Procedural Choreographic Programming

Luís Cruz-Filipe and Fabrizio Montesi

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

Abstract. Choreographic Programming is an emerging paradigm for correct-by-construction concurrent programming. However, its applicability is limited by the current lack of support for reusable procedures. We propose Procedural Choreographies (PC), a choreographic language model with full procedural abstraction. PC includes unbounded process creation and name mobility, yielding a powerful framework for writing correct concurrent algorithms that can be compiled into a process calculus. This increased expressivity requires a typing discipline to ensure that processes are properly connected when enacting procedures.

1 Introduction

Choreographic Programming [20] is a paradigm for programming concurrent software that is deadlock-free by construction, by using an "Alice and Bob" notation to syntactically prevent mismatched I/O communications in programs (called choreographies) and using an EndPoint Projection to synthesise correct process implementations [2,4,24]. Choreographies are found in standards [1,26], languages [6,14,23,25], and specification models [2,4,17]. They are widely used as a design tool in communication-based software [1,23,25,26], since they describe interactions unambiguously and thus help ensure correctness [19].

Driven by these benefits, research on applicability of choreographic programming has recently gained in breadth, ranging from service programming [2,4] to runtime adaptation [11]. We focus on another important aspect: modular programming. Writing procedures that can be arbitrarily instantiated and composed into larger programs is still unsupported. The absence of full procedural abstraction disallows the creation of libraries that can be reused as "black boxes".

Example 1. We discuss a parallel version of merge sort, written as a choreography. Although this is a toy example, it cannot be written in any previous model for choreographic programming. We present more realistic and involved examples in the remainder. We make the standard assumption that we have concurrent processes with local storage and computational capabilities. In this example, each process stores a list and can use the following local functions: `split1` and `split2`, respectively returning the first or the second half of a list; `is_small`, which tests if a list has at most one element; and `merge`, which combines two sorted lists into one. The following (choreographic) procedure, `MS`, implements merge sort on the list stored at its parameter process `p`.[1]

[1] In this work, we use a `monospaced` font for readability of our concrete examples, and other fonts for distinguishing syntactic categories in our formal arguments as usual.

```
MS(p) = if p.is_small then 0
        else p start q1,q2; p.split1 -> q1; p.split2 -> q2;
             MS<q1>; MS<q2>; q1.* -> p; q2.* -> p.merge
```

Procedure `MS` starts by checking whether the list at process `p` is small, in which case it does not need to be sorted (`0` denotes termination); otherwise, `p` starts two other processes `q1` and `q2` (`p start q1,q2`), to which it respectively sends the first and the second half of the list (`p.split1 -> q1` and `p.split2 -> q2`). The procedure is recursively reapplied to `q1` and `q2`, which independently (concurrently) proceed to ordering their respective sub-lists. When this is done, `MS` stores the first ordered half from `q1` to `p` (`q1.* -> p`, where `*` retrieves the data stored in `q1`) and merges it with the ordered sub-list from `q2` (`q2.* -> p.merge`).

Procedure `MS` in Example 1 cannot be written in current choreography models, because it uses two unsupported features: *general recursion*, allowing procedure calls to be followed by arbitrary code; and *parametric procedures*, which can be reused with different processes (as in `MS<q1>` and `MS<q2>`).

We present Procedural Choreographies (PC), a model for choreographic programming that captures these features (§ 2). PC has a simple syntax, but its semantics is expressive enough to infer safe out-of-order executions of choreographic procedures – for example, in `MS<q1>; MS<q2>`, the two calls can be run in parallel because they involve separate processes and are thus non-interfering.

We also illustrate the expressivity of PC with a more involved parallel downloader, showing how our semantics infers parallel executions in a complex scenario of concurrent data streams. This example makes use of additional features: mobility of process names (networks with connections that evolve at runtime) and propagation of choices among processes.

The interplay between name mobility and procedure composition requires careful handling, because of potential dangling process references. We prevent such errors using a decidable typing discipline (§ 3) that supports type inference.

PC includes an EndPoint Projection (EPP) that synthesises correct concurrent implementations in terms of a process calculus (§ 4). This process calculus is an abstraction of systems where processes refer to one another's locations or identifiers (e.g., MPI [22] or the Internet Protocol).

Full definitions, proofs, and further extensions are given in [10]. Additional examples can be found in [8] (which is based on a pre-print of this article).

2 Procedural Choreographies (PC)

Syntax. The syntax of PC is displayed in Figure 1. A procedural choreography is a pair $\langle \mathscr{D}, C\rangle$, where C is a choreography and $\mathscr{D}$ is a set of procedure definitions. Process names ($\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$), identify processes that execute concurrently. Each process is equipped with a memory cell that stores a single value of a fixed type. Specifically, we consider a fixed set $\mathbb{T}$ of datatypes (numbers, lists, etc.); each process p stores only values of type $T_{\mathsf{p}} \in \mathbb{T}$. Statements in a choreography can either be communication actions (η) or compound instructions (I), both of which

$$C ::= \eta; C \mid I; C \mid \mathbf{0} \qquad \eta ::= \mathsf{p}.e \texttt{ -> } \mathsf{q}.f \mid \mathsf{p} \texttt{ -> } \mathsf{q}[l] \mid \mathsf{p}\,\mathsf{start}\,\mathsf{q}^T \mid \mathsf{p}\colon \mathsf{q} \texttt{<->} \mathsf{r}$$

$$\mathscr{D} ::= X(\widetilde{\mathsf{q}^T}) = C, \mathscr{D} \mid \emptyset \qquad I ::= \mathsf{if}\,\mathsf{p}.e\,\mathsf{then}\,C_1\,\mathsf{else}\,C_2 \mid X\langle\tilde{\mathsf{p}}\rangle \mid \mathbf{0}$$

Fig. 1. Procedural Choreographies, Syntax.

can have continuations. Term $\mathbf{0}$ is the terminated choreography, which we often omit in examples. We call all terms but $\mathbf{0}; C$ *program terms*, or simply programs, since these form the syntax intended for developers to use for writing programs. Term $\mathbf{0}; C$ is necessary only for the technical definition of the semantics, to capture termination of procedure calls with continuations, and can appear only at runtime. It is thus called a *runtime term*.

Processes communicate through direct references (names) to each other.[2] In a value communication $\mathsf{p}.e$ -> $\mathsf{q}.f$, process p sends the result of evaluating expression e (replacing the placeholder $*$ at runtime with the data in its memory) to q. When q receives the value from p, it applies to it the (total) function f and stores the result. The definition of f may also access the contents of q's memory.

In a selection term p -> $\mathsf{q}[l]$, p communicates to q its choice of label l, which is a constant. This term is intended to propagate information on which internal choice has been made by a process to another (see Remark 2 below).

In term $\mathsf{p}\,\mathsf{start}\,\mathsf{q}^T$, process p spawns the new process q, which stores data of type T. Process name q is bound in the continuation C of $\mathsf{p}\,\mathsf{start}\,\mathsf{q}^T; C$.

Process spawning introduces the need for name mobility. In real-world systems, after execution of $\mathsf{p}\,\mathsf{start}\,\mathsf{q}^T$, p is the only process that knows q's name. Any other process wanting to communicate with q must therefore be first informed of its existence. This is achieved with the introduction term $\mathsf{p}\colon \mathsf{q} \texttt{<->} \mathsf{r}$, read "$\mathsf{p}$ introduces q and r" (with p, q and r distinct). As its double-arrow syntax suggests, this action represents *two* communications – one where p sends q's name to r, and another where p sends r's name to q. This is made explicit in § 4.

In a conditional term $\mathsf{if}\,\mathsf{p}.e\,\mathsf{then}\,C_1\,\mathsf{else}\,C_2$, process p evaluates e to choose between the possible continuations C_1 and C_2.

The set $\mathscr{D}$ contains global procedures. Term $X(\widetilde{\mathsf{q}^T}) = C_X$ defines a procedure X with body C_X, which can be used anywhere in $\langle \mathscr{D}, C\rangle$ – in particular, inside C_X. The names $\tilde{\mathsf{q}}$ are bound to C_X, and they are exactly the free process names in C_X. Each procedure can be defined at most once in $\mathscr{D}$. Term $X\langle\tilde{\mathsf{p}}\rangle$ calls (invokes) procedure X by passing $\tilde{\mathsf{p}}$ as parameters. Procedure calls inside definitions must be guarded, i.e., they can only occur after some other action.

We assume the Barendregt convention and work up to α-equivalence in choreographies, renaming bound variables as needed when expanding procedure calls.

Example 2. Recall procedure `MS` from our merge sort example in the Introduction (Example 1). If we annotate the parameter p and the started processes q_1 and q_2 with a type, e.g., $\mathbf{List}(T)$ for some T (the type of lists containing elements

[2] PC thus easily applies to settings based on actors, objects, or ranks (e.g., MPI).

of type T), then MS is a valid procedure definition in PC, as long as we allow two straightforward syntactic conventions: (i) $\mathsf{p}\,\mathsf{start}\,\widetilde{\mathsf{q}^T}$ stands for the sequence $\mathsf{p}\,\mathsf{start}\,\mathsf{q}_1^{T_1}; \ldots; \mathsf{p}\,\mathsf{start}\,\mathsf{q}_n^{T_n}$; (ii) a communication of the form $\mathsf{p}.e$ -> q stands for $\mathsf{p}.e$ -> $\mathsf{q}.\mathsf{id}$, where id is the identity function: it sets the content of q to the value received from p. We adopt these conventions also in the remainder.

Remark 1 (Design choices). We comment on two of our design choices.

The introduction action (p : q <-> r) requires a three-way synchronization, essentially performing two communications. The alternative development of PC with asymmetric introduction (an action p: q -> r whereby p sends q's name to r, but not conversely) would be very similar. Since in our examples we always perform introductions in pairs, the current choice makes the presentation easier.

The restriction that each process stores only one value of a fixed type is, in practice, a minor constraint. As shown in Example 2, types can be tuples or lists, which mimics storing several values. Also, a process can create new processes with different types – so we can encode changing the type of p by having p create a new process p' and then continuing the choreography with p' instead of p.

Remark 2 (Label Selection). We motivate the need for selections (p -> $\mathsf{q}[l]$). Consider the choreography if $\mathsf{p}.\mathsf{coinflip}$ then ($\mathsf{p}.*$ -> r) else ($\mathsf{r}.*$ -> p). Here, p flips a coin to decide whether to send a value to r or to receive a value from r. Since processes run independently and share no data, only p knows which branch of the conditional will be executed; but this information is essential for r to decide on its behaviour. To propagate p's decision to r, we use selections:

$$\text{if } \mathsf{p}.\mathsf{coinflip} \text{ then } (\mathsf{p} \text{ -> } \mathsf{r}[\mathrm{L}]; \mathsf{p}.* \text{ -> } \mathsf{r}) \text{ else } (\mathsf{p} \text{ -> } \mathsf{r}[\mathrm{R}]; \mathsf{r}.* \text{ -> } \mathsf{p})$$

Now r receives a label reflecting p's choice, and can use it to decide what to do.

Selections are needed only for compilation (see § 4): the first choreography above is not projectable, whereas the second one is. They can be inferred, and thus could be removed from the user syntax, but it is useful to be able to specify them manually (see Remark 4). See also Example 5 at the end of this section.

Semantics. We define a reduction semantics $\to_{\mathscr{D}}$ for PC, parameterised over $\mathscr{D}$ (Figure 2, top). Given a choreography C, we model the state of its processes with a state function σ, with domain $\mathsf{pn}(C)$, where $\sigma(\mathsf{p})$ denotes the value stored in p. We assume that each type $T \in \mathbb{T}$ has a special value $\perp_T$, representing an uninitialised process state. We also use a connection graph G, keeping track of which processes know each other. In the rules, $\mathsf{p} \xleftrightarrow{G} \mathsf{q}$ denotes that G contains an edge between p and q, and $G \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\}$ denotes the graph obtained from G by adding an edge between p and q (if missing).

Executing a communication action $\mathsf{p}.e$ -> $\mathsf{q}.f$ in rule $\lfloor \mathrm{C}|\mathrm{Com} \rceil$ requires that: p and q are connected in G; e is well typed; and the type of e matches that expected by the function f at the receiver. The last two conditions are encapsulated in the notation $e \downarrow v$, read "e evaluates to v". Choreographies can thus deadlock (be unable to reduce) because of errors in the programming of communications; this issue is addressed by our typing discipline in § 3.

$$\frac{\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q} \qquad e[\sigma(\mathsf{p})/*] \downarrow v \qquad f[\sigma(\mathsf{q})/*](v) \downarrow w}{G, \mathsf{p}.e \ \texttt{->}\ \mathsf{q}.f; C, \sigma \ \to_{\mathscr{D}}\ G, C, \sigma[\mathsf{q} \mapsto w]} \ \lfloor \text{C}|\text{Com} \rceil$$

$$\frac{\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q}}{G, \mathsf{p} \ \texttt{->}\ \mathsf{q}[l]; C, \sigma \ \to_{\mathscr{D}}\ G, C, \sigma} \ \lfloor \text{C}|\text{Sel} \rceil$$

$$\frac{}{G, \mathsf{p}\, \mathsf{start}\, \mathsf{q}^T; C, \sigma \ \to_{\mathscr{D}}\ G \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\}, C, \sigma[\mathsf{q} \mapsto \bot_T]} \ \lfloor \text{C}|\text{Start} \rceil$$

$$\frac{\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q} \qquad \mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{r}}{G, \mathsf{p} \colon \mathsf{q} \ \texttt{<->}\ \mathsf{r}; C, \sigma \ \to_{\mathscr{D}}\ G \cup \{\mathsf{q} \leftrightarrow \mathsf{r}\}, C, \sigma} \ \lfloor \text{C}|\text{Tell} \rceil$$

$$\frac{i = 1 \text{ if } e[\sigma(\mathsf{p})/*] \downarrow \mathsf{true}, \qquad i = 2 \text{ otherwise}}{G, (\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2); C, \sigma \ \to_{\mathscr{D}}\ G, C_i \fatsemi C, \sigma} \ \lfloor \text{C}|\text{Cond} \rceil$$

$$\frac{C_1 \preceq_{\mathscr{D}} C_2 \qquad G, C_2, \sigma \ \to_{\mathscr{D}}\ G', C_2', \sigma' \qquad C_2' \preceq_{\mathscr{D}} C_1'}{G, C_1, \sigma \ \to_{\mathscr{D}}\ G', C_1', \sigma'} \ \lfloor \text{C}|\text{Struct} \rceil$$

$$\frac{\mathsf{pn}(\eta) \# \mathsf{pn}(\eta')}{\eta; \eta' \equiv_{\mathscr{D}} \eta'; \eta} \ \lfloor \text{C}|\text{Eta-Eta} \rceil \qquad \frac{\mathsf{pn}(I) \# \mathsf{pn}(I')}{I; I' \equiv_{\mathscr{D}} I'; I} \ \lfloor \text{C}|\text{I-I} \rceil$$

$$\frac{X(\widetilde{\mathsf{q}^T}) = C_X \in \mathscr{D}}{X\langle \tilde{\mathsf{p}} \rangle; C \preceq_{\mathscr{D}} C_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \fatsemi C} \ \lfloor \text{C}|\text{Unfold} \rceil$$

Fig. 2. Procedural Choreographies, Semantics and Structural Precongruence (selected rules).

Rule $\lfloor$C|Sel$\rceil$ defines selection as a no-op for choreographies (see Remark 2).

Rule $\lfloor$C|Start$\rceil$ models the creation of a process. In the reductum, the starter and started processes are connected and can thus communicate with each other. This rule also extends the domain of the state function σ accordingly. Rule $\lfloor$C|Tell$\rceil$ captures name mobility, creating a connection between two processes q and r when they are introduced by a process p connected to both.

Rule $\lfloor$C|Cond$\rceil$ uses the auxiliary operator $\fatsemi$ to obtain a reductum in the syntax of PC regardless of the forms of the branches C_1 and C_2 and the continuation C. The operator $\fatsemi$ is defined by $\eta \fatsemi C = \eta; C$, $I \fatsemi C = I; C$ and $(C_1; C_2) \fatsemi C = C_1; (C_2 \fatsemi C)$. It extends the scope of bound names: any name p bound in C has its scope extended also to C'. This scope extension is capture-avoiding, as the Barendregt convention guarantees that p is not used in C'.

Rule $\lfloor$C|Struct$\rceil$ uses structural precongruence $\preceq_{\mathscr{D}}$. The main rules defining $\preceq_{\mathscr{D}}$ are given in Figure 2 (bottom). We write $C \equiv_{\mathscr{D}} C'$ when $C \preceq_{\mathscr{D}} C'$ and $C' \preceq_{\mathscr{D}} C$, $\mathsf{pn}(C)$ for the set of process names (free or bound) in a choreography C, and $A \# B$ when two sets A and B are disjoint. These rules formalise the notion of parallelism in PC, recalling out-of-order execution. Rule $\lfloor$C|Eta-Eta$\rceil$ permutes two communications performed by processes that are all distinct, modelling that processes run independently of one another. For example, $\mathsf{p}.* \texttt{ -> } \mathsf{q}; \mathsf{r}.* \texttt{ -> } \mathsf{s} \equiv_{\mathscr{D}}$ $\mathsf{r}.* \texttt{ -> } \mathsf{s}; \mathsf{p}.* \texttt{ -> } \mathsf{q}$ because these two communications are non-interfering, but $\mathsf{p}.* \texttt{ -> } \mathsf{q}; \mathsf{q}.* \texttt{ -> } \mathsf{s} \not\equiv_{\mathscr{D}} \mathsf{q}.* \texttt{ -> } \mathsf{s}; \mathsf{p}.* \texttt{ -> } \mathsf{q}$: since the second communication causally depends on the first (both involve q).

This reasoning is extended to instructions in rule $\lfloor$C|I-I$\rceil$; in particular, procedure calls that share no arguments can be swapped. This is sound, as a procedure can only refer to processes that are either passed as arguments or started inside its body, and the latter cannot be leaked to the original call site. Thus, any actions obtained by unfolding the first procedure call involve different processes than those obtained by unfolding the second one. As the example below shows, calls to the same procedure can be exchanged, since X and Y need not be distinct. Omitted rules include moving actions inside or outside both branches of a conditional, or switching independent nested conditionals. Rule $\lfloor$C|Unfold$\rceil$ unfolds a procedure call, again using the $\fatsemi$ operator defined above.

Example 3. In our merge sort example, structural precongruence $\preceq_{\mathscr{D}}$ allows the recursive calls MS<q_1> and MS<q_2> to be exchanged. Furthermore, after the calls are unfolded, their code can be interleaved in any way.

This example exhibits map-reduce behaviour: each new process receives its input, runs independently from all others, and then sends its result to its creator.

Example 4. In a more refined example of implicit parallelism, we swap communications from procedure calls that share process names. Consider the procedure

```
auth(c,a,r,l) = c.creds -> a.rCreds;
                a.chk -> r.res; a.log -> l.app
```

Client c sends its credentials to an authentication server a, which stores the result of authentication in r and appends a log of this operation at process l. In the choreography auth<c,a1,r1,l>; auth<c,a2,r2,l>, a client c authenticates at two different authentication servers a1 and a2. After unfolding the two calls, rule $\lfloor$C|Eta-Eta$\rceil$ yields the following interleaving:

```
c.creds -> a1.rCreds; c.creds -> a2.rCreds;
a2.chk -> r2.res; a1.chk -> r1.res;
a1.log -> l.app; a2.log -> l.app
```

Thus, the two authentications proceed in parallel. Observe that the logging operations cannot be swapped, since they use the same logging process l.

Example 5. A more sophisticated example involves modularly composing different procedures that take multiple parameters. Here, we write a choreography where a client c downloads a collection of files from a server s. Files are downloaded in parallel via streaming, by having the client and the server each create subprocesses to handle the transfer of each file. Thus, the client can request and start downloading each file without waiting for previous downloads to finish.

```
par_download(c,s) = if c.more
  then c -> s [more]; c start c'; s start s';
       s: c <-> s'; c.top -> s'; pop<c>;
```

```
          c: c' <-> s'; download<c',s'>;
          par_download<c,s>; c'.file -> c.store
  else c -> s [end]
```

At the start of `par_download`, the client `c` checks whether it wants to download more files and informs the server `s` of the result via a label selection. In the affirmative case, the client and the server start two subprocesses, c' and s' respectively, and the server introduces `c` to s' (`s: c <-> s'`). The client `c` sends to s' the name of the file to download (`c.top -> s'`) and removes it from its collection, using procedure `pop` (omitted), afterwards introducing its own subprocess c' to s'. The file download is handled by c' and s' (using procedure `download`), while `c` and `s` continue operating (`par_download<c,s>`). Finally, c' waits until `c` is ready to store the downloaded file.

Procedure `download` has a similar structure. It implements a stream where a file is sequentially transferred in chunks from a process `s` to another process `c`.

```
download(c,s) = if s.more
  then s -> c [more]; s.next -> c.app; pop<s>; download<c,s>
  else s -> c [end]
```

The implementation of `par_download` exploits implicit parallelism considerably. All calls to `download` are made with disjoint sets of parameters (processes), and can thus be fully parallelised: many instances of `download` run at the same time, each one implementing a (sequential) stream. Due to our semantics, we effectively end up executing many streaming behaviours in parallel.

We can even compose `par_download` with `auth`, such that we execute the parallel download only if the client can successfully authenticate with an authentication server `a`. Below, we use the shortcut $\mathsf{p} \texttt{->} \tilde{\mathsf{q}}[l]$ for $\mathsf{p} \texttt{->} \mathsf{q}_1[l]; \ldots; \mathsf{p} \texttt{->} \mathsf{q}_n[l]$.

```
auth<c,a,r,l>; if r.ok then r -> c,s[ok]; par_download<c,s>
                       else r -> c,s[ko]
```

3 Typability and Deadlock-Freedom

We give a typing discipline for PC, to check that (a) the types of functions and processes are respected by communications and (b) processes that need to communicate are first properly introduced (or connected). Regarding (b), two processes created independently can communicate only after they receive the names of each other. For instance, in Example 5, the execution of `download<c',s'>` would get stuck if `c'` and `s'` were not properly introduced in `par_download`, since our semantics requires them to be connected.

Typing judgements have the form $\Gamma; G \vdash C \triangleright G'$, read "$C$ is well-typed according to Γ, and running C with a connection graph that contains G yields a connection graph that includes G'". Typing environments Γ are used to track the types of processes and procedures; they are defined as: $\Gamma ::= \emptyset \mid \Gamma, \mathsf{p} : T \mid \Gamma, X : G \triangleright G'$. A typing $\mathsf{p} : T$ states that process p stores values of type T, and a typing $X : G \triangleright G'$ records the effect of the body of X on graph G.

$$\frac{}{\Gamma; G \vdash \mathbf{0} \triangleright G}\ \lfloor \text{T}|\text{End} \rceil \qquad \frac{\mathsf{p} \xleftrightarrow{G} \mathsf{q} \qquad \Gamma; G \vdash C \triangleright G'}{\Gamma; G \vdash \mathsf{p}\ \texttt{->}\ \mathsf{q}[l]; C \triangleright G'}\ \lfloor \text{T}|\text{Sel} \rceil$$

$$\frac{\begin{array}{c}\mathsf{p} \xleftrightarrow{G} \mathsf{q} \qquad \Gamma \vdash \mathsf{p} : T_{\mathsf{p}}, \mathsf{q} : T_{\mathsf{q}} \qquad \Gamma; G \vdash C \triangleright G' \\ * : T_{\mathsf{p}} \vdash_{\mathbb{T}} e : T_1 \qquad * : T_{\mathsf{q}} \vdash_{\mathbb{T}} f : T_1 \to T_{\mathsf{q}}\end{array}}{\Gamma; G \vdash \mathsf{p}.e\ \texttt{->}\ \mathsf{q}.f; C \triangleright G'}\ \lfloor \text{T}|\text{Com} \rceil$$

$$\frac{\mathsf{p} \xleftrightarrow{G} \mathsf{q} \qquad \mathsf{p} \xleftrightarrow{G} \mathsf{r} \qquad \Gamma; G \cup \{\mathsf{q} \leftrightarrow \mathsf{r}\} \vdash C \triangleright G'}{\Gamma; G \vdash \mathsf{p}\texttt{:}\ \mathsf{q}\ \texttt{<->}\ \mathsf{r}; C \triangleright G'}\ \lfloor \text{T}|\text{Tell} \rceil$$

$$\frac{\Gamma, \mathsf{q} : T; G \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\} \vdash C \triangleright G'}{\Gamma; G \vdash \mathsf{p}\ \mathsf{start}\ \mathsf{q}^T; C \triangleright G'}\ \lfloor \text{T}|\text{Start} \rceil \qquad \frac{\Gamma; G \vdash C \triangleright G'}{\Gamma; G \vdash \mathbf{0}; C \triangleright G'}\ \lfloor \text{T}|\text{EndSeq} \rceil$$

$$\frac{\Gamma \vdash \mathsf{p} : T \qquad * : T \vdash_{\mathbb{T}} e : \mathsf{bool} \qquad \Gamma; G \vdash C_i \triangleright G_i \qquad \Gamma; G_1 \cap G_2 \vdash C \triangleright G'}{\Gamma; G \vdash (\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2); C \triangleright G'}\ \lfloor \text{T}|\text{Cond} \rceil$$

$$\frac{\begin{array}{c}\Gamma \vdash X(\widetilde{\mathsf{q}^T}) : G_X \triangleright G'_X \qquad \Gamma \vdash \mathsf{p}_i : T_i \\ G_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \subseteq G \qquad \Gamma; G \cup (G'_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]) \vdash C \triangleright G'\end{array}}{\Gamma; G \vdash X\langle \tilde{\mathsf{p}} \rangle; C \triangleright G'}\ \lfloor \text{T}|\text{Call} \rceil$$

Fig. 3. Procedural Choreographies, Typing Rules.

The rules for deriving typing judgements are given in Figure 3. We assume standard typing judgements for functions and expressions, and write $* : T \vdash_{\mathbb{T}} e : T$ and $* : T_1 \vdash_{\mathbb{T}} f : T_2 \to T_3$ meaning, respectively "e has type T assuming that $*$ has type T" and "f has type $T_2 \to T_3$ assuming that $*$ has type T_1". Verifying that communications respect the expected types is straightforward, using the connection graph G to track which processes have been introduced to each other. In rule $\lfloor$T$|$Start$\rceil$, we implicitly use the fact that q does not occur in G (again using the Barendregt convention). The final graph G' is only used in procedure calls (rule $\lfloor$T$|$Call$\rceil$). Other rules leave it unchanged.

To type a procedural choreography, we need to type its set of procedure definitions $\mathscr{D}$. We write $\Gamma \vdash \mathscr{D}$ if: for each $X(\widetilde{\mathsf{q}^T}) = C_X \in \mathscr{D}$, there is exactly one typing $X(\widetilde{\mathsf{q}^T}) : G_X \triangleright G'_X \in \Gamma$, and this typing is such that $\Gamma, \widetilde{\mathsf{q} : T}, G_X \vdash C_X \triangleright G'_X$. We say that $\Gamma \vdash \langle \mathscr{D}, C \rangle$ if $\Gamma, \Gamma_{\mathscr{D}}; G_C \vdash C, G'$ for some $\Gamma_{\mathscr{D}}$ such that $\Gamma_{\mathscr{D}} \vdash \mathscr{D}$ and some G', where G_C is the full graph whose nodes are the free process names in C. The choice of G_C is motivated by observing that (i) all top-level processes should know each other and (ii) eventual connections between processes not occuring in C do not affect its typability.

Well-typed choreographies either terminate or diverge.[3]

[3] Since we are interested in communications, we assume evaluation of functions and expressions to terminate on values with the right types (see § 5, Faults).

Theorem 1 (Deadlock freedom/Subject reduction). *Let $\langle \mathscr{D}, C\rangle$ be a procedural choreography. If $\Gamma \vdash \mathscr{D}$ and $\Gamma; G_1 \vdash C \triangleright G_1'$ for some Γ, G_1 and G_1', then either: (i) $C \preceq_{\mathscr{D}} \mathbf{0}$; or, (ii) for every σ, there exist G_2, C' and σ' such that $G_1, C, \sigma \to_{\mathscr{D}} G_2, C', \sigma'$ and $\Gamma'; G_2 \vdash C' \triangleright G_2'$ for some $\Gamma' \supseteq \Gamma$ and G_2'.*

Checking that $\Gamma \vdash \langle \mathscr{D}, C\rangle$ is not trivial, as it requires "guessing" $\Gamma_{\mathscr{D}}$. However, this set can be computed from $\langle \mathscr{D}, C\rangle$.

Theorem 2. *Given Γ, $\mathscr{D}$ and C, $\Gamma \vdash \langle \mathscr{D}, C\rangle$ is decidable.*

The key idea behind the proof of Theorem 2 is that type-checking may require expanding recursive definitions, but their parameters only need to be instantiated with process names from a finite set. A similar idea yields type inference for PC.

Theorem 3. *There is an algorithm that, given any $\langle \mathscr{D}, C\rangle$, outputs: (i) a set Γ such that $\Gamma \vdash \langle \mathscr{D}, C\rangle$, if such a Γ exists; or (ii)* `NO`*, if no such Γ exists.*

Theorem 4. *The types of arguments in procedure definitions and the types of freshly created processes can be inferred automatically.*

Remark 3 (Inferring introductions). These results allow us to omit type annotations in choreographies, if the types of functions and expressions at processes are given (in $\vdash_{\mathbb{T}}$). Thus, programmers can write choreographies as in our examples.

The same reasoning can be used to infer missing introductions ($\mathsf{p}\colon \mathsf{q} \texttt{<->} \mathsf{r}$) in a choreography automatically, thus lifting the programmer also from having to think about connections. However, while the types inferred for a choreography do not affect its behaviour, the placement of introductions does. In particular, when invoking procedures one is faced with the choice of adding the necessary introductions inside the procedure definition (weakening the conditions for its invocation) or in the code calling it (making the procedure body more efficient). For example, consider the procedure $X(\mathsf{p},\mathsf{q},\mathsf{r}) = \mathsf{p}.* \texttt{ -> } \mathsf{q}; \mathsf{p}\colon \mathsf{q} \texttt{<->} \mathsf{r}; \mathsf{q}.* \texttt{ -> } \mathsf{r}$, which requires only that p is connected with q and r to be invoked. If we invoke it twice with the same parameters, e.g., $X\langle \mathsf{p},\mathsf{q},\mathsf{r}\rangle; X\langle \mathsf{p},\mathsf{q},\mathsf{r}\rangle$, then we end up performing the same introduction $\mathsf{p}\colon \mathsf{q} \texttt{<->} \mathsf{r}$ twice. We could optimise this situation by rewriting the procedure without the introduction – $X(\mathsf{p},\mathsf{q},\mathsf{r}) = \mathsf{p}.* \texttt{ -> } \mathsf{q}; \mathsf{q}.* \texttt{ -> } \mathsf{r}$ – and then performing the introduction only once before invoking the procedure – $\mathsf{p} : \mathsf{q} \texttt{<->} \mathsf{r}; X\langle \mathsf{p},\mathsf{q},\mathsf{r}\rangle; X\langle \mathsf{p},\mathsf{q},\mathsf{r}\rangle$. However, this makes invoking the procedure more complicated, and the optimisation is possible only because the procedure is invoked multiple times with the same parameters. (The situation may be even more complicated, e.g., X may be invoked inside of a recursive definition.)

OK?

4 Synthesising Process Implementations

We now present our EndPoint Projection (EPP), which compiles a choreography to a concurrent implementation represented in terms of a process calculus.

4.1 Procedural Processes (PP)

We introduce our target process model, Procedural Processes (PP).

$$B ::= \mathsf{q}!e; B \mid \mathsf{p}?f; B \mid \mathsf{q}!!\mathsf{r}; B \mid \mathsf{p}?\mathsf{r}; B \mid \mathsf{q} \oplus l; B \mid \mathsf{p}\&\{l_i : B_i\}_{i\in I}; B$$
$$\mid \mathbf{0} \mid \mathsf{start}\,\mathsf{q}^T \triangleright B_2; B_1 \mid \mathsf{if}\, e \,\mathsf{then}\, B_1 \,\mathsf{else}\, B_2; B \mid X\langle\tilde{\mathsf{p}}\rangle; B \mid \mathbf{0}; B$$
$$\mathscr{B} ::= X(\tilde{\mathsf{q}}) = B, \mathscr{B} \mid \emptyset \qquad N, M ::= \mathsf{p} \triangleright_v B \mid (N \mid M) \mid \mathbf{0}$$

$$\frac{u = (f[w/*])(e[v/*])}{\mathsf{p} \triangleright_v \mathsf{q}!e; B_1 \mid \mathsf{q} \triangleright_w \mathsf{p}?f; B_2 \to_{\mathscr{B}} \mathsf{p} \triangleright_v B_1 \mid \mathsf{q} \triangleright_u B_2} \lfloor \text{P|Com} \rceil$$

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

$$\frac{\mathsf{q}' \text{ fresh}}{\mathsf{p} \triangleright_v (\mathsf{start}\,\mathsf{q}^T \triangleright B_2); B_1 \to_{\mathscr{B}} \mathsf{p} \triangleright_v B_1[\mathsf{q}'/\mathsf{q}] \mid \mathsf{q}' \triangleright_{\bot_T} B_2} \lfloor \text{P|Start} \rceil$$

$$\frac{}{\mathsf{p} \triangleright_v \mathsf{q}!!\mathsf{r}; B_1 \mid \mathsf{q} \triangleright_w \mathsf{p}?\mathsf{r}; B_2 \mid \mathsf{r} \triangleright_u \mathsf{p}?\mathsf{q}; B_3 \to_{\mathscr{B}} \mathsf{p} \triangleright_v B_1 \mid \mathsf{q} \triangleright_w B_2 \mid \mathsf{r} \triangleright_u B_3} \lfloor \text{P|Tell} \rceil$$

Fig. 4. Procedural Processes, Syntax and Semantics (selected rules).

Syntax. The syntax of PP is given in Figure 4 (top). A term $\mathsf{p} \triangleright_v B$ is a process, where p is its name, v is its value, and B is its behaviour. Networks, ranged over by N, M, are parallel compositions of processes, where $\mathbf{0}$ is the inactive network. Finally, $\langle \mathscr{B}, N\rangle$ is a procedural network, where $\mathscr{B}$ defines the procedures that the processes in N may invoke. Values, expressions and functions are as in PC.

A process executing a send term $\mathsf{q}!e; B$ sends the evaluation of expression e to q, and proceeds as B. Term $\mathsf{p}?f; B$ is the dual receiving action: the process executing it receives a value from p, combines it with its value as specified by f, and then proceeds as B. Term $\mathsf{q}!!\mathsf{r}$ sends process name r to q and process name q to r, making q and r "aware" of each other. The dual action is $\mathsf{p}?\mathsf{r}$, which receives a process name from p that replaces the bound variable r in the continuation. Term $\mathsf{q} \oplus l; B$ sends the selection of a label l to process q. Selections are received by the branching term $\mathsf{p}\&\{l_i : B_i\}_{i\in I}$, which can receive a selection for any of the labels l_i and proceed as B_i. Branching terms must offer at least one branch. Term $\mathsf{start}\,\mathsf{q} \triangleright B_2; B_1$ starts a new process (with a fresh name) executing B_2, and proceeds in parallel as B_1. Conditionals, procedure calls, and termination are standard. Term $\mathsf{start}\,\mathsf{q} \triangleright B_2; B_1$ binds q in B_1, and $\mathsf{p}?\mathsf{r}; B$ binds r in B.

Semantics. The main rules defining the reduction relation $\to_{\mathscr{B}}$ for PP are shown in Figure 4 (bottom). As in PC, they are parameterised on the set of behavioural procedures $\mathscr{B}$. Rule $\lfloor$P|Com$\rceil$ models value communication: a process p executing a send action towards a process q can synchronise with a receive-from-p action at q; in the reductum, f is used to update the memory of q by combining its contents with the value sent by p. The placeholder $*$ is replaced with the current value of p in e (resp. q in f). Rule $\lfloor$P|Sel$\rceil$ is standard selection [15], where the sender process selects one of the branches offered by the receiver.

Rule $\lfloor \text{P}|\text{Tell} \rceil$ establishes a three-way synchronisation, allowing a process to introduce two others. Since the received names are bound at the receivers, we use α-conversion to make the receivers agree on each other's name, as in session types [15]. (Differently from PC, we do not assume the Barendregt convention here, in line with the tradition of process calculi.) Rule $\lfloor \text{P}|\text{Start} \rceil$ requires the name of the created process to be globally fresh.

All other rules are standard. Relation $\to_{\mathscr{B}}$ is closed under a structural precongruence $\preceq_{\mathscr{B}}$, which supports associativity and commutativity of parallel ($|$), standard garbage collection of $\mathbf{0}$, and unfolding of procedure calls.

Example 6. We show a process implementation of the merge sort choreography in Example 1 from § 1. All processes are annotated with type $\mathbf{List}(T)$ (omitted); id is the identity function (Example 2).

```
MSp(p) = if is_small then 0
  else start q1 ▷ (p?id; MSp<q1>; p!*);
       start q2 ▷ (p?id; MSp<q2>; p!*);
       q1!split1; q2!split2; q1?id; q2?merge
```

In the next section, we show that our EPP generates this process implementation automatically from the choreography in Example 1.

4.2 EndPoint Projection (EPP)

We now show how to compile programs in PC to processes in PP.

Behaviour Projection. We start by defining how to project the behaviour of a single process p, a partial function denoted $[\![C]\!]_{\mathsf{p}}$. The rules defining behaviour projection are given in Figure 5. Each choreography term is projected to the local action of the process that we are projecting. For example, a communication term $\mathsf{p}.e$ -> $\mathsf{q}.f$ projects a send action for the sender p, a receive action for the receiver q, or skips to the continuation otherwise. The rules for projecting a selection or an introduction (name mobility) are similar.

The rule for projecting a conditional uses the partial merging operator $\sqcup$: $B \sqcup B'$ is isomorphic to B and B' up to branching, where the branches of B or B' with distinct labels are also included. The interesting rule defining merge is:

$$\begin{aligned}&(\mathsf{p}\&\{l_i : B_i\}_{i\in I}; B) \sqcup \big(\mathsf{p}\&\{l_j : B'_j\}_{j\in J}; B'\big) = \\ &\qquad \mathsf{p}\&\big(\{l_k : (B_k \sqcup B'_k)\}_{k\in I\cap J} \cup \{l_i : B_i\}_{i\in I\setminus J} \cup \{l_j : B'_j\}_{j\in J\setminus I}\big); (B \sqcup B')\end{aligned}$$

The idea of merging comes from [2]. Here, we extend it to general recursion, parametric procedures, and process starts. Merging allows the process that decides a conditional to inform other processes of its choice later on, using selections. It is found repeatedly in most choreography models [2,7,17].

Building on behaviour projection, we define how to project the set $\mathscr{D}$ of procedure definitions. We need to consider two main aspects. The first is that, at runtime, the choreography may invoke a procedure X multiple times, but

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

$$[\![\mathsf{p} : \mathsf{q} \texttt{ <-> } \mathsf{r}; C]\!]_{\mathsf{s}} = \begin{cases} \mathsf{q}!!\mathsf{r}; [\![C]\!]_{\mathsf{s}} & \text{if } \mathsf{s} = \mathsf{p} \\ \mathsf{p}?\mathsf{r}; [\![C]\!]_{\mathsf{s}} & \text{if } \mathsf{s} = \mathsf{q} \\ \mathsf{p}?\mathsf{q}; [\![C]\!]_{\mathsf{s}} & \text{if } \mathsf{s} = \mathsf{r} \\ [\![C]\!]_{\mathsf{s}} & \text{o.w.} \end{cases} \quad [\![X\langle\tilde{\mathsf{p}}\rangle; C]\!]_{\mathsf{r}} = \begin{cases} X_{\mathsf{r}}\langle\tilde{\mathsf{p}}\rangle; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p}_i \\ [\![C]\!]_{\mathsf{r}} & \text{o.w.} \end{cases}$$

$$[\![\mathbf{0}]\!]_{\mathsf{r}} = \mathbf{0} \qquad [\![\mathbf{0}; C]\!]_{\mathsf{r}} = [\![C]\!]_{\mathsf{r}}$$

$$[\![\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C]\!]_{\mathsf{r}} = \begin{cases} \mathsf{if}\ e\ \mathsf{then}\ [\![C_1]\!]_{\mathsf{r}}\ \mathsf{else}\ [\![C_2]\!]_{\mathsf{r}}; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ ([\![C_1]\!]_{\mathsf{r}} \sqcup [\![C_2]\!]_{\mathsf{r}}); [\![C]\!]_{\mathsf{r}} & \text{o.w.} \end{cases}$$

$$[\![\mathsf{p}\ \mathsf{start}\ \mathsf{q}^T; C]\!]_{\mathsf{r}} = \begin{cases} \mathsf{start}\ \mathsf{q} \triangleright [\![C]\!]_{\mathsf{q}}; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C]\!]_{\mathsf{r}} & \text{o.w.} \end{cases}$$

Fig. 5. Procedural Choreographies, Behaviour Projection.

potentially passing a process r at different argument positions each time. This means that r may be called to play different "roles" in the implementation of the procedure. For this reason, we project the behaviour of each possible process parameter p as the local procedure X_{p}. The second aspect is: depending on the role that r is called to play by the choreography, it needs to know the names of the other processes that it is supposed to communicate with in the choreographic procedure. We deal with this by simply passing all arguments (some of which may be unknown to the process invoking the procedure). This is not a problem: for typable choreographies, typing ensures that those parameters are not actually used in the projected procedure (so they act as "dummies"). We do this for clarity, since it yields a simpler formulation of EPP. In practice, we can annotate the EPP by analysing which parameters of each recursive definition are actually used in each of its projections, and instantiating only those.

We thus define $[\![\mathscr{D}]\!] = \bigcup \left\{ [\![X(\widetilde{\mathsf{q}^T}) = C]\!] \mid X(\widetilde{\mathsf{q}^T}) = C \in \mathscr{D} \right\}$ where, for $\widetilde{\mathsf{q}^T} = \mathsf{q}_1^{T_1}, \ldots, \mathsf{q}_n^{T_n}$, we set $[\![X(\widetilde{\mathsf{q}^T}) = C]\!] = \{X_{\mathsf{q}_1}(\tilde{\mathsf{q}}) = [\![C]\!]_{\mathsf{q}_1}, \ldots, X_{\mathsf{q}_n}(\tilde{\mathsf{q}}) = [\![C]\!]_{\mathsf{q}_n}\}$.

Definition 1 (EPP). *Given a procedural choreography $\langle \mathscr{D}, C\rangle$ and a state σ, the EPP $[\![\mathscr{D}, C, \sigma]\!]$ is the parallel composition of the processes in C with all definitions from $\mathscr{D}$: $[\![\mathscr{D}, C, \sigma]\!] = \langle [\![\mathscr{D}]\!], [\![C, \sigma]\!]\rangle = \left\langle [\![\mathscr{D}]\!], \prod_{\mathsf{p} \in \mathsf{pn}(C)} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} [\![C]\!]_{\mathsf{p}} \right\rangle$ where $[\![C, \sigma]\!]$, the EPP of C wrt state σ, is independent of $\mathscr{D}$.*

Since the σs are total, if $[\![C, \sigma]\!]$ is defined for some σ, then $[\![C, \sigma']\!]$ is defined also for all other σ'. When $[\![C, \sigma]\!] = N$ is defined for any σ, we say that C is *projectable* and that N is the projection of C, σ. The same holds for $[\![\mathscr{D}, C, \sigma]\!]$.

Example 7. The EPP of the choreography in Example 1 is given in Example 6.

Example 8. For an example involving merging and introductions, we project the procedure `par_download` (Example 5) for process `s`, omitting type annotations.

```
par_download_s(c,s) = c&{
  more: start s' ▷ (s?c; c?id; c?c'; download_s<c',s'>);
        c!!s'; par_download_s<c,s>
  end: 0                                  }
```

Observe that we invoke procedure `download`$_\mathtt{s}$, since `s'` occurs in the position of `download`'s formal argument `s`.

Properties. EPP guarantees correctness by construction: the code synthesised from a choreography follows it precisely.

Theorem 5 (EPP Theorem). *If $\langle \mathscr{D}, C\rangle$ is projectable, $\Gamma \vdash \mathscr{D}$, and $\Gamma; G \vdash C \triangleright G^*$, then, for all σ: if $G, C, \sigma \to_{\mathscr{D}} G', C', \sigma'$, then $[\![C,\sigma]\!] \to_{[\![\mathscr{D}]\!]}\succ [\![C',\sigma']\!]$ (completeness); and if $[\![C,\sigma]\!] \to_{[\![\mathscr{D}]\!]} N$, then $G, C, \sigma \to_{\mathscr{D}} G', C', \sigma'$ for some G', C' and σ' such that $[\![C',\sigma']\!] \prec N$ (soundness).*

Above, the *pruning relation* $\prec$ from [2] eliminates branches introduced by the merging operator $\sqcup$ when they are not needed anymore to follow the originating choreography ($N \succ N'$ stands for $N' \prec N$). Pruning does not alter reductions, since the eliminated branches are never selected [2]. Combining Theorem 5 with Theorem 1 we get that the projections of typable PC terms never deadlock.

Corollary 1 (Deadlock-freedom by construction). *Let $N = [\![C,\sigma]\!]$ for some C and σ, and assume that $\Gamma; G \vdash C \triangleright G'$ for some Γ such that $\Gamma \vdash \mathscr{D}$ and some G and G'. Then, either: (i) $N \preceq_{[\![\mathscr{D}]\!]} \mathbf{0}$ (N has terminated); or (ii) there exists N' such that $N \to_{[\![\mathscr{D}]\!]} N'$ (N can reduce).*

Remark 4 (Amendment). A choreography can only be unprojectable because of unmergeable subterms, and thus can be made projectable by adding label selections. This can be formalised in an amendment algorithm, similar to [9,18]. For example, the first (unprojectable) choreography in Remark 2 can be amended to the projectable choreography presented at the end of the same remark.

The same argument as in Remark 3 applies: amendment allows us to disregard label selections, but placing them manually can be useful. For example, suppose `p` makes a choice that affects `q` and `r`. If `q` has to perform a slower computation as a result, then it makes sense for `p` to notify `q` first.

5 Related Work and Discussion

Choreographic Programming. Our examples cannot be written in previous models for choreographic programming, which lack full procedural abstraction. In state-of-the-art models [2,4], procedures cannot have continuations, there can only be a limited number of protocols running at any time (modulo dangling asynchronous actions), and the process names used in a procedure are statically determined. In PC, all these limitations are lifted.

Differently from PC, name mobility in choreographies is typically done using channel delegation [4], which is less powerful: a process that introduces two other processes requires a new channel to communicate with them thenceforth.

Some choreography models include explicit parallel composition, $C \mid C'$. Most behaviours of $C \mid C'$ are already captured in PC, for example $X\langle \mathsf{p}, \mathsf{q}\rangle \mid Y\langle \mathsf{r}, \mathsf{s}\rangle$ is equivalent to $X\langle \mathsf{p}, \mathsf{q}\rangle; Y\langle \mathsf{r}, \mathsf{s}\rangle$ in PC (cf. Example 3) – see [4] for a deeper discussion. If a parallel operator is desired, PC can be easily extended (cf. [2]).

In [21], choreographies can be integrated with existing process code by means of a type system, which we could easily integrate in PC.

Asynchrony. Asynchronous communication in choreographic programming was addressed in [4] using an ad-hoc transition rule. Adding asynchrony to PC is straightforward (see the technical report [10]).

Multiparty Session Types (MPST). In MPST [16], global types are choreographic specifications of single protocols, used for verifying the code of manually-written implementations in process models. Global types are similar to a simplified fragment of PC, obtained (among others) by replacing expressions and functions with constants (representing types), removing process creation (the processes are fixed), and restricting recursion to parameterless tail recursion.

MPST leaves protocol composition to the implementors of processes, which can result in deadlocks, unlike in PC. We illustrate this key difference using our syntax; we view a protocol in MPST as a (simplification of a) procedure in PC. Consider the protocols $X(\mathsf{r}, \mathsf{s}) = \mathsf{r}.e \texttt{ -> } \mathsf{s}.f$ and $Y(\mathsf{r}', \mathsf{s}') = \mathsf{r}'.e' \texttt{ -> } \mathsf{s}'.f'$, and their instantiations $X\langle \mathsf{p}, \mathsf{q}\rangle$ and $Y\langle \mathsf{q}, \mathsf{p}\rangle$. In MPST, a valid composition (in PP) is $\mathsf{p} \triangleright_v \mathsf{q}?f'; \mathsf{q}!e \mid \mathsf{q} \triangleright_v \mathsf{p}?f; \mathsf{p}!e'$. This network is obviously deadlocked, but MPST does not detect it because the interleaving of the two protocols is not checked. In PC, we can only obtain correct implementations, because compositions are defined at the level of choreographies, e.g., $X\langle \mathsf{p}, \mathsf{q}\rangle; Y\langle \mathsf{q}, \mathsf{p}\rangle$ or $Y\langle \mathsf{q}, \mathsf{p}\rangle; X\langle \mathsf{p}, \mathsf{q}\rangle$.

Deadlock-freedom for compositions in MPST can be obtained by restricting connections among processes participating in different protocols to form a tree [3,5]. In PC, connections can form an arbitrary graph. Another technique for MPST is to use pre-orders [7], but this is also not as expressive as PC (see [10]).

MPST can be extended to protocols where the number of participants is fixed only at runtime [27], or can grow during execution [13]. These results use ad-hoc primitives and "middleware" terms in the process model, e.g., for tracking the number of participants in a session [13], which are not needed in PC. MPST can be nested [12], partially recalling our parametric procedures. Differently from PC, nested procedures in MPST are invoked by a coordinator (requiring extra communications), and compositions of such nested types can deadlock.

Sessions and Mobility. Recent theories based on session types [2,4,5,7,16] assume that all pairs of processes in a session have a private full-duplex channel to communicate. Thus, processes in a protocol must have a complete connection graph. PC can be used to reason about different kinds of network topologies.

Another important aspect of sessions is that each new protocol execution requires the creation of a new session, whereas procedure calls in PC reuse available connections – allowing for more efficient implementations. Our parallel downloader example uses this feature (Example 5).

The standard results of communication safety found in session-typed calculi can be derived from our EPP Theorem (Theorem 5), as discussed in [4].

Faults. We have abstracted from faults and divergence of internal computations: in PC, we assume that all internal computations terminate successfully. If we relax these conditions, deadlock-freedom can still be achieved simply by using timeouts and propagating faults through communications.

Acknowledgements. We thank the anonymous reviewers for their useful comments. This work was supported by the CRC project, grant no. DFF–4005-00304 from the Danish Council for Independent Research, and by the Open Data Framework project at the University of Southern Denmark.

References

1. Business Process Model and Notation. `http://www.omg.org/spec/BPMN/2.0/`.
2. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
3. M. Carbone, S. Lindley, F. Montesi, C. Schürmann, and P. Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl, 2016.
4. M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274. ACM, 2013.
5. M. Carbone, F. Montesi, C. Schürmann, and N. Yoshida. Multiparty session types as coherence proofs. *Acta Informatica*, 2017.
6. Chor. Programming Language. `http://www.chor-lang.org/`.
7. M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016.
8. L. Cruz-Filipe and F. Montesi. Choreographies in practice. In *FORTE*, volume 9688 of *LNCS*, pages 114–123. Springer, 2016.
9. L. Cruz-Filipe and F. Montesi. A core model for choreographic programming. In *FACS*, LNCS. Springer, 2016. Accepted for publication.
10. L. Cruz-Filipe and F. Montesi. A language for the declarative composition of concurrent protocols. *CoRR*, abs/1602.03729, 2016.
11. M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic choreographies. In *COORDINATION*, LNCS, pages 67–82. Springer, 2015.
12. R. Demangeon and K. Honda. Nested protocols in session types. In *CONCUR*, pages 272–286, 2012.
13. P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446. ACM, 2011.
14. K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT*, pages 55–75. Springer, 2011.

15. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.
16. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. *J. ACM*, 63(1):9, 2016.
17. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *SEFM*, pages 323–332, 2008.
18. I. Lanese, F. Montesi, and G. Zavattaro. Amending choreographies. In *WWV*, pages 34–48, 2013.
19. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *ACM SIGARCH Computer Architecture News*, 36(1):329–339, 2008.
20. F. Montesi. *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen, 2013. http://fabriziomontesi.com/files/choreographic_programming.pdf .
21. F. Montesi and N. Yoshida. Compositional choreographies. In *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.
22. MPI Forum. *MPI: A Message-Passing Interface Standard*. High-Performance Computing Center Stuttgart, 2015. Version 3.1.
23. PI4SOA. http://www.pi4soa.org, 2008.
24. Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In *WWW*, pages 973–982. ACM, 2007.
25. Savara. JBoss Community. `http://www.jboss.org/savara/`.
26. W3C WS-CDL Working Group. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/, 2004.
27. N. Yoshida, P.-M. Deniélou, A. Bejleri, and R. Hu. Parameterised multiparty session types. In *FOSSACS*, volume 6014 of *LNCS*, pages 128–145. Springer, 2010.