p}) = v,\ i = 2 \text{ otherwise}}{\mathsf{if\ p.c} = v \mathsf{\ then\ } C_1 \mathsf{\ else\ } C_2, \sigma \ \to\ C_i, \sigma}$ *, then termination is decidable.*

Label selection. The argument for including label selections in CC is of a different nature. As the construction in § 4 shows, selections are not needed for implementing computable functions in CC; they are used only for obtaining projectable choreographies, via amendment. We now show that we can encode selections introduced by amendment using the other primitives of CC, thereby eliminating the need for them from a purely computational point of view.

We denote by Minimal Choreographies (MC) the fragment of CC that does not contain label selections. We can therefore view amendment as a function from MC into the subset of projectable CC choreographies. Recall that the definition of amendment guarantees that selections only occur in branches of conditionals, and that they are always paired and in the same order (see Definition 6). The fragment of CC obtained by amending choreographies in MC is thus:

$$C ::= \mathsf{p}.e \texttt{ -> } \mathsf{q}; C \mid \mathsf{if\ p} \stackrel{\leftarrow}{=} \mathsf{q\ then\ } S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{l}, C_1) \mathsf{\ else\ } S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{r}, C_2) \mid \mathsf{def}\ X = C_2 \mathsf{\ in\ } C_1 \mid X \mid \mathbf{0}$$

Term $S(\mathsf{p}, \tilde{\mathsf{r}}, l, C)$ denotes a series of selections of label l from p to all processes in the list $\tilde{\mathsf{r}}$. Formally, $S(\mathsf{p}, \emptyset, l, C) = C$ and $S(\mathsf{p}, \mathsf{r} :: \tilde{\mathsf{r}}, l, C) = \mathsf{p} \texttt{ -> } \mathsf{r}[l];\ S(\mathsf{p}, \tilde{\mathsf{r}}, l, C)$.

Definition 8. *Let* C *be obtained by amending a choreography in MC. The encoding* $\llparenthesis C \rrparenthesis^+$ *of* C *in MC uses processes* $\mathsf{p}, \mathsf{p}^\bullet$ *for each* $\mathsf{p} \in \mathsf{pn}(C)$ *and a special process* z*, and is defined as* $\llparenthesis C \rrparenthesis^+ = \mathsf{p}.\varepsilon \texttt{ -> } \mathsf{z};\ \llparenthesis C \rrparenthesis$*, with* $\llparenthesis C \rrparenthesis$ *defined in Figure 2.*

The definition of $\llparenthesis C \rrparenthesis$ exploits the structure of amended choreographies, where selections are always paired at the top of the two branches of conditionals. It is immediate that $|\mathsf{pn}(\llparenthesis C \rrparenthesis^+)| = 2|\mathsf{pn}(C)| + 1$ (the extra process is z). Let $|C|$ be the size of the syntax tree of C. Then, $|\llparenthesis C \rrparenthesis^+| \leq 2^{|C|}$, and in the worst case we get exponential growth. However, EPP collapses all branches of conditionals, hence projections do not grow exponentially: $|[\![(\!|C|\!)^+]\!]_{\mathsf{q}\bullet}| \leq |[\![(\!|C|\!)^+]\!]_{\mathsf{q}}| \leq 3|[\![C]\!]_{\mathsf{q}}|$ for every $\mathsf{q} \in \mathsf{pn}(C)$.

Theorem 9. *For every choreography C in MC, $[\![(\!|\mathsf{Amend}(C)|\!)]\!]$ is defined.*

It is straightforward to prove that C and $(\!|\mathsf{Amend}(C)|\!)$ behave exactly in the same way when we only observe communications between the original processes – except that label selections are replaced by regular messages.

Lemma 3. *If $C, \sigma \to C', \sigma'$ and σ^+ is such that $\sigma^+(\mathsf{p}) = \sigma(\mathsf{p})$ for $\mathsf{p} \in \mathsf{pn}(C)$ and $\sigma^+(\mathsf{z}) = \varepsilon$, then $(\!|\mathsf{Amend}(C)|\!), \sigma^+ \to^* (\!|C'|\!), \sigma'^+$ for some σ'^+ similarly related to σ'. Conversely, if $(\!|\mathsf{Amend}(C)|\!), \sigma^+ \to C', \sigma'$, then $C, \sigma \to C'', \sigma''$ where $C', \sigma' \to^* (\!|\mathsf{Amend}(C'')|\!), \sigma''^+$.*

Corollary 2. *With the notation of the previous lemma, if $C, \sigma \to^* C', \sigma'$, then $(\!|\mathsf{Amend}(C)|\!)^+, \sigma^+ \to^* (\!|\mathsf{Amend}(C')|\!), \sigma'^+$.*

As a consequence, the set $\mathrm{SP}^{\mathrm{MC}} = \{[\![C, \sigma]\!] \mid C \text{ in MC and } [\![C, \sigma]\!] \text{ is defined}\}$ of projections of minimal choreographies is also Turing complete.

Corollary 3. *Every partial recursive function is implementable in SP^{MC}.*

Since choreographies in MC do not have selections, process projections of choreographies in MC never have branchings. This means that, in the case of MC, the merging operator $\sqcup$ used in EPP is exactly syntactic equality (since the only nontrivial case was that of branchings). Consequently, we can replace the rule for projecting conditionals with a simpler one:

$$[\![\mathsf{if}\ \mathsf{p} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!]_{\mathsf{r}} = \begin{cases} \mathsf{if}\ \mathsf{c} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ [\![C_1]\!]_{\mathsf{r}}\ \mathsf{else}\ [\![C_2]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}!\langle \mathsf{c} \rangle; [\![C_1]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \text{ and } [\![C_1]\!]_{\mathsf{r}} = [\![C_2]\!]_{\mathsf{r}} \\ [\![C_1]\!]_{\mathsf{r}} & \text{if } \mathsf{r} \notin \{\mathsf{p}, \mathsf{q}\} \text{ and } [\![C_1]\!]_{\mathsf{r}} = [\![C_2]\!]_{\mathsf{r}} \end{cases}$$

The advantages of eliminating selections are thus a simpler choreography language, a simpler definition of EPP (without merging), and a simpler process language (without selection and branching). The main drawback is that eliminating a selection needed for projectability makes the choreography exponentially larger and requires the addition of extra processes and communications; this significantly changes the structure of the choreography, potentially making it unreadable. Selections are also present in virtually all choreography models [2, 5, 6, 16, 12, 25], therefore we believe that a core model such as CC should have them (in addition to the drawback we mentioned).

Our results suggest the viability of a particular implementation strategy for choreographic programming. Programmers could write choreographies without label selections, and then our results could be used to translate these choreographies to process implementations in a simple language that does not include label communications, thus simplifying the target language. The exponential growth of the intermediate choreography representation can be bypassed by using shared data structures for the syntax tree, since the generated choreographies contain a lot of duplicate terms.

However, this implementation removes an important ability provided in CC and all other standard choreography calculi: deciding at which point of execution selections should be performed. In more expressive languages than CC, processes can perform complex internal computations [10]. For example, assume that p had to assign tasks to other two processes r and s based on a condition. In one case, r would run a slow task and s a fast one; otherwise, r would run a fast task and s a slow one. In this case, p should begin by sending a selection to the process with the slow task and then by sending it the necessary data for its computation, before it sends the selection to the process with the fast task.

6 CC and other languages

CC is representative of the body of previous work on choreographic programming, where choreographies are used for implementations, for example [5, 6, 8, 24, 12, 28]. All the primitives of CC can be encoded in such languages. Thus, we obtain a notion of function implementation for these languages, induced by that for CC, for which they are Turing complete. For the model in [6], we formalise this result in [9]. Below, we discuss the significance of our results for the cited languages.

Differently from CC, other choreography languages typically use channel-based communications (as in the π-calculus [26]). Communications via process references as in CC can be encoded by assigning a dedicated channel to each pair of processes [9]. For example, the calculus in [6], which we refer to as Channel Choreographies (ChC), features an EPP targeting the session-based π-calculus [2]. ChC is a fully-fledged calculus aimed at real-world application that has been implemented as a choreographic programming framework (the Chor language [8]). Our formal translation from CC to ChC (given in [9]) shows that many primitives of ChC are not needed to achieve Turing completeness, including: asynchronous communications, creation of sessions and processes, channel mobility, parameterised recursive definitions, arbitrary local computation, unbounded memory cells at processes, multiparty sessions. While useful in practice, these primitives come at the cost of making the formal treatment of ChC very technically involved. In particular, ChC (and its implementation Chor) requires a sophisticated type system, linearity analysis, and definition of EPP to ensure correctness of projected processes. These features are not needed in CC. Using our encoding from CC to ChC, we can repeat the argument in § 4 to characterise a fragment of the session-based π-calculus from [2] that contains only deadlock-free terms and is Turing complete. ChC has also been translated to the Jolie programming language [14, 23], whence our reasoning also applies to the latter and, in general, to service-oriented languages based on message correlation.

The language WS-CDL from W3C [28] and the formal models inspired by it (e.g., [5]) are very similar to ChC and a similar translation from CC could be formally developed, with similar implications as above. The same applies to the choreography language developed in [12], which adds higher-order features to choreographies in terms of runtime adaptation. Finally, the language of compositional choreographies presented in [24] is an extension of ChC and therefore our translation applies directly. This implies that adding modularity to choreographies does not add any computational power, as expected.

7 Related Work and Discussion

Register Machines. The computational primitives in CC recall those of the Unlimited Register Machine (URM) [11], but CC and URM differ in two main aspects. First, URM programs contain go-to statements, while CC supports only tail recursion. Second, in the URM there is a single sequential program manipulating the cells, whereas in CC computation is distributed among the various cells (the processes), which operate concurrently.

Simulating the URM is an alternative way to prove Turing completeness of CC. However, our proof using partial recursive functions is more direct and gives an algorithm to implement any function in CC, given its proof of membership in $\mathcal{R}$. It also yields the natural interpretation of parallelisation stated in Theorem 6. Similarly, we could establish Turing completeness of CC using only a bounded number of processes. However, such constructions encode data using Gödel numbers, which is not in the spirit of our declarative notion of function implementation. They also restrict concurrency, breaking Theorem 6.

Multiparty Sessions and Types. The communication primitives in CC recall those of protocols for multiparty sessions, e.g., in Multiparty Session Types (MPST) [16] and conversation types [4]. These protocols are not meant for computation, as in choreographic programming (and CC); rather, they are types used to verify that sessions (e.g., π-calculus channels) are used accordingly to their respective protocol specifications. For such formalisms, we know of a strong characterisation result: a variant of MPST corresponds to communicating finite state machines [3] that respect the property of multiparty compatibility [13]. To the best of our knowledge, this is the first work studying the expressivity of choreographic programming (choreographies for implementations).

Full β-reduction and Nondeterminism. Execution in CC is nondeterministic due to the swapping of communications allowed by the structural precongruence $\preceq$. This recalls full β-reduction for λ-calculus, where sub-terms can be evaluated whenever possible. However, the two mechanisms are actually different. Consider the choreography $C \stackrel{\Delta}{=} \mathsf{p.c} \texttt{ -> } \mathsf{q}; \mathsf{q}.\varepsilon \texttt{ -> } \mathsf{r}; \mathbf{0}$. If CC supported full β-reduction, we should be able to reduce the second communication before the first one, since there is no data dependency between the two. Formally, for some σ: $C, \sigma \to \mathsf{p.c} \texttt{ -> } \mathsf{q}; \mathbf{0}, \sigma[\mathsf{r} \mapsto \varepsilon]$. However, this reduction is disallowed by our semantics: rule $\lfloor$C|Eta-Eta$\rceil$ cannot be applied because q is present in both communications. This difference is a key feature of choreographies, stemming from their practical origins: controlling sequentiality by establishing causalities using process identifiers is important for the implementation of business processes [28]. For example, imagine that the choreography C models a payment transaction and that the message from q to r is a confirmation that p has sent its credit card information to q; then, it is a natural requirement that the second communication happens only after the first. Note that we would reach the same conclusions even if we adopted an asynchronous messaging semantics for SP, since the first action by q is a blocking input.

While execution in CC can be nondeterministic, computation results are deterministic as in many other choreography languages [6, 7, 24]: if a choreography terminates, the result will always be the same regardless of how its execution is scheduled (recalling the Church–Rosser Theorem for the λ-calculus). Nondeterministic computation is not necessary for our results. Nevertheless, it can be easily added to CC. Specifically, we could augment CC with the syntax primitive $C_1 \oplus^{\mathsf{p}} C_2$ and the reduction rule $C_1 \oplus^{\mathsf{p}} C_2 \;\to\; C_i$ for $i = 1, 2$. Extending SP with an internal choice $B_1 \oplus B_2$ and our definition of EPP is straightforward: in SP, we would also allow $B_1 \oplus B_2 \;\to\; B_i$ for $i = 1, 2$, and define $[\![C_1 \oplus^{\mathsf{p}} C_2]\!]_{\mathsf{r}}$ to be $[\![C_1]\!]_{\mathsf{r}} \oplus [\![C_2]\!]_{\mathsf{r}}$ if $\mathsf{r} = \mathsf{p}$ and $[\![C_1]\!]_{\mathsf{r}} \sqcup [\![C_2]\!]_{\mathsf{r}}$ otherwise.

Merging and Amendment. Amendment was first studied in [19] for a simple language with finite traces (thus not Turing complete). Our definition is different, since it uses merging for the first time.

Actors and Asynchrony. Processes in SP communicate by using direct references to each other, recalling actor systems. However, there are notable differences: communications are synchronous and inputs specify the intended sender. The first difference comes from minimality: asynchrony would add possible behaviours to CC, which are unnecessary to establish Turing completeness. We leave an investigation of asynchrony in CC to future work. The second difference arises because CC is a choreography calculus, and communication primitives in choreographies typically express both sender and receiver.

References

1. H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 2nd edition, 1984.
2. L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In F. van Breugel and M. Chechik (eds.), *CONCUR*, vol. 5201 of *LNCS*, pp. 418–433. Springer, 2008.
3. D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, Apr. 1983.
4. L. Caires and H. T. Vieira. Conversation types. *Theor. Comput. Sci.*, 411(51-52):4399–4440, 2010.
5. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
6. M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In R. Giacobazzi and R. Cousot (eds.), *POPL*, pp. 263–274. ACM, 2013.

7. M. Carbone, F. Montesi, and C. Schürmann. Choreographies, logically. In P. Baldan and D. Gorla (eds.), *CONCUR*, vol. 8704 of *LNCS*, pp. 47–62. Springer, 2014.
8. Chor. Programming Language. http://www.chor-lang.org/.
9. L. Cruz-Filipe and F. Montesi. Choreographies, computationally. *CoRR*, abs/1510.03271, 2015.
10. L. Cruz-Filipe and F. Montesi. Choreographies, divided and conquered. *CoRR*, abs/1602.03729, 2016. Submitted for publication.
11. N. Cutland. *Computability: an Introduction to Recursive Function Theory*. Cambridge University Press, 1980.
12. M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic choreographies – safe runtime updates of distributed applications. In T. Holvoet and M. Viroli (eds.), *COORDINATION*, vol. 9037 of *LNCS*, pp. 67–82. Springer, 2015.
13. P.-M. Deniélou and N. Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In F. Fomin, R. Freivalds, M. Kwiatkowska, and D. Peleg (eds.), *ICALP (II)*, vol. 7966 of *LNCS*, pp. 174–186. Springer, 2013.
14. M. Gabbrielli, S. Giallorenzo, and F. Montesi. Applied choreographies. *CoRR*, abs/1510.03637, 2015.
15. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In C. Hankin (ed.), *ESOP*, vol. 1381 of *LNCS*, pp. 122–138. Springer, 1998.
16. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In G. Necula and P. Wadler (eds.), *POPL*, pp. 273–284. ACM, 2008.
17. S. Kleene. *Introduction to Metamathematics*. North-Holland Publishing Co., 1952.
18. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In A. Cerone and S. Gruner (eds.), *SEFM*, pp. 323–332. IEEE, 2008.
19. I. Lanese, F. Montesi, and G. Zavattaro. Amending choreographies. In A. Ravara and J. Silva (eds.), *WWV 2013*, vol. 123 of *EPTCS*, pp. 34–48, 2013.
20. T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS*, pp. 517–530. ACM, 2016.
21. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pp. 329–339. ACM, 2008.
22. F. Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. http://fabriziomontesi.com/files/choreographic_programming.pdf.
23. F. Montesi, C. Guidi, and G. Zavattaro. Service-oriented programming with Jolie. In A. Bouguettaya, Q. Sheng, and F. Daniel (eds.), *Web Services Foundations*, pp. 81–107. Springer, 2014.
24. F. Montesi and N. Yoshida. Compositional choreographies. In P. D'Argenio and H. Melgratti (eds.), *CONCUR*, vol. 8052 of *LNCS*, pp. 425–439. Springer, 2013.
25. Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In C. Williamson, M. Zurko, P. Patel-Schneider, and P. Shenoy (eds.), *WWW*, pp. 973–982. ACM, 2007.
26. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
27. A. Turing. Computability and λ-definability. *J. Symb. Log.*, 2(4):153–163, 1937.
28. W3C WS-CDL Working Group. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/, 2004.

A Complete definitions and proofs of results

For reference, we include full descriptions of the reduction semantics for MC and SP. We also indicate the proof strategies for all results in the paper, detailing those that are not immediate.

Section 2 (Semantics of MC). The reduction semantics of MC, given in terms of the relation $\to$, is defined by the set of rules in Figure 3. It uses the structural precongruence $\preceq$ defined by the rules in Figure 4.

$$\frac{v = e[\sigma(\mathsf{p})/\mathsf{c}]}{\mathsf{p}.e \text{ -> } \mathsf{q}; C, \sigma \to C, \sigma[\mathsf{q} \mapsto v]} \lfloor\text{C}|\text{Com}\rceil \qquad \frac{i = 1 \text{ if } \sigma(\mathsf{p}) = \sigma(\mathsf{q}),\ i = 2 \text{ o.w.}}{\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \sigma \to C_i, \sigma} \lfloor\text{C}|\text{Cond}\rceil$$

$$\frac{C_1 \preceq C_2 \quad C_2, \sigma \to C_2', \sigma' \quad C_2' \preceq C_1'}{C_1, \sigma \to C_1', \sigma'} \lfloor\text{C}|\text{Struct}\rceil \qquad \frac{}{\mathsf{p} \text{ -> } \mathsf{q}[l]; C, \sigma \to C, \sigma} \lfloor\text{C}|\text{Sel}\rceil$$

$$\frac{C_1, \sigma \ \to\ C_1', \sigma'}{\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1, \sigma \ \to\ \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1', \sigma'} \lfloor\text{C}|\text{Ctx}\rceil$$

Fig. 3. Semantics of MC.

$$\frac{\mathsf{pn}(\eta) \cap \mathsf{pn}(\eta') = \emptyset}{\eta; \eta' \ \equiv\ \eta'; \eta} \lfloor\text{C}|\text{Eta-Eta}\rceil \qquad \frac{}{\mathsf{def}\ X = C\ \mathsf{in}\ \mathbf{0} \ \preceq\ \mathbf{0}} \lfloor\text{C}|\text{ProcEnd}\rceil$$

$$\frac{\{\mathsf{p},\mathsf{q}\} \cap \mathsf{pn}(\eta) = \emptyset}{\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ (\eta; C_1)\ \mathsf{else}\ (\eta; C_2) \quad \equiv \quad \eta; \mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2} \lfloor\text{C}|\text{Eta-Cond}\rceil$$

$$\frac{\mathsf{pn}(C_i) \cap \mathsf{pn}(\eta) = \emptyset}{\mathsf{def}\ X = C_2\ \mathsf{in}\ (\eta; C_1) \quad \equiv \quad \eta; \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1} \lfloor\text{C}|\text{Eta-Rec}\rceil$$

$$\frac{}{\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1[X] \quad \preceq \quad \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1[C_2]} \lfloor\text{C}|\text{Unfold}\rceil$$

$$\frac{\{\mathsf{p},\mathsf{q}\} \cap \{\mathsf{r},\mathsf{s}\} = \emptyset}{\begin{array}{c}\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ (\mathsf{if}\ \mathsf{r} \overset{\leftarrow}{=} \mathsf{s}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2)\ \mathsf{else}\ (\mathsf{if}\ \mathsf{r} \overset{\leftarrow}{=} \mathsf{s}\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2') \\ \equiv \\ \mathsf{if}\ \mathsf{r} \overset{\leftarrow}{=} \mathsf{s}\ \mathsf{then}\ (\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_1')\ \mathsf{else}\ (\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_2\ \mathsf{else}\ C_2')\end{array}} \lfloor\text{C}|\text{Cond-Cond}\rceil$$

Fig. 4. Structural precongruence in MC.

Proof (Theorem 1). Direct consequence of the definition of the semantics.

Proof (Lemma 1). Straightforward by structural induction on C.

Proof (Theorem 2). By induction over C_1. The converse implication is a consequence of (1) in Lemma 1.

Section 3 (Semantics of SP). The full semantics of SP is given in Figure 5. Again it relies on structural precongruence, which is defined by symmetry and associativity of $|$ together with the rules in Figure 6.

$$\frac{u = e[v/\mathsf{c}]}{\mathsf{p} \triangleright_v \mathsf{q}!\langle e\rangle; B_1 \;\mid\; \mathsf{q} \triangleright_w \mathsf{p}?; B_2 \;\to\; \mathsf{p} \triangleright_v B_1 \;\mid\; \mathsf{q} \triangleright_u B_2} \;\lfloor\text{S}|\text{Com}\rceil$$

$$\frac{j \in I}{\mathsf{p} \triangleright_v \mathsf{q} \oplus l_j; B \;\mid\; \mathsf{q} \triangleright_w \mathsf{p}\&\{l_i : B_i\}_{i\in I} \;\to\; \mathsf{p} \triangleright_v B \;\mid\; \mathsf{q} \triangleright_w B_j} \;\lfloor\text{S}|\text{Sel}\rceil$$

$$\frac{i = 1 \text{ if } v = e[w/\mathsf{c}], \quad i = 2 \text{ otherwise}}{\mathsf{p} \triangleright_v \mathsf{if}\ \mathsf{c} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2 \;\mid\; \mathsf{q} \triangleright_w \mathsf{p}!\langle e\rangle; B' \;\to\; \mathsf{p} \triangleright_v B_i \;\mid\; \mathsf{q} \triangleright_w B'} \;\lfloor\text{S}|\text{Cond}\rceil$$

$$\frac{B_1 \;\to\; B_1'}{\mathsf{p} \triangleright_v \mathsf{def}\ X = B_2\ \mathsf{in}\ B_1 \;\to\; \mathsf{p} \triangleright_v \mathsf{def}\ X = B_2\ \mathsf{in}\ B_1'} \;\lfloor\text{S}|\text{Ctx}\rceil$$

$$\frac{N \;\to\; N'}{N \,|\, M \;\to\; N' \,|\, M} \;\lfloor\text{S}|\text{Par}\rceil \qquad \frac{N \preceq M \quad M \;\to\; M' \quad M' \preceq N'}{N \;\to\; N'} \;\lfloor\text{S}|\text{Struct}\rceil$$

Fig. 5. Semantics of SP.

$$\frac{}{N \,|\, \mathsf{p} \triangleright_v \mathbf{0} \;\preceq\; N} \;\lfloor\text{S}|\text{PZero}\rceil \qquad \frac{}{N \,|\, \mathbf{0} \;\preceq\; N} \;\lfloor\text{S}|\text{NZero}\rceil \qquad \frac{}{\mathsf{def}\ X = B\ \mathsf{in}\ \mathbf{0} \;\preceq\; \mathbf{0}} \;\lfloor\text{S}|\text{ProcEnd}\rceil$$

$$\frac{}{\mathsf{def}\ X = B_2\ \mathsf{in}\ B_1[X] \;\preceq\; \mathsf{def}\ X = B_2\ \mathsf{in}\ B_1[B_2]} \;\lfloor\text{S}|\text{Unfold}\rceil$$

Fig. 6. Structural precongruence in SP.

EPP. The definition of EndPoint Projection relies on the merging operator $\sqcup$, which is fully defined in Figure 7.

Proof (Theorem 3). By induction on the derivation of the reduction of C, σ (completeness) or $[\![C, \sigma]\!]$ (soundness). The cases are adaptations of the proofs for [5, 6].

Proof (Theorem 4). If $N \preceq \mathbf{0}$ then the theorem clearly holds. Otherwise, the thesis follows from Theorems 1 and 3. In other words, projections of choreographies never deadlock.

$$(\mathsf{q}!\langle e\rangle; B) \ \sqcup\ \big(\mathsf{q}!\langle e\rangle; B'\big) \ =\ \mathsf{q}!\langle e\rangle; (B \sqcup B') \qquad (\mathsf{p}?; B) \ \sqcup\ \big(\mathsf{p}?; B'\big) \ =\ \mathsf{p}?; (B \sqcup B')$$

$$(\mathsf{q} \oplus l; B) \ \sqcup\ \big(\mathsf{q} \oplus l; B'\big) \ =\ \mathsf{q} \oplus l; (B \sqcup B') \qquad X \ \sqcup\ X \ =\ X$$

$$\Big(\mathsf{if}\ \mathsf{c} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2\Big) \ \sqcup\ \Big(\mathsf{if}\ \mathsf{c} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1'\ \mathsf{else}\ B_2'\Big) \ =\ \Big(\mathsf{if}\ \mathsf{c} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ (B_1 \sqcup B_1')\ \mathsf{else}\ (B_2 \sqcup B_2')\Big)$$

$$(\mathsf{def}\ X = B_2\ \mathsf{in}\ B_1) \ \sqcup\ \big(\mathsf{def}\ X = B_2'\ \mathsf{in}\ B_1'\big) \quad = \quad \big(\mathsf{def}\ X = (B_2 \sqcup B_2')\ \mathsf{in}\ (B_1 \sqcup B_1')\big)$$

$$B_1 \ \sqcup\ B_2 \ =\ B_1' \ \sqcup\ B_2' \quad \big(\text{if } B_1 \preceq B_1' \text{ and } B_2 \preceq B_2'\big)$$

$$\mathsf{p}\&\{l_i : B_i\}_{i\in J} \sqcup \mathsf{p}\&\{l_i : B_i'\}_{i\in K} = \mathsf{p}\&\big(\{l_i : (B_i \sqcup B_i')\}_{i\in J\cap K} \cup \{l_i : B_i\}_{i\in J\setminus K} \cup \{l_i : B_i'\}_{i\in K\setminus J}\big)$$

Fig. 7. Merging operator in SP.

Proof (Lemma 2). Consequence of the definitions of Amend, EPP and the semantics of MC.

Section 4 (Turing completeness of MC and SP). In the following proof, we use partial specifications of states. For example, $C, \{\mathsf{p} \mapsto v\} \to C', \{\mathsf{q} \mapsto w\}$ denotes that execution of C from *any* state where p contains value v will yield C' in *some* state where q contains value w.

Proof (Theorem 5). The proof is by induction on the definition of the set of partial recursive functions. We use a stronger induction hypothesis – namely, that if $\sigma(\mathsf{p}_i) = \ulcorner x_i \urcorner$ and $f(\tilde{x})$ is defined, then $[\![f]\!]_k^{\tilde{\mathsf{p}}\mapsto\mathsf{q}}, \sigma \to^* \sigma'$ where $\sigma'(\mathsf{p}_i) = \ulcorner x_i \urcorner$ and $\sigma'(\mathsf{q}) = \ulcorner f(\tilde{x}) \urcorner$. The extra assumption that the input values are not changed during execution is essential for the inductive step.

1. For each base case, it is straightforward to compute the sequence of reductions from the rules and the definition of the corresponding actor choreography. We exemplify this with successor.

$$[\![S]\!]_\ell^{\mathsf{p}\mapsto\mathsf{q}} : \mathsf{p}.(\mathsf{s}\cdot\mathsf{c})\ \texttt{->}\ \mathsf{q}, \{\mathsf{p} \mapsto \ulcorner x \urcorner\} \to \mathbf{0}, \begin{Bmatrix} \mathsf{p} \mapsto \ulcorner x \urcorner \\ \mathsf{q} \mapsto \ulcorner x+1 \urcorner \end{Bmatrix}$$

2. Let $h = C(f, g_1, \ldots, g_k) : \mathbb{N}^n \to \mathbb{N}$. The result follows directly from the induction hypothesis and Lemma 1.
3. Let $h = R(f, g) : \mathbb{N}^{n+1} \to \mathbb{N}$. By induction hypothesis, choreographies $[\![f]\!]_{\ell_f}^{\mathsf{p}_1,\ldots,\mathsf{p}_n\mapsto\mathsf{q}}$ and $[\![g]\!]_{\ell_g}^{\mathsf{p}_1,\ldots,\mathsf{p}_{n+2}\mapsto\mathsf{q}}$ implement f and g, respectively, for all $\tilde{\mathsf{p}}$, q, ℓ_f and ℓ_g. Again, assume first that $h(x_0, \tilde{x})$ is defined. Then:

$$\begin{aligned}
[\![h]\!]_\ell^{\mathsf{p}_0,\tilde{\mathsf{p}}\mapsto\mathsf{q}} :\quad & \mathsf{def}\ T = (\ldots)\ \mathsf{in}\ [\![f]\!]_{\ell_f}^{\tilde{\mathsf{p}}\mapsto\mathsf{q}'} \fatsemi \mathsf{r}_t.\varepsilon\ \texttt{->}\ \mathsf{r}_c;\ T, \{\mathsf{p}_i \mapsto \ulcorner x_i \urcorner\} \\
\xrightarrow{IH}{}^{*}\ & \mathsf{def}\ T = (\ldots)\ \mathsf{in}\ \mathsf{r}_t.\varepsilon\ \texttt{->}\ \mathsf{r}_c;\ T, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{q}' \mapsto \ulcorner f(\tilde{x}) \urcorner \end{Bmatrix} \\
\to\ & \mathsf{def}\ T = (\ldots)\ \mathsf{in}\ T, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{q}' \mapsto \ulcorner h(0,\tilde{x}) \urcorner \\ \mathsf{r}_c \mapsto \ulcorner 0 \urcorner \end{Bmatrix}
\end{aligned}$$

We now prove that

$$\mathsf{def}\, T = (\ldots)\,\mathsf{in}\, T, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{q}' \mapsto \ulcorner h(k,\tilde{x}) \urcorner \\ \mathsf{r}_c \mapsto \ulcorner k \urcorner \end{Bmatrix} \to^* \mathsf{def}\, T = (\ldots)\,\mathsf{in}\, T, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{q}' \mapsto \ulcorner h(k+1,\tilde{x}) \urcorner \\ \mathsf{r}_c \mapsto \ulcorner k+1 \urcorner \end{Bmatrix}$$

for all $k < x_0$. We only need to unfold T once, so we omit the $\mathsf{def}\, T = (\ldots)\,\mathsf{in}$ wrapper in the next reduction sequence.
Since $k < x_0$, the definition of T reduces to the else branch:

$$T \to^* [\![g]\!]_{\ell_g}^{\mathsf{r}_c,\mathsf{q}',\tilde{\mathsf{p}}\mapsto \mathsf{r}_t} \fatsemi \mathsf{r}_t\texttt{.c -> }\mathsf{q}';\ \mathsf{r}_c\texttt{.c -> }\mathsf{r}_t;\ \mathsf{r}_t\texttt{.(s}\cdot\texttt{c) -> }\mathsf{r}_c;\ T, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{q}' \mapsto \ulcorner h(k,\tilde{x}) \urcorner \\ \mathsf{r}_c \mapsto \ulcorner k \urcorner \end{Bmatrix}$$

$$\xrightarrow{IH}{}^{*}\ \mathsf{r}_t\texttt{.c -> }\mathsf{q}';\ \mathsf{r}_c\texttt{.c -> }\mathsf{r}_t;\ \mathsf{r}_t\texttt{.(s}\cdot\texttt{c) -> }\mathsf{r}_c;\ T, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{q}' \mapsto \ulcorner h(k,\tilde{x}) \urcorner \\ \mathsf{r}_c \mapsto \ulcorner k \urcorner \\ \mathsf{r}_t \mapsto \ulcorner g(k,h(k,\tilde{x}),\tilde{x}) \urcorner \end{Bmatrix}$$

$$\to^* T, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{q}' \mapsto \ulcorner h(k+1,\tilde{x}) \urcorner \\ \mathsf{r}_c \mapsto \ulcorner k+1 \urcorner \\ \mathsf{r}_t \mapsto \ulcorner k \urcorner \end{Bmatrix}$$

which establishes the thesis, ignoring the value in r_t.
By induction on x_0 we obtain that

$$[\![h]\!]_{\ell}^{\mathsf{p}_0,\tilde{\mathsf{p}}\mapsto \mathsf{q}}, \{\mathsf{p}_i \mapsto \ulcorner x_i \urcorner\} \to^* \mathsf{def}\, T = (\ldots)\,\mathsf{in}\, T, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{q}' \mapsto \ulcorner h(x_0,\tilde{x}) \urcorner \\ \mathsf{r}_c \mapsto \ulcorner x_0 \urcorner \end{Bmatrix}$$

$$\xrightarrow{(1)} \mathsf{def}\, T = (\ldots)\,\mathsf{in}\, \mathsf{q}'\texttt{.c -> q};\ \mathbf{0}, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{q}' \mapsto \ulcorner h(x_0,\tilde{x}) \urcorner \\ \mathsf{r}_c \mapsto \ulcorner x_0 \urcorner \end{Bmatrix}$$

$$\to^* \mathsf{def}\, T = (\ldots)\,\mathsf{in}\, \mathbf{0}, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{q}' \mapsto \ulcorner h(x_0,\tilde{x}) \urcorner \\ \mathsf{r}_c \mapsto \ulcorner x_0 \urcorner \\ \mathsf{q} \mapsto \ulcorner h(x_0,\tilde{x}) \urcorner \end{Bmatrix}$$

and the last process is equivalent to $\mathbf{0}$. In (1) we used the fact that the contents of r_c and p_0 are both equal to $\ulcorner x_0 \urcorner$.
If $h(x_0,\tilde{x})$ is not defined, there are two possible cases. If $f(\tilde{x})$ is not defined, then $[\![f]\!]_{\ell_f}^{\tilde{\mathsf{p}}\mapsto \mathsf{q}'}$ diverges from any state where each p_i contains $\ulcorner x_i \urcorner$, whence so does $[\![h]\!]_{\ell}^{p_0,\tilde{\mathsf{p}}\mapsto \mathsf{q}}$ by Lemma 1 and rule $\lfloor \mathrm{C}|\mathrm{Ctx} \rceil$. If $g(k,h(k,\tilde{x}),\tilde{x})$ is undefined for some $k < x_0$, then divergence is likewise obtained from the fact that $[\![g]\!]_{\ell_g}^{\mathsf{r}_c,\mathsf{q}',\tilde{\mathsf{p}}}$ diverges from any state where r_c contains $\ulcorner k \urcorner$, q' contains $\ulcorner h(k,\tilde{x}) \urcorner$, and p_i contains $\ulcorner x_i \urcorner$.

4. The case where $h = M(f) : \mathbb{N}^n \to \mathbb{N}$ is very similar, the auxiliary result now stating that

$$\mathsf{def}\, T = (\ldots)\,\mathsf{in}\, T, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{r}_c \mapsto \ulcorner k \urcorner \end{Bmatrix} \to^* \mathsf{def}\, T = (\ldots)\,\mathsf{in}\, T, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{r}_c \mapsto \ulcorner k+1 \urcorner \end{Bmatrix}$$

as long as $f(\tilde{x}, k)$ is defined and different from 0.
The only new aspect is that non-termination may arise from the fact that $f(\tilde{x}, k)$ is defined and non-zero for every $k \in \mathbb{N}$, in which case we get an infinite reduction sequence

$$\begin{aligned}
[\![h]\!]_\ell^{\tilde{\mathsf{p}} \to \mathsf{q}}, \{\mathsf{p}_i \mapsto \ulcorner x_i \urcorner\} &\to^* \mathsf{def}\, T = (\ldots)\, \mathsf{in}\, T, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{r}_c \mapsto \ulcorner 0 \urcorner \end{Bmatrix} \\
&\to^* \mathsf{def}\, T = (\ldots)\, \mathsf{in}\, T, \begin{Bmatrix} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{r}_c \mapsto \ulcorner n \urcorner \end{Bmatrix} \\
&\to^* \ldots
\end{aligned}$$

□

Proof (Corollary 3). Let $f \in \mathcal{R}$. By Theorem 5, $C = [\![f]\!]^{\tilde{\mathsf{p}} \mapsto \mathsf{q}}$ for any suitable $\tilde{\mathsf{p}}$ and q implements f. By Lemma 2, $\mathsf{Amend}(C)$ is projectable and operationally equivalent to C. Hence, by Theorem 3, $[\![\mathsf{Amend}(C), \sigma]\!]$ is a term in SP that correctly implements f.

Proof (Theorem 6). Direct consequence of Theorem 2 and the definition of $\{\!\{\}\!\}$.

Section 6 (Minimality and label selections). We begin by sketching how selections can be encoded using conditionals, communications and auxiliary processes.

Consider a choreography where p makes a choice depending on the value stored by q, and then r needs to be notified of the result (because, e.g., it is involved in further communications in one or both of the branches). As an example, we take C to be the choreography $\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ \mathsf{p} \texttt{ -> } \mathsf{r}[\mathrm{L}]; C_1\ \mathsf{else}\ \mathsf{p} \texttt{ -> } \mathsf{r}[\mathrm{R}]; C_2$, where r has different behaviours in C_1 and C_2.

In order to eliminate the label selections, r must be able to perform a conditional that is guaranteed to choose the same branch as taken by p. With this in mind, we introduce an auxiliary process p^* and add communications from p to p^* of ε (then branch) or $\mathsf{s} \cdot \mathsf{c}$ (else branch). Then r can recover this information by first setting its contents to ε and then comparing them with p^*; this requires another auxiliary process r^* to store r's value in the meantime. Furthermore, even though we know at a global level what the result of the comparison will be, the EPP (in particular, merging) demands that we consider both branches in both cases. We therefore rewrite C as follows.

$$\begin{aligned}
&\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ \Big(\mathsf{p}.\varepsilon \texttt{ -> } \mathsf{p}^*;\ \mathsf{r}.\mathsf{c} \texttt{ -> } \mathsf{r}^*;\ \mathsf{r}^*.\varepsilon \texttt{ -> } \mathsf{r};\ \mathsf{if}\ \mathsf{r} \overset{\leftarrow}{=} \mathsf{p}^*\ \mathsf{then}\ (\mathsf{r}^*.\mathsf{c} \texttt{ -> } \mathsf{r};\ C_1)\ \mathsf{else}\ (\mathsf{r}^*.\mathsf{c} \texttt{ -> } \mathsf{r};\ C_2)\Big) \\
&\qquad \mathsf{else}\ \Big(\mathsf{p}.\mathsf{s} \cdot \mathsf{c} \texttt{ -> } \mathsf{p}^*;\ \mathsf{r}.\mathsf{c} \texttt{ -> } \mathsf{r}^*;\ \mathsf{r}^*.\varepsilon \texttt{ -> } \mathsf{r};\ \mathsf{if}\ \mathsf{r} \overset{\leftarrow}{=} \mathsf{p}^*\ \mathsf{then}\ (\mathsf{r}^*.\mathsf{c} \texttt{ -> } \mathsf{r};\ C_1)\ \mathsf{else}\ (\mathsf{r}^*.\mathsf{c} \texttt{ -> } \mathsf{r};\ C_2)\Big)
\end{aligned}$$

Observe that the behaviour of the processes not performing conditionals (p^* and r^*) is the same in all branches, while p and r have two possible behaviours that are independent of each other's choices. This guarantees that merging will work for all projections.

Formally defining this process is however complicated by the possibility of nested conditionals, requiring one to keep track of the auxiliary processes used for each choice. Furthermore, this produces an exponential increase in the size of the choreography (although, interestingly, not of its projections), while also adding new communications that make its structure less clear.

Conditionals. We conclude with the proofs of the results on decidability of termination with restricted or no conditionals.

Proof (Theorem 7). The second part is straightforward, since rule $\lfloor$C$|$Cond$\rceil$ is the only rule whose conclusion depends on the state.

For the first part, we reduce termination to a decidable graph problem. Define $\mathcal{G}_C = \langle V, E\rangle$ to be the graph whose set of vertices V contains C and $\mathbf{0}$, and is closed under the following rules.

- if $\eta; C \in V$, then $C \in V$;
- if $\mathsf{def}\, X = C_2 \,\mathsf{in}\, C_1 \in V$, then $C_1 \in V$;
- if $\mathsf{def}\, X = C_2 \,\mathsf{in}\, \eta; C_1 \in V$, then $\mathsf{def}\, X = C_2 \,\mathsf{in}\, C_1 \in V$;
- if $\mathsf{def}\, X = C_2 \,\mathsf{in}\, \eta; X \in V$, then $\mathsf{def}\, X = C_2 \,\mathsf{in}\, \eta; C_2 \in V$.

This set is finite: all rules add smaller choreographies to V, except the last one, which can only be applied once for each variable in C.

There is an edge between C_1 and C_2 iff $C_1, \sigma \to C_2, \sigma'$ for some σ, σ' without using rule $\lfloor$C$|$Eta-Eta$\rceil$. This is decidable, as the possibility of a reduction does not depend on the state (as observed above). Also, if there is a reduction from C_1, then there is always an edge from C_1 in the graph, as swapping communication actions cannot unblock execution.

Then C terminates iff there is a path from C to $\mathbf{0}$, which can be decided in finite time, as $\mathcal{G}_C$ is finite.

Proof (Theorem 8). We first show that termination is decidable for processes of the form $\mathsf{def}\, X = C_2 \,\mathsf{in}\, X$ and comparison with 0. The proof is by induction on the number of recursive definitions in C_2.

Consider first the case where C_2 has no recursive definitions, and let P be the set of all process names occurring in C_2. We define an equivalence relation on states by

$$\sigma \equiv_P \sigma' \text{ iff } (\forall \mathsf{p} \in P,\, \sigma(\mathsf{p}) = \varepsilon \text{ iff } \sigma'(\mathsf{p}) = \varepsilon)\,.$$

The vertices of the graph are the $2^{|P|}$ equivalence classes of states wrt $\equiv_P$, plus $\top$. Note that $\equiv_P$ is compatible with the transition relation excluding rule $\lfloor$C$|$Eta-Eta$\rceil$: for any choreography C using only process names in P, $\sigma_1 \equiv \sigma_2$ and $C, \sigma_i \to \sigma_i'$, then $\sigma_1' \equiv \sigma_2'$.

The edges in the graph are defined as follows. There is an edge from $[\sigma]$ to $[\sigma']$ if $C_2, \sigma \to X, \sigma'$, and there is an edge from $[\sigma]$ to $\top$ if $C_2, \sigma \to \mathbf{0}, \sigma'$ or $C_2, \sigma \to Y, \sigma'$ for some $Y \neq X$. This is constructible, as reductions in C_2 are always finite, and well-defined, as alternative reduction paths always end in the same state.

Since reductions are deterministic and $\equiv_P$ is compatible with reduction, every node has exactly one edge leaving from it, except for $\top$. Therefore, we can decide if $\mathsf{def}\, X = C_2 \,\mathsf{in}\, X$ terminates from an initial state σ by simply following the path starting at σ and returning Yes if we reach $\top$ and No if we pass some node twice. This procedure terminates, as the graph is finite.

For the inductive step, proceed as above but add an extra node to the graph, labeled $\perp$. When constructing the edges in the graph, if C_2 reduces to a variable Y different than X, we split into two cases. If Y is not bound in C_2, we proceed as in the previous case. If Y is bound, then we apply the induction hypothesis to the choreography $\mathsf{def}\, Y = C_Y \,\mathsf{in}\, Y$ (where $Y = C_Y$ is the same as in C_2) to decide whether the reduction from Y will terminate; if this is not the case, we add an edge to $\perp$, otherwise we proceed with the simulation. At the end, we return No in the case that the path followed leads to $\perp$.

The general case follows, as C has the same behaviour as $\mathsf{def}\, X = C \,\mathsf{in}\, X$ for some X not occurring in C.

If we allow comparisons with other values, the strategy is the same, but the relation $\equiv_P$ has to be made finer. The key observation is that only a finite number of values can be used in comparisons, so we can identify states if they only differ on processes whose contents are larger than all values used in conditionals.

For convenience, we split the proof of Theorem 9 in several lemmas.

Lemma 4. *If* $\mathsf{q} \in \tilde{\mathsf{r}}$*, then* $[\![(\!| S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{R}, C_2) |\!)_1]\!]_{\mathsf{q}} = [\![(\!| S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{R}, C_2) |\!)_2]\!]_{\mathsf{q}}$.

Proof. By induction on the length of $\tilde{\mathsf{r}}$. If $\tilde{\mathsf{r}} = \emptyset$, then the result is vacuously true. If $\tilde{\mathsf{r}}$ does not start with q, then the result follows trivially from the induction hypothesis. So consider the case where $\tilde{\mathsf{r}} = \mathsf{q} :: \tilde{\mathsf{r}'}$. In this case, $(\!| S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{R}, C_2) |\!)$ unfolds to

$$\begin{aligned} \langle &\mathsf{q.c} \texttt{ -> } \mathsf{q}^\bullet;\ \mathsf{p.}\varepsilon \texttt{ -> } \mathsf{q};\ \mathsf{if}\ \mathsf{q} \overset{\leftarrow}{=} \mathsf{z}\ \mathsf{then}\ \mathsf{q}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{q};\ (\!| S(\mathsf{p}, \tilde{\mathsf{r}'}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}'}, \mathrm{R}, C_2) |\!)_1 \\ &\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{q}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{q};\ (\!| S(\mathsf{p}, \tilde{\mathsf{r}'}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}'}, \mathrm{R}, C_2) |\!)_2, \\ &\mathsf{q.c} \texttt{ -> } \mathsf{q}^\bullet;\ \mathsf{p.sc} \texttt{ -> } \mathsf{q};\ \mathsf{if}\ \mathsf{q} \overset{\leftarrow}{=} \mathsf{z}\ \mathsf{then}\ \mathsf{q}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{q};\ (\!| S(\mathsf{p}, \tilde{\mathsf{r}'}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}'}, \mathrm{R}, C_2) |\!)_1 \\ &\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{q}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{q};\ (\!| S(\mathsf{p}, \tilde{\mathsf{r}'}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}'}, \mathrm{R}, C_2) |\!)_2 \rangle \end{aligned}$$

and the endpoint projections of both choreographies for q become

$$\begin{aligned} &\mathsf{q}^\bullet!\langle \mathsf{c}\rangle;\ \mathsf{p}?;\ \mathsf{if}\ \mathsf{c} \overset{\leftarrow}{=} \mathsf{z}\ \mathsf{then}\ \mathsf{q}^\bullet?;\ [\![(\!| S(\mathsf{p}, \tilde{\mathsf{r}'}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}'}, \mathrm{R}, C_2) |\!)_1]\!]_{\mathsf{q}} \\ &\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{q}^\bullet?;\ [\![(\!| S(\mathsf{p}, \tilde{\mathsf{r}'}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}'}, \mathrm{R}, C_2) |\!)_2]\!]_{\mathsf{q}} \end{aligned}$$

which are defined and identical.

Lemma 5. *If* $[\![(\!| C_1 |\!)]\!]_{\mathsf{q}} = [\![(\!| C_2 |\!)]\!]_{\mathsf{q}}$ *and* $\mathsf{p} \neq \mathsf{q} \notin \tilde{\mathsf{r}}$*, then* $[\![(\!| S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{R}, C_2) |\!)_1]\!]_{\mathsf{q}} = [\![(\!| S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{R}, C_2) |\!)_2]\!]_{\mathsf{q}}$.

Proof. By induction on the length of $\tilde{\mathsf{r}}$. If $\tilde{\mathsf{r}} = \emptyset$, then the result reduces to the hypothesis. Otherwise, $(\!| S(\mathsf{p}, \mathsf{r} :: \tilde{\mathsf{r}}, \mathrm{L}, C_1), S(\mathsf{p}, \mathsf{r} :: \tilde{\mathsf{r}}, \mathrm{R}, C_2) |\!)$ unfolds to

$$\begin{aligned} \langle &\mathsf{r.c} \texttt{ -> } \mathsf{r}^\bullet;\ \mathsf{p.}\varepsilon \texttt{ -> } \mathsf{r};\ \mathsf{if}\ \mathsf{r} \overset{\leftarrow}{=} \mathsf{z}\ \mathsf{then}\ \mathsf{r}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{r};\ (\!| S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{R}, C_2) |\!)_1 \\ &\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{r}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{r};\ (\!| S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{R}, C_2) |\!)_2, \\ &\mathsf{r.c} \texttt{ -> } \mathsf{r}^\bullet;\ \mathsf{p.sc} \texttt{ -> } \mathsf{r};\ \mathsf{if}\ \mathsf{r} \overset{\leftarrow}{=} \mathsf{z}\ \mathsf{then}\ \mathsf{r}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{r};\ (\!| S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{R}, C_2) |\!)_1 \\ &\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{r}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{r};\ (\!| S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{R}, C_2) |\!)_2 \rangle \end{aligned}$$

and since $\mathsf{q} \neq \mathsf{r}$ there are three cases to consider.

– q is $\mathsf{r}^{\bullet}$: then both endpoint projections become

$$\mathsf{r}?;\ (\mathsf{r}!\langle\mathsf{c}\rangle;\ [\![(\!|S(\mathsf{p},\tilde{\mathsf{r}},\text{L},C_1),S(\mathsf{p},\tilde{\mathsf{r}},\text{R},C_2)|\!)_1]\!]_{\mathsf{q}^{\bullet}})\sqcup(\mathsf{r}!\langle\mathsf{c}\rangle;\ [\![(\!|S(\mathsf{p},\tilde{\mathsf{r}},\text{L},C_1),S(\mathsf{p},\tilde{\mathsf{r}},\text{R},C_2)|\!)_2]\!]_{\mathsf{q}^{\bullet}})$$

and by induction hypothesis the two processes being merged are identical, so the result is defined.

– q is z: then both endpoint projections become

$$\mathsf{r}!\langle\mathsf{c}\rangle;\ [\![(\!|S(\mathsf{p},\tilde{\mathsf{r}},\text{L},C_1),S(\mathsf{p},\tilde{\mathsf{r}},\text{R},C_2)|\!)_1]\!]_{\mathsf{z}} \sqcup [\![(\!|S(\mathsf{p},\tilde{\mathsf{r}},\text{L},C_1),S(\mathsf{p},\tilde{\mathsf{r}},\text{R},C_2)|\!)_2]\!]_{\mathsf{z}}$$

and again by induction hypothesis the two processes being merged are identical, so the result is defined.

– q is another process: then both endpoint projections become simply

$$[\![(\!|S(\mathsf{p},\tilde{\mathsf{r}},\text{L},C_1),S(\mathsf{p},\tilde{\mathsf{r}},\text{R},C_2)|\!)_1]\!]_{\mathsf{q}} \sqcup [\![(\!|S(\mathsf{p},\tilde{\mathsf{r}},\text{L},C_1),S(\mathsf{p},\tilde{\mathsf{r}},\text{R},C_2)|\!)_2]\!]_{\mathsf{q}}$$

whence the induction hypothesis guarantees again that the two processes being merged are identical, so the result is defined.

Lemma 6. *For every choreography C in MC^{-} and every process r, $[\![(\!|\mathsf{Amend}(C)|\!)]\!]_{\mathsf{r}}$ is defined.*

Proof. By structural induction on $\mathsf{Amend}(C)$. The only non-trivial case is that where $\mathsf{Amend}(C)$ is $\mathsf{if}\ \mathsf{p} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ S(\mathsf{p},\tilde{\mathsf{r}},\text{L},C_1)\ \mathsf{else}\ S(\mathsf{p},\tilde{\mathsf{r}},\text{R},C_2)$, where we need to consider the possible cases for r. If $\mathsf{r} = \mathsf{p}$, then the induction hypothesis establishes the thesis with induction over $\tilde{\mathsf{r}}$. If $\mathsf{r} \in \tilde{\mathsf{r}}$, then Lemma 4 guarantees that both branches of the conditional will be equal, hence the endpoint projection is again defined. Finally, if $\mathsf{r} \notin \tilde{\mathsf{r}}$, then by definition of amendment $[\![\mathsf{Amend}(C_1)]\!]_{\mathsf{r}} = [\![\mathsf{Amend}(C_2)]\!]_{\mathsf{r}}$, whence Lemma 5 applies and establishes the thesis as in the previous case.

Proof (Proof (Theorem 9).). Straightforward consequence of Lemma 6.