Communications in Choreographies, Revisited

Luís Cruz-Filipe
Dept. of Mathematics and Computer Science, University of Southern Denmark
lcf@imada.sdu.dk

Fabrizio Montesi
Dept. of Mathematics and Computer Science, University of Southern Denmark
fmontesi@imada.sdu.dk

Marco Peressotti
Dept. of Mathematics and Computer Science, University of Southern Denmark
peressotti@imada.sdu.dk

ABSTRACT

Choreographic Programming is a paradigm for developing correct-by-construction concurrent programs, by writing high-level descriptions of the desired communications and then synthesising process implementations automatically. So far, choreographic programming has been explored in the *monadic* setting: interaction terms express point-to-point communications of a single value. However, real-world systems often rely on interactions of *polyadic* nature, where multiple values are communicated among two or more parties, like multicast, scatter-gather, and atomic exchanges.

We introduce a new model for choreographic programming equipped with a primitive for grouped interactions that subsumes all the above scenarios. Intuitively, grouped interactions can be thought of as being carried out as one single interaction. In practice, they are implemented by processes that carry them out in a concurrent fashion. After formalising the intuitive semantics of grouped interactions, we prove that choreographic programs and their implementations are correct and deadlock-free by construction.

CCS CONCEPTS

• **Theory of computation → Distributed computing models**; **Process calculi**; • **Software and its engineering** → *Concurrent programming languages*; *Concurrent programming structures*;

KEYWORDS

Choreography; Concurrency; Communication patterns

ACM Reference Format:
Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2018. Communications in Choreographies, Revisited. In *Proceedings of ACM SAC Conference (SAC'18).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

Choreographic Programming [15] is an emerging paradigm for programming communications in concurrent and distributed systems. The key idea is that programs are *choreographies*, which define the communications that we wish to take place from a global viewpoint, using structures inspired by the "Alice-and-Bob" notation for security protocols [17]. Then, an *EndPoint Projection* (EPP) synthesises a correct-by-construction implementation in process models (*e.g.* process calculi), guaranteeing important properties such as progress and operational correspondence [1, 2]. The applicability of the paradigm has been demonstrated in different settings, including service-oriented programming [1, 9], adaptable distributed software [8], cyber-physical systems [13, 14], and software verification [5].

Processes in choreographic programs typically interact via point-to-point message passing. This is expressed by language terms like $\mathsf{p}.e \rightarrow \mathsf{q}.y$, which reads "process p evaluates expression e locally and sends the result to process q, which stores the received value in its local variable y". However, there are application scenarios that require more advanced primitives. We mention two representative such scenarios. First, choreographic languages for cyber-physical systems offer primitives for scatter/gather communications (broadcast/reduce in their terminology) [13, 14]. Intuitively, this means generalising $\mathsf{p}.e \rightarrow \mathsf{q}.y$ to having many receivers (scatter, $\mathsf{p}.e \rightarrow \mathsf{q}_1.x_1 \ldots \mathsf{q}_n.x_n$) or many senders (gather, $\mathsf{q}_1.e_1 \ldots \mathsf{q}_n.e_n \rightarrow \mathsf{p}.f$). Second, choreographic languages for parallel computing and/or asynchronous communications support the idea of exchange [5]. For example, the term $(\mathsf{p}.x \rightarrow \mathsf{q}.y, \mathsf{q}.x \rightarrow \mathsf{p}.y)$ in [5] denotes the parallel exchange between processes p and q of their respective values locally stored in variable x. These scenarios illustrate the need for choreographic languages with more expressive primitives that capture multiple communications. However, the extensions proposed so far differ in their syntax and semantics, and none of them comes with an EndPoint Projection procedure. Hence, it is still unclear how the correctness-by-construction guarantee of choreographic programming can be extended to this kind of primitives.

In this paper, we tackle this issue by extending choreographic programming with language constructs for grouping *sets* of communications into complex group interactions, called *multicoms*. Our construct is unifying, in the sense that it captures both the scatter/gather and exchange patterns, as we exemplify here. (It actually is even more powerful, as we point out later.) Consider the following code snippet, a simple program that crawls stores searching for the best offer for a given item using the scatter/gather pattern:

```
{ p.(item, auth(p,s)) -> s.t | s ∈ S };
{ s.priceof(t) -> p.xs | s ∈ S }
```

In the first line, the search service p queries each store s in the collection S (being sets, multicoms lend themselves to set comprehensions). At first sight, this is essentially a multicast as in previous works, but observe that messages from the same sender are not necessarily the same: as shown by this example p attaches with information to authenticate itself to each receiver. (Hence our primitive is more expressive.) In the second line, responses are collected (and possibly aggregated) by p. Each step of the interaction between p and any given store s is causally dependent, whereas interactions with distinct stores are not (request-response interactions with different stores can proceed independently). So, for the first time in choreographies, our multicom captures both scatter and gather with a single primitive, but it is not limited to those patterns.

Consider now a scenario where two search services, say p_1 and p_2, run the search code above independently and then share their respective offers with each other (one can imagine this to be part of a purchase protocol where the service with the best offer then proceeds to buying the item). This exchange can be succinctly expressed as the following multicom.

$$3 \left\{ \begin{array}{l} \mathsf{p}_1.\mathtt{myoffer} \rightarrow \mathsf{p}_2.x \\ \mathsf{p}_2.\mathtt{myoffer} \rightarrow \mathsf{p}_1.x \end{array} \right\}$$

Programming with multicoms is easy, as we illustrated. Multicom dynamics should also be straightforward: intuitively, to the programmer's eyes, they are interactions among arbitrary groups of processes that are carried out as one. However, this is not quite what happens in reality, since we know that each communication in a multicom may proceed independently from the others whenever possible. To bridge this gap, we formalise both a simple semantics for choreographies with multicom—where statements are run sequentially and multicoms atomically—and a more realistic concurrent semantics—where all independent communications may be executed in any possible order. Then, we show that the concurrent semantics is a refinement of the simple one. This allows for results to be transferred between the two semantics. In particular, for the first time, it enables us to extend the correct-by-construction methodology of choreographic programming to this kind of structures. Namely, we define and prove correct an EndPoint Projection from choreographies to a concurrent process calculus.

2 CHOREOGRAPHY MODEL

Typically, there are two kinds of interaction primitives in choreographic programs: *(value) communications* and *(interface) selections* [6]. We shall maintain this distinction for groups of interactions, since each kind serves different mechanics, and also to avoid unnecessary technicalities. Concretely, we extend choreographies with constructs for grouping communications and for grouping selections called *multicoms* and *multisels*, respectively. In this section, we formalise their semantics and relevant properties.

2.1 Syntax

Terms describing choreographic programs (or choreographies, for short) are generated by the following grammar:

$$C ::= H;C \mid \Phi;C \mid \mathsf{if}\,\mathsf{p}.e\,\mathsf{then}\,C_1\,\mathsf{else}\,C_2 \mid \mathsf{def}\,X = C_2\,\mathsf{in}\,C_1 \mid X \mid \mathbf{0}$$
$$H ::= \{\eta_0, \ldots, \eta_n\} \qquad \eta ::= \mathsf{p}.e \rightarrow \mathsf{q}.y$$
$$\Phi ::= \{\phi_0, \ldots, \phi_n\} \qquad \phi ::= \mathsf{p} \rightarrow \mathsf{q}[\ell]$$

A choreography describes the behaviour of a (fixed) set of processes (denoted by p, q, *etc.*) running concurrently. Each process has a private memory made of named cells (denoted by x, y, *etc.*). We assume that each process has a dedicated full-duplex channel to communicate with each other process (*e.g.* a TCP/IP channel)—in other words, we assume an underlying full-duplex channel for each pair of processes.

Terms $H;C$ and $\Phi;C$ are (grouped) interactions and read "the system may execute H (resp. Φ) and proceed as C". An interaction group is either a (finite) set H of value communications (η-terms like $\mathsf{p}.e \rightarrow \mathsf{q}.y$) or a (finite) set Φ of interface selections (ϕ-terms like $\mathsf{p} \rightarrow \mathsf{q}[\ell]$). In $\mathsf{p}.e \rightarrow \mathsf{q}.y$, p sends its local evaluation of expression e to q, which stores the received value at y. The language of expressions is intentionally not fixed, for generality—we just assume that their evaluation always terminates, possibly through a timeout. In $\mathsf{p} \rightarrow \mathsf{q}[\ell]$, p communicates label ℓ (which is a constant) to q. If you like, labels are abstractions of operations, as in service-oriented computing, or methods, as in object-oriented computing. So in $\mathsf{p} \rightarrow \mathsf{q}[\ell]$, p requires q to proceed with the behaviour labelled by ℓ. In the remainder, we make the standard assumption that choreographies do not contain self-interactions (*e.g.* $\mathsf{p} \rightarrow \mathsf{p}[\ell]$).

Recall from the introduction that interactions grouped into multicoms and multisels should be thought as happening as one or, more precisely, concurrently. This intuition is reflected in the fact that multicoms and multisels are sets. As a consequence, interfering interactions cannot be grouped. The following is an example of a problematic multicom with interfering interactions:

$$\left\{ \begin{array}{l} \mathsf{p}.x \rightarrow \mathsf{q}.x \\ \mathsf{p}.y \rightarrow \mathsf{q}.y \\ \mathsf{r}.x \rightarrow \mathsf{q}.y \\ \mathsf{q}.y \rightarrow \mathsf{s}.x \end{array} \right\}$$

In the first two communications, q receives two values from p on the (distinct) variables x and y. However, the two operations are incompatible: each process can carry out actions inside multicoms in any order, but q cannot know which message will arrive first on its channel with p unless an order is agreed on (it is not in this case). In the implementation of this choreography, it may thus happen that q incorrectly stores in its local variable y the value of x at p. The second and third communications are also incompatible: q stores the content of the received messages in the same variable y, and hence the order in which messages are delivered to q cannot be ignored (even though the senders are distinct). Finally, the third and fourth communications are interfering, too, because the value sent by q in the fourth interaction may depend on whether the third interaction takes place before or after it, so we cannot interpret the result of the two interactions independently from the order in which they are executed.

These observations are formalised by the following syntactic conditions on H terms.

Definition 2.1. A set of communications H is a *(well-formed) multicom* if:

(1) if $\{\mathsf{p}.e \rightarrow \mathsf{q}.y,\ \mathsf{r}.e' \rightarrow \mathsf{q}.y'\} \subseteq H$, then $y \neq y'$ and $\mathsf{p} \neq \mathsf{r}$;
(2) if $\{\mathsf{p}.e \rightarrow \mathsf{q}.y,\ \mathsf{q}.e' \rightarrow \mathsf{r}.y'\} \subseteq H$, then $y \notin e'$.

Note that even if our model allowed for multiple separate channels between two processes, we would still need a requirement like

$\mathsf{p} \neq \mathsf{r}$ in the first condition—*i.e.* we would require inequality of channel names, rather than process names.

Similar observations hold for multisels, as illustrated by the following snippet with interfering selections:

$$\left\{ \begin{array}{c} \mathsf{p} \twoheadrightarrow \mathsf{q}[\ell] \\ \mathsf{r} \twoheadrightarrow \mathsf{q}[\ell'] \\ \mathsf{q} \twoheadrightarrow \mathsf{s}[\ell] \end{array} \right\}$$

Here, process q must concurrently select from an interface at s and await for p and r to select an interface each.

Definition 2.2. A set of selections Φ is a *(well-formed) multisel* provided that: if $\{\mathsf{p} \twoheadrightarrow \mathsf{q}[\ell], \mathsf{r} \twoheadrightarrow \mathsf{s}[\ell']\} \subseteq \Phi$, then $\mathsf{q} \notin \{\mathsf{r}, \mathsf{s}\}$.

In the remainder, we assume that all multicoms and multisels are well-formed. Also, we often abuse notation by writing $\eta;C$ and $\phi;C$ instead of $\{\eta\};C$ and $\{\phi\};C$, respectively.

The remaining choreographic primitives are standard. In a conditional $\mathsf{if}\,\mathsf{p}.e\,\mathsf{then}\,C_1\,\mathsf{else}\,C_2$, the guard e is evaluated in the context of p and then the choreography proceeds executing either branch accordingly—expressions are implicitly assumed to support Boolean values or some equivalent mechanism. Definitions and invocations of recursive procedures are standard. The term $\mathbf{0}$ is the terminated choreography. As common practice, we assume that all procedure invocations refer to defined procedures, and omit $\mathbf{0}$ whenever clear from the surrounding terms.

2.2 Sequential semantics

We now give a semantics to choreographies, which formalises their intuitive dynamics. The semantics is the smallest relation $\to$ between pairs (C, σ) where

- C is a choreography term as defined in Section 2.1;
- σ is a function describing the memory of processes (*i.e.* taking processes and their variable names to values);

that is closed under the rules in Figure 1. Before we discuss each rule observe that:

- reduction rules consume the outermost interaction (note that the outermost term may be a recursive definition);
- interactions are consumed in a single step;
- multicoms are reduced in a single step as if all their interactions were carried out by the involved processes simultaneously.

In this sense, this semantics is called sequential or big-step.

For compactness, the presentation relies on the structural precongruence $\preceq$ via the standard mechanism of rule $\lfloor$C|Str$\rceil$; the relation is defined as the smallest relation on choreographic programs closed under rules $\lfloor$C|Unfold$\rceil$, $\lfloor$C|DefNil$\rceil$ and $\lfloor$C|MEmpty$\rceil$ (discussed below). Herein, $C \equiv C'$ is a shorthand for $C \preceq C'$ and $C' \preceq C$.

The semantics of interactions is defined by rules $\lfloor$C|MCom$\rceil$ and $\lfloor$C|MSel$\rceil$. The expression of each communication $\mathsf{p}.e \twoheadrightarrow \mathsf{q}.y$ in the multicom H is evaluated in the sender context ($e \downarrow_{\sigma(\mathsf{p})} v$) and the resulting value (v) is used in the *reductum* to update the receiver memory, independently. Selections have no effect on process memory. In both cases, the set of communications is required to be non-empty, in order to avoid reductions that do not correspond to any action. The cases $H = \emptyset$ or $\Phi = \emptyset$ are instead dealt with by structural precongruence (rule $\lfloor$C|MEmpty$\rceil$). The semantics of conditionals is modelled by rule $\lfloor$C|If$\rceil$, where the guard is evaluated in the context of the process p performing the choice, and then the program reduces to the corresponding branch. Recursive definitions are implicitly expanded by structural precongruence, as described by rule $\lfloor$C|Unfold$\rceil$: here, $C_1[X]$ indicates that the term X occurs in C_1, and on the term on the righthandside it is replaced by the body of the recursive definition. Rule $\lfloor$C|Ctx$\rceil$ is standard and necessary to reduce the outermost interaction. Rule $\lfloor$C|DefNil$\rceil$ collects recursive definitions from terminated programs, *i.e.* any C s.t. $C \preceq \mathbf{0}$.

Remark 2.3. Label selections do not alter the state of any process, since they simply model the choice of a possible behaviour offered by the receiver. We will use this information to synthesise appropriate interfaces for our processes in Section 3.3. This is a standard method in choreographic programming, but we mention it here for the unfamiliar reader.

For example, consider the following choreography.

```
if p.e then
   p ->> q[L]; p.x ->> q.y
 else
   p ->> q[R]; q.y ->> p.x
```

Here, p makes a local choice and depending on the result selects the appropriate behaviour at q. In the first case, represented by label L, q is expected to receive a value from p. In the second case, represented by label R, q is expected to send a value to p. Without label selections, q would not know how to act, since only p would know which branch has been selected in the choreographic conditional. In Section 3.3, we detail how to synthesise appropriate interfaces for processes such as q in this example.

Choreographic programs never get stuck: they are either successfully terminated or able to reduce.

Theorem 2.4. *For C a choreography, either*

(1) $C \preceq \mathbf{0}$; or,
(2) for every σ there are C' and σ' such that $C, \sigma \to C', \sigma'$.

The sequential semantics of choreographic programs enjoys local confluence, which intuitively states that, whenever a computation can proceed in more than one way, it is always possible to reach a common configuration. Formally:

Theorem 2.5. *For every span of computations $C_0, \sigma_0 \to C_1, \sigma_1$ and $C_0, \sigma_0 \to C_2, \sigma_2$ there are C_3 and σ_3 such that $C_1, \sigma_1 \to^* C_3, \sigma_3$ and $C_2, \sigma_2 \to^* C_3, \sigma_3$, where $\to^*$ is the transitive and reflexive closure of $\to$.*

2.3 Concurrent semantics

This section refines the sequential semantics of choreographic programs, redefining the semantics of grouped interactions to allow primitive interactions to proceed independently whenever possible. When necessary, we distinguish reductions ($\to$) and structural precongruence ($\preceq$) defining the sequential semantics (Section 2.2) and concurrent semantics (Section 2.3) adding subscripts s and c, respectively.

$$\frac{H \neq \emptyset \qquad H = \{\mathsf{p}_i.e_i \rightarrow \mathsf{q}_i.y_i \mid i \in I\} \qquad e_i \downarrow_{\sigma(\mathsf{p}_i)} v_i}{H;C,\sigma \longrightarrow C,\sigma[\mathsf{q}_i.y_i \mapsto u_i]}\lfloor\text{C|MCom}\rceil \qquad \frac{\Phi \neq \emptyset}{\Phi;C,\sigma \longrightarrow C,\sigma}\lfloor\text{C|MSel}\rceil$$

$$\frac{i = 1 \text{ if } e \downarrow_{\sigma(\mathsf{p})} \mathtt{true},\ i = 2 \text{ otherwise}}{\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2,\sigma \longrightarrow C_i,\sigma}\lfloor\text{C|If}\rceil \qquad \frac{C_1 \preceq C_2 \qquad C_2,\sigma \longrightarrow C_2',\sigma' \qquad C_2' \preceq C_1'}{C_1,\sigma \longrightarrow C_1',\sigma'}\lfloor\text{C|Str}\rceil$$

$$\frac{C_1,\sigma \longrightarrow C_1',\sigma'}{\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1,\sigma \longrightarrow \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1',\sigma'}\lfloor\text{C|Ctx}\rceil \qquad \frac{}{\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1[X] \preceq \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1[C_2]}\lfloor\text{C|Unfold}\rceil$$

$$\frac{}{\mathsf{def}\ X = C\ \mathsf{in}\ \mathbf{0} \preceq \mathbf{0}}\lfloor\text{C|DefNil}\rceil \qquad \frac{}{\{\};C \preceq C}\lfloor\text{C|MEmpty}\rceil$$

Figure 1: Sequential semantics of choreographic programs.

$$\frac{e \downarrow_{\sigma(\mathsf{p}_i)} v}{\mathsf{p}.e \rightarrow \mathsf{q}.y;C,\sigma \longrightarrow C,\sigma[\mathsf{q}.y \mapsto v]}\lfloor\text{C|Com}\rceil \qquad \frac{}{\mathsf{p} \rightarrow \mathsf{q}[\ell];C,\sigma \longrightarrow C,\sigma}\lfloor\text{C|Sel}\rceil$$

$$\frac{H_0 = H_1 \uplus H_2}{H_0 \equiv H_1;H_2}\lfloor\text{C|MCom-MCom}\rceil \qquad \frac{\Phi_0 = \Phi_1 \uplus \Phi_2}{\Phi_0 \equiv \Phi_1;\Phi_2}\lfloor\text{C|MSel-MSel}\rceil \qquad \frac{\mathrm{tn}(\Phi) \cap \mathrm{pn}(H) = \emptyset}{H;\Phi \equiv \Phi;H}\lfloor\text{C|MCom-MSel}\rceil$$

$$\frac{x_i \notin e \text{ for all } \mathsf{q}_i.e_i \rightarrow \mathsf{p}.x_i \in H}{\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ (H;C_1)\ \mathsf{else}\ (H;C_2) \equiv H;\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2}\lfloor\text{C|MCom-If}\rceil \qquad \frac{\mathsf{q} \rightarrow \mathsf{p}[\ell] \notin \Phi}{\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ (\Phi;C_1)\ \mathsf{else}\ (\Phi;C_2) \equiv \Phi;\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2}\lfloor\text{C|MSel-If}\rceil$$

$$\frac{}{\mathsf{def}\ X = C_2\ \mathsf{in}\ (H;C_1) \equiv H;\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1}\lfloor\text{C|MCom-Rec}\rceil \qquad \frac{}{\mathsf{def}\ X = C_2\ \mathsf{in}\ (\Phi;C_1) \equiv \Phi;\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1}\lfloor\text{C|MSel-Rec}\rceil$$

$$\frac{}{\mathsf{if}\ \mathsf{p}.e_1\ \mathsf{then}\ (\mathsf{if}\ \mathsf{q}.e_2\ \mathsf{then}\ C_1^1\ \mathsf{else}\ C_2^1)\ \mathsf{else}\ (\mathsf{if}\ \mathsf{q}.e_2\ \mathsf{then}\ C_1^2\ \mathsf{else}\ C_2^2) \equiv \mathsf{if}\ \mathsf{q}.e_2\ \mathsf{then}\ (\mathsf{if}\ \mathsf{p}.e_1\ \mathsf{then}\ C_1^1\ \mathsf{else}\ C_1^2)\ \mathsf{else}\ (\mathsf{if}\ \mathsf{p}.e_1\ \mathsf{then}\ C_2^1\ \mathsf{else}\ C_2^2)}\lfloor\text{C|If-If}\rceil$$

Figure 2: Concurrent semantics of choreographic programs—new rules.

The semantics is defined by the rules in Figure 2 together with all rules in Figure 1 except for rules $\lfloor\text{C|MCom}\rceil$ and $\lfloor\text{C|MSel}\rceil$.

Rules $\lfloor\text{C|Com}\rceil$ and $\lfloor\text{C|Sel}\rceil$ describe the semantics of primitive interactions between two processes as a specialisation of rules $\lfloor\text{C|MCom}\rceil$ and $\lfloor\text{C|MSel}\rceil$ to $\{\eta\};C$ and $\{\phi\};C$, respectively. Rules $\lfloor\text{C|MCom-MCom}\rceil$ and $\lfloor\text{C|MSel-MSel}\rceil$ state that groups of interactions can be merged and split at runtime—as long as they are well-formed. Merging may not always be possible, since merging interactions from distinct groups may violate well-formedness. (In other words, not all interactions can be rescheduled and performed concurrently due to causal dependencies.) In the opposite direction, it is always possible to split groups, and hence freely schedule their interactions.

In rule $\lfloor\text{C|MCom-MSel}\rceil$, $\mathrm{tn}(\Phi)$ and $\mathrm{pn}(H)$ are the sets of process names that occur as selection targets $\{\mathsf{q} \mid \mathsf{p} \rightarrow \mathsf{q}[\ell] \in \Phi\}$ and that occur in a communication $\{\mathsf{p},\mathsf{q} \mid \mathsf{p}.e \rightarrow \mathsf{q}.y \in H\}$, respectively. The rule states that value communications and interface selection can be freely scheduled as long as selection targets are not involved in any other communication. Rules $\lfloor\text{C|MCom-If}\rceil$ and $\lfloor\text{C|MSel-If}\rceil$ state that conditionals and interactions can be swapped as long as the process evaluating the guard is neither the target of an interface selection nor receives a value in a variable that is used by the guard. The remaining rules are straightforward.

Example 2.6. Consider the following program:

$$\left\{\begin{array}{l}\mathsf{p}.e_0 \rightarrow \mathsf{s}_0.y_0\\ \mathsf{p}.e_1 \rightarrow \mathsf{s}_1.y_1\end{array}\right\};\left\{\begin{array}{l}\mathsf{s}_0.e_0' \rightarrow \mathsf{p}.x_0\\ \mathsf{s}_1.e_1' \rightarrow \mathsf{p}.x_1\end{array}\right\}$$

Assuming that e_i' depends on y_i, this program is a minimal example of the same scatter-gather pattern described in Section 1. By immediate applications of rule $\lfloor\text{C|MCom-MCom}\rceil$, the program is structurally equivalent to *e.g.*:

$$\left\{\ \mathsf{p}.e_0 \rightarrow \mathsf{s}_0.y_0\ \right\};\left\{\begin{array}{l}\mathsf{p}.e_1 \rightarrow \mathsf{s}_1.y_1\\ \mathsf{s}_0.e_0' \rightarrow \mathsf{p}.x_0\end{array}\right\};\left\{\ \mathsf{s}_1.e_1' \rightarrow \mathsf{p}.x_1\ \right\}$$

$$\left\{\ \mathsf{p}.e_0 \rightarrow \mathsf{s}_0.y_0\ \right\};\left\{\ \mathsf{s}_0.e_0' \rightarrow \mathsf{p}.x_0\ \right\};\left\{\ \mathsf{p}.e_1 \rightarrow \mathsf{s}_1.y_1\ \right\};$$
$$\left\{\ \mathsf{s}_1.e_1' \rightarrow \mathsf{p}.x_1\ \right\}$$

In fact, all these programs yield the very same set of executions: every possible linearisation of the partial order:

$$\begin{tikzcd}
\mathsf{s}_0.e_0' \rightarrow \mathsf{p}.x_0 \arrow[d, dash] & \mathsf{s}_1.e_1' \rightarrow \mathsf{p}.x_1 \arrow[d, dash] \\
\mathsf{p}.e_0 \rightarrow \mathsf{s}_0.y_0 & \mathsf{p}.e_1 \rightarrow \mathsf{s}_1.y_1
\end{tikzcd}$$

□

Observe that every (well-formed) multicom is equivalent to any sequence of its interactions:

$$\{\eta_1, \ldots, \eta_n\} \equiv \eta_1; \ldots; \eta_n.$$

Therefore, the semantics of multicoms as per rules $\lfloor$C|MCom$\rceil$ and $\lfloor$C|Com$\rceil$ are classified as big- and small-step—likewise for $\rightarrow_s$ and $\rightarrow_c$. The two are related since the former subsumes the latter (once multicoms are "sequentialised") and, conversely, the latter subsumes the former once reductions described by rule $\lfloor$C|Com$\rceil$ are transitively aggregated. The same holds for multisels. We can generalise these observations into a formal relation between our two semantics.

Lemma 2.7. *For any choreography C_0 and state σ_0:*

(1) if $C_0, \sigma_0 \rightarrow_s C_1, \sigma_1$, then $C_0, \sigma_0 \rightarrow_c^+ C_1, \sigma_1$;
(2) if $C_0, \sigma_0 \rightarrow_c C_1, \sigma_1$, then there are C_2 and σ_2 s.t. $C_1, \sigma_1 \rightarrow_c^* C_2, \sigma_2$ and $C_0, \sigma_0 \rightarrow_s^+ C_2, \sigma_2$;

where $(-)^+$ is the transitive closure operator.

The notion of operational correspondence used in Lemma 2.7 is slightly stronger than that studied for process calculi [10]. The latter is commonly used to compare reduction semantics and organise them as refinements and abstractions, but does not preserve and reflect progress. Instead, Lemma 2.7 immediately allows us to infer progress for $\rightarrow_c$.

Theorem 2.8. *For C a choreography, either*

(1) $C \preceq_c \mathbf{0}$; or,
(2) for every σ there are C' and σ' such that $C, \sigma \rightarrow_c C', \sigma'$.

Another consequence of Lemma 2.7 is that the concurrent semantics inherits confluence from the sequential one.

Theorem 2.9. *For every span of computations $C_0, \sigma_0 \rightarrow_c C_1, \sigma_1$ and $C_0, \sigma_0 \rightarrow_c C_2, \sigma_2$ there are C_3 and σ_3 such that $C_1, \sigma_1 \rightarrow_c^* C_3, \sigma_3$ and $C_2, \sigma_2 \rightarrow_c^* C_3, \sigma_3$.*

3 PROCESS MODEL

In this section, we show that our choreography model supports the correctness-by-construction approach of choreographic programming. We first introduce our process calculus for modelling concurrent processes, and then define an EndPoint Projection (EPP) that synthesises correct process code from choreographies.

3.1 Syntax

Terms describing process networks are generated by the grammar below.

$$\begin{aligned} N &::= \mathsf{p} \triangleright B \mid \mathbf{0} \mid N_1 \mid N_2 \\ B &::= \{\theta_1, \ldots, \theta_n\}; B \mid \{\ \mathsf{q}_i \oplus \ell_i\ \}_{i \in I}; B \mid \mathsf{p}\ \&\ \{\ell_i \colon B_i\}_{i \in I} \mid \\ &\quad \mid \mathsf{if}\ e\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2 \mid \mathsf{def}\ X = B_2\ \mathsf{in}\ B_1 \mid X \mid \mathbf{0} \\ \theta &::= \mathsf{q}!\langle e \rangle \mid \mathsf{q}?x \end{aligned}$$

Networks, ranged over by N, are either the inactive network $\mathbf{0}$, processes $\mathsf{p} \triangleright B$, where p is the name of the process and B its behaviour, or parallel compositions. A term $\{\theta_1, \ldots, \theta_n\}; B$ represents a behaviour where multiple sends and receives (θ-terms) are executed concurrently (and thus can be scheduled freely), before proceeding with the continuation B. In particular, $\mathsf{q}!\langle e \rangle$ describes a send operation where the process evaluates (locally) the expression e and sends the outcome to q. Symmetrically, a term $\mathsf{p}?y$ represents a receive operation, where the executing process receives a value from p and stores it in y. A term $\{\mathsf{q}_i \oplus \ell_i\}_{i \in I}; B$ concurrently sends many label selections (each ℓ_i is sent to the respective process q_i) before proceeding with B. The dual operation is the branching term $\mathsf{p}\ \&\ \{\ell_i \colon B_i\}_{i \in I}$, where we await to receive from p the selection of one of the labels ℓ_i and then perform the corresponding behaviour B_i. In a conditional $\mathsf{if}\ e\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2$, the process evaluates the guard e locally and chooses between the continuations B_1 and B_2 accordingly. The remaining terms are standard. We implicitly allow for exchange in the subterms $\{\theta_1, \ldots, \theta_n\}$, $\{\mathsf{q}_i \oplus \ell_i\}_{i \in I}$, and $\{\ell_i \colon B_i\}_{i \in I}$—order does not matter.

3.2 Semantics

The semantics for process networks is given in Figure 3. The key difference with respect to the semantics for choreographic programs is that execution is now distributed: processes progress concurrently and synchronise only when they interact. Network semantics is presented in terms of a relation between pairs of networks and memory configurations $N, \sigma \rightarrow N', \sigma'$.

Communication semantics is defined by rule $\lfloor$P|Com$\rceil$, which synchronises an output action of a process (p in the rule) with an input action at the intended receiver (q in the rule). Specifically, if there are a "send-to-q" term ($\mathsf{q}!\langle e \rangle$) in p's group of currently enabled actions and a "receive-from-p" term ($\mathsf{p}?y$) in q's group of currently enabled actions, then p can send the evaluation of e to q, and the latter updates its local state accordingly ($\sigma[\mathsf{q}.y \mapsto v]$). Similarly, rule $\lfloor$P|Sel$\rceil$ synchronises a process that wishes to select a branch with the process that offers it. Semantics of conditionals is defined by rule $\lfloor$P|If$\rceil$ and is entirely local: the process p performing the choice evaluates the guard e and executes either branch accordingly. The remaining rules are standard (*cf.* [6]).

3.3 EndPoint Projection

Given a choreographic program C, we can translate the behaviour of each process p defined in C into our process model. We write $[\![C]\!]_{\mathsf{p}}$ for this translation, which is defined by structural recursion by the rules in Figure 4. All rules follow the intuition of projecting, for each choreography term, the local action performed by the given process.

Building on $[\![C]\!]_{\mathsf{p}}$, we define the EndPoint Projection of a choreography (EPP) as the parallel composition of the behaviours obtained projecting each process separately.

Definition 3.1. The *EndPoint Projection* (EPP) $[\![C]\!]$ of a choreography C is the parallel composition:

$$[\![C]\!] \triangleq \prod_{\mathsf{p} \in \mathsf{pn}(C)} \mathsf{p} \triangleright [\![C]\!]_{\mathsf{p}}.$$

The key new rules for EPP introduced in this work are the ones for projecting multicoms and multisels.

A multicom term is projected to a group of concurrent send and receive operations, depending on the role interpreted by the given process in each interaction. We illustrate this construction in the following example, where we display the choreography that we are projecting on the left and its EPP on the right.

$$\frac{e \downarrow_{\sigma(\mathsf{p})} v \qquad \Theta = \{\mathsf{q}!\langle e\rangle, \theta_1, \ldots, \theta_n\} \qquad \Theta' = \left\{\mathsf{p}?x, \theta'_1, \ldots, \theta'_m\right\}}{\mathsf{p} \triangleright \Theta; B \mid \mathsf{q} \triangleright \Theta'; B', \sigma \longrightarrow \mathsf{p} \triangleright \{\theta_1, \ldots, \theta_n\}; B \mid \mathsf{q} \triangleright \left\{\theta'_1, \ldots, \theta'_m\right\}; B', \sigma[\mathsf{q}.y \mapsto v]}\lfloor\text{P}|\text{Com}\rceil$$

$$\frac{g \in I \cap J}{\mathsf{p} \triangleright \{\mathsf{q}_i \oplus \ell_i\}_{i\in I}; B \mid \mathsf{q}_g \triangleright \mathsf{p}\ \&\ \{\ell_j\colon B_j\}_{j\in J}, \sigma \longrightarrow \mathsf{p} \triangleright \{\mathsf{q}_i \oplus \ell_i\}_{i\in I\setminus\{g\}}; B \mid \mathsf{q}_g \triangleright B_g, \sigma}\lfloor\text{P}|\text{Sel}\rceil \qquad \frac{i = 1 \text{ if } e \downarrow_{\sigma(\mathsf{p})} \mathtt{true}, i = 2 \text{ otherwise}}{\mathsf{p} \triangleright \mathsf{if}\, e\, \mathsf{then}\, B_1\, \mathsf{else}\, B_2, \sigma \longrightarrow B_i, \sigma}\lfloor\text{P}|\text{If}\rceil$$

$$\frac{\mathsf{p} \triangleright B_1 \mid N, \sigma \longrightarrow \mathsf{p} \triangleright B'_1 \mid N', \sigma'}{\mathsf{p} \triangleright \mathsf{def}\, X = B_2\, \mathsf{in}\, B_1 \mid N, \sigma \longrightarrow \mathsf{p} \triangleright \mathsf{def}\, X = B_2\, \mathsf{in}\, B'_1 \mid N', \sigma'}\lfloor\text{P}|\text{Ctx}\rceil \qquad \frac{N, \sigma \longrightarrow N', \sigma'}{N \mid M, \sigma \longrightarrow N' \mid M, \sigma'}\lfloor\text{P}|\text{Par}\rceil$$

$$\frac{N \preceq M \qquad M \longrightarrow M' \qquad M' \preceq N'}{N \longrightarrow N'}\lfloor\text{P}|\text{Str}\rceil \qquad \frac{}{\mathsf{p} \triangleright \mathbf{0} \preceq \mathbf{0}}\lfloor\text{P}|\text{ProcNil}\rceil \qquad \frac{}{\mathbf{0} \mid N \preceq N}\lfloor\text{P}|\text{ParNil}\rceil \qquad \frac{}{\mathsf{def}\, X = B\, \mathsf{in}\, \mathbf{0} \preceq \mathbf{0}}\lfloor\text{P}|\text{DefNil}\rceil$$

$$\frac{}{\mathsf{def}\, X = B_2\, \mathsf{in}\, B_1[X] \preceq \mathsf{def}\, X = B_2\, \mathsf{in}\, B_1[B_2]}\lfloor\text{P}|\text{Unfold}\rceil \qquad \frac{}{\{\}; B \preceq B}\lfloor\text{P}|\text{MEmpty}\rceil$$

Figure 3: Process semantics.

$$[\![\mathbf{0}]\!]_\mathsf{r} \triangleq \mathbf{0} \qquad [\![H; C]\!]_\mathsf{r} \triangleq \left\{ \begin{array}{c|c} \mathsf{q}!\langle e\rangle & \mathsf{r}.e \rightarrow \mathsf{q}.y \in H \\ \mathsf{p}?y & \mathsf{p}.e \rightarrow \mathsf{r}.y \in H \end{array} \right\}; [\![C]\!]_\mathsf{r} \qquad [\![\Phi; C]\!]_\mathsf{r} \triangleq \begin{cases} \mathsf{p}\ \&\ \left\{\ell\colon [\![C]\!]_\mathsf{r}\right\} & \text{if } \mathsf{p} \rightarrow \mathsf{r}[\ell] \in \Phi \\ \{\mathsf{q} \oplus \ell \mid \mathsf{r} \rightarrow \mathsf{q}[\ell] \in \Phi\}; [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{if}\, \mathsf{p}.e\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2]\!]_\mathsf{r} \triangleq \begin{cases} \mathsf{if}\, e\, \mathsf{then}\, [\![C_1]\!]_\mathsf{r}\, \mathsf{else}\, [\![C_2]\!]_\mathsf{r} & \text{if } \mathsf{p} = \mathsf{r} \\ [\![C_1]\!]_\mathsf{r} \sqcup [\![C_2]\!]_\mathsf{r} & \text{otherwise} \end{cases} \qquad \left[\!\!\left[\mathsf{def}\, X^{\vec{\mathsf{p}}} = C_2\, \mathsf{in}\, C_1\right]\!\!\right]_\mathsf{r} \triangleq \begin{cases} \mathsf{def}\, X = [\![C_2]\!]_\mathsf{r}\, \mathsf{in}\, [\![C_1]\!]_\mathsf{r} & \text{if } \mathsf{r} \in \vec{\mathsf{p}} \\ [\![C_1]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$\left[\!\!\left[X^{\vec{\mathsf{p}}}\right]\!\!\right]_\mathsf{r} \triangleq \begin{cases} X & \text{if } \mathsf{r} \in \vec{\mathsf{p}} \\ \mathbf{0} & \text{otherwise} \end{cases}$$

Figure 4: Behaviour projection.

$$\left\{ \begin{array}{l} \mathsf{p}.x \rightarrow \mathsf{q}.y \\ \mathsf{q}.x \rightarrow \mathsf{p}.y \\ \mathsf{r}.z \rightarrow \mathsf{p}.x \end{array} \right\}; \qquad \begin{array}{l} \mathsf{p} \triangleright \{\mathsf{q}!\langle x\rangle, \mathsf{q}?y\}; \{\mathsf{r}?x\} \\ \mathsf{q} \triangleright \{\mathsf{p}!\langle x\rangle, \mathsf{p}?y\} \\ \mathsf{r} \triangleright \{\mathsf{p}!\langle z\rangle\} \end{array}$$

Multiple selections are handled likewise: a multisel is projected either to a group of selections or to a branch, depending on the role of the given process (recall that if a multisel is well-formed, then processes cannot occur as selection objects and subjects at the same time).

All remaining rules for EPP are (up to minor differences) standard [6]. The rules for projecting recursive definitions and calls assume that procedure names have been annotated with the process names appearing inside the body of the procedure, in order to avoid projecting unnecessary procedure code (*cf.* [1]).

The rule for projecting a conditional is more involved. The (partial) merging operator $\sqcup$ from [1] is used to merge the behaviour of a process that does not know which branch has been chosen yet: $B \sqcup B'$ is isomorphic to B and B' up to branching, where the branches of B or B' with distinct labels are also included. As an example, consider the following choreography and the projection of its processes.

$$\begin{array}{l} \mathsf{if}\, \mathsf{p}.e\, \mathsf{then}\, \mathsf{p} \rightarrow \mathsf{q}[\textsc{l}]; \\ \qquad\qquad \mathsf{p}.x \rightarrow \mathsf{q}.x \\ \quad\ \mathsf{else}\, \mathsf{p} \rightarrow \mathsf{q}[\textsc{r}]; \\ \qquad\qquad \mathsf{q}.y \rightarrow \mathsf{p}.y \end{array} \qquad \begin{array}{l} \mathsf{p} \triangleright\ \mathsf{if}\, e\, \mathsf{then}\, \mathsf{q} \oplus \textsc{l}; \mathsf{q}!\langle x\rangle \\ \qquad\quad \mathsf{else}\, \mathsf{q} \oplus \textsc{r}; \mathsf{q}?y \\ \mathsf{q} \triangleright \mathsf{p}\ \&\ \{\textsc{l}\colon \mathsf{p}?x, \textsc{r}\colon \mathsf{p}!\langle y\rangle\} \end{array}$$

Here, merging allows the projection of q to account for the different possible behaviours based on the label received from p.

If the choreography did not include a selection from p to q, then q would not know which choice p had made in evaluating its condition (*cf.* Remark 2.3). This aspect is typical of choreography models [1, 2, 4, 8, 11, 18]. More specifically, while the originating choreography executes correctly, its projection needs processes that behave differently in the branches of a conditional to be informed through a selection (either directly or indirectly, by receiving a selection from a previously notified process).

Observe that merging is partial and thus there are choreographies whose processes cannot be projected. These are not corner cases but actual programming errors that may appear even in simple programs like the following one:

$$\mathsf{if}\, \mathsf{p}.e\, \mathsf{then}\, \mathsf{p}.e' \rightarrow \mathsf{q}.x\, \mathsf{else}\, \mathbf{0}$$

In this case, the behaviour of process q cannot be projected because q does not know whether it should wait for a message from p or not. In general, projections are undefined whenever choices operated locally are not correctly propagated to all involved processes; explicit selections are thus instrumental to catching such errors at projection time, *i.e.* statically. Since merging is partial, $[\![C]\!]_\mathsf{p}$ may be undefined, and consequently $[\![C]\!]$ is also partial. In the remainder, we say that a choreography C is projectable if $[\![C]\!]$ is defined.

Example 3.2. The projection of the choreographic program discussed in Section 1 is the parallel composition of:

$$
\begin{array}{l}
\quad \mathsf{p} \triangleright \left\{\ \mathsf{s}!\langle(\mathtt{item},\ \mathtt{auth(p,s)})\rangle \ \middle|\ \mathsf{s} \in S\ \right\}; \\
\qquad \left\{\ \mathsf{s}?x_{\mathsf{s}} \ \middle|\ \mathsf{s} \in S\ \right\} \\
\prod_{\mathsf{s}\in S} \mathsf{s} \triangleright \mathsf{p}?t; \mathsf{p}!\langle \mathtt{priceof}(t)\rangle
\end{array}
$$

3.4 Properties

We end our technical discussion by showing that our framework supports the hallmark correctness-by-construction property of choreographic programming. Formally, this is achieved by proving that a choreography and its EPP are in an operational correspondence (they mimic each other); as a corollary, we obtain that the EPP of a choreography is deadlock-free.

In our setting, proving an operational correspondence result for EPP is more interesting than in previous work on choreographic programming, because we have two semantics for choreographies (the sequential relation $\to_s$ and the concurrent relation $\to_c$). Ideally, we would like to get an operational correspondence result for each choreographic semantics. A naive way of proceeding would be to prove the result twice, once for $\to_s$ and once for $\to_c$. But since we know that $\to_s$ and $\to_c$ are related (Lemma 2.7), we can do better.

First, we prove the following lemma. We again write $\to^+$ for one or more applications of $\to$.

Lemma 3.3. *If C is projectable, then:*

1. *if $C, \sigma \to_s C', \sigma'$ then, there is N such that $[\![C']\!] \sqsubset N$ and $[\![C]\!], \sigma \to^+ N, \sigma'$;*
2. *if $[\![C]\!], \sigma \to N, \sigma'$ then, there is C' such that $[\![C']\!] \sqsubset N$ and $[\![C]\!], \sigma \to_c [\![C']\!], \sigma'$.*

Above, the pruning relation $\sqsubset$ (from [1, 2]) drops branches introduced by the merging operator $\sqcup$ when they are no longer needed to follow the originating choreography. Pruning is completely orthogonal to our development, so we refer to [1] for a detailed explanation.

The choice of $\to_s$ and $\to_c$, respectively, for the two directions in Lemma 3.3 is strategic. Namely, for the first direction, considering $\to_s$ is easier because it is a simpler semantics, and it is then straightforward to show that the EPP of the choreography can implement, for example, a multicom by executing all its projected process actions. Conversely, for the second direction, using $\to_c$ is convenient because it allows us to execute exactly the single move performed by the projected network (this may require *e.g.* cherry-picking a single interaction in a multicom, or using out-of-order execution for the choreography).

By combining Lemma 3.3 with Lemma 2.7, we get operational correspondences for both $\to_s$ and $\to_c$.

Theorem 3.4. *If C is projectable, then:*

1. *if $C, \sigma \to_s C', \sigma'$ then, there is N such that $[\![C']\!] \sqsubset N$ and $[\![C]\!], \sigma \to^+ N, \sigma'$;*
2. *if $[\![C]\!], \sigma \to N, \sigma'$ then, there are N', C', and σ'' such that $[\![C']\!] \sqsubset N', C, \sigma \to_s^+ C', \sigma''$, and $N, \sigma' \to^* N', \sigma''$.*

Theorem 3.5. *If C is projectable, then:*

1. *if $C, \sigma \to_c C', \sigma'$ then, there are N, C'', and σ'' such that $[\![C'']\!] \sqsubset N, C', \sigma' \to_c^* C'', \sigma''$, and $[\![C]\!], \sigma \to^+ N, \sigma''$;*
2. *if $[\![C]\!], \sigma \to N, \sigma'$ then, there is C' such that $[\![C']\!] \sqsubset N$ and $[\![C]\!], \sigma \to_c [\![C']\!], \sigma'$.*

As a corollary of the operational correspondences that we developed and the progress property of choreographies we get that projected networks are deadlock-free.

Corollary 3.6. *Let $N = [\![C]\!]$ for some C. Either*

1. *$N \preceq \mathbf{0}$ (N has terminated), or*
2. *for any σ there exist N' and σ' such that $N, \sigma \to N', \sigma'$ (N can always reduce).*

4 RELATED WORK AND CONCLUSIONS

Scatter/gather primitives for choreographic programs were introduced in [14], in order to use choreographies for modelling cyber-physical systems. The primitive of asynchronous exchange, where two processes exchange a value at the same time, was introduced in [5]. Neither of these works discussed how such primitives may be supported in the paradigm of Choreographic Programming [15], in order to generate correct-by-construction implementations. Furthermore, these primitives are not general, in the sense that one construct cannot be used to obtain the same effect as the other. In this work, we have addressed both issues, by introducing unifying primitives (our multicoms/multisels) that can capture both patterns and defining an EndPoint Projection that generates correct process terms in a calculus of concurrent processes.

Some previous works on choreographic programming includes a parallel composition operator for choreographic terms ($C \mid C'$), for example [1]. Implementing our multicom/multisel using parallel composition requires the possibility to join the two terms C and C' after they have finished execution, and then to proceed with a continuation. These are not supported in [1]. Instead, both a parallel composition operator and a general sequential composition operator ($C; C'$) are present in [8], which would in theory allow for encoding our grouped interactions. However, the combination of these two operators can cause EPP to generate interfering communication actions between the parallel branches, and between the parallel branches and the continuation. A correct EPP in [8] is then obtained by adding (i) distinct auxiliary communication channels for communications and (ii) hidden communications for the propagation of information about internal choices at participants. Our approach is more efficient, because (i) we simply assume one reusable duplex channel for each pair of processes (used by all communications between them) and (ii) our well-formedness condition for multisels combined with merging guarantees that all participants agree without the need for hidden communications. Furthermore, the model in [8] does not consider our well-formedness conditions for multicoms and may thus lead to confusing data races. For example, the (equivalent of the) exchange $\{\mathsf{p}.x \twoheadrightarrow \mathsf{q}.x, \mathsf{q}.x \twoheadrightarrow \mathsf{p}.x\}$ is allowed (notice that x is used both for receiving and sending), which in a synchronous system would *never* yield the expected exchange, but rather a copy of x from p to q or vice versa (since one of the two interactions must be fully performed before the other can).

Most works on choreographic programming fall into two categories: those that use a sequential semantics (like [1, 3, 8, 12]), and those that allow for out-of-order concurrent execution (like

[2, 6, 14, 16]). So far, adopting the first view meant requiring the programmer to write all concurrent behaviour manually, which could be error-prone (*cf.* the complex verification techniques for detecting some mistakes in [1]). And, adopting the second view meant sacrificing the straightforward semantic interpretation of choreographies. Our development bridges this gap and offers a third view: programmers can use the sequential semantics of choreographies to reason about communication behaviour—in a language where concurrency does not need to be manually specified, because we simply abstract from it—and then stand on the shoulders of our results for the concurrent semantics (by operational correspondence) to know that safety is preserved in concurrent implementations. This result has an important practical implication: it is feasible to build a debugger for choreographies that uses the sequential semantics, since all results will be equivalent anyway—this would help programmers, since they would have to debug many fewer possible executions. But we do not need to give up the efficiency and realism of the concurrent semantics for runtime process implementations.

In [7], an operational correspondence result is presented in the setting of asynchronous communications for the calculus of core choreographies [6]; specifically, the authors show that programmers can interpret core choreographies as using synchronous communications, and that there is a safe way of obtaining more efficient asynchronous implementations without needing manual intervention. We conjecture that our development may be combined with that in [7], to obtain an asynchronous semantics for grouped interactions. We leave this combination to future work.

The congruence rules for swapping independent choreographic interactions were first introduced in [2]; we have extended them to deal with groups of interactions (multicoms/multisels). Our terms for recursion and conditionals in choreographies are standard, from [1, 6]. Likewise, the terms for recursion, conditionals, and parallel composition of networks in our process model are borrowed from [6].

ACKNOWLEDGMENTS

This work was partially supported by the Open Data Framework project at the University of Southern Denmark, and by the Independent Research Fund Denmark, Natural Sciences, grant DFF-7014-00041.

REFERENCES

[1] Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2012. Structured Communication-Centered Programming for Web Services. *ACM Trans. Program. Lang. Syst.* 34, 2 (2012), 8.
[2] Marco Carbone and Fabrizio Montesi. 2013. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, R. Giacobazzi and R. Cousot (Eds.). ACM, 263–274.
[3] Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. 2017. Choreographies, logically. *Distributed Computing* (2017). https://doi.org/10.1007/s00446-017-0295-1
[4] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science* 26, 2 (2016), 238–302. https://doi.org/10.1017/S0960129514000188
[5] Luís Cruz-Filipe, Kim S. Larsen, and Fabrizio Montesi. 2017. The Paths to Choreography Extraction. In *FoSSaCS (LNCS)*, Javier Esparza and Andrzej S. Murawski (Eds.), Vol. 10203. Springer, 424–440.
[6] Luís Cruz-Filipe and Fabrizio Montesi. 2016. A Core Model for Choreographic Programming. In *FACS (LNCS)*, Olga Kouchnarenko and Ramtin Khosravi (Eds.), Vol. 10231. Springer, 17–35. https://doi.org/10.1007/978-3-319-57666-4_3
[7] Luís Cruz-Filipe and Fabrizio Montesi. 2017. On Asynchrony and Choreographies. In *Proceedings of ICE 2017*. Accepted for publication.
[8] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. 2017. Dynamic Choreographies: Theory And Implementation. *Logical Methods in Computer Science* 13, 2 (2017).
[9] Chor development team. 2013. Chor Programming Language. (2013). http://www.chor-lang.org/.
[10] Daniele Gorla. 2010. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.* 208, 9 (2010), 1031–1053.
[11] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9. https://doi.org/10.1145/2827695
[12] Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. 2008. Bridging the Gap between Interaction- and Process-Oriented Choreographies. In *SEFM*, A. Cerone and S. Gruner (Eds.). IEEE, 323–332.
[13] Hugo A. López and Kai Heussen. 2017. Choreographing cyber-physical distributed control systems for the energy sector. In *SAC*. ACM, 437–443.
[14] Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. 2016. Enforcing Availability in Failure-Aware Communicating Systems. In *FORTE (Lecture Notes in Computer Science)*, Vol. 9688. Springer, 195–211.
[15] Fabrizio Montesi. 2013. *Choreographic Programming*. Ph.D. Thesis. IT University of Copenhagen. http://fabriziomontesi.com/files/choreographic_programming.pdf
[16] Fabrizio Montesi and Nobuko Yoshida. 2013. Compositional Choreographies. In *CONCUR (LNCS)*, Vol. 8052. Springer, 425–439.
[17] R.M. Needham and M.D. Schroeder. 1978. Using encryption for authentication in large networks of computers. *Commun. ACM* 21, 12 (Dec. 1978), 993–999. https://doi.org/10.1145/359657.359659
[18] Z. Qiu, X. Zhao, C. Cai, and H. Yang. 2007. Towards the theoretical foundation of choreography. In *WWW*. ACM, 973–982.