First-Person Choreographic Programming with Continuation-Passing Communications

Sung-Shik Jongmans[0000−0002−4394−8745]

University of Groningen, the Netherlands

Abstract. Choreographic programming (CP) is a method to implement distributed systems that ensures communication deadlock freedom by design. To use CP, though, the number of processes and the network among them must be known statically. Often, that information is known only dynamically. Thus, existing CP languages cannot be used to implement *process-parametric* distributed systems.
This paper introduces *first-person* choreographic programming (1CP) to support the implementation of process-parametric distributed systems while also ensuring communication deadlock freedom. We present both a design of 1CP (new calculus, formalised in Isabelle/HOL) and an implementation (new language and tooling, integrated in VS Code).

1 Introduction

1.1 Background: Choreographic Programming

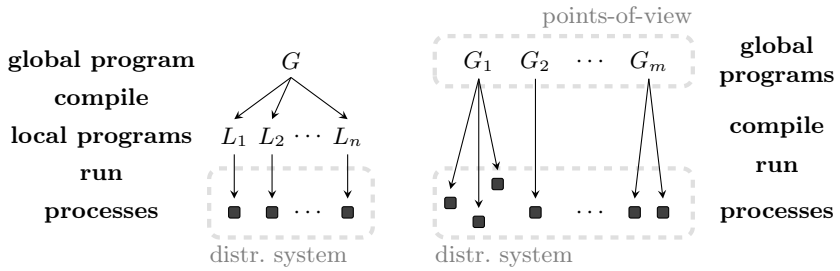
Implementing distributed systems is hard. One of the challenges is to "prove"—broadly construed—the absence of *communication deadlocks* among message-passing processes. A communication deadlock is a form of misbehaviour that arises when all processes get stuck in cyclic communication dependencies.

Choreographic programming [3, 5, 39] is a top–down method to make the implementation of distributed systems easier. Fig. 1a visualises the idea:

1. First, construct a **global program** G ("choreography"). A global program defines the behaviour of all n processes in the distributed system, collectively.

   Example 1. The *TwoBuyer* distributed system consists of processes *Buyer1* (**B1**), *Buyer2* (**B2**), and *Seller* (**S**). As introduced by Honda et al. [30]: "Buyer1 and Buyer2 wish to buy an expensive book from Seller by combining their money. Buyer1 sends the title of the book to Seller, Seller sends to both Buyer1 and Buyer2 its quote, Buyer1 tells Buyer2 how much she can pay, and Buyer2 either accepts the quote or rejects the quote by notifying Seller." The following global program defines the behaviour of Buyer1, Buyer2, and Seller, collectively (excerpt):

$$
\begin{aligned}
G^{\text{TwoBuyer}} = {} & \texttt{B1."foo"} \twoheadrightarrow \texttt{S.title}\,; \\
& \texttt{S.quote(title)} \twoheadrightarrow [\texttt{B1},\texttt{B2}]\texttt{.x}\,; && \textit{// Seller computes the quote} \\
& \texttt{B1.(x / 3)} \twoheadrightarrow \texttt{B2.y}\,; && \textit{// Buyer1 contributes } 1/3 \\
& \textbf{if}\ \texttt{B2.(x - y < x / 2)}\ (\ldots)\ (\ldots) && \textit{// Buyer2 contribs. at most } 1/2
\end{aligned}
$$



(a) Existing: Third-person approach (b) New: First-person approach

Fig. 1: Choreographic programming

Global program $p.E \twoheadrightarrow q.y$ defines a *communication* of (the value of) expression E, from the process named p, to the process named q, into variable y. Global programs **if** $p.E$ G_1 G_2 and $G_1 \,;\, G_2$ define *conditional choice*, at the process named p, and *sequencing*. We write $p.E \twoheadrightarrow [q_1 \ldots, q_n].y$ as a shorthand for $p.E \twoheadrightarrow q_1.y \,;\, \ldots \,;\, p.E \twoheadrightarrow q_n.y$. □

2. Next, statically decompose the global program into **local programs** $L_1, L_2, \ldots, L_n$, by **compiling** G separately for each of the n processes. A local program defines the behaviour of one process, individually.

   Example 2. The following local programs, compiled from G^{TwoBuyer}, define the behaviour of Buyer1, Buyer2, and Seller, individually (excerpts):

   $$L^{\text{TwoBuyer}}_{\texttt{B1}} = \texttt{S!"foo"} \,;\, \texttt{S?x} \,;\, \texttt{B2!(x / 2)}$$
   $$L^{\text{TwoBuyer}}_{\texttt{B2}} = \texttt{S?x} \,;\, \texttt{B1?y} \,;\, \textbf{if } \texttt{(x - y < x / 2)}\ (\ldots)\ (\ldots)$$
   $$L^{\text{TwoBuyer}}_{\texttt{S}} = \texttt{B1?title} \,;\, \texttt{B1!quote(title)} \,;\, \texttt{B2!quote(title)} \,;\, (\ldots)$$

   Local programs $q!E$ and $p?y$ define a *send* to the process named q and a *receive* from the process named p. □

3. Last, dynamically (re)compose the local programs as **processes** into a distributed system by **running** them together, in parallel.

The main result of choreographic programming is that, if the compilation of G succeeds for each process, then the run of $L_1, L_2, \ldots, L_n$ is free of communication deadlocks. That is, we enjoy *communication deadlock freedom by design*.

Early work on choreographic programming is due to Carbone et al. [3,4] and Carbone–Montesi [5]; substantial progress has been made since. For instance, Montesi and Yoshida developed a theory of compositional choreographic programming [40]; Carbone et al. studied connections with linear logic [2, 6, 7]; Dalla Preda et al. combined choreographic programming with dynamic adaptation [45–47]; Cruz-Filipe et al. presented a technique to extract global programs from local programs [14]; Giallorenzo et al. studied a correspondence with multitier languages [26]. Furthermore, theoretical results are supported in practice by several tools, including Chor [5], Choral [25,26], and HasChor [50].

1.2 Open Problem: Process-Parametric Distributed Systems

Central to the compilation of global programs is the concept of static *projection*. Formally, static projection is a function $\upharpoonright$ that consumes a global program G and a name p as input, and that produces a local program $G \upharpoonright p$ as output.

Essentially, to decompose global program G into local programs, the compiler statically projects G onto each name that occurs in G. In this way, *the number of names* that occur in G at compile-time fully determines *the number of processes* that occur in the distributed system at run-time. In many distributed systems, though, the number of processes that occur at run-time is known only dynamically, not statically. Such distributed systems cannot be implemented using existing choreographic programming languages.

Example 3. An *AnyBuyer* distributed system consists of processes *Buyer1* (**B1**), *Buyer2* (**B2**), ..., *Buyern* (**B***n*), and *Seller* (**S**). It generalises the TwoBuyer distributed system from two buyers to any number. Each buyer contributes a fraction of the quote; Buyer*n* decides to accept/reject. The following global programs define the behaviour of all processes, collectively, for $n \in \{3, 4\}$ (excerpts):

$G^{\text{ThreeBuyer}} =$

```
B1."foo" -> S.title ;
S.quote(title) -> [B1, B2, B3].x ;
B1.(x / f)     -> B2.y ;
B2.(x / f + y) -> B3.y ;
if B3.(x - y < x / f) (...) (...)
```

$G^{\text{FourBuyer}} =$

```
B1."foo" -> S.title ;
S.quote(title) -> [B1, B2, B3, B4].x ;
B1.(x / f)     -> B2.y ;
B2.(x / f + y) -> B3.y ;
B3.(x / f + y) -> B4.y ;
if B4.(x - y < x / f) (...) (...)
```

Each buyer has its own variables `x`, `y`, and `f` (initialised when the run begins).

Thus, for each n, a separate global program needs to be written: no existing choreographic programming language supports *parametrisation* in n. However, n is typically known only dynamically. As a result, realistically, AnyBuyer cannot be implemented using choreographic programming. □

Generally, existing choreographic programming languages require not only the number of processes to be known statically (always one per name), but also *the network* among them (always fully connected). These constraints seriously limit the applicability of choreographic programming. Thus, the **open problem** addressed in this paper is how to support **process-parametric distributed systems**, in which such information is known only dynamically. Related work suggests that this is difficult: in *multiparty session typing* (MPST) [30], in which static projection is used to decompose *global types* into *local types*, extensions for process parametricity lose decidability [51], soundness [41], or expressiveness [8].

1.3 This Paper

Whereas the decomposition of global types in MPST *must* happen statically (i.e., local types are used at compile-time), the decomposition of global programs in

choreographic programming *may*—in principle—happen dynamically (i.e., local programs are used only at run-time). That is, in contrast to MPST, there is no fundamental reason in choreographic programming to insist on *static* projection. This insight paves the way to a completely different approach to choreographic programming based on *dynamic* projection. Exploring this approach is the aim of this paper. To summarise the contributions:

- **What:** A novel approach, called *first-person choreographic programming*
- **How:** A new choreographic primitive, called *continuation-passing communication*, that enables *dynamic projection* (at run-time) of choreographies that are still *statically well-typed* (at compile-time)
- **Why:** To implement *process-parametric distributed systems* using choreographic programming, including *communication deadlock freedom by design*

Concretely, we present both a design of first-person choreographic programming (new calculus, formalised in Isabelle/HOL) and an implementation (new language and tooling, integrated in VS Code). Sect. 2 presents an overview. Sect. 3 presents the design. Sect. 4 presents the implementation (including type checking and interpretation; without code generation). The paper is concluded in Sect. 5.

2 Overview

2.1 Design: A Tour of First-Person Choreographic Programming

In existing choreographic programming languages, global programs are written from the *third-person* point-of-view *of the programmer*. For instance, $\mathtt{A.5} \twoheadrightarrow \mathtt{B.x}$ means "a communication from Alice to Bob" from the programmer's point-of-view. At heart, the key novelty of this paper is to shift the narrative perspective in global programs to the *first-person* points-of-view *of the processes*. For instance, $\mathtt{5} \twoheadrightarrow \mathtt{B.x}$ means "a communication from *me* to Bob" from Alice's point-of-view. This shift in perspective is instrumental for us to support process-parametricity.

We call the existing approach *third-person choreographic programming* (3CP) and our new approach *first-person choreographic programming* (1CP). Fig. 1b visualises the idea. We clarify it below and introduce our calculus along the way.

1. First, construct global programs $G_1, G_2, \ldots, G_m$. Like in 3CP, each global program in 1CP defines the behaviour of all processes, collectively. However:
   - In 3CP, a *single* global program reflects *multiple* points-of-view, each taken by *one* process.
   - In 1CP, in contrast, *multiple* global programs each reflect a *single* point-of-view, taken by *one-or-more* processes (process parametricity).

   Example 4. To clarify the terminology and difference between 3CP and 1CP:
   - In 3CP, TwoBuyer consists of three processes (Buyer1, Buyer2, Seller), and three points-of-view (Buyer1, Buyer2, Seller; we overload the process names when the correspondence is one-to-one), reflected by G^{TwoBuyer}.

– In 1CP, AnyBuyer consists of n+1 processes (Buyer1, ..., Buyer$n$, Seller), but three points-of-view (Buyer1, Buyer$\geq$2, Seller), reflected by three global programs. Thus, a single point-of-view (Buyer$\geq$2) is taken by more than one process (Buyer2, ..., Buyer$n$). We note that Buyer1 has a different point-of-view than Buyer$i$, for any $i \geq 2$, as Buyer1 is initially *active* (sends a message), whereas Buyer$i$ is initially *reactive* (awaits a message). In our calculus, the following global programs reflect this:

$$
\begin{array}{ll}
G^{\text{AnyBuyer}}_{\mathtt{B1}} = \ldots \text{ (Exmp. 5)} & \text{// } \textit{Point-of-view: Buyer1} \\
G^{\text{AnyBuyer}}_{\mathtt{Bi}} = \mathbf{skip} & \text{// } \textit{Point-of-view: Buyer}{\geq}2 \\
G^{\text{AnyBuyer}}_{\mathtt{S}} = \mathbf{skip} & \text{// } \textit{Point-of-view: Seller}
\end{array}
$$

Global program **skip** defines a *no-op*. Thus, $G^{\text{AnyBuyer}}_{\mathtt{Bi}}$ and $G^{\text{AnyBuyer}}_{\mathtt{S}}$ define the absence of initial behaviour. □

In more detail, each global program is written in terms of *continuation-passing communications* and *choreographic procedures*.

Example 5. To demonstrate continuation-passing communications, in our calculus, the following global program reflects the point-of-view of Buyer1:

$G^{\text{AnyBuyer}}_{\mathtt{B1}} =$

```
"foo" ⇾ S.title ▷                    // Active: Buyer1
  quote(title) ⇾ B1.x ▷              // Active: Seller
    (x / f) ⇾ neigh.y ▷               // Active: Buyer1
      contrib ;                      // Active: Buyer2
  foreach_Bi neigh ∈ neighs do       // Active: Seller
    quote(title) ⇾ neigh.x ▷         // Active: Seller
      contrib                        // Active: Buyeri, for any i ≥ 2
```

Global program $E \twoheadrightarrow q.y \triangleright G$ defines a *continuation-passing communication*
– of (the value of) expression E, from an implicit *active process*,
– to the explicit *reactive process* named q, into variable y,
– followed by the asynchronous execution, by q, of global program G, written from q's point-of-view.

Thus, as the first active process, Buyer1 passes `"foo"` and a continuation—written from Seller's point-of-view—to Seller. When Seller receives, as the next active process, it passes `quote(title)` and a continuation to Buyer1.

– When Buyer1 receives, as one of the now-active processes, it passes `x / f` and a continuation to `neigh` (a reference to Buyer2). When Buyer2 receives, it calls choreographic procedure `contrib`.
– In parallel, Seller—still active—passes `quote(title)` and a continuation to each `neigh` in `neighs` (a list of references to Buyer2, ..., Buyer$n$).

This example demonstrates that, due to the asynchronous execution of continuations, multiple processes can become active in parallel. □

Intuitively, each occurrence of "▷" signifies a lexical context switch in which the previous active process passes a message and a continuation to the next active process.[1] In this way, continuation passing is a form of *process-driven* dynamic projection: the processes at run-time tell each other how to behave (from their first-person points-of-view), instead of a choreographic programming tool at compile-time (from its third-person point-of-view).

Example 6. To demonstrate choreographic procedures, in our calculus, the following *choreographic libraries* map choreographic procedure *names* to *bodies*, for each of the three points-of-view in AnyBuyer:

$$\begin{aligned}
&\Psi_{\texttt{B1}}^{\text{AnyBuyer}} = \emptyset && \text{// } \textit{Point-of-view: Buyer1}\\
&\Psi_{\texttt{Bi}}^{\text{AnyBuyer}} = \{\texttt{contrib} \mapsto G_{\texttt{contrib}}^{\text{AnyBuyer}}\} && \text{// } \textit{Point-of-view: Buyer}{\geq}2\\
&\Psi_{\texttt{S}}^{\text{AnyBuyer}} = \emptyset && \text{// } \textit{Point-of-view: Seller}
\end{aligned}$$

Thus, Buyer1 and Seller do not have choreographic procedures, while Buyer$i$, for any $i \geq 2$, has `contrib`. The following global program defines its body:

$$\begin{aligned}
G_{\texttt{contrib}}^{\text{AnyBuyer}} = \ &\textbf{if } \texttt{(x != 0 \&\& y != 0)}\\
&\quad \textbf{if } \texttt{last}\\
&\quad\quad \textbf{if } \texttt{(x - y < x / f)}\ (\ldots)\ (\ldots)\\
&\quad\quad \texttt{(x / f + y)} \multimap \texttt{neigh.y} \triangleright \texttt{contrib}
\end{aligned}$$

Thus, each time Buyer$i$ *calls* `contrib`, it checks if `x` (quote) and `y` (contribution up to Buyer$i{-}1$) are already set; `0` indicates unset. If so, and if `last` (flag to indicate whether or not Buyer$i$ is the n-th buyer) is true, then Buyer$i$ decides to accept/reject the quote; otherwise, using `f` (denominator of the fraction that Buyer$i$ can contribute), Buyer$i$ passes `x / f + y` (total contribution up to Buyer$i$) and a continuation to `neigh` (reference to Buyer$i{+}1$).
Putting the pieces in this example and in Exmp. 5 together, Buyer$i$ calls `contrib` twice: once in a continuation passed to it by Seller (and the incoming message sets `x`), and once in a continuation passed to it by Buyer$i{-}1$ (and the incoming message sets `y`), but in any order (due to asynchrony). □

2. Next, **at compile-time**, statically analyse each global program, including the bodies of choreographic procedures, by means of *type checking*. Our main result (Thm. 1) is that if each global program is well-typed, then any *properly initialised* run of those global programs is free of communication deadlocks. Proper initialisation means that the initial values of variables are well-typed; this involves a lightweight check right at the beginning of a run. Thus, just as third-person choreographic programming, first-person choreographic programming guarantees communication deadlock freedom by design.

[1] Passing code around in this way is a programming abstraction. It can be "compiled away" and avoided at run-time; see the end of Sect. 4.2.

Example 7. Intuitively, to ensure that none of the processes in AnyBuyer get stuck, they must have the following variables and choreographic procedures:
- Buyer1: `x`, `f` (numbers) and `neigh` (process reference);
- Buyer2, ..., Buyer$n$: `x`, `y`, `f` (numbers), `last` (boolean), `neigh` (process reference), and `contrib` (choreographic procedure);
- Seller: `title` (string) and `neighs` (list of process references).

In our calculus, the following *data/choreographic type environments* capture these assumptions, for each of the three points-of-view:

$$\Gamma_{\mathtt{B1}}^{\text{AnyBuyer}} = \{\mathtt{x} \mapsto \mathtt{Nat}, \mathtt{f} \mapsto \mathtt{Nat}, \mathtt{neigh} \mapsto \mathtt{Proc\langle Bi\rangle}\}$$
$$\Gamma_{\mathtt{Bi}}^{\text{AnyBuyer}} = \{\mathtt{x} \mapsto \mathtt{Nat}, \mathtt{y} \mapsto \mathtt{Nat}, \mathtt{f} \mapsto \mathtt{Nat}, \mathtt{last} \mapsto \mathtt{Bool}, \mathtt{neigh} \mapsto \mathtt{Proc\langle Bi\rangle}\}$$
$$\Gamma_{\mathtt{S}}^{\text{AnyBuyer}} = \{\mathtt{title} \mapsto \mathtt{Str}, \mathtt{neighs} \mapsto \mathtt{List\langle Proc\langle Bi\rangle\rangle}\}$$
$$\Delta_{\mathtt{B1}}^{\text{AnyBuyer}} = \emptyset \qquad \Delta_{\mathtt{Bi}}^{\text{AnyBuyer}} = \{\mathtt{contrib} \mapsto \mathtt{Chor[Bi]}\} \qquad \Delta_{\mathtt{S}}^{\text{AnyBuyer}} = \emptyset$$

Using these environments, we can statically prove that $G_{\mathtt{B1}}^{\text{AnyBuyer}}$, $G_{\mathtt{Bi}}^{\text{AnyBuyer}}$, $G_{\mathtt{S}}^{\text{AnyBuyer}}$, and $G_{\mathtt{contrib}}^{\text{AnyBuyer}}$ (all defined in previous examples) are well-typed; we do not need to know n to make this judgment. Thus, any properly initialised run—for any n—is free of communication deadlocks.
We note that the typing rules do not guarantee functional correctness. For instance, if both Buyer2 and Buyer3 have a reference to Buyer4 in `neigh`, then the system is deadlock-free (well-typed) but functionally incorrect. □

3. Last, **at run-time**, dynamically create a *configuration* for each process, and compose those configurations into a distributed systems by running them together, in parallel. If the point-of-view of a process is p, then its initial configuration is of the shape (Φ, Ψ, G), where:
   - Φ defines the initial values of the variables of the process;
   - Ψ defines the choreographic library from point-of-view p;
   - G defines the behaviour of all processes, collectively, from point-of-view p.

   Thus, all processes that have the same point-of-view, have the same choreographic library and the same global program in their initial configurations. To be able to reference different processes that have the same point-of-view, we use *indexed names* of the shape $p\mathtt{[}i\mathtt{]}$, where p is a point-of-view, and i is an index. We write just p as a shorthand for $p\mathtt{[0]}$.

   Example 8. The following configurations initialise AnyBuyer for $n = 4$:

$$(\Phi_{p\mathtt{[}i\mathtt{]}}, \Psi_p^{\text{AnyBuyer}}, G_p^{\text{AnyBuyer}}) \quad \text{for} \quad p\mathtt{[}i\mathtt{]} \in \{\mathtt{B1}, \mathtt{Bi[2]}, \mathtt{Bi[3]}, \mathtt{Bi[4]}, \mathtt{S}\}$$

$$\begin{aligned}
\text{where} \quad \Phi_{\mathtt{B1}} &= \{\mathtt{neigh} \mapsto \mathtt{Bi[2]}, \mathtt{x} \mapsto \mathtt{0}, \mathtt{f} \mapsto \mathtt{3}\} \\
\Phi_{\mathtt{Bi[2]}} &= \{\mathtt{neigh} \mapsto \mathtt{Bi[3]}, \mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \mathtt{0}, \mathtt{f} \mapsto \mathtt{5}, \mathtt{last} \mapsto \mathtt{false}\} \\
\Phi_{\mathtt{Bi[3]}} &= \{\mathtt{neigh} \mapsto \mathtt{Bi[4]}, \mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \mathtt{0}, \mathtt{f} \mapsto \mathtt{8}, \mathtt{last} \mapsto \mathtt{false}\} \\
\Phi_{\mathtt{Bi[4]}} &= \{\mathtt{neigh} \mapsto \mathtt{Bi[4]}, \mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \mathtt{0}, \mathtt{f} \mapsto \mathtt{2}, \mathtt{last} \mapsto \mathtt{true}\ \} \\
\Phi_{\mathtt{S}} &= \{\mathtt{neighs} \mapsto \mathtt{[Bi[2], Bi[3], Bi[4]]}, \mathtt{title} \mapsto \mathtt{""}\}
\end{aligned}$$

   Each $\Phi_{p\mathtt{[}i\mathtt{]}}$ is well-typed in $\Gamma_p^{\text{AnyBuyer}}$; this is a near-trivial check. □

Example 9. To demonstrate a run, we show the first few *reductions* of the configurations of Buyer1 and Seller in our calculus. First, the following configurations are initial:

$$\begin{aligned}&\mathtt{B1} \mapsto (\Phi_{\mathtt{B1}}, \Psi_{\mathtt{B1}}, \mathtt{"foo"} \rightarrowtriangle \mathtt{S.title} \triangleright ((\mathtt{quote(title)} \rightarrowtriangle \mathtt{B1.x} \triangleright G_1)\,;G_2))\\&\mathtt{S} \mapsto (\Phi_{\mathtt{S}}, \Psi_{\mathtt{S}}, \mathbf{skip})\end{aligned}$$

Next, Buyer1 passes `"foo"` and a continuation to Seller:

$$\begin{aligned}&\mathtt{B1} \mapsto (\Phi_{\mathtt{B1}}, \Psi_{\mathtt{B1}}, \mathbf{skip})\\&\mathtt{S} \mapsto (\Phi_{\mathtt{S}}, \Psi_{\mathtt{S}}, \mathtt{title} := \mathtt{"foo"}\,;(\mathtt{quote(title)} \rightarrowtriangle \mathtt{B1.x} \triangleright G_1)\,;G_2)\end{aligned}$$

Global program $x := E$ defines the assignment of (the value of) E, to variable x. Thus, as part of the communication from Buyer1 to Seller, the assignment of the incoming message and the execution of the continuation are explicitly scheduled to be executed by Seller, using sequencing.
Next, Seller executes the assignment. Let $\Phi'_{\mathtt{S}} = \Phi_{\mathtt{S}}\{\mathtt{title} \mapsto \mathtt{"foo"}\}$:

$$\mathtt{B1} \mapsto (\Phi_{\mathtt{B1}}, \Psi_{\mathtt{B1}}, \mathbf{skip}) \qquad \mathtt{S} \mapsto (\Phi'_{\mathtt{S}}, \Psi_{\mathtt{S}}, (\mathtt{quote(title)} \rightarrowtriangle \mathtt{B1.x} \triangleright G_1)\,;G_2)$$

Next, Seller passes the value of `quote(title)` (i.e., this expression is eagerly evaluated at Seller instead of lazily at Buyer1) and a continuation to Buyer1. Let $\Phi'_{\mathtt{S}}[\![\mathtt{quote(title)}]\!] = \mathtt{123}$ (the evaluation of `quote(title)` in $\Phi'_{\mathtt{S}}$):

$$\mathtt{B1} \mapsto (\Phi_{\mathtt{B1}}, \Psi_{\mathtt{B1}}, \mathtt{x} := \mathtt{123}\,;G_1) \qquad \mathtt{S} \mapsto (\Phi'_{\mathtt{S}}, \Psi_{\mathtt{S}}, G_2)$$

Next, Buyer1 executes the assignment or, in parallel, Seller asynchronously executes G_2. And so on. □

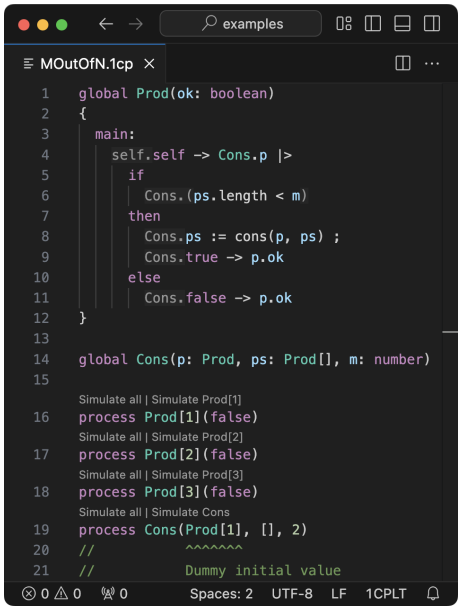
2.2 Implementation: 1CPLT by Example

About To explore how to possibly transfer the calculus from theory to practice, we initiated the *1CPLT* project. It consists of a prototype language, along with proof-of-concept tooling (including modern editor support), to apply first-person choreographic programming "for real". It is built as an extension to *VS Code*.[2]

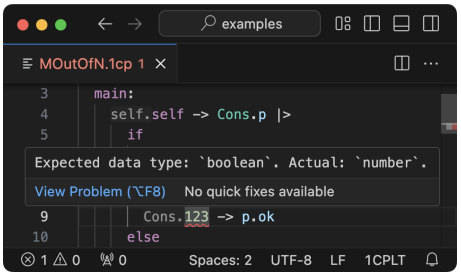
1CPLT includes a parser (implements the syntax of the calculus), a type checker (static semantics), and an interpreter (dynamic semantics). The examples below demonstrate 1CPLT implementations of basic distributed systems that are not supported by existing choreographic programming languages.

M-out-of-N First, we demonstrate the structure of the 1CPLT language.

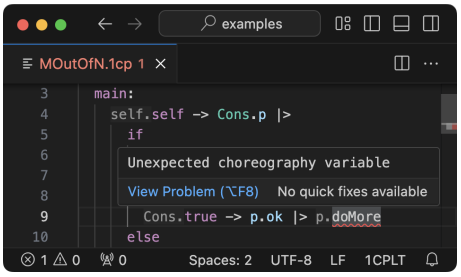
Example 10. An *M-out-of-N* distributed system consists of processes *Producer*1, ..., *Producern*, and *Consumer*. A self-reference is communicated from each producer to the consumer, unordered. The first m references received by the consumer, from any subset of producers, are stored; the last $n-m$ references are



(a) Well-typed



(b) Ill-typed: Unexpected type



(c) Ill-typed: Unexpected procedure

Fig. 2: M-out-of-N in 1CPLT

lost. The consumer tells each producer whether or not its reference was stored. There are two points-of-view: that of any producer and that of the consumer.

Fig. 2a shows a 1CPLT implementation of M-out-of-N.

Lines 1–14 contain two `global` blocks, each of which pertains to one point-of-view. In general, each `global` block defines:

- the name of a point-of-view (e.g., `Prod`);
- the variables of any process that has that point-of-view (e.g., `ok`);
- a distinguished choreographic procedure `main` whose body is the initial global program of any process that has that point-of-view (when `main` is absent, the initial global program is assumed to be **skip**);
- additional choreographic procedures (none in this example) that constitute the choreographic library of any process that has that point-of-view.

Lines 16–19 contain four `process` statements: each line defines an initial configuration of a process in terms of the name of its point-of-view (e.g., `Prod`), its index (e.g., `1`; when absent, it is assumed to be `0`), and its initial values of variables.

A key point is that **only `global` blocks are needed for static type checking, while `process` statements are needed only for dynamic execution.**

We note that the consumer also uses each received reference `p` to reply. □

[2] `https://github.com/microsoft/vscode`

Next, we demonstrate two forms of modern editor support offered by 1CPLT to improve the first-person choreographic programming experience.

Example 11. Fig. 2 shows that 1CPLT leverages two special features of VS Code:

- **Inlay hints** provide the programmer contextual information about the code, displayed in grey font wherever relevant. In 1CPLT, inlay hints inform the programmer about which process is active by prefixing code fragments with process references, *in the style of third-person choreographic programming*. For instance, "`self.`" on line 4 and "`Cons.`" on lines 6, 8–9, and 11, are inlay hints: they are *not* part of the code fragments but inserted by 1CPLT's inlay hint generator. Inlay hints are intended to improve code comprehension.
- **Code lenses** allow the programmer to trigger contextual actions, displayed as clickable text between lines of code (e.g., lines 16–19). In 1CPLT, "Simulate ..." uses the 1CPLT interpreter to run a simulation either of the full distributed system or of one particular process. □

Last, we demonstrate 1CPLT implementations with errors.

Example 12. Fig. 2b and Fig. 2c show examples of bugs caught by 1CPLT:

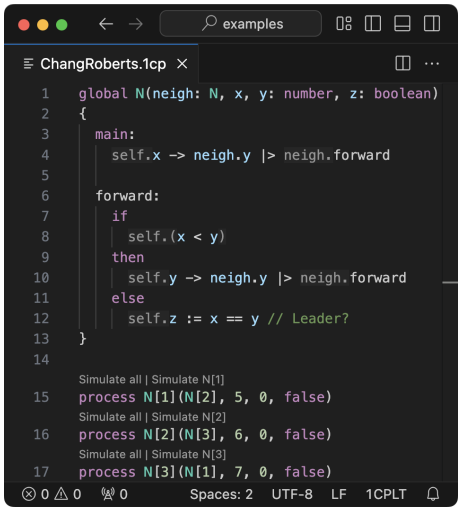
- In Fig. 2b, an unexpected message is passed from the consumer to a producer.
- In Fig. 2c, a continuation with an unexpected choreographic procedure call is passed from the consumer to a producer.

In both cases, the consumer burdens a producer with input that it cannot handle, which causes that producer to fail. The 1CPLT type checker, which implements the typing rules of the calculus, detects this. □

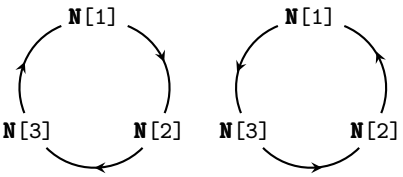
Even when m and n are fixed, most existing choreographic programming languages cannot be used to implement M-out-of-N. This is because these languages have limited support for non-determinism, which is crucial in M-out-of-N. That is, even when m and n are known statically, the order in which the consumer receives from the producers—*and how to reply*—is known only dynamically, which is problematic. In contrast, the event-driven nature of this paper's approach supports non-deterministic receives. Another notable exception is the recent work by Plyukhin et al. [43], which offers special support for out-of-order execution.

Chang–Roberts In the remainder of this section, we further demonstrate the expressiveness of 1CPLT (and, in turn, of the new calculus) by considering two classical *distributed algorithms*. First, we consider a distributed algorithm for *leader election*. This is the problem of computing a unique leader among the processes in a distributed system. The *Chang–Roberts* algorithm solves the problem for *non-anonymous* processes in a directed ring network [11].

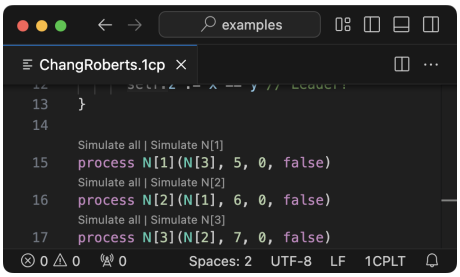
Example 13. A *Chang–Roberts* distributed system consists of processes *Node1*, ..., *Noden*. Each node has a unique numeric identifier. The nodes elect the node



(a) Clockwise



(b) Directions of the ring



(c) Anticlockwise

Fig. 3: Chang–Roberts

with the greatest identifier among them as the leader. There is only one point-of-view: that of any node.

Fig. 3a shows a 1CPLT implementation of Chang–Roberts. As there is only one point-of-view, there is only one `global` block. Each node has four variables: a reference to its neighbour (`neigh`), its own identifier (`x`), the greatest identifier it has received so far (`y`), and a flag to indicate whether or not it is elected (`z`).

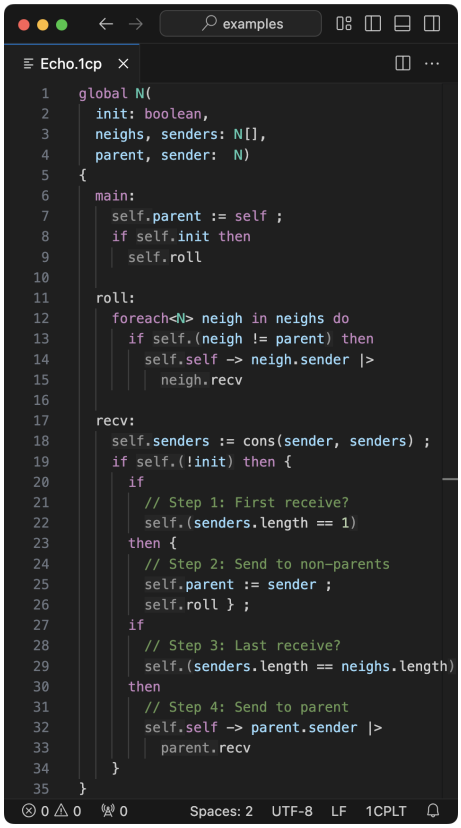
First, as defined by choreographic procedure `main`, the identifier of each node is passed to that node's neighbour (i.e., each node is initially active). Next, as defined by choreographic procedure `forward`, if the neighbour's own identifier is less than the incoming identifier, then the neighbour forwards the incoming identifier to the neighbour's neighbour. And so on. In this way, only the greatest identifier is forwarded by all nodes and makes a full pass along the ring. When a node receives its own identifier, it declares itself the leader. □

Next, we demonstrate that the network among the processes can be initialised at run-time, through the process references in their states.

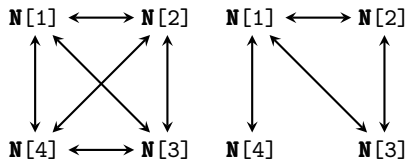
Example 14. Fig. 3a and Fig. 3c show, on lines 15–17, definitions of two alternative sets of initial configurations for three nodes: a clockwise-directed ring and an anticlockwise-directed ring. That is, statically, it is known *that* each node has a neighbour, but only dynamically, it is known *which one*. □

Even when the number of nodes and the neighbours of all nodes were fixed, *but not the identifiers*,[3] existing choreographic programming languages cannot

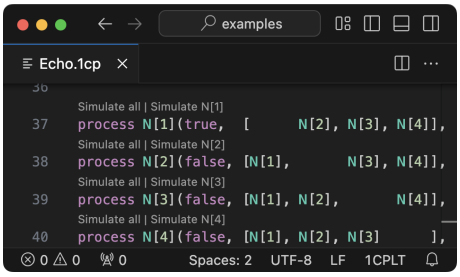
[3] If the identifiers were fixed, the process with the greatest identifier would be known already at compile-time, so there would be no need to elect a leader at run-time.



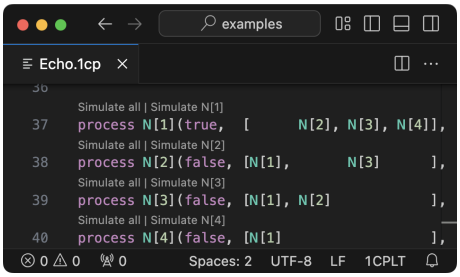
(a) Global program



(b) Two possible topologies



(c) Processes for left topology



(d) Processes for right topology

Fig. 4: Echo

be used to implement Chang–Roberts. This is because the total number of communications during a run (message complexity) depends on the assignment of identifiers to nodes (key property of Chang–Roberts). For instance, if the identifiers happen to be assigned in ascending order in the direction of the ring, then $2{\cdot}n - 1$ communications are needed. In contrast, if the identifiers happen to be assigned in descending order, then $^1/_2{\cdot}n^2 + ^1/_2{\cdot}n$ communications are needed. Existing choreographic programming language require at least n^2 communications.

Echo Next, we consider a distributed algorithm for *waves*. This is the problem of scattering/gathering information to/from all processes. The *Echo* algorithm solves the problem for undirected networks *of any topology* [49].

Example 15. An *Echo* distributed system consists of processes *Node1*, ..., *Noden*. One of the nodes acts as the *initiator* that triggers the wave. Each node except the initiator goes through the following steps:

1. Receive a message from any neighbour; it becomes the node's *parent*.
2. Send a message to each non-parent neighbour.
3. Receive a message back from each non-parent neighbour.
4. Send a message to the parent.

The initiator skips steps 1 and 4, while it sends/receives a message to/from all neighbours in steps 2 and 3. When step 3 of the initiator ends, it is guaranteed that each node has participated (and has been given the opportunity to scatter/gather information accordingly). Because the initiator and the non-initiators are very similar, we consider there to be only one point-of-view: that of any node. In general, there is some flexibility in determining the granularity at which points-of-view are identified for a given distributed system; this is a design decision.

Fig. 4a shows a 1CPLT implementation of Echo. Each node has five variables: a flag to indicate "initiator-hood" (`init`), a constant list of references to all neighbours (`neighs`), a list of references to all neighbours from which it has received (`senders`), a reference to the parent neighbour (`parent`), and a reference to the latest neighbour from which it has received (`sender`).

First, as defined by choreographic procedure `main`, each node checks if it is the initiator (i.e., each node is initially active, but only one of them triggers the wave). Next, as defined by choreographic procedure `roll`, the initiator sends a message and a continuation to each neighbour (i.e., `parent` has `self` as dummy initial value, which is not in `neighs`). Next, as defined by choreographic procedure `recv`, each neighbour of the initiator:

- updates `senders`;
- if it is the first receive, then it sets its parent and calls `roll` to send a message and a continuation to each neighbour (executed asynchronously by them);
- if it is the last receive, then it sends a message and a continuation back to its parent. □

Example 16. Fig. 4c and Fig. 4d show definitions of two alternative sets of initial configurations for four nodes, each of which has a different topology. The figure highlights that we need to write and statically analyse only one (collection of) global program(s) that can subsequently be initialised for any network. □

3 Design: Calculus of First-Person Choreographies

3.1 Organisation of the Section

Having introduced the main ingredients of the new calculus in Sect. 2.1, we present the details in this section. In three separate subsections, we define the syntax of the calculus (Sect. 3.2), its static semantics (Sect. 3.3), and its dynamic semantics (Sect. 3.4). Our presentation in these sections is layered: the first layer concerns "data to communicate", while on top of that, the second layer concerns "choreographies to communicate data". A final subsection states our main result (Sect. 3.5): communication deadlock freedom by design. All definitions, as well as the main result, are formalised in Isabelle/HOL; see also the artifact [35].

3.2 Syntax

Data We define the following syntax for data:

- Let $\mathbb{X} = \{\texttt{x}, \texttt{y}, \texttt{z}, \ldots\}$ denote the set of *variables*, ranged over by x, y, z.
- Let $\mathbb{P}$ denote the set of *point-of-view names*, ranged over by p, q, r.
- Let $\mathbb{T}$ denote the set of *data types*, ranged over by T:

  $$T ::= \texttt{Proc<}p\texttt{>} \mid \texttt{Bool} \mid \texttt{Nat} \mid \texttt{List<}T\texttt{>} \mid \cdots$$

  We note that many more data types may be available, but this is orthogonal to the contributions of this paper, so they are omitted here.
- Let $\mathbb{V}$ denote the set of *values*, ranged over by V:

  $$V_{\text{Nat}} ::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \cdots$$
  $$V ::= p\texttt{[}V_{\text{Nat}}\texttt{]} \mid \texttt{true} \mid \texttt{false} \mid V_{\text{Nat}} \mid \texttt{[}V_1\texttt{,}\ldots\texttt{,}V_n\texttt{]<}T\texttt{>} \mid \cdots$$

  Value $p\texttt{[}V_{\text{Nat}}\texttt{]}$—an "indexed point-of-view name"—defines a *process reference*. Value $\texttt{[}V_1\texttt{,}\ldots\texttt{,}V_n\texttt{]<}T\texttt{>}$ defines a *list* of length n with elements of data type T.
- Let $\mathbb{E}$ denote the set of *expressions*, ranged over by E:

  $$E ::= x \mid V \mid \texttt{!}E \mid E_1 \texttt{ \&\& } E_2 \mid E_1 \texttt{ || } E_2 \mid \cdots$$
- Let $\mathbb{X} \rightharpoonup \mathbb{V}$ denote the set of *process states* (partial functions from variables to values), ranged over by Φ.

Choreographies We define the following syntax for choreographies:

- Let $\mathbb{K}$ denote the set of *choreographic procedure names*, ranged over by k.
- Let $\mathbb{C} = \{\texttt{Chor<}p\texttt{>} \mid p \in \mathbb{P}\}$ denote the set of *choreographic types*.
- Let $\mathbb{G}$ denote the set of global programs, ranged over by G:

  $$G ::= k \mid \textbf{skip} \mid x := E \mid E_1 \twoheadrightarrow E_2.y \triangleright G \mid G_1 \,;\, G_2 \mid \textbf{if } E\ G_1\ G_2 \mid \textbf{while } E\ G$$

  Global program k defines a *call* of choreographic procedure k. Global program **skip** defines a *no-op*. Global program $x := E$ defines an *assignment* of (the value of) E, to variable x. Global program $E_1 \twoheadrightarrow E_2.y \triangleright G$ defines a *continuation-passing communication* of (the value of) E_1, to (the value of) E_2—"from an implicit active process to the explicit reactive process"—into variable y, followed by the asynchronous execution of G. Thus, E_1 is evaluated to the value to communicate, while E_2 is evaluated to (a reference to) the process to communicate with. Global programs $G_1\,;G_2$, **if** $E\ G_1\ G_2$, and **while** $E\ G$ define sequencing, conditional choice, and conditional iteration. In Exmp. 5, we also used the following syntactic sugar:

  $$\begin{aligned}\textbf{foreach}_T\ x \in E\ \textbf{do}\ G \quad \equiv \quad & \texttt{\_iter}_T := E\,; \\ & \textbf{while } \texttt{!isNil(\_iter}_T\texttt{)} \\ & \quad x := \texttt{headOrDefault(\_iter}_T\texttt{,}\ x\texttt{)}\,;\, G\,; \\ & \quad \texttt{\_iter}_T := \texttt{tailOrDefault(\_iter}_T\texttt{, [])}\end{aligned}$$

  Here, $\texttt{\_iter}_T$ is a variable that contains a list of data type T.
- Let $\mathbb{K} \rightharpoonup \mathbb{G}$ denote the set of choreographic *libraries* (partial functions from choreographic procedure names to global programs), ranged over by Ψ.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ [\vdash\text{-Var}] \qquad \frac{}{\Gamma \vdash p[V_{\text{Nat}}] : \texttt{Proc<}p\texttt{>}}\ [\vdash\text{-Ref}] \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}}\ [\vdash\text{-True}]$$

$$\frac{\Gamma \vdash E : \texttt{Bool}}{\Gamma \vdash \texttt{!}E : \texttt{Bool}}\ [\vdash\text{-Neg}] \qquad \cdots \qquad \frac{\emptyset \vdash \Phi(x) : \Gamma(x) \text{ for each } x \in \operatorname{dom}\Gamma}{\vdash \Phi : \Gamma}\ [\vdash\text{-State}]$$

Fig. 5: Static semantics of data (excerpt)

3.3 Static Semantics

Data Let $\mathbb{X} \rightharpoonup \mathbb{T}$ denote the set of all *data type environments* (partial functions from variables to data types), ranged over by Γ. Let

$$\Gamma \vdash E : T \quad \text{and} \quad \vdash \Phi : \Gamma$$

denote the *well-typedness* of E by T in Γ, and Φ by Γ. It is the relation induced by the rules in Fig. 5:

- Rules [⊢-Var], [⊢-True], and [⊢-Neg] are standard [42].
- Rule [⊢-Ref] states that an indexed point-of-view name $p[V]$ (process reference) is well-typed by $\texttt{Proc<}p\texttt{>}$. Shortly, this judgment will allow us to make assumptions about the presence/absence of variables and choreographic procedures in the configurations of any process $p[V]$ that has point-of-view p.
- Rule [⊢-State] states that a process state is well-typed when it has a well-typed value for each variable in the data type environment.

Choreographies Let $\mathbb{K} \rightharpoonup \mathbb{C}$ denote the set of *choreographic type environments* (partial functions from choreographic procedure names to choreographic types), ranged over by Δ. If R is a set of point-of-view names, then let

$$\{\Gamma_r\}_{r\in R}, \{\Delta_r\}_{r\in R} \vdash G : \texttt{Chor<}p\texttt{>} \quad \text{and} \quad \{\Gamma_r\}_{r\in R}, \{\Delta_r\}_{r\in R} \vdash \Psi : \Delta$$

denote the *well-typedness* of G by $\texttt{Chor<}p\texttt{>}$, and Ψ by Δ, in *families* of environments $\{\Gamma_r\}_{r\in R}$ and $\{\Delta_r\}_{r\in R}$. It is the relation induced by the rules in Fig. 6:

- Rule [⊢-Call] states that a choreographic procedure call is well-typed when that procedure exists and is well-typed.
- Rule [⊢-NoOp] states that a no-op is well-typed.
- Rule [⊢-Asgn] states that an assignment is well-typed from point-of-view p when the variable and the expression are well-typed by the same type in the data type environment of p.
- Rule [⊢-Comm] states that a continuation-passing communication is well-typed from point-of-view p (input of the rule, algorithmically) when:
  - the type of the expression from point-of-view p (first premise), and the type of the variable from point-of-view q (third premise), are the same;
  - the receiver has point-of-view q (second premise);
  - the continuation is well-typed from point-of-view q (fourth premise).

Let $\mathbf{\Gamma} = \{\Gamma_r\}_{r \in R}$, let $\mathbf{\Delta} = \{\Delta_r\}_{r \in R}$, and let $p \in R$, s.t. $\mathbf{\Gamma}(p) = \Gamma_p$ and $\mathbf{\Delta}(p) = \Delta_p$.

$$\frac{\mathbf{\Delta}(p)(k) = \texttt{Chor<}p\texttt{>}}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash k : \texttt{Chor<}p\texttt{>}}\ [\vdash\text{-Call}] \qquad \frac{}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \mathbf{skip} : \texttt{Chor<}p\texttt{>}}\ [\vdash\text{-NoOp}]$$

$$\frac{\mathbf{\Gamma}(p)(x) = T \quad \mathbf{\Gamma}(p) \vdash E : T}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash x := E : \texttt{Chor<}p\texttt{>}}\ [\vdash\text{-Asgn}]$$

$$\frac{\mathbf{\Gamma}(p) \vdash E_1 : T \quad \mathbf{\Gamma}(p) \vdash E_2 : \texttt{Proc<}q\texttt{>} \quad \mathbf{\Gamma}(q)(y) = T \quad \mathbf{\Gamma}, \mathbf{\Delta} \vdash G : \texttt{Chor<}q\texttt{>}}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash E_1 \twoheadrightarrow E_2.y \triangleright G : \texttt{Chor<}p\texttt{>}}\ [\vdash\text{-Comm}]$$

$$\frac{\mathbf{\Gamma}, \mathbf{\Delta} \vdash G_1 : \texttt{Chor<}p\texttt{>} \quad \mathbf{\Gamma}, \mathbf{\Delta} \vdash G_2 : \texttt{Chor<}p\texttt{>}}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash G_1 \,;\, G_2 : \texttt{Chor<}p\texttt{>}}\ [\vdash\text{-Seq}]$$

$$\frac{\Gamma_p \vdash E : \texttt{Bool} \quad \mathbf{\Gamma}, \mathbf{\Delta} \vdash G_1 : \texttt{Chor<}p\texttt{>} \quad \mathbf{\Gamma}, \mathbf{\Delta} \vdash G_2 : \texttt{Chor<}p\texttt{>}}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \mathbf{if}\ E\ G_1\ G_2 : \texttt{Chor<}p\texttt{>}}\ [\vdash\text{-If}]$$

$$\frac{\Gamma_p \vdash E : \texttt{Bool} \quad \mathbf{\Gamma}, \mathbf{\Delta} \vdash G : \texttt{Chor<}p\texttt{>}}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \mathbf{while}\ E\ G : \texttt{Chor<}p\texttt{>}}\ [\vdash\text{-While}]$$

$$\frac{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \Psi(k) : \Delta(k) \text{ for each } k \in \operatorname{dom} \Delta}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \Psi : \Delta}\ [\vdash\text{-Lib}]$$

Fig. 6: Static semantics of choreographies

$$\Phi[\![x]\!] = \Phi(x) \qquad \Phi[\![V]\!] = V \qquad \Phi[\![\texttt{!}E]\!] = \begin{cases} \texttt{true} & \text{if } \Phi[\![E]\!] = \texttt{false} \\ \texttt{false} & \text{otherwise} \end{cases} \qquad \ldots$$

Fig. 7: Dynamic semantics of data (excerpt)

This rule is the reason why *families* of data/choreographic type environments are needed in the judgments instead of individual environments: as we "cascade" along continuation-passing communications, at each next level, different environments are needed than at the previous level.
- Rules [⊢-Seq], [⊢-If], and [⊢-While] are straightforward.
- Rule [⊢-Lib] states that a choreographic library is well-typed when it has a well-typed global program for each name in the library.

We note that checking well-typedness is decidable and, in fact, linear in the size of the global program (because the typing rules are syntax-directed).

3.4 Dynamic Semantics

Data Let $\Phi[\![E]\!]$ denote the *evaluation* of E in Φ. It is the smallest function induced by the equations in Fig. 7. We note that the function is only partially defined. For instance, $\emptyset[\![\texttt{x}]\!]$ and $\Phi[\![\texttt{!123}]\!]$ are undefined. Undefinedness is potentially problematic: it causes processes to get stuck, which in turn causes com-

$$\frac{}{(\Phi, \Psi, \mathbf{skip}) \downarrow}\ [\downarrow\text{-NoOp}] \qquad \frac{(\Phi, \Psi, G_1) \downarrow \qquad (\Phi, \Psi, G_2) \downarrow}{(\Phi, \Psi, G_1 \,;\, G_2) \downarrow}\ [\downarrow\text{-Seq}]$$

$$\frac{\Psi(k) = G}{(\Phi, \Psi, k) \xrightarrow{\tau} (\Phi, \Psi, G)}\ [\rightarrow\text{-Call}] \qquad \frac{\Phi\{x \mapsto \Phi[\![E]\!]\} = \Phi'}{(\Phi, \Psi, x := E) \xrightarrow{\tau} (\Phi', \Psi, \mathbf{skip})}\ [\rightarrow\text{-Asgn}]$$

$$\frac{\Phi[\![E_1]\!] = V \qquad \Phi[\![E_2]\!] = q\texttt{[}j\texttt{]}}{(\Phi, \Psi, E_1 \twoheadrightarrow E_2.y \triangleright G) \xrightarrow{q\texttt{[}j\texttt{]}\,!\,(y\,:=\,V\,;\,G)} (\Phi, \Psi, \mathbf{skip})}\ [\rightarrow\text{-Send}]$$

$$\frac{}{(\Phi, \Psi, G_1) \xrightarrow{p\texttt{[}i\texttt{]}?G_2} (\Phi, \Psi, G_1 \,;\, G_2)}\ [\rightarrow\text{-Recv}]$$

$$\frac{(\Phi, \Psi, G_1) \xrightarrow{A} (\Phi', \Psi, G_1') \qquad A \neq p\texttt{[}i\texttt{]}?G \text{ for each } p\texttt{[}i\texttt{]}, G}{(\Phi, \Psi, G_1 \,;\, G_2) \xrightarrow{A} (\Phi', \Psi, G_1' \,;\, G_2)}\ [\rightarrow\text{-Seq1}]$$

$$\frac{(\Phi, \Psi, G_1) \downarrow \qquad (\Phi, \Psi, G_2) \xrightarrow{A} (\Phi', \Psi, G_2') \qquad A \neq p\texttt{[}i\texttt{]}?G \text{ for each } p\texttt{[}i\texttt{]}, G}{(\Phi, \Psi, G_1 \,;\, G_2) \xrightarrow{A} (\Phi', \Psi, G_2')}\ [\rightarrow\text{-Seq2}]$$

$$\frac{\Phi[\![E]\!] = \texttt{true}}{(\Phi, \Psi, \mathbf{if}\ E\ G_1\ G_2) \xrightarrow{\tau} (\Phi, \Psi, G_1)}\ [\rightarrow\text{-If1}] \qquad \frac{\Phi[\![E]\!] = \texttt{false}}{(\Phi, \Psi, \mathbf{if}\ E\ G_1\ G_2) \xrightarrow{\tau} (\Phi, \Psi, G_2)}\ [\rightarrow\text{-If2}]$$

$$\frac{\Phi[\![E]\!] = \texttt{true}}{(\Phi, \Psi, \mathbf{while}\ E\ G) \xrightarrow{\tau} (\Phi, \Psi, G \,;\, \mathbf{while}\ E\ G)}\ [\rightarrow\text{-While1}] \qquad \frac{\Phi[\![E]\!] = \texttt{false}}{(\Phi, \Psi, \mathbf{while}\ E\ G) \xrightarrow{\tau} (\Phi, \Psi, \mathbf{skip})}\ [\rightarrow\text{-While2}]$$

Fig. 8: Dynamic semantics of choreographies (part 1)

munication deadlocks among processes. However, using the static semantics of the previous section, undefinedness at run-time is detected at compile-time.

Choreographies Let $\mathbb{A}$ denote the set of *actions* (labels), ranged over by A:

$$A ::= q\texttt{[}j\texttt{]}\,!G \mid p\texttt{[}i\texttt{]}?G \mid \tau$$

Actions $q\texttt{[}j\texttt{]}\,!G$ and $p\texttt{[}i\texttt{]}?G$ define a *send* and a *receive* of global program G from process $p\texttt{[}i\texttt{]}$ to process $q\texttt{[}j\texttt{]}$. In this paper, G is of the form $y := V \,;\, G'$, where V and G' are a message and a continuation. Action τ defines an internal action.

Let $(\Phi, \Psi, G) \downarrow$ and $(\Phi, \Psi, G) \xrightarrow{A} (\Phi', \Psi', G')$ denote the *termination* of configuration (Φ, Ψ, G) and the *reduction* of source configuration (Φ, Ψ, G) to target configuration (Φ', Ψ', G') through action A. They are the relations induced by the rules in Fig. 8:

- Rule [↓-NoOp] states that a no-op can terminate.
- Rule [↓-Seq] states that sequencing can terminate when the operands can.
- Rule [→-Call] states that a call can reduce to the body of the corresponding choreographic procedure through an internal action.

- Rule [→-Asgn] states that an assignment can reduce to a no-op through an internal action, while the state in the configuration is updated accordingly.
- Rule [→-Send] states that a continuation-passing communication can reduce to a no-op through a send. The outgoing global program contains the message and the continuation. In this way, continuation-passing is a form of *process-driven* dynamic projection: the sender at run-time tells the receiver how to behave (from its first-person point-of-view), instead of a choreographic programming tool at compile-time (from its third-person point-of-view).
- Rule [→-Recv] states that *any* global program can reduce through a receive. The first operand of the resulting sequence is the original global program; the second operand is the incoming global program, which contains a message and a continuation. In this way, the incoming global program is explicitly scheduled to be run, but only when all existing work is done. Thus, scheduling is non-preemptive, reminiscent of event-driven programming.
  We note that there is a difference between: **(1)** the "physical receive" that takes the message/continuation off-network (happens when rule [→-Recv] is applied, and G_2 is appended to G_1); **(2)** the "logical receive" that actually processes the message/continuation (happens when the reduction of G_1 is finished, and the reduction of G_2 is started). Because of this decoupling, no explicit receive operations need to be defined in global programs: they happen implicitly, at any time, induced by the dynamic semantics (in practice, the run-time system takes care of this). This is reminiscent of: **(1)** buffering a message in a local event queue; **(2)** taking that message out of the event queue in a later iteration of the event loop to process it.
- Rules [→-Seq1] and [→-Seq2] are standard in the dynamic semantics of imperative languages [1] and process algebra [22], except that receives are excluded. Thus, receives happen only at the top-level and cannot preempt.
- Rules [→-If1], [→-If2], [→-While1], and [→-While2] are standard in the dynamic semantics of imperative languages [1] and process algebra [22].

We note that choreographic procedures enhance expressiveness relative to conditional iteration: not every recursive call can be simulated with **while**. The difference is that each iteration of **while** must start at the same process (the active process in the loop's enclosing scope), while each recursive call may start at a different process (the active process in the call site's enclosing scope).

If Π is a set of indexed point-of-view names, ranged over by π, then let

$$\{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi\in\Pi} \downarrow \quad \text{and} \quad \{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi\in\Pi} \rightarrow \{(\Phi'_\pi, \Psi'_\pi, G'_\pi)\}_{\pi\in\Pi}$$

denote the *termination* and *reduction* of *families* of configurations. That is, each configuration models a process, while the family models the distributed system. Formally, $\downarrow$ and $\rightarrow$ are the relations induced by the rules in Fig. 9:

- Rule [↓-Sys] states that a family can terminate when each of its configurations can terminate. This rule is standard in the dynamic semantics of parallel composition in process algebra [22].

$$\frac{(\Phi_\pi, \Psi_\pi, G_\pi)\downarrow \text{ for each } \pi \in \Pi}{\{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi\in\Pi}\downarrow}\ [\downarrow\text{-Sys}]$$

$$\frac{(\Phi_{p[i]}, \Psi_{p[i]}, G_{p[i]}) \xrightarrow{\tau} C'_{p[i]} \quad p[i] \in \Pi}{\{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi\in\Pi} \to \{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi\in\Pi\setminus\{p[i]\}} \cup \{C'_\pi\}_{\pi\in\{p[i]\}}}\ [\to\text{-Sys1}]$$

$$\frac{\begin{array}{c}(\Phi_{p[i]}, \Psi_{p[i]}, G_{p[i]}) \xrightarrow{q[j]!G} C'_{p[i]} \quad p[i] \in \Pi \\ (\Phi_{q[j]}, \Psi_{q[j]}, G_{q[j]}) \xrightarrow{p[i]?G} C'_{q[j]} \quad q[j] \in \Pi \setminus \{p[i]\}\end{array}}{\{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi\in\Pi} \to \{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi\in\Pi\setminus\{p[i],q[j]\}} \cup \{C'_\pi\}_{\pi\in\{p[i],q[j]\}}}\ [\to\text{-Sys2}]$$

$$\frac{(\Phi_{p[i]}, \Psi_{p[i]}, G_{p[i]}) \xrightarrow{p[i]!G} C'_{p[i]} \xrightarrow{p[i]?G} C''_{p[i]} \quad p[i] \in \Pi}{\{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi\in\Pi} \to \{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi\in\Pi\setminus\{p[i]\}} \cup \{C''_\pi\}_{\pi\in\{p[i]\}}}\ [\to\text{-Sys3}]$$

Fig. 9: Dynamic semantics of choreographies (part 2)

- Rule [→-Sys1] states that a family can reduce when one of its configurations can reduce through an internal action. This rule models independent evolution of processes in the absence of communication dependencies.
- Rule [→-Sys2] states that a family can reduce when one of its configurations can reduce through a send, while another configuration can reduce through a corresponding receive. This rule models communication between processes. We note that the send and the *physical* receive are synchronous (as usual in choreographic programming), but the *logical* receive is asynchronous.
- Rule [→-Sys3] states that a family can reduce when one of its configurations can reduce through a self-communication (first a send, next the receive, by one process). This allows a process to schedule work for itself for a later time. We note that the (self-)send and the physical (self-)receive are, like in rule [→-Sys2], effectively synchronous (i.e., the two reductions in the premise result in a single reduction in the conclusion), but they need to be ordered.

3.5 Main Result: Communication Deadlock Freedom by Design

Soundness The following theorem states our main result. To formulate it, let $\{(\Phi_{p[i]}, \Psi_p, G_p)\}_{p[i]\in\Pi}$ denote an initial family of configurations, where:

- Π denotes a set of indexed point-of-view names (process references);
- $\Phi_{p[i]}$ denotes an initial state for each $p[i] \in \Pi$;
- Ψ_p and G_p denote a choreographic library and an initial global program for each point-of-view $p \in \{p \mid p[i] \in \Pi\}$ (i.e., processes of the same point-of-view, have the same choreographic library and initial global program).

Now, the initial family of configurations either reduces finitely many times and terminates, or it reduces infinitely many times (i.e., it never gets stuck, so we enjoy communication deadlock freedom), when the following conditions are met:

1. Each choreographic library Ψ_p is well-typed. This is a static check.
2. Each initial global program G_p is well-typed. This is a static check.

3. Each initial state $\Phi_{p[i]}$ is well-typed. This is a lightweight dynamic check, part of initialisation (input validation).
4. The family of configurations is *closed*: if a process is referenced in a configuration, then it must have a configuration in the family. This is a lightweight dynamic check, part of initialisation (network validation). In the theorem below, let $\mathsf{refs}(\{(\Phi_{p[i]}, \Psi_p, G_p)\}_{p[i] \in \Pi})$ denote the set of process references that occur in any configuration of the family.

Intuitively, the static checks ensure that, if the initial configurations are proper, then a process never passes a continuation to another process that cannot run to completion: well-typedness guarantees that *expected* variables of *expected* types, as well as *expected* choreographic procedures, *actually* exist. Complementarily, the lightweight dynamic checks ensure that the initial configurations are proper.

Theorem 1. *Let* $R = \{p \mid p[i] \in \Pi\}$. *Assume:*

1. ***Statically:*** $\{\Gamma_r\}_{r \in R}, \{\Delta_r\}_{r \in R} \vdash \Psi_p : \Delta_p$ *for each* $p \in R$
2. ***Statically:*** $\{\Gamma_r\}_{r \in R}, \{\Delta_r\}_{r \in R} \vdash G_p :$ `Chor<`p`>` *for each* $p \in R$
3. ***Dynamically:*** $\vdash \Phi_{p[i]} : \Gamma_p$ *for each* $p[i] \in \Pi$
4. ***Dynamically:*** $\mathsf{refs}(\{(\Phi_{p[i]}, \Psi_p, G_p)\}_{p[i] \in \Pi}) \subseteq \Pi$

If $\{(\Phi_{p[i]}, \Psi_p, G_p)\}_{p[i] \in \Pi} \rightarrow^* C'$, *then either* $C' \downarrow$, *or* $C' \rightarrow C''$, *for some* C''.

We formalised all the definitions of this section in Isabelle/HOL and mechanised the proof of this theorem and all auxiliary lemmas. The following outline summarises the main steps and structure. See the Isabelle/HOL code for details. Extending this result to livelock is left for future work.

Proof (outline). The proof of Thm. 1 is split into two parts: *progress* and *preservation*. Progress states that if the four assumption hold for a family, then that family can either terminate or reduce. Preservation states that if the four assumptions hold for a source family, and if that source family reduces to a target family, then the four assumptions also hold for the target family. Thus, if the four assumptions hold for the initial family, then by applying progress and preservation inductively, we obtain the main result in Thm. 1. The proofs of progress and preservation use several forms of induction (on the structure of the terms; on the derivation of the judgments), nested inside each other through lemmas. □

Expressiveness Expressiveness of choreographic programming calculi can be measured along (at least) two relevant dimensions:

– Computability: Which classes of functional behaviour can be expressed?
– Message complexity: How many messages are needed and in which patterns?

Regarding computability, it has been shown that a simple core calculus of choreographies is Turing-complete (e.g., [16]). That calculus can be trivially embedded in ours. Regarding message complexity, none of the existing choreographic programming languages we know of supports all realisable *regular* message patterns

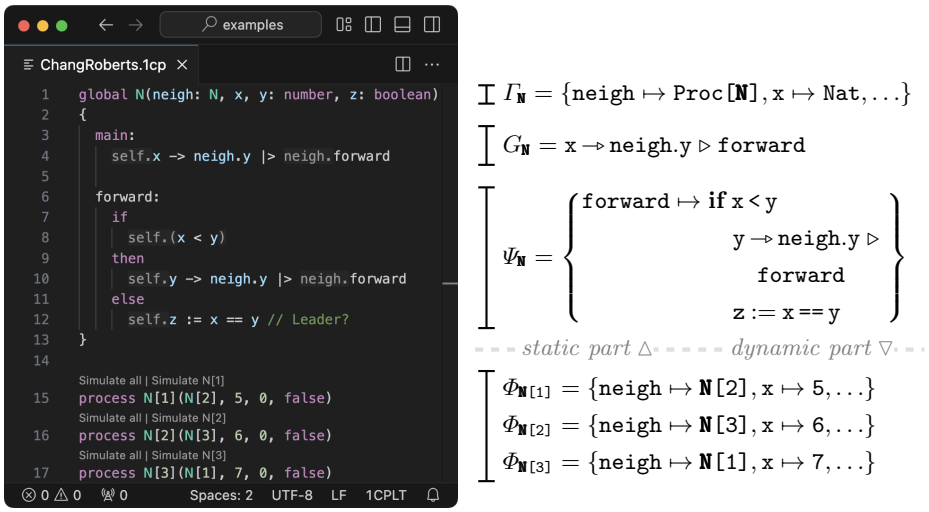


Fig. 10: Correspondence between 1CPLT and the calculus

(i.e., message patterns that can be specified with a regular expression over synchronous communications; e.g., M-out-of-N for any concrete m and n). In contrast, in addition to supporting parametricity, we conjecture that our approach also supports all realisable regular message patterns.

4 Implementation: 1CPLT

4.1 From Theory to Practice

The calculus presented in the previous section is expressive, but it(s notation) is unwieldy to use directly to implement distributed systems. With the development of 1CPLT, demonstrated in Sect. 2.2, we set out to explore how to possibly transfer the calculus from theory to practice, in a way that intuitively and concisely hides its complexities; see also the artifact [35].

Fig. 10 demonstrates the correspondence between the concrete syntax of 1CPLT and the ingredients of the calculus. Inspired by classes in object-oriented programming, the idea is to combine, for each point-of-view, the declarations of variables and the definitions of choreographic procedures in the same "unit" (`global` block), using a special choreographic procedure (`main`) to define the initial global program. This is all the information needed for static type checking. For simulation purposes, the creation of configurations is expressed separately (`process` statements) by defining the initial values of variables for each process.

4.2 Technical Details

General 1CPLT is implemented in *Rascal* [38], a meta-programming environment geared towards implementing programming languages. Among other fea-

tures, the *Rascal language* has core support to write context-free grammars (for defining concrete syntax), algebraic data types (for defining abstract syntax), and advanced pattern matching on grammar rules and ADT constructors. Together with standard programming abstractions, these features aim to simplify the implementation of parsers, type checkers, interpreters, and code generators.

Parsing The parser consumes a `.1cp` file as input and produces as output:

- for each point-of-view: abstract syntax trees for a data type environment, a choreographic type environment, and a choreographic library (`global` block);
- for each process: abstract syntax trees for an initial state (`process` stmnt).

The parser proceeds in two steps:

1. First, it transforms the input file into a concrete syntax tree using the *1CPLT grammar*. For instance, the following *production rules* in the 1CPLT grammar define the concrete syntax of assignments and communications (simplified to save space; see the artifact for the full version):

```
1 syntax ChorExpression
2   = ...
3   | Identifier     ":=" DataExpression
4   | DataExpression "->" DataExpression "." Identifier "|>" ChorExpression
5   | ... ;
```

2. Next, the parser transforms the concrete syntax tree into an abstract version using algebraic data types. For instance, the following *constructors* define the abstract syntax of assignments and continuation-passing communications (simplified to save space; see the artifact for the full version):

```
1 data CHOR_EXPRESSION(loc src = |unknown:///|)
2   = ...
3   | asgn(str x, DATA_EXPRESSION e)
4   | comm(DATA_EXPRESSION e, DATA_EXPRESSION qj, str y, CHOR_EXPRESSION c)
5   | ... ;
```

   We note that each constructor has an optional argument `src` (declared on line 1), which associates each value of a constructor with a particular location in the input file (`loc` is a core data type in Rascal for locations). This is essential for the type checker to generate properly located error messages.

Type checking The type checker consumes abstract syntax trees as input and produces a list of error messages as output; the list is empty if, and only if, the input is well-typed. Essentially, the type checker implements the typing rules in the static semantics of the calculus. For instance, the following code type-checks `asgn` constructors (simplified to save space; see the artifact for the full version):

```
1 data CHOR_CONTEXT = context(
2   map[str /* pov */, map[str /* data variables */, DATA_TYPE]] gammas,
3   map[str /* pov */, map[str /* chor variables */, CHOR_TYPE]] deltas) ;
4
5 list[Message] check(chor(p), CHOR_CONTEXT c, tree: asgn(x, e))
6   = [error("Unexpected data variable", tree.src)     | x notin c.gammas[p]]
7   + [*check(c.gammas[p][x], context(c.gammas[p]), e) | x    in c.gammas[p]] ;
```

For reference and comparison, typing rule [⊢-Asgn] (Fig. 6) is reproduced below.

$$\frac{\mathbf{\Gamma}(p)(x) = T \quad \mathbf{\Gamma}(p) \vdash E : T}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash x := E : \texttt{Chor<}p\texttt{>}}$$

Lines 1–3 define a general ADT for choreographic type environments. Line 5 declares the arguments to type-check `asgn` constructors, each of which corresponds with a variable in the conclusion of the typing rule: `p` with p, `c` with $\mathbf{\Gamma}$ and $\mathbf{\Delta}$, `x` with x, and `e` with e; `tree` is just a shorthand for `asgn(x, e)`.

Line 6 checks if `x` exists in the data type environment of `p`; if not, an error message is generated. The second argument of constructor `error` is the location in the input file where the error message should be displayed. Line 7 recursively calls function `check` to collect all error messages that result from type-checking data expression `e` (E) against data type `c.gammas[p][x]` ($\mathbf{\Gamma}(p)(x)$) in data environment `c.gammas[p]` ($\mathbf{\Gamma}(p)$). This is expressed using Rascal's notation for list comprehension: `[x | P(x), Q(x), ...]` constructs a list such that each `x` satisfies the conjunction of `P(x)`, `Q(x)`, ... Operator `*` splices a list into another list.

The functions to type-check the other constructors are defined similarly.

Interpretation The interpreter consumes abstract syntax trees as input (initial states and choreographic libraries) and produces a simulation as output. Essentially, the interpreter implements the reduction rules of the dynamic semantics of the calculus. For instance, the following code reduces `asgn` constructors (simplified to save space; see the artifact for the full version):

```
1 tuple[CHOR_STATE, CHOR_EXPRESSION] reduce(<state(phi, psi, {}), asgn(x, e)>)
2   = <state(phi + (x: eval(phi, e)), psi, {}), skip()> ;
```

Line 1 declares that function `reduce` consumes a source configuration as input (i.e., the arguments are directly destructured using pattern matching). Line 2 defines that function `reduce` produces a target configuration as output by updating the value of `x` in `phi`, in accordance with rule [→-Asgn] (Fig. 8).

The functions to reduce the other constructors are defined similarly.

Before each simulation, it is also checked whether or not each initial process state is well-typed, as well as closedness (points 3–4 of Thm. 1).

Code generation (*outlook*) We aim to extend 1CPLT with a code generator to be able to deploy 1CPLT implementations on real networks.

The most interesting aspect of such a code generator is that it should "compile away" continuation-passing: while continuation-passing is instrumental in our approach *as a programming abstraction*, physically passing around code-to-be-executed tends to be problematic in terms of security, privacy, and performance. Instead, we envisage a static transformation in which: **(a)** for each continuation, a choreographic procedure is introduced with a random label; **(b)** instead of passing the continuation, only the random label is passed.

5 Conclusion

5.1 Related Work

Process-parametric MPST Close to the work in this paper, in terms of starting point and aim, are several extensions of *multiparty session typing* (MPST)

[30] to support process-parametricity. First, Deniélou et al. [51] designed a dependently typed calculus to support process-parametricity, but the resulting typing relation is undecidable and was never implemented. Next, taking the opposite approach, Ng and Yoshida [41] implemented an extension of the *Scribble* tools for MPST [31, 32], called *Pabble*, to support process-parametricity, but it was never formalised and does not provide type-soundness guarantees. Last, Castro et al. [8] presented a calculus (all static analysis decidable) and a tool (formalised by the calculus) to support process-parametricity, but the expressiveness is limited (many distributed systems remain unsupported, including those in Sect. 2.2), while the static analysis algorithm is of exponential complexity. Thus, there is tension between decidability, soundness, and expressiveness.[4] However, by developing process-driven dynamic projection through continuation-passing communications (which diverges considerably from classical static MPST), our approach in this paper manages to combine these properties.

Dynamic projection Shen et al. [50] developed a choreographic programming library in Haskell, called *HasChor*, that uses dynamic projection to be able to build their library on top of the *freer monad* [37]. In this way, dynamic projection is primarily an implementation mechanism in HasChor. HasChor does not support process-parametricity, and it has not been formalised yet.

In MPST, dynamic projection has been studied by Hamers et al. [27, 34] in the form of a run-time verification framework based on global types, called *Discourje*. Discourje also supports process-parametricity, but it imposes run-time overhead, as all analysis happens dynamically. In contrast, the analysis in this paper happens almost completely statically.

Behavioural-typed actors/mailboxes The event-driven nature of our approach to choreographic programming is reminiscent of programming methods based on event loops/queues and asynchronous execution of event handlers. Closest in this area is the work on behavioural typing of actors and mailboxes by Fowler et al. [23, 24], De'Liguoro and Padovani [21], and Scalas et al. [48].

More fundamentally, we speculate that a deeper connection may be uncovered between actors/mailboxes and the approach in this paper. Our initial observation is that a mailbox-based runtime for 1CPLT (in combination with code generation; end of Sect. 4.2) would align perfectly with the dynamic semantics of our calculus—much more so than a (point-to-point) channel-based runtime would. This leads us to hypothesise that mailbox-based actors are to first-person choreographies what (point-to-point) channel-based processes are to third-person choreographies. That is, the following table would be completed:

[4] While we are not aware of any formal results about inherent limitations of static projection, we strongly suspect such limitations exist (because, fundamentally, fewer information is available at compile-time than at run-time).

global model of concurrency **(at development-time)**	(third-person) choreographies	??? $\Longleftarrow$ 1CP
local model of concurrency **(at deployment-time)**	processes + channels	actors + mailboxes
behavioural typing discipline	session types	mailbox types

Proof mechanisation In recent years, mechanisation of choreographic programming theory has received considerable attention, including the work of Cruz-Felipe et al. [15, 17–20], Hirsch and Garg [29], and Pahjola et al. [44]. Similar initiatives for behavioural typing include the work of Castro et al. [9,10], Hinrichson et al. [28], and Jacobs et al. [33]. Thus, the mechanisation of our proofs in Isabelle/HOL follows this trend in our community.

5.2 Future Work

On the theoretical side, we identify two directions for future work. First, we are keen to investigate the integration of dependent typing into our approach, to support arbitrary expressions as indices of point-of-view names (process references) instead of only values. Second, it will be interesting to develop dedicated testing and verification techniques for first-person choreographies, following recent work for third-person choreographies (e.g., [12, 13, 36]).

Another interesting question is whether a first-person approach could also be applied in MPST. Most MPST theories have a tight correspondence between the syntax of global/local types and the syntax of processes. If this correspondence is kept tight (i.e., a first-person approach is used both in global/local types and in processes), then we expect it to be possible—but still non-trivial—to develop typing rules accordingly and prove soundness. These typing rules would not rely on projection, but verify processes directly against global types. An alternative (presumably more challenging, but arguably more usable) would be to develop an MPST theory in which the correspondence is loosened: combining a first-person approach in global types with a more conventional style for processes.

On the practical side, in combination with an extension of 1CPLT with a code generator (Sect. 4.2), we are eager to conduct a case study on *Paxos*. This is a popular (family of) distributed algorithm(s) for *consensus*. Paxos has a number of properties (particularly related to non-determinism) that are fundamentally hard to support with third-person choreographies. We believe first-person choreographies can address these issues.

Data Availability Statement

The artifact is available on Zenodo [35]. It contains: (1) the calculus in Isabelle/HOL; (2) the language and tooling in VS Code.

References

1. Apt, K.R., de Boer, F.S., Olderog, E.: Verification of Sequential and Concurrent Programs. Texts in Computer Science, Springer (2009)
2. Carbone, M., Cruz-Filipe, L., Montesi, F., Murawska, A.: Multiparty classical choreographies. In: LOPSTR. Lecture Notes in Computer Science, vol. 11408, pp. 59–76. Springer (2018)
3. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: ESOP. Lecture Notes in Computer Science, vol. 4421, pp. 2–17. Springer (2007)
4. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. **34**(2), 8:1–8:78 (2012)
5. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: POPL. pp. 263–274. ACM (2013)
6. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. In: CONCUR. Lecture Notes in Computer Science, vol. 8704, pp. 47–62. Springer (2014)
7. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. Distributed Comput. **31**(1), 51–67 (2018)
8. Castro, D., Hu, R., Jongmans, S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. PACMPL **3**(POPL), 29:1–29:30 (2019)
9. Castro-Perez, D., Ferreira, F., Gheri, L., Yoshida, N.: Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In: PLDI. pp. 237–251. ACM (2021)
10. Castro-Perez, D., Ferreira, F., Yoshida, N.: EMTST: engineering the meta-theory of session types. In: TACAS (2). Lecture Notes in Computer Science, vol. 12079, pp. 278–285. Springer (2020)
11. Chang, E.J.H., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. Commun. ACM **22**(5), 281–283 (1979)
12. Coto, A., Guanciale, R., Tuosto, E.: An abstract framework for choreographic testing. J. Log. Algebraic Methods Program. **123**, 100712 (2021)
13. Cruz-Filipe, L., Graversen, E., Montesi, F., Peressotti, M.: Reasoning about choreographic programs. In: COORDINATION. Lecture Notes in Computer Science, vol. 13908, pp. 144–162. Springer (2023)
14. Cruz-Filipe, L., Larsen, K.S., Montesi, F.: The paths to choreography extraction. In: FoSSaCS. Lecture Notes in Computer Science, vol. 10203, pp. 424–440 (2017)
15. Cruz-Filipe, L., Lugovic, L., Montesi, F.: Certified compilation of choreographies with hacc. In: FORTE. Lecture Notes in Computer Science, vol. 13910, pp. 29–36. Springer (2023)
16. Cruz-Filipe, L., Montesi, F.: A core model for choreographic programming. Theor. Comput. Sci. **802**, 38–66 (2020)
17. Cruz-Filipe, L., Montesi, F.: Now it compiles! certified automatic repair of uncompilable protocols. In: ITP. LIPIcs, vol. 268, pp. 11:1–11:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023)
18. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Certifying choreography compilation. In: ICTAC. Lecture Notes in Computer Science, vol. 12819, pp. 115–133. Springer (2021)

19. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Formalising a turing-complete choreographic language in coq. In: ITP. LIPIcs, vol. 193, pp. 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
20. Cruz-Filipe, L., Montesi, F., Peressotti, M.: A formal theory of choreographic programming. J. Autom. Reason. **67**(2), 21 (2023)
21. de'Liguoro, U., Padovani, L.: Mailbox types for unordered interactions. In: ECOOP. LIPIcs, vol. 109, pp. 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
22. Fokkink, W.: Introduction to Process Algebra. Texts in Theoretical Computer Science. An EATCS Series, Springer (2000)
23. Fowler, S., Attard, D.P., Sowul, F., Gay, S.J., Trinder, P.: Special delivery: Programming with mailbox types. Proc. ACM Program. Lang. **7**(ICFP), 78–107 (2023)
24. Fowler, S., Lindley, S., Wadler, P.: Mixing metaphors: Actors as channels and channels as actors. In: ECOOP. LIPIcs, vol. 74, pp. 11:1–11:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017)
25. Giallorenzo, S., Montesi, F., Peressotti, M.: Choral: Object-oriented choreographic programming. ACM Trans. Program. Lang. Syst. **46**(1), 1:1–1:59 (2024)
26. Giallorenzo, S., Montesi, F., Peressotti, M., Richter, D., Salvaneschi, G., Weisenburger, P.: Multiparty languages: The choreographic and multitier cases (pearl). In: ECOOP. LIPIcs, vol. 194, pp. 22:1–22:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
27. Hamers, R., Jongmans, S.: Discourje: Runtime verification of communication protocols in clojure. In: TACAS (1). Lecture Notes in Computer Science, vol. 12078, pp. 266–284. Springer (2020)
28. Hinrichsen, J.K., Bengtson, J., Krebbers, R.: Actris: session-type based reasoning in separation logic. Proc. ACM Program. Lang. **4**(POPL), 6:1–6:30 (2020)
29. Hirsch, A.K., Garg, D.: Pirouette: higher-order typed functional choreographies. Proc. ACM Program. Lang. **6**(POPL), 1–27 (2022)
30. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284. ACM (2008)
31. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: FASE. Lecture Notes in Computer Science, vol. 9633, pp. 401–418. Springer (2016)
32. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: FASE. Lecture Notes in Computer Science, vol. 10202, pp. 116–133. Springer (2017)
33. Jacobs, J., Balzer, S., Krebbers, R.: Multiparty GV: functional multiparty session types with certified deadlock freedom. Proc. ACM Program. Lang. **6**(ICFP), 466–495 (2022)
34. Jongmans, S.: Discourje: Run-time verification of communication protocols in clojure - live at last. In: FM (2). Lecture Notes in Computer Science, vol. 14934, pp. 158–166. Springer (2024)
35. Jongmans, S.: First-person choreographic programming with continuation-passing communications (artifact) (2025), `https://doi.org/10.5281/zenodo.14624355`
36. Jongmans, S., van den Bos, P.: A predicate transformer for choreographies - computing preconditions in choreographic programming. In: ESOP. Lecture Notes in Computer Science, vol. 13240, pp. 520–547. Springer (2022)
37. Kiselyov, O., Ishii, H.: Freer monads, more extensible effects. In: Haskell. pp. 94–105. ACM (2015)
38. Klint, P., van der Storm, T., Vinju, J.J.: EASY meta-programming with rascal. In: GTTSE. Lecture Notes in Computer Science, vol. 6491, pp. 222–289. Springer (2009)

39. Montesi, F.: Introduction to Choreographies. Cambridge University Press (2023)
40. Montesi, F., Yoshida, N.: Compositional choreographies. In: CONCUR. Lecture Notes in Computer Science, vol. 8052, pp. 425–439. Springer (2013)
41. Ng, N., Yoshida, N.: Pabble: parameterised scribble. Service Oriented Computing and Applications **9**(3-4), 269–284 (2015)
42. Pierce, B.C.: Types and programming languages. MIT Press (2002)
43. Plyukhin, D., Peressotti, M., Montesi, F.: Ozone: Fully out-of-order choreographies. In: ECOOP. LIPIcs, vol. 313, pp. 31:1–31:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024)
44. Pohjola, J.Å., Gómez-Londoño, A., Shaker, J., Norrish, M.: Kalas: A verified, end-to-end compiler for a choreographic language. In: ITP. LIPIcs, vol. 237, pp. 27:1–27:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
45. Preda, M.D., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J.: Dynamic choreographies - safe runtime updates of distributed applications. In: COORDINATION. Lecture Notes in Computer Science, vol. 9037, pp. 67–82. Springer (2015)
46. Preda, M.D., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J.: Dynamic choreographies: Theory and implementation. Log. Methods Comput. Sci. **13**(2) (2017)
47. Preda, M.D., Giallorenzo, S., Lanese, I., Mauro, J., Gabbrielli, M.: AIOCJ: A choreographic framework for safe adaptive distributed applications. In: SLE. Lecture Notes in Computer Science, vol. 8706, pp. 161–170. Springer (2014)
48. Scalas, A., Yoshida, N., Benussi, E.: Verifying message-passing programs with dependent behavioural types. In: PLDI. pp. 502–516. ACM (2019)
49. Segall, A.: Distributed network protocols. IEEE Trans. Inf. Theory **29**(1), 23–34 (1983)
50. Shen, G., Kashiwa, S., Kuper, L.: Haschor: Functional choreographic programming for all (functional pearl). Proc. ACM Program. Lang. **7**(ICFP), 541–565 (2023)
51. Yoshida, N., Deniélou, P., Bejleri, A., Hu, R.: Parameterised multiparty session types. In: FoSSaCS. Lecture Notes in Computer Science, vol. 6014, pp. 128–145. Springer (2010)