A Core Model for Choreographic Programming

Luís Cruz-Filipe, Fabrizio Montesi

University of Southern Denmark, Department of Mathematics and Computer Science, Campusvej 55, 5230 Odense M, Denmark

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.

Keywords: Choreography, Computability, Process Calculi

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 (Lu et al., 2008); (ii) combining different communication protocols in a single application is a major source of mistakes (Leesatapornwongsa et al., 2016).

The paradigm of Choreographic Programming (Montesi, 2013) 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"

Email addresses: `lcf@imada.sdu.dk` (Luís Cruz-Filipe), `fmontesi@imada.sdu.dk` (Fabrizio Montesi)

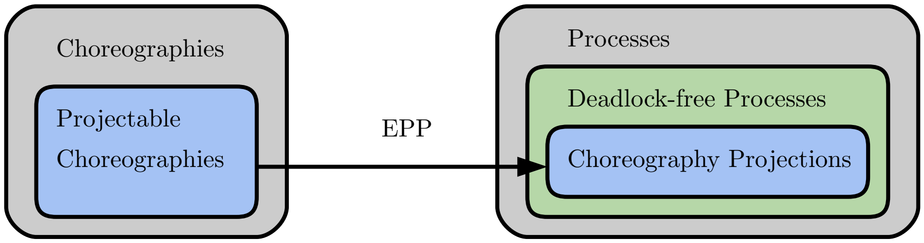

Figure 1: Choreographic Programming

notation of security protocols (Needham and Schroeder, 1978). 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 (Carbone et al., 2012; Carbone and Montesi, 2013).

So far, work on choreographic programming focused on features of practical value – including web services (Carbone et al., 2012), multiparty sessions (Carbone and Montesi, 2013; Chor, 2016), modularity (Montesi and Yoshida, 2013), and runtime adaptation (Dalla Preda et al., 2015). 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 how much 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.

Yet, 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 (Carbone et al., 2012; Carbone and Montesi, 2013). 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 1).

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 that CC is a simple common setting for the 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 (Carbone et al., 2012), which is typically one of the most complicated steps in EPP. Our EPP for MC enjoys an elegant definition.

Structure of the paper. Section 2 defines Core Choreographies (CC) and its subcalculus of Minimal Choreographies (MC). Section 3 introduces Stateful Processes (SP), our target process model, and its sublanguage of Minimal Processes (MP), together with an EndPoint Projection (EPP) from CC to SP (and from MC to MP). We also show that every unprojectable choreography in CC can be amended (transformed into a projectable one) by adding only label selections. We prove that CC and its set of choreography projections are Turing complete in Section 4. In Section 5 we discuss label selections, and show that they can be encoded by communications; this yields an amendment strategy for MC. In Section 6, we show that all the remaining primitives of CC are necessary to achieve Turing completeness, and discuss the implications of our work for other choreography languages and process calculi – in particular, we identify a Turing complete and deadlock-free fragment of value-passing CCS. Related work and discussion are given in Section 7.

$$C ::= \eta; C \mid \mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1 \mid X \mid \mathbf{0}$$

$$\eta ::= \mathsf{p}.e\ \texttt{->}\ \mathsf{q} \mid \mathsf{p}\ \texttt{->}\ \mathsf{q}[l] \qquad e ::= \varepsilon \mid \mathsf{c} \mid \mathsf{s} \cdot \mathsf{c} \qquad l ::= \textsc{l} \mid \textsc{r}$$

Figure 2: Core Choreographies, Syntax.

The current article extends material previously presented in (Cruz-Filipe and Montesi, 2016b) with detailed proofs of all results, a complete presentation of the semantics for all calculi (namely, including the full definition of structural precongruence for CC and SP), full definitions of EPP and the merge operator, and selection elimination (Section 5). Moreover, the detailed examples in Section 4, the proofs of the main results (in particular, Theorems 5, 7, 8 and 9) and the encoding of CC into Channel Choreographies (Section 6.2) have not been published previously.

2. Core Choreographies and Minimal Choreographies

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

2.1. Syntax of CC

The syntax of CC is displayed in Figure 2, where C ranges over choreographies. 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.

2.2. Semantics of CC

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

$$\frac{v = e[\sigma(\mathsf{p})/\mathsf{c}]}{\mathsf{p}.e \text{ -> } \mathsf{q}; C, \sigma \to C, \sigma[\mathsf{q} \mapsto v]} \lfloor \text{C|Com} \rceil \qquad \frac{}{\mathsf{p} \text{ -> } \mathsf{q}[l]; C, \sigma \to C, \sigma} \lfloor \text{C|Sel} \rceil$$

$$\frac{i = 1 \text{ if } \sigma(\mathsf{p}) = \sigma(\mathsf{q}),\ i = 2 \text{ o.w.}}{\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \sigma \to C_i, \sigma} \lfloor \text{C|Cond} \rceil$$

$$\frac{C_1 \preceq C_2 \quad C_2, \sigma \to C_2', \sigma' \quad C_2' \preceq C_1'}{C_1, \sigma \to C_1', \sigma'} \lfloor \text{C|Struct} \rceil$$

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

Figure 3: Core Choreographies, Semantics.

$\ulcorner n \urcorner = \mathsf{s}^n \cdot \varepsilon$. The reduction relation $\to$ is defined by the rules given in Figure 3. below and closed under structural precongruence $\preceq$.

Rule $\lfloor$C|Com$\rceil$ models a value communication $\mathsf{p}.e$ -> q. In the premise, 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. Rule $\lfloor$C|Sel$\rceil$ is similar, but does not change σ. Rules $\lfloor$C|Struct$\rceil$ and $\lfloor$C|Ctx$\rceil$ are standard.

The semantics of CC also allows non-interfering actions to be executed in any order. This is achieved by means of a *structural precongruence*, $\preceq$, which is the smallest precongruence satisfying the rules in Figure 4. We write $C \equiv C'$ for $C \preceq C'$ and $C' \preceq C$. The key ingredient is: if $\mathsf{pn}(\eta) \cap \mathsf{pn}(\eta') = \emptyset$, then $\eta; \eta' \ \equiv \ \eta'; \eta$.

In rule $\lfloor$C|Unfold$\rceil$, we write $C_1[X]$ to indicate that the call term X occurs in C_1, and replace it with the body of the recursive procedure on the right. In rules $\lfloor$C|Eta-Eta$\rceil$, $\lfloor$C|Eta-Cond$\rceil$ and $\lfloor$C|Cond-Cond$\rceil$, we swap two terms describing actions performed by independent processes, modelling concurrent process execution; where the auxiliary function $\mathsf{pn}(C)$ returns the set of process names in C. They observe the same conditions as in previous choreography models (where these rules are used for the swapping relation $\simeq_C$, e.g., in (Carbone and Montesi, 2013)).

2.3. Label Selection and Minimal Choreographies

To the reader unfamiliar with choreographies, the role of selection – p -> $\mathsf{q}[l]$ – may be unclear at this point. In existing choreography calculi, they are crucial in making choreographies projectable, as we illustrate with an example.

Example 1. *Consider the following choreography.*

$$C = \mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ (\mathsf{p.c} \text{ -> } \mathsf{r}; \mathbf{0})\ \mathsf{else}\ (\mathsf{r.c} \text{ -> } \mathsf{p}; \mathbf{0})$$

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*

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

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

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

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

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

Figure 4: Core Choreographies, Structural precongruence $\preceq$.

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 have a problem because we are asking* r *to act differently based on a decision made by another process,* p*, and there is no propagation of this decision from* p *to* r *(either directly or indirectly, through other processes). We can easily fix the example by adding selections:*

$$C' = \mathsf{if}\ \mathsf{p} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ (\mathsf{p} \texttt{ -> } \mathsf{r}[\mathrm{L}]; \mathsf{p.c} \texttt{ -> } \mathsf{r}; \mathbf{0})\ \mathsf{else}\ (\mathsf{p} \texttt{ -> } \mathsf{r}[\mathrm{R}]; \mathsf{r.c} \texttt{ -> } \mathsf{p}; \mathbf{0})\,.$$

Now, p *tells* r *about its choice by sending a different label. This intuition will be formalised in our definition of EndPoint Projection in § 3.3. The choreography C (without label selections) is not projectable, whereas C' is.* □

One of the main results in this paper is showing that the same effect can actually be achieved in a language without label selections. For this purpose, we introduce a fragment of CC, which we call Minimal Choreographies (MC). The syntax of MC is the same as that of CC, except that the action p -> q$[l]$ is disallowed; the semantics of MC is the same as that of CC, disregarding the rules that pertain to terms outside the language.

2.4. Properties

In the remainder of this section we discuss some properties of Core Choreographies. Unless otherwise stated, those properties also hold for Minimal Choreographies.

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

Theorem 1 (Deadlock-freedom by design)**.** *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).*

Theorem 1. Direct consequence of the definition of the semantics. □ □

Using this theorem, in § 3.3 we prove that the process implementations obtained by projecting choreographies is deadlock-free.

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).

In later sections, we need to characterize choreographies that are equivalent wrt a set of processes. We use the state function σ for this purpose.

Definition 2 (Computational equivalence)**.** *Two states σ_1, σ_2 are* equivalent *wrt a set of process names $\tilde{\mathsf{p}}$, written $\sigma_1 \equiv_{\tilde{\mathsf{p}}} \sigma_2$, if $\sigma_1(\mathsf{p}) = \sigma_2(\mathsf{p})$ for every $\mathsf{p} \in \tilde{\mathsf{p}}$.*

Two choreographies C_1 and C_2 are equivalent *wrt a set of process names $\tilde{\mathsf{p}}$ if: whenever $\sigma_1 \equiv_{\tilde{\mathsf{p}}} \sigma_2$, if $C_1, \sigma_1 \to^* \mathbf{0}, \sigma_1'$ then $C_2, \sigma_2 \to^* \mathbf{0}, \sigma_2'$ with $\sigma_1' \equiv_{\tilde{\mathsf{p}}} \sigma_2'$, and conversely.*

Throughout this paper, we focus on choreographies with only one exit point (a single occurence of $\mathbf{0}$). When C has a single exit point, we write $C \fatsemi C'$ for the choreography obtained by replacing $\mathbf{0}$ in C with C'. (Requiring C to have a single exit point makes this construction linear in the sizes of C and C', and simplifies its theoretical analysis.) This does not add expressivity to CC, but it allows for the usage of macros (as in the examples below). Then, $C \fatsemi C'$ behaves as a "sequential composition" of C and C', as induction over C shows.

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 $C \fatsemi C', \sigma \not\to^* \mathbf{0}$.*

Proof. Straightforward by structural induction on C. □

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 3. *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:*

- $\widetilde{\sigma'_2} = \widetilde{\sigma} \setminus \widetilde{\sigma'_1}$*;*
- $\widetilde{\sigma'_1(\mathsf{p})} = \widetilde{\sigma_1(\mathsf{p})}$ *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 a constant sequence for all* $\mathsf{r} \notin \tilde{\mathsf{p}} \cup \tilde{\mathsf{q}}$*.*

Definition 4 (Parallel Run)**.** *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* $C_1 \,⨟\, C_2$ *if: whenever* $C_i \xrightarrow{\tilde{\sigma_i}}{}^* \mathbf{0}$*, then* $C_1 \,⨟\, C_2 \xrightarrow{\tilde{\sigma}}{}^* \mathbf{0}$ *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 (Parallelisation)**.** *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* $C_1 \,⨟\, C_2$*.*

Proof. The converse implication is a consequence of (1) in Lemma 1. The direct implication follows by induction over C_1. □

Definition 4 and Theorem 2 straightforwardly generalise to an arbitrary number of processes. We provide an example of such parallel behaviour in Theorem 6.

2.5. Examples

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

Example 2. *The macro* $\textsc{inc}(\mathsf{p},\mathsf{t})$ *increments the value of* p *using an auxiliary process* t*.*

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

Using 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 *as auxiliary processes. The compositional usage of* INC *is allowed, as it has exactly one exit point.*

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

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

Figure 5: Stateful Processes, syntax.

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* $\lfloor C|$*Eta-Eta*$\rceil$ *for* $\preceq$ *repeatedly we can check that:*

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

□

3. Stateful Processes, Minimal Processes and EndPoint Projections

We present Stateful Processes (SP), our target process model, and define an EndPoint Projection (EPP) that synthesises process implementations from choreographies in CC. By restricting SP adequately, we obtain a target process calculus for MC, which we call Minimal Processes (MP), with a corresponding, simpler, EPP.

3.1. Syntax of SP and MP

The syntax of SP is given in Figure 5.

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.

We comment on behaviours. 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 $\mathsf{if}\ \mathsf{c} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2$, 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.

The syntax of MP is obtained by disallowing the terms $\mathsf{q}\oplus l; B$ and $\mathsf{p}\&\{l_i : B_i\}_{i\in I}$.

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

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

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

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

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

Figure 6: Stateful Processes, Semantics.

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

$$\frac{}{\mathsf{p} \triangleright_v \mathbf{0} \;\preceq\; \mathbf{0}} \;\lfloor \mathrm{S|PZero} \rceil \qquad \frac{}{N \,|\, \mathbf{0} \;\preceq\; N} \;\lfloor \mathrm{S|NZero} \rceil$$

$$\frac{}{\mathsf{def}\ X = B\ \mathsf{in}\ \mathbf{0} \;\preceq\; \mathbf{0}} \;\lfloor \mathrm{S|ProcEnd} \rceil$$

Figure 7: Stateful Processes, Structural Precongruence.

3.2. Semantics

The reduction rules for SP are mostly standard, from process calculi, and are included in Figure 6. The key difference from CC is that execution is now distributed over processes. 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 (Honda et al., 1998), 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, and use a structural precongruence defined in Figure 7, 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 5 (Function implementation in SP)**.** *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*

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

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

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

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

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

Figure 8: Core Choreographies, Behaviour Projection.

$N \preceq (\prod_{i\in[1,n]} \mathsf{p}_i \triangleright_{v_i} B_i) \mid \mathsf{q} \triangleright_w B' \mid N'$ *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.*

3.3. EndPoint 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 8. Selections are projected similarly to communications, and $\mathbf{0}$ is projected to $\mathbf{0}$. 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 $\mathsf{p}.e \texttt{ -> } \mathsf{q}$, 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 (Carbone et al., 2012).

$$(\mathsf{q}!\langle e\rangle; B) \quad \sqcup \quad (\mathsf{q}!\langle e\rangle; B') \quad = \quad \mathsf{q}!\langle e\rangle; (B \sqcup B')$$

$$(\mathsf{p}?; B) \quad \sqcup \quad (\mathsf{p}?; B') \quad = \quad \mathsf{p}?; (B \sqcup B')$$

$$(\mathsf{q} \oplus l; B) \quad \sqcup \quad (\mathsf{q} \oplus l; B') \quad = \quad \mathsf{q} \oplus l; (B \sqcup B')$$

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

$$\Big(\mathsf{if}\ \mathsf{c} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2\Big) \quad \sqcup \quad \Big(\mathsf{if}\ \mathsf{c} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B'_1\ \mathsf{else}\ B'_2\Big) \quad = \quad \Big(\mathsf{if}\ \mathsf{c} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ (B_1 \sqcup B'_1)\ \mathsf{else}\ (B_2 \sqcup B'_2)\Big)$$

$$X \quad \sqcup \quad X \quad = \quad X$$

$$(\mathsf{def}\ X = B_2\ \mathsf{in}\ B_1) \quad \sqcup \quad (\mathsf{def}\ X = B'_2\ \mathsf{in}\ B'_1) \quad = \quad (\mathsf{def}\ X = (B_2 \sqcup B'_2)\ \mathsf{in}\ (B_1 \sqcup B'_1))$$

$$B_1 \quad \sqcup \quad B_2 \quad = \quad B'_1 \sqcup B'_2 \quad (\text{if } B_1 \preceq B'_1 \text{ and } B_2 \preceq B'_2)$$

Figure 9: Core Choreographies, Merge Operator in Behaviour Projection.

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. The formal definition is found in Figure 9. Merging is a homomorphic binary operator; for all terms but branchings it requires isomorphism, $\mathsf{q}!\langle e\rangle; B \sqcup \mathsf{q}!\langle e\rangle; B' = \mathsf{q}!\langle e\rangle; (B \sqcup B')$. The only case where branching terms can have unmergeable continuations is when they are guarded by distinct labels, in which case merge returns a larger branching including all options (merging branches with the same label).

Merging explains the role of selections in CC, common in choreography models (Coppo et al., 2016; Carbone et al., 2012; Carbone and Montesi, 2013; Honda et al., 2008; Dalla Preda et al., 2015; Qiu et al., 2007). Recall the choreographies from Example 1. In choreography C, 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). Choreography C' 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 6 (EPP from CC to SP)**.** *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.*

$$[\![C,\sigma]\!] = \prod_{\mathsf{p}\in\mathsf{pn}(C)} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} [\![C]\!]_{\mathsf{p}}$$

The EPP from MC to MP is defined by restricting the EPP from CC to SP to the relevant cases. 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 the following, simpler, one.

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

Since the state function σ is 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.*

Example 3. *Recall the definition of* $\textsc{inc}(\mathsf{p},\mathsf{t})$.

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

Given any σ*, the EPP of* $\textsc{inc}(\mathsf{p},\mathsf{t})$ *is:*

$$[\![\textsc{inc}(\mathsf{p},\mathsf{t}),\sigma]\!] = \ \mathsf{p} \triangleright_{\sigma(\mathsf{p})} \mathsf{t}!\langle \mathsf{c} \rangle; \mathsf{t}?; \mathbf{0} \ \mid\ \mathsf{t} \triangleright_{\sigma(\mathsf{t})} \mathsf{p}?; \mathsf{p}!\langle \mathsf{s}\cdot\mathsf{c} \rangle; \mathbf{0}$$

EPP guarantees the following operational correspondence.

Theorem 3 (Operational Correspondence (CC $\leftrightarrow$ SP))**.** *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 $[\![C',\sigma']\!] \prec N$.*

Proof. By induction on the derivation of the reduction of C,σ (completeness) or $[\![C,\sigma]\!]$ (soundness). The cases are adaptations of the proofs for (Carbone et al., 2012; Carbone and Montesi, 2013). □

The *pruning relation* $\prec$ (Carbone et al., 2012; Carbone and Montesi, 2013) 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 (Carbone et al., 2012; Lanese et al., 2008; Dalla Preda et al., 2015). As a consequence of Theorems 1 and 3, choreography projections never deadlock.

Theorem 4 (Deadlock-freedom by construction)**.** *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).*

Proof. 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. □

3.4. 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 7 (Amendment)**.** *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 $\mathsf{if}\ \mathsf{p} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ 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 $\mathsf{if}\ \mathsf{p} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ in C is replaced with:*

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

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

Lemma 2 (Amendment Lemma)**.** *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 $\mathsf{Amend}(C), \sigma \to^* \mathsf{Amend}(C'), \sigma'$.*

Proof. Consequence of the definitions of Amend, EPP and the semantics of CC. □

Example 4. *Applying Amend to the choreography C in Example 1 yields the choreography C' in the same example.* □

Example 5. *Thanks to merging, amendment can also recognise some situations where additional selections are not needed. For example, in the choreography $C = \mathsf{if}\ \mathsf{p} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ (\mathsf{p}.(\mathsf{s} \cdot \mathsf{c})\ \texttt{->}\ \mathsf{r}; \mathbf{0})\ \mathsf{else}\ (\mathsf{p}.(\mathsf{c})\ \texttt{->}\ \mathsf{r}; \mathbf{0})$, r does not need to know the choice made by p, as it always performs the same input action. Here, C is projectable and $\mathsf{Amend}(C) = C$.* □

Note that amending a choreography from MC returns a choreography in CC. In § 5 we show that we can prove an amendment lemma for MC, but this will require much more work.

4. Turing Completeness of MC and Its Consequences

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 (Kleene, 1952), 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 MC is Turing complete, since any choreography in MC can be amended to one in CC 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). By projecting our choreographies via EPP, we obtain corresponding function implementations in the process calculus SP.

Establishing that CC is Turing complete is long, but not difficult. Our proof is in line with other traditional proofs of computational completeness (Cutland, 1980; Kleene, 1952; Turing, 1937), where data and programs are distinct. This differs from other proofs of similar results for, e.g., π-calculus (Sangiorgi and Walker, 2001) and λ-calculus (Barendregt, 1984), 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).

4.1. Partial Recursive Functions

Our definition of the class of partial recursive functions $\mathcal{R}$ is slightly simplified, but equivalent to, that in (Kleene, 1952), 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. $P^n_m(x_1, \dots, x_n) = x_m$ for all $x_1, \dots, x_n \in \mathbb{N}$.

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 $g_1, \dots, g_k$ as: $h(\tilde{x}) = f(g_1(\tilde{x}), \dots, g_k(\tilde{x}))$.

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 $z < y$.

Example 6 (Addition and Subtraction). *We show that* $\mathsf{add} \in \mathcal{R}$*, where* $\mathsf{add} : \mathbb{N}^2 \to \mathbb{N}$ *adds its two arguments. Since* $\mathsf{add}(0, y) = y$ *and* $\mathsf{add}(x + 1, y) = \mathsf{add}(x, y) + 1$*, we can define* add *by recursion as* $\mathsf{add} = R\big(P^1_1, C(S, P^3_2)\big)$*. Indeed, the function* $y \mapsto \mathsf{add}(0, y)$ *is simply* P^1_1*, whereas* $1 + \mathsf{add}(x, y)$ *is* $h(x, \mathsf{add}(x, y), y)$ *where* $h(x, y, z) = y + 1$*, which is the composition of the successor function with* P^3_2*.*

From addition, we can define subtraction by minimization, since $\mathsf{sub}(x, y) = x - y$ *is the smallest* z *such that* $y + z = x$ *(subtraction is not defined if* $y > x$*). We use an auxiliary function* $\mathsf{eq}(x, y)$ *that returns* 0 *if* $x = y$ *and a non-zero* *value otherwise, which is known to be partial recursive. Then we can define subtraction as* $\mathsf{sub} = M\Big(C\big(\mathsf{eq}, C(\mathsf{add}, P^3_2, P^3_3), P^3_1\big)\Big)$*. Indeed, composing* add *with* P^3_2 *and* P^3_3 *produces* $(x, y, z) \mapsto y + z$*, and the outer composition yields* $(x, y, z) \mapsto \mathsf{eq}(y + z, x)$*. This function evaluates to* 0 *precisely when* $z = y - x$*, and applying minimization computes this value from* x *and* y*.*

4.2. Encoding Partial Recursive Functions in MC and CC

All functions in $\mathcal{R}$ can be implemented in CC, in the sense of Definition 1. Since selections can be inferred by amendment, we develop our encoding in MC and discuss projectability later.

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

$$\pi(S) = \pi(Z) = \pi\left(P^n_m\right) = 0 \qquad\qquad \pi(R(f, g)) = \pi(f) + \pi(g) + 3$$

$$\pi\left(C(f, g_1, \ldots, g_k)\right) = \pi(f) + \textstyle\sum_{i=1}^k \pi(g_i) + k \qquad \pi(M(f)) = \pi(f) + 3$$

For simplicity, we write $\tilde{\mathsf{p}}$ for $\mathsf{p}_1, \ldots, \mathsf{p}_n$ (when n is known) and $\{A_i\}^n_{i=1}$ for $A_1 \fatsemi \ldots \fatsemi A_n$.

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:

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

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 INC from Example 2 for brevity:

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

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.

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

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 8 (Encoding)**.** *Let $f \in \mathcal{R}$. The* encoding *of f in MC is* $[\![f]\!]^{\tilde{\mathsf{p}}\mapsto\mathsf{q}} = [\![f]\!]_0^{\tilde{\mathsf{p}}\mapsto\mathsf{q}}$.

Example 7. *We illustrate this construction by showing the encoding of the* add *and* sub *functions given in Example 6. Recall that* $\mathsf{add} = R(P^1_1, C(S, P^3_2))$. *Expanding* $[\![\mathsf{add}]\!]^{\mathsf{p}_x,\mathsf{p}_y\mapsto\mathsf{q}}$ *we obtain:*

$$
\begin{aligned}
& [\![\mathsf{add}]\!]_0^{\mathsf{p}_x,\mathsf{p}_y\mapsto\mathsf{q}} = \\
& \mathsf{def}\ T = \mathsf{if}\ (\mathsf{r}_1 \overset{\leftarrow}{=} \mathsf{p}_x)\ \mathsf{then}\ (\mathsf{r}_0.\mathsf{c} \;\texttt{->}\; \mathsf{q};\ \mathbf{0}) \\
& \qquad \mathsf{else}\ \underbrace{\mathsf{r}_0.\mathsf{c} \;\texttt{->}\; \mathsf{r}_3}_{[\![P^3_2]\!]_4^{\mathsf{r}_1,\mathsf{r}_0,\mathsf{p}_y\mapsto\mathsf{r}_3}};\ \underbrace{\mathsf{r}_3.(\mathsf{s}\cdot\mathsf{c}) \;\texttt{->}\; \mathsf{r}_2}_{[\![S]\!]_4^{\mathsf{r}_3\mapsto\mathsf{r}_2}};\ \mathsf{r}_2.\mathsf{c} \;\texttt{->}\; \mathsf{r}_0;\ \underbrace{\mathsf{r}_1.\mathsf{c} \;\texttt{->}\; \mathsf{r}_2;\ \mathsf{r}_2.(\mathsf{s}\cdot\mathsf{c}) \;\texttt{->}\; \mathsf{r}_1}_{\textsc{inc}(\mathsf{r}_1,\mathsf{r}_2)};\ T \\
& \mathsf{in}\ \underbrace{\mathsf{p}_y.\mathsf{c} \;\texttt{->}\; \mathsf{r}_0}_{[\![P^1_1]\!]_3^{\mathsf{p}_y\mapsto\mathsf{r}_0}};\ \mathsf{r}_2.\varepsilon \;\texttt{->}\; \mathsf{r}_1;\ T
\end{aligned}
$$

The first two actions in the else *branch correspond to* $[\![C(S, P^3_2)]\!]_3^{\mathsf{r}_1,\mathsf{r}_0,\mathsf{p}_y\mapsto\mathsf{r}_2}$.

For subtraction, we first show how to implement equality directly in MC, without resorting to its proof of membership in $\mathcal{R}$. This choreography is not the simplest possible because we want it to have only one exit point; its construction illustrates how any choreography can be transformed to have this property.

$$
\begin{aligned}
\textsc{eq}(\mathsf{p}_x, \mathsf{p}_y, \mathsf{q}, \mathsf{r}) \triangleq \mathsf{def}\ T = {} & (\mathsf{r}.\mathsf{c} \;\texttt{->}\; \mathsf{q};\ \mathbf{0})\ \mathsf{in} \\
& \mathsf{if}\ (\mathsf{p}_x \overset{\leftarrow}{=} \mathsf{p}_y)\ \mathsf{then}\ (\mathsf{p}_x.\varepsilon \;\texttt{->}\; \mathsf{r};\ T)\ \mathsf{else}\ (\mathsf{p}_x.(\mathsf{s}\cdot\mathsf{c}) \;\texttt{->}\; \mathsf{r};\ T)
\end{aligned}
$$

Recall now that $\mathsf{sub} = M(C(\mathsf{eq}, C(\mathsf{add}, P^3_2, P^3_3), P^3_1))$. *Unfolding the encoding of* *minimization and composition, we obtain that* $\llbracket \mathsf{sub} \rrbracket_0^{\mathsf{p}_x,\mathsf{p}_y \mapsto \mathsf{q}}$ *is*

$$
\begin{aligned}
\mathsf{def}\ T = {} & \llbracket P_2^3 \rrbracket_7^{\mathsf{p}_x,\mathsf{p}_y,\mathsf{r}_1 \mapsto \mathsf{r}_5} \fatsemi \llbracket P_2^3 \rrbracket_7^{\mathsf{p}_x,\mathsf{p}_y,\mathsf{r}_1 \mapsto \mathsf{r}_6} \fatsemi \llbracket \mathsf{add} \rrbracket_7^{\mathsf{r}_5,\mathsf{r}_6 \mapsto \mathsf{r}_3} \fatsemi \\
& \llbracket P_1^3 \rrbracket_{11}^{\mathsf{p}_x,\mathsf{p}_y,\mathsf{r}_1 \mapsto \mathsf{r}_4} \fatsemi \textsc{eq}(\mathsf{r}_3, \mathsf{r}_4, \mathsf{r}_0, \mathsf{r}_{11}) \fatsemi \mathsf{r}_1.\varepsilon \texttt{ -> } \mathsf{r}_2; \\
& \mathsf{if}\ \mathsf{r}_2 \stackrel{\leftarrow}{=} \mathsf{r}_0\ \mathsf{then}\ (\mathsf{r}_1.\mathsf{c} \texttt{ -> } \mathsf{q}; \mathbf{0})\ \mathsf{else}\ (\textsc{inc}(\mathsf{r}_1, \mathsf{r}_2) \fatsemi T) \\
\mathsf{in}\ & \mathsf{r}_2.\varepsilon \texttt{ -> } \mathsf{r}_1;\ T
\end{aligned}
$$

The first line in the definition of T *is* $\llbracket C(\mathsf{add}, P_2^3, P_3^3) \rrbracket_5^{\mathsf{p}_x,\mathsf{p}_y,\mathsf{r}_1 \mapsto \mathsf{r}_3}$*; the first five processes composed therewithin are*

$$\llbracket C(\mathsf{eq}, C(\mathsf{add}, P_2^3, P_3^3), P_1^3) \rrbracket_3^{\mathsf{p}_x,\mathsf{p}_y,\mathsf{r}_1 \mapsto \mathsf{r}_0}\ .$$

Fully unfolding the base cases, we obtain

$$
\begin{aligned}
\llbracket \mathsf{sub} \rrbracket_0^{\mathsf{p}_x,\mathsf{p}_y \mapsto \mathsf{q}} = {} & \mathsf{def}\ T = \mathsf{p}_y.\mathsf{c} \texttt{ -> } \mathsf{r}_5;\ \mathsf{r}_1.\mathsf{c} \texttt{ -> } \mathsf{r}_6; \\
& \quad \mathsf{def}\ R = \mathsf{if}\ (\mathsf{r}_8 \stackrel{\leftarrow}{=} \mathsf{r}_5) \\
& \quad\quad \mathsf{then}\ \mathsf{r}_7.\mathsf{c} \texttt{ -> } \mathsf{r}_3;\ \mathsf{p}_x.\mathsf{c} \texttt{ -> } \mathsf{r}_4; \\
& \quad\quad\quad \mathsf{def}\ S = \mathsf{r}_{11}.\mathsf{c} \texttt{ -> } \mathsf{r}_0;\ \mathsf{r}_1.\varepsilon \texttt{ -> } \mathsf{r}_2; \\
& \quad\quad\quad\quad \mathsf{if}\ (\mathsf{r}_2 \stackrel{\leftarrow}{=} \mathsf{r}_0)\ \mathsf{then}\ (\mathsf{r}_1.\mathsf{c} \texttt{ -> } \mathsf{q};\ \mathbf{0})\ \mathsf{else}\ (\mathsf{r}_1.\mathsf{c} \texttt{ -> } \mathsf{r}_2;\ \mathsf{r}_2.(\mathsf{s}\cdot\mathsf{c}) \texttt{ -> } \mathsf{r}_1;\ T) \\
& \quad\quad\quad \mathsf{in}\ \mathsf{if}\ (\mathsf{r}_3 \stackrel{\leftarrow}{=} \mathsf{r}_4)\ \mathsf{then}\ (\mathsf{r}_3.\varepsilon \texttt{ -> } \mathsf{r}_{11};\ S)\ \mathsf{else}\ (\mathsf{r}_3.(\mathsf{s}\cdot\mathsf{c}) \texttt{ -> } \mathsf{r}_{11};\ S) \\
& \quad\quad \mathsf{else}\ \mathsf{r}_7.\mathsf{c} \texttt{ -> } \mathsf{r}_{10};\ \mathsf{r}_{10}.(\mathsf{s}\cdot\mathsf{c}) \texttt{ -> } \mathsf{r}_9; \\
& \quad\quad\quad \mathsf{r}_9.\mathsf{c} \texttt{ -> } \mathsf{r}_7;\ \mathsf{r}_8.\mathsf{c} \texttt{ -> } \mathsf{r}_9;\ \mathsf{r}_9.(\mathsf{s}\cdot\mathsf{c}) \texttt{ -> } \mathsf{r}_8;\ R \\
& \quad \mathsf{in}\ \mathsf{r}_6.\mathsf{c} \texttt{ -> } \mathsf{r}_7;\ \mathsf{r}_9.\varepsilon \texttt{ -> } \mathsf{r}_8;\ R \\
& \mathsf{in}\ \mathsf{r}_2.\varepsilon \texttt{ -> } \mathsf{r}_1;\ T
\end{aligned}
$$

Due to the way sequential composition works, the structure of the definition of sub *is no longer clear in this fully unfolded encoding.*

4.3. Soundness and Main Results

We prove that our construction is sound by induction.

By induction we show that the construction presented above is sound. In the 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.

Theorem 5 (Turing completeness of MC)**.** *If* $f : \mathbb{N}^n \to \mathbb{N}$ *and* $f \in \mathcal{R}$*, then, for every* k*,* $\llbracket f \rrbracket_k^{\tilde{\mathsf{p}} \mapsto \mathsf{q}}$ *implements* f *with input processes* $\tilde{\mathsf{p}} = \mathsf{p}_1, \ldots, \mathsf{p}_n$ *and output process* q*.*

Proof. 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 $\llbracket f \rrbracket_k^{\tilde{\mathsf{p}} \mapsto \mathsf{q}}, \sigma \to^* \sigma'$ 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 choreography. We exemplify this with successor.

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

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 $[\![f]\!]^{\mathsf{p}_1,\ldots,\mathsf{p}_n\mapsto\mathsf{q}}_{\ell_f}$ and $[\![g]\!]^{\mathsf{p}_1,\ldots,\mathsf{p}_{n+2}\mapsto\mathsf{q}}_{\ell_g}$ 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:

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

We now prove that

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

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:

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

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

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

which establishes the thesis, ignoring the value in r_t.

By induction on x_0 we obtain that

$$\llbracket h \rrbracket_\ell^{\mathsf{p}_0,\tilde{\mathsf{p}}\mapsto\mathsf{q}}, \{\ \mathsf{p}_i \mapsto \ulcorner x_i \urcorner\ \} \to^* \mathsf{def}\ T = (\dots)\ \mathsf{in}\ T, \left\{ \begin{array}{l} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{q}' \mapsto \ulcorner h(x_0,\tilde{x}) \urcorner \\ \mathsf{r}_c \mapsto \ulcorner x_0 \urcorner \end{array} \right\}$$

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

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

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 $\llbracket f \rrbracket_{\ell_f}^{\tilde{\mathsf{p}}\mapsto\mathsf{q}'}$ diverges from any state where each p_i contains $\ulcorner x_i \urcorner$, whence so does $\llbracket h \rrbracket_\ell^{p_0,\tilde{\mathsf{p}}\mapsto\mathsf{q}}$ by Lemma 1 and rule $\lfloor$C|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 $\llbracket g \rrbracket_{\ell_g}^{\mathsf{r}_c,\mathsf{q}',\tilde{\mathsf{p}}}$ 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

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

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

$$\llbracket h \rrbracket_\ell^{\tilde{\mathsf{p}}\to\mathsf{q}}, \{\ \mathsf{p}_i \mapsto \ulcorner x_i \urcorner\ \} \to^* \mathsf{def}\ T = (\dots)\ \mathsf{in}\ T, \left\{ \begin{array}{l} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{r}_c \mapsto \ulcorner 0 \urcorner \end{array} \right\}$$

$$\to^* \mathsf{def}\ T = (\dots)\ \mathsf{in}\ T, \left\{ \begin{array}{l} \mathsf{p}_i \mapsto \ulcorner x_i \urcorner \\ \mathsf{r}_c \mapsto \ulcorner n \urcorner \end{array} \right\}$$

$$\to^* \dots$$

□

Since MC is a fragment of CC, this result trivially implies Turing completeness of CC. Let $\mathrm{SP}^{\mathrm{CC}} = \{\llbracket C,\sigma \rrbracket \mid \llbracket C,\sigma \rrbracket \text{ is defined}\}$ 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 $\llbracket f \rrbracket$ to an equivalent projectable choreography. Then $\mathrm{SP}^{\mathrm{CC}}$ is Turing complete by Theorems 3 and 5.

Corollary 1 (Turing completeness of $\mathrm{SP}^{\mathrm{CC}}$)**.** *Every partial recursive function is implementable in* $\mathrm{SP}^{\mathrm{CC}}$.

Proof. 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. $\square$

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 $h = C(f, g_1, \ldots, g_k)$, $\{\!\{h\}\!\}_\ell^{\tilde{\mathsf{p}}\mapsto\mathsf{q}}$ is

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

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* $\{\!\{g_i\}\!\}_{\ell_i}^{\tilde{\mathsf{p}^i}\mapsto\mathsf{r}'_i}$ *run in parallel in* $\{\!\{h\}\!\}^{\tilde{\mathsf{p}}\mapsto\mathsf{q}}$.

Proof. Direct consequence of Theorem 2 and the definition of $\{\!\{\}\!\}$. $\square$

This parallelism is preserved by EPP, through Theorem 3.

5. Removing Selections

As we saw earlier, MC is the fragment of CC that does not contain selections, and MC choreographies can be amended into projectable CC choreographies. We now show that selections are not necessary to ensure projectability: we can encode those selections introduced by amendment using only conditionals and extra communications.

5.1. Selections as Value Communications

Selections are used pervasively for projectability in previous choreography languages, so the fact that they are technically unnecessary is both interesting and somewhat unexpected. This construction increases the size of the choreography exponentially, but both the number of processes and the size of the endpoint projections grow only by a linear factor.

We motivate our construction with an example.

Example 8. *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.*

$$\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\left(\mathsf{p}.\varepsilon \texttt{ -> } \mathsf{p}^*;\ \mathsf{r.c} \texttt{ -> } \mathsf{r}^*;\ \mathsf{r}^*.\varepsilon \texttt{ -> } \mathsf{r};\ \mathsf{if}\ \mathsf{r} \overset{\leftarrow}{=} \mathsf{p}^*\ \mathsf{then}\ (\mathsf{r}^*.\mathsf{c} \texttt{ -> } \mathsf{r};\ C_1)\ \mathsf{else}\ (\mathsf{r}^*.\mathsf{c} \texttt{ -> } \mathsf{r};\ C_2)\right)$$
$$\mathsf{else}\left(\mathsf{p.s}\cdot\mathsf{c} \texttt{ -> } \mathsf{p}^*;\ \mathsf{r.c} \texttt{ -> } \mathsf{r}^*;\ \mathsf{r}^*.\varepsilon \texttt{ -> } \mathsf{r};\ \mathsf{if}\ \mathsf{r} \overset{\leftarrow}{=} \mathsf{p}^*\ \mathsf{then}\ (\mathsf{r}^*.\mathsf{c} \texttt{ -> } \mathsf{r};\ C_1)\ \mathsf{else}\ (\mathsf{r}^*.\mathsf{c} \texttt{ -> } \mathsf{r};\ C_2)\right)$$

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.* □

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. These properties are essential to our construction. The fragment of CC obtained by amending choreographies in MC can be inductively generated by

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

where $S(\mathsf{p}, \tilde{\mathsf{r}}, \ell, C)$ prepends selections of label ℓ from p to all processes in the list $\tilde{\mathsf{r}}$. Formally, S is defined as

$$S(\mathsf{p}, \emptyset, \ell, C) = C \qquad S(\mathsf{p}, \mathsf{r} :: \tilde{\mathsf{r}}, \ell, C) = \mathsf{p} \texttt{ -> } \mathsf{r}[\ell];\ S(\mathsf{p}, \tilde{\mathsf{r}}, \ell, C)$$

Definition 9 (Selection elimination)**.** *Let* C *be a choreography obtained by amending a choreography in MC. The encoding* $(\!|C|\!)^+$ *of* C *in MC uses processes* $\mathsf{p}, \mathsf{p}^\bullet$ *for each* $\mathsf{p} \in \mathsf{pn}(C)$*, plus a special process* z*, and is defined in Figure 10.*

This definition significantly exploits the structure of amended choreographies, where selections are always paired at the top of the two branches of conditionals. It follows from it that $|\mathsf{pn}((\!|C|\!)^+)| = 2|\mathsf{pn}(C)|+1$ and that $|(\!|C|\!)^+| \leq 2^{|C|}$. However, the EPP from MC to MP collapses all branches of conditionals, hence $|[\![(\!|C|\!)^+]\!]_{\mathsf{q}^\bullet}| \leq |[\![(\!|C|\!)^+]\!]_{\mathsf{q}}| \leq 3|[\![C]\!]_{\mathsf{q}}$ for every $\mathsf{q} \in \mathsf{pn}(C)$.

Theorem 7 (Selection elimination)**.** *For every choreography* $C \in MC^-$*,* $[\![(\!|\mathsf{Amend}(C)|\!)]\!]$ *is defined.*

For convenience, we split the proof of this result in several lemmas.

Lemma 3. *If* $\mathsf{q} \in \tilde{\mathsf{r}}$*, then* $[\![(\!|S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{R}, C_2)|\!)_1]\!]_{\mathsf{q}} = [\![(\!|S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{L}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \mathrm{R}, C_2)|\!)_2]\!]_{\mathsf{q}}$.

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

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

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

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

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

Figure 10: Elimination of selections from amended choreographies.

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, $\llparenthesis S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{l}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{r}, C_2) \rrparenthesis$ unfolds to

$$\langle \mathsf{q.c} \texttt{ -> } \mathsf{q}^\bullet;\ \mathsf{p}.\varepsilon \texttt{ -> } \mathsf{q};\ \mathsf{if}\ \mathsf{q} \stackrel{\leftarrow}{=} \mathsf{z} \mathsf{\ then\ } \mathsf{q}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{q};\ \llparenthesis S(\mathsf{p}, \tilde{\mathsf{r}'}, \textsc{l}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}'}, \textsc{r}, C_2) \rrparenthesis_1$$
$$\mathsf{else\ } \mathsf{q}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{q};\ \llparenthesis S(\mathsf{p}, \tilde{\mathsf{r}'}, \textsc{l}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}'}, \textsc{r}, C_2) \rrparenthesis_2,$$
$$\mathsf{q.c} \texttt{ -> } \mathsf{q}^\bullet;\ \mathsf{p.sc} \texttt{ -> } \mathsf{q};\ \mathsf{if}\ \mathsf{q} \stackrel{\leftarrow}{=} \mathsf{z} \mathsf{\ then\ } \mathsf{q}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{q};\ \llparenthesis S(\mathsf{p}, \tilde{\mathsf{r}'}, \textsc{l}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}'}, \textsc{r}, C_2) \rrparenthesis_1$$
$$\mathsf{else\ } \mathsf{q}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{q};\ \llparenthesis S(\mathsf{p}, \tilde{\mathsf{r}'}, \textsc{l}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}'}, \textsc{r}, C_2) \rrparenthesis_2 \rangle$$

and the endpoint projections of both choreographies for q become

$$\mathsf{q}^\bullet!\langle \mathsf{c} \rangle;\ \mathsf{p}?;\ \mathsf{if}\ \mathsf{c} \stackrel{\leftarrow}{=} \mathsf{z} \mathsf{\ then\ } \mathsf{q}^\bullet?;\ \llbracket \llparenthesis S(\mathsf{p}, \tilde{\mathsf{r}'}, \textsc{l}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}'}, \textsc{r}, C_2) \rrparenthesis_1 \rrbracket_{\mathsf{q}}$$
$$\mathsf{else\ } \mathsf{q}^\bullet?;\ \llbracket \llparenthesis S(\mathsf{p}, \tilde{\mathsf{r}'}, \textsc{l}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}'}, \textsc{r}, C_2) \rrparenthesis_2 \rrbracket_{\mathsf{q}}$$

which are defined and identical. □

Lemma 4. *If* $\llbracket \llparenthesis C_1 \rrparenthesis \rrbracket_{\mathsf{q}} = \llbracket \llparenthesis C_2 \rrparenthesis \rrbracket_{\mathsf{q}}$ *and* $\mathsf{p} \neq \mathsf{q} \notin \tilde{\mathsf{r}}$*, then* $\llbracket \llparenthesis S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{l}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{r}, C_2) \rrparenthesis_1 \rrbracket_{\mathsf{q}} = \llbracket \llparenthesis S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{l}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{r}, C_2) \rrparenthesis_2 \rrbracket_{\mathsf{q}}$.

Proof. By induction on the length of $\tilde{\mathsf{r}}$. If $\tilde{\mathsf{r}} = \emptyset$, then the result reduces to the hypothesis. Otherwise, $\llparenthesis S(\mathsf{p}, \mathsf{r} :: \tilde{\mathsf{r}}, \textsc{l}, C_1), S(\mathsf{p}, \mathsf{r} :: \tilde{\mathsf{r}}, \textsc{r}, C_2) \rrparenthesis$ unfolds to

$$\langle \mathsf{r.c} \texttt{ -> } \mathsf{r}^\bullet;\ \mathsf{p}.\varepsilon \texttt{ -> } \mathsf{r};\ \mathsf{if}\ \mathsf{r} \stackrel{\leftarrow}{=} \mathsf{z} \mathsf{\ then\ } \mathsf{r}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{r};\ \llparenthesis S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{l}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{r}, C_2) \rrparenthesis_1$$
$$\mathsf{else\ } \mathsf{r}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{r};\ \llparenthesis S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{l}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{r}, C_2) \rrparenthesis_2,$$
$$\mathsf{r.c} \texttt{ -> } \mathsf{r}^\bullet;\ \mathsf{p.sc} \texttt{ -> } \mathsf{r};\ \mathsf{if}\ \mathsf{r} \stackrel{\leftarrow}{=} \mathsf{z} \mathsf{\ then\ } \mathsf{r}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{r};\ \llparenthesis S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{l}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{r}, C_2) \rrparenthesis_1$$
$$\mathsf{else\ } \mathsf{r}^\bullet\mathsf{.c} \texttt{ -> } \mathsf{r};\ \llparenthesis S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{l}, C_1), S(\mathsf{p}, \tilde{\mathsf{r}}, \textsc{r}, C_2) \rrparenthesis_2 \rangle$$

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

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

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

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

- q is z: then both endpoint projections become

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

  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

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

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

Lemma 5. *For every choreography C in MC$^{\leftarrow}$ 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 $\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ S(\mathsf{p},\tilde{\mathsf{r}},\mathrm{L},C_1)\ \mathsf{else}\ S(\mathsf{p},\tilde{\mathsf{r}},\mathrm{R},C_2)$, 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 3 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 4 applies and establishes the thesis as in the previous case. □

Proof (Theorem 7). Straightforward consequence of Lemma 5. □

The operational semantics of C and $(\!|\mathsf{Amend}(C)|\!)$ are related by the following results, which are straightforward to prove by structural induction.

Lemma 6. *Choreographies C and $(\!|\mathsf{Amend}(C)|\!)^+$ are equivalent wrt* $\mathsf{pn}(C)$.

Lemma 7. *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'. Furthermore, the latter reduction consists of only one step except for the case when the former uses rule $\lfloor C|Cond\rceil$.*

Conversely, if $(\!|\mathsf{Amend}(C)|\!),\sigma^+\to C',\sigma'$, then $C,\sigma\to C'',\sigma''$ where $C',\sigma'\to^*(\!|\mathsf{Amend}(C'')|\!),\sigma''^+$. Furthermore, the latter reduction is non-empty only in the case when the former uses rule $\lfloor C|Cond\rceil$.

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 $\mathrm{SP}^{\mathrm{MC}} = \{[\![C,\sigma]\!] \mid [\![C,\sigma]\!] \text{ is defined}\}$ of projections of minimal choreographies is also Turing complete.

Corollary 3 (Turing completeness of $\mathrm{SP}^{\mathrm{MC}}$)**.** *Every partial recursive function is implementable in* $\mathrm{SP}^{\mathrm{MC}}$*.*

5.2. Discussion

The results in this section show that label selection is not a necessary primitive in a choreography calculus, and thus we could take MC (rather than CC) as our core choreography language. Furthermore, the construction in § 4.2 shows that selections are not needed for implementing computable functions in CC; they are used only for obtaining projectable choreographies, via amendment.

There is however a strong argument for including label selections in a core choreography calculus. The advantages of eliminating selections are 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 (Coppo et al., 2016; Carbone et al., 2012; Carbone and Montesi, 2013; Honda et al., 2008; Dalla Preda et al., 2015; Qiu et al., 2007), 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 (Cruz-Filipe and Montesi, 2016c). 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. Minimality in Choreographies

We now discuss our choice of primitives for CC, showing that it is indeed a minimal core language for choreographic programming. We first show that if we remove or simplify any primitive from MC we are no longer able to compute all partial recursive functions using projectable choreographies. Since label selection can be encoded in MC, we also discuss why it should be included in a core language. Then we discuss the implications of our results for other choreography languages.

6.1. Minimality of MC

We proceed by analysing each primitive of MC. Recall that Turing completeness of MC is a pre-requisite for the Turing completeness of choreography projections. In most cases, simplifying MC yields a decidable termination problem (thus breaking Turing completeness). We start with the easiest terms.

Lemma 8. *Let C be a choreography with no exit points. Then C does not terminate.*

Proof. Straightforward by structural induction on C. □

Lemma 9. *Let C be a choreography with no communications. If C implements a function $f : \mathbb{N}^n \to \mathbb{N}$, then, for all inputs $\vec{x} \in \mathbb{N}^n$, either $f(\vec{x}) = x_i$ for some i or $f(\vec{x})$ is undefined.*

Proof. By the semantics of MC, only communication actions can change the state σ, hence structural induction on C shows that $C\sigma \not\to C'\sigma'$ with $\sigma' \neq \sigma$. The thesis is a consequence of the definition of function implementation. □

Observe further that the syntax of expressions is trivially minimal: ε (zero) is the only terminal, removing `c` makes termination decidable (since values become statically defined), and likewise for `s` (since no new values can be computed).

Lemma 10. *Let C be a choreography with no recursive definitions. Then C always terminates.*

Proof. Without recursive definitions, rule $\lfloor$C|Unfold$\rceil$ is never applicable, hence execution C always reduces the size of the choreography. □

Again we observe that recursive definitions are already severely restricted: MC supports only tail recursion and definitions are not parameterised.

Removing conditionals naturally also breaks Turing completeness.

Theorem 8. *Let C be a choreography with no conditionals. Then termination of C is decidable and independent of the initial state.*

Proof. 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. □

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.

Theorem 9. *If the conditional is replaced by* $\mathsf{if}\ \mathsf{p.c} = v\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$*, where* v *is a value, and rule* $\lfloor C|Cond\rceil$ *by*

$$\frac{i = 1 \textit{ if } \sigma(\mathsf{p}) = v,\ i = 2 \textit{ otherwise}}{\mathsf{if}\ \mathsf{p.c} = v\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \sigma \ \to\ C_i, \sigma}\,,$$

then termination is decidable.

Proof. 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. $\square$

Summarising, simplifying MC in any of the ways described above makes it no longer a representative model of choreographic programming.

6.2. CC as a core language: Channel Choreographies

CC is representative of the body of previous work on choreographic programming, where choreographies are used for implementations, for example (Carbone et al., 2012; Carbone and Montesi, 2013; Chor, 2016; Montesi and Yoshida, 2013; Dalla Preda et al., 2015; W3C WS-CDL Working Group, 2004). All the primitives of CC (and therefore of MC) 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.

In this section we make this claim precise for the model in (Carbone and Montesi, 2013), which we refer to in this work as Channel Choreographies (ChC). ChC is designed to be projected to a variant of the session-typed π-calculus (Coppo et al., 2016), which we refer to as Channel Processes (ChP). Communications in ChP are based on channels, instead of process names as in SP. This layer of indirection means that a process performing an I/O action does not know with which other process it is going to communicate, and that there can be race conditions on the usage of channels. ChC comes with a typing discipline for checking that the usage of channels specified in a choreography does not cause errors in the process code generated by EPP.

6.2.1. Channel Choreographies

Syntax. We report the full syntax of ChC in Figure 11. Several terms are unnecessary for our translation; we $\boxed{\text{box}}$ such terms in our presentation of the syntax. In the original presentation of ChC, expressions e may contain any basic values (integers, strings, etc.) or computable functions, making the language trivially Turing complete. Also, labels l range over an infinite set. Here, for

$$C ::= \eta; C \quad | \quad \mathsf{if}\ \mathsf{p}.(e = e')\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \quad | \quad \mathbf{0}$$
$$\quad | \quad \mathsf{def}\ X(\tilde{D}) = C_2\ \mathsf{in}\ C_1 \quad | \quad X\langle \tilde{E} \rangle \quad | \quad \boxed{(\boldsymbol{\nu} r)\ C}$$

$$\eta ::= \boxed{\widetilde{\mathsf{p[A]}}\ \mathsf{start}\ \widetilde{\mathsf{q[B]}} : a(k)} \quad | \quad \mathsf{p[A]}.e\ \texttt{->}\ \mathsf{q[B]}.x : k$$
$$\quad | \quad \mathsf{p[A]}\ \texttt{->}\ \mathsf{q[B]} : k[l] \quad | \quad \boxed{\mathsf{p[A]}\ \texttt{->}\ \mathsf{q[B]} : k\langle k'[\mathsf{C}]\rangle}$$

$$D ::= \mathsf{p}(\tilde{x}, \tilde{k}) \qquad E ::= \mathsf{p}(\tilde{e}, \tilde{k})$$

Figure 11: Channel Choreographies, Syntax.

our development, we need only to consider expressions of the form ε or $\mathtt{s} \cdot x$, and labels L and R (as in CC). The major difference between CC and ChC is the usage of *public channels* a and *session channels* k. Public channels are used to create new processes and channels at runtime, whereas session channels are used for point-to-point communications between processes. We only need a single session channel in our development.

An interaction η in ChC can be either a start, a value communication, a selection, or a delegation. In a start term $\widetilde{\mathsf{p[A]}}\ \mathsf{start}\ \widetilde{\mathsf{q[B]}} : a(k)$, the processes $\tilde{\mathsf{p}}$ on the left synchronise at the public channel a in order to create a new private session k and spawn some new processes $\tilde{\mathsf{q}}$ (k and $\tilde{\mathsf{q}}$ are bound to the continuation). Each process is annotated with the role it plays in the created session. Roles are ranged over by $\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots$. They are used in the typing discipline of CC to check whether sessions are used according to protocol specifications, given as multiparty session types (Honda et al., 2016).

In a value communication $\mathsf{p[A]}.e\ \texttt{->}\ \mathsf{q[B]}.x : k$, process p sends its evaluation of expression e over session k to process q, which stores the result in its local variable x; the name x appearing under q is bound to the continuation. Differently from CC, where each process has only one memory cell accessed through the placeholder c, in ChC each process has an unbounded number of cells (variables). Selections in ChC, of the form $\mathsf{p[A]}\ \texttt{->}\ \mathsf{q[B]} : k[l]$, are very similar to those in CC: the only difference is that we also have to write which role each process plays and the session used for communicating. In a delegation term $\mathsf{p[A]}\ \texttt{->}\ \mathsf{q[B]} : k\langle k'[\mathsf{C}]\rangle$, process p delegates its role C in session k' to process q; delegation in ChC is a typed form of channel mobility, inspired by the π-calculus.

In a conditional, process p chooses a continuation based on whether the expressions e and e' evaluate to the same value according to its own local state. The restriction term $(\boldsymbol{\nu} r)\ C$ is standard and binds the scope of r (which can be either a process name p or a session channel name k) to C. Finally, in the definition of a recursive procedure, the parameters $\tilde{D}$ indicate which processes are used in the body of the procedure and which variables and sessions are used by each process. In the invocation of a procedure $X\langle \tilde{E} \rangle$, each process can pass

$$\frac{}{\mathsf{p}[\mathtt{A}].v \text{ -> } \mathsf{q}[\mathtt{B}].x : k; C \ \rightarrow\ C[v/x@\mathsf{q}]} \ \lfloor\text{Ch}|\text{Com}\rceil \qquad \frac{}{\mathsf{p}[\mathtt{A}] \text{ -> } \mathsf{q}[\mathtt{B}] : k[l]; C \ \rightarrow\ C} \ \lfloor\text{Ch}|\text{Sel}\rceil$$

$$\frac{}{\mathsf{p}[\mathtt{A}] \text{ -> } \mathsf{q}[\mathtt{B}] : k\langle k'[\mathtt{C}]\rangle; C \ \rightarrow\ C} \ \lfloor\text{Ch}|\text{Del}\rceil \qquad \frac{}{\widetilde{\mathsf{p}[\mathtt{A}]}\ \mathsf{start}\ \widetilde{\mathsf{q}[\mathtt{B}]} : a(k); C \ \rightarrow\ (\boldsymbol{\nu}\tilde{\mathsf{q}}, k)\ C} \ \lfloor\text{Ch}|\text{Start}\rceil$$

$$\frac{i = 1 \text{ if } v = w,\ i = 2 \text{ otherwise}}{\mathsf{if}\ \mathsf{p}.(v = w)\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \ \rightarrow\ C_i} \ \lfloor\text{Ch}|\text{Cond}\rceil$$

$$\frac{C_1 \preceq C_2 \quad C_2 \ \rightarrow\ C_2' \quad C_2' \preceq C_1'}{C_1 \ \rightarrow\ C_1'} \ \lfloor\text{Ch}|\text{Struct}\rceil$$

$$\frac{C_1 \ \rightarrow\ C_1'}{\mathsf{def}\ X(\tilde{D}) = C_2\ \mathsf{in}\ C_1 \ \rightarrow\ \mathsf{def}\ X(\tilde{D}) = C_2\ \mathsf{in}\ C_1'} \ \lfloor\text{Ch}|\text{Ctx}\rceil$$

Figure 12: Channel Choreographies, Semantics.

generic expressions as parameters to itself.

Semantics. ChC was originally presented with an asynchronous semantics (Carbone and Montesi, 2013). We first present our results using only the (simpler) synchronous variant of the semantics of ChC, and defer the discussion of the general asynchronous case to the end of this section. This semantics is given in terms of a reduction relation, presented in Figure 12. Rule $\lfloor\text{Ch}|\text{Com}\rceil$ is the key rule, where the value sent from a process p is received by a process q. Technically, this is modelled by replacing variable x with v in the continuation C, but only when it appears under the process name q (the *smart substitution* $C[v/x@\mathsf{q}]$). Rule $\lfloor\text{Ch}|\text{Cond}\rceil$ models an internal choice: p chooses a continuation depending on whether the two values v and w are the same. Rules $\lfloor\text{Ch}|\text{Del}\rceil$ and $\lfloor\text{Ch}|\text{Start}\rceil$ implement the informal semantics of delegation and start described earlier; we do not use them in our development. The other rules are similar to those of CC. The structural precongruence $\preceq$ is defined as expected, following the same intuition as that for CC. In particular, it supports swapping two terms whenever they involve disjoint process names.

As expected, ChC offers a deadlock-freedom-by-design property in the style of Theorem 1 (Carbone and Montesi, 2013).

6.2.2. Channel Processes (ChP)

We now present Channel Processes (ChP), the target language that choreographies in ChC can be projected to. We discuss only the terms used in our work (see (Carbone and Montesi, 2013) for a complete presentation).

Syntax. The relevant part of the syntax of processes (P, Q) is reported in Figure 13. Binding occurrences are denoted by the usage of round parentheses. Terms $\overline{a}[\tilde{\mathtt{A}}](k)$, $a[\mathtt{A}](k)$ and $!a[\mathtt{A}](k)$ are used to start a new session k by synchro-

$$P, Q ::= \quad k[\mathtt{A}]!\mathtt{B}\langle e\rangle; P \;\mid\; k[\mathtt{B}]?\mathtt{A}(x); P \;\mid\; k[\mathtt{A}]!\mathtt{B} \oplus l; P \;\mid\; k[\mathtt{B}]?\mathtt{A}\&\{l_i : P_i\}_{i\in I}$$
$$\mid \quad P \,|\, Q \;\mid\; \mathsf{if}\, e = e' \,\mathsf{then}\, P \,\mathsf{else}\, Q \;\mid\; \mathsf{def}\, X(\tilde{x}, \tilde{k}) = Q \,\mathsf{in}\, P \;\mid\; X\langle \tilde{e}, \tilde{k}\rangle \;\mid\; \mathbf{0}$$

Figure 13: Channel Processes, Syntax (selection).

$$\frac{}{k[\mathtt{A}]!\mathtt{B}\langle v\rangle; P \;\mid\; k[\mathtt{B}]?\mathtt{A}(x); Q \quad\to\quad P \;\mid\; Q[v/x]} \;\lfloor\text{CP}|\text{Com}\rceil$$

$$\frac{(j \in I)}{k[\mathtt{A}]!\mathtt{B} \oplus l_j; P \;\mid\; k[\mathtt{B}]?\mathtt{A}\&\{l_i : Q_i\}_{i\in I} \quad\to\quad P \;\mid\; Q_j} \;\lfloor\text{CP}|\text{Sel}\rceil$$

Figure 14: Channel Processes, Semantics (selection).

nising on the public channel a, and model respectively: the process requesting the creation of the session (responsible for playing the first role in $\tilde{\mathtt{A}}$); a process accepting to play role $\mathtt{A}$ in the session; and, finally, a replicated process that will spawn a fresh process for playing role $\mathtt{A}$. In the first line we have the terms for in-session communications. In term $k[\mathtt{A}]!\mathtt{B}\langle e\rangle; P$, as role $\mathtt{A}$ on session k, we send the value of expression e to $\mathtt{B}$ on the same session; then, we proceed as P. Dually, term $k[\mathtt{B}]?\mathtt{A}(x); P$ receives a message for role $\mathtt{B}$ from role $\mathtt{A}$ on session k and stores it in variable x. Terms $k[\mathtt{A}]!\mathtt{B} \oplus l$ and $k[\mathtt{B}]?\mathtt{A}\&\{l_i : P_i\}_{i\in I}$ model, respectively, branch selection and offering. Finally, terms $k[\mathtt{A}]!\mathtt{B}\langle k'[\mathtt{C}]\rangle$ and $k[\mathtt{B}]?\mathtt{A}(k'[\mathtt{C}])$ capture channel mobility. The other terms are the standard parallel composition, procedure definition, procedure call, conditional (restricted to checking for equality), and terminated process.

Semantics. As before, we discuss only the synchronous semantics of ChP. We discuss only the communication rules, shown in Figure 14, as all the other rules are standard – see (Kouzapas and Yoshida, 2013). As in typical calculi for multiparty sessions equipped with roles, each role in a session is a distinct communication endpoint. Therefore, a send action on a session k from a role $\mathtt{A}$ towards a role $\mathtt{B}$ synchronises with a receive action on the same session k by the target role $\mathtt{B}$ wishing to receive from the sender role $\mathtt{A}$.

6.2.3. Endpoint Projection and Typing

As for CC, the Endpoint Projection from ChC to ChP is defined by first defining how to project the behaviour of a single process. The projection of a process p from a choreography C, written $[\![C]\!]_{\mathsf{p}}$, is inductively defined on the structure of C in a similar way as the behaviour projection given in Section 3.3. The complete EPP procedure from ChC to ChP is technically involved, because the start term $\widetilde{\mathsf{p}[\mathtt{A}]}\,\mathsf{start}\,\widetilde{\mathsf{q}[\mathtt{B}]} : a(k)$ found in ChC enables the reuse of the same services exposed at a public channel a for spawning processes with potentially different behaviour. However, since start terms and restriction of names

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

$$\left[\!\!\left[\begin{array}{l} \mathsf{if}\ \mathsf{p}.(e = e') \\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \end{array} \right]\!\!\right]_{\mathsf{r}} = \begin{cases} \begin{array}{l} \mathsf{if}\ e = e' \\ \mathsf{then}\ [\![C_1]\!]_{\mathsf{r}}\ \mathsf{else}\ [\![C_2]\!]_{\mathsf{r}} \end{array} & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C_1]\!]_{\mathsf{r}} \sqcup [\![C_2]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

Figure 15: Channel Choreographies, EndPoint Projection (relevant cases).

are unnecessary for our development, we can use a much simpler definition – see (Carbone and Montesi, 2013) for the general case. We report the rules for projecting value communications and conditionals in Figure 15. The merging operator $P \sqcup Q$ works as in CC: it is isormorphic to P and Q aside from input branches with distinct labels, which are instead included in a larger input branching.

6.2.4. Typing ChC

Differently from CC, the EPP of a choreography in ChC does not always yield correct results. Consider the following choreography:

$$C \;=\; \mathsf{p}[\mathtt{A}] \text{ -> } \mathsf{q}[\mathtt{B}] : k[\textsc{l}];\ \mathsf{q}[\mathtt{B}].\varepsilon \text{ -> } \mathsf{p}[\mathtt{A}].x : k;\ \mathsf{r}[\mathtt{A}] \text{ -> } \mathsf{q}[\mathtt{B}] : k[\textsc{l}]$$

The choreography C above always terminates by reaching $\mathbf{0}$ (by using rules $\lfloor$Ch|Sel$\rceil$, then $\lfloor$Ch|Com$\rceil$, and then $\lfloor$Ch|Sel$\rceil$ again). However, its EPP (albeit defined) may get stuck:

$$[\![C]\!] = \overbrace{k[\mathtt{A}]!\mathtt{B} \oplus \textsc{l};\ k[\mathtt{A}]?\mathtt{B}(x)}^{[\![C]\!]_{\mathsf{p}}} \quad | \quad \overbrace{k[\mathtt{A}]!\mathtt{B} \oplus \textsc{l}}^{[\![C]\!]_{\mathsf{r}}}$$
$$|\ \ \underbrace{k[\mathtt{B}]?\mathtt{A}\&\{\textsc{l} :\ k[\mathtt{B}]!\mathtt{A}\langle\varepsilon\rangle;\ k[\mathtt{B}]?\mathtt{A}\&\{\textsc{l} : \mathbf{0}\}\}}_{[\![C]\!]_{\mathsf{q}}}$$

Above, we have a race between the projections of process p and process r for the selection of label $\textsc{l}$ offered by process q. This is because both p and r play the same role $\mathtt{A}$ in session k and therefore the receiver (the projection of process q) cannot distinguish them. In the case where the race is won by the projection of process r, not only do we obtain a reduction not defined by the originating choreography, but we even get into a deadlocked situation:

$$[\![C]\!] \;\to\; k[\mathtt{A}]!\mathtt{B} \oplus \textsc{l};\ k[\mathtt{A}]?\mathtt{B}(x)\ \mid\ k[\mathtt{B}]!\mathtt{A}\langle\varepsilon\rangle;\ k[\mathtt{B}]?\mathtt{A}\&\{\textsc{l} : \mathbf{0}\}$$

To avoid such situations, ChC comes with a typing discipline based on multiparty session types that guarantees the absence of races.

$$G ::= \mathtt{A} \mathbin{\texttt{->}} \mathtt{B}{:}\langle \mathbf{nat} \rangle; G \mid \mathtt{A} \mathbin{\texttt{->}} \mathtt{B} : \{l_i : G_i\}_{i \in I} \mid \mu \mathbf{t}; G \mid \mathbf{t} \mid \mathsf{end}$$

$$S ::= \mathbf{nat} \mid \mathbf{string} \mid \ \ldots \qquad\qquad l ::= \textsc{l} \mid \textsc{r}$$

Figure 16: Global Types, Syntax.

A typing judgement for CC has the form $\Gamma; \Theta \vdash C \triangleright \Delta$, where Δ types the usage of sessions, Θ the ownership of roles by processes, and Γ variables and public channels.

Formally, the typing environment Γ contains variable typings of the form $x@\mathsf{p} : S$, typing variable x at p with data type S (which can only be **nat** in our case). An environment Θ contains ownership typings of the form $\mathsf{p} : k[\mathtt{A}]$, read "process p owns role $\mathtt{A}$ in k" (when writing $\Theta, \mathsf{p} : k[\mathtt{A}]$, it is assumed that no other process owns the same role for the same session in Θ). The environment Δ contains session typings of the form $k : G$, where G is a global type (Honda et al., 2016). The syntax of global types is given in Figure 16. A global type G abstracts a communication between two roles in a session. A value communication is abstracted by $\mathtt{A} \mathbin{\texttt{->}} \mathtt{B} : \langle \mathbf{nat} \rangle$ (we restrict values to be natural numbers). A global type $\mathtt{A} \mathbin{\texttt{->}} \mathtt{B} : \{l_i : G_i\}_{i \in I}$ allows any selection from $\mathtt{A}$ to $\mathtt{B}$ of one of the labels l_i, provided that then the session proceeds as specified by the corresponding continuation G_i. The other terms are for recursion ($\mu\mathbf{t}$ and $\mathbf{t}$) and termination (end).

We discuss the most relevant typing rules for ChC, given in Figure 17. Rule $\lfloor$T|Com$\rceil$ checks that, in a value communication on session k, the sender and receiver processes own their respective roles in session k ($\Theta \vdash \mathsf{p} : k[\mathtt{A}], \mathsf{q} : k[\mathtt{B}]$), that the protocol for session k expects a communication for their respective roles ($k : \mathtt{A} \mathbin{\texttt{->}} \mathtt{B} : \langle S \rangle; G$), and that the expression sent by the sender has the expected type S. Rule $\lfloor$T|Sel$\rceil$ checks that a selection uses one of the labels expected by the protocol for the session ($j \in I$). Rule $\lfloor$T|Cond$\rceil$ is standard, requiring both branches to have the same typing; observe that different communication behaviour in the two branches may still occur, because of rule $\lfloor$T|Sel$\rceil$. Rules $\lfloor$T|Call$\rceil$ and $\lfloor$T|Def$\rceil$ type, respectively, recursive calls and recursive procedures. These rules are simplified compared to the presentation in (Carbone and Montesi, 2013), taking into account that our encoding always calls procedures with exactly the same arguments (processes and variables) as they are declared.

Well-typedness is preserved by reductions. Furthermore, using this type system we get an operational correspondence result for EPP from ChC to ChP.

Theorem 10 (Operational Correspondence (ChC $\leftrightarrow$ ChP) (Carbone and Montesi, 2013))**.** *Let C be a well-typed channel choreography without start subterms (terms of the form $\widetilde{\mathsf{p}[\mathtt{A}]}\, \mathsf{start}\, \widetilde{\mathsf{q}[\mathtt{B}]} : a(k)$) and such that its endpoint projection $[\![C]\!]$ is defined. Then:*

- *(Completeness) $C \to C'$ implies $[\![C]\!] \to\succ [\![C']\!]$;*

$$\frac{\Gamma \vdash e@\mathsf{p} : S \quad \Theta \vdash \mathsf{p} : k[\mathtt{A}], \mathsf{q} : k[\mathtt{B}] \quad \Gamma, x@\mathsf{q} : S; \Theta \vdash C \triangleright \Delta, k : G}{\Gamma; \Theta \vdash \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathsf{q}[\mathtt{B}].x : k; C \triangleright \Delta, k : \mathtt{A} \texttt{ -> } \mathtt{B} : \langle S \rangle; G} \lfloor \text{T|Com} \rceil$$

$$\frac{\Theta \vdash \mathsf{p} : k[\mathtt{A}], \mathsf{q} : k[\mathtt{B}] \quad j \in I \quad \Gamma; \Theta \vdash C \triangleright \Delta, k : G_j}{\Gamma; \Theta \vdash \mathsf{p}[\mathtt{A}] \texttt{ -> } \mathsf{q}[\mathtt{B}] : k[l_j]; C \triangleright \Delta, k : \mathtt{A} \texttt{ -> } \mathtt{B} : \{l_i : G_i\}_{i \in I}} \lfloor \text{T|Sel} \rceil$$

$$\frac{\Gamma; \Theta \vdash C_1 \triangleright \Delta \quad \Gamma; \Theta \vdash C_2 \triangleright \Delta}{\Gamma; \Theta \vdash \mathsf{if}\ \mathsf{p}.(e = e')\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \triangleright \Delta} \lfloor \text{T|Cond} \rceil$$

$$\frac{\Delta\ \mathsf{end}\ \text{only} \quad \Gamma' \subseteq \Gamma}{\Gamma, X(\tilde{D}) : (\Gamma'; \Theta; \Delta'); \Theta \vdash X\langle \tilde{D} \rangle \triangleright \Delta, \Delta'} \lfloor \text{T|Call} \rceil$$

$$\frac{\begin{array}{c}\Gamma, X(\tilde{D}) : (\Gamma'; \Theta'; \Delta'); \Theta \vdash C_1 \triangleright \Delta \quad \Gamma' \subseteq \Gamma \\ \Gamma', X(\tilde{D}) : (\Gamma'; \Theta'; \Delta'); \Theta' \vdash C_2 \triangleright \Delta' \quad \Theta' \subseteq \Theta\end{array}}{\Gamma; \Theta \vdash \mathsf{def}\ X(\tilde{D}) = C_2\ \mathsf{in}\ C_1 \triangleright \Delta} \lfloor \text{T|Def} \rceil$$

Figure 17: Channel Choreographies, Typing Rules (selection).

- *(Soundness)* $[\![C]\!] \to P$ *implies* $C \to C'$ *and* $[\![C']\!] \prec P$.

where $\prec$ *is the pruning relation defined in (Carbone and Montesi, 2013).*

As for CC, the EPP of a well-typed channel choreography never deadlocks.

6.2.5. Embedding CC into ChC

Defining an embedding from CC to ChC is nontrivial, as the communication primitives of CC and ChC are different. In CC, messages are passed directly between processes: each process knows whom it is sending to or receiving from in each communication step; in ChC, communication is between roles in a session channel. To translate core choreographies into channel choreographies, we therefore assign to each process a role syntactically identical to its name, and perform all communication over a fixed channel k.

Conditional terms are also not directly translatable, as ChC evaluates guards in a single process. For this reason, each translated process uses two variables: x, storing its internal value, and y, used exclusively for temporary storage of a value required for a test.

For recursion, we recall that $\mathsf{pn}(C)$ returns the set of process names in C.

Definition 10 (Embedding of CC in ChC)**.** *The* embedding *of a core choreog-*

raphy C in ChC is $\{C\}$, inductively defined as follows.

$$\{\mathsf{p}.e \texttt{ -> } \mathsf{q}; C\} = \mathsf{p}[\mathsf{p}].e[x/\mathsf{c}] \texttt{ -> } \mathsf{q}[\mathsf{q}].x : k; \{C\}$$
$$\{\mathsf{p} \texttt{ -> } \mathsf{q}[l]; C\} = \mathsf{p}[\mathsf{p}] \texttt{ -> } \mathsf{q}[\mathsf{q}] : k[l]; \{C\}$$
$$\left\{\mathsf{if}\ \mathsf{p} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\right\} = \mathsf{q}[\mathsf{q}].x \texttt{ -> } \mathsf{p}[\mathsf{p}].y : k; \quad \mathsf{if}\ \mathsf{p}.(x = y)\ \mathsf{then}\ \{C_1\}\ \mathsf{else}\ \{C_2\}$$
$$\{\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1\} = \left(\mathsf{def}\ X(*) = \{C_2\}\right)[|C_2|/*]\ \mathsf{in}\ \{C_1\}$$
$$\{X\} = X\langle * \rangle \qquad \{\mathbf{0}\} = \mathbf{0}$$

where $|A| = \{\mathsf{p}(\{x,y\},k) \mid \mathsf{p} \in \mathsf{pn}(A)\}$.

Lemma 11. *Let C and C' be core choreographies. Then $C \preceq C'$ if and only if $\{C\} \preceq \{C'\}$.*

Proof. For the direct implication, observe that all structural precongruence rules in CC become valid instances of precongruence in ChC when mapped by $\{\cdot\}$. Conversely, given a structural precongruence rule in ChC, if its arguments are in the image of $\{\cdot\}$, then the rule can be pulled back to a valid precongruence in CC. □

In order to compare the semantics of core and channel choreographies, we need to take the state into account. This is done by viewing each state as a substitution, replacing all free occurrences of x with the actual content of the process it belongs to.

Definition 11 (Substitution induced by state)**.** *Let C be a core choreography and σ be a state. The substitution σ_C is defined as $\sigma_C = [\sigma(\mathsf{p})/x@\mathsf{p} \mid \mathsf{p} \in \mathsf{pn}(C)]$, and the embedding of C in ChC via σ is the channel choreography $\{C\}_\sigma = \sigma_C(\{C\})$.*

Below, $\to^+$ denotes a chain of one or more applications of $\to$, and $\to^?$ denotes identity or one application of $\to$.

Theorem 11 (Operational Correspondence (CC $\leftrightarrow$ ChC))**.** *Let C be a choreography in CC. Then, for all σ:*

- (Completeness) *$C, \sigma \to C', \sigma'$ implies $\{C\}_\sigma \to^+ \{C'\}_{\sigma'}$;*
- (Soundness) *$\{C\}_\sigma \to C'$ implies $C, \sigma \to C^*, \sigma^*$ and $C' \to^? \{C^*\}_{\sigma^*}$.*

Proof.
- *(Completeness)* We analyze the possible cases for the rule justifying $C \xrightarrow{\eta} C'$. The only non-trivial case is rule $\lfloor$C|Com$\rceil$. Suppose C is $\mathsf{p}.e \texttt{ -> } \mathsf{q}; C'$. If $C, \sigma \to C', \sigma'$, then $\{C\}_\sigma = \mathsf{p}.\sigma_C(e[x/\mathsf{c}]) \texttt{ -> } \mathsf{q}.x : k; \{C'\}_{\sigma''}$ where $\sigma'' = \sigma_C \setminus \{\sigma(\mathsf{q})/x@\mathsf{q}\}$ can make a transition to $\{C'\}_{\sigma''}[e[\sigma_C(\mathsf{p})/\mathsf{c}]/x@\mathsf{q}]$, which coincides with $\{C'\}_{\sigma'}$.

- *(Soundness)* The proof is again by case analysis on the transition from $\{\!\!\{C\}\!\!\}_\sigma$ to C', noting that this cannot involve delegation or start actions. Most cases are straightforward, except when the transition is a communication obtained from translating a conditional. In this case, C' must execute the full conditional action, and $\{\!\!\{C^*\}\!\!\}_{\sigma^*}$ reduces to C' by application of rule $\lfloor \text{Ch}|\text{Cond} \rceil$.

$\square$

We now define a notion of function implementation in ChC. Since the semantics of ChC does not have state, this definition is slightly different than that for CC. The embedding of CC is then a Turing complete fragment of ChC, which we show to be projectable.

Definition 12 (Implementation in ChC)**.** *A channel choreography C implements a function $f : \mathbb{N}^n \to \mathbb{N}$ with input variables $\mathsf{p}_1.z_1, \ldots, \mathsf{p}_n.z_n$ and output variable $\mathsf{q}.z$ if, for all $x_1, \ldots, x_n \in \mathbb{N}$:*

- *if $f(\tilde{x})$ is defined, then $C[\widetilde{\ulcorner x_i \urcorner / z_i @ \mathsf{p}_i}] \to^* \mathbf{0}$, and q receives exactly one message with $\ulcorner f(\tilde{x}) \urcorner$ as the value transmitted;*
- *if $f(\tilde{x})$ is not defined, then $C[\widetilde{\ulcorner x_i \urcorner / z_i @ \mathsf{p}_i}] \not\to^* \mathbf{0}$, and q never receives any messages.*

Theorem 12 (Soundness)**.** *If $f : \mathbb{N}^n \to \mathbb{N}$ is a partial recursive function, then $\{\!\!\{ [\![f]\!]^{\tilde{\mathsf{p}} \mapsto \mathsf{q}} \}\!\!\}$ implements f with input variables $\widetilde{\mathsf{p}.x}$ and output variable $\mathsf{q}.x$.*

Proof. Consequence of the proof of Theorem 5 and of Theorem 11: as the only free variables in $\{\!\!\{ [\![f]\!]^{\tilde{\mathsf{p}} \mapsto \mathsf{q}} \}\!\!\}$ are $\mathsf{p}_i.x$ for $1 \leq i \leq n$, $\{\!\!\{ [\![f]\!]^{\tilde{\mathsf{p}} \mapsto \mathsf{q}} \}\!\!\} [\ulcorner x_i \urcorner / z_i @ \mathsf{p}_i]$ coincides with $\{\!\!\{ [\![f]\!]^{\tilde{\mathsf{p}} \mapsto \mathsf{q}} \}\!\!\}_\sigma$ whenever σ contains $\ulcorner x_i \urcorner$ at each process p_i. $\square$

We end our development by combining our results to characterise a Turing-complete and deadlock-free fragment of ChP.

Let ChP^{ChC} be the smallest fragment of ChP containing the projections of all typable and projectable choreographies in ChC, formally: $\text{ChP}^{\text{ChC}} = \{ [\![C]\!] \mid [\![C]\!] \text{ is defined} \}$. By Theorem 10, all terms in ChP^{ChC} are deadlock-free.

We now show that ChP^{ChC} is also Turing powerful. The development is similar to that for SP^{CC} (Section 4.3), but we need two additional steps. First, the operational correspondence theorem for the EPP of ChC (Theorem 10) needs the projected channel choreography to be well-typed. Fortunately, this is always the case for the channel choreographies obtained by embedding amended CC terms.

Lemma 12. *Let C be a core choreography and σ a state. Then $C' = \{\!\!\{\mathsf{Amend}(C)\}\!\!\}_\sigma$ implies $\Gamma; \Theta \vdash C' \triangleright \Delta$ for some Γ, Θ and Δ.*

Proof. Choosing Θ is trivial, as each process has its own role. For Γ, we assign type **nat** to all variables. Finally, $\Delta = k : G$, where G is inferred by abstracting the communications in C. The inductive construction of the latter is always possible since we applied Amend, so we can type each conditional with either the same global type or a branching global type with two labels. $\square$

Second, we need to know that the embedding of a projectable core choreography is also projectable in ChC.

Lemma 13. *If C is projectable, then $\{C\}_\sigma$ is projectable for any σ.*

Proof. The thesis follows from the fact that: if the projections of two choreographies are mergeable in CC, then the projections of their embeddings into ChC are mergeable. This is proven by structural induction. □

Using these results, the proof of Corollary 1 can be adapted to yield the following property.

Corollary 4 (Turing completeness of $\mathrm{ChP}^{\mathrm{ChC}}$)**.** *Every partial recursive function is implementable in* $\mathrm{ChP}^{\mathrm{ChC}}$.

We thus characterise a fragment of the session-based π-calculus from (Coppo et al., 2016) that contains only deadlock-free terms and is Turing complete.

6.2.6. Asynchronous semantics

The original semantics of ChC is made asynchronous by means of an additional rule that allows some transitions protected by a prefix to be executed, yielding more possible reduction sequences. In this scenario, the operational correspondence between CC and ChC is no longer as strong as stated in Theorem 11; in particular, the result for soundness now reads

(Soundness) $\{C\}_\sigma \to^+ C'$ implies $C, \sigma \to^+ C^*, \sigma^*$ and $C' \to^* \{C^*\}_{\sigma^*}$

However, the reduction relation in the asynchronous setting, restricted to the language fragment we consider (embeddings of core choreographies, which in particular are typable) is still confluent, and it includes all synchronous executions. Therefore, Theorem 12 remains valid in this general case.

6.3. Implications of the Embedding

Channel Choreographies. Our formal translation from CC to ChC 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, and multiparty sessions. While useful in practice, these primitives come at the cost of making the formal treatment of ChC technically involved. In particular, ChC (as well as 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.

Other Choreography Languages. The language WS-CDL from W3C (W3C WS-CDL Working Group, 2004) and the formal models inspired by it – e.g., (Carbone et al., 2012) – 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 (Dalla Preda et al., 2015), which adds higher-order features to choreographies to achieve runtime adaptation. Finally, the language of compositional choreographies presented in (Montesi and Yoshida, 2013) 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.

Process Languages. Our embedding of CC in ChC identifies a fragment of ChP, via EPP, that is also Turing complete. This fragment is isomorphic to value-passing CCS (Milner, 1980): since we only have one channel, we can interpret the constructs $k[\mathtt{A}]!\mathtt{B}\langle e\rangle$ and $k[\mathtt{B}]?\mathtt{A}(x)$ as sending and receiving over a channel with name $k\mathtt{AB}$. We thus obtain a deadlock-free and Turing complete fragment of value-passing CCS.

Since ChC has also been translated to the Jolie programming language (Gabbrielli et al., 2015; Montesi et al., 2014), our reasoning also applies to the latter and, in general, to service-oriented languages based on message correlation. Namely, our results identify a deadlock-free and Turing complete fragment of Jolie.

7. Related Work and Discussion

Register Machines. The computational primitives in CC recall those of the Unlimited Register Machine (URM) (Cutland, 1980), 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) (Honda et al., 2016) and conversation types (Caires and Vieira, 2010). 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 (Brand and Zafiropulo, 1983) that respect the property of multiparty compatibility (Deniélou and Yoshida, 2013). By contrast, for choreographies used as concrete implementations (our interest here), this question has barely been scratched before this work: session-typed choreographies with finite traces correspond to proofs in multiplicative-additive linear logic (Carbone et al., 2014). The language in (Carbone et al., 2014) does not include any constructs for programming repetitive behaviour. To the best of our knowledge, MC is the first choreography language to be identified as minimally Turing complete.

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 $C \triangleq \mathsf{p.c} \texttt{ -> } \mathsf{q}; \mathsf{q}.\varepsilon \texttt{ -> } \mathsf{r}; \mathbf{0}$. 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 (W3C WS-CDL Working Group, 2004). 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 (Carbone and Montesi, 2013; Carbone et al., 2014; Montesi and Yoshida, 2013): 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 (Church and Rosser, 1936). 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 by Lanese et al. (2013) for a simple language with finite traces (thus not Turing complete). Our definition is different, since it uses merging for the first time.

We could define our amendment procedure in different ways, e.g., by propagating selections from a process to another as a chain, rather than from one process to all the others. This would not influence our results.

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.

8. Conclusions

We developed Core Choreographies (CC), a calculus which we argued can constitute a foundation for choreographic programming. We motivated the primitives for CC by showing that they extend a minimal set necessary for Turing completeness by including only label selection, which is a characteristic feature of choreography calculi. For completeness' sake, we showed how to encode label selections as value communications, thus obtaining a smaller calculus of Minimal Choreographies (MC) that is still Turing complete.

The development of CC has already enabled us to study theoretical questions in choreographic programming in a minimal setting. In (Cruz-Filipe and Montesi, 2016d), we showed that CC is unable to encode asynchronous communication, and discussed which primitives to add in order to be able to mimic this behaviour in a synchronous semantics. In (Cruz-Filipe et al., 2016), we considered the problems of (i) deciding whether a process implementation can be described by a choreography, and (ii) extracting such a choreography in the affirmative case; we showed that, for CC and SP, both problems are solvable in exponential time. Finally, in (Cruz-Filipe and Montesi, 2016c) we studied how to extend CC with general sequential composition – a feature that is unavailable in most choreography calculi equipped with recursion – together with other commonly occurring primitives that are useful in practice (process spawning and name passing). The expressivity of the resulting calculus was demonstrated in (Cruz-Filipe and Montesi, 2016a).

The results in Section 6.2 justify that it makes sense to use CC for these studies, in addition to the technical convenience of working in a setting that is as simple as possible. Our study of encoding asynchronous communications (Cruz-Filipe and Montesi, 2016d) again identifies a minimal set of primitives that must be present in a choreography language to build such an encoding, and the encoding itself can be structurally extended to most languages. The results on extraction (Cruz-Filipe et al., 2016) give us a lower bound for the complexity of deciding the same problem in other choreography calculi (since they all include CC), and identify some technical challenges that will always be present. The work in (Cruz-Filipe and Montesi, 2016c) and (Cruz-Filipe and Montesi, 2016a), on the other hand, illustrates how concepts defined for CC (for example, the notion of "running in parallel", Definition 4) are naturally applicable in more complex settings.

Our results support the claim that there are substantial advantages to gain from first studying choreographic programming in itself, abstracting from features that are specific to other models (like channels in process calculi), and then applying the obtained insights to particular scenarios. We believe that CC is a useful step in this direction, and that it will serve as a stepping stone for the future developments of the paradigm of choreographic programming in general.

Acknowledgements

We thank Hugo Torres Vieira and Gianluigi Zavattaro for their useful comments. Montesi was supported by CRC (Choreographies for Reliable and efficient Communication software), grant no. DFF–4005-00304 from the Danish Council for Independent Research.

References

Barendregt, H., 1984. The Lambda Calculus: Its Syntax and Semantics, 2nd Edition. North Holland.

Brand, D., Zafiropulo, P., Apr. 1983. On communicating finite-state machines. J. ACM 30 (2), 323–342.
URL `http://doi.acm.org/10.1145/322374.322380`

Caires, L., Vieira, H. T., 2010. Conversation types. Theor. Comput. Sci. 411 (51-52), 4399–4440.

Carbone, M., Honda, K., Yoshida, N., 2012. Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. 34 (2), 8.

Carbone, M., Montesi, F., 2013. Deadlock-freedom-by-design: multiparty asynchronous global programming. In: Giacobazzi, R., Cousot, R. (Eds.), POPL. ACM, pp. 263–274.

Carbone, M., Montesi, F., Schürmann, C., 2014. Choreographies, logically. In: Baldan, P., Gorla, D. (Eds.), CONCUR. Vol. 8704 of LNCS. Springer, pp. 47–62.

Chor, 2016. Programming Language. http://www.chor-lang.org/.

Church, A., Rosser, J., 1936. Some properties of conversion. Transactions of the American Mathematical Society 39 (3), 472–482.
URL `http://www.jstor.org/stable/1989762`

Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L., 2016. Global progress for dynamically interleaved multiparty sessions. Mathematical Structures in Computer Science 26 (2), 238–302.
URL `http://dx.doi.org/10.1017/S0960129514000188`

Cruz-Filipe, L., Larsen, K. S., Montesi, F., 2016. The paths to choreography extraction. CoRR abs/1610.10050, submitted for publication.

Cruz-Filipe, L., Montesi, F., 2016a. Choreographies in practice. In: Albert, E., Lanese, I. (Eds.), FORTE 2016. Vol. 9688 of LNCS. Springer, pp. 114–123.

Cruz-Filipe, L., Montesi, F., 2016b. A core model for choreographic programming. In: Koshravi, R., Kouchnarenko, O. (Eds.), FACS. LNCS. Springer, accepted for publication.
URL `http://arxiv.org/abs/1510.03271`

Cruz-Filipe, L., Montesi, F., 2016c. A language for the declarative composition of concurrent protocols. CoRR abs/1602.03729, submitted for publication.

Cruz-Filipe, L., Montesi, F., 2016d. That's enough: Asynchrony with standard choreography primitives, submitted for publication.

Cutland, N., 1980. Computability: an Introduction to Recursive Function Theory. Cambridge University Press.

Dalla Preda, M., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J., 2015. Dynamic choreographies – safe runtime updates of distributed applications. In: Holvoet, T., Viroli, M. (Eds.), COORDINATION. Vol. 9037 of LNCS. Springer, pp. 67–82.
URL `http://dx.doi.org/10.1007/978-3-319-19282-6_5`

Deniélou, P.-M., Yoshida, N., 2013. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In: Fomin, F., Freivalds, R., Kwiatkowska, M., Peleg, D. (Eds.), ICALP (II). Vol. 7966 of LNCS. Springer, pp. 174–186.

Gabbrielli, M., Giallorenzo, S., Montesi, F., 2015. Applied choreographies. CoRR abs/1510.03637.
URL `http://arxiv.org/abs/1510.03637`

Honda, K., Vasconcelos, V., Kubo, M., 1998. Language primitives and type disciplines for structured communication-based programming. In: Hankin, C. (Ed.), ESOP. Vol. 1381 of LNCS. Springer, pp. 122–138.

Honda, K., Yoshida, N., Carbone, M., 2008. Multiparty asynchronous session types. In: Necula, G., Wadler, P. (Eds.), POPL. ACM, pp. 273–284.

Honda, K., Yoshida, N., Carbone, M., 2016. Multiparty asynchronous session types. J. ACM 63 (1), 9.
URL `http://doi.acm.org/10.1145/2827695`

Kleene, S., 1952. Introduction to Metamathematics. North-Holland Publishing Co.

Kouzapas, D., Yoshida, N., 2013. Globally governed session semantics. In: CONCUR. pp. 395–409.

Lanese, I., Guidi, C., Montesi, F., Zavattaro, G., 2008. Bridging the gap between interaction- and process-oriented choreographies. In: Cerone, A., Gruner, S. (Eds.), SEFM. IEEE, pp. 323–332.

Lanese, I., Montesi, F., Zavattaro, G., 2013. Amending choreographies. In: Ravara, A., Silva, J. (Eds.), WWV 2013. Vol. 123 of EPTCS. pp. 34–48.

Leesatapornwongsa, T., Lukman, J. F., Lu, S., Gunawi, H. S., 2016. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In: ASPLOS. ACM, pp. 517–530.

Lu, S., Park, S., Seo, E., Zhou, Y., 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS. ACM, pp. 329–339.

Milner, R., 1980. A Calculus of Communicating Systems. Vol. 92 of LNCS. Springer, Berlin.

Montesi, F., 2013. Choreographic programming. Ph.D. thesis, IT University of Copenhagen, http://fabriziomontesi.com/files/choreographic_programming.pdf.

Montesi, F., Guidi, C., Zavattaro, G., 2014. Service-oriented programming with Jolie. In: Bouguettaya, A., Sheng, Q., Daniel, F. (Eds.), Web Services Foundations. Springer, pp. 81–107.

Montesi, F., Yoshida, N., 2013. Compositional choreographies. In: D'Argenio, P., Melgratti, H. (Eds.), CONCUR. Vol. 8052 of LNCS. Springer, pp. 425–439.

Needham, R. M., Schroeder, M. D., Dec. 1978. Using encryption for authentication in large networks of computers. Commun. ACM 21 (12), 993–999. URL `http://doi.acm.org/10.1145/359657.359659`

Qiu, Z., Zhao, X., Cai, C., Yang, H., 2007. Towards the theoretical foundation of choreography. In: Williamson, C., Zurko, M., Patel-Schneider, P., Shenoy, P. (Eds.), WWW. ACM, pp. 973–982.

Sangiorgi, D., Walker, D., 2001. The π-calculus: a Theory of Mobile Processes. Cambridge University Press.

Turing, A., 1937. Computability and λ-definability. J. Symb. Log. 2 (4), 153–163.

W3C WS-CDL Working Group, 2004. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/.