


Now It Compiles!

Certified Automatic Repair of Uncompilable Protocols

Luís Cruz-Filipe ✉ 

Department of Mathematics and Computer Science, University of Southern Denmark

Fabrizio Montesi ✉ 

Department of Mathematics and Computer Science, University of Southern Denmark

Abstract

Choreographic programming is a paradigm where developers write the global specification (called choreography) of a communicating system, and then a correct-by-construction distributed implementation is compiled automatically. Unfortunately, it is possible to write choreographies that cannot be compiled, because of issues related to an agreement property known as knowledge of choice. This forces programmers to reason manually about implementation details that may be orthogonal to the protocol that they are writing.

Amendment is an automatic procedure for repairing uncompilable choreographies. We present a formalisation of amendment from the literature, built upon an existing formalisation of choreographic programming. However, in the process of formalising the expected properties of this procedure, we discovered a subtle counterexample that invalidates the original published and peer-reviewed pen-and-paper theory. We discuss how using a theorem prover led us to both finding the issue, and stating and proving a correct formulation of the properties of amendment.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Automated reasoning; Software and its engineering → Concurrent programming languages

Keywords and phrases choreographic programming, theorem proving, compilation, program repair

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.26

Related Version *Preprint*: <https://arxiv.org/abs/2302.14622>

Funding This work was partially supported by Villum Fonden, grants no. 29518 and 50079, and the Independent Research Fund Denmark, grant no. 0135-00219.

Acknowledgements We thank the anonymous reviewers for their useful comments, which helped us improve the quality of this article.

1 Introduction

Programming correct implementations of protocols for communicating systems is challenging, because it requires writing a correct program for each participant that performs the right send and receive actions at the right times [23]. *Choreographic programming* [27] is an emerging paradigm that offers a direct solution: protocols are written in a “choreographic” programming language, and then automatically compiled to correct implementations by means of an operation known as *Endpoint Projection* (EPP or projection for short) [5, 14, 16, 17, 20, 24, 25].

Choreographic languages are inspired by the Alice and Bob notation of security protocol [29], in the sense that they offer primitives for expressing communications between different processes. Implementations are usually modelled in terms of a process calculus. Besides being simple, choreographic programming is interesting because it typically includes strong theoretical guarantees, most notably deadlock-freedom and an operational correspondence between choreographies and the (models of the) generated distributed implementations.

Not all choreographies can be compiled (or *projected*) to a distributed implementation due to a problem known as “knowledge of choice” [6]. Consider the following choreography



© Luís Cruz-Filipe and Fabrizio Montesi;

licensed under Creative Commons License CC-BY 4.0

14th International Conference on Interactive Theorem Proving (ITP 2023).

Editors: Adam Naumowicz and René Thiemann; Article No. 26; pp. 26:1–26:19



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

for a simple purchase scenario (this example also anticipates some of our syntax).

```
buyer.offer → seller.x;
If seller.acceptable(x) Then seller.product → buyer.y; End
Else End
```

■ **Listing 1** An unprojectable choreography.

This choreography reads: a **buyer** communicates their offer for the purchase of a product to a **seller**, who stores the offer in their local variable **x**; the **seller** then checks whether the offer is acceptable, and in the affirmative case sends the **product** to **buyer**. This choreography cannot be projected to a behaviourally-equivalent implementation, because **buyer** has to behave differently in the two branches of the conditional. However, this conditional is evaluated by **seller**, and **buyer** has no way of discerning which branch gets chosen.

Choreographies are typically made projectable by adding selections, i.e., communications of constants called *selection labels*.¹ A projectable version of Listing 1 looks as follows.

```
buyer.offer → seller.x;
If seller.acceptable(x) Then seller → buyer[left]; seller.product → buyer.y; End
Else seller → buyer[right]; End
```

■ **Listing 2** A projectable choreography.

This choreography differs from the previous one by the presence of a selection in each branch of the conditional. Specifically, if **seller** chooses the **Then** branch, they now communicate the label **left** to **buyer**. Otherwise, if the **Else** branch is chosen, the label **right** is communicated instead. The key idea is that now the implementation generated for **buyer** can read the label received by **seller** and know which branch of the conditional should be executed. Since labels are constants, compilation can statically verify that **buyer** receives different labels for the different branches, and therefore has “knowledge of choice”.

Projection can be smart about knowledge of choice, allowing selections to be kept to a minimum [3]. A process only needs to know which branch of a conditional has been chosen if its behaviour depends on that choice; if the process has to perform the same actions in both branches of a conditional, then this knowledge is irrelevant to it. Knowledge of choice can also be propagated: if a process **q** knows of a choice performed by another process **p**, then either process can forward this information to any other process that needs it.

Amendment. Previous work investigated how unprojectable choreographies can be automatically transformed into projectable ones. Such a transformation is called *amendment* [10, 22] or *repair* [1, 15]. For example, applying the amendment procedure from [10] to the choreography in Listing 1 returns the choreography in Listing 2 (up to minor differences in notation).

Amendment is interesting for (at least) two reasons. On a practical level, it can suggest valid selection strategies to developers to make their choreographies executable—or even do it automatically, so that they do not have to worry about knowledge of choice. On a theoretical level, it allows porting completeness properties of the set of all choreographies to the set of projectable choreographies.

An example of the latter occurs in the study of *Core Choreographies* (CC), a minimalistic theory of choreographic programming [10], where we showed that the set of projectable

¹ Selections are essentially the choreographic version of branch selections in session types, or the additive connectives in linear logic [4, 18].

choreographies in CC is Turing-complete in two steps. First, we showed that CC is Turing-complete, ignoring the question of projectability (the choreographies constructed in the proof are clearly not projectable); then, we defined an amendment procedure and prove an operational correspondence between choreographies and their amendments. As a consequence, the subset of projectable choreographies is also Turing-complete. A similar argument, using the operational correspondence result between projectable choreographies and their implementations, shows that the process calculus used (*Stateful Processes*, or SP) is Turing-complete.

The problem. Our original objective was to formalise amendment and its properties from [10] in the Coq theorem prover, building upon our previous formalisation of CC [12] and its accompanying notion of projection [11]. That formalisation uses a variation of CC based on the theory from [28], which we found more amenable to formalisation. Unfortunately, after formalising the definition of amendment, our attempt to prove its operational correspondence result failed. An inspection of the state of the failed proof quickly led us to a counterexample.

The incorrectness of the original statement jeopardises the subsequent developments that rely on it, in particular Turing completeness of the set of projectable choreographies and of SP. These results were instrumental in substantiating the claim that CC is a “good” minimalistic model for choreographic programming. This finding pointed us towards a more ambitious goal: reformulate the operational correspondence for amendment such that it is correct, and still powerful enough to obtain the aforementioned consequences.

Our motivation to formalise our results in Coq is in line with the increasing awareness in the community that these types of bugs are not uncommon in concurrency theory [26]. In particular, the authors of [15] identified a number of incorrect results in previous work on choreographies, which affect part of the work on choreography repair from [1].

Contribution. To the best of our knowledge, this is the first time that choreography amendment has been formalised. We state and prove a relaxed version of the operational correspondence between choreographies and their amendments in the Coq theorem prover, thus increasing confidence in its correctness. We discuss how working with an interactive theorem prover was instrumental to identifying counterexamples that guided us towards this new, correct formulation that considers all corner cases. We then use our result to formalise the proofs of Turing completeness of projectable choreographies and SP from [10], which were not included in [12].

Structure of the paper. We present the relevant background on CC and its formalisation in Section 2. Section 3 presents the definition of amendment, its formalisation, and discusses and corrects the operational correspondence result from [10]. Section 4 shows that the revised semantic property is still strong enough to derive the Turing completeness results in that work. We discuss related work in Section 5 and conclude in Section 6.

Our exposition assumes some familiarity with interactive theorem proving. We include some Coq code in the article, but the work is intended to be accessible to non-Coq experts.

2 Background

We summarise the latest version of the Coq formalisation of CC [13]. For simplicity, we omit two ingredients that are immaterial for our work: the fact that the language is parameterised on a signature, and the fact that communications have annotations (these are meant to

include information relevant for future implementations in actual programming languages). This allows us to omit some subterms that play no role in the development of amendment.

In our presentation, we use Coq notation with some simplifications for enhanced readability: choreography and process terms are written by overloading the dot symbol (this is not allowed by the Coq notation mechanism), and inductive definitions and inference rules are given with the usual mathematical notation.

2.1 Core Choreographies

We start by giving an overview of Core Choreographies (CC) together with its formalisation in Coq [12].

Syntax. The syntax of CC is given by the following grammar.

```
C ::=  $\eta$ ; C | If p.b Then C1 Else C2 | Call X | RT_Call X ps C | End
 $\eta$  ::= p.e  $\longrightarrow$  q.x | p  $\longrightarrow$  q[l]
```

A choreography C can be either: a communication η followed by a continuation ($\eta; C$); a conditional **If** $p.b$ **Then** $C1$ **Else** $C2$, where the process p evaluates the boolean expression b to choose between the branches $C1$ and $C2$; a procedure call **Call** X , where X is the name of the procedure being invoked; a runtime term **RT_Call** X ps C ;² or the terminated choreography **End**. A communication η can be: a value communication $p.e \longrightarrow q.x$, read “process p evaluates expression e locally and sends the result to process q , which stores it in its local variable x ”; or a selection $p \longrightarrow q[l]$, where the label l can be either **left** or **right**, read “ p sends label l to q ”.

Choreographies are formalised in Coq as an inductive type called **Choreography**. Table 1 summarises the Coq types used in this paper and our conventions for ranging over their elements.

Executing a choreography requires knowing the definitions of the choreographies associated to the procedures that can be invoked, as well as the processes involved in those procedures. A set of procedure definitions is defined as a mapping from procedure names to pairs of process names and choreographies.

Definition $\text{DefSet} := \text{RecVar} \rightarrow (\text{list Pid}) * \text{Choreography}$.

For simplicity, this is defined as a total function – any procedure that is not used in the choreography can simply be mapped to $([p], \text{End})$ for some process p . (For technical reasons, the set of process names is not allowed to be empty.)

A *choreographic program* is then a pair consisting of a set of procedure definitions and a choreography (which represents the “main” or “running” choreography).

Definition $\text{Program} := \text{DefSet} * \text{Choreography}$.

We write **Procedures** P and **Main** P for the two components of P . The set of all processes used by a program P is defined as $\text{CCP_pn } P$.

It is standard practice to assume some well-formedness conditions about choreographies, e.g., that no process communicates with itself. Choreographic programs have additional well-formedness conditions that must hold for all procedures that can be reached at runtime.

² Runtime terms are needed for technical reasons in the definition of the semantics of choreographies [12]. These aspects are irrelevant for the present development.

Type	Variable	Description
Choreography	C	Choreographies
Pid	p, q, r, s	Process names (identifiers)
list Pid	ps	List of process names
Var	x, y, z	Variable names
Val	v	Values
Expr	e	Expressions (evaluate to values)
BExpr	b	Boolean expressions (evaluate to Booleans)
Label	l	Labels (left and right)
RecVar	X	Procedure names (or recursive variables)
DefSet	D	Sets of procedure definitions in CC
State	s	Maps from variables to values
Configuration	c	Choreographic programs equipped with states
TransitionLabel	t	Transition labels
list TransitionLabel	tl	Lists of transition labels
Behaviour	B	Behaviours
option Behaviour	mB	option monad for behaviours
Network	N	Networks
DefSetB	D	Sets of procedure definitions in SP
Program	P	Choreographies/networks with procedure definitions

■ **Table 1** Summary of types in the original Coq formalisation [12, 11].

This notion is not decidable in general, but it becomes so in the practical case of programs that only use a finite number of procedures. We return to this aspect at the end of Section 3.2, where it becomes relevant.

► **Example 1.** The choreographies in Listings 1 and 2 are well-formed.

Semantics. The intuitive system assumptions in CC are that: processes run independently of each other (concurrently) and possess local stores (associating their variables to values); communications are synchronous; and the network is reliable (messages are not lost nor duplicated, and they are delivered in the right order between any two processes). These assumptions are imported from process calculi, where they are quite standard.

► **Example 2.** Since processes run concurrently, it is possible to express choreographies with concurrent behaviour. Consider the following simplification of the factory example in [28].

```
o.order → p.x; o'.order' → p'.y; End
```

■ **Listing 3** Parallel orders.

In Listing 3, two processes *o* and *o'* place their respective orders to two different providers *p* and *p'*. Since all processes are distinct and there is no causal dependency between the two communications, the two communications can in principle be executed in any order. This gives rise to a notion of out-of-order execution for choreographies.

The semantics of choreographies in [12] is given as a labelled transition system on configurations, which consist of a program and a (memory) state. States associate to each process a map from variable names to values, which defines the memory of that process.

```
Definition State := Pid → Var → Value
```

$$\begin{array}{c}
\frac{v := \text{eval } e \text{ s } p \quad s' \llbracket q, x \Rightarrow v \rrbracket}{(D, p.e \rightarrow q.x; C, s) \xrightarrow{[\text{TL_Com } p \ v \ q]} (D, C, s')} \text{CC_Com} \\
\\
\frac{s \llbracket == \rrbracket s'}{(D, p \rightarrow q[1]; C, s) \xrightarrow{[\text{TL_Sel } p \ q \ 1]} (D, C, s')} \text{CC_Sel} \\
\\
\frac{\text{beval } b \text{ s } p = \text{true} \quad s \llbracket == \rrbracket s'}{(D, \text{If } p.b \text{ Then } C1 \text{ Else } C2, s) \xrightarrow{[\text{TL_Tau } p]} (D, C1, s')} \text{CC_Then} \\
\\
\frac{\text{disjoint_eta_rl } \eta \ t \quad (D, C, s) \xrightarrow{[t]} (D, C', s')}{(D, \eta; C, s) \xrightarrow{[t]} (D, \eta; C', s')} \text{CC_Delay_Eta} \\
\\
\frac{\text{disjoint_p_rl } p \ t \quad (D, C1, s) \xrightarrow{[t]} (D, C1', s') \quad (D, C2, s) \xrightarrow{[t]} (D, C2', s')}{(D, \text{If } p.b \text{ Then } C1 \text{ Else } C2, s) \xrightarrow{[t]} (D, \text{If } p.b \text{ Then } C1' \text{ Else } C2', s')} \text{CC_Delay_Cond}
\end{array}$$

■ **Figure 1** Semantics of choreographic configurations (selected rules).

States come with some notation: $s \llbracket == \rrbracket s'$ says that s and s' are extensionally equal, and $s \llbracket p, x \Rightarrow v \rrbracket$ is the state obtained from updating s with the mapping $p, x \mapsto v$.

With these concepts in place, we can show some representative transition rules for choreographic configurations in Figure 1.³ Transitions have the form $(D, C, s) \xrightarrow{[t]} (D, C', s')$, where t is a transition label that allows for observing what happened in the transition.

Rule **CC_Com** deals with the execution of a value communication from a process p to a process q : if the expression e at p can be evaluated to a value v (first condition, which uses the auxiliary function **eval**), then the communication term is consumed and the state of the receiver is updated such that its receiving variable x is now mapped to value v . The transition label $\text{TL_Com } p \ v \ q$ denotes that p has communicated the value v to q , modelling what would be visible on a network.

Rule **CC_Sel** is similar but does not alter the state of the receiver (the role of selections will be clearer when we explain the language for modelling implementations of choreographies). The transition label $\text{TL_Sel } p \ q \ 1$ registers the communication of label 1 from p to q .

Rule **CC_Then** deals with the case in which a process p can evaluate the guard b of a conditional to **true** (using the auxiliary function **beval**), proceeding to the then-branch of the conditional. The transition label $\text{TL_Tau } p$ denotes that process p has executed an internal action (τ is the standard symbol for such actions in process calculi).

Rule **CC_Delay_Eta** deals with out-of-order execution of communications, formalising the reasoning anticipated in Example 2. Specifically, the continuation of an interaction η is allowed to perform a transition (without affecting η) as long as the transition does not involve any of the processes in η . The latter condition is checked by the first premise of the rule, **disjoint_eta_rl** $\eta \ t$, which checks that the processes mentioned in η are distinct from those mentioned by the transition label t . Rule **CC_Delay_Cond** applies the same reasoning to the out-of-order execution of conditionals.

The reflexive and transitive closure of the transition relation is written $\xrightarrow{[t1]}^*$, where $t1$ is a list of transition labels.

► **Example 3.** For any D and s such that **order** evaluates to v at o and **order'** evaluates to v'

³ In the actual formalisation, the transition relation was defined in two layers for technical reasons. This technicality is immaterial for our development, since our results follow from the rules shown here.

at o' , according to `eval`,

$$(D, o.order \rightarrow p.x; o'.order' \rightarrow p'.y; \text{End}, s) \xrightarrow{[TL_Com\ o\ v\ p; TL_Com\ o'\ v'\ p']}^* (D, \text{End}, s')$$

and

$$(D, o.order \rightarrow p.x; o'.order' \rightarrow p'.y; \text{End}, s) \xrightarrow{[TL_Com\ o'\ v'\ p'; TL_Com\ o\ v\ s]}^* (D, \text{End}, s')$$

where $s' \llbracket == \rrbracket s \llbracket s1, x \Rightarrow v \rrbracket \llbracket s2, x \Rightarrow v' \rrbracket$.

2.2 Processes

Implementations of choreographies are modelled in Stateful Processes (SP) [11], a formalised process calculus following [28]. SP follows the standard way of representing systems of communicating processes, where the code of each process is given separately and communication is achieved when processes perform compatible I/O actions.

Syntax. The code of a process is written as a behaviour (B), following the grammar below.

```
B ::= p!e; B | p?x; B | p+1; B | p & mB1 // mB2 | If b Then B1 Else B2 | Call X | End
mB ::= None | Some B
```

These terms are the local counterparts to the choreographic terms of CC. The first two productions deal with value communication. Specifically, a send action $p!e; B$ sends the result of evaluating e to the process p and then continues as B . Dually, a receive action $p?x; B$ receives a value from p , stores it in x , and then continues as B .

Selections are implemented by the primitives $p+1; B$ and $p \& mB1 // mB2$. The former sends the label 1 to the process p and continues as B . The latter is a branching term, where $mB1$ and $mB2$ are the behaviours that the process will execute upon receiving **left** or **right**, respectively. To cover the case where a process does not offer a behaviour for a specific label, $mB1$ and $mB2$ have type **option** Behaviour.

Conditionals (**If** b **Then** $B1$ **Else** $B2$), procedure calls (**Call** X), and the terminated behaviour (**End**) are standard.

Processes are intended to run together in networks. These are formalised as maps from processes to behaviours.

Definition $\text{Network} := \text{Pid} \rightarrow \text{Behaviour}$.

Networks come with some convenient notation for their construction: $p[B]$ is the network that maps p to B and all other processes to **End**; and $N \mid N'$ is the composition of N and N' . In particular, $(N \mid N')\ p$ returns $N\ p$ if this is different from **End**, and $N'\ p$ otherwise.⁴

► **Example 4.** The following network implements the choreography in Listing 2.

```
buyer[ seller!offer; seller & Some (seller?y; End) // Some End ] |
seller[ buyer?x; If acceptable(x) Then buyer+left; buyer!product; End
      Else buyer+right; End ]
```

⁴ This asymmetry does not matter for our results, since we never compose networks that define nonterminated behaviours for the same processes.

$$\begin{array}{c}
\frac{
\begin{array}{l}
N \ p = q!e; B \\
N \ q = p?x; B'
\end{array}
\quad
v := \text{eval } e \ s \ p
\quad
\begin{array}{l}
N' \ (==) N \setminus p \setminus q \mid p[B] \mid q[B'] \\
s' \ [==] s[q, x \Rightarrow v]
\end{array}
}{
(D, N, s) \xrightarrow{[TL_Com \ p \ v \ q]} (D, N', s')
} \text{ SP_Com}
\\[1.5cm]
\frac{
\begin{array}{l}
N \ p = q+\text{left}; B \\
N \ q = p \ \& \ \text{Some } B1 \ // \ mBr
\end{array}
\quad
\begin{array}{l}
N' \ (==) N \setminus p \setminus q \mid p[B] \mid q[B1] \\
s \ [==] s'
\end{array}
}{
(D, N, s) \xrightarrow{[TL_Sel \ p \ q \ \text{left}]} (D, N', s')
} \text{ SP_LSel}
\\[1.5cm]
\frac{
\begin{array}{l}
N \ p = \text{If } b \ \text{Then } B1 \ \text{Else } B2 \\
\text{beval } b \ s \ p = \text{true}
\end{array}
\quad
\begin{array}{l}
N' \ (==) N \setminus p \mid p[B1] \\
s \ [==] s'
\end{array}
}{
(D, N, s) \xrightarrow{[TL_Tau \ p]} (D, N', s')
} \text{ SP_Then}
\end{array}$$

■ **Figure 2** Semantics of network configurations (selected rules).

For the semantics of networks, we need two additional ingredients. The network $N \setminus p$ is obtained from N by redefining p 's behaviour as **End** (p is “removed” from N). The relation $N \ (==) N'$ holds if the networks N and N' are extensionally equal.

As in CC, processes in a network can invoke procedures defined in a separate set.

Definition $\text{DefSetB} := \text{RecVar} \rightarrow \text{Behaviour}$.

A **Program** in SP consists of a set of procedure definitions and a network. Assuming that this set is globally accessible simplifies the definition of the semantics of SP; in implementations, each process would have a local copy of this set, or of the subset of procedures it needs to invoke.

Definition $\text{Program} := \text{DefSetB} * \text{Network}$.

We use D to range over elements of DefSetB and P to range over elements of **Program**, as for choreographies (the difference will be clear from the context).

Semantics. The semantics of SP is also given as a labelled transition system on configurations that consist of a program and a memory state, as in CC. A selection of the transition rules defining this semantics is displayed in Figure 2.

Rule **SP_Com** matches a send action at a process p with a compatible receive action at another process q (conditions $N \ p = q!e; B$ and $N \ q = p?x; B'$). The resulting network N' is obtained from N by replacing the behaviours of these processes with their continuations ($N \setminus p \setminus q \mid p[B] \mid q[B']$). The update to the state is handled as in CC.

Rules **SP_LSel** and its dual **SP_RSel** model, respectively, the selection of the left and right branches offered by a branching term, by inspecting the label sent by the sender. Rule **SP_Then** captures the case in which a conditional enters its then-branch.

2.3 Endpoint Projection (EPP)

Choreographies are compiled to networks by a procedure defined in two layers. We begin by defining a *behaviour projection*, which compiles the desired behaviour from a choreography for a given process. This procedure is a partial function, and since all functions in Coq are total it was formalised as the following inductive relation.

$$\begin{array}{c}
\frac{\llbracket D, C \mid p \rrbracket == B}{\llbracket D, p \longrightarrow q[1]; C \mid p \rrbracket == q+1; B} \text{ bproj_Pick} \quad \frac{p \neq r \quad q \neq r \quad \llbracket D, C \mid r \rrbracket == B}{\llbracket D, p \longrightarrow q[1]; C \mid r \rrbracket == B} \text{ bproj_Sel} \\
\\
\frac{p \neq q \quad \llbracket D, C \mid q \rrbracket == B}{\llbracket D, p \longrightarrow q[\text{left}]; C \mid q \rrbracket == p \ \& \ \text{Some } B \ // \ \text{None}} \text{ bproj_Left} \\
\\
\frac{\llbracket D, C1 \mid p \rrbracket == B1 \quad \llbracket D, C2 \mid p \rrbracket == B2}{\llbracket D, \text{If } p.b \text{ Then } C1 \text{ Else } C2 \mid p \rrbracket == \text{If } b \text{ Then } B1 \text{ Else } B2} \text{ bproj_Cond} \\
\\
\frac{p \neq r \quad \llbracket D, C1 \mid p \rrbracket == B1 \quad \llbracket D, C2 \mid p \rrbracket == B2 \quad B1 \ [V] \ B2 == B}{\llbracket D, \text{If } p.b \text{ Then } C1 \text{ Else } C2 \mid p \rrbracket == B} \text{ bproj_Cond'}
\end{array}$$

■ **Figure 3** Selected rules for behaviour projection.

bproj : DefSet → Choreography → Pid → Behaviour → Prop

Term **bproj** $D \ C \ p \ B$, written $\llbracket D, C \mid p \rrbracket == B$, reads “the projection of C on p in the context of the set of procedure definitions D is B ”.⁵

Intuitively, behaviour projection is computed by going through the choreography; for each choreographic term, projection constructs the local action that the input process should perform to implement it. The rules defining **bproj** that are relevant for this work are those that deal with selections and conditionals. These are shown in Figure 3.

A label selection $p \longrightarrow q[1]$ is projected as either: (i) the sending of label 1 to q for process p (rule **bproj_Pick**); (ii) the appropriate branching term that receives 1 from p for process q , where only the branch for 1 offers a behaviour (rules **bproj_Left** and the dual rule **bproj_Right**); or (iii) no action for any other process (rule **bproj_Sel**).

Similarly, a conditional in a choreography is projected to a conditional for the process that evaluates the guard (rule **bproj_Cond**). However, projecting conditionals becomes complex when considering the other processes, because this requires dealing with the problem of knowledge of choice discussed in Section 1. This case is handled by rule **bproj_Cond'**, which sets the result of projection to be the “merging” of the projections of the two branches, written $B1 \ [V] \ B2 == B$, if this is defined.

Intuitively, merging attempts to build a behaviour B from two behaviours $B1$ and $B2$ that have similar structures, but may differ in the labels that they accept in branching terms. For all terms but branchings, merging requires term equality and then proceeds homomorphically in subterms. This is exemplified by the rules **merge_End**, **merge_Sel**, and **merge_Cond** in Figure 4.

The interesting part regards the merging of branching terms, which has a rule for every possible combination. Figure 4 shows two representative cases. If two branching terms have branches for different labels, then we obtain a branching term where the two branches are combined as exemplified by rule **merge_Branching_SNNS**. If two branching terms have overlapping branches, then we try to merge them as exemplified by rule **merge_Branching_SSSS**.⁶

As we remarked, merging (seen as a partial function) can be undefined, for example **End** and $p+1$; **End** cannot be merged. This gives rise to the notion of *projectability* anticipated in

⁵ The parameter D is used for projecting procedure calls, which is immaterial to the current work.

⁶ Due to space constraints, the names of these rules have been abbreviated in Figure 4.

$$\begin{array}{c}
\frac{}{\text{End } [V] \text{ End} == \text{End}} \text{merge_End} \quad \frac{B1 [V] \ B2 == B}{p+1; B1 [V] \ p+1; B2 == p+1; B} \text{merge_Sel} \\
\\
\frac{Bt1 [V] \ Bt2 == Bt \quad Be1 [V] \ Be2 == Be}{\text{If } p.e \text{ Then } Bt1 \text{ Else } Bt2 [V] \ \text{If } p.e \text{ Then } Be1 \text{ Else } Be2 == \text{If } p \text{ Then } Bt \text{ Else } Be} \text{merge_Cond} \\
\\
\frac{}{p \ \& \ \text{Some } bL \ // \ \text{None } [V] \ p \ \& \ \text{None} \ // \ \text{Some } bR == p \ \& \ \text{Some } bL \ // \ \text{Some } bR} \text{SNNS} \\
\\
\frac{bL1 [V] \ bL2 == bL \quad bR1 [V] \ bR2 == bR}{p \ \& \ \text{Some } bL1 \ // \ \text{Some } bR1 [V] \ p \ \& \ \text{Some } bL2 \ // \ \text{Some } bR2 == p \ \& \ \text{Some } bL \ // \ \text{Some } bR} \text{SSSS}
\end{array}$$

■ **Figure 4** Definition of the merge relation (selected rules).

Section 1: a choreography C is projectable on a process p in the context of a set of procedure definitions D if bproj is defined for those parameters.

Definition $\text{projectable_B } D \ C \ p := \exists B, [D, C \mid p] == B$.

This is generalised by $\text{projectable_C } D \ C \ ps$, which states that C is projectable for all processes in the list ps . For a choreographic program P to be projectable, written $\text{projectable_P } P$, we require that $\text{Main } P$ be projectable for all processes in $\text{CCP_pn } P$ and that all procedures be projectable for the processes that they use.

With projectability in place, Endpoint Projection (EPP) is defined as a function that maps a projectable choreographic program to a process program in SP.

Definition $\text{epp } P : \text{projectable_P } P \rightarrow \text{Program}$.

The second argument of epp is a proof of $\text{projectable_P } P$, but the formalisation includes a lemma showing that the result does not depend on this term.

► **Example 5.** The behaviours of `buyer` and `seller` in Example 4 are the respective projections for these two processes of the choreography in Listing 2.

The definition of epp allows us to apply program extraction and obtain a certified compiler from choreographies to networks – see [8] for a discussion and examples.

2.4 Turing completeness

The authors of [12] formalise that CC is Turing-complete, in the sense that all of Kleene’s partial recursive functions [21] can be implemented as a choreography for a suitable notion of implementation. The proof is interesting because it considers CC instantiated with very restricted computational capabilities at processes: values are natural numbers; expressions can only be the constant zero, a variable, or the successor of a variable; and Boolean expressions can only check if the two variables at a process contain the same value. Kleene’s partial recursive functions are then implemented concurrently, by making processes communicate according to appropriate patterns.

According to [12], a choreographic program P implements $f : \text{PRFunction } m$ (representing a partial recursive function $f : \mathbb{N}^m \rightarrow \mathbb{N}$) with input processes ps_1, \dots, ps_m and output process q iff: for any state s where ps_1, \dots, ps_m contain the values n_1, \dots, n_m in their variable x , (i) if $f(n_1, \dots, n_m) = n$, then all executions of P from s terminate, and do so in a state where q

stores n in its variable x ; and (ii) if $f(n_1, \dots, n_m)$ is undefined, then execution of P from s never terminates.⁷ This is captured by the Coq term `implements P m f ps q`, where ps is the vector ps_1, \dots, ps_m .

The proof of Turing completeness encodes partial recursive functions to choreographies that are not always projectable, since they contain no selections but some processes behave differently in conditionals.

3 Amendment

Several works have studied how unprojectable choreographies can be automatically amended to obtain projectable versions [1, 10, 22]. In particular, [10] developed an amendment procedure based on merging. The informal idea that we explore below is that, whenever a choreography contains a conditional, amendment adds selections, in both branches, from the process evaluating the guard to any processes whose behaviour projection is undefined. Intuitively, this makes the output choreography projectable.

► **Example 6.** Let C be the choreography:

```
p.e → q.x; If r.b Then (r.e' → p.y; End) Else End
```

Amending C as described yields the following choreography, A :

```
p.e → q.x; If r.b Then (r → p[l]; r.e' → p.y; End)
                  Else (r → p[r]; End)
```

Amendment is claimed to have the following properties.

► **Lemma 7** (Amendment Lemma [10], rephrased). *For every choreography C :*

1. *The amendment of C is well-formed.*
2. *The amendment of C is projectable.*
3. *If DA , A , and A' are obtained by amending all procedures in D as well as C and C' , then $(D, C, s) \xrightarrow{[tl]}^* (D, C', s')$ iff $(DA, A, s) \xrightarrow{[tl']}^* (DA, A', s')$ for some tl' .*

In point one, well-formedness refers to a set of syntactic conditions that exclude ill-written choreographies, e.g., self-communications (interactions where a process communicates with itself) [10]. Points one and two are simple to prove by induction on the structure of the choreography. Point three, unfortunately, is wrong. When attempting to formalise this result, we failed, and the state of the proof led us to the following counterexample.

► **Example 8.** Given a suitable state, the choreography C from Example 6 can make a transition to C' defined as

```
p.e → q.x; r.e' → p.y; End
```

by rules `CC_Delay_Eta` and `CC_Then`. However, C 's amendment A can move to

```
p.e → q.x; r → p[l]; r.e' → p.y; End
```

by the same rules, but this is neither the amendment of C' , nor can it reach it since the offending selection term is blocked by the initial communication.

⁷ This is a straightforward adaption of the definition of function implementation by a Turing machine [31].

In hindsight, this is not so surprising: amendment introduces causal dependencies that were not present in the source choreography. However, this intuition was completely missed by both authors and reviewers of the original publications discussing amendment [9, 10]. Therefore, amending a choreography can remove some execution paths.

In the rest of this section, we show how to define amendment formally in Coq, and formulate a correct variation of Lemma 7.

3.1 Definition

We decompose the definition of amendment in three functions: one for identifying the processes that need to be informed of the outcome of a specific conditional; one for prepending a list of selections to a choreography; and one that recursively amends a whole choreography by using the former two. This division simplifies not only the definition, but also the structure of proofs about amendment since they can be modularised.

To identify the processes that require knowledge of choice, we define a function `up_list` (`up` is short for “unprojectable processes”). This function recursively goes through a list `ps` of processes and checks for each process in the list whether the choreography `If p.b Then C1 Else C2` can be projected on that process (function `projectable_B_dec` does precisely this test). If this is not the case, then the process is added to the result. (Since projectability is relative to a set of procedure definitions, this also needs to be given as an argument, `D`.)

```
Fixpoint up_list D p b ps C1 C2 : list Pid := match ps with
| nil => nil
| r :: ps' => let ps'' := up_list D p b ps' C1 C2 in
  if (r =? p) then ps''
  else if projectable_B_dec D (If p.b Then C1 Else C2) r
    then ps''
    else (r :: ps'') end.
```

Note that `p`, as the evaluator of the conditional, does not need to be informed of the outcome. This justifies the check `r =? p`, whose inclusion also avoids introducing self-communications and simplifies subsequent proofs. (Function `up_list` is essentially a filter, but since it tests two predicates we found the current definition to be easier to work with than either a composition of two filters or a filter with a predicate defined as a conjunction.)

The second ingredient is straightforward: given a process `p`, a selection label `l`, and a choreography `C`, it recursively adds selections of `l` from `p` to each element of a list `ps`.

```
Fixpoint add_sels p l ps C : Choreography := match ps with
| nil => C
| r :: ps' => p -> r[l]; add_sels p l ps' C end.
```

We can now define amendment following the informal procedure described in [10]. Given a list of processes `ps`, we go through a choreography `C`; whenever we meet a conditional on a process `p`, we compute the list of processes from `ps` with an undefined projection and prepend the branches of the conditional with appropriate selections. (We show only the most interesting cases.)

```
Fixpoint amend D ps C := match C with
| eta; C' => eta; (amend D ps C')
| If p.b Then C1 Else C2 =>
  let l := up_list D p b ps (amend D ps C1) (amend D ps C2) in
  If p.b Then (add_sels p left l (amend D ps C1))
  Else (add_sels p right l (amend D ps C2))
```

```
| ... end.
```

Amendment is generalised to sets of procedure definitions in the obvious way.

Definition `amend_D` $D \text{ ps} : \text{DefSet} := \text{fun } X \Rightarrow (\text{fst } (D \ X), \text{amend } D \ \text{ps} \ (\text{snd } (D \ X)))$.

To amend a program P , the parameter ps of the previous functions is instantiated with the set of processes used in P .

Definition `amend_P` $P := (\text{amend_D } (\text{Procedures } P) \ (\text{CCP_pn } P), \text{amend } (\text{Procedures } P) \ (\text{CCP_pn } P) \ (\text{Main } P))$.

This formal definition corresponds to the informal one given in [10]. In particular, all our examples are formalised in Coq.

► **Example 9.** Consider the following choreography.

```
If p.b Then (p.e → q.x; q.e' → r.y; End)
Else (q.e'' → r.y; End)
```

Here, p decides if (i) it will communicate a value to q that can be used in the computation of a later message from q to r (so q acts as a sort of proxy) or (ii) q should just compute the value that it will communicate to r by itself. Amendment is smart enough to notice that while q requires a selection from p , r does not since it behaves in the same way (receive from q on x). Therefore, amending the choreography returns the following.

```
If p.b Then (p → q[left]; p.e → q.x; q.e' → r.y; End)
Else (p → q[right]; q.e'' → r.y; End)
```

3.2 Syntactic Properties

We now discuss the key properties of amendment.

Amendment preserves well-formedness of choreographies (`Choreography_WF`) and choreographic programs (`Program_WF`). This follows from the fact that `add_sels` preserves all syntactic properties of well-formedness, using induction.

Lemma `amend_Choreography_WF` : `Choreography_WF C → Choreography_WF (amend D ps C)`.

Lemma `amend_Program_WF` : `Program_WF (D,C) → Program_WF (amend_D D ps, amend D ps C)`.

(For simplicity, we omit universal quantifiers at the beginning of lemmas.)

Likewise, it is straightforward to prove that amending for some processes guarantees that the output choreography is projectable on all those processes.

Lemma `amend_projectable_C` : `projectable_C (amend_D D ps) (amend D ps C) ps`.

We do not generalise this result to choreographic programs: it is not straightforward to do and our later development does not need it. The issue we encounter is related to a problem discussed in [12, 11]: computing the set of processes and procedures that are used by a choreography can require an infinite number of steps, and is therefore not definable as a function in Coq. (A simple example is a program with an infinite set of procedure definitions where each procedure X_i invokes the next procedure X_{i+1} .)

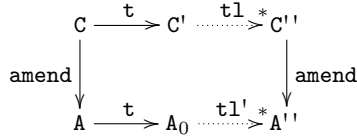
The function `CCP_pn` used in the definition of `amend_P` does return the set of processes involved in a program P , but it does not check that P does not define unused procedures. If this is the case, these procedures may use processes not in `CCP_pn P`, and therefore they may be

unprojectable for these processes. Rather than stating a result with complex side-conditions as hypotheses, we prove projectability of particular programs applying `amend_projectable_C` to `Main P` and to the bodies of all procedure definitions. The development in the next section uses this strategy.

3.3 Semantic Properties

We now discuss how the formulation of the semantic relation between a choreography and its amendment needs to be changed.

The counterexample shown earlier suggests allowing both choreographies to perform additional transitions in order to unblock and remove lingering selections introduced by amendment. (In our example, this would be the communication from `p` to `q`.) The correspondence would then look as follows, where the dotted lines correspond to existentially quantified terms:



and the list of transition labels `tl` can be obtained from `tl'` by removing some selections.

Our attempt to prove this result showed that it holds for all cases but one: when the transition `t` is obtained by applying rule `CC_Delay_Cond`.

► **Example 10.** We show a minimal counterexample. Consider the choreography

```
If p.b Then (q.e → r.x; q.e' → p.x; End)
Else (q.e → r.x; End)
```

and its amendment

```
If p.b Then (p → q[left]; q.e → r.x; q.e' → p.x; End)
Else (p → q[right]; q.e → r.x; End) .
```

The original choreography can execute the communication between `q` and `r`, reaching

```
If p.b Then (q.e' → p.x; End) Else End
```

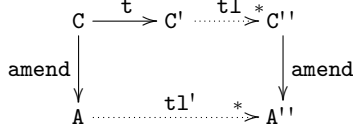
but its amendment needs to run the conditional and a selection before it can execute the same communication.

There are two ways to solve this problem: changing the definition of amendment, or refining the correspondence result further. We opted for the second route, for two reasons: first, we get to keep the original definition given on paper in [10]; second, making amendment clever enough to recognise this kind of situations requires a non-local analysis of the choreography (i.e., looking at the structure of the branches of conditionals instead of simply checking for projectability of the term). In our example, such an analysis could detect that the additional selections from `p` to `q` could be added only after the communication from `q` to `r`, solving the issue.

Therefore, our final correspondence result requires that the amendment of a choreography be allowed to perform additional transitions *before* it matches the transition performed by the original choreography. Since a transition may invoke rule `c_delay_Cond` more than once, this means that the orders of the transitions performed by the original choreography and its

amendment can be arbitrary permutations of each other that respect causal dependencies between transitions (ignoring the extra selections).

The correspondence result we prove looks as follows:



where $t :: t1$ can be obtained from $t1'$ by removing some selections and permuting labels.

To formalise this in Coq, we introduce a relation `sel_exp` (“selection expansion”) between lists of transition labels.

```

Inductive sel_exp :=
| se_base t1 t1' : Permutation t1 t1' → sel_exp t1 t1'
| se_extra p q l t1 t1' t1'' : sel_exp t1 t1' →
  Permutation (TL_Sel p q l :: t1') t1'' → sel_exp t1 t1''.

```

We can now prove a correct version of the correspondence between choreographies and their amendments. There are four results in total: the one depicted above and its generalisation to the case where t is replaced with a list of transition labels; and the two dual results where the amendment of a choreography moves first. We show the two more general statements.

```

Lemma amend_complete_many : Program_WF (D,C) → (D,C,s) → [t1]→* (D,C',s') →
  ∃ t1' t1'' C'' s'', sel_exp (t1++ t1') t1'' ∧ (D,C',s') → [t1']→* (D,C'',s'')
  ∧ (amend_D D ps, amend D ps C,s) → [t1'']→* (amend_D D ps, amend D ps C'',s'').

```

```

Lemma amend_sound_many : Program_WF (D,C) → let (D' := amend_D D ps) in
  (D', amend D ps C,s) → [t1]→* (D',C',s') →
  ∃ t1' t1'' C'' s'', (D',C',s') → [t1']→* (D', amend D ps C'',s'')
  ∧ (D,C,s) → [t1'']→* (D,C'',s'') ∧ sel_exp t1' (t1++ t1').

```

The challenging part of the work in this section was understanding what the correct formulation of these results should be. Once we reached this formulation, proofs were relatively straightforward inductions on the given transitions (10–15 lines of Coq code per case).

The formalisation of the amendment lemma consists of 6 definitions, 50 lemmas, and 4 examples, with a total of roughly 1050 lines of Coq code.

4 Implications of Amendment

In the previous section, we had to weaken the original statement for the semantic correspondence guaranteed by amendment that was given in [10]. Since the original statement was used in the proofs of Turing completeness for projectable core choreographies and SP, it is natural to investigate whether our new formulation still yields these results.

For uniformity, we start by reformulating the Turing completeness result for core choreographies from [12], where process names are identified with natural numbers.

```

Theorem CC_Turing_Complete : ∀ n (f:PRFunction n),
  ∃ P, Program_WF P ∧ implements P f (vec_1_to_n n) 0.

```

The theorem states that, for any partial recursive function f , there exists a well-formed choreographic program P that implements f with input processes $1, \dots, n$ and output process 0 . The proof is a straightforward combination of results already presented in [12].

Combining this result with our lemmas about amendment yields that the fragment of projectable core choreographies is also Turing-complete.

```
Lemma projCC_Turing_Complete :  $\forall n (f:PRFunction\ n),$   

 $\exists P, Program\_WF\ P \wedge projectable\_P\ P \wedge implements\ P\ f\ (vec\_1\_to\_n\ n)\ 0.$ 
```

The proof is split into several steps. The most interesting sublemma is the one establishing that amending a choreography that implements a function yields a choreography that implements the same function. This is formulated as a general result about amendment.

```
Lemma amend_implements : Program_WF P  $\rightarrow$   

 $implements\ P\ f\ ps\ q \rightarrow implements\ (amend\_P\ P)\ f\ ps\ q.$ 
```

The proof uses the fact that terminated choreographies cannot execute further to show that the list of additional transitions added to the original choreography by the amendment lemma (`tl` in the last diagram) must be empty.

The remaining lemmas for `projCC_Turing_Complete` deal with projectability of the amended choreography, as discussed in the previous section, and are simple to prove.

Since amended choreographies are projectable, we can further apply the EPP theorem from [11] to show that SP is also Turing-complete.

```
Theorem SP_Turing_Complete :  $\forall n (f:PRFunction\ n),$   

 $\exists P, Network\_WF\ (Net\ P) \wedge SP\_implements\ P\ f\ (vec\_1\_to\_n\ n)\ 0.$ 
```

The definition of `SP_implements` is a straightforward adaptation of the definition of `implements` for choreographies. The proof of `SP_Turing_Complete` follows a similar strategy to the one for `projCC_Turing_Complete`: we prove a sublemma `epp_implements` stating that the EPP of a choreography that implements a function `f` is a process program that implements `f`.

The formalisation of this section consists of 2 definitions and 11 lemmas, totaling about 250 lines of Coq code. The conciseness of this development substantiates our previous comment on not providing a complex lemma for projectability of programs, at the end of Section 3.2.

5 Related Work

To the best of our knowledge, our work is the first formalisation of amendment, its properties, and its intended consequences.

The work nearest to ours is the original presentation of the amendment procedure that inspired us [10]. As we discussed, the behavioural correspondence for amendment that the authors state is wrong. We developed a correct statement and managed to update and formalise the proofs of Turing completeness for CC and SP accordingly. Our formalisation of the behavioural correspondence also clarifies what semantic property amendment actually guarantees, which might be important for future work and practical applications of amendment.

Amendment or similar procedures have been investigated also for other choreographic languages [1, 22]. In all these works, the general idea is to repair choreographies by identifying the specific places where additional communications are required for implementability. However, the differences between the underlying languages and the techniques used make the resulting procedures very different from ours.

The pioneer work of [22] only deals with finite choreographies without out-of-order execution. This allows for an amendment procedure that analyses the syntax of the choreography:

it inspects the choreography and checks that the first communications in the two branches of a conditional have the same sender.

Instead, the choreographies in [1] are automata. Their technique also follows the idea of looking at the possible transitions the choreography can perform in order to repair it, and it uses a notion of realisability inherited from previous work [2] to establish its correctness. Unfortunately, later work [15] showed that Theorem 2 in [2] is incorrect, invalidating the proof of the result that is used in [1].

Differently, our definition of amendment uses merging, first introduced in [3], and projection. While the underlying idea remains the same, this formulation is more intuitive, as the connection between unprojectability and amendment becomes direct. This also simplifies our development and yields shorter proofs.

Our work is based on the most recent version of the formalisation of CC, SP, and EPP [13], which was originally introduced in [11, 12]. We did not need to modify this formalisation in order to use it for our development, which shows that it reached a sufficient level of maturity for being used as a library to reason about choreographies.

Other formalisations of choreographies include: Kalas, a choreographic programming language that targets CakeML [30]; the choreographic DSL Zooid, a Coq library for verifying that message passing code respects a given multiparty session type (these are abstract choreographies without computation) [7]; and multiparty GV, a formalised functional language with a similar goal to Zooid [19].

6 Conclusion

We have presented the first formalisation of an amendment procedure for choreographies. Our work is based on a previous formalisation of CC and its accompanying notion of EPP, which we used as a library. We found this formalisation to be modular and complete enough to support the separate development presented here. In the same spirit of generality and reusability, our formalisation does not add any assumptions about CC that were not present in the library.

Our development is an illustration of how theorem provers can assist in research: interacting with Coq guided us to (i) discovering that the semantic property of amendment found in the background literature for this work is wrong, and (ii) a correct formulation that is still powerful enough for its intended use in previous work.

The formalisation of amendment is amenable to extraction, and therefore our work potentially offers a basis for a certified transformer from arbitrary choreographies in CC to projectable ones. In the future, we plan on studying how this transformer can be integrated into existing frameworks for choreographic programming.

Our notion of amendment is intrinsically related to how EPP is defined for CC. In the literature, there are choreographic languages with a more permissive notion of knowledge of choice, e.g., where replicated processes intended to be used as services are allowed to be involved in only one branch of a conditional [3, 5]. It would be interesting to study how amendment can be adapted to these settings.

References

- 1 Samik Basu and Tevfik Bultan. Automated choreography repair. In Perdita Stevens and Andrzej Wasowski, editors, *Procs. FASE*, volume 9633 of *Lecture Notes in Computer Science*, pages 13–30. Springer, 2016. doi:10.1007/978-3-662-49665-7_2.

- 2 Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In John Field and Michael Hicks, editors, *Procs. POPL*, pages 191–202. ACM, 2012. doi:10.1145/2103656.2103680.
- 3 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012. doi:10.1145/2220365.2220367.
- 4 Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In Josée Desharnais and Radha Jagadeesan, editors, *Procs. CONCUR*, volume 59 of *LIPICs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.
- 5 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *Procs. POPL*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.
- 6 Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party sessions. In Roberto Bruni and Jürgen Dingel, editors, *Procs. FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2011. doi:10.1007/978-3-642-21461-5_1.
- 7 David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In Stephen N. Freund and Eran Yahav, editors, *Procs. PLDI*, pages 237–251. ACM, 2021. doi:10.1145/3453483.3454041.
- 8 Luís Cruz-Filipe, Lovro Lugović, and Fabrizio Montesi. Certified compilation of choreographies with *hacc*. In Marieke Huisman and António Ravara, editors, *Procs. FORTE*, Lecture Notes in Computer Science. Springer, 2023. Accepted for publication.
- 9 Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. In Olga Kouchnarenko and Ramtin Khosravi, editors, *Procs. FACS*, volume 10231 of *Lecture Notes in Computer Science*, pages 17–35. Springer, 2017. doi:10.1007/978-3-319-57666-4_3.
- 10 Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020. doi:10.1016/j.tcs.2019.07.005.
- 11 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Procs. ICTAC*, volume 12819 of *Lecture Notes in Computer Science*, pages 115–133. Springer, 2021. doi:10.1007/978-3-030-85315-0_8.
- 12 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Formalising a Turing-complete choreographic language in Coq. In Liron Cohen and Cezary Kaliszyk, editors, *Procs. ITP*, volume 193 of *LIPICs*, pages 15:1–15:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ITP.2021.15.
- 13 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *Journal of Automated Reasoning*, Accepted for publication.
- 14 Mila Dalla Preda, Maurizio Gabbriellini, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Log. Methods Comput. Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:1)2017.
- 15 Alain Finkel and Étienne Lozes. Synchronizability of communicating finite state machines is not decidable. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *Procs. ICALP*, volume 80 of *LIPICs*, pages 122:1–122:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.ICALP.2017.122.
- 16 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects. *CoRR*, abs/2005.09520, 2020. URL: <https://arxiv.org/abs/2005.09520>.
- 17 Andrew K. Hirsch and Deepak Garg. Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022. doi:10.1145/3498684.
- 18 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016. Also: *POPL*, pages 273–284, 2008. doi:10.1145/2827695.

- 19 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.*, 6(ICFP):466–495, 2022. doi:10.1145/3547638.
- 20 Sung-Shik Jongmans and Petra van den Bos. A predicate transformer for choreographies – computing preconditions in choreographic programming. In Ilya Sergey, editor, *Procs. ESOP*, volume 13240 of *Lecture Notes in Computer Science*, pages 520–547. Springer, 2022. doi:10.1007/978-3-030-99336-8_19.
- 21 Stephen Cole Kleene. *Introduction to Metamathematics*, volume 1. North-Holland Publishing Co., 1952.
- 22 Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Amending choreographies. In António Ravara and Josep Silva, editors, *Procs. WWW*, volume 123 of *EPTCS*, pages 34–48, 2013. doi:10.4204/EPTCS.123.5.
- 23 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In Tom Conte and Yuanyuan Zhou, editors, *Procs. ASPLOS*, pages 517–530. ACM, 2016. doi:10.1145/2872362.2872374.
- 24 Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. Discretionary information flow control for interaction-oriented specifications. In Narciso Martí-Oliet, Peter Csaba Ölveczky, and Carolyn L. Talcott, editors, *Logic, Rewriting, and Concurrency*, volume 9200 of *Lecture Notes in Computer Science*, pages 427–450. Springer, 2015. doi:10.1007/978-3-319-23165-5_20.
- 25 Hugo A. López and Kai Heussen. Choreographing cyber-physical distributed control systems for the energy sector. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Procs. SAC*, pages 437–443. ACM, 2017. doi:10.1145/3019612.3019656.
- 26 Petar Maksimovic and Alan Schmitt. HOCore in Coq. In Christian Urban and Xingyuan Zhang, editors, *Procs. ITP*, volume 9236 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2015. doi:10.1007/978-3-319-22102-1_19.
- 27 Fabrizio Montesi. *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen, 2013. URL: http://www.fabriziomontesi.com/files/choreographic_programming.pdf.
- 28 Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023.
- 29 Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978. doi:10.1145/359657.359659.
- 30 Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. Kalas: A verified, end-to-end compiler for a choreographic language. In June Andronick and Leonardo de Moura, editors, *Procs. ITP*, volume 237 of *LIPICs*, pages 27:1–27:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ITP.2022.27.
- 31 Alan M. Turing. Computability and λ -definability. *J. Symb. Log.*, 2(4):153–163, 1937. doi:10.2307/2268280.