EPiC Series in Computing

Volume 94, 2023, Pages 144–163

Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Keep me out of the loop: a more flexible choreographic projection

Luís Cruz-Filipe, Fabrizio Montesi, and Robert R. Rasmussen

University of Southern Denmark, Odense, Denmark
{lcf,fmontesi,rrr}@imada.sdu.dk

Abstract

Choreographic programming is a paradigm where programmers write global descriptions of distributed protocols, called *choreographies*, and correct implementations are automatically generated by a mechanism called *projection*. Not all choreographies are projectable, because decisions made by one process must be communicated to other processes whose behaviour depends on them – a property known as *knowledge of choice*.

The standard formulation of knowledge of choice disallows protocols such as third-party authentication with retries, where two processes iteratively interact, and other processes wait to be notified at the end of this loop. In this work we show how knowledge of choice can be weakened, extending the class of projectable choreographies with these and other interesting behaviours. The whole development is formalised in Coq. Working with a proof assistant was crucial to our development, because of the help it provided with detecting counterintuitive edge cases that would otherwise have gone unnoticed.

1 Introduction

Choreographic programming [40] is a programming paradigm for concurrent and distributed software, where developers write the desired communication behaviour that should be enacted by processes from a global viewpoint in a program called *choreography* [41]. Choreographies can be automatically *projected* to a distributed implementation consisting of a program for each process. A hallmark result in choreographic programming is *deadlock-freedom by design* [12]: distributed implementations projected from choreographies are guaranteed to be deadlock-free, since the syntax of choreographies cannot express mismatched communication actions.

An important open problem in choreographic programming is that current notions of projection are too restrictive when it comes to recursion: if some processes need to repeat some actions, then all other processes not involved in these actions have to know that the repetition will take place [41, Chapter 12]. This prevents applying choreographic programming to protocols where a process waits for the result of a "loop" that involves other processes, but does not itself participate in the loop. The aim of this paper is to address this problem.

Example 1. Consider a protocol for authentication with retry: a sensitive server is guarded from clients until the client has received a token (from an external identity provider) allowing the interaction to proceed. A simple way to express this protocol is by defining the following recursive choreographic procedure `X`, expressed in the choreographic language from [22].

```
X = Client.req ⟶ IP.x;
    If IP.check(x) Then IP.token ⟶ Client.x; Client.x ⟶ Server.x; Call Y
                   Else Call X
```

Procedure `X` starts with process `Client` communicating a request (`req`) to process `IP`, which stores the request in its local variable `x` (first line). Then, `IP` checks if the received request should be allowed to reach the `Server`. If so, `IP` communicates a `token` to `Client`, which `Client` can forward to `Server` and then enter procedure `Y` (which implements subsequent interactions between `Client` and `Server`, left unspecified here). Otherwise, this request is dropped and the procedure is recursively invoked to process another request by the `Client`.

This definition of `X` is not projectable in the current theory because of a problem known as *knowledge of choice* [15, 41]. Here specifically, the behaviour of `Client` and `Server` depends on the result of a test that only `IP` is aware of. To fix this, the standard solution is to propagate the necessary knowledge about a choice by means of *selections*: communications that carry constant, pre-defined values (also called *labels*) from which the receivers can infer what to do. In our example, we use the labels `left` and `right`, obtaining the following definition.

```
X = Client.req ⟶ IP.x;
    If IP.check(x) Then IP ⟶ Client[left]; IP ⟶ Server[left];
                        IP.token ⟶ Client.x; Client.x ⟶ Server.x; Call Y
                   Else IP ⟶ Client[right]; IP ⟶ Server[right]; Call X
```

From this choreography, we can generate implementations of `X` for all participants by using the projection procedure and process language formalised in [21].

```
X_IP = Client?x; If check(x) Then Client+left; Server+left; Call Y_IP
                             Else Client+right; Server+right; Call X_IP

X_Client = IP!req; IP & { left: IP?x; Server!x; Call Y_Client
                        | right: Call X_Client }

X_Server = IP & { left: Client?x; Call Y, Server
                | right: Call X_Server }
```

The procedure for `IP`, `X_IP`, receives (?) the request from the client and checks it. If the request is allowed (then-branch) then `IP` selects (+) the label `left` at both the client and the server. In the code for the client, `X_Client`, we start by sending (!) the request to `IP`. Then, we wait to receive a selection from the same process and switch (&) on the received label: if we receive `left`, then we know that the request is allowed and proceed accordingly; otherwise (`right`), we recursively invoke the procedure to try again. ◁

While there is in principle nothing wrong with the code shown at the end of Example 1, `Server` is forced to be aware of and communicate about every attempt performed by `Client`. That is, each time `Client` tries to obtain a token, `Server` receives a selection from `IP` informing it about the result of the authentication. This is impractical: it should be possible for the server to wait idly until `Client` has received a token, without being interrupted for every attempt. Furthermore, it can raise security issues, as it leaks unnecessary information to `Server`.

The general pattern from Example 1 is used in many applications, including message validation, cacheing, authentication and circuit breaker (server overload protection) [30, 43, 46]. Unfortunately, current theories of choreographic programming do not support it [12, 19, 26, 29, 41, 48]. In this work, we propose a new theory of projection for choreographies where it is enough for `IP` to notify `Client` of the success of the authentication, yielding the choreography below.

```
X = Client.req ⟶ IP.x; If IP.check(x) Then IP ⟶ Client[left]; IP.token ⟶ Client.x;
                                                Client.x ⟶ Server.x; Call Y
                                            Else IP ⟶ Client[right]; Call X [Server]
```

Crucially, the recursive call to `X` is annotated to indicate that `Server` does not need to be aware of it. (Whereas it is natural that the client does, since it needs to participate in enacting it.) Using this device, our new notion of projection produces the simpler definition `X_Server = Client?x; Call Y_Server`, which matches our intended behaviour.

Improving projection to deal with this pattern without renouncing formal correctness turned out to be technically challenging. Intuitively, the key reason resides in the fact that projection needs to determine which parts of a choreography should be translated to code for a given process and which should be skipped. Previously, this amounted to a simple check of whether the given process was involved in each individual choreography action. However, when we add the ability to declare that a process is not involved in a procedure call (through our new syntax) this check sometimes requires a more complex analysis of the structure of the remainder of the choreography.

As we show, one can write choreographies where declaring a process not to be involved in a procedure call (through our new syntax) does not make sense given the invoked procedure. Deciding when a choreography is well-formed in this sense requires considering several tricky corner cases.

To develop our notion of projection and prove it correct (the projected code is semantically correspondent to the source choreography), we turned to interactive theorem proving. Specifically, we based our development on the formalisation of choreographic programming given in Coq [5] presented in [21, 22]. Using an interactive theorem prover in our development was essential to the discovery of corner cases that we would have otherwise missed—and, ultimately, to gaining confidence that we had dealt with all such cases!

Our main contribution consists of the definition and proof of correctness of the new projection procedure, which is given for an existing and well-studied choreographic programming language [22] (modulo our enriched syntax for procedure calls). Our entire development is formalised in the Coq theorem prover. We also show that all previously proven results still hold for our extension of the language.

2 Related work

Choreographic programming has been applied to several settings, including service-oriented computing [12, 26, 40], cyberphysical systems [37, 38], security protocols [7, 36], distributed agreement [35], and general concurrent and distributed programming [18, 28, 29, 50]. A comprehensive introduction to theory of choreographic languages is given in [41]. Many of the principles behind projection were studied in the context of web services [10, 49] and can be traced back to research on message sequence charts and similar structures for describing interaction protocols [1, 2, 34]. Available implementations of choreographic programming languages include Chor [12, 40], Choral [28], AIOCJ [26], HasChor [50], and hacc [18]. None of these existing theories and implementations can compile choreographies with the pattern that we introduce.

Our work joins the line of work on formalised theories of choreographic programming. The Coq formalisation we used as starting point for our development [23] (the extended version of [21, 22]) formalises a substantial part of the theory in [41], and was inspired by earlier research on a core model of choreographic programming (Core Choreographies [19]). The Coq development from [23] has been used as a certified component of a tool [18] that compiles

choreographies to the service-oriented language Jolie [42]. We expect that our development could be used to produce a new improved version of this component. Other formalisations of choreographic programming include Kalas [48], which generates executable code in CakeML [45], and Pirouette [29], which features higher-order composition.

Languages for expressing choreographies are used also for the purposes of documenting, specifying, testing, and verifying the communication behaviour of concurrent systems, in textual and visual forms [3, 4, 6, 8, 16, 17, 24, 25, 31, 33, 34, 41, 47, 51]. For example, multiparty session types are abstract choreographies that do not specify how message payloads are computed (as in choreographic programming), but only their types [31]. The relation between choreographic programming and multiparty session types is well-studied: the latter can be used as types for programs in the former [12, 27, 44], and their integration has been explained also in terms of linear logic [9, 11, 13, 14].

Research into multiparty sessions types has also observed shortcomings like Example 1 in their context [32, 39], showing that this is not a defect of our particular theory, but rather a more general problem with these kinds of frameworks. While the extensions proposed by these authors are based upon similar intuitions to ours, the technical development is different due to the differences in the underlying frameworks.

3 Background

We dedicate this section to recap the parts of [23] directly relevant to our work. Throughout, we use (slightly simplified) Coq notation to explain the concepts and formalisation details. Moreover, we ignore some aspects of the formalisation that are immaterial to our work, namely: that the language is paramaterised on a signature; and that interactions have annotations.

3.1 Syntax of choreographies

The syntax of our choreography language is given by the following grammar.

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

A choreography `C` can be: an interaction η followed by a continuation (η; `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`, where `ps` is a list of processes that remain to invoke `X`;[1] or the terminated choreography `End`. An interaction η can be: a communication `p.e` $\longrightarrow$ `q.x`, where process `p` sends the result of local evaluating expression `e` to process `q`, which stores it in its local variable `x`; or a selection `p` $\longrightarrow$ `q[l]`, where `p` sends label `l` (either `left` or `right`) to process `q`. Choreographies are formalised in Coq as an inductive type called `Choreography`.

Executing a choreography requires knowing procedure definitions. A set of procedure definitions is defined as a mapping from procedure names to pairs of process names (the procedure's annotation) and choreographies.

```
Definition DefSet := RecVar → (list Pid)*Choreography.
```

A *choreographic program* is then a pair consisting of a set of procedure definitions and a (main) choreography. The procedures are static while the main choreography is dynamic, that is, executing the program changes the main choreography but not the procedure definitions.

[1] Runtime terms are used in the semantics of choreographies, described below.

$$\frac{\texttt{v} := \texttt{eval e s p} \quad \texttt{s'} \ [==] \ \texttt{s}[\![\texttt{q,x} \Rightarrow \texttt{v}]\!]}{(\texttt{D}, \texttt{p.e} \longrightarrow \texttt{q.x; C,s}) \ \text{—}[\texttt{TL\_Com p v q}]\text{⟶} \ (\texttt{D,C,s'})} \ \texttt{CC\_Com}$$

$$\frac{\texttt{beval b s p} = \texttt{true} \quad \texttt{s} \ [==] \ \texttt{s'}}{(\texttt{D}, \texttt{If p.b Then C1 Else C2,s}) \ \text{—}[\texttt{TL\_Tau p}]\text{⟶} \ (\texttt{D,C1,s'})} \ \texttt{CC\_Then}$$

$$\frac{[\#](\texttt{fst (D X)}) = 1 \quad \texttt{In p (fst (D X))} \quad \texttt{s} \ [==] \ \texttt{s'}}{(\texttt{D}, \ \texttt{Call X, s}) \ \text{—}[\texttt{TL\_Tau p}]\text{⟶} \ (\texttt{D, snd (D X), s'})} \ \texttt{CC\_Call\_Local}$$

$$\frac{[\#](\texttt{fst (D X)}) > 1 \quad \texttt{In p (fst (D X))} \quad \texttt{s} \ [==] \ \texttt{s'}}{(\texttt{D}, \ \texttt{Call X, s}) \ \text{—}[\texttt{TL\_Tau p}]\text{⟶} \ (\texttt{D, RT\_Call X (fst (D X)) [\textbackslash] p) (snd (D X)), s'})} \ \texttt{CC\_Call\_Start}$$

$$\frac{[\#]\texttt{ps} > 1 \quad \texttt{In p ps} \quad \texttt{s} \ [==] \ \texttt{s'}}{(\texttt{D}, \ \texttt{RT\_Call X ps C, s}) \ \text{—}[\texttt{TL\_Tau p}]\text{⟶} \ (\texttt{D, RT\_Call X (ps [\textbackslash] p) C, s'})} \ \texttt{CC\_Call\_Enter}$$

$$\frac{[\#]\texttt{ps} = 1 \quad \texttt{In p ps} \quad \texttt{s} \ [==] \ \texttt{s'}}{(\texttt{D}, \ \texttt{RT\_Call X ps C, s}) \ \text{—}[\texttt{TL\_Tau p}]\text{⟶} \ (\texttt{D, C, s'})} \ \texttt{CC\_Call\_Finish}$$

Figure 1: Semantics of choreographies (selected rules).

```
Definition Program := DefSet * Choreography.
```

We write `Main P` for the dynamic part (choreography) of `P`. Programs are subject to a number of well-formedness conditions, e.g., no process communicates with itself. We discuss these in Section 4.1, in the context of our extended theory.

3.2 Semantics of choreographies

The semantics of choreographies makes some assumptions inspired from process calculi: each process runs independently of each other and has access to a local store; communications are synchronous; and the network is reliable. Formally, this semantics is given as a labelled transition system on configurations, consisting of a program and a (global) state. States associate to each process a map from variable names to values, corresponding to the local store of that process. States can be updated: $\texttt{s}[\![\texttt{p,x} \Rightarrow \texttt{v}]\!]$ is the state obtained from updating `s` with the mapping $\texttt{p,x} \mapsto \texttt{v}$. We write `s [==] s'` to denote that `s` and `s'` are extensionally equal.

The transition relation of the labelled transition system is defined implicitly by a number of transition rules for choreographic configurations; a selection of these are shown in Figure 1.[2] Transitions have the form $(\texttt{D,C,s}) \ \text{—}[\texttt{t}]\text{⟶} \ (\texttt{D,C',s'})$, where `t` is a transition label describing the leakage – what can be observed from the transition happening. The reflexive and transitive closure of the transition relation is written $\text{—}[\texttt{tl}]\text{⟶}^*$, where `tl` is a list of transition labels.

Rule `CC_Com` deals with executing a communication from a process `p` to a process `q`: if the expression `e` at `p` evaluates to a value `v` (using 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 leakage is `TL_Com p v q`, denoting that `p` has communicated the value `v` to `q`. Rule `CC_Sel` (not shown) for selections is similar, but does not change the state.

Rule `CC_Then` allows a process `p` to evaluate the guard `b` of a conditional to `true` (using the

[2] The formalisation defines this relation in two layers for technical reasons immaterial to our development.

auxiliary function `beval`), proceeding to the then-branch of the conditional. The leakage is `TL_Tau p`, denoting that `p` has executed an internal action. The dual rule `CC_Else` is not shown.

The remaining rules shown deal with procedure calls. A procedure call reduces when a process enters it (rule `CC_Call_Start`), and is replaced by a runtime term including the procedure's name, the list of processes that still need to enter it, and the procedure's definition. Other processes can now enter the procedure (rule `CC_Call_Enter`), and are simply removed from the runtime term's list. When the last process enters the procedure, rule `CC_Call_Finish` consumes the runtime term, leaving only the choreographic fragment `C` that still needs to be executed. In case of a procedure annotated with a single process, rule `CC_Call_Local` is used rather than `CC_Call_Start`: it reduces the procedure call directly into the procedure's definition. All cases where a process `p` enters a procedure (either reducing a call or runtime term) leaks `TL_Tau p`.

There are transition rules allowing for out-of-order execution, but these are not important for our work. (In particular, they allow for the choreography `C` in runtime terms `RT_Call X ps C` to reduce, which justifies its inclusion.)

An important consequence of these definitions is that choreographic programs enjoy *deadlock-freedom by design*: any non-terminated program (i.e., any program whose main choreography is not `End`) can always perform some action.

3.3 The process calculus

Implementations of choreographies are modelled in a formalised process calculus following [21, 41]. This calculus follows the standard way of representing systems of communicating processes, by giving the code of each process separately and achieving communication when processes perform compatible I/O actions.

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

```
B ::= p!e; B | p?x; B | p+l; 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. The first two productions deal with communication. A process executing a send action `p!e; B` evaluates expression `e` and sends the result to process `p`, afterwards continuing as `B`. Dually, a process executing a receive action `p?x; B` receives a value from `p` and stores it in `x` before continuing as `B`.

Selections are implemented by `p+l; B` and `p & mB1 // mB2`. The former behaviour describes sending label `l` to process `p` and continuing as `B`. The latter is a branching term, where `mB1` and `mB2` are the behaviours that the process executes upon receiving `left` or `right` from `p`, respectively. Since a process does not need to offer behaviours for all labels, both `mB1` and `mB2` have type `option Behaviour`. Conditionals (`If b Then B1 Else B2`), procedure calls (`Call X`), and the terminated behaviour (`End`) are as in choreographies.

Processes run together in networks, which are maps from processes to behaviours. A program pairs a network with a set of procedure definitions that all processes in the network can invoke. The semantics of networks is also given as a labelled transition system on configurations that consist of a program and a memory state. We omit its formal definition, as this presentation does not depend on it, and refer the interested reader to [21].

3.4 Endpoint Projection (EPP)

Choreographies are compiled to behaviours by a procedure called *endpoint projection*. This procedure is a partial function, and since all functions in Coq are total it was formalised as an inductive relation `bproj : DefSet → Choreography → Pid → Behaviour → Prop`. We use the

$$\frac{[\![\mathtt{D}, \mathtt{C} \mid \mathtt{p}]\!] == \mathtt{B}}{[\![\mathtt{D}, \mathtt{p}.\mathtt{e} \longrightarrow \mathtt{q}.\mathtt{x}; \mathtt{C} \mid \mathtt{p}]\!] == \mathtt{q}!\mathtt{e}; \mathtt{B}}\ \mathtt{bproj\_Send} \qquad \frac{\mathtt{p} \neq \mathtt{q} \quad [\![\mathtt{D}, \mathtt{C} \mid \mathtt{q}]\!] == \mathtt{B}}{[\![\mathtt{D}, \mathtt{p}.\mathtt{e} \longrightarrow \mathtt{q}.\mathtt{x}; \mathtt{C} \mid \mathtt{q}]\!] == \mathtt{p}?\mathtt{x}; \mathtt{B}}\ \mathtt{bproj\_Recv}$$

$$\frac{\mathtt{p} \neq \mathtt{r} \quad \mathtt{q} \neq \mathtt{r} \quad [\![\mathtt{D}, \mathtt{C} \mid \mathtt{r}]\!] == \mathtt{B}}{[\![\mathtt{D}, \mathtt{p}.\mathtt{e} \longrightarrow \mathtt{q}.\mathtt{x}; \mathtt{C} \mid \mathtt{r}]\!] == \mathtt{B}}\ \mathtt{bproj\_Com}$$

$$\frac{\mathtt{p} \in \mathtt{fst}(\mathtt{D}\ \mathtt{X})}{[\![\mathtt{D},\ \mathtt{Call}\ \mathtt{X} \mid \mathtt{p}]\!] == \mathtt{Call}\ (\mathtt{X}, \mathtt{p})}\ \mathtt{bproj\_Call\_in} \qquad \frac{\mathtt{p} \notin \mathtt{fst}(\mathtt{D}\ \mathtt{X})}{[\![\mathtt{D},\ \mathtt{Call}\ \mathtt{X} \mid \mathtt{p}]\!] == \mathtt{End}}\ \mathtt{bproj\_Call\_out}$$

$$\frac{\mathtt{p} \neq \mathtt{r} \quad [\![\mathtt{D}, \mathtt{C1} \mid \mathtt{p}]\!] == \mathtt{B1} \quad [\![\mathtt{D}, \mathtt{C2} \mid \mathtt{p}]\!] == \mathtt{B2} \quad \mathtt{B1}\ [\mathtt{V}]\ \mathtt{B2} == \mathtt{B}}{[\![\mathtt{D}, \mathtt{If}\ \mathtt{r}.\mathtt{b}\ \mathtt{Then}\ \mathtt{C1}\ \mathtt{Else}\ \mathtt{C2} \mid \mathtt{p}]\!] == \mathtt{B}}\ \mathtt{bproj\_Cond'}$$

Figure 2: Selected rules for behaviour projection.

$$\frac{}{\mathtt{p}\ \&\ \mathtt{Some}\ \mathtt{bL}\ //\ \mathtt{None}\ [\mathtt{V}]\ \mathtt{p}\ \&\ \mathtt{None}\ //\ \mathtt{Some}\ \mathtt{bR} == \mathtt{p}\ \&\ \mathtt{Some}\ \mathtt{bL}\ //\ \mathtt{Some}\ \mathtt{bR}}\ \mathtt{merge\_SNNS}$$

$$\frac{\mathtt{bL1}\ [\mathtt{V}]\ \mathtt{bL2} == \mathtt{bL} \quad \mathtt{bR1}\ [\mathtt{V}]\ \mathtt{bR2} == \mathtt{bR}}{\mathtt{p}\ \&\ \mathtt{Some}\ \mathtt{bL1}\ //\ \mathtt{Some}\ \mathtt{bR1}\ [\mathtt{V}]\ \mathtt{p}\ \&\ \mathtt{Some}\ \mathtt{bL2}\ //\ \mathtt{Some}\ \mathtt{bR2} == \mathtt{p}\ \&\ \mathtt{Some}\ \mathtt{bL}\ //\ \mathtt{Some}\ \mathtt{bR}}\ \mathtt{merge\_SSSS}$$

Figure 3: Definition of the merge relation (selected rules).

suggestive notation write $[\![\mathtt{D}, \mathtt{C} \mid \mathtt{p}]\!] == \mathtt{B}$ for `bproj D C p B`, read "the projection of `C` on `p` in the context of the set of procedure definitions `D` is `B`".

Intuitively, behaviour projection is computed by going through the choreography and trying to construct the local action corresponding to each choreographic construct. For example, a communication $\mathtt{p}.\mathtt{e} \longrightarrow \mathtt{q}.\mathtt{x}; \mathtt{C}$ is projected for `p` as a send action to `q` followed by the projection of the continuation `C`; for `q` as a receive action from `p`, again followed by the projection of `C`; and for other processes simply as the projection of `C`.

A representative selection of rules is given in Figure 2. The rules for projecting communications formalise the intuition given above; the rules for projecting selections and the terminated choreography follow the same principle. A call to a procedure `X` is projected by checking whether the process is involved in the call (the first argument in `D X`).

Projection of conditionals is special because of the requirement of knowledge of choice mentioned in the Introduction. Projecting a conditional for the process evaluating the guard is still simple – the projection is a local conditional whose two branches computed recursively. For other processes (rule `bproj_Cond'`, the two branches are projected separately and then combined using another partial function called *merging* (the premise `B1 [V] B2 == B`).

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. The key rule is `merge_SNNS` (Figure 3), merging two branching terms where one offers a behaviour on `left` and the other offers a behaviour on `right`. If both terms offer behaviours on the same label, these are recursively combined as in rule `merge_SSSS`. In all cases, the process choosing the label must be the same. If `B1` and `B2` are built from the same constructor other than branching, merge requires its first action to coincide (e.g. send the same expression to the same process), tries to merge their continuations, and prepends the common first action to the result.

Merging is a partial function (for example, `End` and `p+l; End` cannot be merged), giving rise to the following notion of *projectability*.

```
Definition projectable_B D C p := ∃ B, ⟦D,C | p⟧ == B.
```

A choreographic program `P` is projectable, `projectable_P P`, if `Main P` is projectable for all processes in `P` and all procedures are projectable for the processes that they use. Finally, endpoint projection (EPP) is defined as a function that maps a projectable choreographic program to the process program obtained by combining all these projections.

The relationship between choreographic programs and their projections is described by the *EPP theorem*. Informally, this theorem states that choreographic programs and their projections can simulate each other.

Two aspects make its formulation and proof particularly challenging: (1) the interplay between conditionals and selections during choreographic execution creates lingering options at branching terms when projecting; (2) care must be taken to formalise the correct usage of runtime terms. The former can be solved by introducing the concept called *branching order*, which is the least upper bound of merge. The latter is partly handled by the concepts of *initial choreographies* and *well-formed programs*, which are syntactic requirements. A further refinement is the notion *strong projectability*, which captures the requirement that the choreography `C` in `RT_Call X ps C` must have originated from the definition of `X` by executing some transitions. The interested reader should consult [23] for details.

One of the important consequences of the EPP theorem is that projections of choreographic programs cannot deadlock (since choreographies are deadlock-free). In general, the EPP theorem is of particular interest, since EPP has little relevance without it; thus, while extending the theory we take special care to ensure that it still holds.

4 A new choreographic theory

As we anticipated in Example 1, we extend our choreography language with the possiblity of marking some processes as *muted* in recursive procedure calls – they should not enter the procedure again, but rather ignore that instruction. We achieve this by changing the constructor for procedure calls in choreographies, so that the syntax of choreographies becomes

```
C ::= η; C | If p.b Then C1 Else C2 | Call X ps | RT_Call X ps C | End
```

and updating the semantic inference rules `CC_Call_Local` and `CC_Call_Start` as follows

$$\frac{[\#](\texttt{fst (D X) [\textbackslash\textbackslash] ps}) = 1 \quad \texttt{In p (fst (D X) [\textbackslash\textbackslash] ps)} \quad \texttt{s [==] s'}}{(\texttt{D, Call X ps, s}) \xrightarrow{\texttt{[TL\_Tau p]}} (\texttt{D, snd (D X), s'})}\ \texttt{CC\_Call\_Local}$$

$$\frac{[\#](\texttt{fst (D X) [\textbackslash\textbackslash] ps}) > 1 \quad \texttt{In p (fst (D X) [\textbackslash\textbackslash] ps)} \quad \texttt{s [==] s'}}{(\texttt{D, Call X ps, s}) \xrightarrow{\texttt{[TL\_Tau p]}} (\texttt{D, RT\_Call X (fst (D X) [\textbackslash] p [\textbackslash\textbackslash] ps) (snd (D X)), s'})}\ \texttt{CC\_Call\_Start}$$

These rules capture the intended semantics of our new procedure calls: the first process entering a procedure is in its annotation and not muted; and the generated runtime term excludes muted processes. This exclusion uses the operator for set difference, denoted `[\\]`. These rules model the intuition that muted processes have already entered the procedure, and their premises ensure that only one rule can be applied (as before).

This language also enjoys deadlock freedom – the proof is relatively easy to adapt, as it only requires minor changes in the case for procedure calls, where the syntax and semantics were modified.

4.1 Well-formedness of choreographies

The original syntax of choreographies allows for expressions that do not respect the intended usage of the language, such as $\texttt{p.e} \longrightarrow \texttt{p.x}$ (self-communications). In order to exclude such terms, our choreographic language included a notion of well-formedness [22, 23].

The possibility of muting processes in procedure calls requires us to add a number of constraints to this concept, to ensure that our semantics for muted processes makes sense. Furthermore, well-formedness should be preserved by execution, in the sense that[3]

```
Lemma CCP_ToStar_Program_WF : Program_WF P → (P,s) —[l]→* (P',s') → Program_WF P'
```

from the original formalisation [23] should still hold for the new language.

Initial choreographies. We start by discussing restrictions that are specific to main choreographies that programmers should write – *initial* choreographies.

The first restriction, inherited from the original formalisation, is that initial choreographies cannot use runtime terms: these are generated by the semantics.[4] Additionally, we now require initial choreographies not to have muted processes initially (every procedure call must have an empty list): muted processes should only appear when expanding recursive procedure definitions.

```
Definition initial (C:Choreography) : Prop := no_runtime_terms C ∧ passive_pn C = nil.
```

Both `no_runtime_terms` and `passive_pn` are defined recursively in the natural way.

Well-formed programs. The full list of requirements is summarised in Table 1. Requirements annotated with † and ‡ arise from technical aspects related to projection or EPP, and are discussed in the corresponding sections. We discuss the remaining ones below.

As can be seen in the table, the requirements for well-formed main choreographies and well-formed procedure definitions are distinct; for example, procedure definitions cannot include runtime terms, which naturally arise in choreography bodies throughout execution.

Some requirements are inherited from the original theory: `no_self_comm`, `no_runtime_terms`, `no_empty_ann`, `consistent`, and `well_ann`. The first two are explained above; the remaining ones ensure that runtime terms were correctly generated from some procedure definition (the processes waiting to enter a procedure form a non-empty subset of the procedure's annotation).

The four new requirements `no_total_silence`, `no_silent_intruders`, `correctly_muted`, and `no_muted_external_call` reflect our intention that muted processes are those that do not need to be notified of a procedure recursively calling itself.

The first, `no_total_silence`, forbids wrongly muting all processes, while `no_silent_intruders` disallows meaningless muting of processes not involved in the procedure, and simplifies the proof of several results. These two predicates are defined recursively in the natural way.

The third, `correctly_muted`, describes when a process is allowed to "wait" in a call: if a process is muted in a call, then it should not be active before it is being muted. This captures the intuition that the process is waiting for some other processes before starting to execute the procedure. This predicate takes two additional arguments: the process we are checking, and a `bool` (initially `false`) indicating whether the process has been used up to this point. In recursive calls, this second argument is changed to `true` if the process occurs at the top of the choreography body.

[3] To alleviate the notation, we omit all universally quantified variables at the top of formulas.

[4] In the original theory, this was the only requirement for initial choreographies, which was then imposed to both the main choreography and all procedure definitions.

Name	C	P	Description
Interactions			
`no_self_comm`	✓	✓	Processes in interactions are different.
Runtime terms			
`no_runtime_terms`		✓	There are no runtime terms.
`no_empty_ann`	✓		All runtime terms only contains non-empty lists.
`consistent`	✓		Lists in runtime terms only contains processes in the associated procedure's annotation.
Procedures			
`well_ann`		✓	Processes in a procedure are in its annotation, which is non-empty.
Muted processes			
`no_total_silence`	✓	✓	Every call has at least one non-muted process.
`no_silent_intruders`	✓	✓	Muted processes are in the annotation of the procedure.
`correctly_muted`		✓	Muted processes are not active before they are muted.
`no_muted_external_call`		✓	Calls with muted processes are to the procedure they occur in.
† `passive_active`		✓	Passive processes are also active elsewhere.
† `branch_passiveness`		✓	If a process is passive in a branch of a conditional, it is passive in all its branches.
† `passive_bridge`	✓		If a process is muted in a call to some procedure, then it is muted in that procedure's body.
‡ `no_passive_activity`		✓	If a process is passive, then it interacts with a non-passive process in the branch where it is non-passive.

Table 1: Well-formedness requirements that apply to the main choreography (C) or to procedure definitions (P). See Sections 5 and 6 for requirements annotated † and ‡, respectively.

Example 2. Consider the definition of `X` given at the start of Example 1. Process `Client` would not be correctly muted, since it participates in an interaction with `IP` before the call. However, `Server` is correctly muted. This motivates our choice to mute `Server`, but not `Client`. ◁

Early in the development, we believed that `correctly_muted` should also be imposed on the main choreography. However, while trying to show that this property was preserved by execution, Coq guided us to corner cases that turned out to yield a counterexample. This is an example of the usefulness of Coq as a research tool, which helped us abandon an incorrect development path early on.

The last requirement, `no_muted_external_call`, also defined recursively, again expresses the intuition that processes should only be muted when a procedure recursively calls itself. This design option simplifies our development. In future work, we plan to investigate how to relax this requirement, for example to allow for mutually recursive procedures with muted processes.

The requirements for well-formed procedures are all gathered into a predicate `Procedures_WF` :

`DefSet` → `Prop`, which states that all defined procedures satisfy them. This definition gives us a compact way to reason about all procedures in a program.

5 Projection strikes back

Muted processes in a well-formed program can only appear in recursive calls of procedures where they do not perform any actions in the branch of the procedure's execution leading to the recursive call. This suggests that the projection rules for the revised call terms should be:

$$\frac{\texttt{p} \notin \texttt{fst(D X)}}{\texttt{[\![D, Call X ps | p]\!] == End}}\ \texttt{bproj\_Call\_out} \qquad \frac{\texttt{p} \in \texttt{fst(D X)} \quad \texttt{p} \notin \texttt{ps}}{\texttt{[\![D, Call X ps | p]\!] == Call X}}\ \texttt{bproj\_Call\_in}$$

$$\frac{\texttt{p} \in \texttt{fst(D X)} \quad \texttt{p} \in \texttt{ps} \quad \texttt{[\![D, snd (D X) | p]\!] == B}}{\texttt{[\![D, Call X ps | p]\!] == B}}\ \texttt{bproj\_Call\_in'}$$

Unfortunately, this intuitive "definition" does not terminate: if we try to project the definition of `X` in Example 1 for `Server`, we end up in an infinite loop.

We illustrate this problem with a simplified version of our earlier example.

Example 3. Consider the choreographic procedure

`X = If p.check() Then (p.e ⟶ q.y; End) Else Call X [q]`

and suppose that we want to compute the projection of this procedure for `q`. This projection is obtained by merging the projections of both branches in the conditional. The projection of the then-branch is simply `p?y`, while the rule above states that the projection of the else-branch is again the projection that we are trying to compute. ◁

Example 3 shows that the fundamental problem comes from the projection of the conditional – both branches should be projected, and the resulting projections merged. But projecting the else-branch requires projecting the original choreography, leading to the infinite loop.

In order to avoid this, we introduce a secondary projection `pproj D C r B`, which we use to project procedure bodies in the premise of rule `bproj_Call_in'`. We denote this relation as `{{D, C | r }}== B`, so that this rule now becomes:

$$\frac{\texttt{p} \in \texttt{fst(D X)} \quad \texttt{p} \in \texttt{ps} \quad \texttt{\{\{D, snd (D X) | p\}\} == B}}{\texttt{[\![D, Call X ps | p]\!] == B}}\ \texttt{bproj\_Call\_in'}$$

The key insight defining `pproj` is that, in Example 3, we should not even care to project the else-branch. From `q`'s perspective, the interesting behaviour occurs in the then-branch – recall our intuition that `q` is essentially waiting for `p`, which in our example means that it is silently waiting for the choreography to enter the then-branch. The next section discusses how to formalise this intuition.

5.1 Introducing passive processes

To develop a smarter way of projecting conditionals, we introduce the concepts of *active* and *passive* processes, already anticipated in the previous section.

A process `r` is passive in a choreography `C` if `r` occurs muted in some procedure call occurring in `C`. A process `r` is active in a choreography `C` if `r` occurs in `C` outside of a procedure call where it is muted. These properties are defined recursively over `C` as two predicates `passive` and `active`.

To compute the sets of passive and active processes in a choreography, we define two auxiliary functions `passive_pn` and `active_pn`, both of type `Choreography → set Pid`. These functions are used to formalise well-formedness: they simplify some decidability proofs, since membership in a list is decidable.[5] Both formulations can be used interchangeably, as stated by e.g.

```
Lemma passive_iff : passive D p C ↔ In p (passive_pn D C).
```

Using these notions, we can define a smarter notion of projection for conditionals. The design philosophy is: if a process is active in a branch of a conditional, then something interesting happens in that branch, and we need to project it; otherwise we should ignore it. This yields four cases for projecting a conditional for a process `p` not evaluating the guard: (i) `p` is active in both branches: then something interesting happens in both branches, and we project both branches and merge them as before; (ii) `p` is inactive in both branches: then it is not involved in any of the branches, and we project it as `bnil`; (iii) `p` is active in the then-branch and inactive[6] and passive in the else-branch: this is the situation illustrated in Examples 1 and 3, and as discussed we should only project the then-branch – see rule `pproj_cond_ap` below; (iv) the symmetric case where `p` is active in the else-branch and inactive and passive in the then-branch.

$$\frac{\texttt{p} \neq \texttt{r} \quad \{\{\texttt{D,C1} \mid \texttt{p}\}\} == \texttt{B} \quad \texttt{active D r C1} \quad \sim \texttt{active D r C2} \quad \texttt{passive r C2}}{\{\{\texttt{D,If p.b Then C1 Else C2} \mid \texttt{p}\}\} == \texttt{B}}\ \texttt{pproj\_Cond\_ap}$$

Example 4. It is easy to see that the definition of `X` in Example 3 can be projected for `q` as `p?y` using `pproj`. As a consequence, the projection using `bproj` can now be obtained by merging `p?y` with itself, yielding `p?y`. ◁

The definition of `pproj` still calls itself on rule `pproj_Call_in'` (analogous to `bproj_Call_in'`), but intuitively this rule should never be applied: it does not make sense to mute a process in all recursive calls in a procedure definition – such processes should simply not be part of the procedure's annotation, as it is simply waiting without ever doing anything. This intuition is captured by well-formedness, as we discuss below.

The attentive reader may have noticed that our rules for projecting conditionals are not exhaustive – what if a process `r` is active in the then-branch of a conditional, but inactive and non-passive in the else-branch? In this case, the then-branch projects to something other than `bnil` (since `r` is active), but the else-branch projects to `bnil` (since `r` does not occur in it). But then `r` is missing knowledge of choice, whence the choreography should be unprojectable. The same argument applies to the dual case.

5.2 Passiveness and well-formedness

We can now discuss the next set of well-formedness requirements from Table 1.

The first, `passive_active`, is motivated by the situation suggested above, where a process is never active inside a procedure – the simplest example would be the procedure definition `X = Call X [q]`. Clearly, nothing interesting happens for `q`[7], and attempting to compute `pproj` will result in an infinite loop. This is avoided by requiring all passive processes in a procedure definition to be also active in the same procedure definition.

```
Definition passive_active D X := (passive_pn (snd (D X))) [C] (active_pn D (snd (D X))).
```

[5] Note that sets are represented as lists in the formalisation.

[6] Well-formedness prevents a process from being active and passive simultaneously, but the definitions allow for it, and including this hypothesis simplifies a number of proofs.

[7] Other processes will still reduce, as they re-enter `X` infinitely many times.

This requirement has some useful consequences, when combined with `correctly_muted`:

```
Lemma procedure_WF_passive_has_if : Procedures_WF D →
  passive r (snd (D X)) → has_if (snd (D X)).

Lemma procedure_WF_passive_clear_until_if : Procedures_WF D → passive r (snd (D X)) →
  (∃ p t C1 C2, clear_until_if (snd (D X)) = If p ?? t Then C1 Else C2).
```

The first lemma says that well-formed procedure definitions with passive processes must include a conditional as a subterm, while the second is used to "extract" this conditional from the definition by ignoring anything coming before it. As usual, the proofs are by induction over `C`.

The idea is that `correctly_muted` forbids processes from being muted after being active. Thus, if a process is both active and passive in a procedure, there needs to be a conditional such that it can be active in one branch and inactive/passive in the other.

The next requirement, `branch_passiveness`, is meant to disallow choreographies requiring knowledge of choice in relation to termination. Consider the procedure definition

```
X = If p.b Then p.done → q; End
          Else (If p.b' Then Call X q Else End)
```

This choreography is problematic, because the else-branch (where `q` is not active) can terminate or recur without `q` being notified. Therefore, we require that a process that is passive in a branch of a conditional must be passive everywhere inside that branch.

Using `branch_passiveness`, we can prove:

```
Lemma procedure_WF_clear_until_if_Cond : ∀ D X p t C1 C2 r, Procedures_WF D →
  passive r (snd (D X)) → clear_until_if (snd (D X)) = If p ?? t Then C1 Else C2 →
  ((p≠ r ∧ active D r C1 ∧ ~passive r C1 ∧ ~active D r C2 ∧ passive_all r C2 ) ∨
  (p≠ r ∧ active D r C2 ∧ ~passive r C2 ∧ ~active D r C1 ∧ passive_all r C1 )).
```

describing the specific passive/active structure of well-formed procedures at the top conditional.

The last requirement we discuss in this section stems from the intuition that the main choreography should originate from an `initial` choreography by execution. While this notion is extremely challenging to formalise precisely, we only need one simpler property to guarantee preservation of well-formedness throughout execution.

```
Definition passive_bridge P := ∀ r X,
   In (r, X) (passive_call_pn (Main P)) → passive r (Procs P X).
```

This property states that: if the main choreography contains a procedure call to `X` where `r` is muted, then `r` must occur muted in `X` – since this is the only way it could have been introduced in the main choreography. The definition uses `passive_call_pn (Main P)`, which computes a set containing all pairs `(X,r)` such that `Main P` contains call to `X` where `r` is muted.

Preservation of well-formedness through transitions now is established by adding the result

```
Lemma CCP_To_passive_bridge: Program_WF P → (P,s) —[l]⟶ (P',s') → passive_bridge _ P'.
```

since all remaining properties in `Program_WF` were either present in the original formalisation or refer to `Procs P` (which does not change throughout execution).[8]

Intuitively, the lemma holds since muted calls can only be introduced by procedures, and well-formed procedures can only make muted recursive calls as per `no_muted_external_call`. The lemma is formally proven by induction over transitions.

[8]The function `Procs P` maps each procedure to its actual definition (excluding the list of involved processes).

5.3 Projections, compared

The separation of projection into `bproj` and `pproj` was instrumental into making projection computable. Using well-formedness, we can show that they work as intended, in the sense that they coincide on choreographies where either one could be applied.

The easiest case is that of projecting a choreography for a process that is not passive, as the cases where `bproj` and `pproj` differ are excluded.

```
Lemma bproj_pproj_not_passive : ~passive r C → ⟦D, C | r⟧ == B ↔ {{D, C | r}} == B
```

For passive processes, establishing this correspondence requires more work. Therefore, we prove the two implications separately.

```
Lemma bproj_pproj_passive : Procedures_WF _ D → passive r (snd (D X)) →
  ⟦D, (snd (D X)) | r ⟧ == B → {{D, (snd (D X)) | r}} == B.

Lemma pproj_bproj_passive : Procedures_WF _ D → passive r (snd (D X)) →
  {{D, (snd (D X)) | r }}== B → ⟦D, (snd (D X)) | r ⟧ == B.
```

Both proofs take advantage of the structure of well-formed procedures. Since `r` is passive, lemma `procedure_WF_passive_has_if` ensures that the definition of `X` contains a conditional as a subterm, which also satisfies all properties of well-formed procedure definitions. Moreover, `procedure_WF_clear_until_if_Cond` ensures that `r` cannot be the process evaluating the guard of the conditional, and that it has a rather specific passive/active structure.

Let us focus on the lemma `bproj_pproj_passive`. We use a lemma to deduce that the projection of `X` and the first conditional is equivalent for `bproj`:

```
Lemma bproj_procedure_clear_until_if : Procedures_WF _ D →
  passive r (snd (D X)) → clear_until_if _ (snd (D X)) = If p ?? b Then C1 Else C2 →
  ⟦D, snd (D X) | r ⟧ == B ↔ ⟦D, If p ?? b Then C1 Else C2 | r⟧ == B
```

We consider first the case where `r` is active and not passive in the then-branch, while it is inactive and passive in all branches in the else-branch (there is a symmetric case). Using `bproj_pproj_not_passive`, we conclude that both projection procedures behave identically in the then-branch. In the else-branch, we instead use

```
Lemma bproj_procedure_passive_all_not_active : passive_all r C →
  ~active D r C → no_muted_external_call _ D X C → no_silent_intruders D C →
  {{D, (snd (D X)) | r}} == B ↔ ⟦D, C |r⟧ == B
```

to conclude that the projecting the else-branch using `bproj` produces the same result as projecting `snd (D X)` using `pproj`. Now, we use a third lemma to deduce that the projection of `snd (D X)` and the first conditional coincide:

```
Lemma pproj_procedure_clear_until_if : Procedures_WF _ D →
  passive r (snd (D X)) → clear_until_if _ (snd (D X)) = If p ?? b Then C1 Else C2 →
  {{D, (snd (D X)) | r }}== B ↔ {{D, (If p ?? b Then C1 Else C2) | r }}== B.
```

Since `r` is active and not passive in the then-branch, we conclude that `pproj` projects the conditional as the then-branch. But projecting the then-branch using `pproj` gives the same result as projecting either the then-branch or the else-branch using `bproj`. Applying uniqueness of `pproj` together with uniqueness and idempotency of `merge` allows us to complete the proof. All helper lemmas are proved by induction on the appropriate choreography.

The proof of `pproj_bproj_passive` follows a similar approach. The corresponding auxiliary results `bproj_procedure_clear_until_if` and `pproj_procedure_clear_until_if` are straightforward to prove. Each proof is split into two helper lemmas – one for each direction – and they

proceed by induction over C, using that fact that r is not present in any instruction before reaching the conditional (since r is passive and correctly_muted hold).

The result bproj_procedure_passive_all_not_active is also not surprising, since bproj projects muted calls using pproj. Since r is not active in C, it is not present in any instruction in C. However, r is passive in all branches of C, which must be due to muted calls to X (by no_muted_external_call assumption). These ideas are used in the formalised proof, which is performed by induction over C.

Therefore bproj and pproj coincide on procedure definitions in well-formed programs.

```
Lemma bproj_pproj : Procedures_WF _ D → {{D, (snd (D X)) | r }}== B ↔ ⟦D, (snd (D X)) | r ⟧== B
```

5.4 Decidability

Projection is a partial function formalised as a relation. In order to be able to compute it, we need to have some decidability results. The major change with respect to the old theory is that these results now require the choreography to be well-formed, to exclude choreographies such as X = Call X [q], where projection loops infinitely. This is not a big issue, as well-formedness is decidable in practice (i.e. for programs that only use a finite number of procedures).

```
Lemma bproj_dec : Program_WF (D, C) → { B | ⟦D,C | p⟧ == B } + { ~projectable_B D C p}
```

As in the original theory, the proof is by induction over C. To handle muted calls we use:

```
Lemma pproj_dec : Procedures_WF _ D →
  { B |  {{D, (snd (D X)) |  p }}== B } + { ~∃  B, {{D, (snd (D X)) | p }}== B}.
```

The idea is: if p is inactive in X, then it is not passive (the contrapositive of requiring that passive processes are active), so its projection is bnil. Otherwise, p is active in X, and we apply

```
Lemma pproj_dec' : active D p C → correctly_muted _ D C false p →
  no_self_comm C → no_runtime_terms C → branch_passiveness _ p C →
  { B |  {{D, C | p }}== B } + { ~∃  B, {{D, C | p }}== B}.
```

whose assumptions follow either from well-formedness (if p is passive in X) or from other lemmas (if p is not passive in X). This proof is again by induction on C; it requires more case analysis due to the more complex way of projecting conditionals, but the previously problematic case of muted calls does not arise anymore since p is active in C.

6 The return of the EPP theorem

The proof of the EPP theorem is mostly unaffected by our changes. To make it more manageable, the original formalisation split this in two parts (soundness and completeness), and the proof of soundness is further split into auxiliary lemmas for each subcase. These proofs are adapted to the new projection simply by adding the case that deals with muted processes in procedure calls.[9] This new case also motivates the last condition in well-formedness.

Example 5. Consider the following procedure definition.

```
X = If p.b Then (r.y ⟶ s.x; End) Else (Call X [r,s])
```

[9] These lemmas talk about the main choreography, so the new way to project conditionals in procedure definitions does not affect their proofs. This is a nice consequence of having two distinct projections.

Projecting `X` for `r` and `s`, respectively, yields `s!y; End` and `r?x; End`. This means that, in the projected network, `r` and `s` can communicate directly without waiting for the recursive calls in `X` to terminate. However, the choreography cannot model this behaviour, as these processes have to wait for `p.b` to evaluate to false. ◁

This example illustrates a form of knowledge of choice: `r` and `s` need to be informed that they can communicate by some process that is involved in the recursive part of `X`. To disallow it, we require that: if `p` is passive in the definition of `X`, then `p` must interact with a non-passive process in every branch of `X` where `p` is non-passive. In the example above, this means that both `r` and `s` must communicate with `p` before they can interact with each other. This requirement is defined as a predicate `no_passive_activity`, defined recursively over choreographies. This requirement might sound restrictive, but it captures our original motivation for muted processes: these processes are waiting for some external event before proceeding.

This requirement excludes the case where a choreography `Call X ps` reduces by an action involving only muted processes (i.e. processes in `ps`), allowing us to prove lemmas like

```
Lemma procedure_WF_no_passive_com : Procedures_WF _ D →
  passive p (snd (D X)) → passive q (snd (D X)) →
  ⟦D, snd (D X) | q ⟧ == (@Recv Sig' p x a B1) →
  ⟦D, snd (D X) | p ⟧ == (@Send Sig' q e a' B2) → False.
```

The proof of soundness then invokes `passive_bridge` to find a procedure definition satisfying the premises of this (or a similar) lemma.

7 Discussion, conclusions and future work

The most debatable design option in our formalisation is the splitting of projection in two functions. The main advantage is that this duplication allows us to reuse the pre-existing development more easily – the new cases for projecting conditionals would significantly complicate the proof of soundness, for example. Furthermore, since `bproj` and `pproj` coincide for procedure definitions, many results need only be proved for one of them anyway.

Having only one projection function significantly increases the complexity of the development, and yields a less intuitive notion for the main choreography. Furthermore, many lemmas included corner cases that could never arise from executing initial choreographies – but a syntactic characterisation of such choreographies turned out to be extremely nontrivial. The current choice thus leads to a cleaner formalisation, and in hindsight acknowledges that procedure definitions and the main choreography are slightly different in their handling of muted processes.

Our new theory serves as a useful additional tool for choreographic programmers, allowing us to write the choreography from Example 1 in the more intuitive way shown at the end of the Introduction. This extended choreographic language still enjoys deadlock freedom; additionally, it allows programmers to express *livelocks*, i.e. scenarios where some processes are not able to make progress. For example, in Example 1 `Server` is not able to make progress unless `IP` sends the token to `Client`; thus, it could be waiting forever. This kind of livelocks was not possible before, since repeating communication structures required recursion with all processes involved being notified at every iteration.

Our language is a pure extension of the previous one, as any old choreography can be trivially written in the new language by adding empty lists of muted processes to all procedure calls. Furthermore, inspection of the semantics shows that the changes have no impact on these choreographies, while the new well-formedness requirements only affect muted processes.

Likewise, projection (`bproj`) is unchanged except when dealing with muted calls. This implies that previous results like Turing-completeness [22] still hold for the extended language.

Our whole development was done using Coq. This means that not only it was formalised in Coq, but the theorem prover was actively used as a research tool. As suggested in the text, well-formedness requirements were introduced in three stages – immediately after extending the syntax, when defining projection, and when proving the EPP theorem. In several cases, these arose from obscure corner cases in the proofs some of the more technical lemmas where Coq was instrumental in detecting counter-examples that might otherwise have gone unnoticed.

In the future, we would like to adapt the notion of amendment [20] to our more robust projection, so that a choreography like that in Example 1 can be automatically repaired taking advantage of the fact that muted processes do not need to be informed of the outcomes of all conditionals. Dually, such a procedure could also detect that some processes could be muted in some procedure calls, further simplifying the final result.

We would also like to relax the definition of well-formedness to increase the expressiveness of projectable choreographies further, allowing for a more complex structure of procedure calls with muted processes in nested conditionals.

Lastly, it would also be interesting to apply our development to other choreographic theories, e.g. multiparty sessions types.

Our development can be downloaded from https://doi.org/10.5281/zenodo.7746646.

Acknowledgements. We thank the anonymous reviewers for their useful comments, which helped us improve the quality of this article. This work was partially supported by Villum Fonden, grants 29518 and 50079, and Independent Research Fund Denmark, grant 0135-00219.

References

[1] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. *IEEE Trans. Software Eng.*, 29(7):623–633, 2003.

[2] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of MSC graphs. *Theor. Comput. Sci.*, 331(1):97–114, 2005.

[3] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2–3):95–230, 2016.

[4] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In John Field and Michael Hicks, editors, *Procs. POPL*, pages 191–202. ACM, 2012.

[5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. Springer, 2004.

[6] Mario Bravetti and Gianluigi Zavattaro. Choreographies and behavioural contracts on the way to dynamic updates. In Marcello Maria Bersani, Davide Bresolin, Luca Ferrucci, and Manuel Mazzara, editors, *Procs. MOD**, volume 168 of *EPTCS*, pages 12–31, 2014.

[7] Alessandro Bruni, Marco Carbone, Rosario Giustolisi, Sebastian Mödersheim, and Carsten Schürmann. Security protocols as choreographies. In Daniel Dougherty, José Meseguer, Sebastian Alexander Mödersheim, and Paul D. Rowe, editors, *Protocols, Strands, and Logic*, volume 13066 of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2021.

[8] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration conformance for system design. In Paolo Ciancarini and Herbert Wiklicky,

editors, *Procs. COORDINATION*, volume 4038 of *Lecture Notes in Computer Science*, pages 63–81. Springer, 2006.

[9] Marco Carbone, Luís Cruz-Filipe, Fabrizio Montesi, and Agata Murawska. Multiparty classical choreographies. In Fred Mesnard and Peter J. Stuckey, editors, *Procs. LOPSTR*, volume 11408 of *Lecture Notes in Computer Science*, pages 59–76. Springer, 2019.

[10] 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.

[11] 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.

[12] 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.

[13] Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. *Distributed Comput.*, 31(1):51–67, 2018.

[14] Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. *Acta Informatica*, 54(3):243–269, 2017.

[15] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Log. Methods Comput. Sci.*, 8(1), 2012.

[16] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Event structure semantics for multiparty sessions. *J. Log. Algebraic Methods Program.*, 131:100844, 2023.

[17] Alex Coto, Roberto Guanciale, and Emilio Tuosto. An abstract framework for choreographic testing. *J. Log. Algebraic Methods Program.*, 123:100712, 2021.

[18] Luís Cruz-Filipe, Lovro Lugović, and Fabrizio Montesi. Certified compilation of choreographies with hacc. *CoRR*, abs/2303.03972, 2023. Accepted for publication in *Procs. FORTE*.

[19] Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020.

[20] Luís Cruz-Filipe and Fabrizio Montesi. Now it compiles! Certified automatic repair of uncompilable protocols. *CoRR*, abs/2302.14622, 2023. Accepted for publication in *Procs. ITP*.

[21] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Procs. ICTAC*, volume 12819 of *LNCS*, pages 115–133. Springer, 2021.

[22] 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.

[23] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *Journal of Automated Reasoning*, Accepted for publication.

[24] Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in rust with multiparty session types. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *Procs. PPoPP*, pages 246–261. ACM, 2022.

[25] Francesco Dagnino, Paola Giannini, and Mariangiola Dezani-Ciancaglini. Deconfined global types for asynchronous sessions. *Log. Methods Comput. Sci.*, 19(1), 2023.

[26] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Log. Methods Comput. Sci.*, 13(2), 2017.

[27] Saverio Giallorenzo, Fabrizio Montesi, and Maurizio Gabbrielli. Applied choreographies. In Christel Baier and Luís Caires, editors, *Procs. FORTE*, volume 10854 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2018.

[28] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Object-oriented Choreographic Pro-

gramming. *CoRR*, abs/2005.09520, 2020.

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

[30] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[31] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016. Also: POPL, pages 273–284, 2008.

[32] Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In Marieke Huisman and Julia Rubin, editors, *Procs. FASE*, volume 10202 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2017.

[33] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.

[34] Intl. Telecommunication Union. Recommendation Z.120: Message Sequence Chart, 1996.

[35] 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.

[36] 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.

[37] 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.

[38] Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. Enforcing availability in failure-aware communicating systems. In Elvira Albert and Ivan Lanese, editors, *Procs. FORTE*, volume 9688 of *Lecture Notes in Computer Science*, pages 195–211. Springer, 2016.

[39] Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. Generalising projection in asynchronous multiparty session types. In Serge Haddad and Daniele Varacca, editors, *Procs. CONCUR*, volume 203 of *LIPIcs*, pages 35:1–35:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[40] Fabrizio Montesi. *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen, 2013.

[41] Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023.

[42] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with Jolie. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Web Services Foundations*, pages 81–107. Springer, 2014.

[43] Fabrizio Montesi and Janine Weber. From the decorator pattern to circuit breakers in microservices. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *Procs. SAC*, pages 1733–1735. ACM, 2018.

[44] Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In Pedro R. D'Argenio and Hernán C. Melgratti, editors, *Procs. CONCUR*, volume 8052 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2013.

[45] Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In Peter Thiemann and Robby Bruce Findler, editors, *Procs. ICFP*, pages 115–126. ACM, 2012.

[46] Michael Nygard. *Release It! Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

[47] Object Management Group. Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0/, 2011.

[48] 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.

[49] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *Procs. WWW*, pages 973–982. ACM, 2007.

[50] Gan Shen, Shun Kashiwa, and Lindsey Kuper. Haschor: Functional choreographic programming for all (functional pearl). *CoRR*, abs/2303.00924, 2023.

[51] Vasco T. Vasconcelos, Francisco Martins, Hugo-Andrés López, and Nobuko Yoshida. A type discipline for message passing parallel programs. *ACM Trans. Program. Lang. Syst.*, 44(4):26:1–26:55, 2022.