Procedural Choreographic Programming

Cruz-Filipe, Luis; Montesi, Fabrizio

Published in:
Formal Techniques for Distributed Objects, Components, and Systems

DOI:
10.1007/978-3-319-60225-7_7

Publication date:
2017

Document version:
Submitted manuscript

Citation for pulished version (APA):
Cruz-Filipe, L., & Montesi, F. (2017). Procedural Choreographic Programming. In A. Bouajjani, & A. Silva (Eds.), *Formal Techniques for Distributed Objects, Components, and Systems: 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017* (pp. 92-107). Springer. https://doi.org/10.1007/978-3-319-60225-7_7

A Language for the Declarative Composition of Concurrent Protocols

Luís Cruz-Filipe and Fabrizio Montesi

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

Abstract. A recent study of bugs in real-world concurrent and distributed systems found that, while implementations of individual protocols tend to be robust, the composition of multiple protocols and its interplay with internal computation is the culprit for most errors.
Multiparty Session Types and Choreographic Programming are methodologies for developing correct-by-construction concurrent and distributed software, based on global descriptions of communication flows. However, protocol composition is either limited or left unchecked. Inspired by these two methodologies, in this work we present a new language model for the safe composition of protocols, called Procedural Choreographies (PC). Protocols in PC are procedures, parameterised on the processes that enact them. Procedures define communications declaratively using global descriptions, and programs are written by invoking and composing these procedures. An implementation in terms of a process model is then mechanically synthesised, guaranteeing correctness and deadlock-freedom. We study PC in the settings of synchronous and asynchronous communications, and illustrate its expressivity with some representative examples.

1 Introduction

In the last decades, advances in multi-core hardware and large-scale networks have made concurrent and distributed systems widespread. Unfortunately, programming such systems is a notoriously error-prone activity. In [22], 105 randomly-selected concurrency bugs from real-world software projects are analysed. Of these, 31 are caused by deadlocks. Of the 74 remaining ones, 97% are caused by violations of the programmer's intentions on atomicity or the ordering of actions.

The theory of Multiparty Session Types tackles these problems in communication protocols [18]. The idea is that developers express their intentions on communications declaratively, by writing protocol specifications from a global viewpoint using an "Alice and Bob" notation. Given such a global specification, an EndPoint Projection (EPP) mechanically synthesises the local specifications of the I/O actions for each participant. The local specifications are thus correct by construction. Then, a type system can be used to check that implementations in process models follow the generated local specifications. Multiparty session types guarantee that the implementation of each protocol, taken in isolation, is deadlock-free and faithful to its global specification – which, by its declarative nature, respects the programmer's intentions. However, errors can still occur

due to the composition of multiple protocol executions, which is left unchecked. Different approaches for avoiding deadlocks in protocol compositions have been proposed [8,9], but these limit how protocols can be composed – e.g., protocols can be instantiated only in a certain order, or connections among processes should form a tree structure – and do not offer a means to easily and correctly translate the programmer's intentions on ordering: the language for composing protocols is not declarative.

A more recent empirical study [21] reveals that protocol composition deserves more attention. It presents a taxonomy of 104 Distributed Concurrency (DC) bugs. The authors' findings include a significant insight, which we quote (emphasis ours):

> "Real-world DC bugs are hard to find because many of them linger in complex concurrent executions of multiple protocols. [...] *Individual protocols tend to be robust in general.* Only 18 DC bugs occur in individual protocols without any input fault condition [...]. *On the other hand, a large majority of DC bugs happen due to concurrent executions of multiple protocols* [...]"

In this quote, a protocol is not intended as an abstract specification as in multiparty session types, but rather as the concrete series of events that happen in an implementation, including internal computation. This motivates us to address the problem of protocol composition in the framework of Choreographic Programming [23]: a paradigm similar to Multiparty Session Types, where global descriptions of communications are not used as types, but rather as programs. In such programs, called choreographies, developers declaratively program communications among processes together with the internal computations that they perform. EPP is then used to synthesise an implementation in terms of a process model [5]. However, state of the art models for choreographic programming still do not support many of the features used in real-world programs, such as those in [21]. One prominent aspect, which is the focus of this paper, is that these models do not support arbitrary compositions of protocol executions, since they are not modular. In particular, the absence of full procedural abstraction disallows the development of reusable libraries that can be composed as "black boxes".

Inspired by these observations, we propose a language model for the correct programming of concurrent and distributed systems based on message passing, called Procedural Choreographies (PC). PC is a new model for choreographic programming, where we can define a protocol execution as a procedure, parameterised on the processes that will actually enact it. Composition of protocol executions is then obtained by allowing for the arbitrary composition of procedure calls, a feature lacked by previous choreography models. Nevertheless, PC inherits all the good properties of languages based on declarative global descriptions of communications – as in choreographic programming and multiparty session types – and extends them to protocol compositions. In particular, the process implementations synthesised from choreographies are correct by construction and deadlock-free. Thus, PC contributes to bringing the current body

of work on safe concurrent programming nearer to dealing with the kinds of bugs analysed in [21].

Example 1. We discuss a parallel version of merge sort in PC. Although this is a toy example, it cannot be written in any previous model for choreographic programming, thus allowing us to present the key features of our work for a simple scenario. More realistic and involved examples are presented after the formal presentation of PC. 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 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]

```
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 $\texttt{p.split}_2\texttt{ -> q2}$). 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`).

Our merge sort example showcases the key desiderata for PC:

General recursion. Procedure calls can be followed by arbitrary code.

Parameterised procedures. Procedures are parametric on their processes (`p` in `MS`), and can thus be reused with different processes (as in `MS<q1>` and `MS<q2>`).

Process spawning. The ability of starting new processes. There must be no bound on how many processes can be started, since this is decided at runtime. (In `MS`, the number of spawned processes depends on the size of the initial list.)

Implicit Parallelism. In the composition `MS<q1>; MS<q2>`, the two calls can be run in parallel because they involve separate processes and are thus non-interfering. Thus, the code synthesised from this program should be parallel.

In the remainder, we explore more sophisticated programs that require additional features, e.g., mobility of process names.

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

1.1 Contributions

We summarise our development and contributions.

Procedural Choreographies. We introduce Procedural Choreographies (PC), a new language model that supports all the features discussed above (§ 2). We evaluate the expressivity of PC not only with our concurrent merge sort example, but also with a more involved parallel downloader. This example makes use of additional features: mobility of process names (networks with connections that evolve at runtime) and propagation of choices among processes. It also makes heavy use of implicit parallelism to deal with the parallelisation of multiple streams.

Asynchrony. PC can be endowed with both a synchronous (§ 2) and an asynchronous semantics (§ 5). All our results hold for both versions, and the two semantics enjoy a strong correspondence result (Theorem 6). This allows developers to reason about their programs in the simpler synchronous setting and then to use the asynchronous semantics to obtain a more concurrent implementation, without worrying about introducing unsafe behaviour.

Typing. Mobility of process names requires careful handling, especially its interplay with procedure composition: if a procedure specifies that two processes interact, then they should be properly connected (know each other's names). We introduce a novel typing discipline (§ 3) that prevents such errors by tracking the connections required by each procedure. It also checks that processes store data of the correct type for the local functions that use it. PC enjoys decidable type checking (Theorem 2) and type inference (Theorems 3 and 4).

Endpoint Projection. We define an EndPoint Projection (EPP) that, given a choreography, synthesises a concurrent implementation in Procedural Processes (PP), our target process calculus (§ 4). PP is an abstraction of systems where concurrent processes communicate by referring to each other's locations or identifiers, as it happens, e.g., in MPI [25] or the Internet Protocol. The synthesised code is correct by construction: it faithfully follows the behaviour of the originating choreography (Theorem 5). Also, EPP is transparent, in the sense that it does not introduce any auxiliary communications or computations. This means that a choreography always faithfully represents the actual efficiency and behaviour of the algorithm written by the programmer. All generated implementations are deadlock-free by construction (Corollary 1).

Extensions. We further discuss two extensions that allow us to write more general procedures, enhancing the expressive power of PC: allowing parameters to be lists of processes and allowing procedure bodies to contain holes that can be filled at runtime with arbitrary code (§ 6). These extensions can be elegantly obtained by introducing minimal additions to the theory of PC, demonstrating its robustness, but they are nevertheless very useful in practice.

$$C ::= \eta; C \mid I; C \mid \mathbf{0} \qquad \eta ::= \mathsf{p}.e \texttt{ -> } \mathsf{q}.f \mid \mathsf{p} \texttt{ -> } \mathsf{q}[l] \mid \mathsf{p}\,\mathsf{start}\,\mathsf{q}^T \mid \mathsf{p} : \mathsf{q} \texttt{ <-> } \mathsf{r}$$
$$\mathscr{D} ::= X(\widetilde{\mathsf{q}^T}) = C, \mathscr{D} \mid \emptyset \qquad I ::= \mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid X\langle\tilde{\mathsf{p}}\rangle \mid \mathbf{0}$$

Fig. 1. Procedural Choreographies, Syntax.

2 Procedural Choreographies (PC)

We begin by introducing the language model of Procedural Choreographies (PC). We focus on the synchronous semantics in this section, as the underlying theory is a bit simpler. The asynchronous model is discussed in § 5.

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 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*. This distinction plays a bigger role in § 5, which introduces more runtime terms to track the state of asynchronous communications.

Processes communicate via 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 to q. In e, the placeholder $*$ is replaced at runtime with the data stored at process p. 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 mobility of process names. In real-world systems, after execution of $\mathsf{p}\,\mathsf{start}\,\mathsf{q}^T$, p is the only process that knows the

[2] This makes PC easy to apply to mainstream settings based on actors, objects, or ranks (e.g., MPI). We could give a formulation of PC based on standard channels (from process calculi), where each pair of process names is a channel. This is evident also from the connection graph used in the semantics of PC, defined below.

name of q. Any other process wanting to communicate with q must therefore be first informed of its existence (as happens, e.g., in object- and service-oriented computing [13,16]). This is achieved with the introduction term $\mathsf{p} : \mathsf{q} \texttt{<->} \mathsf{r}$, 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 will become 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 (or invokes) procedure X by passing $\tilde{\mathsf{p}}$ as parameters. Procedure calls inside definitions must be guarded, i.e., they can only occur after a communication action.

We work up to α-equivalence in choreographies, assuming the Barendregt convention. Bound variables are renamed 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 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 ($\mathsf{p} : \mathsf{q} \texttt{<->} \mathsf{r}$) requires a three-way synchronization, and essentially performs two communications. The alternative development of PC with asymmetric introduction (an action $\mathsf{p} : \mathsf{q}$ `->` r whereby p sends q's name to r, but not conversely) is essentially the same as ours. 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 briefly motivate the need for selections (p `->` $\mathsf{q}[l]$). Consider the choreography $\mathsf{if}\ \mathsf{p}.\mathsf{coinflip}\ \mathsf{then}\ (\mathsf{p}.* \texttt{->} \mathsf{r})\ \mathsf{else}\ (\mathsf{r}.* \texttt{->} \mathsf{p})$. 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

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

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

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

$$\frac{\mathsf{p} \xleftrightarrow{G} \mathsf{q} \quad \mathsf{p} \xleftrightarrow{G} \mathsf{r}}{G, \mathsf{p} : \mathsf{q} \texttt{ <-> } \mathsf{r}; C, \sigma \;\rightarrow_{\mathscr{D}}\; G \cup \{\mathsf{q} \leftrightarrow \mathsf{r}\}, C, \sigma} \;\lfloor \text{C}|\text{Tell} \rceil$$

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

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

Fig. 2. Procedural Choreographies, Semantics.

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:

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

Now r receives a label reflecting the choice made by p, and can decide what to do.

This intuition is formalised by the definition of EndPoint Projection in § 4. The first choreography above is not projectable, whereas the second one is. See also Example 5 at the end of this section.

Semantics. We define a reduction semantics $\rightarrow_{\mathscr{D}}$ for PC, parameterised over $\mathscr{D}$. We model the state of processes with a (total) state function σ, where $\sigma(\mathsf{p})$ denotes the value stored in p. We assume that each type $T \in \mathbb{T}$ has a special value $\bot_T$, representing an uninitialised process state. The semantics of PC also includes 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 $\lfloor$C|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.

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

$$\frac{\mathsf{pn}(\eta) \cap \mathsf{pn}(\eta') = \emptyset}{\eta;\eta' \ \equiv_{\mathscr{D}} \ \eta';\eta} \ \lfloor \text{C|Eta-Eta} \rceil \quad \frac{\mathsf{pn}(I) \cap \mathsf{pn}(I') = \emptyset}{I;I' \ \equiv_{\mathscr{D}} \ I';I} \ \lfloor \text{C|I-I} \rceil \quad \frac{\mathsf{pn}(I) \cap \mathsf{pn}(\eta) = \emptyset}{\eta;I \ \equiv_{\mathscr{D}} \ I;\eta} \ \lfloor \text{C|I-Eta} \rceil$$

$$\frac{}{\mathbf{0};C \preceq_{\mathscr{D}} C} \ \lfloor \text{C|End} \rceil \quad \frac{X(\widetilde{\mathsf{q}^T}) = C_X \in \mathscr{D}}{X\langle \tilde{\mathsf{p}} \rangle;C \ \preceq_{\mathscr{D}} \ C_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \fatsemi C} \ \lfloor \text{C|Unfold} \rceil$$

$$\frac{\{\mathsf{p},\mathsf{q}\} \cap \mathsf{pn}(\eta) = \emptyset}{\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ (\eta;C_1)\ \mathsf{else}\ (\eta;C_2) \ \equiv_{\mathscr{D}} \ \eta;\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2} \ \lfloor \text{C|Eta-Cond} \rceil$$

$$\frac{}{\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ (C_1;\eta)\ \mathsf{else}\ (C_2;\eta) \ \equiv_{\mathscr{D}} \ \mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2;\eta} \ \lfloor \text{C|Cond-Eta} \rceil$$

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

Fig. 3. Procedural Choreographies, Structural precongruence $\preceq$.

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. Rule $\lfloor$C|Tell$\rceil$ captures name mobility, by creating a connection between two processes q and r when they are introduced by a process p that is 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. Operator $\fatsemi$ is defined by $\eta \fatsemi C = \eta;C$, $I \fatsemi C = I;C$ and $(C_1;C_2) \fatsemi C = C_1;(C_2 \fatsemi C)$. This operator 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$\rceil$ makes use of the structural precongruence $\preceq_{\mathscr{D}}$, which is defined by the rules in Figure 3. We write $C \equiv_{\mathscr{D}} C'$ when $C \preceq_{\mathscr{D}} C'$ and $C' \preceq_{\mathscr{D}} C$, and denote the set of process names (free or bound) in a choreography C by $\mathsf{pn}(C)$. Rule $\lfloor$C|Unfold$\rceil$ unfolds a procedure call, again using the $\fatsemi$ operator defined above, and rule $\lfloor$C|End$\rceil$ is garbage collection of $\mathbf{0}$.

The other rules formalise the notion of implicit parallelism anticipated in § 1. 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, two procedure calls can be swapped if they share no arguments. This is sound because a procedure can only define actions for processes that are either passed as arguments or started inside of the procedure itself, 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. The other rules follow similar intuitions; we omit rules ⌊C|I-Cond⌉ and ⌊C|Cond-I⌉, analogous to ⌊C|Eta-Cond⌉ and ⌊C|Cond-Eta⌉.

Example 3. In our merge sort example, rule ⌊C|I-I⌉ allows the recursive calls `MS<q1>` and `MS<q2>` to be exchanged. Furthermore, after the calls are unfolded, implicit parallelism allows their code to be interleaved in any way.

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

Example 4. A more refined example of implicit parallelism involves swapping 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 ⌊C|Eta-Eta⌉ 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`. The key idea is to download all files in parallel via streaming, by having the client and the server each create subprocesses to handle the transfer of each file. This allows the client to 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, $\mathtt{c}'$ and $\mathtt{s}'$ respectively, and the server introduces $\mathtt{c}$ to $\mathtt{s}'$ (`s: c <-> s'`). The client $\mathtt{c}$ sends to $\mathtt{s}'$ 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 $\mathtt{c}'$ to $\mathtt{s}'$. The file download is handled by $\mathtt{c}'$ and $\mathtt{s}'$ (using procedure `download`), while $\mathtt{c}$ and $\mathtt{s}$ continue operating (`par_download<c,s>`). Finally, $\mathtt{c}'$ waits until $\mathtt{c}$ 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 $\mathtt{s}$ to another process $\mathtt{c}$.

```
download(c,s) = if s.more
  then s -> c [more]; s.next -> c.append; 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), thus they can be fully parallelised by our semantics: many instances of `download` run at the same time, each one implementing a (sequential) stream. By implicit parallelism, 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 $\mathtt{a}$. Below, we use the shortcut $\mathtt{p} \texttt{->} \tilde{\mathtt{q}}[l]$ for $\mathtt{p} \texttt{->} \mathtt{q}_1[l]; \ldots; \mathtt{p} \texttt{->} \mathtt{q}_n[l]$.

```
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 checks 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 the typings in Γ, and when executed from a connection graph that contains G it produces a connection graph that includes G'". Typing environments Γ are used to track the types of processes and procedures; they are defined as: $\Gamma ::= \emptyset \mid \Gamma, \mathtt{p} : T \mid \Gamma, X : G \triangleright G'$. A typing $\mathtt{p} : T$ states that process $\mathtt{p}$ stores values of type T, and a typing $X : G \triangleright G'$ records the effect of the body of X on graph G.

The rules for deriving typing judgements are given in Figure 4. 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".

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

$$\frac{\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q} \quad \Gamma \vdash \mathsf{p} : T_{\mathsf{p}}, \mathsf{q} : T_{\mathsf{q}} \quad * : T_{\mathsf{p}} \vdash_{\mathbb{T}} e : T_1 \quad * : T_{\mathsf{q}} \vdash_{\mathbb{T}} f : T_1 \to T_{\mathsf{q}} \quad \Gamma; G \vdash C \triangleright G'}{\Gamma; G \vdash \mathsf{p}.e \texttt{ -> } \mathsf{q}.f; C \triangleright G'}\ \lfloor \mathrm{T}|\mathrm{Com} \rceil$$

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

$$\frac{\Gamma, \mathsf{q} : T; G \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\} \vdash C \triangleright G'}{\Gamma; G \vdash \mathsf{p}\ \mathsf{start}\ \mathsf{q}^T; C \triangleright G'}\ \lfloor \mathrm{T}|\mathrm{Start} \rceil \qquad \frac{\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q} \quad \mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{r} \quad \Gamma; G \cup \{\mathsf{q} \leftrightarrow \mathsf{r}\} \vdash C \triangleright G'}{\Gamma; G \vdash \mathsf{p} : \mathsf{q}\ \texttt{<->}\ \mathsf{r}; C \triangleright G'}\ \lfloor \mathrm{T}|\mathrm{Tell} \rceil$$

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

Fig. 4. Procedural Choreographies, Typing Rules.

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 $\lfloor \mathrm{T}|\mathrm{Start} \rceil$, we implicitly use the fact that q does not appear yet in G, which is another consequence of using the Barendregt convention. The final graph G' is only used in procedure calls (rule $\lfloor \mathrm{T}|\mathrm{Call} \rceil$). 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]

Theorem 1 (Deadlock freedom and Subject reduction). *Given a choreography C and a set $\mathscr{D}$ of procedure definitions, if $\Gamma \vdash \mathscr{D}$ and $\Gamma; G_1 \vdash C \triangleright G'_1$ for some Γ, G_1 and G'_1, then either:*

- *$C \preceq_{\mathscr{D}} \mathbf{0}$; or,*
- *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.*

(Proofs of theorems can be found in the Appendix.)

[3] Since we are interested in communications, we assume evaluation of functions and expressions to terminate on values with the right types (see § 7, Faults).

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 $\langle \mathscr{D}, C \rangle$, entailing type inference properties for PC.

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

Theorem 2 may seem a bit surprising; the key idea of its proof is that type-checking may require expanding recursive definitions, but their parameters only need to be instantiated with process names from a finite set. With a similar idea, we also obtain type inference.

Theorem 3. *There is an algorithm that, given any $\langle \mathscr{D}, C \rangle$, outputs:*

- *a set Γ such that $\Gamma \vdash \langle \mathscr{D}, C \rangle$, if such a Γ exists;*
- `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). Theorems 3 and 4 allow us to omit type annotations in choreographies, if the types of functions and expressions at processes are known (from $\vdash_{\mathbb{T}}$). Thus, programmers can write choreographies as in our examples.

The same reasoning could be adopted to infer missing introductions (`p : q <-> r`) in a choreography automatically, thus lifting the programmer also from having to think about connections entirely. 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).

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).

Syntax. The syntax of PP is given in Figure 5. 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

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

Fig. 5. Procedural Processes, Syntax.

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

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

$$\frac{i = 1 \text{ if } e[v/*] = \mathsf{true}, \quad i = 2 \text{ otherwise}}{\mathsf{p} \triangleright_v \mathsf{if}\ e\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2 \ \rightarrow_{\mathscr{B}}\ \mathsf{p} \triangleright_v B_i} \ \lfloor \text{P|Cond} \rceil$$

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

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

$$\frac{N \ \rightarrow_{\mathscr{B}}\ N'}{N \mid M \ \rightarrow_{\mathscr{B}}\ N' \mid M} \ \lfloor \text{P|Par} \rceil \qquad \frac{N \preceq_{\mathscr{B}} M \quad M \ \rightarrow_{\mathscr{B}}\ M' \quad M' \preceq_{\mathscr{B}} N'}{N \ \rightarrow_{\mathscr{B}}\ N'} \ \lfloor \text{P|Struct} \rceil$$

Fig. 6. Procedural Processes, Semantics.

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 according to 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 rules defining the reduction relation $\rightarrow_{\mathscr{B}}$ for PP are shown in Figure 6. 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|Tell$\rceil$ establishes a three-way synchronisation,

$$\frac{}{\mathsf{p} \triangleright_v \mathbf{0} \ \preceq_{\mathscr{B}} \ \mathbf{0}} \ \lfloor \text{P|AZero} \rceil \qquad \frac{}{N \mid \mathbf{0} \ \preceq_{\mathscr{B}} \ N} \ \lfloor \text{P|NZero} \rceil$$

$$\frac{}{\mathbf{0}; B \ \preceq_{\mathscr{B}} \ B} \ \lfloor \text{P|End} \rceil \qquad \frac{X(\widetilde{\mathsf{q}^T}) = B_X \in \mathscr{B}}{X\langle \tilde{\mathsf{p}} \rangle; B \preceq_{\mathscr{B}} B_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \fatsemi B} \ \lfloor \text{P|Unfold} \rceil$$

Fig. 7. Procedural Processes, Structural precongruence $\preceq_{\mathscr{B}}$.

allowing a process to introduce two others. Since the received names are bound at the receivers, we rely on α-conversion to make the receivers agree on each other's name, as done in session types [17]. (Differently from PC, we do not assume the Barendregt convention here, in line with the tradition of process calculi.) Rule $\lfloor$P|Sel$\rceil$ is standard selection [17], where the sender process selects one of the branches offered by the receiver. In rule $\lfloor$P|Start$\rceil$, we require the name of the created process to be globally fresh. All other rules are standard. Rule $\lfloor$P|Struct$\rceil$ uses structural precongruence $\preceq_{\mathscr{B}}$, which is the smallest precongruence satisfying associativity and commutativity of parallel ($\mid$) and the rules in Figure 7. Rule $\lfloor$P|Unfold$\rceil$ expands procedure calls. It uses again the $\fatsemi$ operator, defined as for PC but with terms in the PP language.

Remark 4. Our three-way synchronisation in rule $\lfloor$P|Tell$\rceil$ could be easily encoded with two standard two-way communications of names, as in the π-calculus [27] (see also Remark 1). Our choice gives a clearer formulation of EPP.

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).

```
MS_p(p) = if is_small then 0
  else start q1 ▷ (p?id; MS_p<q1>; p!*);
       start q2 ▷ (p?id; MS_p<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 8. 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.

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

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

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

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

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

Fig. 8. Procedural Choreographies, Behaviour Projection.

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:

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

The idea of merging comes from [5]. Here, we extend it to general recursion, parametric procedures, and process starts. The complete definition of merging is given in the Appendix. 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 [5,9,19].

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 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, which means that some of them may even be unknown to the process invoking the procedure. This is not a problem: we focus on typable choreographies, and 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 (see Appendix). We can now define

$$[\![\mathscr{D}]\!] = \bigcup \left\{ [\![X(\widetilde{\mathsf{q}^T}) = C]\!] \mid X(\widetilde{\mathsf{q}^T}) = C \in \mathscr{D} \right\}$$

where, for $\widetilde{\mathsf{q}^T} = \mathsf{q}_1^{T_1}, \ldots, \mathsf{q}_n^{T_n}$,

$$[\![X(\widetilde{\mathsf{q}^T}) = C]\!] \;=\; \{X_1(\tilde{\mathsf{q}}) = [\![C]\!]_{\mathsf{q}_1}, \ldots, X_n(\tilde{\mathsf{q}}) = [\![C]\!]_{\mathsf{q}_n}\} \; .$$

Definition 1 (EPP). *Given a procedural choreography $\langle \mathscr{D}, C \rangle$ and a state σ, the endpoint projection $[\![\mathscr{D}, C, \sigma]\!]$ is the parallel composition of the processes in C with all definitions from $\mathscr{D}$:*

$$[\![\mathscr{D}, C, \sigma]\!] \;=\; \langle [\![\mathscr{D}]\!], [\![C, \sigma]\!] \rangle \;=\; \left\langle [\![\mathscr{D}]\!], \textstyle\prod_{\mathsf{p} \in \mathsf{pn}(C)} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} [\![C]\!]_{\mathsf{p}} \right\rangle$$

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. Similar considerations apply to $[\![\mathscr{D}, C, \sigma]\!]$.

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

Example 8. We give a more sophisticated example involving merging and introductions: the projection of procedure `par_download` (Example 5) for process `s`. (We omit the 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 call procedure $\texttt{download}_\texttt{s}$, as `s'` occurs in the position of that procedure'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:*

- *(Completeness) if $G, C, \sigma \to_{\mathscr{D}} G', C', \sigma'$, then $[\![C, \sigma]\!] \to_{[\![\mathscr{D}]\!]} \succ [\![C', \sigma']\!]$;*
- *(Soundness) if $[\![C, \sigma]\!] \to_{[\![\mathscr{D}]\!]} N$, then $G, C, \sigma \to_{\mathscr{D}} G', C', \sigma'$ for some G', σ' such that $[\![C', \sigma']\!] \prec N$.*

Above, the *pruning relation* $\prec$ from [5] eliminates the branches introduced by the merging operator $\sqcup$ when they are not needed anymore to follow the originating choreography (we write $N \succ N'$ when $N' \prec N$). Pruning does not alter reductions, since the eliminated branches are never selected [5]. 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:*

- *$N \preceq_{[\![\mathscr{D}]\!]} \mathbf{0}$ (N has terminated);*
- *or there exists N' such that $N \rightarrow_{[\![\mathscr{D}]\!]} N'$ (N can reduce).*

Remark 5 (Amendment). A choreography in PC can only be unprojectable because of unmergeable subterms. Thus, every choreography can be made projectable by only adding label selections. This can be formalized in an amendment algorithm, as in other choreography languages [20,11], reported in the Appendix. For example, the first (unprojectable) choreography in Remark 2 can be amended to the projectable choreography presented at the end of the same remark.

In practice, the same argument as for inferring introduction terms (Remark 3) applies. Although amendment allows us to write choreographies without worrying about label selections, it is useful to give the programmer the option to place them where most convenient. For example, consider a process p making an internal choice that affects processes q and r. If one of these two processes has to perform a slower computation in response to that choice, then it makes sense for p to send a label selection to it first, and only afterwards to notify the other process.

5 Asynchrony

We define an alternative semantics to PC and PP, whereby communication becomes asynchronous. We show that the results stated in the previous sections also hold for the asynchronous case.

5.1 Asynchronous PC (aPC)

Syntax. In asynchronous PC (aPC), communications are not atomically executed anymore, but consist of multiple actions. In particular, the sender of a message can proceed in its execution without waiting for the receiver to receive such message.

The intuition behind aPC is that the language is extended with new runtime terms that capture this refinement of execution steps.[4] At runtime communications are expanded into multiple actions. For example, a communication $\mathsf{p}.e \twoheadrightarrow \mathsf{q}.f$ expands in $\mathsf{p}.e \overset{x}{\twoheadrightarrow} \bullet_{\mathsf{q}}$ – a send action from p – and $\bullet_{\mathsf{p}} \overset{x}{\twoheadrightarrow} \mathsf{q}.f$ – a receive action by q. The process subscripts at $\bullet$ are immaterial for the semantics of aPC, but are useful for the definition of EPP. The tag x specifies that, in the original choreography, that message from p should reach that receive action at q. Executing $\mathsf{p}.e \overset{x}{\twoheadrightarrow} \bullet_{\mathsf{q}}$ replaces x in the corresponding receive action with the

[4] Recall that runtime terms are assumed never to be used by programmers (like term $\mathbf{0}; C$ in PC). They are used only to represent runtime states.

$$\eta ::= \ldots \mid \mathsf{p}.e \xrightarrow{x} \bullet_{\mathsf{q}} \mid \bullet_{\mathsf{p}} \xrightarrow{\hat{v}} \mathsf{q}.f \mid \mathsf{p} \xrightarrow{x} \bullet_{\mathsf{q}}[l] \mid \bullet_{\mathsf{p}} \xrightarrow{\hat{l}} \mathsf{q}[l] \mid \mathsf{p} : \bullet_{\mathsf{q}} \xleftrightarrow{x,y} \bullet_{\mathsf{r}} \mid \bullet_{\mathsf{p}}.\mathsf{r} \xrightarrow{\hat{\mathsf{p}}} \mathsf{q}$$

$$\hat{v} ::= x \mid v \qquad \hat{l} ::= x \mid l \qquad \hat{\mathsf{p}} ::= x \mid \mathsf{p}$$

Fig. 9. Asynchronous PC, Syntax of New Runtime Terms.

actual value v computed from e at p, yielding $\bullet_{\mathsf{p}} \xrightarrow{v} \mathsf{q}.f$; executing $\bullet_{\mathsf{p}} \xrightarrow{v} \mathsf{q}.f$ updates the state of q.

We now define aPC formally. The new terms are given in Figure 9, and they are all runtime terms. The terms follow the intuition given for value communications, extended to selection and introduction. Label selection expands in $\mathsf{p} \xrightarrow{x} \bullet_{\mathsf{q}}[l]$ and $\bullet_{\mathsf{p}} \xrightarrow{\hat{l}} \mathsf{q}[l]$, and introduction expands in $\mathsf{p} : \bullet_{\mathsf{q}} \xleftrightarrow{x,y} \bullet_{\mathsf{r}}$ and two $\bullet_{\mathsf{p}}.\mathsf{r} \xrightarrow{\hat{\mathsf{p}}} \mathsf{q}$ actions. The tags in these actions are not essential for propagating values as above, but they make the treatment of all new actions similar. Process names for the new runtime terms are defined as follows, ignoring process names in tags and in $\bullet$ subscripts.

$$\mathsf{pn}(\mathsf{p}.e \xrightarrow{x} \bullet_{\mathsf{q}}) = \mathsf{pn}(\mathsf{p} \xrightarrow{x} \bullet_{\mathsf{q}}[l]) = \mathsf{pn}(\mathsf{p} : \bullet_{\mathsf{q}} \xleftrightarrow{x,y} \bullet_{\mathsf{r}}) = \{\mathsf{p}\}$$

$$\mathsf{pn}(\bullet_{\mathsf{p}} \xrightarrow{\hat{v}} \mathsf{q}.f) = \mathsf{pn}(\bullet_{\mathsf{p}} \xrightarrow{\hat{l}} \mathsf{q}[l]) = \mathsf{pn}(\bullet_{\mathsf{p}}.\mathsf{r} \xrightarrow{\hat{\mathsf{p}}} \mathsf{q}) = \{\mathsf{q}\}$$

Semantics. The interplay between asynchronous communications and name mobility requires the connection graph to be directed in aPC, since the processes q and r in an introduction term $\mathsf{p} : \mathsf{q} \leftrightarrow \mathsf{r}$ may receive names at different times now. An edge from p to q in G, denoted $\mathsf{q} \xrightarrow{G} \mathsf{p}$, now means that p knows q's name – so p is able to send messages to q or to listen for incoming messages from q.

The semantics for aPC includes:

- the rules from Figure 10, which are the asynchronous counterpart to $\lfloor$C|Com$\rceil$ in PC, and similar rules for selection and introduction;
- rules $\lfloor$C|Cond$\rceil$ and $\lfloor$C|Struct$\rceil$ from PC;
- rule $\lfloor$C|Start$\rceil$ from PC, where $G \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\}$ is the graph obtained by adding the two directed edges between p and q to G;
- the rules defining $\preceq_{\mathscr{D}}$ (Figure 3), with η now ranging over all communication actions in aPC;
- the new rules for $\preceq_{\mathscr{D}}$ in Figure 11.

The missing new rules are given in the Appendix. By the Barendregt convention, tags introduced by the rules in Figure 11 are globally fresh. Thus, they maintain the correspondence between the value being sent and that being received.

The key to the semantics of aPC lies in the new swaps allowed by $\preceq_{\mathscr{D}}$, due to the definition of pn for the new terms. This captures the concurrency that arises from asynchronous communications.

$$\frac{\mathsf{p} \xrightarrow{G} \mathsf{q} \quad e[\sigma(\mathsf{p})/*] \downarrow v}{G, \mathsf{p}.e \xrightarrow{x} \bullet_{\mathsf{q}}; C, \sigma \;\to_{\mathscr{D}}\; G, C[v/x], \sigma} \lfloor \text{C}|\text{Com-S} \rceil$$

$$\frac{\mathsf{q} \xrightarrow{G} \mathsf{p} \quad f[\sigma(\mathsf{q})/*](v) \downarrow w}{G, \bullet_{\mathsf{p}} \xrightarrow{v} \mathsf{q}.f; C, \sigma \;\to_{\mathscr{D}}\; G, C, \sigma[\mathsf{q} \mapsto w]} \lfloor \text{C}|\text{Com-R} \rceil$$

Fig. 10. Asynchronous PC, Semantics of New Runtime Terms.

$$\frac{}{\mathsf{p}.e \to \mathsf{q}.f \;\preceq_{\mathscr{D}}\; \mathsf{p}.e \xrightarrow{x} \bullet_{\mathsf{q}}; \bullet_{\mathsf{p}} \xrightarrow{x} \mathsf{q}.f} \lfloor \text{C}|\text{Com-U} \rceil$$

$$\frac{}{\mathsf{p} \to \mathsf{q}[l] \;\preceq_{\mathscr{D}}\; \mathsf{p} \xrightarrow{x} \bullet_{\mathsf{q}}[l]; \bullet_{\mathsf{p}} \xrightarrow{x} \mathsf{q}[l]} \lfloor \text{C}|\text{Sel-U} \rceil$$

$$\frac{}{\mathsf{p} : \mathsf{q} \leftrightarrow \mathsf{r} \;\preceq_{\mathscr{D}}\; \mathsf{p} : \bullet_{\mathsf{q}} \overset{x,y}{\leftrightarrow} \bullet_{\mathsf{r}}; \bullet_{\mathsf{p}}.\mathsf{r} \xrightarrow{x} \mathsf{q}; \bullet_{\mathsf{p}}.\mathsf{q} \xrightarrow{y} \mathsf{r}} \lfloor \text{C}|\text{Tell-U} \rceil$$

Fig. 11. Asynchronous PC, Structural Precongruence (New Additional Rules).

Example 9. Let C be $\mathsf{p}.e \to \mathsf{q}.f; \mathsf{p}.e' \to \mathsf{r}.f'$. To execute C in aPC, we expand it:

$$C \quad \preceq_{\mathscr{D}} \quad \mathsf{p}.e \xrightarrow{x} \bullet_{\mathsf{q}};\ \bullet_{\mathsf{p}} \xrightarrow{x} \mathsf{q}.f';\ \mathsf{p}.e' \xrightarrow{y} \bullet_{\mathsf{r}};\ \bullet_{\mathsf{p}} \xrightarrow{y} \mathsf{r}.f'$$

Using $\preceq_{\mathscr{D}}$, we can swap the second term to the end (it shares no process names with the subsequent terms according to pn):

$$C \quad \preceq_{\mathscr{D}} \quad \mathsf{p}.e \xrightarrow{x} \bullet_{\mathsf{q}};\ \mathsf{p}.e' \xrightarrow{y} \bullet_{\mathsf{r}};\ \bullet_{\mathsf{p}} \xrightarrow{y} \mathsf{r}.f';\ \bullet_{\mathsf{p}} \xrightarrow{x} \mathsf{q}.f'$$

Now p can send both messages immediately, and r can receive its message before q.

Example 10. Due to asynchrony, a process q can now send a message to another process r that does not yet know about q. However, r is still unable to receive it before learning q's name, as expected [27]. Assume that $\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q}$ and $\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{r}$ and consider the choreography: $C \overset{\Delta}{=} \mathsf{p} : \mathsf{q} \leftrightarrow \mathsf{r};\ \mathsf{q}.* \to \mathsf{r}.f$. We can unfold C to

$$C \quad \preceq_{\mathscr{D}} \quad \mathsf{p} : \bullet_{\mathsf{q}} \overset{x,y}{\leftrightarrow} \bullet_{\mathsf{r}};\ \bullet_{\mathsf{p}}.\mathsf{r} \xrightarrow{x} \mathsf{q};\ \bullet_{\mathsf{p}}.\mathsf{q} \xrightarrow{y} \mathsf{r};\ \mathsf{q}.* \xrightarrow{z} \bullet_{\mathsf{r}};\ \bullet_{\mathsf{q}} \xrightarrow{x} \mathsf{r}.f$$

and, by $\preceq_{\mathscr{D}}$, swap the third and fourth term

$$C \quad \preceq_{\mathscr{D}} \quad \mathsf{p} : \bullet_{\mathsf{q}} \overset{x,y}{\leftrightarrow} \bullet_{\mathsf{r}};\ \bullet_{\mathsf{p}}.\mathsf{r} \xrightarrow{x} \mathsf{q};\ \mathsf{q}.* \xrightarrow{z} \bullet_{\mathsf{r}};\ \bullet_{\mathsf{p}}.\mathsf{q} \xrightarrow{y} \mathsf{r};\ \bullet_{\mathsf{q}} \xrightarrow{x} \mathsf{r}.f$$

which corresponds to an execution path where q sends its contents to r before r is notified of q's name. The last two actions can not be swapped, so r cannot access q's message before receiving its name.

$$\frac{\mathsf{p} \xrightarrow{G} \mathsf{q} \quad \Gamma \vdash \mathsf{p} : T_{\mathsf{p}} \quad * : T_{\mathsf{p}} \vdash_{\mathbb{T}} e : T \quad \Gamma \oplus (x : T); G \vdash C \triangleright G'}{\Gamma; G \vdash \mathsf{p}.e \overset{x}{\text{->}} \bullet_{\mathsf{q}}; C \triangleright G'} \lfloor \text{T|Com-S} \rceil$$

$$\frac{\mathsf{q} \xrightarrow{G} \mathsf{p} \quad \Gamma \vdash \mathsf{q} : T_{\mathsf{q}} \quad * : T_{\mathsf{q}} \vdash_{\mathbb{T}} f : T \to T_{\mathsf{q}} \quad \Gamma \oplus (x : T); G \vdash C \triangleright G'}{\Gamma; G \vdash \bullet_{\mathsf{p}} \overset{x}{\text{->}} \mathsf{q}.f; C \triangleright G'} \lfloor \text{T|Com-RV} \rceil$$

$$\frac{\mathsf{q} \xrightarrow{G} \mathsf{p} \quad \Gamma \vdash \mathsf{q} : T_{\mathsf{q}} \quad \vdash_{\mathbb{T}} v : T \quad * : T_{\mathsf{q}} \vdash_{\mathbb{T}} f : T \to T_{\mathsf{q}} \quad \Gamma; G \vdash C \triangleright G'}{\Gamma; G \vdash \bullet_{\mathsf{p}} \overset{v}{\text{->}} \mathsf{q}.f; C \triangleright G'} \lfloor \text{T|Com-RT} \rceil$$

Fig. 12. Asynchronous PC, Typing Rules (New Runtime Terms).

To restate Theorems 1–4 for aPC, we extend our type system to the new runtime terms. In particular, we augment contexts Γ to contain also type declarations for tags ($x : T$) and assignments of labels or process identifiers to tags ($x = l$ or $x = \mathsf{p}$). The type system of aPC contains the rules for PC (Figure 4) together with the new rules given in Figure 12 and analogous ones for selections and introductions. Furthermore, in rule $\lfloor$T|End$\rceil$ (from PC) we require that Γ does not contain any assertions $x : T$ or $x = t$, meaning that there should be no dangling actions.

The context $\Gamma \oplus (x : T)$ used in the rules is defined below; the interpretation of $\Gamma \oplus (x = t)$ is similar. The operator $\oplus$ is partial. In particular, any judgement involving $\Gamma \oplus (x : T)$ is false if Γ already declares x with a different type.

$$\Gamma \oplus (x : T) = \begin{cases} \Gamma, x : T & \text{if } x \text{ is not declared in } \Gamma \\ \Gamma \setminus \{x : T\} & \text{if } x : T \in \Gamma \\ \text{undefined} & \text{otherwise} \end{cases}$$

Thus, typing a receiving action depends on whether or not it corresponds to a message that has already been sent according to Γ.

If we consider only program choreographies (without runtime terms), then the typing systems for aPC and PC coincide (as the connection graph is always symmetric); in particular, Theorems 2–4 automatically hold for program choreographies in aPC. The proofs of these results, as well as that of Theorem 1, can be readily adapted to aPC, as the new cases do not pose any difficulty.

PC and aPC enjoy the following operational correspondence.

Theorem 6. *For any C in PC, connection graph G, and state σ:*

- *If $G, C, \sigma \to_{\mathscr{D}} G', C', \sigma'$ (in PC), then $G, C, \sigma \to^*_{\mathscr{D}} G', C', \sigma'$ (in aPC).*
- *If $G, C, \sigma \to_{\mathscr{D}} G', C', \sigma'$ (in aPC), then there exist C'', G'' and σ'' such that $G, C, \sigma \to_{\mathscr{D}} G'', C'', \sigma''$ (in PC) and $G', C', \sigma' \to^*_{\mathscr{D}} G'', C'', \sigma''$ (in aPC).*

In particular, the sequence of messages from a process p to another process q is received by q in exactly the same order as it is sent by p. Thus, our semantics

$$\frac{\rho'_{\mathsf{q}} = \rho_{\mathsf{q}} \cdot \langle \mathsf{p}, e[v/*] \rangle}{\mathsf{p} \triangleright_v^{\rho_{\mathsf{p}}} \mathsf{q}!e; B_{\mathsf{p}} \mid \mathsf{q} \triangleright_w^{\rho_{\mathsf{q}}} B_{\mathsf{q}} \;\to_{\mathscr{B}}\; \mathsf{p} \triangleright_v^{\rho_{\mathsf{p}}} B_{\mathsf{p}} \mid \mathsf{q} \triangleright_w^{\rho'_{\mathsf{q}}} B_{\mathsf{q}}} \;\lfloor \text{P|Com-S} \rceil$$

$$\frac{\rho_{\mathsf{q}} \preceq \langle \mathsf{p}, v \rangle \cdot \rho'_{\mathsf{q}} \quad u = (f[w/*])(v)}{\mathsf{q} \triangleright_w^{\rho_{\mathsf{q}}} \mathsf{p}?f; B \;\to_{\mathscr{B}}\; \mathsf{q} \triangleright_u^{\rho'_{\mathsf{q}}} B} \;\lfloor \text{P|Com-R} \rceil$$

Fig. 13. Asynchronous Procedural Processes, Semantics (New Rules).

matches an interpretation of asynchrony supported by queues between processes, as we make evident in the asynchronous version of PP below.

5.2 Asynchronous PP (aPP)

Syntax. The asynchronous variant of PP, aPP, is easier to define, as the underlying language is almost unchanged. The only syntactic difference is that processes now have the form $\mathsf{p} \triangleright_v^{\rho} B$, where ρ is a queue of incoming messages. A message is a pair $\langle \mathsf{q}, m \rangle$, where q is the sender process and m is a value, label, or process identifier.

Semantics. The semantics of aPP consists of rules $\lfloor$P|Cond$\rceil$, $\lfloor$P|Par$\rceil$, $\lfloor$P|Cond$\rceil$, and $\lfloor$P|Start$\rceil$ from PP (Figure 6) together with the rules given in Figure 13 and analogous variants for selection and introduction (see Appendix). In the reductum of $\lfloor$P|Start$\rceil$, the newly created process is initialized with an empty queue. Structural precongruence for aPP is defined exactly as for PP. We write $\rho \cdot \langle \mathsf{q}, m \rangle$ to denote the queue obtained by appending message $\langle \mathsf{q}, m \rangle$ to ρ, and $\langle \mathsf{q}, m \rangle \cdot \rho$ for the queue with $\langle \mathsf{q}, m \rangle$ at the head and ρ as tail. We simulate having one separate queue for each other process by allowing incoming messages from different senders to be exchanged, which we represent using the congruence $\rho \preceq \rho'$ defined by the rule $\langle \mathsf{p}, m \rangle \cdot \langle \mathsf{q}, m' \rangle \preceq \langle \mathsf{q}, m' \rangle \cdot \langle \mathsf{p}, m \rangle$ if $\mathsf{p} \neq \mathsf{q}$.

All behaviours of PP are valid also in aPP.

Theorem 7. *Let N be a PP network. If $N \to_{\mathscr{B}} N'$ (in PP), then $N_{[]} \to_{\mathscr{B}}^* N'_{[]}$ (in aPP), where $N_{[]}$ denotes the asynchronous network obtained by adding an empty queue to each process.*

The converse is not true, so the relation between PP and aPP is not so strong as in Theorem 6 for PC and aPC. This is because of deadlocks: in PP, a communication action can only take place when the sender and receiver are ready to synchronize; in aPP, a process can send a message to another process, even though the intended recipient is not yet able to receive it. For example, the network $\mathsf{p} \triangleright_v \mathsf{q}!* \mid \mathsf{q} \triangleright_w \mathbf{0}$ is deadlocked in PP but not in aPP.

5.3 EndPoint Projection (EPP) from aPC to aPP

Defining an EPP from aPC to aPP requires extending the previous definition with clauses for the new runtime terms. The most interesting part is gener-

ating the local queues for each process: when compiling $\bullet_{\mathsf{p}}.v \twoheadrightarrow \mathsf{q}.f$, for example, we need to add $\langle \mathsf{p}, v \rangle$ at the top of q's queue. However, if we follow this intuition, Theorem 5 no longer holds for arbitrary aPC choreographies. This happens because we can write choreographies that use runtime terms in a "wrong" way, and these are not correctly compiled to aPP. Consider the choreography $C = \mathsf{p}.1 \twoheadrightarrow \mathsf{q}.f ; \bullet_{\mathsf{p}} \overset{2}{\twoheadrightarrow} \mathsf{q}.f$. If we naively project it for some σ, we obtain $\mathsf{p} \triangleright^{[]}_{\sigma(\mathsf{p})} \mathsf{q}!1 \mid \mathsf{q} \triangleright^{\langle \mathsf{p},2 \rangle}_{\sigma(\mathsf{q})} \mathsf{p}?f ; \mathsf{p}?f$, where [] is the empty queue, and q will receive 2 before 1.

To avoid this undesired behaviour, we recall that runtime terms should not be used by the programmer and restrict ourselves to *well-formed* choreographies: those that can arise from executing a choreography that does not contain runtime terms (i.e., a program).

Definition 2 (Well-formedness). *A choreography C in aPC is* well-formed *if $C^R \fatsemi C^{PC} \preceq C$ where:*

- *$\preceq$ is structural precongruence without rule $\lfloor C \mid Unfold \rceil$;*
- *C^R only contains unmatched and instantiated receive actions: $\bullet_{\mathsf{p}} \overset{v}{\twoheadrightarrow} \mathsf{q}.f$, $\bullet_{\mathsf{p}} \overset{l}{\twoheadrightarrow} \mathsf{q}[l]$, $\bullet_{\mathsf{p}}.\mathsf{r} \overset{\mathsf{r}}{\twoheadrightarrow} \mathsf{q}$;*
- *C^{PC} is a PC choreography.*

A procedure $X(\widetilde{\mathsf{q}^T}) = C$ is well-formed *if C does not contain any runtime actions. A procedural choreography $\langle \mathscr{D}, C \rangle$ is well-formed if all procedures in $\mathscr{D}$ are well-formed and C is well-formed.*

Well-formedness is decidable, since the set of choreographies equivalent up to $\preceq$ is decidable. More efficiently, one can check that C is well-formed by swapping all runtime actions to the beginning and folding all paired send/receive terms. Furthermore, choreography execution preserves well-formedness. The problematic choreography described above is not well-formed.

Definition 3 (EPP from aPC to aPP). *Let $\langle \mathscr{D}, C \rangle$ be in aPC and σ be a state. The EPP of $\langle \mathscr{D}, C \rangle$ and σ is defined as*

$$[\![\mathscr{D}, C, \sigma]\!] \;=\; \langle [\![\mathscr{D}]\!], [\![C, \sigma]\!] \rangle \;=\; \left\langle [\![\mathscr{D}]\!], \textstyle\prod_{\mathsf{p} \in \mathsf{pn}(C)} \mathsf{p} \triangleright^{(\!|C|\!)_{\mathsf{p}}}_{\sigma(\mathsf{p})} [\![C]\!]_{\mathsf{p}} \right\rangle$$

where $[\![C]\!]_{\mathsf{p}}$ is defined as for PC (Figure 8) with the extra rules in Figure 14 and $(\!|C|\!)_{\mathsf{p}}$ is defined by the rules in Figure 15. The similar rules for selection and introduction are defined in the Appendix.

In the last case of the definition of $(\!|C|\!)_{\mathsf{p}}$ (bottom-right), η and I range over all cases that are not covered previously. The rule for the conditional may seem a bit surprising: in the case of projectable choreographies, mergeability and well-formedness together imply that unmatched receive actions at a process must occur in the same order in both branches. Note that projection of choreographies with ill-formed runtime terms (e.g., $\bullet_{\mathsf{p}} \overset{l'}{\twoheadrightarrow} \mathsf{q}[l]$ with $l \neq l'$, which cannot occur in a well-formed choreography) is not defined.

With this definition, we can restate Theorem 5 for aPC and aPP.

$$\llbracket \mathsf{p}.e \xrightarrow{x} \bullet_{\mathsf{q}}; C \rrbracket_{\mathsf{r}} = \begin{cases} \mathsf{q}!e; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases} \qquad \llbracket \bullet_{\mathsf{p}} \xrightarrow{\hat{v}} \mathsf{q}.f; C \rrbracket_{\mathsf{r}} = \begin{cases} \mathsf{p}?f; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{otherwise} \end{cases}$$

Fig. 14. Asynchronous PC, Behaviour Projection (New Rules).

$$(\!| \bullet_{\mathsf{p}} \xrightarrow{v} \mathsf{q}.f; C |\!)_{\mathsf{r}} = \begin{cases} \langle \mathsf{p}, v \rangle \cdot (\!| C |\!)_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ (\!| C |\!)_{\mathsf{r}} & \text{otherwise} \end{cases} \qquad \begin{array}{l} (\!| \mathsf{if}\ \mathsf{p} <= \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C |\!)_{\mathsf{r}} = (\!| C_1 |\!)_{\mathsf{r}} \cdot (\!| C |\!)_{\mathsf{r}} \\ (\!| \eta; C |\!)_{\mathsf{r}} = (\!| I; C |\!)_{\mathsf{r}} = (\!| C |\!)_{\mathsf{r}} \end{array}$$

Fig. 15. Asynchronous PC, State Projection.

Theorem 8 (Asynchronous EPP Theorem). *If $\langle \mathscr{D}, C \rangle$ in aPC is projectable and well-formed, $\Gamma \vdash \mathscr{D}$, and $\Gamma; G \vdash C \triangleright G^*$, then, for all σ:*

- *(Completeness) if $G, C, \sigma \to_{\mathscr{D}} G', C', \sigma'$, then $\llbracket C, \sigma \rrbracket \to_{\llbracket \mathscr{D} \rrbracket} \succ \llbracket C', \sigma' \rrbracket$;*
- *(Soundness) if $\llbracket C, \sigma \rrbracket \to_{\llbracket \mathscr{D} \rrbracket} N$, then $G, C, \sigma \to_{\mathscr{D}} G', C', \sigma'$ for some G', σ' such that $\llbracket C', \sigma' \rrbracket \prec N$.*

As a consequence, Corollary 1 applies also to the asynchronous case: the processes projected from aPC into aPP are deadlock-free.

6 Language Extensions

We sketch two simple extensions that enhance the expressivity of (synchronous and asynchronous) PC without affecting its underlying theory: they only require simple modifications, and all the results from the previous sections still hold.

Parameter Lists. Our first extension allows procedure parameters to be lists of processes, on which procedures can then act uniformly by recursion. Formally, a procedure parameter can now be either a process or a list of processes all with the same type. We restrict the usage of such lists to the arguments of procedure calls; however, they may be manipulated by means of pure total functions that take a list as their only argument. For example, if procedure X takes a list P as an argument, then X's body may call other procedures on P, $\mathsf{head}(P)$ or $\mathsf{tail}(P)$, but it may not include a communication involving $\mathsf{head}(P)$ and another process.

The semantics is extended with a new garbage collection rule: $X\langle [] \rangle \preceq_{\mathscr{D}} \mathbf{0}$, i.e., calling a procedure with an empty list is equivalent to termination ($\mathbf{0}$). In order to type procedures that take process lists as arguments, in a typing $X : G \triangleright G'$ we allow the vertices of G and G' to be not only processes, but also (formal) lists. This requires adjusting the premise of $\lfloor \mathrm{T} | \mathrm{Call} \rceil$ to the case when some of the arguments are lists. For when a concrete list is given as argument, we update the premise $G_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \subseteq G$ to check that: if G_X contains an edge between two process lists P and Q, then G must contain edges between p and q for each p

in the list supplied for P and each q in the list supplied for Q. The interpretation of an edge between an argument process and an argument list is similar.

We extend EPP to this enriched language by similarly extending the syntax of PP and adding the rule $\mathsf{q} \triangleright_v X\langle \tilde{\mathsf{p}} \rangle \preceq_{\mathscr{D}} \mathsf{q} \triangleright_v \mathbf{0}$ if $\mathsf{q} \notin \tilde{\mathsf{p}}$. Although we cannot call procedures with parameter lists based on the result of a conditional without changing PP (in particular, due to the definition of merge), in this language we can write more sophisticated examples. In [10], we show how to implement Quicksort, Gaussian elimination and the Fast Fourier Transform in PC, and discuss how implicit parallelism yields efficient implementations of these algorithms.

Procedures with Holes. The second extension allows the presence of holes in procedure definitions. Holes are denoted $\Box_h$, where h is a unique name for the hole in the procedure. Procedure calls are then allowed to specify a choreography to inject in each hole by means of the syntax $X\langle \tilde{\mathsf{p}} \rangle$ with $[h_i \mapsto C_i]_i$ where each C_i is a choreography (in particular, it can be another procedure call) with $\mathsf{pn}(C_i) \subseteq \tilde{\mathsf{p}}$. We assume unspecified holes to be filled with $\mathbf{0}$.

The typing of procedure definitions is extended to $X(\widetilde{\mathsf{q}^T}) : G \triangleright G'$ with $\{h_i : G_i\}_i$, where G_i is the guaranteed connection graph when reaching hole $\Box_{h_i}$. The remaining adaptations (syntax and semantics of PC and PP, and EPP) are trivial.

Example 11. Holes enable higher-order composition of choreographies. Consider the following `exchange` procedure:

```
exchange(p,q,r,s) = p.item -> q.get; □h1; r.item -> s.get; □h2
```

We can either leave the holes empty, or use them to perform some extra actions, e.g., inserting a payment:

```
exchange<p,q,r,s> with h1 ↦ pay<p,q>, h2 ↦ pay<r,s>
```

We can even insert new causal dependencies between the two communications in `exchange`, e.g., using `q` as a broker:

```
exchange<p,q,r,s> with h1 ↦ q.item -> r.get
```

Using holes, we can also define a general-purpose iterator (see Appendix).

7 Discussion, Future and Related Work

Design of PC. PC is part of a long-term research effort motivated by empirical studies on concurrency bugs, such as the taxonomies [22] and [21]. They distinguish between deadlock and non-deadlock bugs. In distributed systems, both are mostly caused by the wrong composition of different protocols, rather than the wrong implementation of single protocols. Deadlocks are prevented in PC (Corollary 1). Non-deadlock bugs are more subtle, and typically due to the informal specification (or lack thereof) of the programmer's intentions wrt protocol composition, which leads to unexpected bad executions. In PC, the programmer's

intentions are formalised as a choreography and the composition of procedures in code. Synthesis (EPP) guarantees that these intentions are respected (Theorem 5). Typical non-deadlock bugs include lack of causality between messages, or between an internal computation and a message [21]. For example, a client may forget to wait for an authentication protocol to complete and include the resulting security key in its request to a server in another protocol. We can formalise the correct scenario in PC using our `auth` procedure from Example 4 (`c` is the client and `s` is the server):

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

There is still ground to be covered to support all the features (e.g., preemption [21]) used in some of the programs included in these empirical studies. Thus, this kind of studies will also be important for future developments of PC.

Multiparty Session Types. In Multiparty Session Types (MPST) [18], global types are 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 other restrictions) 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 the composition of protocols as an implementation detail to the implementors of processes. As a consequence, protocol compositions may lead to deadlocks, unlike in PC. We illustrate this key difference with an example, using our syntax. Consider the protocols $X(\mathsf{r},\mathsf{s}) = \mathsf{r}.e$ `->` $\mathsf{s}.f$ and $Y(\mathsf{r}',\mathsf{s}') = \mathsf{r}'.e'$ `->` $\mathsf{s}'.f'$, and assume that we want to compose their instantiations $X\langle \mathsf{p},\mathsf{q}\rangle$ and $Y\langle \mathsf{q},\mathsf{p}\rangle$. According to MPST, a valid implementation (paraphrased in PP) is $\mathsf{p} \triangleright_v \mathsf{q}?f'; \mathsf{q}!e \mid \mathsf{q} \triangleright_v \mathsf{p}?f; \mathsf{p}!e'$. Although this network is obviously deadlocked, this is not detected by MPST 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 $Y\langle \mathsf{q},\mathsf{p}\rangle; X\langle \mathsf{p},\mathsf{q}\rangle$.

The authors of [9] augment MPST with a type system that prevents protocol compositions that may lead to deadlocks, but their approach is too restrictive when actions from different protocols are interleaved. Consider the following example, reported as untypable but deadlock-free in [9] (adapted to PC).

```
X(p,q) = p.* -> q; □h1; q.* -> p; p.* -> q; □h2

X<p,q> with h1 ↦ q.* -> p; p.* -> q, h2 ↦ q.* -> p
```

This code is both typable and projectable in PC, yielding a correct implementation.

Another technique for deadlock-freedom in MPST and similar models is to restrict connections among processes participating in different protocols to form

a tree [4,6,8]. PC is more expressive, since connections can form an arbitrary graph.

Recent work investigated how to extend MPST to capture protocols where the number of participants in a session is fixed only at runtime [30], or can grow during execution [15]. These results use ad-hoc primitives and "middleware" terms in the process model, e.g., for tracking the number of participants in a session [15]. Such machinery is not needed in PC. The authors of [14] propose a theory of nested MPST, which partially recalls our notion of parametric procedures. Differently from PC, procedures are invoked by a coordinator (requiring extra communications), and compositions of such nested types can deadlock.

Choreographic Programming. None of the examples in this work can be written in previous models for choreographic programming, which lack full procedural abstraction. As far as we know, this is also the first work exploring choreographic programming in a general setting, instead of in the context of web services [5,7,29].

The models in [5,7] are based on choreographic programming and thus support deadlock-freedom even when multiple protocols are used. However, these models do not support general recursion and parametric procedures, both of which are novel in this work. Composition and reusability in these models are thus limited: 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, preventing reuse. In PC, all these limitations are lifted.

Asynchronous communications in choreographic programming were addressed in [7] using an ad-hoc transition rule. Our approach in aPC, which builds on implicit parallelism, is simpler and yields an EPP Theorem (Theorem 8) that states a lockstep correspondence between the programmer's choreography and the synthesised implementation. Instead, in [7], EPP guarantees only a weaker and more complex confluence correspondence.

Holes in PC recall adaptation scopes for choreographies in [12], used to pinpoint where runtime adaptation of code can take place. However, that work deals only with choreographies with a finite number of processes.

A major distinguishing feature of PC is the management of connections among processes, using graphs that are manipulated at runtime. Channel delegation in choreographies [7] behaves in a similar way, but it is less expressive: a process that introduces two other processes cannot communicate with them thenceforth without requiring a new auxiliary channel.

Other paradigms. Reo is another approach that, like PC, disentangles computation from communication and focuses on the programming of interactions, rather than the actions that implement them. In Reo, protocols are obtained by composing (graphical) connectors, which mediate communications among components [1]. As such, Reo produces rather different and more abstract artifacts than PC, which require additional machinery to be implemented (a compiler from Reo to Java uses constraint automata [2]). Programs in PC can be implemented rather directly via standard send and receive primitives (which is

typical of choreographies [7]). PC also supports dynamic creation of processes and guarantees deadlock-freedom by construction, through its simple type system and EPP. Reo supports many interesting connectors and compositions (e.g., multicast), which would be interesting to investigate in PC.

In a different direction, there have been proposals of integrating communication protocols (given as session types) with functional programming [28]. These approaches allow parallel computations of functions to synchronise and exchange values. As for MPST, protocol compositions are left to the programmer, and the same limitations discussed for MPST apply.

Local concurrency. Some choreography models support an explicit parallel operator, e.g., $C \mid C'$. We chose not to present it here to keep our model simple, as most behaviours of $C \mid C'$ are captured by implicit parallelism and asynchrony 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 (Rule ⌊C|I-I⌉, Figure 3). However, when some processes names are shared in the parallel composition, then the parallel operator is more expressive, because it allows for *local concurrency*. As an example, consider $X\langle \mathsf{p}, \mathsf{q} \rangle \mid Y\langle \mathsf{p}, \mathsf{s} \rangle$: here, p is doing more actions concurrently. In PC, we can encode this by spawning new processes, e.g. our example would become $\mathsf{p}\ \mathsf{start}\ \mathsf{r}; \mathsf{p} : \mathsf{r} \texttt{<->} \mathsf{s}; X\langle \mathsf{p}, \mathsf{q} \rangle; Y\langle \mathsf{r}, \mathsf{s} \rangle$. The price we pay is the extra communications to introduce r (p's new process) to s (plus, we may need to send p's value to r, and maybe send r's value to p after Y terminates).

If local concurrency in PC is desired, there are different options for introducing it. A straightforward one is to adapt the parallel operator from [5,18], which, however, abstracts from how local concurrency is implemented. A more interesting strategy would be to support named local threads in processes. Threads would share the process location, such that they can receive at the same name without the need to be introduced. Then, implicit parallelism would allow for swapping actions at threads (not just processes) with distinct names. Threads at the same process should not receive from the same sender process at the same time, to avoid races, which can be prevented via typing (cf. [5]). Integration with separation logic [26] would also be interesting, to allow threads also to safely share data.

Sessions, Connections, and Mobility. All recent theories based on session types (e.g., [5,7,8,9,18]) assume that all processes in a session (a protocol execution) have a private full-duplex channel to communicate with each other in that session. This amounts to requiring that the graph of connections among processes in a protocol is always complete. This assumption is not necessary in PC, since our semantics and typing require only the processes that actually communicate to be connected. This makes PC a suitable model for reasoning about different kinds of network topologies. In the future, it would be interesting to see whether our type system and connection graphs can be used to enforce pre-defined network structures (e.g., hypercubes or butterflies), making PC a candidate for the programming of choreographies that account for hardware restrictions.

Another important aspect of sessions is that each new protocol execution requires the creation of a new session (and all the connections among its par-

ticipants!). By contrast, protocol executions (running procedures) in PC can reuse all available connections – allowing for more efficient implementations. We use this feature in our parallel downloader example (Example 5), where `c'` can still communicate with `c` even after the latter introduced it to `s'`. Also, aPC supports for the first time the asynchronous establishment of new connections among processes.

The standard results of communication safety found in session-typed calculi can be derived from our EPP Theorem (Theorem 5), as discussed in [7].

Faults. Some interesting distributed bugs [21] are triggered by unexpected fault conditions at nodes, making such faults an immediate candidate for the future developments of PC. Useful inspiration to this aim may be provided by [3].

We have also 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.

Integration. Sometimes, the code synthesised from a choreography has to be used in combination with legacy process code. For example, we may want to use an existing authentication server in our `auth` procedure in Example 4. This issue is addressed in [24] by using a type theory based on sessions. We leave an adaptation of this idea in our setting to future work.

References

1. Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
2. Farhad Arbab, Christel Baier, Jan J. M. M. Rutten, and Marjan Sirjani. Modeling component connectors in reo by constraint automata: (extended abstract). *Electr. Notes Theor. Comput. Sci.*, 97:25–46, 2004.
3. Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):156–205, 2016.
4. Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In *Proc. of ICE'10*, 2010.
5. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
6. Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: a logical explanation of multiparty session types. In *CONCUR*, 2016. To appear.
7. Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 263–274. ACM, 2013.
8. Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. In L. Aceto and D. de Frutos-Escrig, editors, *CONCUR*, volume 42 of *LIPIcs*, pages 412–426. Schloss Dagstuhl, 2015.
9. Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016.
10. Luís Cruz-Filipe and Fabrizio Montesi. Choreographies in practice. In Elvira Albert and Ivan Lanese, editors, *FORTE*, volume 9688 of *LNCS*, pages 1–10. Springer, 2016.
11. Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. 2016. Accepted for publication.
12. Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies – safe runtime updates of distributed applications. In T. Holvoet and M. Viroli, editors, *COORDINATION*, volume 9037 of *LNCS*, pages 67–82. Springer, 2015.
13. Frank S. de Boer, Mohammad Mahdi Jaghoori, Cosimo Laneve, and Gianluigi Zavattaro. Decidability problems for actor systems. *Logical Methods in Computer Science*, 10(4), 2014.
14. Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR*, pages 272–286, 2012.
15. Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In T. Ball and M. Sagiv, editors, *POPL*, pages 435–446. ACM, 2011.
16. Maurizio Gabbrielli, Saverio Giallorenzo, and Fabrizio Montesi. Applied choreographies. *CoRR*, abs/1510.03637, 2015.
17. Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In C. Hankin, editor, *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.
18. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *J. ACM*, 63(1):9, 2016.

19. Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In A. Cerone and S. Gruner, editors, *SEFM*, pages 323–332. IEEE, 2008.
20. Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Amending choreographies. In *Proceedings 9th International Workshop on Automated Specification and Verification of Web Systems, WWV 2013, Florence, Italy, 6th June 2013.*, pages 34–48, 2013.
21. Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS*, pages 517–530. ACM, 2016.
22. Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *ACM SIGARCH Computer Architecture News*, 36(1):329–339, 2008.
23. Fabrizio Montesi. *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen, 2013. http://fabriziomontesi.com/files/choreographic_programming.pdf .
24. Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In P.R. D'Argenio and H.C. Melgratti, editors, *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.
25. MPI Forum. *MPI: A Message-Passing Interface Standard*. High-Performance Computing Center Stuttgart, 2015. Version 3.1.
26. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.
27. Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
28. Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1–2):64–87, 2006.
29. 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.
30. Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In C.-H. Luke Ong, editor, *FOSSACS*, volume 6014 of *LNCS*, pages 128–145. Springer, 2010.

A Detailed definitions and proofs

We report on definitions and proofs that were omitted from the main part of this paper, as well as some more detailed examples.

A.1 Type checking and type inference

We start with some technical lemmas about typing.

Lemma 1 (Monotonicity). *Let Γ and Γ' be typing contexts with $\Gamma \subseteq \Gamma'$, G_1, G'_1 and G be connection graphs such that $G_1 \subseteq G$, and C be a choreography. If $\Gamma; G_1 \vdash C \triangleright G'_1$, then $\Gamma'; G \vdash C \triangleright G \cup G'_1$.*

Proof. Straightforward by induction on the derivation of $\Gamma; G_1 \vdash C \triangleright G'_1$.

Lemma 2 (Sequentiality). *Let Γ be a typing context, G_1, G'_1, G_2 and G'_2 be connection graphs such that $G_2 \subseteq G'_1$, and C_1, C_2 be choreographies. If $\Gamma; G_1 \vdash C_1 \triangleright G'_1$ and $\Gamma; G_2 \vdash C_2 \triangleright G'_2$, then $\Gamma; G_1 \vdash C_1 \fatsemi C_2 \triangleright G'_1 \cup G'_2$.*

Proof. Straightforward by induction on the derivation of $\Gamma; G_2 \vdash C_2 \triangleright G'_2$.

Lemma 3 (Substitution). *Let Γ be a typing context, G and G' be connection graphs, and C be a choreography. Let $\tilde{\mathsf{p}}$ be a set of process names that are free in C and $\tilde{\mathsf{q}}$ be a set of process names that do not occur (free or bound) in C. If $\Gamma; G \vdash C \triangleright G'$, then $\Gamma[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]; G[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \vdash C[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \triangleright G'[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]$.*

Proof. Straightforward by induction on the derivation of $\Gamma; G \vdash C \triangleright G'$, as all typing rules are valid when substitutions are applied.

We are now ready to start proving Theorem 1. The following lemma takes care of the base cases, and is required for one of the inductive steps.

Lemma 4. *Let Γ be a set of typing judgements, $\mathscr{D}$ a set of procedure definitions, G_1 and G'_1 connection graphs, and C a choreography that does not start with $\mathbf{0}$ or a procedure call. Assume that $\Gamma \vdash \mathscr{D}$ and $\Gamma; G_1 \vdash C \triangleright G'_1$. For every state σ, there exist Γ', σ', C', G_2 and G'_2 such that $G_1, C, \sigma \to_{\mathscr{D}} G_2, C', \sigma'$ and $\Gamma'; G_2 \vdash C' \triangleright G'_2$.*

Proof. By case analysis on the last step of the proof of $\Gamma; G_1 \vdash C \triangleright G'_1$. By hypothesis, this proof cannot end with an application of rules $\lfloor$T|End$\rceil$, $\lfloor$T|EndSeq$\rceil$ or $\lfloor$T|Call$\rceil$; we detail all cases for completeness, but the only non-trivial one is the last.

- $\lfloor$T|Start$\rceil$: then C is $\mathsf{p}\,\mathsf{start}\,\mathsf{q}^T; C^\circ$ and by hypothesis

$$\Gamma, q : T; G_1 \cup \{p \leftrightarrow q\} \vdash C^\circ \triangleright G'_1 \,.$$

  Since $G_1, \mathsf{p}\,\mathsf{start}\,\mathsf{q}^T; C, \sigma \to_{\mathscr{D}} G_1 \cup \{p \leftrightarrow q\}, C^\circ, \sigma[\mathsf{q} \mapsto \bot_T]$ by rule $\lfloor$C|Start$\rceil$, taking $\Gamma' = \Gamma, \mathsf{q} : T$, $\sigma' = \sigma[\mathsf{q} \mapsto \bot_T]$, $C' = C^\circ$, $G_2 = G_1 \cup \{p \leftrightarrow q\}$ and $G'_2 = G'_1$ establishes the thesis.

- $\lfloor$T|Com$\rceil$: then C is $\mathsf{p}.e$ -> $\mathsf{q}.f; C^\circ$ and by hypothesis $p \xleftrightarrow{G_1} q$, $f[\sigma(\mathsf{q})/*](e[\sigma(\mathsf{p})/*])$ is a valid expression of type T_q, and $\Gamma; G_1 \vdash C \triangleright G'_1$. Then all the preconditions of $\lfloor$C|Com$\rceil$ are met, so taking $\Gamma' = \Gamma$, $\sigma' = \sigma[\mathsf{q} \mapsto f[\sigma(\mathsf{q})/*](e[\sigma(\mathsf{p})/*])]$, $C' = C^\circ$, $G_2 = G_1$ and $G'_2 = G'_1$ establishes the thesis.
- $\lfloor$T|Sel$\rceil$: then C is p -> $\mathsf{q}[l]; C^\circ$ and by hypothesis $p \xleftrightarrow{G_1} q$ and $\Gamma; G_1 \vdash C \triangleright G'_1$. By $\lfloor$C|Sel$\rceil$, G_1, p -> $\mathsf{q}[l]; C^\circ, \sigma \to_{\mathscr{D}} G_1, C^\circ, \sigma$, so taking $\Gamma' = \Gamma$, $\sigma' = \sigma$, $C' = C^\circ$, $G_2 = G_1$ and $G'_2 = G'_1$ again establishes the thesis.
- $\lfloor$T|Tell$\rceil$: then C is $\mathsf{p} : \mathsf{q}$ <-> $\mathsf{r}; C^\circ$ and by hypothesis both $p \xleftrightarrow{G_1} q$, $p \xleftrightarrow{G_1} r$, and $\Gamma; G_1 \cup \{q \leftrightarrow r\} \vdash C^\circ \triangleright G'_1$. Since the preconditions of rule $\lfloor$C|Tell$\rceil$ are met, by taking $\Gamma' = \Gamma$, $\sigma' = \sigma$, $C' = C^\circ$, $G_2 = G_1 \cup \{q \leftrightarrow r\}$ and $G'_2 = G'_1$ establishes the thesis.
- $\lfloor$T|Cond$\rceil$: then C is $\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C^\circ$ and by hypothesis $e[\sigma(\mathsf{p})/*]$ is a valid Boolean expression, $\Gamma; G_1 \vdash C_i \triangleright G_i^\circ$ and $\Gamma; G_1^\circ \cap G_2^\circ \vdash C^\circ \triangleright G'_1$. Suppose $e[\sigma(\mathsf{p})/*] = \mathsf{true}$ (the other case is similar). Then $G_1, \mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C^\circ, \sigma \to_{\mathscr{D}} G_1, C_1 \fatsemi C, \sigma$. Since $G_1^\circ \cap G_2^\circ \subseteq G_1^\circ$, Lemma 2 allows us to conclude that $\Gamma; G_1 \vdash C_1 \fatsemi C \triangleright G'_1 \cup G_1^\circ$, whence the thesis follows by taking $\Gamma' = \Gamma$, $C' = C_1 \fatsemi C$, $G_2 = G_1$ and $G'_2 = G'_1 \cup G_1^\circ$.

Proof (Theorem 1). If $C \preceq_{\mathscr{D}} \mathbf{0}$, then the first case holds. Assume that $C \not\preceq_{\mathscr{D}} \mathbf{0}$; we show that the second case holds by induction on the proof of $\Gamma; G_1 \vdash C \triangleright G'_1$. By hypothesis, the last rule applied in this proof cannot be $\lfloor$T|End$\rceil$; the cases where the last rule applied is $\lfloor$T|Start$\rceil$, $\lfloor$T|Com$\rceil$, $\lfloor$T|Sel$\rceil$, $\lfloor$T|Tell$\rceil$ or $\lfloor$T|Cond$\rceil$ follow immediately from Lemma 4, while the case of rule $\lfloor$T|EndSeq$\rceil$ is straightforward from the induction hypothesis.

We focus on the case of rule $\lfloor$T|Call$\rceil$. In this case, C has the form $X\langle\tilde{\mathsf{p}}\rangle; C^\circ$, and we know that $\Gamma \vdash X(\widetilde{\mathsf{q}^T}) : (G_X \triangleright G'_X)$, $\Gamma \vdash \tilde{\mathsf{p}} : T$, $G_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \subseteq G_1$ and $\Gamma; G_1 \cup (G'_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]) \vdash C^\circ \triangleright G'_1$. From the hypothesis that $\Gamma \vdash \mathscr{D}$ we also know that $\Gamma_X; G_X \vdash C_X \triangleright G'_X$, where C_X is the body of X as defined in $\mathscr{D}$. By Lemma 3, $\Gamma_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]; G_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \vdash C_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \triangleright G'_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]$, whence by Lemma 1 also $\Gamma; G_1 \vdash C_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \triangleright G'_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \cup G_1$. By applying rule $\lfloor$C|Unfold$\rceil$, we conclude that $X\langle\tilde{\mathsf{p}}\rangle; C^\circ \preceq_{\mathscr{D}} C_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \fatsemi C^\circ$, and Lemma 2 allows us to conclude that $\Gamma; G_1 \vdash C_X[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}] \fatsemi C^\circ \triangleright G'_1$. Since procedure calls are guarded, C_X does not begin with a procedure call, and Lemma 4 establishes the thesis.

Proof (Theorem 2). The proof of this result proceeds in several stages. We first observe that deciding whether $\Gamma; G \vdash C \triangleright G'$ is completely mechanical, as the typing rules are deterministic. Furthermore, those rules can also be used to construct G' from G and C; therefore, the key step of this proof is showing, given Γ and $\langle\mathscr{D}, C\rangle$, how to find a "canonical typing" for the recursive definitions, the set $\Gamma_{\mathscr{D}}$, such that $\Gamma_{\mathscr{D}} \vdash \mathscr{D}$ and $\Gamma, \Gamma_{\mathscr{D}}; G_C \vdash C \triangleright G'$ (with G' inferred) iff $\Gamma, \Gamma'; G_C \vdash C \triangleright G''$ for some Γ' and G''. More precisely, we need to find graphs G_X and G'_X for each procedure X defined in $\mathscr{D}$.

Our proof proceeds in three steps. First, for each X we compute an underapproximation G_X° of the output graph G'_X, containing all the relevant connections that executing X can add. Using this, we are able to compute the input graph

G_X and the output graph $G'_X = G_X \cup G^\circ_X$. Both these steps are achieved by computing a minimal fixpoint of a monotonic operator in the set of all graphs whose vertices are the parameters of X. Finally, we argue that the typing $X : G_X \triangleright G_X$ is minimal, and therefore the set $\Gamma_{\mathscr{D}}$ of all such typings fulfills the property we require.

Throughout the remainder of this proof, we assume $\mathscr{D} = \{X_i(\widetilde{\mathsf{q}^i}) = C_i \mid i = 1, \ldots, n\}$.

1. In order to compute $G^\circ_{X_i}$, we define an auxiliary function fwd with intended meaning as follows: $\mathsf{fwd}^{\tilde{G}_i}_{C_j}(G)$ computes the communication graph obtained from G after one execution of the body of X_j, assuming that $X_i(\widetilde{\mathsf{q}_i}) : \emptyset \triangleright G_i$ for all i and ignoring newly created processes. We use a conditional union operator $\uplus$ where $G \uplus \{e\}$ denotes $G \cup \{e\}$ if e is an edge connecting two vertices in G, and G otherwise. The function fwd is defined as follows.

$$\mathsf{fwd}^{\tilde{G}_i}_{\mathbf{0}}(G) = G$$
$$\mathsf{fwd}^{\tilde{G}_i}_{\mathbf{0};C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(G)$$
$$\mathsf{fwd}^{\tilde{G}_i}_{\mathsf{p}.e \texttt{ -> } \mathsf{q}.f;C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(G)$$

$$\mathsf{fwd}^{\tilde{G}_i}_{\mathsf{p} \texttt{ -> } \mathsf{q}[l];C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(G)$$
$$\mathsf{fwd}^{\tilde{G}_i}_{\mathsf{p} \texttt{ start } \mathsf{q}^T;C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(G)$$
$$\mathsf{fwd}^{\tilde{G}_i}_{\mathsf{p}:\mathsf{q} \texttt{ <-> } \mathsf{r};C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(G \uplus \{\mathsf{q} \leftrightarrow \mathsf{r}\})$$
$$\mathsf{fwd}^{\tilde{G}_i}_{\texttt{if } \mathsf{p}.e \texttt{ then } C_1 \texttt{ else } C_2;C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(\mathsf{fwd}^{\tilde{G}_i}_{C_1}(G) \cap \mathsf{fwd}^{\tilde{G}_i}_{C_2}(G))$$
$$\mathsf{fwd}^{\tilde{G}_i}_{X_i\langle\tilde{\mathsf{p}}\rangle;C}(G) = \mathsf{fwd}^{\tilde{G}_i}_{C}(G \uplus G_i[\tilde{\mathsf{p}}/\widetilde{\mathsf{q}^i}])$$

   Using fwd, we define an operator $\mathscr{T}_{\mathsf{fwd}}$ over the set $\mathscr{G}$ of tuples of graphs over the parameters of X_i, i.e. $\mathscr{G} = \{\widetilde{G_i} \mid G_i \text{ is a graph over } \widetilde{\mathsf{q}^i}\}$. Observe that $\mathscr{G}$ is a complete lattice wrt componentwise inclusion.

$$\mathscr{T}_{\mathsf{fwd}}(\widetilde{G_i}) = \widetilde{\mathsf{fwd}^{\tilde{G}_i}_{C_i}(G_i)}$$

   This operator is monotonic, since fwd only adds edges to its argument, and thus has a least fixpoint that can be computed by iterating $\mathscr{T}_{\mathsf{fwd}}$ from the tuple of empty graphs over the right sets of vertices. Furthermore, since $\mathscr{G}$ is finite (each graph has a finite number of vertices) this fixpoint corresponds to a finite iterate, and can thus be computed in finite time. We denote this fixpoint by $\widetilde{G^\circ_{X_i}}$.
2. The construction of the input graphs G_{X_i} follows the same idea: we go through the C_is noting the edges that are required for all communications to be able to take place. It is however slightly more complicated, because we have to keep track of edges that the choreography adds to the graph; we therefore need a function bck that manipulates two graphs instead of one.

More precisely, $\mathsf{bck}_C^{\tilde{G}_i}(G, G)$ returns the graph extending G that is needed for correctly executing C (ignoring newly created processes); the first argument keeps track of the edges that need to be added to G, and the second argument keeps track of edges added by executing C. This function uses the graphs $G^\circ_{X_i}$ computed earlier, which explains why it has to be defined afterwards. We use the same notational conventions as above, and let $\mathsf{fst}\langle a, b\rangle = a$ and $\mathsf{snd}\langle a, b\rangle = b$. The definition of bck is given in Figure 16; it is not the simplest

$$
\begin{aligned}
\mathsf{bck}_{\mathbf{0}}^{\tilde{G}_i}(G_a, G_b) &= \langle G_a, G_b\rangle \\
\mathsf{bck}_{\mathbf{0};C}^{\tilde{G}_i}(G_a, G_b) &= \mathsf{bck}_C^{\tilde{G}_i}(G_a, G_b) \\
\mathsf{bck}_{\mathsf{p}.e\ \texttt{->}\ \mathsf{q}.f;C}^{\tilde{G}_i}(G_a, G_b) &= \begin{cases} \mathsf{bck}_C^{\tilde{G}_i}(G_a, G_b) & \text{if } \mathsf{p} \leftrightarrow \mathsf{q} \in G_b \\ \mathsf{bck}_C^{\tilde{G}_i}(G_a \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}\}, G_b \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}\}) & \text{otherwise} \end{cases} \\
\mathsf{bck}_{\mathsf{p}\ \texttt{->}\ \mathsf{q}[l];C}^{\tilde{G}_i}(G_a, G_b) &= \begin{cases} \mathsf{bck}_C^{\tilde{G}_i}(G_a, G_b) & \text{if } \mathsf{p} \leftrightarrow \mathsf{q} \in G_b \\ \mathsf{bck}_C^{\tilde{G}_i}(G_a \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}\}, G_b \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}\}) & \text{otherwise} \end{cases} \\
\mathsf{bck}_{\mathsf{p}\ \mathsf{start}\ \mathsf{q}\ T;C}^{\tilde{G}_i}(G_a, G_b) &= \mathsf{bck}_C^{\tilde{G}_i}(G_a, G_b) \\
\mathsf{bck}_{\mathsf{p}:\mathsf{q}\ \texttt{<-->}\ \mathsf{r};C}^{\tilde{G}_i}(G_a, G_b) &= \begin{cases} \mathsf{bck}_C^{\tilde{G}_i}(G_a, G_b \uplus \{\mathsf{q} \leftrightarrow \mathsf{r}\}) & \text{if } \mathsf{p} \leftrightarrow \mathsf{q}, \mathsf{p} \leftrightarrow \mathsf{r} \in G_b \\ \mathsf{bck}_C^{\tilde{G}_i}(G_a \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}\}, G_b \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}, \mathsf{q} \leftrightarrow \mathsf{r}\}) & \text{if } \mathsf{p} \leftrightarrow \mathsf{q} \notin G_b,\ \mathsf{p} \leftrightarrow \mathsf{r} \in G_b \\ \mathsf{bck}_C^{\tilde{G}_i}(G_a \uplus \{\mathsf{p} \leftrightarrow \mathsf{r}\}, G_b \uplus \{\mathsf{p} \leftrightarrow \mathsf{r}, \mathsf{q} \leftrightarrow \mathsf{r}\}) & \text{if } \mathsf{p} \leftrightarrow \mathsf{q} \in G_b,\ \mathsf{p} \leftrightarrow \mathsf{r} \notin G_b \\ \mathsf{bck}_C^{\tilde{G}_i}(G_a \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}, \mathsf{p} \leftrightarrow \mathsf{r}\}, G_b \uplus \{\mathsf{p} \leftrightarrow \mathsf{q}, \mathsf{p} \leftrightarrow \mathsf{r}, \mathsf{q} \leftrightarrow \mathsf{r}\}) & \text{if } \mathsf{p} \leftrightarrow \mathsf{q}, \mathsf{p} \leftrightarrow \mathsf{r} \notin G_b \end{cases} \\
\mathsf{bck}_{\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2;C}^{\tilde{G}_i}(G_a, G_b) &= \mathsf{bck}_C^{\tilde{G}_i}(\mathsf{fst}(\mathsf{bck}_{C_1}^{\tilde{G}_i}(G_a, G_b)) \cup \mathsf{fst}(\mathsf{bck}_{C_2}^{\tilde{G}_i}(G_a, G_b)), \mathsf{snd}(\mathsf{bck}_{C_1}^{\tilde{G}_i}(G_a, G_b)) \cap \mathsf{snd}(\mathsf{bck}_{C_2}^{\tilde{G}_i}(G_a, G_b))) \\
\mathsf{bck}_{X_i\,\langle \tilde{\mathsf{p}}\rangle;C}^{\tilde{G}_i}(G_a, G_b) &= \mathsf{bck}_C^{\tilde{G}_i}(G_a \uplus (G_i[\tilde{\mathsf{p}}/\tilde{\mathsf{q}^i}] \setminus G_b), G_b \uplus G_i[\tilde{\mathsf{p}}/\tilde{\mathsf{q}^i}] \uplus G_i^\circ[\tilde{\mathsf{p}}/\tilde{\mathsf{q}^i}])
\end{aligned}
$$

Fig. 16. Definition of bck (case 2 in the proof of Theorem 2).

possible, but the formulation given is sufficient for our purposes.
Again we define a monotonic operator over the same $\mathscr{G}$ as above.

$$\mathscr{T}_{\mathsf{bck}}(\widetilde{G_i}) = \mathsf{fst}(\widetilde{\mathsf{bck}_{C_i}^{\tilde{G}_i}(G_i, G_i)})$$

We do not need to recompute G_i°, since these graphs contain all edges that can possibly be added by executing C_i. The least fixpoint of $\mathscr{T}_{\mathsf{bck}}$ can again be computed by finitely iterating this operator, and it is precisely $\widetilde{G_{X_i}}$. We then define $G'_{X_i} = G_{X_i} \cup G^\circ_{X_i}$.

3. We now show that $\Gamma_{\mathscr{D}} = \{X_i(\tilde{\mathsf{q}_i}) : G_{X_i} \triangleright G'_{X_i}\}$ is a minimal typing of $\mathscr{D}$, in the sense explained earlier. Observe that it is possible that $\Gamma_{\mathscr{D}} \not\vdash \mathscr{D}$, in particular if the X_i are ill-formed choreographies.
Suppose that $\Gamma, \Gamma'; G_C \vdash C \triangleright G$ for some Γ' and G. We argue that $\Gamma, \Gamma_{\mathscr{D}}; G_C \vdash C \triangleright G'$, where G' is inferred from the typing rules. For each procedure $X_i(\tilde{\mathsf{q}_i}) = C_i$, there must be a unique typing $X_i(\tilde{\mathsf{q}_i}) : G^*_{X_i} \triangleright G^{**}_{X_i}$ in Γ'. By a simple inductive argument one can show that $G^\circ_{X_i} \subseteq G^{**}$ (since $\emptyset \subseteq G^{**}_{X_i}$ and $\mathscr{T}_{\mathsf{fwd}}$ preserves inclusion in $G^{**}_{X_i}$). Similarly, one shows that $G_{X_i} \subseteq G^*_{X_i}$

and that $G^{**}_{X_i} \setminus G^{*}_{X_i} \subseteq G'_{X_i} \setminus G_{X_i}$. As a consequence, the typing derivation for $\Gamma, \Gamma'; G_C \vdash C \triangleright G$ can be used for $\Gamma, \Gamma_{\mathscr{D}}; G_C \vdash C \triangleright G'$, as all applications of rule $\lfloor$T|Call$\rceil$ are guaranteed to be valid (their preconditions hold) and to produce the same results (they change the communication graph in the same way).

A consequence of this result is that we can obtain a type inference algorithm (Theorem 3), as we only need to "guess" types for the processes in the choreography. As a corollary of the proof, we can also infer the types for parameters of procedural definitions and freshly created processes.

Proof (Theorem 3). Construct Γ by going through C and adding $\mathsf{p} : T_{\mathsf{p}}$ every time there is an action that depends on p's type (i.e. p is a sender or receiver in a communication, or an argument of a procedure call). If Γ contains two different types for any process, then output `NO`, else output Γ. This algorithm will not necessarily assign a type to all processes in C, in case C contains processes whose memory is never accessed.

Proof (Theorem 4). Inferring the types of freshly created processes is analogous to the previous proof.

As for parameters of procedure definitions, we omit the details of the proof, as it repeats ideas previously used. Define an operator $\mathscr{T}_{\mathbb{T}}$ over tuples of typing contexts (one for each X_i defined in $\mathscr{D}$) that generates a typing context for each X_i in the same way as in the previous proof. If any contradictions are found, then fail. Iterate $\mathscr{T}_{\mathbb{T}}$ until either failure occurs (in which case the X_is are not properly defined) or a fixpoint is reached. Finally, assign a random type (e.g. $\mathbb{N}$) to each process variable that has not received a type during this procedure. The algorithm readily extends to infer the types of processes created inside procedure definitions.

A.2 EndPoint Projection

Merging. The full definition of merging is given in Figure 17.

Projections of procedure definitions. When projecting procedure definitions, we simply kept all arguments, and relied on typing to guarantee that no process will ever attempt to communicate with another process it does not know. However, it is "cleaner" to refine the definition of projection so that such parameters are not even formally used.

This is technically not challenging, as we can use typing information to decide which parameters should be kept in each projection. In other words, when computing $[\![X(\widetilde{\mathsf{q}^T}) = C]\!]$, the projected procedure X_i will only contain as arguments those q_j such that $\mathsf{q}_i \overset{G_X}{\longleftrightarrow} \mathsf{q}_j$, where G_X is given by typing. (This definition is non-deterministic, but it can be made deterministic by using the minimal graph G_X computed by the type inference algorithm.) A simple annotation can then ensure that projected procedure calls only keep the arguments

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

$$(\mathsf{q}!!\mathsf{r}; B) \sqcup (\mathsf{q}!!\mathsf{r}; B') = \mathsf{q}!!\mathsf{r}; (B \sqcup B') \qquad (\mathsf{q}?\mathsf{r}; B) \sqcup (\mathsf{q}?\mathsf{r}; B') = \mathsf{q}?\mathsf{r}; (B \sqcup B')$$

$$(\mathsf{q} \oplus l; B) \sqcup (\mathsf{q} \oplus l; B') = \mathsf{q} \oplus l; (B \sqcup B') \qquad (X\langle \tilde{\mathsf{p}} \rangle; B) \sqcup (X\langle \tilde{\mathsf{p}} \rangle; B') = X\langle \tilde{\mathsf{p}} \rangle; (B \sqcup B')$$

$$(\mathsf{start}\ \mathsf{q} \triangleright B_2; B_1) \sqcup (\mathsf{start}\ \mathsf{q} \triangleright B_2'; B_1') = \mathsf{start}\ \mathsf{q} \triangleright (B_2 \sqcup B_2'); (B_1 \sqcup B_1')$$

$$B_1 \sqcup B_2 = B_1' \sqcup B_2' \quad (\text{if } B_1 \preceq B_1' \text{ and } B_2 \preceq B_2') \qquad (\mathbf{0}; B) \sqcup B' = B \sqcup B'$$

$$(\mathsf{if}\ e\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2); B \sqcup (\mathsf{if}\ e\ \mathsf{then}\ B_1'\ \mathsf{else}\ B_2'); B' = (\mathsf{if}\ e\ \mathsf{then}\ (B_1 \sqcup B_1')\ \mathsf{else}\ (B_2 \sqcup B_2')); (B \sqcup B')$$

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

Fig. 17. Merging operator in PP.

in the corresponding positions, and typing guarantees that all those arguments are known at runtime by the process invoking the recursive call.

An implementation of Quicksort showcasing this reasoning was previously published in [10].

EPP. We sketch the proof of the EPP Theorem.

Proof (Theorem 5 (sketch)). The structure of the proof is standard, from [23], so we only show the most interesting differing details. In particular, we need to be careful about how we deal with connections, which is a new key ingredient in PC wrt to previous work. We demonstrate this point for the direction of *(Completeness)*; the direction for *(Soundness)* is proven similarly. The proof proceeds by induction on the derivation of $G, C, \sigma \to_{\mathscr{D}} G'', C', \sigma'$. The interesting cases are reported below.

- $\lfloor$C|Tell$\rceil$: From the definition of EPP we get:

  $$[\![\mathsf{p}\colon \mathsf{q}\ \texttt{<->}\ \mathsf{r}; C^\circ, \sigma]\!] \preceq \mathsf{p} \triangleright_{\sigma(\mathsf{p})} \mathsf{q}!!\mathsf{r}; [\![C^\circ]\!]_{\mathsf{p}} \mid \mathsf{q} \triangleright_{\sigma(\mathsf{q})} \mathsf{p}?\mathsf{r}; [\![C^\circ]\!]_{\mathsf{q}} \mid \mathsf{r} \triangleright_{\sigma(\mathsf{r})} \mathsf{p}?\mathsf{q}; [\![C^\circ]\!]_{\mathsf{r}} \mid N$$

  By $\lfloor$P|Tell$\rceil$ we get:

  $$[\![\mathsf{p}\colon \mathsf{q}\ \texttt{<->}\ \mathsf{r}; C^\circ, \sigma]\!] \to \mathsf{p} \triangleright_{\sigma'(\mathsf{p})} [\![C^\circ]\!]_{\mathsf{p}} \mid \mathsf{q} \triangleright_{\sigma'(\mathsf{q})} [\![C^\circ]\!]_{\mathsf{q}} \mid \mathsf{r} \triangleright_{\sigma'(\mathsf{r})} [\![C^\circ]\!]_{\mathsf{r}} \mid N$$

  which proves the thesis, since we can assume that the projection of C' remains unchanged for the other processes (N stays the same).
- $\lfloor$C|Start$\rceil$: This is the most interesting case. From the definition of EPP we get:

  $$[\![\mathsf{p}\ \mathsf{start}\ \mathsf{q}^T; C^\circ, \sigma]\!] \preceq \mathsf{p} \triangleright_{\sigma(\mathsf{p})} \mathsf{start}\ \mathsf{q}^T \triangleright [\![C^\circ]\!]_{\mathsf{q}}; [\![C^\circ]\!]_{\mathsf{p}} \mid N$$

From the semantics of PP we get:

$$[\![\mathsf{p}\,\mathsf{start}\,\mathsf{q}^T; C^\circ, \sigma]\!] \to \mathsf{p} \triangleright_{\sigma'(\mathsf{p})} ([\![C^\circ]\!]_\mathsf{p})[\mathsf{q}'/\mathsf{q}] \mid \mathsf{q}' \triangleright_{\bot_T} [\![C^\circ]\!]_{\mathsf{q}'} \mid N$$

The Barendregt convention implies $C \preceq (\mathsf{p}\,\mathsf{start}\,\mathsf{q}^T; C^\circ)[\mathsf{q}'/\mathsf{q}]$. We now have to prove that:

$$[\![C^\circ[\mathsf{q}'/\mathsf{q}], \sigma]\!] \preceq \mathsf{p} \triangleright_{\sigma'(\mathsf{p})} ([\![C^\circ]\!]_\mathsf{p})[\mathsf{q}'/\mathsf{q}] \mid \mathsf{q}' \triangleright_{\bot_T} [\![C^\circ]\!]_{\mathsf{q}'} \mid N$$

We observe that this is true only if process q does not occur free in N, i.e., q appear in N only inside the scope of a binder. The latter must be of the form $\mathsf{r}?\mathsf{q}; B$. This is guaranteed by the fact that C is well-typed, since the typing rules prevent other processes in N to communicate with q without being first introduced. □

Choreography Amendment. We now define precisely an amendment function that makes every choreography projectable. As mentioned earlier, we follow ideas previously used in other choreography models [20,11]. Observe that, by definition, the only choreography construct that can lead to unprojectability is the conditional.

Definition 4 (Amendment). *Given a choreography C, the transformation* $\mathsf{Amend}(C)$ *repeatedly applies the following procedure until it is no longer possible, starting from the inner-most subterms in C. For each conditional subterm* $\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ *in C, let* $\tilde{\mathsf{r}} \subseteq (\mathsf{pn}(C_1) \cup \mathsf{pn}(C_2))$ *be such that* $[\![C_1]\!]_\mathsf{r} \sqcup [\![C_2]\!]_\mathsf{r}$ *is undefined for all* $\mathsf{r} \in \tilde{\mathsf{r}}$*; then* $\mathsf{if}\ \mathsf{p}\ \texttt{<=}\ \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ *in C is replaced with:*

$$\begin{array}{l}\mathsf{if}\ \mathsf{p}.e\ \ \mathsf{then}\ \mathsf{p}\ \texttt{->}\ \mathsf{r}_1[\mathrm{L}]; \cdots ; \mathsf{p}\ \texttt{->}\ \mathsf{r}_n[\mathrm{L}]; C_1 \\ \qquad\mathsf{else}\ \ \mathsf{p}\ \texttt{->}\ \mathsf{r}_1[\mathrm{R}]; \cdots ; \mathsf{p}\ \texttt{->}\ \mathsf{r}_n[\mathrm{R}]; C_2\end{array}$$

By the definitions of Amend and EPP and the semantics of PC, we get the following properties, where $\to^*$ is the transitive closure of $\to$.

Theorem 9 (Amendment). *Let C be a choreography. Then:*

(Completeness) $\mathsf{Amend}(C)$ *is defined;*
(Projectability) *for all σ, $[\![\mathsf{Amend}(C), \sigma]\!]$ is defined;*
(Correspondence) *for all G, σ and $\mathscr{D}$:*
- *if $G, C, \sigma \to_\mathscr{D} G', C', \sigma$, then $G, \mathsf{Amend}(C), \sigma \to^*_\mathscr{D} G', \mathsf{Amend}(C'), \sigma'$;*
- *if $G, \mathsf{Amend}(C), \sigma \to_\mathscr{D} G', C', \sigma'$, then there exist C'' and σ'' such that $G, C, \sigma \to_\mathscr{D} G', C'', \sigma''$ and $G', C', \sigma' \to^*_\mathscr{D} G', C'', \sigma''$.*

Proof (Theorem 9). (Completeness) and (Projectability) are immediate by definition of Amend. For (Correspondence), the proof is by analysis of the possible transitions. The only interesting cases occur when the transition consumes a conditional. In the case $G, C, \sigma \to G', C', \sigma$, then $\mathsf{Amend}(C)$ also has to consume the label selections introduced by amendment in the branch taken in order to reach $\mathsf{Amend}(C')$. Conversely, if $G, \mathsf{Amend}(C), \sigma$ makes a transition that consumes a conditional, then C' needs to consume the label selections introduced by amendment in order to match the corresponding move by C.

$$\frac{\mathsf{p} \overset{G}{\to} \mathsf{q} \quad e[\sigma(\mathsf{p})/*] \downarrow v}{G, \mathsf{p}.e \overset{x}{\text{->}} \bullet_{\mathsf{q}}; C, \sigma \;\to_{\mathscr{D}}\; G, C[v/x], \sigma} \lfloor \text{C}|\text{Com-S} \rceil$$

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

$$\frac{\mathsf{p} \overset{G}{\to} \mathsf{q}}{G, \mathsf{p} \overset{x}{\text{->}} \bullet_{\mathsf{q}}[l]; C, \sigma \;\to_{\mathscr{D}}\; G, C[l/x], \sigma} \lfloor \text{C}|\text{Sel-S} \rceil$$

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

$$\frac{\mathsf{p} \overset{G}{\to} \mathsf{q} \quad \mathsf{p} \overset{G}{\to} \mathsf{r}}{G, \mathsf{p} : \bullet_{\mathsf{q}} \overset{x,y}{\text{<->}} \bullet_{\mathsf{r}}; C, \sigma \;\to_{\mathscr{D}}\; G, C[\mathsf{q}/x, \mathsf{r}/y], \sigma} \lfloor \text{C}|\text{Tell-S} \rceil$$

$$\frac{\mathsf{q} \overset{G}{\to} \mathsf{p}}{G, \bullet_{\mathsf{p}}.\mathsf{r} \overset{\mathsf{r}}{\text{->}} \mathsf{q}; C, \sigma \;\to_{\mathscr{D}}\; G \cup \{\mathsf{q} \to \mathsf{r}\}, C, \sigma} \lfloor \text{C}|\text{Tell-R} \rceil$$

Fig. 18. Asynchronous PC, Semantics of New Runtime Terms.

This result also holds if we replace $\mathscr{D}$ by $\mathsf{Amend}(\mathscr{D})$ in the relevant places.

The first choreography in Remark 2, which is unprojectable, can be amended to the projectable choreography presented at the end of the same remark.

A.3 Asynchrony

We detail the whole sets of rules for the semantics of aPC (Figure 18) and its type system (Figure 19), as well as for the semantics of aPP (Figure 20). The full definition of EPP is given in Figures 21 and 22.

The proofs of the relationships between PC/PP and their asynchronous counterparts are mechanical.

Proof (Theorem 6). Straightforward by case analysis on the possible transitions of C.

Proof (Theorem 7). Straightforward by case analysis on the possible transitions of N.

A.4 Procedures with holes

We illustrate the use of holes to define a general-purpose iterator, procedure `Iter` below. Using `Iter`, we define a procedure `Reverse` for reversing the list of a process `q`. We omit the straightforward procedures for `dec`rementing a counter and `pop`ping a list.

$$\frac{\mathsf{p} \xrightarrow{G} \mathsf{q} \quad \Gamma \vdash \mathsf{p} : T_{\mathsf{p}} \quad * : T_{\mathsf{p}} \vdash_{\mathbb{T}} e : T \quad \Gamma \oplus (x : T); G \vdash C \triangleright G'}{\Gamma; G \vdash \mathsf{p}.e \xrightarrow{x} \bullet_{\mathsf{q}}; C \triangleright G'} \lfloor \text{T|Com-S} \rceil$$

$$\frac{\mathsf{q} \xrightarrow{G} \mathsf{p} \quad \Gamma \vdash \mathsf{q} : T_{\mathsf{q}} \quad * : T_{\mathsf{q}} \vdash_{\mathbb{T}} f : T \to T_{\mathsf{q}} \quad \Gamma \oplus (x : T); G \vdash C \triangleright G'}{\Gamma; G \vdash \bullet_{\mathsf{p}} \xrightarrow{x} \mathsf{q}.f; C \triangleright G'} \lfloor \text{T|Com-RV} \rceil$$

$$\frac{\mathsf{q} \xrightarrow{G} \mathsf{p} \quad \Gamma \vdash \mathsf{q} : T_{\mathsf{q}} \quad \vdash_{\mathbb{T}} v : T \quad * : T_{\mathsf{q}} \vdash_{\mathbb{T}} f : T \to T_{\mathsf{q}} \quad \Gamma; G \vdash C \triangleright G'}{\Gamma; G \vdash \bullet_{\mathsf{p}} \xrightarrow{v} \mathsf{q}.f; C \triangleright G'} \lfloor \text{T|Com-RT} \rceil$$

$$\frac{\mathsf{p} \xrightarrow{G} \mathsf{q} \quad \Gamma \oplus (x = l); G \vdash C \triangleright G'}{\Gamma; G \vdash \mathsf{p} \xrightarrow{x} \bullet_{\mathsf{q}}[l]; C \triangleright G'} \lfloor \text{T|Sel-S} \rceil$$

$$\frac{\mathsf{q} \xrightarrow{G} \mathsf{p} \quad \Gamma \oplus (x = l); G \vdash C \triangleright G'}{\Gamma; G \vdash \bullet_{\mathsf{p}} \xrightarrow{x} \mathsf{q}[l]; C \triangleright G'} \lfloor \text{T|Sel-RV} \rceil \qquad \frac{\mathsf{q} \xrightarrow{G} \mathsf{p} \quad \Gamma; G \vdash C \triangleright G'}{\Gamma; G \vdash \bullet_{\mathsf{p}} \xrightarrow{l} \mathsf{q}[l]; C \triangleright G'} \lfloor \text{T|Sel-RT} \rceil$$

$$\frac{\mathsf{p} \xrightarrow{G} \mathsf{q} \quad \mathsf{p} \xrightarrow{G} \mathsf{r} \quad \Gamma \oplus (x = \mathsf{q}) \oplus (y = \mathsf{r}) \vdash C \triangleright G'}{\Gamma; G \vdash \mathsf{p}\colon \bullet_{\mathsf{q}} \xleftrightarrow{x,y} \bullet_{\mathsf{r}}; C \triangleright G'} \lfloor \text{T|Tell-S} \rceil$$

$$\frac{\mathsf{q} \xrightarrow{G} \mathsf{p} \quad \Gamma; G \cup \{\mathsf{q} \to \mathsf{r}\} \vdash C \triangleright G'}{\Gamma; G \vdash \bullet_{\mathsf{p}}.\mathsf{r} \xrightarrow{\mathsf{r}} \mathsf{q}; C \triangleright G'} \lfloor \text{T|Tell-RV} \rceil$$

$$\frac{\mathsf{q} \xrightarrow{G} \mathsf{p} \quad \Gamma \oplus (x = \mathsf{r}); G \cup \{\mathsf{q} \to \mathsf{r}\} \vdash C \triangleright G'}{\Gamma; G \vdash \bullet_{\mathsf{p}}.\mathsf{r} \xrightarrow{x} \mathsf{q}; C \triangleright G'} \lfloor \text{T|Tell-RT} \rceil$$

Fig. 19. Asynchronous PC, Typing Rules (New Runtime Terms).

```
Iter(p,q,r) = if p.is_zero
  then p -> q,r[stop]
  else dec<p>; p -> q,r[cont]; □ₕ; Iter<p,q,r>

Reverse(q) =
  q starts p,r; q: p<->r;
  q.size -> p; q.empty -> r;
  Iter<p,q,r> with h ↦ q.top -> r.append; pop<q>;
  r.* -> q
```

$$\frac{\rho'_{\mathsf{q}} = \rho_{\mathsf{q}} \cdot \langle \mathsf{p}, e[v/*] \rangle}{\mathsf{p} \triangleright_v^{\rho_{\mathsf{p}}} \mathsf{q}!e; B_{\mathsf{p}} \mid \mathsf{q} \triangleright_w^{\rho_{\mathsf{q}}} B_{\mathsf{q}} \;\to_{\mathscr{B}}\; \mathsf{p} \triangleright_v^{\rho_{\mathsf{p}}} B_{\mathsf{p}} \mid \mathsf{q} \triangleright_w^{\rho'_{\mathsf{q}}} B_{\mathsf{q}}} \;\lfloor \text{P|Com-S} \rceil$$

$$\frac{\rho_{\mathsf{q}} \preceq \langle \mathsf{p}, v \rangle \cdot \rho'_{\mathsf{q}} \quad u = (f[w/*])(v)}{\mathsf{q} \triangleright_w^{\rho_{\mathsf{q}}} \mathsf{p}?f; B \;\to_{\mathscr{B}}\; \mathsf{q} \triangleright_u^{\rho'_{\mathsf{q}}} B} \;\lfloor \text{P|Com-R} \rceil$$

$$\frac{\rho'_{\mathsf{q}} = \rho_{\mathsf{q}} \cdot \langle \mathsf{p}, \mathsf{r} \rangle \quad \rho'_{\mathsf{r}} = \rho_{\mathsf{r}} \cdot \langle \mathsf{p}, \mathsf{q} \rangle}{\mathsf{p} \triangleright_v^{\rho_{\mathsf{p}}} \mathsf{q}!!\mathsf{r}; B_{\mathsf{p}} \mid \mathsf{q} \triangleright_w^{\rho_{\mathsf{q}}} B_{\mathsf{q}} \mid \mathsf{r} \triangleright_z^{\rho_{\mathsf{r}}} B_{\mathsf{r}} \;\to_{\mathscr{B}}\; \mathsf{p} \triangleright_v^{\rho_{\mathsf{p}}} B_{\mathsf{p}} \mid \mathsf{q} \triangleright_w^{\rho'_{\mathsf{q}}} B_{\mathsf{q}} \mid \mathsf{r} \triangleright_z^{\rho'_{\mathsf{r}}} B_{\mathsf{r}}} \;\lfloor \text{P|Tell-S} \rceil$$

$$\frac{\rho_{\mathsf{q}} \preceq \langle \mathsf{p}, \mathsf{r} \rangle \cdot \rho'_{\mathsf{q}}}{\mathsf{q} \triangleright_w^{\rho_{\mathsf{q}}} \mathsf{p}?\mathsf{r}; B \;\to_{\mathscr{B}}\; \mathsf{q} \triangleright_w^{\rho'_{\mathsf{q}}} B} \;\lfloor \text{P|Tell-R} \rceil$$

$$\frac{\rho'_{\mathsf{q}} = \rho_{\mathsf{q}} \cdot \langle \mathsf{p}, l \rangle}{\mathsf{p} \triangleright_v^{\rho_{\mathsf{p}}} \mathsf{q} \oplus l; B \mid \mathsf{q} \triangleright_w^{\rho_{\mathsf{q}}} B_{\mathsf{q}} \;\to_{\mathscr{B}}\; \mathsf{p} \triangleright_v^{\rho_{\mathsf{p}}} B \mid \mathsf{q} \triangleright_w^{\rho'_{\mathsf{q}}} B_{\mathsf{q}}} \;\lfloor \text{P|Sel-S} \rceil$$

$$\frac{\rho_{\mathsf{q}} \preceq \langle \mathsf{p}, l_j \rangle \cdot \rho'_{\mathsf{q}} \quad j \in I}{\mathsf{q} \triangleright_w^{\rho_{\mathsf{q}}} \mathsf{p}\&\{l_i : B_i\}_{i \in I} \;\to_{\mathscr{B}}\; \mathsf{q} \triangleright_w^{\rho'_{\mathsf{q}}} B_j} \;\lfloor \text{P|Sel-R} \rceil$$

Fig. 20. Asynchronous Procedural Processes, Semantics (New Rules).

$$[\![\mathsf{p}.e \overset{x}{\text{->}} \bullet_{\mathsf{q}}; C]\!]_{\mathsf{r}} = \begin{cases} \mathsf{q}!e; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases} \qquad [\![\bullet_{\mathsf{p}} \overset{\hat{v}}{\text{->}} \mathsf{q}.f; C]\!]_{\mathsf{r}} = \begin{cases} \mathsf{p}?f; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{p} \overset{x}{\text{->}} \bullet_{\mathsf{q}}[l]; C]\!]_{\mathsf{r}} = \begin{cases} \mathsf{q} \oplus l & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases} \qquad [\![\bullet_{\mathsf{p}} \overset{\hat{l}}{\text{->}} \mathsf{q}[l]; C]\!]_{\mathsf{r}} = \begin{cases} \mathsf{p}\&\{l : [\![C]\!]_{\mathsf{r}}\} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{p} : \bullet_{\mathsf{q}} \overset{x,y}{\text{<->}} \bullet_{\mathsf{r}}; C]\!]_{\mathsf{s}} = \begin{cases} \mathsf{q}!!\mathsf{r}; [\![C]\!]_{\mathsf{s}} & \text{if } \mathsf{s} = \mathsf{p} \\ [\![C]\!]_{\mathsf{s}} & \text{otherwise} \end{cases} \qquad [\![\bullet_{\mathsf{p}}.\mathsf{r} \overset{\hat{\mathsf{p}}}{\text{->}} \mathsf{q}; C]\!]_{\mathsf{s}} = \begin{cases} \mathsf{p}?\mathsf{r}; [\![C]\!]_{\mathsf{s}} & \text{if } \mathsf{s} = \mathsf{q} \\ [\![C]\!]_{\mathsf{s}} & \text{otherwise} \end{cases}$$

Fig. 21. Asynchronous PC, Behaviour Projection (New Rules).

$$(\!|\bullet_{\mathsf{p}} \overset{v}{\text{->}} \mathsf{q}.f; C|\!)_{\mathsf{r}} = \begin{cases} \langle \mathsf{p}, v \rangle \cdot (\!|C|\!)_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ (\!|C|\!)_{\mathsf{r}} & \text{otherwise} \end{cases} \qquad (\!|\bullet_{\mathsf{p}} \overset{l}{\text{->}} \mathsf{q}[l]; C|\!)_{\mathsf{r}} = \begin{cases} \langle \mathsf{p}, l \rangle \cdot (\!|C|\!)_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$(\!|\bullet_{\mathsf{p}}.\mathsf{r} \overset{\mathsf{r}}{\text{->}} \mathsf{q}; C|\!)_{\mathsf{s}} = \begin{cases} \langle \mathsf{p}, \mathsf{r} \rangle \cdot (\!|C|\!)_{\mathsf{s}} & \text{if } \mathsf{s} = \mathsf{q} \\ [\![C]\!]_{\mathsf{s}} & \text{otherwise} \end{cases}$$

$$(\!|\mathsf{if}\ \mathsf{p} <= \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C|\!)_{\mathsf{r}} = (\!|C_1|\!)_{\mathsf{r}} \cdot (\!|C|\!)_{\mathsf{r}} \qquad (\!|\eta; C|\!)_{\mathsf{r}} = (\!|I; C|\!)_{\mathsf{r}} = (\!|C|\!)_{\mathsf{r}}$$

Fig. 22. Asynchronous PC, State Projection.