Choreographies First, Session Types Later: Decoupling Deadlock Freedom from Endpoint Projection DRAFT

PHILIPP SCHUSTER, University of Tübingen, Germany
DAVID RICHTER, Technische Universität Darmstadt, Germany
MARIUS MÜLLER, University of Tübingen, Germany
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany
MIRA MEZINI, Technische Universität Darmstadt, Germany

Distributed programs are traditionally implemented as separate processes that communicate via message passing, making it easy to introduce deadlocks. Session types address this by specifying communication protocols and statically verifying protocol adherence, but writing and maintaining global and local types by hand is laborious. Choreographic programming offers an alternative: a single choreography describes the behavior of all participants, from which endpoint processes are generated and proved deadlock-free via a simulation argument between the choreography and the process, tightly coupling the proof to a specific projection.

In this paper, we present a choreographic framework in which deadlock freedom is instead mediated by automatically inferred protocols. From a choreography, we infer a global type and project one local type per role; any pool of processes that implements these local types is guaranteed to be deadlock-free, regardless of how the processes were obtained. We instantiate this idea with a choreographic language, a process language, and a compiler performing endpoint projection from choreographies to processes. We prove progress and preservation for well-typed pools of processes, and show that projected processes implement their inferred local types. This treats global and local types as internal compiler artifacts that both justify endpoint projection and support post-hoc verification of independently developed or refactored implementations.

CCS Concepts: • **Computing methodologies → Concurrent programming languages**; • **Theory of computation** → *Type theory*.

Additional Key Words and Phrases: distributed programming, choreographic programming, session types

ACM Reference Format:
Philipp Schuster, David Richter, Marius Müller, Jonathan Immanuel Brachthäuser, and Mira Mezini. 2026. Choreographies First, Session Types Later: Decoupling Deadlock Freedom from Endpoint Projection DRAFT. 1, 1 (March 2026), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' Contact Information: Philipp Schuster, Department of Computer Science, University of Tübingen, Tübingen, Germany, philipp.schuster@uni-tuebingen.de; David Richter, Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany, david.richter@tu-darmstadt.de; Marius Müller, Department of Computer Science, University of Tübingen, Tübingen, Germany, mari.mueller@uni-tuebingen.de; Jonathan Immanuel Brachthäuser, Department of Computer Science, University of Tübingen, Tübingen, Germany, jonathan.brachthaeuser@uni-tuebingen.de; Mira Mezini, Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany, mezini@informatik.tu-darmstadt.de.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2026/3-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

A distributed system consists of multiple programs running at different locations and interacting through a network. Traditionally, developers write each such program separately and deploy it as a *process* that communicates via message passing. Developing such systems is challenging, as it is easy to introduce deadlocks, i.e., situations in which one or more processes are waiting indefinitely for each other.

To rule out deadlocks, the system as a whole must follow a common *protocol*. Multiparty session types [Honda et al. 2008] provide a formal way to specify such protocols and verify that process implementations conform to them. In this approach, the protocol is a global type that describes all expected communication and control flow. Individual processes are then implemented to follow their respective local projections, and a global check ensures that the implementations collectively adhere to the protocol. While this approach guarantees strong correctness properties, it can also make development cumbersome. Because both the global type and the processes are interdependent, changes to the design often require coordinated updates to both, making iterative development tedious.

An alternative approach is choreographic programming [Carbone et al. 2012], where the entire system – both computation and communication – is described within a single program, called a *choreography*. This unified description allows the developer to reason about interactions sequentially. A compiler then automatically generates the distributed processes from the choreography, ensuring that their communication structure follows the original specification. The classical proof strategy [Carbone et al. 2012] for deadlock freedom in this setting consists of two steps: first, proving that all well-formed choreographies are deadlock-free; and second, lifting this property to the generated processes via a simulation between the semantics of choreographies and processes. Deadlock freedom is thereby guaranteed for all generated programs once and for all by the compiler's correctness proof. Since the proof depends on the precise correspondence between the choreography and its generated code, even minimal modifications to the produced processes invalidate the guarantee.

In this paper, we combine the strengths of choreographic programming and multiparty session types. Figure 1 summarizes the relationships between choreographies, global types, processes, and local types, and contrasts our approach with previous ones. In choreographic programming (Figure 1a), programmers write a choreography from which processes are projected; deadlock freedom follows from a simulation proof between the choreography and its projection, tying the guarantee to the generated code. With multiparty session types (Figure 1c), programmers instead write a global type and the individual processes, and a separate type check ensures protocol conformance and deadlock freedom, at the cost of additional specification effort. Our approach (Figure 1b), combines these ideas: The programmer only writes a choreography that specifies the behavior of the system as a whole. From this choreography, we automatically infer a global type, project one local type per role, and generate one process per role for distributed execution via *endpoint projection*. The types both justify the correctness of the generated processes and can be reused to verify alternative implementations. Our key technical insight is that *the proof of deadlock freedom can be factored through the inferred session types, rather than being tied directly to a particular endpoint projection of the choreography*. We prove that whenever each process in a system implements its local type, the system as a whole is free from deadlocks, independently of how those processes were obtained. The processes generated by our compiler inherit deadlock freedom, because we show that they implement their respective local types.

This perspective treats global and local session types as compiler artifacts that mediate between choreographies and processes, rather than as separately written specifications. It retains the rapid

99 development style of choreographic programming – developers merely write choreographies –
100 while enabling the kind of modular post-hoc verification familiar from multiparty session types:
101 processes may be refactored, reimplemented, or replaced, as long as they continue to satisfy their
102 local types.

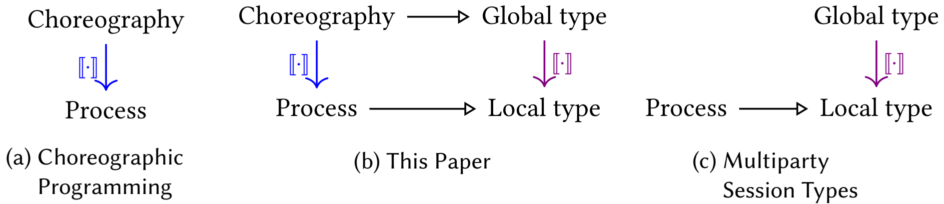


Fig. 1. Overview

Contributions.

- We introduce a choreographic language, **LangGlobal**, a process language **LangLocal**, and a compiler performing endpoint projection from choreographies to processes.
- Choreographies are typed with global types, and processes are typed with local types; we define a projection from global types to local types.
- We prove the standard theorems of progress (Theorem 3.1) and preservation (Theorem 3.2) for pools of processes, given that they implement their local types projected from a global type. Together, these theorems imply deadlock freedom (Corollary 3.3). The statement as well as the proofs of these properties straightforwardly follow the syntactic method.
- Finally, we prove that for any well-typed **LangGlobal** choreography, the **LangLocal** processes generated by endpoint projection are well-typed with respect to the local types projected from its inferred global type (Theorem 3.7 and Corollary 3.8). The projected pools are hence guaranteed to be deadlock-free instances of our general session-typed framework.

These results collectively support our view of global and local session types as internal compiler artifacts that both justify the correctness of endpoint projection and enable independent implementations to be verified post-hoc against their local types.

The remainder of the paper is organized as follows. In Section 2, we introduce our languages **LangGlobal** and **LangLocal** with motivating examples. In Section 3, we develop the formalization of our languages and the compiler between them, along with their properties. In Section 4, we discuss some design decisions of our system concerning synchronicity, which we further illustrate with case studies from the literature in Section 5. In Section 6, we discuss related work, before we conclude in Section 7.

2 Overview

In this section, we motivate choreographic programming and demonstrate how it simultanously enables sequential reasoning as well as distributed execution by means of simple examples. Generally, a programmer writes a single choreography that describes the joint behavior of a fixed number of roles. Our compiler then automatically generates one process for each role through endpoint projection. The programmer deploys these individual processes at different locations and we guarantee that they run without type errors and deadlocks.

Choreography	Process A	Process B	Process C	Process D
`// map`	*// map*	*// map*	*// map*	*// map*
`x1 @B ← x1 @A`	*B ← x1*	*x1 ← A*		
`x2 @C ← x2 @A`	*C ← x2*		*x2 ← A*	
`x3 @D ← x3 @A`	*D ← x3*			*x3 ← A*
`let y0 @A = x0*x0`	*let y0 = x0*x0*			
`let y1 @B = x1*x1`		*let y1 = x1*x1*		
`let y2 @C = x2*x2`			*let y2 = x2*x2*	
`let y3 @D = x3*x3`				*let y3 = x3*x3*
`// reduce`	*// reduce*	*// reduce*	*// reduce*	*// reduce*
`y1 @A ← y1 @B`	*y1 ← B*	*A ← y1*		
`y3 @C ← y3 @D`			*y3 ← D*	*C ← y3*
`let z0 @A = y0+y1`	*let z0 = y0+y1*			
`let z1 @C = y2+y3`			*let z1 = y2+y3*	
`z1 @A ← z1 @C`	*z1 ← C*		*A ← z1*	
`let s0 @A = z0+z1`	*let s0 = z0+z1*			

Fig. 2. Map-reduce choreography and processes.

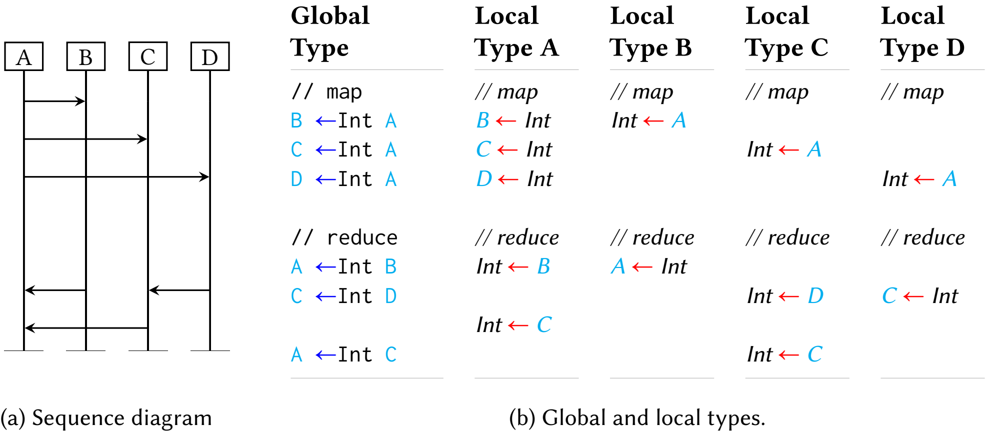
2.1 Map Reduce

As our first example, we consider the distributed computation of the sum of squares of components of a four-dimensional vector (x_1, x_2, x_3, x_4). It is written as a *choreography* in LangGlobal and listed on the left-hand side in Figure 2. This example involves four *roles* `A`, `B`, `C`, and `D`, where initially all four components of the vector are stored at role `A` in variables `x0`, `x1`, `x2`, `x3`. To track this fact, in choreographic programming variables are *located*, written `x @R` for variable `x` and role `R`.

In this example, we first distribute the three components `x1`, `x2`, and `x3` from role `A` to the other roles `B`, `C`, and `D`, using a communication statement such as `x1 @B ← x1 @A`. It communicates the value at role `A` bound to `x1` to role `B` and binds it to `x1` there. After distribution, we perform local computation with statements like `let y1 @B = x1 * x1`, which computes `x1 * x1` at role `B` and binds the result to `y1`. All variables involved in the computed expression must be available at role `B`, which is statically enforced. No implicit communication happens.

Finally, using statements for communication and computations, we perform a tree reduction and bind the resulting sum of squares to `s0` at `A`. Execution proceeds in parallel at the different roles, yet the result is deterministic and guaranteed to be the same as under sequential execution. By inspecting this program, we can read off the total amount of work, four multiplications and three additions. Moreover, we can see that the four multiplications and also two of the additions happen in parallel. In the future, it might be possible to automatically infer other performance properties, such as the span of the computation.

From this choreographic program, we automatically generate one *process* for each role. Figure 2 on the right-hand side shows the generated processes, which are written in LangLocal, a language of communicating processes. This language has statements for sending to a role, e.g., *B ← x1*, for receiving from a role, e.g., *y1 ← B*, and for computation, e.g., *let y0 = x0 * x0*. It assumes that all roles are connected to each other, and that sending and receiving between two roles is synchronous. Computation at different roles happens in parallel. Even in this simple example, the code generated for the different roles is very different, and manually ensuring freedom from deadlock would be a non-trivial task.



(a) Sequence diagram

(b) Global and local types.

Fig. 3. Map-reduce.

Additionally, from the choreography in Figure 2, we infer the *global type*, which describes the communication from a global point of view. The global type is presented on the left-hand side in Figure 3. For intuition, we also provide the map-reduce global type visualized as a sequence diagram (Figure 3a). Just as we project the choreography onto the different roles to obtain processes, we can project the global type onto the different roles to obtain *local types*. These local types are shown in Figure 3 on the right-hand side.

As long as individual processes conform to the local types, no deadlock can happen (Corollary 3.3). As it happens, the processes we generate from the choreography do conform to them (Corollary 3.8). From a software engineering perspective, it is conceivable that programmers replace individual processes by a different implementation or perform refactorings on individual processes. The above-mentioned guarantees hold as long as the updated processes still conform to their respective local types.

In this first example, we have seen how choreographies enable sequential reasoning for distributed programs, which is useful for expressing parallel distributed algorithms that terminate with a deterministic result. Next, we introduce control-flow constructs that make it possible to express loops.

2.2 Pipeline Stages

As a second example, let us consider a pipeline of functions `f`, `g`, and `h`, computed at three different roles `A`, `B`, and `C`. In order to express loops, **LangLocal** features definitions and jumps. Definitions must be on the top level, and jumps must be in tail position. The example in Figure 4 defines `loop`, which takes two parameters. Like all variables, parameters are located. In the body of the loop, we compute the results of all three functions in parallel, and communicate the results to the next stages, before finally performing the recursive jump.

After the definitions, in order to fill the pipeline, we have to perform some computation and communication. Our type system makes sure that we do not attempt to use a value at a different role without communicating it first. By inspecting this program, we can see that we have some sequential computation until the pipeline is filled, but once the loop has started, we proceed in parallel. Moreover, only the parameters have to be kept live across iterations.

From this choreography, we automatically generate the processes in Figure 4. Each of them defines a loop with different parameters. In our semantics, jumps are synchronous, and so all

processes have to wait until each of them finishes their iteration of the loop before continuing. See Section 4 for a discussion of the benefits and limitations of this model.

The generated code for filling the pipeline is non-trivial, yet we guarantee its safety by inferring the global type in Figure 5 from the choreography. The structure of the global type mirrors the structure of the choreography, i.e., for every choreography definition, a corresponding nominal global type is created. When the choreography features recursion (or even mutual recursion between multiple choreography definitions), so do the corresponding global types. Again, from the global type, we can project individual local types for each role (Figure 5). We guarantee safety as long as processes conform to these local types.

In this second example, we have seen how choreographies with definitions and jumps allow us to express non-terminating distributed programs, and how we guarantee safety of these by inferring a recursive global type, which we then project onto recursive local types. It is of course also desirable to express programs that behave differently depending on dynamic conditions, which we cover next.

Choreography	**Process A**	**Process B**	**Process C**
`def loop(x1 @B, y1 @C) {`	*def loop() {*	*def loop(x1) {*	*def loop(y1) {*
`  let x2 @A = f()`	*let x2 = f()*		
`  let y2 @B = g(x1)`		*let y2 = g(x1)*	
`  let z2 @C = h(y1)`			*let z2 = h(y1)*
`  x2 @B ← x2 @A`	*B ← x2*	*x2 ← A*	
`  y2 @C ← y2 @B`		*C ← y2*	*y2 ← B*
`  loop(x2, y2)`	*loop()*	*loop(x2)*	*loop(y2)*
`}`	*}*	*}*	*}*
`let x0 @A = f()`	*let x0 = f()*		
`x0 @B ← x0 @A`	*B ← x0*	*x0 ← A*	
`let y1 @B = g(x0)`		*let y1 = g(x0)*	
`y1 @C ← y1 @B`		*C ← y1*	*y1 ← B*
`let x1 @A = f()`	*let x1 = f()*		
`x1 @B ← x1 @A`	*B ← x1*	*x1 ← A*	
`loop(x1, y1)`	*loop()*	*loop(x1)*	*loop(y1)*

Fig. 4. Pipeline choreography and processes.

Global Type	**Local Type A**	**Local Type B**	**Local Type C**
`type loop {`	*type loop {*	*type loop {*	*type loop {*
`  B ←Int A`	*B ← Int*	*Int ← A*	
`  C ←Int B`		*C ← Int*	*Int ← B*
`  loop`	*loop*	*loop*	*loop*
`}`	*}*	*}*	*}*
`B ←Int A`	*B ← Int*	*Int ← A*	
`C ←Int B`		*C ← Int*	*Int ← B*
`B ←Int A`	*B ← Int*	*Int ← A*	
`loop`	*loop*	*loop*	*loop*

Fig. 5. Pipeline protocol and sessions.

Choreography

```
def loop(n @A) {
  if (n > 0 @A) {
    n @B ← n @A
    let r @B = f(n)
    let i @A = n - 1
    if (r < 0 @B) {
      loop(i)
    } else { end }
  } else { end }
}
```

Process A

```
def loop(n) {
 choose (n > 0) {
  B ← n

  let i = n − 1
  wait {
   loop(i)
  } else { done }
 } else { done }
}
```

Process B

```
def loop() {
 wait {
  n ← A
  let r = f(n)

  choose (r < 0) {
   loop()
  } else { done }
 } else { done }
}
```

Fig. 6. Producer–consumer choreography and processes.

Global Type

```
type loop {
  if (A) {
    B ←Int A
    if (B) {
      loop
    } { end }
  } { end }
}
```

Local Type A

```
type loop {
 ⊕ {
  B ← Int
  & {
   loop
  } { exit }
 } { exit }
}
```

Local Type B

```
type loop {
 & {
  Int ← A
  ⊕ {
   loop
  } { exit }
 } { exit }
}
```

Fig. 7. Producer–consumer global type and local types.

2.3 Producer Consumer

In our final example, we have a producer and a consumer of numbers, at roles A and B respectively. While role A decrements, role B applies some function `f`. We want to allow both of them to terminate the overall pipeline, based on the results of their local computation. We invite the reader to stop briefly and think about how to achieve this with, for example, communicating processes and channels. The program as a choreography in LangGlobal is presented in Figure 6.

To express this situation, our language of choreographies has a conditional statement. The condition is an expression that is located at a role. This role leads in which branch to take, and the other roles follow. In this example, role A decides to end when its variable `n` becomes zero, and role B decides to end if the result `r` of calling function `f` is smaller than zero. Indeed, the processes we project for the two roles take turns in waiting for the other process's choice, as is visible on the right-hand side in Figure 6.

Generally, in our semantics, all roles synchronize at these points of decision and wait for the single leader to choose. The global type we infer from this choreography, shown in Figure 7, reflects this fact by mentioning the leading role in the choice `if` (... *@r*) ... `else` .... Similarly, when we project this global type onto the roles to obtain local types, also presented in Figure 7, they contain these points of decision. At each such decision, only one of them, the leader, has an *internal choice* (denoted ⊕ *{...} {...}*), while all others follow this *external choice* (written & *{...} {...}*). We again refer to Section 4 for a discussion of this design.

This approach to safety seamlessly scales to many roles, by virtue of the global type that we infer from the choreography. All processes that conform to the projections of this global type

Expressions:

e	$::=$	$1 \mid 2 \mid \ldots \mid e + e \mid \ldots$	primitives
	$\mid$	x	variables

Contexts:

τ	$::=$	$\mathsf{Int} \mid \ldots$	base types
Γ	$::=$	$\bullet \mid \Gamma, x : \tau$	local contexts
Δ	$::=$	$\bullet \mid \Delta, x : \tau\ @r$	global contexts

Roles:

r	$::=$	$\mathsf{A} \mid \mathsf{B} \mid \ldots$

Fig. 8. Common syntax.

onto their roles, are free from deadlock. Again, the processes we obtain from the choreography by endpoint projection do so. In the next section, we make these claims formal by defining the respective languages, type systems, and translations.

3 Technical Development

In this section, we formally present our language of choreographies **LangGlobal** and the global types describing them, as well as our language of processes **LangLocal** and the local types describing them. We project choreographies onto one process for each role, and we project global types onto one local type for each role. We define an operational semantics for pools of processes, where processes can act independently, and which consequently is non-deterministic. We prove that a pool of processes can never deadlock if the processes in the pool behave in accordance with the local types projected from some global type. Moreover, we can straightforwardly infer a global type from a given choreography. Finally, we show that processes projected from a choreography in fact do comply with the local types projected from its global type. Together, these properties entail that a pool of processes projected from a choreography indeed does not deadlock.

The paper is accompanied by an intrinsically-typed mechanization of both languages and their operational semantics in Idris 2 [Brady 2020], which implies type safety (Theorem 3.1, Theorem 3.2) and consequently deadlock freedom (Corollary 3.3). This mechanization also includes the inference of the global type, and the endpoint projection of processes from choreographies. The intrinsically-typed nature implies typability preservation for the projection (Theorem 3.7 and Corollary 3.8).

3.1 Syntax

We start with the syntax of choreographies and global types, and that of processes and local types. Both, choreographies and processes make use of a language for local expressions whose syntax is presented in Figure 8. The expression language is very simple, only consisting of variables and primitives, and its types only consist of base types like integers, for illustration. The results we prove are independent of the concrete expression language, however, as long as its evaluation does not get stuck. Figure 8 also shows the two different kinds of typing contexts we use: local contexts Γ map variables to types as usual, while global contexts Δ additionally track the information for which role r, i.e., which process, this binding is valid. Roles are unique identifiers drawn from a finite set.

Choreographies and global types. The syntax for choreographies and global types is given in Figure 9. A program C is a list of top-level labels with a role-annotated parameter $x : \tau\ @r$, whose

393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409

Choreographies:

c	$::=$	$\texttt{let}\ x\ @r = e;\ c$	computation
	$\mid$	$l(e)$	jump
	$\mid$	$x\ @r \leftarrow e\ @r;\ c$	communication
	$\mid$	$\texttt{if}\ (e\ @r)\ c\ \texttt{else}\ c$	conditional
	$\mid$	$\texttt{end}\ x\ @r$	end
C	$::=$	$\texttt{def}\ l(x : \tau\ @r)\ \{\ c\ \}, \ldots$	global program

Global types:

θ	$::=$	l	jump
	$\mid$	$r \leftarrow\!\tau\ r;\ \theta$	communication
	$\mid$	$\texttt{if}\ (r)\ \theta\ \texttt{else}\ \theta$	conditional
	$\mid$	$\texttt{end}$	end
Θ	$::=$	$\texttt{type}\ l\ \{\ \theta\ \}, \ldots$	global types

Fig. 9. Syntax of LangGlobal.

410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441

bodies consist of a choreography c, and which can be jumped to with $l(e)$ by providing an argument e for the parameter. For ease of presentation, top-level labels have only one parameter, but it is straightforward to allow for multiple parameters, and our mechanization does so. A computation $\texttt{let}\ x\ @r = e;\ c$ binds the result of evaluating an expression e at role r to variable x. The binding can then be used in the remaining choreography c. A role r_1 can communicate the result of evaluating an expression e to another role r_2 with $x\ @r_2 \leftarrow e\ @r_1;\ c$. This result is then bound to x in the remaining choreography c. The conditional construct $\texttt{if}\ (e\ @r)\ c_0\ \texttt{else}\ c_1$ evaluates an expression e at a role r, executing the first branch c_0 if the result is 0, and the second branch c_1 otherwise. A program can be ended with $\texttt{end}\ x\ @r$, returning the value bound to variable x at role r.

Global types θ describe the behavior of choreographies, mirorring their control flow, but omitting any concrete terms. No expressions or variables are visible in global types, and consequently local computation is not represented at all. The global types for communication and conditionals, however, track which roles are involved, and the communication also tracks the type of the value communicated. This avoids miscommunication between endpoint processes.

Processes and local types. Figure 10 presents the syntax for processes and local types. In analogy to global programs, a local program P is a list of top-level labels whose bodies consist of a process p. They can have either one parameter $x : \tau$ – this time without an annotated role – or no parameter. For ease of presentation, we do not show the version without parameter here. The syntax of the constructs for computations and jumps to labels is essentially the same as for choreographies, but without any roles and with jumps having either zero or one argument. As we will see later, jumps are synchronized between all processes. The other constructs for choreographies each decompose into two different process constructs. A communication decomposes into a sending construct $r \leftarrow e;\ p$, and a receiving construct $x \leftarrow r;\ p$. The sending construct sends the result of evaluating an expression e to the process with role r, before continuing with the remaining process p. The receiving construct binds the value it receives from the process with role r to variable x in the remaining process p. Similarly, conditionals decompose into a choosing construct $\textit{choose}\ (e)\ p_0\ \textit{else}\ p_1$, and a waiting construct $\textit{wait}\ p_0\ \textit{else}\ p_1$. The choosing construct evaluates its expression e, choosing the first branch p_0 if the result is 0, and the second branch p_1 otherwise. The waiting construct waits for the leading process to make its choice. Similar to jumps, conditionals are synchronous. This means that all processes wait for one leading process to make the choice of

442 **Processes:**

p	$::=$	$\mathit{let}\ x = e;\ p$	compute
	$\mid$	$l(e) \mid l()$	jump
	$\mid$	$r \leftarrow e;\ p$	send
	$\mid$	$x \leftarrow r;\ p$	receive
	$\mid$	$\mathit{choose}\ (e)\ p\ \mathit{else}\ p$	choose
	$\mid$	$\mathit{wait}\ p\ \mathit{else}\ p$	wait
	$\mid$	$\mathit{exit}\ x$	exit
	$\mid$	done	done
P	$::=$	$\mathit{def}\ l(x : \tau)\ \{\ p\ \}, \ldots$	local program

Local types:

σ	$::=$	l	jump
	$\mid$	$r \leftarrow \tau;\ \sigma$	send
	$\mid$	$\tau \leftarrow r;\ \sigma$	receive
	$\mid$	$\oplus\ \sigma\ \sigma$	internal choice
	$\mid$	$\&\ \sigma\ \sigma$	external choice
	$\mid$	exit	exit
Σ	$::=$	$\mathit{type}\ l\ \{\ \sigma\ \}, \ldots$	local types

Fig. 10. Syntax of LangLocal.

which branch is taken. Finally, there are two terminal constructs for processes: $\mathit{exit}\ x$ for returning the value bound to a variable, and done, which does not return anything.

In analogy to global types, local types σ describe the behavior of processes, again mirorring the control flow, but omitting concrete terms. The local types $r \leftarrow \tau;\ \sigma$ and $\tau \leftarrow r;\ \sigma$ track which type of value is sent to or received from which role, respectively. The local types $\oplus\ \sigma_0\ \sigma_1$ for choosing a branch, and $\&\ \sigma_0\ \sigma_1$ for waiting for an external choice, however, do not need to track any role information.

3.2 Typing

Figure 11 presents the typing rules for expressions, choreographies, and processes. The typing rules for expressions are completely standard.

Choreography typing. In the typing judgment $\Delta \vdash c : \theta$, the choreography c is typed against the global type θ, and can use the bindings in Δ and the top-level labels defined in the program C. As C is not changed by any rule and is only used once in rule Jump, we leave it implicit for better readability.

In rule Comp for computations, the expression e evaluated must be well-typed, but it can only use those bindings from the context Δ that are annotated with the corresponding role annotated in the `let`-binding. The projection from the global context onto a local context is accomplished by the following function:

$$\llbracket \cdot \rrbracket_{\odot} : \text{Global context} \times \text{Role} \to \text{Local context}$$

$$\begin{array}{rcll} \llbracket \bullet \rrbracket_{r_0} & = & \bullet & \\ \llbracket \Delta, x : \tau\ @r \rrbracket_{r_0} & = & \llbracket \Delta \rrbracket_{r_0}, x : \tau & \text{if } r \equiv r_0 \\ & & \llbracket \Delta \rrbracket_{r_0} & \text{else} \end{array}$$

The binding is then added to the context for the remaining choreography c. The overall global type of the `let`-binding is the same as the global type of the remaining choreography, as computations

491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539

Expression Typing $\Gamma \vdash e : \tau$ (↑ ↑ ↓)

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{}{\Gamma \vdash n : \mathsf{Int}}\ \textsc{Lit} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}}\ \textsc{Add}$$

Choreography Typing $\Delta \vdash c : \theta$ (↑ ↑ ↓)

$$\frac{[\![\Delta]\!]_r \vdash e : \tau \qquad \Delta, x : \tau\ @r \vdash c : \theta}{\Delta \vdash \mathtt{let}\ x\ @r = e;\ c : \theta}\ \textsc{Comp} \qquad \frac{\mathtt{def}\ l(x : \tau\ @r)\,\{\,c\,\} \in C \qquad [\![\Delta]\!]_r \vdash e : \tau}{\Delta \vdash l(e) : l}\ \textsc{Jump}$$

$$\frac{[\![\Delta]\!]_{r_1} \vdash e : \tau \qquad \Delta, x : \tau\ @r_2 \vdash c : \theta \qquad r_1 \neq r_2}{\Delta \vdash x\ @r_2 \leftarrow e\ @r_1;\ c : r_2 \leftarrow\!\tau\ r_1;\ \theta}\ \textsc{Comm}$$

$$\frac{[\![\Delta]\!]_r \vdash e : \mathsf{Int} \qquad \Delta \vdash c_0 : \theta_0 \qquad \Delta \vdash c_1 : \theta_1}{\Delta \vdash \mathtt{if}\ (e\ @r)\ c_0\ \mathtt{else}\ c_1 : \mathtt{if}\ (r)\ \theta_0\ \mathtt{else}\ \theta_1}\ \textsc{If} \qquad \frac{x : \mathsf{Int} \in [\![\Delta]\!]_r}{\Delta \vdash \mathtt{end}\ x\ @r : \mathtt{end}}\ \textsc{End}$$

Process Typing $\Gamma \vdash p : \sigma$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash p : \sigma}{\Gamma \vdash \mathit{let}\ x = e;\ p : \sigma}\ \textsc{Let} \qquad \frac{\mathit{def}\ l(x : \tau)\,\{\,p\,\} \in P \qquad \Gamma \vdash e : \tau}{\Gamma \vdash l(e) : l}\ \textsc{Jump}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash p : \sigma}{\Gamma \vdash r \leftarrow e;\ p : r \leftarrow \tau;\ \sigma}\ \textsc{Send} \qquad \frac{\Gamma, x : \tau \vdash p : \sigma}{\Gamma \vdash x \leftarrow r;\ p : \tau \leftarrow r;\ \sigma}\ \textsc{Receive}$$

$$\frac{\Gamma \vdash e : \mathsf{Int} \qquad \Gamma \vdash p_0 : \sigma_0 \qquad \Gamma \vdash p_1 : \sigma_1}{\Gamma \vdash \mathit{choose}\ (e)\ p_0\ \mathit{else}\ p_1 : \oplus\ \sigma_0\ \sigma_1}\ \textsc{Choose} \qquad \frac{x : \mathsf{Int} \in \Gamma}{\Gamma \vdash \mathit{exit}\ x : \mathit{exit}}\ \textsc{Exit}$$

$$\frac{\Gamma \vdash p_0 : \sigma_0 \qquad \Gamma \vdash p_1 : \sigma_1}{\Gamma \vdash \mathit{wait}\ p_0\ \mathit{else}\ p_1 : \&\ \sigma_0\ \sigma_1}\ \textsc{Wait} \qquad \frac{}{\Gamma \vdash \mathit{done} : \mathit{exit}}\ \textsc{Done}$$

Fig. 11. Typing rules.

are not tracked in global types. In all other rules, the global type is determined by the construct being typed. In rule Jump, the argument supplied for the label jumped to must be compatible with the parameter of the label. Rule Comm requires the evaluated expression e to be well-typed in the context projected for the sending role r_1, and the binding added to the context for the remaining choreography is annotated with the receiving role r_2. The global type exactly mirrors this structure, also tracking the type of the expression e. We additionally require the roles r_1 and r_2 to be different, i.e., a process cannot communicate with itself. For conditionals in rule If, the evaluated expression must be of type Int, and the global type again mirrors the branching control flow of the construct. Finally, in rule End, the variable returned at role r must be present in the context.

Global type inference. Since global types closely follow the control structure of choreographies, the typing rules in Figure 11 are syntax-directed. They hence yield an algorithm for type inference. This is indicated in the typing judgments by marking the inputs with a ↑ and the output with a ↓. The premises exactly tell where and how turning a choreography into a typing derivation can go wrong:

$$\llbracket \cdot \rrbracket_{\odot} \;:\; \text{Global type} \times \text{Role} \rightarrow \text{Local type}$$

$$\begin{array}{rcll}
\llbracket l \rrbracket_{r_0} & = & l & \\
\llbracket r_2 \xleftarrow{}\tau\ r_1;\ \theta \rrbracket_{r_0} & = & r_2 \leftarrow \tau;\ \llbracket \theta \rrbracket_{r_0} & \text{if } r_1 \equiv r_0 \wedge r_2 \not\equiv r_0 \\
\llbracket r_2 \xleftarrow{}\tau\ r_1;\ \theta \rrbracket_{r_0} & = & \tau \leftarrow r_1;\ \llbracket \theta \rrbracket_{r_0} & \text{if } r_1 \not\equiv r_0 \wedge r_2 \equiv r_0 \\
\llbracket r_2 \xleftarrow{}\tau\ r_1;\ \theta \rrbracket_{r_0} & = & \llbracket \theta \rrbracket_{r_0} & \text{if } r_1 \not\equiv r_0 \wedge r_2 \not\equiv r_0 \\
\llbracket \texttt{if}\ (r)\ \theta_0\ \texttt{else}\ \theta_1 \rrbracket_{r_0} & = & \oplus\ \llbracket \theta_0 \rrbracket_{r_0}\ \llbracket \theta_1 \rrbracket_{r_0} & \text{if } r \equiv r_0 \\
\llbracket \texttt{if}\ (r)\ \theta_0\ \texttt{else}\ \theta_1 \rrbracket_{r_0} & = & \&\ \llbracket \theta_0 \rrbracket_{r_0}\ \llbracket \theta_1 \rrbracket_{r_0} & \text{if } r \not\equiv r_0 \\
\llbracket \texttt{end} \rrbracket_{r_0} & = & \mathit{exit} &
\end{array}$$

Fig. 12. Endpoint projection on types.

- Type inference for an expression fails. For our simple expression language, this can only happen if one of the operands of a primitive operator is not of the appropriate base type, or if the program is not well-scoped, i.e., if we encounter a variable that is free for the role the expression is computed at.
- The type inferred for a condition expression or for a final result is not an Int.
- The inferred type of the argument of a jump does not match the label's parameter (for multi-argument labels, there can also be an arity mismatch).
- The label jumped to is not defined in the program.
- The roles of a communication are not distinct.

If these causes are ruled out, inference for choreographies always succeeds, with a global type according to the inference rules.

Process typing. The judgment $\Gamma \vdash p : \sigma$ for process typing is similar to the one for choreography typing. We again leave the program P implicit. Just as some of the constructs of choreographies decompose into two process constructs, so do their typing rules. The rules Let and Jump for computations and synchronous jumps are mostly the same as their choreography counterparts. The main difference is that no roles are involved anymore, turning the rules into standard rules for `let`-bindings and jumps with argument. We omit the rule for jumps with no argument. The rule for communications decomposes into a rule Send for sending, and a rule Receive for receiving. Similarly, the rule for conditionals decomposes into a rule Choose for the leading process making the choice, and a rule Wait for the processes waiting for the external choice to be made. Finally, the rule for terminating a program decomposes into rules Exit and Done.

Endpoint projection on types. We now show how a global type can be projected onto indvidual local types for roles. This projection is written $\llbracket \theta \rrbracket_r$ and is defined in Figure 12. The global type projection function computes the local type for a role r_0 by examining the roles in the given global type. For jumps and terminal global types, this is trivial, as they are independent of the roles. For communication $r_2 \leftarrow\tau\ r_1;\ \theta$, there are three cases: if r_0 is the sending role r_1, the projection generates a sending local type; if r_0 is the receiving role r_2, it generates a receiving local type; otherwise r_0 is not involved in the communication and its local type only consists of the projection from the remaining global type. For conditionals `if` $(r)\ \theta_0$ `else` θ_1, there are two cases: if r_0 is the role r that makes the choice, the projection generates an internal choice; otherwise it generates a waiting external choice.

3.3 Semantics

We now describe the operational semantics as an abstract machine of a pool of processes that communicate with each other. Figure 13 shows the syntax. A machine state for a process is a

589 **Threads:**

v	$::=$	$0 \mid 1 \mid \ldots$	values
E	$::=$	$x \mapsto v, \ldots$	environments
T	$::=$	$\langle p \mid E \mid P \rangle$	threads

Pools:

Π	$::=$	$r \mapsto T \parallel r \mapsto T \parallel \ldots$	pools

Fig. 13. Syntax of the abstract machine.

Thread Typing $\vdash T : \sigma$

$$\frac{\Gamma \vdash p : \sigma \qquad \vdash E : \Gamma \qquad \vdash P \text{ OK}}{\vdash \langle p \mid E \mid P \rangle : \sigma}\ \textsc{Thread}$$

Pool Typing $\vdash \Pi : \theta$

$$\frac{\forall r_i \in \theta : \vdash T_i : [\![\theta]\!]_{r_i}}{\vdash r_1 \mapsto T_1 \parallel \ldots : \theta}\ \textsc{Pool}$$

Fig. 14. Typing rules of the abstract machine.

thread T, consisting of the process p, an environment E mapping variables to values v, and the local program P. For our simple expression language, the only values are those of base types, like integers. A pool Π is a mapping from roles to threads. We say that a thread is final if its process is either *exit* x or *done*, and a pool is final if all its threads are final.

The typing rules are given in Figure 14. A thread is typed against a local type σ in rule Thread, which requires the process p to be well-typed against σ with a local context Γ that must cover the environment E. The latter means that each value in E is pointwise well-typed against the corresponding type in Γ, written $\vdash E : \Gamma$. We do not show the straightforward definition here. Moreover, we require the program to be well-formed, written $\vdash P$ ok, which simply means that for each top-level label *def* $l(x : \tau)\,\{\,p\,\}$ in P, the body process is well-typed in the context of the parameter, i.e., $x : \tau \vdash p : \sigma$ for some local type σ (and likewise for labels without parameter).

A pool Π is typed against a global type θ by rule Pool, which requires that for each role r_i mentioned in θ, there is a thread T_i in the pool, and this thread is well-typed against the local local type projected for its role r_i from the global type θ. Typing each thread in a pool against the local type projected from a common global type is crucial to show deadlock freedom.

Stepping Relations. The stepping relations for threads and pools are defined in Figure 15. The steps a pool can make as a whole depend on which steps the threads in the pool can make. We therefore formulate the stepping relation for threads as a labeled transition system. The transition labels t track the information necessary to relate individual threads that must step at the same time to combine into a valid step of the whole pool. For ease of presentation, we again leave the program P implicit in the rules, as it does not change in any way.

With rule AnyLet, a pool can make a step if one of its threads can make a computation step, labeled with **let**, according to rule Let. The latter requires that the bound expression evaluates to a value v, which is then bound to the variable x in the environment for the remaining process. Rule AllJump performs a synchronous jump to some top-level label l. All threads must make a jump step for l by rule Jump, indicated by the transition label **jump** l. A jump for a thread involves evaluating the argument to a value, which then constitutes the new environment, and looking up the process of the top-level label jumped to. Two threads in a pool can exchange a message via rule Exchange. This requires one thread with role r_1 to make a send step with a transition label **send** $v\ r_2$ – tracking the role r_2 to which the value v is sent – and the thread with role r_2 to

Transition Labels:

$$t \quad ::= \quad \textbf{let} \mid \textbf{jump}\ l \mid \textbf{send}\ v\ r \mid \textbf{receive}\ v\ r \mid \textbf{choose}\ 0 \mid \textbf{choose}\ 1 \mid \textbf{wait}\ 0 \mid \textbf{wait}\ 1$$

Thread Steps $T \xrightarrow{t} T$

$$\frac{e \longrightarrow^* v}{\langle \mathit{let}\ x = e;\ p \mid E\rangle \xrightarrow{\textbf{let}} \langle p \mid E, x \mapsto v\rangle}\ \textsc{Let} \qquad \frac{e \longrightarrow^* v \qquad \mathit{def}\ l(x : \tau)\,\{\,p\,\} \in P}{\langle l(e) \mid E\rangle \xrightarrow{\textbf{jump}\ l} \langle p \mid x \mapsto v\rangle}\ \textsc{Jump}$$

$$\frac{e \longrightarrow^* v}{\langle e \leftarrow r;\ p \mid E\rangle \xrightarrow{\textbf{send}\ v\ r} \langle p \mid E\rangle}\ \textsc{Send} \qquad \frac{}{\langle x \leftarrow r;\ p \mid E\rangle \xrightarrow{\textbf{receive}\ v\ r} \langle p \mid E, x \mapsto v\rangle}\ \textsc{Receive}$$

$$\frac{e \longrightarrow^* v \qquad v \equiv 0}{\langle \mathit{choose}\ (e)\ p_0\ \mathit{else}\ p_1 \mid E\rangle \xrightarrow{\textbf{choose}\ 0} \langle p_0 \mid E\rangle}\ \textsc{Choose0} \qquad \frac{}{\langle \mathit{wait}\ p_0\ \mathit{else}\ p_1 \mid E\rangle \xrightarrow{\textbf{wait}\ 0} \langle p_0 \mid E\rangle}\ \textsc{Wait0}$$

$$\frac{e \longrightarrow^* v \qquad v \not\equiv 0}{\langle \mathit{choose}\ (e)\ p_0\ \mathit{else}\ p_1 \mid E\rangle \xrightarrow{\textbf{choose}\ 1} \langle p_1 \mid E\rangle}\ \textsc{Choose1} \qquad \frac{}{\langle \mathit{wait}\ p_0\ \mathit{else}\ p_1 \mid E\rangle \xrightarrow{\textbf{wait}\ 1} \langle p_1 \mid E\rangle}\ \textsc{Wait1}$$

Pool Steps $\Pi \longrightarrow \Pi$

$$\frac{T_i \xrightarrow{\textbf{let}} T_i' \qquad \forall j.\, i \neq j \implies T_j = T_j'}{\overline{r \mapsto T} \longrightarrow \overline{r \mapsto T'}}\ \textsc{AnyLet} \qquad \frac{T_1 \xrightarrow{\textbf{send}\ v\ r_2} T_1' \qquad T_2 \xrightarrow{\textbf{receive}\ v\ r_1} T_2' \qquad \forall i.\, i \neq 1 \wedge i \neq 2 \implies T_i = T_i'}{\overline{r \mapsto T} \longrightarrow \overline{r \mapsto T'}}\ \textsc{Exchange}$$

$$\frac{\forall i.\ T_i \xrightarrow{\textbf{jump}\ l} T_i'}{\overline{r \mapsto T} \longrightarrow \overline{r \mapsto T'}}\ \textsc{AllJump}$$

$$\frac{T_i \xrightarrow{\textbf{choose}\ 0} T_i' \qquad \forall j.\, i \neq j \implies T_j \xrightarrow{\textbf{wait}\ 0} T_j'}{\overline{r \mapsto T} \longrightarrow \overline{r \mapsto T'}}\ \textsc{If0} \qquad \frac{T_i \xrightarrow{\textbf{choose}\ 1} T_i' \qquad \forall j.\, i \neq j \implies T_j \xrightarrow{\textbf{wait}\ 1} T_j'}{\overline{r \mapsto T} \longrightarrow \overline{r \mapsto T'}}\ \textsc{If1}$$

Fig. 15. Abstract machine steps.

make a corresponding receive step with transition label **receive** $v\ r_1$ – tracking the value received from role r_1. All other threads are unchanged. The rule Send for a sending thread simply evaluates its expression to a value before sending it, and the rule Receive for a receiving thread binds the received value to the corresponding variable in the environment. Similar to jump steps, the rules If0 and If1 for making a synchronous choice require all threads to make a step, but only one thread, labeled with **choose** i, makes the choice, while all other threads, labeled with **wait** i, wait for the choice to be made. Rules Choose0 and Choose1 for the choosing thread evaluate the condition expression, and then pick the first branch if the result is 0, and the second branch otherwise. The waiting threads are informed of this choice in rules Wait0 and Wait1, and then step accordingly.

3.4 Properties

With the languages, their types, and their operational semantics in place, we want to show that they are well-behaved; in particular, that the types guarantee deadlock freedom. To this end, we show progress and preservation theorems for pools.

687 Theorem 3.1 (Progress).
688 *If* $\vdash \Pi : \theta$, *then either* Π *is final, or there exists* Π' *such that* $\Pi \longrightarrow \Pi'$.
689
690 Theorem 3.2 (Preservation).
If $\vdash \Pi : \theta$ *and* $\Pi \longrightarrow \Pi'$, *then there exists* θ' *such that* $\vdash \Pi' : \theta'$.
691
692 Deadlock freedom follows from these as a simple corollary. (Here we write $\longrightarrow^*$ for the reflexive-
693 transitive closure of the stepping relation, as usual.)
694
Corollary 3.3 (Deadlock Freedom).
695 *If* $\vdash \Pi : \theta$ *and* $\Pi \longrightarrow^* \Pi'$, *then either* Π' *is final, or there exists* Π'' *such that* $\Pi' \longrightarrow \Pi''$.
696
697 These results have been proven in Idris 2 via an intrinsically-typed mechanization, uploaded
698 as supplementary material. The proof for progress is straightforward. The proof for preservation
699 works as follows. First, we show that the stepping relation for threads preserves thread typability
700 (this is immediate from our intrinsically-typed definition of the stepping relation).
701
Lemma 3.4 (Preservation (Threads)).
702
703 *If* $\vdash T : \sigma$ *and* $T \xrightarrow{t} T'$, *then there exists* σ' *such that* $\vdash T' : \sigma'$.
704 Hence, if all threads in a pool are well-typed, then they are still well-typed after a pool step.
705
706 Corollary 3.5 (Preservation (Local types)).
707 *If for every* $r \mapsto T \in \Pi$ *we have* $\vdash T : \sigma$ *and* $\Pi \longrightarrow \Pi'$, *then for every* $r \mapsto T' \in \Pi'$ *there exists* σ'
708 *such that* $\vdash T' : \sigma'$.
709 For preservation, we then show that if the local types before a step were projected from some
710 global type, then so are the local types after the step.
711
712 Lemma 3.6 (Preservation (Global types)).
713 *If* $\vdash \Pi : \theta$ *and* $\Pi \longrightarrow \Pi'$ *such that for every* $r \mapsto T'$ *in* Π' *there exists* σ' *with* $\vdash T' : \sigma'$, *then there*
714 *exists* θ' *such that* $[\![\theta']\!]_r \equiv \sigma'$ *for all* r, *i.e.,* $\vdash \Pi' : \theta'$.
715 Together, Corollary 3.5 and Lemma 3.6 directly imply Theorem 3.2, because if a pool is well-
716 typed, then all threads in it are well-typed by definition. The crucial step thus is to show that all of
717 the pool steps preserve synchronization between threads. This is considerably simplified by having
718 synchronous jumps and conditionals.
719
720 ### 3.5 Endpoint Projection
721 So far, we have not considered *endpoint projection for programs*, demonstrating that we have de-
722 coupled deadlock freedom for pools of processes from any concrete endpoint projection for chore-
723 ographies. The above development only makes use of the straightforward endpoint projection *for*
724 *types*. Hence, our results above are valid for any processes that behave according to the local types
725 projected from any global type. We now define an endpoint projection for choreographies, and
726 then show that the local types of the projected processes indeed have this property. It is presented
727 in Figure 16.
728 The endpoint projection function computes the process for a role r_0 by examining the roles in the
729 given choreography. A local computation is only kept for the role it is supposed to be performed at.
730 Jumps are kept for all roles, but the argument is only kept for the role annotated at the parameter
731 of the label jumped to. For communication $x \,@r_2 \leftarrow e \,@r_1;\ c$, there are again three cases: if r_0 is the
732 sending role r_1, a sending process is generated; if r_0 is the receiving role r_2, a receiving process is
733 generated; otherwise r_0 is not involved in the communication, and its process only consists of the
734 projection from the remaining choreography. For conditionals `if` $(e \,@r)\ c_0$ `else` c_1, there are again
735

736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784

$$
\begin{array}{rcll}
 & \llbracket\cdot\rrbracket_{\odot} : \text{Choreography} \times \text{Role} \rightarrow \text{Process} & & \\
\llbracket \texttt{let}\ x\ @r = e;\ c \rrbracket_{r_0} & = & \mathit{let}\ x = e;\ \llbracket c \rrbracket_{r_0} & \text{if } r \equiv r_0 \\
\llbracket \texttt{let}\ x\ @r = e;\ c \rrbracket_{r_0} & = & \llbracket c \rrbracket_{r_0} & \text{if } r \not\equiv r_0 \\
\llbracket l(e) \rrbracket_{r_0} & = & l(e) & \text{if } r \equiv r_0 \text{ where } \texttt{def}\ l(\ldots\ @r)\ \{ c \} \\
\llbracket l(e) \rrbracket_{r_0} & = & l() & \text{if } r \not\equiv r_0 \text{ where } \texttt{def}\ l(\ldots\ @r)\ \{ c \} \\
\llbracket x\ @r_2 \leftarrow e\ @r_1;\ c \rrbracket_{r_0} & = & r_2 \leftarrow e;\ \llbracket c \rrbracket_{r_0} & \text{if } r_1 \equiv r_0 \land r_2 \not\equiv r_0 \\
\llbracket x\ @r_2 \leftarrow e\ @r_1;\ c \rrbracket_{r_0} & = & x \leftarrow r_1;\ \llbracket c \rrbracket_{r_0} & \text{if } r_1 \not\equiv r_0 \land r_2 \equiv r_0 \\
\llbracket x\ @r_2 \leftarrow e\ @r_1;\ c \rrbracket_{r_0} & = & \llbracket c \rrbracket_{r_0} & \text{if } r_1 \not\equiv r_0 \land r_2 \not\equiv r_0 \\
\llbracket \texttt{if}\ (e\ @r)\ c_0\ \texttt{else}\ c_1 \rrbracket_{r_0} & = & \mathit{choose}\ (e)\ \llbracket c_0 \rrbracket_{r_0}\ \mathit{else}\ \llbracket c_1 \rrbracket_{r_0} & \text{if } r \equiv r_0 \\
\llbracket \texttt{if}\ (e\ @r)\ c_0\ \texttt{else}\ c_1 \rrbracket_{r_0} & = & \mathit{wait}\ \llbracket c_0 \rrbracket_{r_0}\ \mathit{else}\ \llbracket c_1 \rrbracket_{r_0} & \text{if } r \not\equiv r_0 \\
\llbracket \texttt{end}\ x\ @r \rrbracket_{r_0} & = & \mathit{exit}\ x & \text{if } r \equiv r_0 \\
\llbracket \texttt{end}\ x\ @r \rrbracket_{r_0} & = & \mathit{done} & \text{if } r \not\equiv r_0
\end{array}
$$

Fig. 16. Endpoint Projection.

two cases: if r_0 is the role r that makes the choice, a choosing process is generated; otherwise a waiting process is generated. Finally, ending a program end x $@r$ becomes a *exit* x for role r, and *done* for all other roles.

The following theorem shows that for a well-typed choreography, the projected endpoint processes are well-typed against the local types projected from the global type of the choreography, for any role in the program.

Theorem 3.7 (Typability Preservation).
If $\Delta \vdash c : \theta$, *then* $\llbracket\Delta\rrbracket_r \vdash \llbracket c\rrbracket_r : \llbracket\theta\rrbracket_r$ *for any role* r *in the program.*

As a corollary, we obtain that a pool consisting of one thread obtained by projection for each role in the program from a well-typed global choreography, is again well-typed. Here we write $\vdash E : \Delta$ to mean that there is one well-typed value in E for each binding in Δ, similar to the judgment $\vdash E : \Gamma$ for local contexts. Moreover, we write $\llbracket E\rrbracket^{\Delta}_{r_0}$ for the environment filtered for role r_0, according to the role annotations in Δ.

Corollary 3.8 (Typability Preservation (Pool)).
If $\Delta \vdash c : \theta$ *with* $\vdash E : \Delta$ *and* $\Pi = r_1 \mapsto \langle \llbracket c\rrbracket_{r_1} \mid \llbracket E\rrbracket^{\Delta}_{r_1} \mid \llbracket C\rrbracket_{r_1}\rangle \parallel \ldots$ *contains one thread for each role* r_i *in the program, then* $\vdash \Pi : \theta$.

Hence, a pool obtained by endpoint projection for all roles is indeed deadlock-free. This is again proven by our intrinsically-typed mechanization.

4 Discussion

We now discuss the design of our system, more specifically, the synchronicity of our constructs. We distinguish three axes of synchronicity in our language: synchronous communication, synchronous control transfers, and synchronous conditionals. While it is rather common in choreographic programming, and also with multiparty session types (MPST), to make explicit communication synchronous, we have decided to also make control transfer and conditionals synchronous. The latter is particularly uncommon in existing systems, as we will also discuss further in Section 6.

A major advantage of having synchronous constructs is that they simplify meta-theoretical considerations, in particular, the proof of deadlock freedom. Moreover, we believe that the loss of expressivity is modest, as is corroborated by several case studies we present in Section 5. A major downside, however, is that there may be too much synchronization, forcing processes to wait for

785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833

other processes, even though they could already safely make progress independently from the processes they wait for. We will discuss each of the three axes in turn.

Synchronous Communication. Whenever two roles in our system communicate, the exchange only happens when both sender and receiver are ready, in the style of rendezvous channels in CSP [Hoare 1978]. This is in contrast to fully asynchronous communication as found, for example, in the actor model [Hewitt et al. 1973], where the sender dispatches its message independently of the state of the receiver. The latter hence needs an unbounded buffer to guarantee that no message is lost. Synchronous communication needs no such buffer and is thus very memory-efficient. The downside is that it cannot directly model common asynchronous patterns, such as pipelined stages with decoupled producers and consumers. A middle ground is the usage of a bounded buffer on the receiver side, so that the receiver only blocks the sender if the buffer is full. This allows for asynchronous communication as long as the buffer is not full, while keeping the memory usage under control. It would be interesting to see how the introduction of asynchronous communication affects our meta-theoretical results.

Synchronous Control. Following Hirsch and Garg [2022], transfers of control with jumps are also synchronous in our system, i.e., processes wait until all processes are ready to jump. This considerably simplifies the proof of deadlock freedom, because the processes do not get "out of sync" with each other and with the common global type. That is, no process in a pool can be ahead of the other processes, and hence we can always type the whole pool with a global type, as shown by Lemma 3.6.

In practice, jumps often occur at loop boundaries. If the loop is infinite. synchronous jumps prevent "runaway" processes from outpacing the others by an unbounded number of steps. However, if the loop contains a conditional before the recursive jump, each iteration already synchronizes all processes, leading to unnecessary double-synchronization. Moreover, synchronous jumps rule out patterns where different subsets of roles progress at different rates or join at merge points without global synchronization. It would be interesting to explore the possibility of selectively introducing asynchronous control transfers, in particular if we extend **LangGlobal** with explicit parallel sub-choreographies or join points.

Synchronous Conditionals. Finally, and most uncommonly, our conditionals always synchronize all roles. Typically, in choreographic programming or MPST, roles are informed through explicit communication about the choice of which branch is taken. During endpoint projection, differing branches of processes are then merged, and projection either fails if the branches cannot be merged, or further messages are inserted to guarantee mergeability (cf. Section 6). Synchronous conditionals avoid the necessity of such a merge procedure, making endpoint projection straightforward.

Moreover, similar to synchronous control transfer, synchronous conditionals simplify our proof of deadlock freedom, because it is always clear which part of the global type can type the pool of processes after reduction of the conditional. However, they also may introduce unnecessary synchronization and superfluous control messages if some roles are unaffected by the branching decision, potentially limiting concurrency and adding coordination overhead. We expect such cases to be rare in workloads with a focus on parallelism, as found, for example, in Bulk Synchronous Parallel algorithms [Valiant 1990], but larger case studies may reveal patterns where partially synchronized conditionals or finer-grained *knowledge-of-choice* (see Section 6) mechanisms would be advantageous.

Summary. Synchronicity greatly simplifies the proofs of meta-theoretic properties. Moreover, it mainly influences time and space usage and the class of concurrent interaction patterns that can be expressed directly, but does not affect functional outcomes. It is well-suited for distributed parallel

programming, but less expressive for genuinely concurrent, highly asynchronous algorithms. By featuring it, we trade some expressiveness for simpler resource reasoning and a clean and simple semantics. A deeper understanding of this trade-off calls for experimentation with realistic parallel and concurrent benchmarks, including those that stress asynchrony. It will be interesting to explore the integration of asynchronous behavior into synchronous choreographies, to see how far our deadlock freedom guarantees can be pushed in these settings.

5 Case Studies

In this section, we demonstrate the expressiveness of our choreographic language by implementing the *Less is More* case studies [Castro-Perez et al. 2026; Scalas and Yoshida 2019]. They stem from the literature on multiparty session types, and we will discuss their origin further in Section 6. They also illustrate the consequences of our design decision of having synchronous communication, conditionals, and control.

Following the original textual description taken from [Castro-Perez et al. 2026; Scalas and Yoshida 2019], we implement three of the four case studies as choreographies in **LangGlobal**, and discuss a possible extension to also implement the last case study in the future. With our work, the developer *only* writes the choreography, and both its global type is automatically inferred and the processes are automatically generated by endpoint projection from the choreography. Additionally, the local types are generated by projection from the global type. This treats global and local types as internal compiler artifacts that justify endpoint projection.

OAuth2 Fragment. This case study describes a simple part of an authorization protocol. The `Server` starts with a decision how the rest of the system should behave. In our language, we automatically propagate this branching decision to all roles, avoiding the necessity to explicitly communicate it and avoiding the possibility of forgetting to do so. We can implement it as follows:

Description:

Server tells Client it can continue the session by logging in, or it cancels the session. In the former case, Client tells Authorisation Service its password, after which Authorisation Service tells Server whether the login succeeded. In the latter case, Client tells Authorisation Service to quit. We can implement it as follows:

Choreography:

```
if (continue @Server) {
  password @Authorizer ← "asdf" @Client
  let check @Authorizer = password == "asdf"
  success @Server ← check @Authorizer
  end
} else { end }
```

Global Type (Inferred):

```
if (Server) {
  Authorizer ←String Client

  Server ←Bool Authorizer
  end
} else { end }
```

Recursive Two-Buyers Protocol. This case study describes two persons buying an item at an online store. Again, since our conditionals automatically inform all parties, no explicit communication of the branching decision is required. The potentially infinite bargaining loop is a recursive definition, which naturally leads to a recursive protocol.

Description:

Alice asks Store for a quote of an item. Store tells Alice the price. Alice asks Bob to split the price

or to cancel the session.
In the former case, Bob tells Alice whether or not he is willing to split. If he is, then Alice tells Store that the purchase goes through, but if not, then she asks Bob again to split the price or to cancel the session.
In the latter case, Alice tells Store that no purchase will be made.

Choreography:

```
item  @Store ← "tapl" @Alice
price @Alice ← 20     @Store
price @Bob   ← price  @Alice
bargain(item, price)

def bargain(item @Store, price @Bob) {
  if (continue() @Alice) {
    if (willing(price) @Bob) {
      okay @Store ← true @Bob
      end
    } else {
      bargain(item, price)
    }
  } else { end }
}
```

Global Type (Inferred):

```
Store ←String Alice
Alice ←Int Store
Bob   ←Int Alice
bargain

type bargain {
  if (Alice) {
    if (Bob) {
      Store ←Bool Bob
      end
    } else {
      bargain(item, price)
    }
  } else { end }
}
```

In the original, the protocol is recursive but the processes are not. Rather Alice immediately cancels and Bob always is willing, which corresponds to `continue()` returning `false` and `willing(price)` returning `true` in this example. From such a non-recursive choreography we would not generate a recursive protocol, which is why we chose to leave the choices of Alice and Bob open.

Recursive Map/Reduce. This case study describes repeated iterations of mapping and reducing, similar to our introductory example (Section 2.1). Again, there is no explicit propagation of decisions. The original `Reducer` would stop after one iteration, which again would lead to a non-recursive protocol. We choose to leave open the choice with `continue(result)`.

Description:

Mapper tells Worker 1 and Worker 2 to each process a datum.
Worker 1 and Worker 2 tell Reducer the results of their processing.
Reducer tells Mapper to enter another iteration of mapping/reducing or to stop.
In the latter case, Mapper tells Worker 1 and Worker 2 to stop, too.

932
933 **Choreography:**
934
935
936
937
938
939
940
941
942
943
944
945
946

```
def loop() {
  datum1 @Worker1 ← 123 @Mapper
  datum2 @Worker2 ← 123 @Mapper
  let result1 @Worker1 = process(datum1)
  let result2 @Worker2 = process(datum2)
  result1 @Reducer ← result1 @Worker1
  result2 @Reducer ← result2 @Worker2
  let result @Reducer =
    combine(result1, result2)
  if (continue(result) @Reducer) {
    loop()
  } else { end }
}
```

Global Type (Inferred):

```
type loop() {
  Worker1 ←Int Mapper
  Worker2 ←Int Mapper


  Reducer ←Int Worker1
  Reducer ←Int Worker2


  if (Reducer) {
    loop
  } else { end }
}
```

947 All case studies so far were straightforward to express using our choreographic language. We
948 have demonstrated that in choreographic programming, there is no need to create and maintain
949 separate artifacts for the global and local types and the processes; writing the choreography alone
950 is sufficient. The next case study, however, poses a problem.
951
952 *Independent Multiparty Workers.* This case study describes the behavior of multiple groups of
953 workers that, after receiving a start item, proceed independently. While we can describe the behav-
954 ior of a single group of workers, we currently do not have the ability to compose choreographies
955 in parallel.
956 **Description:**
957 Starter tells Worker A1 and Worker A2 to each process a
958 datum. In parallel:
959 1) Worker A1 tells Worker B1 to process the datum or to
960 stop. In the former case, Worker B1 tells Worker C1 to
961 process the datum, after which Worker C1 tells Worker
962 A1 the result, after which Worker A1 again tells Worker
963 B1 to process the datum or to stop. In the latter case,
964 Worker B1 tells Worker C1 to stop, too.
965 2) Workers A2, B2, C2 follow the same sub-protocol as
966 Workers A1, B1, C1, independently.

Similar Choreography:

```
def loop1(x1 @WorkerA1) {
  if (continue(x1) @WorkerA1) {
    x1 @WorkerB1 ← x1 @WorkerA1
    let y1 @WorkerB1 = process(x1)
    y1 @WorkerC1 ← y1 @WorkerB1
    let z1 @WorkerC1 = process(y1)
    z1 @WorkerA1 ← z1 @WorkerC1
    loop1(z1)
  } else { end }
}
```

967
968 In the future we might investigate the possibility of having enclaves [Bates and Near 2024] of
969 independent roles acting in sub-choreographies.
970
971 We have evaluated the expressiveness of our choreographic language on multiple case studies
972 from the literature. While generally we could express the examples, we lack a feature for indepen-
973 dent parallel composition. Moreover, we have demonstrated that in comparison with multiparty
974 session types, programmers just write one artifact, and we automatically generate the others.
975
6 Related Work

976 *Choreographic Programming and Multiparty Session Types.* Multiparty Session Types (MPST)
977 provide a type-theoretic foundation for verifying communication protocols among multiple partic-
978 ipants. They extend binary session types to settings involving more than two parties, ensuring that
979 programs adhere to a global specification of communication. The notion of a *global type*, describing
980

981 overall system behavior, and its projection onto *local types*, prescribing each participant's behav-
982 ior, plays a central role in this framework. Early formulations of MPST focused on asynchronous
983 communication models, allowing participants to exchange messages without assuming strict syn-
984 chronization. The works by Bettini et al. [2008]; Coppo et al. [2016] and Honda et al. [2008, 2016]
985 established the core foundations of asynchronous MPST. In contrast, the synchronous model en-
986 forces tighter coordination, where interactions proceed in lockstep. Synchronous interpretations
987 were developed by Bejleri and Yoshida [2008]; Kouzapas and Yoshida [2013, 2014].
988 Both asynchronous and synchronous variants of MPST share a core goal with choreographic
989 programming: to reject programs, at compile time, that could lead to communication mismatches
990 or deadlocks. In both frameworks, the endpoint-projection function translates a global protocol
991 (or a choreography) onto local behaviors for individual participants. Certain programs—for exam-
992 ple, where a participant must act differently in alternative branches without being informed of the
993 actual branch—cannot be sensibly projected. To handle this, projection employs a *merge* operator
994 that combines the behaviors of uninformed roles across branches. Scalas and Yoshida [2018] dis-
995 tinguish between early systems with an conservative approach and a more permissive approach.
996 They call the conservative approach, which only merges branches where an endpoint is explicitly
997 told which branch it is in, *plain merge*. This yields strong safety guarantees, but rejects many be-
998 nign cases. They call the more permissive approach, which enlarges the projection domain, but
999 complicates the metatheory, *full merge*.
1000 More precisely, with the original MPST, any set of local types projected from a global type was
1001 consistent, ensuring deadlock freedom for well-typed systems. However, projection was only de-
1002 finable for a limited set of global types. To demonstrate this, Scalas and Yoshida [2019] presented
1003 four case studies (cf. Section 5) which are not projectable with MPST. To implement these case
1004 studies with classical multiparty session types [Honda et al. 2008], one would need to write both
1005 the global type, multiple processes, and for each process a local type. Then, MPST would type check
1006 them by checking that the specified local types match the processes, and that the specified global
1007 type does project onto the specified local types. However, the projection of the global types, or
1008 more precisely the merging of branches that happens during projection, would simply fail. Alter-
1009 native formulations have been explored as well, performing the type checking in a different way
1010 and omitting one of these redundant definitions, thereby avoiding problems with limited range
1011 of endpoint projection: Scalas and Yoshida [2019] proposed to only write the processes and their
1012 local types, and infer a global type – if one exists – via model checking automatically. Castro-Perez
1013 et al. [2026] proposed to only write the processes and a single global type, and infer the local types
1014 – if they exist – automatically. Both papers demonstrated that their approach works for the case
1015 studies. This ongoing development reflects the evolving understanding of MPST and their central
1016 role in the formal verification of distributed systems.
1017 A similar move away from rejection and towards repair of bad programs is visible on the chore-
1018 ographic programming side. Castagna et al. [2011] introduced the concept of *knowledge of choice*,
1019 which is often quoted in choreographic programming literature. The concept describes the prob-
1020 lem where a party in a distributed program would need to act without being informed. Carbone
1021 et al. [2012] expanded on these fundamentals by formally connecting global choreographies with
1022 projected process behaviors, proposing the classic technique for proving deadlock freedom via
1023 soundness and completeness proofs for choreographic programming languages. Montesi [2013]
1024 consolidated the emerging notion of *Choreographic Programming*, detailing its semantics and guar-
1025 antees, such as deadlock freedom by design. Cruz-Filipe and Montesi [2016] then contributed a min-
1026 imal semantic core to clarify the expressiveness and correctness of Choreographic Programming.
1027 Cruz-Filipe et al. [2022] proposed a formal model for choreographies in a functional programming
1028 setting. Eventually, library-based encodings of choreographic programming like *HasChor* were
1029

proposed. They automatically broadcast control messages to inform participants [Shen et al. 2023]. Cruz-Filipe and Montesi [2023] brought this idea back to choreography compilers, implemented as a program translation that "repairs" choreographies with knowledge-of-choice problems, by adding the missing control-flow messages to inform parties when they need to act. To bridge the gap between libraries and compilers, Krook and Hammersberg [2024] showed how GHC's rewrite rules can be used to perform endpoint projection over a choreographic library as an optimization pass. To respond to the criticique that broadcasting sends messages to too many participants, even those not involved in the conditional, enclaves have been proposed to further reduce unnecessary synchronization and replicate the precision of compiler-based projections [Bates et al. 2025; Bates and Near 2024].

Besides session types and choreographic programming, multitier programming also lives in a similar design space, for a discussion of the similarities and difference, we recommend the survey by Giallorenzo et al. [2021].

In this work, we follow the above-described evolution, eliminating failure during projection by avoiding merging operations for branches entirely. Inspired by the library-based approach, we have adopted broadcasted control messages: every participant is explicitly informed of branch choices. This guarantees that endpoint projection always succeeds, although it may lead to redundant synchronization or additional messages, as discussed in Section 4.

Mechanizations and Verification. Cruz-Filipe et al. [2021] pioneered mechanized certification of choreography compilation in Rocq/Coq. Pohjola et al. [2022] expanded verification coverage to end-to-end correctness with the Kalas compiler. Hirsch and Garg [2022] tackled higher-order typed choreographies in the functional setting with *Pirouette*. Cruz-Filipe et al. [2023a] followed with a certified higher-order compiler via the *hacc* framework. Cruz-Filipe et al. [2023b] then provided a fully formalized theory of Choreographic Programming. Thiemann [2023] further explored dependent type systems for session calculi with first-class callbacks. We have mechanized our system via an intrinsically-typed implementation in Idris 2 [Brady 2020].

Asynchrony. Carbone and Montesi [2013] extended global typing with asynchronous semantics that retain deadlock freedom by design. Cruz-Filipe and Montesi [2017b] first investigated the relationship between synchronous and asynchronous communication in choreographies, establishing that asynchronous semantics can preserve correctness guarantees. In a companion paper in the same year [Cruz-Filipe and Montesi 2017a], they proposed a concrete encoding technique to represent asynchrony directly within synchronous choreographic models, laying the groundwork for further asynchronous semantics. Lugovic and Montesi [2024] extended this direction by integrating practical, interoperable asynchronous communication patterns into real-world choreographic systems. Our work, in contrast, does not have any asynchronous operations.

7 Conclusions and Future Work

We have presented a choreographic approach to developing deadlock-free distributed systems in which session types are inferred and used as internal compiler artifacts rather than hand-written specifications. Our starting point is a simple choreographic language, **LangGlobal**, and a process language, **LangLocal**, together with an endpoint projection from choreographies to processes. Choreographies are typed by global types, while processes are typed by local types projected from these global types, thereby connecting the global description of a protocol with the local views of individual roles.

On the semantic side, we have formalized an operational semantics for pools of communicating processes. We have proved progress and preservation theorems for pools that are well-typed with respect to local types projected from a common global type. These results entail deadlock freedom

for all such pools, independently of how the processes were obtained. We have further shown that global types can be inferred directly from choreographies, and that endpoint projection preserves typability: processes generated from a well-typed choreography are guaranteed to implement the local types projected from its inferred global type.

Conceptually, this factors the deadlock freedom argument through global and local session types, decoupling the proof from any particular realization of endpoint projection. In contrast to traditional choreography-based proofs, our guarantees do not rely on a tight simulation between a fixed choreography and its generated processes. Instead, any implementation that satisfies the inferred local types enjoys the same deadlock freedom property. This perspective combines the rapid, choreographic development experience with the modular verification style of multiparty session types, and it supports post-hoc refactoring or replacement of processes as long as their local types are preserved.

Looking ahead, we see several directions for further work. One line is to enrich the choreographic language with more realistic features such as richer data types, asynchronous communication and fault-tolerant primitives, or dynamic role creation, while maintaining a clean typing and inference story. Another is to investigate compositional reasoning principles that allow larger systems to be assembled from independently verified choreographies and protocols. Finally, we are interested in exploring tool support that makes inferred global and local types directly usable by programmers. Examples of tool support could include protocol documentation, guiding refactorings, or a basis for interoperability between independently developed components.

References

Mako Bates, Shun Kashiwa, Syed Jafri, Gan Shen, Lindsey Kuper, and Joseph P. Near. 2025. Efficient, Portable, Census-Polymorphic Choreographic Programming. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1143–1166. doi:10.1145/3729296

Mako Bates and Joseph P. Near. 2024. We Know I Know You Know; Choreographic Programming With Multicast and Multiply Located Values. *CoRR* abs/2403.05417 (2024). arXiv:2403.05417 doi:10.48550/ARXIV.2403.05417

Andi Bejleri and Nobuko Yoshida. 2008. Synchronous Multiparty Session Types. In *Proceedings of the First Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@DisCoTec 2008, Oslo, Norway, June 7, 2008 (Electronic Notes in Theoretical Computer Science, Vol. 241)*, Vasco T. Vasconcelos and Nobuko Yoshida (Eds.). Elsevier, 3–33. doi:10.1016/J.ENTCS.2009.06.002

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5201)*, Franck van Breugel and Marsha Chechik (Eds.). Springer, 418–433. doi:10.1007/978-3-540-85361-9_33

Edwin Brady. 2020. *Idris 2: Quantitative Type Theory in Action.* Technical Report. University of St Andrews, Scotland, UK. https://www.type-driven.org.uk/edwinb/papers/idris2.pdf

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2012. Structured Communication-Centered Programming for Web Services. *ACM Trans. Program. Lang. Syst.* 34, 2 (2012), 8:1–8:78. doi:10.1145/2220365.2220367

Marco Carbone and Fabrizio Montesi. 2013. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 263–274. doi:10.1145/2429069.2429101

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2011. On Global Types and Multi-party Sessions. In *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6722)*, Roberto Bruni and Jürgen Dingel (Eds.). Springer, 1–28. doi:10.1007/978-3-642-21461-5_1

David Castro-Perez, Francisco Ferreira, and Sung-Shik Jongmans. 2026. A Synthetic Reconstruction of Multiparty Session Types. *Proc. ACM Program. Lang.* 10, POPL, Article 50 (Jan. 2026), 29 pages. doi:10.1145/3776692

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global progress for dynamically interleaved multiparty sessions. *Math. Struct. Comput. Sci.* 26, 2 (2016), 238–302. doi:10.1017/S0960129514000188

Luís Cruz-Filipe, Eva Graversen, Lovro Lugovic, Fabrizio Montesi, and Marco Peressotti. 2022. Functional Choreographic Programming. In *Theoretical Aspects of Computing - ICTAC 2022 - 19th International Colloquium, Tbilisi, Georgia, September 27-29, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13572)*, Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu (Eds.). Springer, 212–237. doi:10.1007/978-3-031-17715-6_15

Luís Cruz-Filipe, Lovro Lugovic, and Fabrizio Montesi. 2023a. Certified Compilation of Choreographies with hacc. In *Formal Techniques for Distributed Objects, Components, and Systems - 43rd IFIP WG 6.1 International Conference, FORTE 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13910)*, Marieke Huisman and António Ravara (Eds.). Springer, 29–36. doi:10.1007/978-3-031-35355-0_3

Luís Cruz-Filipe and Fabrizio Montesi. 2016. A Core Model for Choreographic Programming. In *Formal Aspects of Component Software - 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10231)*, Olga Kouchnarenko and Ramtin Khosravi (Eds.). 17–35. doi:10.1007/978-3-319-57666-4_3

Luís Cruz-Filipe and Fabrizio Montesi. 2017a. Encoding asynchrony in choreographies. In *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng (Eds.). ACM, 1175–1177. doi:10.1145/3019612.3019901

Luís Cruz-Filipe and Fabrizio Montesi. 2017b. On Asynchrony and Choreographies. In *Proceedings 10th Interaction and Concurrency Experience, ICE@DisCoTec 2017, Neuchâtel, Switzerland, 21-22nd June 2017 (EPTCS, Vol. 261)*, Massimo Bartoletti, Laura Bocchi, Ludovic Henrio, and Sophia Knight (Eds.). 76–90. doi:10.4204/EPTCS.261.8

Luís Cruz-Filipe and Fabrizio Montesi. 2023. Now It Compiles! Certified Automatic Repair of Uncompilable Protocols. In *14th International Conference on Interactive Theorem Proving, ITP 2023, Białystok, Poland, July 31 - August 4, 2023 (LIPIcs, Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:19. doi:10.4230/LIPICS.ITP.2023.11

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021. Certifying Choreography Compilation. In *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12819)*, Antonio Cerone and Peter Csaba Ölveczky (Eds.). Springer, 115–133. doi:10.1007/978-3-030-85315-0_8

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2023b. A Formal Theory of Choreographic Programming. *J. Autom. Reason.* 67, 2 (2023), 21. doi:10.1007/S10817-023-09665-3

Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. 2021. Multiparty Languages: The Choreographic and Multitier Cases (Pearl). In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:27. doi:10.4230/LIPICS.ECOOP.2021.22

Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Stanford, USA) *(IJCAI'73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

Andrew K. Hirsch and Deepak Garg. 2022. Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–27. doi:10.1145/3498684

C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677. doi:10.1145/359576.359585

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 273–284. doi:10.1145/1328438.1328472

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67. doi:10.1145/2827695

Dimitrios Kouzapas and Nobuko Yoshida. 2013. Globally Governed Session Semantics. In *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8052)*, Pedro R. D'Argenio and Hernán C. Melgratti (Eds.). Springer, 395–409. doi:10.1007/978-3-642-40184-8_28

Dimitrios Kouzapas and Nobuko Yoshida. 2014. Globally Governed Session Semantics. *Log. Methods Comput. Sci.* 10, 4 (2014). doi:10.2168/LMCS-10(4:20)2014

Robert Krook and Samuel Hammersberg. 2024. Welcome to the Parti(tioning) (Functional Pearl): Using Rewrite Rules and Specialisation to Partition Haskell Programs. In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium, Haskell 2024, Milan, Italy, September 6-7, 2024*, Niki Vazou and J. Garrett Morris (Eds.). ACM, 27–40. doi:10.1145/3677999.3678276

Lovro Lugovic and Fabrizio Montesi. 2024. Real-World Choreographic Programming: Full-Duplex Asynchrony and Interoperability. *Art Sci. Eng. Program.* 8, 2 (2024). doi:10.22152/PROGRAMMING-JOURNAL.ORG/2024/8/8

Fabrizio Montesi. 2013. *Choreographic Programming*. Ph. D. Dissertation. IT University of Copenhagen. https://www.fabriziomontesi.com/files/choreographic-programming.pdf

Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. 2022. Kalas: A Verified, End-To-End Compiler for a Choreographic Language. In *13th International Conference on Interactive Theorem Proving, ITP 2022, Haifa, Israel, August 7-10, 2022 (LIPIcs, Vol. 237)*, June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:18. doi:10.4230/LIPICS.ITP.2022.27

Alceste Scalas and Nobuko Yoshida. 2018. Multiparty session types, beyond duality. *J. Log. Algebraic Methods Program.* 97 (2018), 55–84. doi:10.1016/J.JLAMP.2018.01.001

Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.* 3, POPL, Article 30 (Jan. 2019), 29 pages. doi:10.1145/3290343

Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). *Proc. ACM Program. Lang.* 7, ICFP (2023), 541–565. doi:10.1145/3607849

Peter Thiemann. 2023. Intrinsically Typed Sessions with Callbacks (Functional Pearl). *Proc. ACM Program. Lang.* 7, ICFP (2023), 711–739. doi:10.1145/3607854

Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111. doi:10.1145/79173.79181