Omission Failures in Choreographic Programming

Eva Graversen[1,2], Fabrizio Montesi[2] and Marco Peressotti[2]

[1]Tallinn University of Technology
[2]University of Southern Denmark

Abstract

Choreographic programming promises a simple approach to concurrent and distributed programming: write the collective communication behaviour of a system of processes as a *choreography*, and then the programs for these processes are automatically compiled through a provably-correct procedure known as endpoint projection. While this promise prompted substantial research, a theory that can deal with realistic *communication failures* in a distributed network remains elusive. In this work, we provide the first theory of choreographic programming that addresses realistic communication failures taken from the literature of distributed systems: processes can send or receive fewer messages than they should (*send* and *receive omission*), and the network can fail at transporting messages (*omission failure*). Our theory supports the programming of strategies for failure recovery and is expressive enough to capture different communication patterns that usually required ad-hoc choreographic primitives and realistic protocols like two-phase commit.

1. Introduction

In the paradigm of choreographic programming [1], programs express coordination plans for communicating processes as compositions of high-level communication primitives inspired by security protocol notation [2]. In particular, the choreographic primitive $\mathsf{p}.e \to \mathsf{q}.x$ reads 'process p communicates the evaluation of expression e to process q, which stores it in its variable x'.

Key to choreographic programming is the notion of endpoint projection (EPP), a procedure for compiling choreographies into distributed implementations in terms of appropriate send and receive actions [3, 4]. EPP provides an escape from the challenge of separately writing compatible process programs, which is notoriously hard even for expert developers [5]. This motivated a number of theoretical investigations and implementations, including a full-fledged object-oriented choreographic programming language that extends Java [6], libraries for Haskell [7] and Rust [8], several mechanisations [9–12], and the correct implementation of distributed cryptographic applications [13, 14].

Despite all the recent interest in choreographic programming and neighbouring approaches, like multiparty session types [15], the theories presented so far rely on strong assumptions about communication actions. Most works just assume that communications never fail. Others allow for some communications or processes to fail, but with limitations. For example, they might assume synchronous communication to detect link failures [16], that some processes can never crash, that failures are permanent (crash, fail-stop), or reliable FIFO communications [17]. These limitations obscure the applicability of these approaches to the programming of protocols that designed to deal with realistic communication failures (like two-phase commit, or 2PC), without assuming costly middleware that masks these failures.

To meet this need, in this work we present a new theory of choreographic programming: Lossy Choreographies (LC). LC is the first choreographic programming theory that can deal with realistic communication failures from the literature of distributed systems: processes can send or receive fewer messages than they should – *send* and *receive omission* [18, 19] – and the network can fail at transporting messages – *omission failure* [20]. Our only assumptions are that messages do not get corrupted and participants are not malicious, i.e., they run the code projected from the choreography.

The key technical challenge in developing LC lies in designing its syntax and semantics, because we need to equip programmers with the ability to program recovery strategies for failures. These strategies

ICTCS 2025: Italian Conference on Theoretical Computer Science, September 10–12, 2025, Pescara, Italy
eva.graversen@taltech.ee (E. Graversen); fmontesi@imada.sdu.dk (F. Montesi); peressotti@imada.sdu.dk (M. Peressotti)
0000-0002-9430-4907 (E. Graversen); 0000-0003-4666-901X (F. Montesi); 0000-0002-0243-0480 (M. Peressotti)

can be asymmetric, e.g., the sender of a message might use exponential backoff while the intended receiver a timeout. This calls for deconstructing the syntax and semantics of the usual choreographic communication primitive to allow for independent implementations and executions of the send and receive sides. However, at the same time, we need to retain the high-level view of choreographies on what communications and computations the programmer wants to take place.

LC offers a simple solution by introducing a notion of *frames* to choreographic programming. Specifically, we declare the intent to communicate a message of type T from p to q by writing $(k, k')^{\mathsf{T}} : \mathsf{p} \to \mathsf{q}$,which creates the *frames* k at p and k' at q. Intuitively, the frame k is a *promise* that p will send a value of type T for this particular communication, and dually k' is a *future* where q can try to receive that value. Frames can then be used separately at the two processes in their own strategies for performing their respective obligations (sending or receiving). For example, p could use a procedure with exponential backoff and q one with a timeout, written as $\texttt{sendExpBackoff}(\mathsf{p}; k); \texttt{recvTimeout}(\mathsf{q}; k')$. Leveraging this design and the high-level view of choreographic programming, LC enables processes to interleave the handling of multiple frames, each with its own recovery strategy. LC is thus expressive enough to capture different communication patterns that usually required ad-hoc choreographic primitives like scatter and to model realistic protocols like 2PC. Indeed, we will show that LC can serve as a general model where such primitives can be reconstructed as syntactic sugar.

Structure and contributions We introduce Lossy Choreographies in Section 2 and give some examples of protocols modelled in this language in Section 3. In Section 4 we equip LC with a type system and show that well-typed choreographies enjoy progress. In Section 5 we present a language for local processes and endpoint projection for compiling choreographies to processes. Finally, we compare our development to related work in Section 6 and conclude in Section 7. In the accompanying technical report [21], we provide the full technical details of LC and further proofs of its adequacy: (i) a static analysis for verifying delivery guarantees on frames (at-most-once and exactly-once, depending on which omissions can take place) and (ii) an implementation given as a library for programming recovery strategies in Choral [6], a state-of-the-art choreographic programming language that compiles to Java.

2. Lossy Choreographies

Syntax The language of Lossy Choreographies (LC) is defined by the following grammar.

$$C ::= I; C \mid \mathbf{0} \qquad s ::= l \mid e \qquad b ::= e \mid c! \mid c? \mid c?l \qquad c ::= k \mid (\mathsf{p}, m)$$
$$I ::= (k, k')^{\mathsf{T}} : \mathsf{p} \to \mathsf{q} \mid \mathsf{p}.c!s \mid \mathsf{p}.c?x \mid \mathsf{p}.x := s \mid \texttt{if}\ \mathsf{p}.b\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid X\langle \vec{\mathsf{p}}; \vec{c} \rangle$$

A choreography C is a sequence of instructions (I) ending in $\mathbf{0}$. Term $(k, k')^{\mathsf{T}} : \mathsf{p} \to \mathsf{q}$ declares a communication of a message of type T from p to q, binding the *frame names* k for the sender and k' for the receiver to the continuation — for simplicity we assume the Barendregt convention replacing bound names as needed. When this declaration is executed, frame names are substituted with pairs – (q, m) – that contain the name of the other process in the communication (q) and a natural number (m) called *frame number* that identifies this specific communication from p to q.[1] Communications are then implemented with send terms like $\mathsf{p}.c!s$ (read 'p sends s on frame c') and receive terms like $\mathsf{q}.c'?x$ (read 'q receives a message on c' and stores it in x'). These actions may fail, as we will show in our semantics.

Following standard practice from choreographic programming and session types, we syntactically distinguish when a process sends the result of a local expression (e) or a statically-defined literal (l, also known as selection labels) [4, 15]. We leave the language of local expressions as a parameter of our theory, as in other choreographic languages [3, 4, 9, 11]. Selection labels make the development of endpoint projection clearer, because it streamlines detecting whether all processes involved in a conditional communicate sufficient information about which branch has been chosen [3]. This standard solution adapts to our setting without major changes.

[1] Frame numbers are inspired by the sequence numbers found TCP.

In the local assignment term $\mathsf{p}.x := s$, p assigns its local variable x the value of local computation s. In a conditional $\texttt{if}\,\mathsf{p}.b\,\texttt{then}\,C_1\,\texttt{else}\,C_2$, p evaluates the guard b and chooses between the possible continuations C_1 and C_2 accordingly. Guards can be of four forms: $c!$ evaluates to true if the last send attempt for c was successful and to false otherwise; $c?$ behaves like $c?$ but for receive attempts; $c?l$ behaves like $c?$ but additionally checks that the received values is the label l; and e evaluates according to the semantics of the language of local expressions. Term $X\langle\vec{\mathsf{p}};\vec{c}\rangle$ invokes procedure X with arguments $\vec{\mathsf{p}}$ and $\vec{c}$. Procedures are defined by providing equations as usual in process calculi [22].

Example 1. We formalise the procedures `sendExpBackoff` and `recvTimeout` from the introduction as a family of procedures indexed by the expression at the sender (e) and the variable at the receiver (x). We use some auxiliary variables to store the number of send attempts (n) and the timeout (to).

```
1 sendExpBackoff_e(p; k):
2   p.k!e;  //Try sending
3   if ¬p.k! then  //Check whether send has succeeded
4     p.n := waitAndReturn(n);  //Wait 2^n milliseconds and return n + 1
5     sendExpBackoff_e⟨p; k⟩  //Call procedure recursively
6 else 0  //If send is successful, finish
```

```
1 recvTimeout_x(p; k):
2   p.now := currentTime();
3   p.k?x;  //Try receiving
4   p.to := to − (currentTime() − now);
5   if p.to > 0 ∧ ¬p.k? then  //Check whether receive has succeeded or time has elapsed
6     recvTimeout_x⟨p; k⟩  //Call procedure recursively
7 else 0  //If receive is successful or the time has elapsed, finish
```

Using these procedures, we can define as syntactic sugar a communication primitive with a timeout that recovers the usual simplicity of choreographic syntax. Below, $\mathsf{p}.e \to^{t}_{\mathsf{T}} \mathsf{q}.x$ is a communication of $\mathsf{p}.e$ (of type T) to $\mathsf{q}.x$ with timeout t.

$$(k,k')^{\mathsf{T}}\colon \mathsf{p}\to\mathsf{q};\mathsf{p}.n := 0;\texttt{sendExpBackoff}_e\langle\mathsf{p};k\rangle;\mathsf{q}.to := t;\texttt{recvTimeout}_x\langle\mathsf{p};k\rangle;\mathbf{0}$$

LC is expressive enough to capture other strategies as syntactic sugar, including acknowledgements and compensations (custom code triggered in case of failure). Examples are given in Section 3.

Semantics We give the semantics for LC in terms of a labelled transition system (LTS). The states of this system are triples of the form $\langle C,\Sigma,K\rangle$ called *configurations* comprised by a choreography C (a term in LC extended with two runtime terms discussed later), a choreographic store Σ modelling the processes memory, and a communication transit K modelling messages in transit over the network.

A *choreographic store* Σ is a map from process names to their local memory stores ranged over by σ, similarly to several previous theories [3, 4, 9, 23–25]. We write $\Sigma(\mathsf{p}.x) = v$ to denote that $\Sigma(\mathsf{p}) = \sigma$ such that $\sigma(x) = v$, read 'variable x at process p has value v'. Differently from prior work, we reserve two locations (**fc**,**fb**) to store data that processes use for implementing frames. The reserved location **fc** stores counters used by the current process to generate frame identifiers, one for each other process: $\Sigma(\mathsf{p}.\mathbf{fc}.\mathsf{q})$ is the counter at p used for frames from/to q. The reserved location **fb** tracks the state of frames used by the current process. We write $\Sigma(\mathsf{p}.\mathbf{fb}.\mathsf{q}.n)$ for the state of the frame that p uses for sending or receiving a message to q with identifier n. This state can take different values: (1) it is $\perp$ when the frame is first created, (2) it is $\checkmark$, when the frame is outgoing and the frame has been successfully handed over to the network, (3) it is v when the frame is incoming and the network has delivered the value v for this frame but the process has not consumed it yet, and (4) it is $v\checkmark$ when the value has been finally read by the process. Frame counters and identifiers are similar to the device used in TCP to correlate asynchronous messages at sender and receiver. The idea is that: (1) each process maintains a counter for each other process it interacts with; (2) frame declarations increment

$$\frac{\Sigma(\mathsf{p}) \vdash s \downarrow v}{\langle \mathsf{p}.x := s; C, \Sigma, K\rangle \xrightarrow{\tau@\mathsf{p}}_{\mathcal{C}} \langle C, \Sigma[\mathsf{p}.x \mapsto v], K\rangle}\textsc{CAssign}$$

$$\frac{\Sigma(\mathsf{p}.\mathbf{fc}.\mathsf{q}) = n \quad \Sigma' = \Sigma[\mathsf{p}.\mathbf{fc}.\mathsf{q} \mapsto n+1][\mathsf{p}.\mathbf{fb}.\mathsf{q}.n \mapsto \bot]}{\langle \mathsf{p}.(\mathsf{q},k)^{\mathsf{T}}; C, \Sigma, K\rangle \xrightarrow{\tau@\mathsf{p}}_{\mathcal{C}} \langle C[(\mathsf{q},n)/k], \Sigma', K\rangle}\textsc{CFrame}$$

$$\frac{\langle \mathsf{p}.(\mathsf{q},k)^{\mathsf{T}}; \mathsf{q}.(\mathsf{p},k')^{\mathsf{T}}; C, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{C}} \langle C', \Sigma', K\rangle \quad \mu \in \{\tau@\mathsf{p}, \tau@\mathsf{q}\}}{\langle (k,k')^{\mathsf{T}} : \mathsf{p} \to \mathsf{q}; C, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{C}} \langle C', \Sigma', K\rangle}\textsc{CCom}$$

$$\frac{\Sigma(\mathsf{p}) \vdash s \downarrow v}{\langle \mathsf{p}.(\mathsf{q},m)!s; C, \Sigma, K\rangle \xrightarrow{\mathsf{p}.(\mathsf{q},m)\rightarrow}_{\mathcal{C}} \langle C, \Sigma[\mathsf{p}.\mathbf{fb}.\mathsf{q}.m \mapsto \checkmark], K \uplus (\mathsf{p},\mathsf{q},m,v)\rangle}\textsc{CSend}$$

$$\frac{\Sigma(\mathsf{p}.\mathbf{fb}.\mathsf{q}.m) \in \{v, v\checkmark\}}{\langle \mathsf{p}.(\mathsf{q},m)?x; C, \Sigma, K\rangle \xrightarrow{\rightarrow\mathsf{p}.(\mathsf{q},m)}_{\mathcal{C}} \langle C, \Sigma[\mathsf{p}.\mathbf{fb}.\mathsf{q}.m \mapsto v\checkmark][\mathsf{p}.x \mapsto v], K\rangle}\textsc{CRecv}$$

$$\frac{\Sigma(\mathsf{p}) \vdash s \downarrow v}{\langle \mathsf{p}.(\mathsf{q},m)!s; C, \Sigma, K\rangle \xrightarrow{\mathsf{p}.(\mathsf{q},m)\rightarrow}_{\mathcal{C}} \langle C, \Sigma, K\rangle}\textsc{CSendFail} \qquad \frac{\Sigma(\mathsf{p}.\mathbf{fb}.\mathsf{q}.m) = \bot}{\langle \mathsf{p}.(\mathsf{q},m)?x; C, \Sigma, K\rangle \xrightarrow{\rightarrow\mathsf{p}.(\mathsf{q},m)}_{\mathcal{C}} \langle C, \Sigma, K\rangle}\textsc{CRecvFail}$$

$$\frac{}{\langle C, \Sigma, K \uplus (\mathsf{p},\mathsf{q},m,v)\rangle \xrightarrow{\tau}_{\mathcal{C}} \langle C, \Sigma[\mathsf{q}.\mathbf{fb}.\mathsf{p}.m \mapsto v], K\rangle}\textsc{CDel} \qquad \frac{}{\langle C, \Sigma, K \uplus (\mathsf{p},\mathsf{q},m,v)\rangle \xrightarrow{\tau}_{\mathcal{C}} \langle C, \Sigma, K\rangle}\textsc{CLoss}$$

$$\frac{\Sigma(\mathsf{p}) \vdash b \downarrow \mathsf{true}}{\langle \mathtt{if}\ \mathsf{p}.b\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2; C, \Sigma, K\rangle \xrightarrow{\mathsf{left}@\mathsf{p}}_{\mathcal{C}} \langle C_1 \fatsemi C, \Sigma, K\rangle}\textsc{CThen}$$

$$\frac{\Sigma(\mathsf{p}) \vdash b \downarrow \mathsf{false}}{\langle \mathtt{if}\ \mathsf{p}.b\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2; C, \Sigma, K\rangle \xrightarrow{\mathsf{right}@\mathsf{p}}_{\mathcal{C}} \langle C_2 \fatsemi C, \Sigma, K\rangle}\textsc{CElse}$$

$$\frac{X(\vec{\mathsf{p}}; \vec{k}) \mathtt{:}\ C \in \mathcal{C} \qquad \mathsf{r} \in \vec{\mathsf{p}}}{\langle X\langle \vec{\mathsf{p}}; \vec{c}\rangle; C', \Sigma, K\rangle \xrightarrow{\tau@\mathsf{r}}_{\mathcal{C}} \langle \vec{\mathsf{p}} \setminus \mathsf{r} : X[\vec{\mathsf{p}}; \vec{c}].C'; C[\vec{\mathsf{p}}/\vec{\mathsf{q}}][\vec{c}/\vec{k}] \fatsemi C', \Sigma, K\rangle}\textsc{CCall}$$

$$\frac{\mathsf{r} \in \vec{\mathsf{q}} \quad \vec{\mathsf{q}} \setminus \mathsf{r} \neq \emptyset}{\langle \vec{\mathsf{q}} \mathtt{:} X[\vec{\mathsf{p}}; \vec{c}].C'; C, \Sigma, K\rangle \xrightarrow{\tau@\mathsf{r}}_{\mathcal{C}} \langle \vec{\mathsf{q}} \setminus \mathsf{r} : X[\vec{\mathsf{p}}; \vec{c}].C'; C, \Sigma, K\rangle}\textsc{CEnter}$$

$$\frac{}{\langle \mathsf{r} : X[\vec{\mathsf{p}}; \vec{c}].C'; C, \Sigma, K\rangle \xrightarrow{\tau@\mathsf{r}}_{\mathcal{C}} \langle C, \Sigma, K\rangle}\textsc{CFinish} \qquad \frac{\langle C, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{C}} \langle C', \Sigma', K'\rangle \quad \mathsf{pn}(I) \cap \mathsf{pn}(\mu) = \emptyset}{\langle I; C, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{C}} \langle I; C', \Sigma', K'\rangle}\textsc{CDelayI}$$

$$\frac{\langle C_1, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{C}} \langle C_1', \Sigma', K'\rangle \quad \langle C_2, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{C}} \langle C_2', \Sigma', K'\rangle \quad \mathsf{p} \notin \mathsf{pn}(\mu)}{\langle \mathtt{if}\ \mathsf{p}.b\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2; C, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{C}} \langle \mathtt{if}\ \mathsf{p}.b\ \mathtt{then}\ C_1'\ \mathtt{else}\ C_2'; C, \Sigma', K'\rangle}\textsc{CDelayC}$$

Figure 1: Lossy Choreographies, operational semantics.

counters locally, i.e., without synchronising with the other party; (3) frames are assigned the value held by the corresponding counter when they are created. Our semantics ensures that frame counters remain consistent through the execution of a choreography and so it suffices to agree on an initial value — we formalise this property after we present the semantics (Definition 1).

The network and the messages transiting on it are modelled by a *communication transit* K: a multiset of messages that have been successfully handed over to the network from the sender, but have not been delivered to the receiver. Each message has the form $(\mathsf{p},\mathsf{q},m,v)$ where p is the sender, q is the receiver, m is the frame number, and v is the payload. K may contain messages with the same sender, receiver, and frame number but different payloads e.g., when a sender makes multiple attempts updating a timestamp or counter in the payload each time and these are still in transit. The defining K as a multiset, we model a network without any ordering guarantee.

The transition relation of the LTS is given by the SOS specification in Figure 1 and it is parameterised on a set $\mathcal{C}$ of procedure definitions $X(\vec{\mathsf{p}}; \vec{k})\mathtt{:}\ C$ where X is the procedure name, $\vec{\mathsf{p}}$ and $\vec{k}$ are its formal parameters for process and frame names, the choreography C (free from runtime terms) is its body.

In the remainder, we write $\mathsf{pn}(C)$ and and $\mathsf{pn}(\mu)$ for the set of process names that appear in the choreography C and the label μ, respectively.

Rule CAssign models local assignments and is standard: the update notation $\Sigma[\mathsf{p}.x \mapsto v]$ denotes a choreographic store such that $(\Sigma[\mathsf{p}.x \mapsto v])(\mathsf{p}.x) = v$ and behaves like Σ otherwise [4]. The premise

$\Sigma(\mathsf{p}) \vdash s \downarrow v$ states that the evaluation of s in the process store $\Sigma(\mathsf{p})$ yields value v – the definition of this relation on local expressions (e) is a parameter of the theory (evaluation may be partial or nondeterministic) whereas on labels it is assumed as $\Sigma(\mathsf{p}) \vdash l \downarrow l$.

Rules CCom and CFrame model the creation of a pair of frames for a communication from p to q without relying on any rendezvous mechanism. Instead, these rules rely on the runtime term $\mathsf{p}.(\mathsf{q}, k)^{\mathsf{T}}$ to represent the creation of a frame to/from q at p: in rule CFrame, $\mathsf{p}.(\mathsf{q}, k)^{\mathsf{T}}$ is consumed, the frame k is assigned a number n, its state is initialised to $\bot$, and the frame counter for q is incremented; and in rule CCom, the semantics of $(k, k')^{\mathsf{T}} : \mathsf{p} \to \mathsf{q}$ is essentially defined as that of $\mathsf{p}.(\mathsf{q}, k)^{\mathsf{T}}; \mathsf{q}.(\mathsf{p}, k')^{\mathsf{T}}$. The order of the runtime terms in the premise of rule CCom is immaterial since the semantics allows these to be executed in any order via rule CDelayI (explained below).

Rules CSend and CSendFail capture the possible executions of a send attempt for a frame (q, m) at p. The first, Rule CSend models a successful send: a payload v is computed and the message $(\mathsf{p}, \mathsf{q}, m, v)$ is successfully handed off to the communication transit (K). It also updates the frame state at the sender accordingly. Rule CSendFail, instead, models a *send omission* failure: the send action is consumed but it omits (fails at) adding the message to K. In both cases, no information about the attempt is propagated to the receiver: our semantics is asynchronous and the only way to exchange information across processes is through fallible communication actions and a lossy K. Note that rule CSend does not check the state of the frame and that the new one is added to K with a multiset union ($\uplus$), to reflect the real-world situation that a sender may perform multiple send actions resulting in the transmission of multiple messages (regardless of differences for the payload m) for the same frame.

Once a message is in transit, there are two possibilities: It can either be successfully delivered to the receiver, which causes a corresponding update to its frame state (rule CDel), or it can incur an *omission failure* and be lost (rule CLoss). The sender has no knowledge of what happens.

Rules CRecv and CRecvFail model a receive attempt. The attempt can be successful (CRecv) only if the receiver's state for the desired frame contains a value (previously put there by rule CDel). Otherwise, if no message for that frame has reached the receiver yet, the receive attempt fails (rule CRecvFail).

Rules CThen and CElse model conditionals. The evaluation of guards on frames is as follows:

$$\frac{\Sigma(\mathsf{p}.\mathbf{fb}.\mathsf{q}.m) = v\checkmark}{\Sigma(\mathsf{p}) \vdash (\mathsf{q}, m)? \downarrow \mathsf{true}} \qquad \frac{\Sigma(\mathsf{p}.\mathbf{fb}.\mathsf{q}.m) = \checkmark}{\Sigma(\mathsf{p}) \vdash (\mathsf{q}, m)! \downarrow \mathsf{true}} \qquad \frac{\Sigma(\mathsf{p}.\mathbf{fb}.\mathsf{q}.m) = l\checkmark}{\Sigma(\mathsf{p}) \vdash (\mathsf{q}, m)?l \downarrow \mathsf{true}}$$

$$\frac{\Sigma(\mathsf{p}.\mathbf{fb}.\mathsf{q}.m) \neq v\checkmark}{\Sigma(\mathsf{p}) \vdash (\mathsf{q}, m)? \downarrow \mathsf{false}} \qquad \frac{\Sigma(\mathsf{p}.\mathbf{fb}.\mathsf{q}.m) \neq \checkmark}{\Sigma(\mathsf{p}) \vdash (\mathsf{q}, m)! \downarrow \mathsf{false}} \qquad \frac{\Sigma(\mathsf{p}.\mathbf{fb}.\mathsf{q}.m) \neq l\checkmark}{\Sigma(\mathsf{p}) \vdash (\mathsf{q}, m)?l \downarrow \mathsf{false}}$$

We use a meta-operator for sequential composition of choreographies '$\fatsemi$' [4, 26] which replaces $\mathbf{0}$ in a choreography with another choreography ($\mathbf{0} \fatsemi C \triangleq C$ and $(I; C) \fatsemi C' \triangleq I; (C \fatsemi C')$).

Remaining rules are standard for the theory of choreographic languages [4]. Rules CCall, CEnter and CFinish model a decentralised call to a choreographic procedure (i.e., without synchronising the processes that enter the procedure). These rules rely on the runtime term $\vec{\mathsf{q}}{:}X[\vec{\mathsf{p}}; \vec{c}].C'$ to track which processes have not joined the call ($\vec{\mathsf{q}}$) and thus cannot perform any action from its continuation via the delay rules. Rule CCall unfolds X with its body where formal arguments ($\vec{\mathsf{q}}; \vec{k}$) are instantiated with the actual ones $\vec{\mathsf{p}}; \vec{c}$ (parameters are positional). Rules CEnter and CFinish update and remove the runtime term as the remaining processes join the call. Rules CDelayI and CDelayC model that actions performed by distinct processes are executed concurrently by delaying the execution of a term.

With the language fully defined, we formalise the property that frame states, counters, and runtime terms are consistent across a choreography and store.

Definition 1. Σ is *consistent with* C if, for any two p and q in C: (1) $\Sigma(\mathsf{p}.\mathbf{fb}.\mathsf{q}.n)$ is defined for any $n < \Sigma(\mathsf{p}.\mathbf{fc}.\mathsf{q})$; and (2) $|\phi^{\mathsf{p}}_{\mathsf{q}}(C)| = \max(0, \Sigma(\mathsf{p}.\mathbf{fc}.\mathsf{q}) - \Sigma(\mathsf{q}.\mathbf{fc}.\mathsf{p}))$ where the (partial) function $\phi^{\mathsf{p}}_{\mathsf{q}}$ computes the set of frame names to q that p has yet to initialise as follows.

$$\phi^{\mathsf{p}}_{\mathsf{q}}(\mathbf{0}) \triangleq \emptyset \qquad \phi^{\mathsf{p}}_{\mathsf{q}}(I; C) \triangleq \phi^{\mathsf{p}}_{\mathsf{q}}(I) \cup \phi^{\mathsf{p}}_{\mathsf{q}}(C) \text{ where } \phi^{\mathsf{p}}_{\mathsf{q}}(I) \cap \phi^{\mathsf{p}}_{\mathsf{q}}(C) = \emptyset$$

$$\phi^{\mathsf{p}}_{\mathsf{q}}(\mathsf{p}.(\mathsf{q}, k)^{\mathsf{T}}) \triangleq \{k\} \qquad \phi^{\mathsf{p}}_{\mathsf{q}}(\mathtt{if}\ \mathsf{p}.b\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2) \triangleq \phi^{\mathsf{p}}_{\mathsf{q}}(C_1) \text{ where } \phi^{\mathsf{p}}_{\mathsf{q}}(C_1) = \phi^{\mathsf{p}}_{\mathsf{q}}(C_2)$$

$$\phi^{\mathsf{p}}_{\mathsf{q}}(I) = \emptyset \text{ for } I \notin \{\mathsf{p}.(\mathsf{q}, k)^{\mathsf{T}}, \mathtt{if}\ \mathsf{p}.b\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\}$$

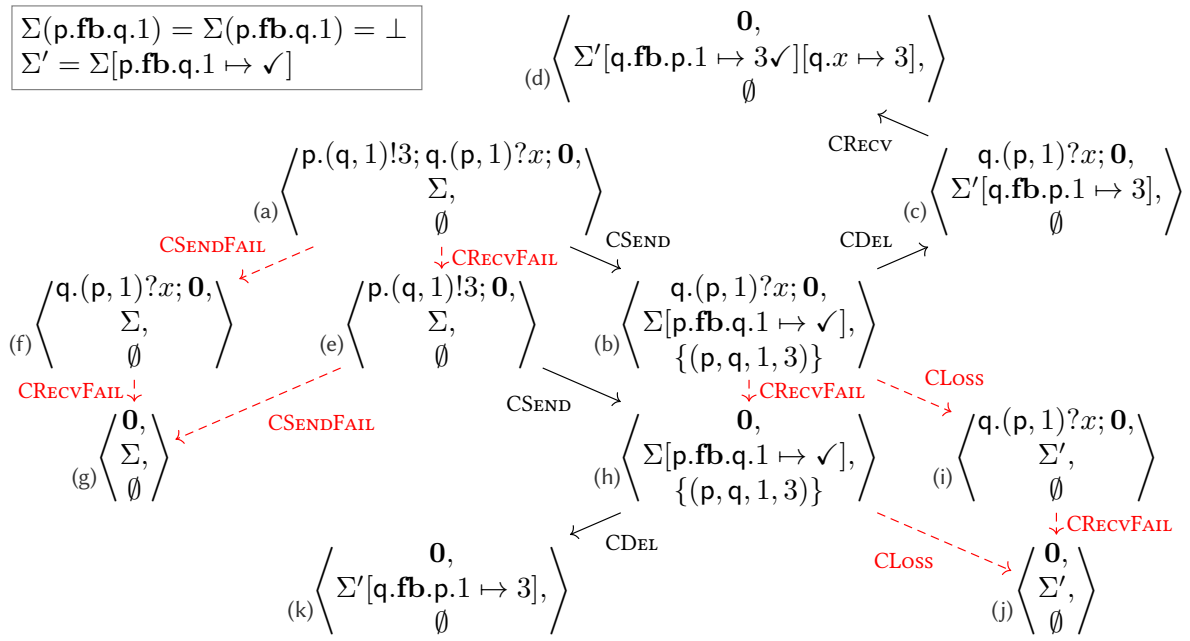

Figure 2: Labelled transition system illustrating all possible executions of an end-to-end communication in LC – dashed (red) arrows are transitions that model failures.

Note that the function $\phi^{\mathsf{p}}_{\mathsf{q}}$ is partial and rejects choreographies that misuse runtime terms for frame creation e.g., the condition on the case $I; C$ excludes repetitions of runtime terms for the same k.

Proposition 1. *If* $\langle C, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{C}} \langle C', \Sigma', K'\rangle$ *and* Σ *is consistent with* C*, then* Σ' *is consistent with* C'*.*

Understanding failures We illustrate the realism of our model wrt communication failures with a simple end-to-end communication. Consider the choreography $\mathsf{p}.(\mathsf{q},1)!3;\ \mathsf{q}.(\mathsf{p},1)?x;\ \mathbf{0}$ where p attempts to communicate the number 3 to q; the LTS in Figure 2 captures all the possible executions for this program. For convenience of exposition, we assign a letter to each state. Also, we do not show transition labels but rather the name of the rule applied to derive the transition – the transition from (a) to (e) is the only one that also requires rule CDelayI.

Every run of the program begins in state (a) and eventually terminates reaching $\mathbf{0}$. We describe the different situations at the ends of the possible executions.

- (d) This is the final state of the only successful execution: the value 3 is marked as delivered and no transition from (a) to (d) uses any of the rules that model send, receive, and omission failures.
- (g) This state is reached if we have both a send and a receive omissions. In the path via (f), a send omission stops the receive from ever succeeding. In the path via (e), we have a receive omission because the receive is attempted before the send action had a chance to put the message in the network. The latter case exemplifies how our semantics captures timeouts at the receiver. In fact, (e) may transition to (h) failing because of timing, as this is before the successful transit.
- (j) This state is reached because of an omission failure. In the path via (i), the omission failure causes the receive to fail. In the path via (e), the receive fails regardless of the omission because of timing.
- (k) This state is reached if the send succeeds but the receive fails because of timing issues: in the path through (e), the receive is executed before the send; in the path through (b), the receive is executed before the network could carry the message to the receiver.

3. Applications

Acknowledgements Assume a setting with omission failure (rule CLoss). In this setting, the definition of $\mathsf{p}.e \rightarrow^{t}_{\mathsf{T}} \mathsf{q}.x$ presented in Example 1 risks ending up in a state where the sender believes

it has completed a communication but the receiver has timed out due to the message being lost by the network. This is a common problem in practice, which is addressed by switching to "best-effort" strategies where delivery is possible (to varying degrees) but not certain.

The procedure below implements a simple communication protocol with capped retries and acknowledgements to the sender. In this, the strategy use by $\texttt{comACK}_{e,x}$ can be regarded as a simplification of that of TCP; four-phase handshakes or other protocols are implementable in LC as well.

```
1 comACK_{e,x}(s, r):
2   (k, k')^T : s → r;   //Create new frame for this communication
3   (k'_ack, k_ack)^Unit : r → s;   //Create new frame for the acknowledgement
4   sendExpBackoff_e(s; k);   //Send
5   recvTimeout_x(r; k');   //Receive
6   sendExpBackoff_unit(r; k'_ack);   //Send acknowledgement
7   sendUntilACK_{e,x}(s; k, k_ack);   //Call send procedure
8 sendUntilACK_{e,x}(s; k, k_ack):
9   recvTimeout_x(s; k_ack);   //Try receiving acknowledgement
10  if s.(n > 0) ∧ ¬s.k_ack? then   //Check number of attempts and received acknowledgement
11    s.n := n − 1;   //Update send attempt number
12    sendExpBackoff_e(s; k);   //Make another attempt at sending
13    sendUntilACK_{e,x}(s; k, k_ack);   //Repeat
14  else 0   //When acknowledgement has been received or we run out of attempts, end.
```

Compensations With $\texttt{comACK}_{e,x}$ we can also use LC to develop a new variant of choreographies that does not assume reliable transmission, i.e., when we are in a setting with omission failures. In this setting, a common pattern to deal with failures of best-effort communications are *compensations*. Fault compensations can be defined in LC (for both settings with and without omission failures) using conditionals, $\texttt{comACK}_{e,x}$ (or variations thereof), and some syntax sugar to improve readability. An expression $\mathsf{s}.e \Rightarrow^{BE} \mathsf{r}.x\{C_\mathsf{s}\}\{C_\mathsf{r}\}$ is a communication as in $\texttt{comACK}_{e,x}(\mathsf{s},\mathsf{r})$ where choreographies C_s and C_r are executed as compensations for faults detected by the sender s (no ack) or the receiver r, respectively. An example of communications with fault compensations is the communication construct defined in [16] where communication operations specify default values as compensations; this is recovered in LC using local computations as, e.g., in $\mathsf{s}.e \Rightarrow^{BE} \mathsf{r}.x\{\mathsf{s}.x := \texttt{foo}\}\{\mathsf{r}.x := 42\}$.

Any/Many communications We can also implement more complex communication primitives, like those in [27, 28]. Below are procedures that iteratively attempt at sending some frames until the sender stack accepts all or any of them, respectively, using a round-robin strategy.

```
1 sendAll_{n,e}(s; k_1, ..., k_n):
2   s.k_n!e;   //Send to the last frame in the list
3   if s.k_n! then   //Check if send was successful
4     sendAll_{n−1,e}(s; k_1, ..., k_{n−1})   //Send to the remaining frames
5   else sendAll_{n,e}(s; k_2, ..., k_n, k_1)   //Otherwise, try this frame again later
```

```
1 sendAny_{n,e}(s; k_1, ..., k_n):
2   s.k_1!e;   //Send to the first frame in the list
3   if ¬s.k_1! then   //Check if send was unsuccessful
4     sendAny_{n,e}(s; k_2, ..., k_n, k_1)   //Retry cycling through the frames
5   else 0   //End function if the send succeeded
```

We omit the dual procedures for receiving all or some frames, which are similarly defined. Combining these it is possible to implement scatter/gather communication primitives from [27].

```
1 scatter_{e,x}(s, r_1, ..., r_n; ∅):
2   (k_1, k'_1)^T : s → r_1; ... (k_n, k'_n)^T : s → r_n;   //Create frames
3   sendAll_e(s; k_1, ..., k_n);   //Call send all on the new frames
4   recvTimeout_x(r_1; k'_1); ... recvTimeout_x(r_n; k'_n)   //Each receiver receives on its frame.
```

Two-phase commit LC can be used to implement common protocols for dealing with failures. The two-phase commit protocol [29] is a protocol for getting a set of participants to agree to either commit or abort their local transaction thus providing a mechanism for distributed transactions in presence of unreliable communication. Each participant sends the controller a vote to commit or a veto for aborting the transaction. The controller tallies all the votes and decides to globally commit if all are in favour (a missing vote is regarded as a veto) and to globally abort otherwise. The controller then sends its decision to each participant, trying repeatedly to send until it receives an acknowledgement. Meanwhile the participants wait for the decision before they commit or abort their local transaction acknowledging the outcome to the controller. We base our code on the version of the protocol presented in [29].

```
12 PhaseCommit(c, p_1, ..., p_n; ):
2   (k_{v_1}, k'_{v_1})^Bool : p_1 → c; ... ; (k_{v_n}, k'_{v_n})^Bool : p_n → c;   //Frames for votes
3   (k_{d_1}, k'_{d_1})^Bool : c → p_1; ... ; (k_{d_n}, k'_{d_n})^Bool : c → p_n;   //Frames for decision
4   (k_{a_1}, k'_{a_1})^Unit : p_1 → c; ... ; (k_{a_n}, k'_{a_n})^Unit : p_n → c;   //Frames for acknowledgements
5   c.v_1 := false; ... ; c.v_n := false;   //All votes start out false at c
6   p_1.k_{v_1}!vote(); ... ; p_n.k_{v_n}!vote();   //Participants send votes
7   c.k'_{v_1}?v_1; ... ; c.k'_{v_n}?v_n;   //Controller receives votes (they remain false if receive fails)
8   c.decision := v_1 ∧ ··· ∧ v_n;   //Decision on whether to commit is made - only if all voted yes
9   sendAllUntilAck(c; k_{d_1}, ..., k_{d_n}, k'_{a_1}, ..., k'_{a_n});   //c sends the decision
10  recvDec(p_1; k'_{d_1}, k_{a_1}); ... ; recvDec(p_n; k'_{d_n}, k_{a_n})   //ps receive decisions
11 sendAllUntilAck_n(c; k_{d1}, ..., k_{d_n}, k_{a_1}, ..., k_{a_n}):
12  c.k_{d_n}!decision;   //c sends the decision to a participant
13  c.k_{a_n}?x;   //c receives acknowledgement from that participant
14  if k_{a_n}? then   //If the acknowledgement was received
15    sendAllUntilAck_{n-1}(c; k_{d_1}, ..., k_{d_{n-1}}, k_{a_1}, ..., k_{a_{n-1}})   //Send to other participants
16  else sendAllUntilAck_n(c; k_{d_2}, ..., k_{d_n}, k_{d_1}, k_{a_2}, ..., k_{a_n}, k_{a_1})   //Try again later
17 recvDec(p; k_d, k_a):
18  p.k_d?dec;   //Try receiving the decision
19  if p.k_d? then   //If receive was successful
20    if p.dec then p.memory := commit()
21    else p.memory := abort()
22    sendExpBackoff_unit(p, k_a)   //Keep sending acknowledgement until successful
23  else recvDec(p; k_d, k_a)   //If receive decision was unsuccessful, try again
```

Note that `sendAllUntilAck` is a combination of `sendAll` and `sendUntilAck` without the check on the number of send attempts; we omitted this for brevity but it could easily be reintroduced if one is concerned about c being blocked by a lost acknowledgement.

4. Typing

LC programs can get stuck if procedures are called on wrong arguments, communication actions are performed against the wrong frames or processes, and guards of conditionals are not of the expected type. We introduce a type system for LC that rejects this kind of programs and ensures progress.

Type judgements are of the form $\Gamma \vdash C$ where Γ is an environment for tracking the types of choreographic procedures, frames and the local memory.

$$\Gamma ::= X\langle \vec{\mathsf{p}}; \overrightarrow{k : \mathsf{T}} \rangle \mid \mathsf{p}.c\colon F \mid \mathsf{p}.x\colon \mathsf{T} \qquad F ::= !\mathsf{T} \mid ?\mathsf{T}$$

$$\frac{\Gamma, \mathsf{p}.k : !\mathsf{T}, \mathsf{q}.k' : ?\mathsf{T} \vdash C}{\Gamma \vdash (k, k')^{\mathsf{T}} : \mathsf{p} \to \mathsf{q}; C}\text{TCom} \qquad \frac{\Gamma(\mathsf{p}.k) \in \{!\mathsf{T}, ?\mathsf{T}\} \quad \Gamma \vdash C}{\Gamma \vdash \mathsf{p}.(\mathsf{q}, k)^{\mathsf{T}}; C}\text{TFrame}$$

$$\frac{\Gamma(\mathsf{p}.c) = !\mathsf{T} \quad \Gamma(\mathsf{p}) \vdash s : \mathsf{T} \quad \Gamma \vdash C}{\Gamma \vdash \mathsf{p}.c!s; C}\text{TSend} \qquad \frac{\Gamma(\mathsf{p}.c) = ?\mathsf{T} \quad \Gamma(\mathsf{p}.x) = \mathsf{T} \quad \Gamma \vdash C}{\Gamma \vdash \mathsf{p}.c?x; C}\text{TRecv}$$

Figure 3: Lossy Choreographies, typing rules (selection), see [21, §4].

The frame types $!\mathsf{T}$ and $?\mathsf{T}$ denote a frame that sends a payload of type T and one that receives a payload of type T. We write $\Gamma(\mathsf{p}.c)$ and $\Gamma(\mathsf{p}.x)$ for the type of frame $\mathsf{p}.c$ and of variable $\mathsf{p}.x$, and $\Gamma(\mathsf{p})$ for the environment local to p. Just like we assumed a user-defined local language, we do the same for the local type system, which we assume has local judgements $\Gamma(\mathsf{p}) \vdash s : \mathsf{T}$ extended to account for guards about frame status ($c!$, $c?$, $c?l$) as one would expect (see [21, §4] for details).

For the most part, these rules are fairly intuitive. We report a selection of illustrative derivation rules of the type system in Figure 3, the full system is available in the report [21, §4]. Rule TCom adds both ends of the frame to the environment with the expected send and receive types and rule TFrame checks that the runtime term for frame creation refers to a frame in the environment.[2] Rules TSend and TRecv check that the type of the frame matches the type of the payload and the variable it gets stored on.

Definition 2. We say that a configuration $\langle C, \Sigma, K\rangle$ is *well-typed* if there is a type environment Γ such that (1) $\Gamma \vdash C$; (2) $\Gamma \vdash \Sigma(\mathsf{p}.x) : \Gamma(\mathsf{p}.x)$ for any variable $\mathsf{p}.x \in \Sigma$; (3) $\Gamma(\mathsf{p}.(\mathsf{q}, m)) = !\mathsf{T}$, $\Gamma(\mathsf{q}.(\mathsf{p}, m)) = ?\mathsf{T}$, and $\vdash v : \mathsf{T}$ for any $(\mathsf{p}, \mathsf{q}, m, v) \in K$; (4) Σ is consistent with C.

Our type system enjoys typability preservation: if a configuration is well-typed then so is any configuration that can be reached from it.

Proposition 2 (Typability Preservation)**.** *Given a well-typed configuration $\langle C, \Sigma, K\rangle$ and set of procedure definitions $\mathcal{C}$, if $\langle C, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{C}} \langle C', \Sigma', K'\rangle$, then $\langle C', \Sigma', K'\rangle$ is also well-typed.*

Well-typed configurations enjoy progress: they either do an action or their choreography is $\mathbf{0}$.

Theorem 1 (Progress)**.** *Given a well-typed $\langle C, \Sigma, K\rangle$ and $\mathcal{C}$, $C = \mathbf{0}$ or $\langle C, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{C}} \langle C', \Sigma', K'\rangle$.*

5. Compilation to Process Implementations

We present an EndPoint Projection (EPP) procedure that compiles a choreography to a concurrent implementation in terms of a process calculus, which assumes the same failure model as in LC.

Lossy Processes The target process language, Lossy Processes, is based on Recursive Processes – the textbook target for choreography projection [4]. We extend the semantics so that (i) send and receive operations may fail and (ii) messages are tagged with numeric (frame) identifiers. Numeric frame identifiers act as message sequence numbers. However, the model does not offer any mechanism for maintaining counters synchronised among connected processes nor can such mechanism be programmed since these counters are inaccessible. The only way to maintain synchrony is to write programs where frame declarations are carefully matched at communicating process. In the next section we leverage the fact that processes projected from a choreography enjoy this by construction.

The next grammar allows for writing process programs, or behaviours (B).

$$B ::= L; B \mid \mathbf{0} \qquad s ::= l \mid e \qquad b ::= e \mid c \qquad c ::= k \mid (\mathsf{p}, m)$$
$$L ::= (\mathsf{p}, k) \mid c!s \mid c?x \mid x := s \mid \texttt{if}\, b\, \texttt{then}\, B_1\, \texttt{else}\, B_2 \mid c\&\{l_1 : B_i\}_{i\in I} \mid X\langle \vec{\mathsf{p}}; \vec{c}\rangle$$

[2] For conciseness, neither rule check for misuses of runtime terms like $\mathsf{p}.(\mathsf{q}, k)^{\mathsf{T}}; \mathsf{p}.(\mathsf{q}, k)^{\mathsf{T}}$ or $(k, k')^{\mathsf{T}} : \mathsf{p} \to \mathsf{q}; \mathsf{p}.(\mathsf{q}, k)^{\mathsf{T}}$ as these are already excluded by Definition 1; these checks amount to adding the premises $k \notin \phi^{\mathsf{p}}_{\mathsf{q}}(C)$ and $k' \notin \phi^{\mathsf{q}}_{\mathsf{p}}(C)$.

$$\frac{m = \Sigma(\mathsf{p}.\mathbf{fc}.\mathsf{q}) \quad \Sigma' = \Sigma[\mathsf{p}.\mathbf{fc}.\mathsf{q} \mapsto m+1, \mathsf{p}.\mathbf{fb}.\mathsf{q}.m \mapsto \bot]}{\langle \mathsf{p}[(\mathsf{q},k);B], \Sigma, K\rangle \xrightarrow{\mathsf{p}\to\mathsf{q}}_{\mathcal{B}} \langle \mathsf{p}[B[(\mathsf{q},m)/k]], \Sigma', K\rangle}\text{PFrame}$$

$$\frac{\sigma(\mathsf{p}) \vdash S \downarrow v \quad \Sigma' = \sigma[\mathsf{p}.\mathbf{fb}.\mathsf{q}.m \mapsto \checkmark]}{\langle \mathsf{p}[(\mathsf{q},m)!s;B], \Sigma, K\rangle \xrightarrow{\mathsf{p}.(\mathsf{q},m)\to}_{\mathcal{B}} \langle \mathsf{p}[B], \Sigma', K \cup \{(\mathsf{p},\mathsf{q},m,v)\}\rangle}\text{PSend}$$

$$\frac{\Sigma(\mathsf{p}.\mathbf{fb}.\mathsf{q}.m) \in \{v, v\checkmark\}}{\langle \mathsf{p}[(\mathsf{q},m)?x;B], \Sigma, K\rangle \xrightarrow{\to\mathsf{p}.(\mathsf{q},m)}_{\mathcal{B}} \langle \mathsf{p}[B], \Sigma[\mathsf{p}.\mathbf{fb}.\mathsf{q}.m \mapsto v, x \mapsto v\checkmark], K\rangle}\text{PRecv}$$

$$\frac{\langle N, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{B}} \langle N', \Sigma', K'\rangle}{\langle N \mid M, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{B}} \langle N' \mid M, \Sigma', K'\rangle}\text{NPar} \qquad \frac{}{\langle \mathsf{p}[(\mathsf{q},m)!s;B], \Sigma, K\rangle \xrightarrow{\mathsf{p}.(\mathsf{q},m)\to}_{\mathcal{B}} \langle \mathsf{p}[B], \Sigma, K\rangle}\text{PSendFail}$$

$$\frac{\Sigma(\mathsf{p}.\mathbf{fb}.\mathsf{q}.m) = \bot}{\langle \mathsf{p}[(\mathsf{q},m)?x;B], \Sigma, K\rangle \xrightarrow{\to\mathsf{p}.(\mathsf{q},m)}_{\mathcal{B}} \langle \mathsf{p}[B], \Sigma, K\rangle}\text{PRecvFail} \qquad \frac{}{\langle N, \Sigma, K \uplus \{(\mathsf{p},\mathsf{q},m,v)\}\rangle \xrightarrow{\tau}_{\mathcal{B}} \langle N, \Sigma, K\rangle}\text{NLoss}$$

$$\frac{}{\langle N, \Sigma, K \uplus \{(\mathsf{p},\mathsf{q},m,v)\}\rangle \xrightarrow{\tau}_{\mathcal{B}} \langle N, \Sigma[\mathsf{q}.\mathbf{fb}.\mathsf{p}.m \mapsto v], K\rangle}\text{NDel}$$

Figure 4: Process model, operational semantics (selection), see [21, §5].

Term (p, k) represent the creation of a new frame. Terms $c!s$ and $c?x$ express send and receive actions for c. Term $c\&\{l_1 : B_i\}_{i\in I}$ describes a branching based on a label communicated for c, if any label l_i has been successfully received then, the process proceeds with the corresponding behaviour B_i otherwise it proceeds with the one labelled with DEFAULT which is reserved for this purpose and cannot be sent. If $I = \emptyset$, then the term is simply discarded. Guard c states that the last communication action for frame c has been successfully completed. Remaining terms are standard.

Borrowing notation from [4], a network (of processes) N is a map from process names to process behaviours such that only finitely many are mapped to behaviours other than $\mathbf{0}$; the set of these processes is called *domain* of the network. We write $\mathsf{p}[B]$ for the network where process p has behaviour B and every other process has behaviour $\mathbf{0}$. The parallel composition of two networks N and N' with disjoint domains, $N \mid N'$, simply assigns to each process its behaviour in the network defining it.

The semantics of networks is given as an LTS. States are configurations $\langle N, \Sigma, K\rangle$ where N is a network and Σ and K are as in the LTS of LC. The transition relation uses the same labels of LC and is parameterised in a set $\mathcal{B}$ of procedure definitions. We report a selection of the core rules in Figure 4 which we discuss below—the remaining ones are standard and can be found in [21, §5].

Rule PFRAME models the creation at p of a new frame for q using the value m of the corresponding counter. Rule PSEND models the sending of a message v to frame (q, m), marking the frame as sent and rule PSENDFAIL models the case where the send fails. Dually, rules PRECV and PRECVFAIL model the successful reception of a message from frame (q, m), marking the frame as received and storing in x the received value v, and the case where the receive fails, respectively. Rules NLOSS and NDEL model the loss and delivery of messages by the network like rules CLOSS and CDEL.

EndPoint Projection Given a choreography C, the projected behaviour of process p in C is defined as $[\![C]\!]_{\mathsf{p}}$ where $[\![-]\!]_{\mathsf{p}}$ is the partial function defined by structural recursion in Figure 5. Each case in the definition follows the intuition of writing, for each choreographic term, the local actions performed by the given process. For instance, $\mathsf{p}.c!s$ is skipped during the projection of any process but p, for which the send action $c!s$ is produced. Cases for receiving, procedure calls, and frame creation are similar. The projection of frame declaration expands it to two frame creation runtime terms similarly to its semantics, ensuring that as long as the starting configuration is consistent, the frame counters will remain synchronised for the entire execution. The case for conditionals combines the normal conditionals and branching over labels selection but otherwise follows the standard approach (see e.g., [3, 4, 26, 30–32]). The (partial) merging operator $\sqcup$ from [3] is used to merge the behaviour of a process that does not know (yet) which branch has been chosen by the the process evaluating the guard. Intuitively, $B \sqcup B'$ behaves as B and B' up to branching, where branches of B or B' with distinct

$$\llbracket (k,k')^{\mathsf{T}} : \mathsf{p} \to \mathsf{q}; C \rrbracket_{\mathsf{r}} \triangleq \llbracket \mathsf{p}.(\mathsf{q},k)^{\mathsf{T}}; \mathsf{q}.(\mathsf{p},k')^{\mathsf{T}}; C \rrbracket_{\mathsf{r}} \qquad \llbracket \mathsf{p}.c!s; C \rrbracket_{\mathsf{r}} \triangleq \begin{cases} c!s; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{else} \end{cases}$$

$$\llbracket \mathsf{p}.(\mathsf{q},k)^{\mathsf{T}}; C \rrbracket_{\mathsf{r}} \begin{cases} (\mathsf{q},k); \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{else} \end{cases} \qquad \llbracket \mathsf{p}.c?x; C \rrbracket_{\mathsf{r}} \triangleq \begin{cases} c?x; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{else} \end{cases}$$

$$\llbracket \mathsf{p}.x := s; C \rrbracket_{\mathsf{r}} \triangleq \begin{cases} x := s; \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{else} \end{cases} \quad \llbracket X\langle \vec{\mathsf{p}}; \vec{c} \rangle; C \rrbracket_{\mathsf{r}} \triangleq \begin{cases} X_i(\vec{\mathsf{p}} \setminus \mathsf{r}; \vec{c}); \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} = \vec{\mathsf{p}}[i] \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{else} \end{cases}$$

$$\llbracket \vec{\mathsf{q}}{:}X[\vec{\mathsf{p}}; \vec{c}].C'; C \rrbracket_{\mathsf{r}} \triangleq \begin{cases} X_i(\vec{\mathsf{p}} \setminus \mathsf{r}; \vec{c}); \llbracket C' \rrbracket_{\mathsf{r}} & \text{if } \mathsf{r} \in \vec{\mathsf{q}} \text{ and } \mathsf{r} = \vec{\mathsf{p}}[i] \\ \llbracket C \rrbracket_{\mathsf{r}} & \text{else} \end{cases} \qquad \llbracket \mathbf{0} \rrbracket_{\mathsf{r}} \triangleq \mathbf{0}$$

$$\left\llbracket \begin{array}{l} \texttt{if}\, \mathsf{p}.b\, \texttt{then}\, C_1 \\ \texttt{else}\, C_2; C \end{array} \right\rrbracket_{\mathsf{r}} \triangleq \begin{cases} (c\&\{l : \llbracket C_1 \rrbracket_{\mathsf{r}}, \textsc{default} : \llbracket C_2 \rrbracket_{\mathsf{r}}\}); \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{p}.b = \mathsf{r}.c?l \\ (\texttt{if}\, e\, \texttt{then}\, \llbracket C_1 \rrbracket_{\mathsf{r}}\, \texttt{else}\, \llbracket C_2 \rrbracket_{\mathsf{r}}); \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{p}.b = \mathsf{r}.e \\ (\texttt{if}\, c\, \texttt{then}\, \llbracket C_1 \rrbracket_{\mathsf{r}}\, \texttt{else}\, \llbracket C_2 \rrbracket_{\mathsf{r}}); \llbracket C \rrbracket_{\mathsf{r}} & \text{if } \mathsf{p}.b = \mathsf{r}.c! \text{ or } \mathsf{p}.b = \mathsf{r}.c? \\ (\llbracket C_1 \rrbracket_{\mathsf{r}} \sqcup \llbracket C_2 \rrbracket_{\mathsf{r}}); \llbracket C \rrbracket_{\mathsf{r}} & \text{else} \end{cases}$$

Figure 5: EndPoint Projection, process behaviours.

labels are also included. One proceeds homomorphically (e.g., $k!e; B \sqcup k!e; B'$ is $k!e; (B \sqcup B')$) on all terms but branches which are handled defining the merge of $c\&\{l_i : B_i\}_{i\in I}; B$ and $c\&\{l_j : B'_j\}_{j\in J}; B'$ as $c\&\{l_h : B''_h\}_{h\in H}; (B \sqcup B')$ where $\{l_h : B''_h\}_{h\in H}$ is the union of $\{l_i : B_i\}_{i\in I\setminus J}$, $\{l_j : B'_j\}_{j\in J\setminus I}$, and $\{l_g : B_g \sqcup B'_g\}_{g\in I\cap J}$. A choreography C is projected to $\llbracket C \rrbracket \triangleq \lambda \mathsf{p}.\, \llbracket C \rrbracket_{\mathsf{p}}$ and procedure definitions as:

$$\llbracket \mathcal{C} \rrbracket \triangleq \bigcup_{X(\mathsf{p}_1,\dots \mathsf{p}_n; \vec{k}) = C \in \mathcal{C}} \left\{ X_i(\mathsf{p}_1, \dots, \mathsf{p}_{i-1}, \mathsf{p}_{i+1}, \dots \mathsf{p}_n; \vec{k}) \llbracket C \rrbracket_{\mathsf{p}_i} \;\middle|\; 1 \le i \le n \right\}.$$

Observe that since a procedure X may be called multiple times on any combination of its arguments it is necessary to project the behaviour of each possible process parameter p_i as the procedure X_i.

There is an operational correspondence between choreographies and their projections, which we can be formulated in the standard way up to the 'branching' relation $\sqsupseteq$ (the natural order induced by merging i.e., $B_1 \sqsupseteq B_2$ iff $B_1 \sqcup B_3 = B_2$ for some B_3 [4]). This relation accounts for the well-known fact in choreographic programming that, after a conditional is executed at the choreography level, some processes get to know about the chosen branch only after a while (through selections) and thus have temporary 'dead code' (branches that are never going to be selected) [4, Chapter 6].

Theorem 2 (EPP). *Given a well-typed $\langle C, \Sigma, K\rangle$ and $\mathcal{C}$ such that $\llbracket C \rrbracket$ and $\llbracket \mathcal{C} \rrbracket$ are defined, then:*

- *If $\langle C, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{C}} \langle C', \Sigma', K'\rangle$ then $\llbracket C, \Sigma, K \rrbracket \xrightarrow{\mu}_{\llbracket \mathcal{C} \rrbracket} N$ such that $\llbracket C' \rrbracket \sqsupseteq N$.*
- *If $\langle \llbracket C \rrbracket, \Sigma, K\rangle \xrightarrow{\mu}_{\llbracket \mathcal{C} \rrbracket} \langle N, \Sigma', K'\rangle$, then $\langle C, \Sigma, K\rangle \xrightarrow{\mu}_{\mathcal{C}} \langle C', \Sigma', K'\rangle$ such that $\llbracket C' \rrbracket \sqsupseteq N$.*

6. Related Work

The work nearest to ours is [16], which extends multiparty session types (MPST) [15] – protocol specifications without data or computation – with optional blocks that can become no-op non-deterministically. While our initial motivation is similar, we address a different setting and see our work as complementary. Our focus is providing guarantees on implementations, and thus choreographies in LC are concrete programs, unlike protocol specifications. Another key difference is that communication in [16] is synchronous – both sender and receiver know if the communication failed – whereas in LC it is asynchronous. This requires defining and analysing send and receive actions separately, as success in sending does not guarantee success in receiving. This separation is is crucial for enabling different recovery strategies for senders and receivers in LC whereas in [16] recovery strategies cannot be specified at all.

Another extension of MPST focuses on process failures (instead of communication failures) [17]. Like LC and unlike [16], it supports asynchronous communication. However, it differs significantly

from LC by assuming some processes cannot fail and that communication between non-failed processes is reliable. In contrast, LC allows messages to arrive in any order or not at all. The failure mode of [17] lies between crash [33] and fail-stop [34, 35] modes, which impose stronger reliability assumptions than to the omission modes of LC [19, 20]. Process failures can be encoded in LC, e.g., via internal nondeterministic choices to proceed or shut down.

In [36], MPST are augmented with controlled exceptions. These are different from communication failures, because they are controlled by the programmer and their propagation is ensured through communications that are assumed reliable. The same difference applies to a later refinement of this approach [37] and a similar extension of session types to exception handling in a functional setting [38].

A different approach to MPST with failures is seen in [39]. Like LC, they allow communication to fail, but when this happens they kill the session where the failure occurred. Similarly to [36], this mechanism for killing sessions assumes a reliable way to communicate failures between processes.

In [27], the authors propose a choreographic language for programming systems where processes are implemented as pools of redundant replicas that may and communicate via reliable multicast. No recovery can be programmed, and there is no presentation of how the approach can be adopted in realistic process models. Therefore, this approach is from from ours and and [16].

Some work on choreographies include choice operators that behave nondeterministically, e.g., $C + C'$, read 'run either C or C'' [3, 4, 15, 31, 40]. These operators do not capture the failure modes we are interested in for two reasons: they are explicitly programmed and thus predictable, and their formalizations assume reliable propagation of choice information among processes.

Our approach of recovering high-level choreographic constructs by wrapping lower-level communication actions (as in Example 1) follows the practice established by Choral, the most expressive implementation of choreographic programming to date [6, 41]. However, Choral does not come with a formal semantics. LC thus offers a useful theory to reason about recovery strategies before they are implemented, which can be done through our accompanying implementation as a Choral library [21].

Recent work explored how choreographic programming can be offered as embedded domain-specific languages in, for example, Haskell and Rust [7, 8, 42, 43]. Differently from Choral, these languages are based on interpretation rather than compilation. We think that our ideas are applicable also to these languages, but it would require updating both the languages and their interpretation functions – whereas our implementation does not require any change to Choral itself, it is 'just a library'.

Our new primitive for communication declaration is reminiscent of the 'cut' operator found in session-typed process calculi for connecting two endpoints of a channel [44–47]. While there is a superficial similarity in that both endpoints and our frames are created in pairs, endpoint pairs represent channels, whereas frame pairs represent single communications. More importantly, the dynamic creation of endpoints requires synchronisation, unlike our frames, which is crucial for our development due to unreliable communication. Session types typically guarantee system-wide progress by restricting communication structures or process topologies, whereas LC does not require such restrictions.

7. Conclusion

Choreographic programming has seen significant advancements over the past decade, but always under the strong assumption of reliable communication [1, 4, 6–8, 11, 23–25, 48–51]. Our study frees the paradigm from this assumption, reaching all the way to standard failure modes and realistic protocols.

Our work covers all failure modes with honest participants. A natural next step is therefore to investigate Byzantine failures modes where participants may act maliciously. Exploring this direction would touch on a realm where possible guarantees are more limited. Therefore, it would be interesting and challenging to understand how close choreographic programming can be brought to the theoretical limit. Another interesting direction is to explore a quantitative semantics of Lossy Choreographies. For instance, in a probabilistic settings, failures are characterised by probability distributions. This would enable reasoning about aspects such as quality of service, throughput, probability of critical failure, etc.

Acknowledgments

Partially supported by Villum Fonden (grant no. 29518), and by the Ministry of Education and Research Centres of Excellence grant TK213 Estonian Centre of Excellence in Artificial Intelligence (EXAI). Co-funded by the European Union (ERC, CHORDS, 101124225). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] F. Montesi, Choreographic Programming, Ph.D. Thesis, IT University of Copenhagen, 2013. URL: https://www.fabriziomontesi.com/files/choreographic-programming.pdf.

[2] R. Needham, M. Schroeder, Using encryption for authentication in large networks of computers, Commun. ACM 21 (1978) 993–999. doi:10.1145/359657.359659.

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

[4] F. Montesi, Introduction to Choreographies, Cambridge University Press, 2023. doi:10.1017/9781108981491.

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

[6] S. Giallorenzo, F. Montesi, M. Peressotti, Choral: Object-oriented choreographic programming, ACM Trans. Program. Lang. Syst. 46 (2024). URL: https://doi.org/10.1145/3632398. doi:10.1145/3632398.

[7] G. Shen, S. Kashiwa, L. Kuper, Haschor: Functional choreographic programming for all (functional pearl), Proc. ACM Program. Lang. 7 (2023) 541–565. URL: https://doi.org/10.1145/3607849. doi:10.1145/3607849.

[8] S. Kashiwa, G. Shen, S. Zare, L. Kuper, Portable, efficient, and practical library-level choreographic programming, CoRR abs/2311.11472 (2023). URL: https://doi.org/10.48550/arXiv.2311.11472. doi:10.48550/ARXIV.2311.11472. arXiv:2311.11472.

[9] L. Cruz-Filipe, F. Montesi, M. Peressotti, A formal theory of choreographic programming, J. Autom. Reason. 67 (2023) 21. URL: https://doi.org/10.1007/s10817-023-09665-3. doi:10.1007/S10817-023-09665-3.

[10] L. Cruz-Filipe, F. Montesi, M. Peressotti, Certifying choreography compilation, in: A. Cerone, P. C. Ölveczky (Eds.), Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings, volume 12819 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 115–133. doi:10.1007/978-3-030-85315-0_8.

[11] A. K. Hirsch, D. Garg, Pirouette: higher-order typed functional choreographies, Proc. ACM Program. Lang. 6 (2022) 1–27. URL: https://doi.org/10.1145/3498684. doi:10.1145/3498684.

[12] J. Å. Pohjola, A. Gómez-Londoño, J. Shaker, M. Norrish, Kalas: A verified, end-to-end compiler for a choreographic language, in: J. Andronick, L. de Moura (Eds.), 13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel, volume 237 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 27:1–27:18. URL: https://doi.org/10.4230/LIPIcs.ITP.2022.27. doi:10.4230/LIPICS.ITP.2022.27.

[13] C. Acay, J. Gancher, R. Recto, A. C. Myers, Secure synthesis of distributed cryptographic applications (technical report), CoRR abs/2401.04131 (2024). URL: https://doi.org/10.48550/arXiv.2401.04131. doi:10.48550/ARXIV.2401.04131. arXiv:2401.04131.

[14] C. Acay, R. Recto, J. Gancher, A. C. Myers, E. Shi, Viaduct: an extensible, optimizing compiler for secure distributed programs, in: S. N. Freund, E. Yahav (Eds.), PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021, ACM, 2021, pp. 740–755. URL: https://doi.org/10.1145/3453483.3454074. doi:10.1145/3453483.3454074.

[15] K. Honda, N. Yoshida, M. Carbone, Multiparty Asynchronous Session Types, J. ACM 63 (2016) 9. URL: http://doi.acm.org/10.1145/2827695. doi:10.1145/2827695.

[16] M. Adameit, K. Peters, U. Nestmann, Session types for link failures, in: FORTE, volume 10321 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 1–16.

[17] M. Viering, R. Hu, P. Eugster, L. Ziarek, A multiparty session typing discipline for fault-tolerant event-driven distributed programming, Proc. ACM Program. Lang. 5 (2021) 1–30. URL: https://doi.org/10.1145/3485501. doi:10.1145/3485501.

[18] V. Hadzilacos, Issues of fault tolerance in concurrent computations (databases, reliability, transactions, agreement protocols, distributed computing), Ph.D. thesis, USA, 1985. AAI8520209.

[19] K. J. Perry, S. Toueg, Distributed agreement in the presence of processor and communication faults, IEEE Transactions on Software Engineering SE-12 (1986) 477–482. doi:10.1109/TSE.1986.6312888.

[20] F. Cristian, Understanding fault-tolerant distributed systems, Commun. ACM 34 (1991) 56–78. URL: https://doi.org/10.1145/102792.102801. doi:10.1145/102792.102801.

[21] E. Graversen, F. Montesi, M. Peressotti, A promising future: Omission failures in choreographic programming, CoRR abs/1712.05465 (2025). URL: https://arxiv.org/abs/1712.05465v3. doi:10.48550/arXiv.1712.05465. arXiv:1712.05465v3.

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

[23] S.-S. Jongmans, P. van den Bos, A predicate transformer for choreographies, in: I. Sergey (Ed.), Programming Languages and Systems, Springer International Publishing, Cham, 2022, pp. 520–547.

[24] M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, J. Mauro, Dynamic choreographies: Theory and implementation, Logical Methods in Computer Science 13 (2017).

[25] L. Cruz-Filipe, E. Graversen, F. Montesi, M. Peressotti, Reasoning about choreographic programs, in: S. Jongmans, A. Lopes (Eds.), Coordination Models and Languages, volume 13908 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 144–162. doi:10.1007/978-3-031-35361-1_8.

[26] L. Cruz-Filipe, F. Montesi, Procedural choreographic programming, in: FORTE, LNCS, Springer, 2017.

[27] H. A. López, F. Nielson, H. R. Nielson, Enforcing availability in failure-aware communicating systems, in: FORTE, volume 9688 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 195–211.

[28] L. Cruz-Filipe, F. Montesi, M. Peressotti, Communication in choreographies, revisited, in: SAC, ACM, 2018. To Appear.

[29] P. A. Bernstein, E. Newcomer, Chapter 8 - two-phase commit, in: P. A. Bernstein, E. Newcomer (Eds.), Principles of Transaction Processing (Second Edition), The Morgan Kaufmann Series in Data Management Systems, second edition ed., Morgan Kaufmann, San Francisco, 2009, pp. 223–244. URL: https://www.sciencedirect.com/science/article/pii/B9781558606234000081. doi:10.1016/B978-1-55860-623-4.00008-1.

[30] L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, N. Yoshida, Global progress in dynamically interleaved multiparty sessions, in: CONCUR, volume 5201 of *LNCS*, Springer, 2008, pp. 418–433.

[31] I. Lanese, C. Guidi, F. Montesi, G. Zavattaro, Bridging the gap between interaction- and process-oriented choreographies, in: SEFM, 2008, pp. 323–332.

[32] L. Cruz-Filipe, F. Montesi, A core model for choreographic programming, in: O. Kouchnarenko, R. Khosravi (Eds.), FACS, volume 10231 of *LNCS*, Springer, 2016. doi:10.1007/978-3-319-57666-4_3.

[33] L. Lamport, M. Fischer, Byzantine generals and transaction commit protocols, Technical Report, Technical Report 62, SRI International, 1982.

[34] R. D. Schlichting, F. B. Schneider, Fail-stop processors: an approach to designing fault-tolerant computing systems, ACM Trans. Comput. Syst. 1 (1983) 222–238. URL: https://doi.org/10.1145/357369.357371. doi:10.1145/357369.357371.

[35] F. B. Schneider, Byzantine generals in action: implementing fail-stop processors, ACM Trans. Comput. Syst. 2 (1984) 145–154. URL: https://doi.org/10.1145/190.357399. doi:10.1145/190.357399.

[36] S. Capecchi, E. Giachino, N. Yoshida, Global escape in multiparty sessions, Mathematical Structures in Computer Science 26 (2016) 156–205. URL: http://dx.doi.org/10.1017/S0960129514000164. doi:10.1017/S0960129514000164.

[37] T. Chen, M. Viering, A. Bejleri, L. Ziarek, P. Eugster, A type theory for robust failure handling in distributed systems, in: FORTE, volume 9688 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 96–113.

[38] S. Fowler, S. Lindley, J. G. Morris, S. Decova, Exceptional asynchronous session types: session types without tiers, Proc. ACM Program. Lang. 3 (2019) 28:1–28:29. URL: https://doi.org/10.1145/3290341. doi:10.1145/3290341.

[39] N. Lagaillardie, R. Neykova, N. Yoshida, Stay safe under panic: Affine rust programming with multiparty session types, in: K. Ali, J. Vitek (Eds.), 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany, volume 222 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 4:1–4:29. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2022.4. doi:10.4230/LIPICS.ECOOP.2022.4.

[40] Z. Qiu, X. Zhao, C. Cai, H. Yang, Towards the theoretical foundation of choreography, in: WWW, ACM, 2007, pp. 973–982.

[41] D. Plyukhin, M. Peressotti, F. Montesi, Ozone: Fully out-of-order choreographies, in: J. Aldrich, G. Salvaneschi (Eds.), 38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria, volume 313 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 31:1–31:28. doi:10.4230/LIPICS.ECOOP.2024.31.

[42] S. Laddad, A. Cheung, J. M. Hellerstein, Suki: Choreographed distributed dataflow in rust, CoRR abs/2406.1473 (2024). URL: https://arxiv.org/abs/2406.1473. doi:10.48550/arXiv.2406.1473. arXiv:2406.1473.

[43] G. Shen, L. Kuper, Toward verified library-level choreographic programming with algebraic effects, CoRR abs/2407.06509 (2024). URL: https://arxiv.org/abs/2407.06509. doi:10.48550/arXiv.2407.06509. arXiv:2407.06509.

[44] V. T. Vasconcelos, Fundamentals of session types, Inf. Comput. 217 (2012) 52–70.

[45] M. Carbone, S. Lindley, F. Montesi, C. Schürmann, P. Wadler, Coherence generalises duality: A logical explanation of multiparty session types, in: CONCUR, volume 59 of *LIPIcs*, Schloss Dagstuhl, 2016, pp. 33:1–33:15.

[46] W. Kokke, F. Montesi, M. Peressotti, Better late than never: a fully-abstract semantics for classical processes, Proc. ACM Program. Lang. 3 (2019) 24:1–24:29. URL: https://doi.org/10.1145/3290337. doi:10.1145/3290337.

[47] W. Kokke, F. Montesi, M. Peressotti, Taking linear logic apart, in: T. Ehrhard, M. Fernández, V. de Paiva, L. T. de Falco (Eds.), Proceedings Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity-TLLA@FLoC 2018, Oxford, UK, 7-8 July 2018., volume 292 of *Electronic Proceedings in Theoretical Computer Science*, Open Publishing Association, 2018, pp. 90–103. doi:10.4204/EPTCS.292.5.

[48] M. Carbone, F. Montesi, Deadlock-freedom-by-design: multiparty asynchronous global programming, in: POPL, ACM, 2013, pp. 263–274.

[49] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, G. Zavattaro, Foundations of session types and behavioural contracts, ACM Comput. Surv. 49 (2016) 3:1–3:36.

[50] L. Cruz-Filipe, E. Graversen, L. Lugović, F. Montesi, M. Peressotti, Functional choreographic programming, in: H. Seidl, Z. Liu, C. Pasareanu (Eds.), Theoretical Aspects of Computing - ICTAC 2022 - 19th International Colloquium, Tbilisi, Georgia, September 27-29, 2022, Proceedings, Lecture Notes in Computer Science, Springer, 2022, pp. 212–237. doi:10.1007/978-3-031-17715-6_15.

[51] L. Cruz-Filipe, E. Graversen, L. Lugović, F. Montesi, M. Peressotti, Modular compilation for higher-order functional choreographies, in: K. Ali, G. Salvaneschi (Eds.), 37th European Conference on Object-Oriented Programming (ECOOP 2023), volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2023, pp. 7:1–7:37. doi:10.4230/LIPIcs.ECOOP.2023.7.