Journal of Automated Reasoning (2023) 67:21
https://doi.org/10.1007/s10817-023-09665-3

A Formal Theory of Choreographic Programming

Luís Cruz-Filipe[1] · Fabrizio Montesi[1] · Marco Peressotti[1]

Received: 7 September 2022 / Accepted: 9 April 2023 / Published online: 27 May 2023

Abstract
Choreographic programming is a paradigm for writing coordination plans for distributed systems from a global point of view, from which correct-by-construction decentralised implementations can be generated automatically. Theory of choreographies typically includes a number of complex results that are proved by structural induction. The high number of cases and the subtle details in some of these proofs has led to important errors being found in published works. In this work, we formalise the theory of a choreographic programming language in Coq. Our development includes the basic properties of this language, a proof of its Turing completeness, a compilation procedure to a process language, and an operational characterisation of the correctness of this procedure. Our formalisation experience illustrates the benefits of using a theorem prover: we get both an additional degree of confidence from the mechanised proof, and a significant simplification of the underlying theory. Our results offer a foundation for the future formal development of choreographic languages.

Keywords Choreographic programming · Theorem proving · Concurrency · Process calculi

1 Introduction

In the setting of concurrent and distributed systems, choreographic languages are used to define interaction protocols that communicating processes should abide by [32, 43, 46]. These languages are akin to the "Alice and Bob" notation found in security protocols, and inherit the key idea of making data communication manifest in programs [42]. This is usually obtained through a linguistic primitive like $\texttt{Alice.e} \rightarrow \texttt{Bob.x}$, read "`Alice` communicates the result of evaluating expression `e` to `Bob`, which stores it in its local variable `x`".

In recent years, the communities of concurrency theory and programming languages have been prolific in developing methodologies based on choreographies, yielding results in pro-

✉ Luís Cruz-Filipe
lcf@imada.sdu.dk

Fabrizio Montesi
fmontesi@imada.sdu.dk

Marco Peressotti
peressotti@imada.sdu.dk

1 Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark

gram verification, monitoring, and program synthesis [2, 31]. For example, in *multiparty session types*, types are choreographies used for checking statically that a system of processes implements protocols correctly [30]. Further, in *choreographic programming*, choreographic languages are elevated to full-fledged programming languages [40], which can express how data should be pre- and post-processed by processes (encryption, validation, anonymisation, etc.).

Choreographic programming languages come with a procedure known as *Endpoint Projection* (*EPP*), which automatically synthesises executable code for each process described in a choreography, with the guarantee that executing these processes together implements the communications prescribed in the choreography [7, 8]. These languages showed promise in a number of contexts, including parallel algorithms [11], cyber-physical systems [28, 37, 38], self-adaptive systems [23], system integration [27], information flow [36], and the implementation of security protocols [28].

EPP involves three elements: the source choreographic language, the target process language, and the compiler. The interplay between these components, where a single instruction at the choreographic level might be implemented by multiple instructions in the target language, makes the theory of choreographic programming error-prone: for even simpler approaches, like abstract choreographies without computation, it has been recently discovered that a few proofs published in peer-reviewed articles do not hold and their theories required adjustments. While in most cases these adjustments amounted to correcting small details in proofs or deal with missing cases, there were situations that required finding new proof strategies or reformulating statements [39, 45]. In exceptional situations, it has been discovered that key results actually did not hold [3–5, 25, 35].

This article presents a formalisation of a core theory of choreographic programming in the theorem prover Coq, the process of developing this formalisation, the challenges encountered, and how tackling these challenges led to improvements of the original theory.

A note on the process. We argue that computer-aided verification can be successfully applied to the study of choreographies and to provide solid foundations for future developments. To substantiate this claim, we summarise the story behind this article, which illustrates how interactive theorem proving can do more than just checking what we already know.

Our starting point was the theory of Core Choreographies (CC), a minimalistic language that the first two authors previously proposed for the study of choreographic programming [16]. CC includes only the essential features of choreographic languages and minimal computational capabilities at processes (computing the successor of a natural number and deciding equality of two natural numbers), yet it is expressive enough to be Turing complete.

We started formalising CC in Coq in late 2018. In mid-2019, we gave an informal progress report on the promising status of the formalisation at the TYPES conference [19]. Unfortunately, we soon stumbled upon an unexpected source of complexity for the formalisation: a set of term-rewriting rules for a precongruence relation used in the semantics of the language for (i) expanding procedure calls and (ii) reshuffling independent communications to model concurrent execution. In addition to being time consuming, reasoning with precongruence systematically made the formalisation significantly more complicated than the development in [16] (for a more technical discussion, see Sect. 3.5).

At the time, the second author was responsible for a Master course on theory of choreography for students in Computer Science. It quickly became apparent that the technical aspects (including, but not only, structural precongruence) that complicated the formalisation of CC were also the most challenging for the students. This observation led that author to develop an alternative theory of CC for his course material that dispenses with these problematic

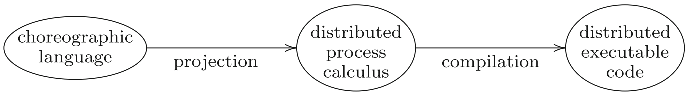

Fig. 1 Two-stage compilation process from choreographies to executable code

notions without changing its essence [41]. The formalisation in this article uses this revised choreography theory.

Thus, our work also shows that theorem proving can be used in research: the insights obtained while doing this formalisation led to changes in the original theory. We show that this did not come at the cost of expressive power: the original proof of Turing completeness from [16] still works for the theory in [41] without essential changes [21]. Furthermore, formalising the theory also allowed us to identify unnecessary assumptions in some lemmas, yielding stronger results.

Publication history and contribution. As mentioned previously, a first informal progress report on this formalisation was presented at the TYPES conference in 2019 [19], following an approach that later turned to be unfeasible. The first formalisation of the choreographic language, including the proof of Turing completeness, was presented in [21], while the formalisation of EPP appeared originally in [20]. The current presentation discusses an updated formalisation, which (i) no longer uses Coq's module system and (ii) differs significantly in the treatment of partial functions, which significantly simplifies the definition of EPP. We do not discuss the formalisation of the proof of Turing completeness, as this is essentially unchanged from [21]. Instead, we place a stronger emphasis on the formalisation challenges compared to the works cited.

The direct result of our work is a formalisation that can be used as a basis for future work on choreographic programming, both in theory and in practice. Subsequent developments already include a formalisation of choreography repair [17], a more flexible notion of projection allowing for livelocks [15], and a toolchain for generating executable code from choreographies [14]. These developments capitalise on the current contribution in different ways, showing that the current formalisation is reusable, extendable, and amenable to be incorporated in tools for software development.

Furthermore, our formalisation dispels any concerns that there may be regarding the correctness of our results—which is especially relevant in an area where many proofs are extremely technical and tedious both to write down and to check in detail.

Lastly, we provide more evidence to substantiate the general claim that interactive theorem proving is a valid tool for conducting research in theoretical computer science, by showing that formalising a state-of-the-art theoretical development is feasible and can provide valuable insights that help improve the theoretical development.

The big picture. This work is the first step towards a more ambitious goal: the development of a certified framework for choreographic programming. At a later stage, we plan on developing compilers that can translate the process implementations generated by EPP into executable code in different programming languages (see Fig. 1). This would yield end-to-end compilation from choreographies to actual executable code.

Our goal motivated two important design choices in the current work that are not present in [20, 21]. First, we want to extract a correct implementation of EPP from our formalisation.

This motivated us to move away from Coq's module system, as we found the Haskell code generated by extraction to be unidiomatic. Second, we introduce the possibility of annotating terms in choreographies with data that may be needed for (second-stage) compilation to executable programming languages.

Our language has two characteristics that are inherited from the choreographic model in [16]. First, the semantics that we present in Sect. 3.3 is synchronous. This is a standard choice for choreographic models, as it makes the development simpler; the interested reader is referred to [12] for a lengthier discussion on asynchronous models. The second choice is that we assume a fixed set of labels with only two elements (see Sect. 3.2). This is again a standard choice in theoretical developments, as any label from a fixed finite set of labels can be encoded by a sequence of labels from a two-element set, but it leads to inefficiency in practical applications. Formalising a more general theory in Coq poses complex technical challenges, as discussed in Sect. 5.5.

Structure. A full understanding of the more technical details of our formalisation benefits from some background knowledge on choreographies. For convenience, Sect. 2 features a short introduction to the main intuitions and results of choreography theory, which can be skipped by readers familiar with the topic. Our choreographic language (syntax and semantics) is presented together with its Coq formalisation in Sect. 3, where it is also shown that it enjoys the usual properties of choreographic languages. Section 4 defines the target process language, together with its semantics. EPP is formalised in Sect. 5, and its soundness and completeness are discussed in Sect. 6. We review related formalisation efforts in Sect. 7, before concluding in Sect. 8.

Our development was made using Coq 8.13.2. The source code is available at [22].

2 Background: Choreographic Languages and Endpoint Projection

In this section we describe the language of Simple Choreographies [41], which introduces the basic principles of choreographies and EPP. We include this material to make our development accessible to the reader not familiar with the topic, but it is not directly used in our development.

2.1 Simple Choreographies

Simple Choreographies can express finite sequences of communications between processes. Processes are identified by names (p, q, etc.). Choreographies, ranged over by C, are constructed according to the following grammar.

$$C ::= \mathsf{p} \to \mathsf{q}; C \mid \mathbf{0}$$

A choreography $\mathsf{p} \to \mathsf{q}; C$ represents a communication from a process p to a process q with continuation C; $\mathbf{0}$ is the terminated choreography. We omit trailing $\mathbf{0}$s in examples.

Example 1 (*Ring protocol* [41]) The choreography below describes a ring protocol among three participants: Alice communicates to Bob; then Bob communicates to Carol; and finally Carol communicates back to Alice.

$$\mathsf{Alice} \to \mathsf{Bob}; \mathsf{Bob} \to \mathsf{Carol}; \mathsf{Carol} \to \mathsf{Alice} \tag{1}$$

The semantics of Simple Choreographies is given as the labelled transition system induced by the rules displayed in Fig. 2. Transition labels have the form $\mathsf{p} \to \mathsf{q}$, allowing for observing the communications performed by a choreography.

Fig. 2 Semantics of simple choreographies

$$\frac{}{\mathsf{p} \to \mathsf{q}; C \xrightarrow{\mathsf{p}\to\mathsf{q}} C}\ \textsc{com} \qquad \frac{C \xrightarrow{\mathsf{r}\to\mathsf{s}} C' \quad \{\mathsf{p},\mathsf{q}\} \# \{\mathsf{r},\mathsf{s}\}}{\mathsf{p} \to \mathsf{q}; C \xrightarrow{\mathsf{p}\to\mathsf{q}} \mathsf{p} \to \mathsf{q}; C'}\ \textsc{delay}$$

Rule com models the execution of a communication at the beginning of a choreography. Rule delay, instead, allows for performing a transition within the continuation of a choreography, provided that the transition does not involve any of the processes in preceding instructions. This rule captures the fact that processes run independently of each other, and thus choreographic instructions can be executed out-of-order. The independence requirement is captured by the the side-condition $\{\mathsf{p}, \mathsf{q}\} \# \{\mathsf{r}, \mathsf{s}\}$, where # relates disjoint sets.

Example 2 (*Ring protocol, continued* [41]) Let C be the choreography in (1). Then, by rule com, we have the following chain of transitions.

$$C \xrightarrow{\mathsf{Alice}\to\mathsf{Bob}} \mathsf{Bob} \to \mathsf{Carol}; \mathsf{Carol} \to \mathsf{Alice} \xrightarrow{\mathsf{Bob}\to\mathsf{Carol}} \mathsf{Carol} \to \mathsf{Alice} \xrightarrow{\mathsf{Carol}\to\mathsf{Alice}} \mathbf{0}$$

These communications cannot be executed out-of-order, because of the chain of causality between them: each instruction involves a process that needs to participate in a previous instruction.

Example 3 Consider now the choreography (inspired from the factory examples in [41]), which models a system where two "ordering" processes o_1 and o_2 independently communicate two respective orders to the servers s_1 and s_2.

$$\mathsf{o}_1 \to \mathsf{s}_1; \mathsf{o}_2 \to \mathsf{s}_2 \tag{2}$$

The following derivation shows that $\mathsf{o}_2 \to \mathsf{s}_2$ can be executed first.

$$\frac{\dfrac{}{\mathsf{o}_2 \to \mathsf{s}_2 \xrightarrow{\mathsf{o}_2\to\mathsf{s}_2} \mathbf{0}}\ \textsc{com} \qquad \{\mathsf{o}_1, \mathsf{s}_1\} \# \{\mathsf{o}_2, \mathsf{s}_2\}}{\mathsf{o}_1 \to \mathsf{s}_1; \mathsf{o}_2 \to \mathsf{s}_2 \xrightarrow{\mathsf{o}_2\to\mathsf{s}_2} \mathsf{o}_1 \to \mathsf{s}_1}\ \textsc{delay}$$

2.2 Simple Processes

Implementations of Simple Choreographies are modelled in a process language called Simple Processes [41]. First, we define a grammar for writing process behaviours.

$$P, Q, R := \mathsf{p}!; P \mid \mathsf{p}?; P \mid \mathbf{0}$$

These actions are the local counterparts to the communication action in choreographies. A send action $\mathsf{p}!$ sends a message to a process p, and the dual receive action $\mathsf{p}?$ receives a message from a process p. The term $\mathbf{0}$ is the terminated process.

Processes are composed into networks (N, M, etc.), which are maps from process names to processes. We introduce some notation: $\mathbf{0}$ is the terminated network, where all process names are mapped to $\mathbf{0}$; $\mathsf{p}[P]$ is the network where p is mapped to P and all other process names are mapped to $\mathbf{0}$; and $N \mid M$ ("N parallel M") is the union of N and M, assuming that their supports[1] are disjoint. Under extensional equality of functions, the set of networks equipped with parallel composition forms a partial commutative monoid with $\mathbf{0}$ as identity element: $N \mid \mathbf{0} = N$, $N \mid M = M \mid N$, and $N_1 \mid (N_2 \mid N_3) = (N_1 \mid N_2) \mid N_3$ [41].

[1] The support of a network is the set of all processes not mapped to $\mathbf{0}$.

Fig. 3 Semantics of simple processes

$$\frac{}{\mathsf{p}[\mathsf{q}!; P] \mid \mathsf{q}[\mathsf{p}?; Q] \xrightarrow{\mathsf{p}\to\mathsf{q}} \mathsf{p}[P] \mid \mathsf{q}[Q]}\ \textsc{com} \qquad \frac{N \xrightarrow{\mathsf{p}\to\mathsf{q}} N'}{N \mid M \xrightarrow{\mathsf{p}\to\mathsf{q}} N' \mid M}\ \textsc{par}$$

Fig. 4 Process projection for simple choreographies

$$[\![\mathsf{p} \to \mathsf{q}; C]\!]_{\mathsf{r}} = \begin{cases} \mathsf{q}!; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}?; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases} \qquad [\![\mathbf{0}]\!]_{\mathsf{r}} = \mathbf{0}$$

Example 4 The following network implements the choreography in (1).

$$\mathsf{Alice}[\mathsf{Bob}!; \mathsf{Carol}?] \mid \mathsf{Bob}[\mathsf{Alice}?; \mathsf{Carol}!] \mid \mathsf{Carol}[\mathsf{Bob}?; \mathsf{Alice}!] \tag{3}$$

The semantics of Simple Processes is given by the transition rules in Fig. 3. Rule COM synchronises processes with matching send and receive actions. Rule PAR allows for parallel execution.

Example 5 The transitions of the choreography in (1) coincide with those of the network in (3). Technically, the labelled transition systems generated by the choreography and the network are isomorphic, showing that the network is indeed a precise implementation of the choreography.

Example 6 Out-of-order execution for choreographies corresponds to parallelism at the level of networks. The following network implements the choreography in (2).

$$\mathsf{o}_1[\mathsf{s}_1!] \mid \mathsf{o}_2[\mathsf{s}_2!] \mid \mathsf{s}_1[\mathsf{o}_1?] \mid \mathsf{s}_2[\mathsf{o}_2?] \tag{4}$$

Using rule PAR and the monoidal structure of parallel composition, the network can start by executing either the communication between o_1 and s_1 or the one between o_2 and s_2.

2.3 Endpoint Projection

In general, writing correct implementations of protocols is hard, especially for more expressive choreographic languages as the one that we use later in this article. Endpoint projection (EPP) is a mechanical procedure for translating choreographies into networks by splitting choreographic terms into their local counterparts [7, 8, 16, 30, 41]. The idea is that given a choreography C and a process p, we first compute the process term $[\![C]\!]_{\mathsf{p}}$ that implements the actions that p should perform to implement its part in C. Then, EPP is defined as the parallel composition of all such terms.

In the case of Simple Choreographies and Simple Processes, the process projection map $[\![C]\!]_{\mathsf{p}}$ is defined in a natural way by the recursive equations in Fig. 4. In particular, a communication term $\mathsf{p} \to \mathsf{q}; C$ is projected to a send action and the projection of the continuation if we are projecting the sender (first case), a receive action and the projection of the continuation if we are projecting the receiver, or just the projection of the continuation if we are projecting a process that is not involved in the communication.

Given a choreography C, its EPP $[\![C]\!]$ is defined as the network $[\![C]\!](\mathsf{p}) = [\![C]\!]_{\mathsf{p}}$. This network is a correct implementation of C.

Theorem 1 (Correctness of EPP [41]) *The following statements hold for every choreography C and transition label μ in the language of Simple Choreographies.*

Completeness. *For any C', if $C \xrightarrow{\mu} C'$ then $[\![C]\!] \xrightarrow{\mu} [\![C']\!]$.*
Soundness. *For any N, if $[\![C]\!] \xrightarrow{\mu} N$ then $C \xrightarrow{\mu} C'$ for some C' such that $N = [\![C']\!]$.*

Example 7 The networks in (3) and (4) are, respectively, the EPPs of the choreographies in (1) and (2).

The completeness part of Theorem 1 is proven by case analysis on the transition performed by the choreography. This gives information about the choreography's structure that can be used to infer the shape of the network, which in turn shows that it can perform the corresponding transition. The soundness part is by induction on the choreography, with two cases depending on whether the transition requires applying the delay rule (and invoking the induction hypothesis).

2.4 Taking Stock

Theorem 1 is used to prove other notable results given by the choreographic approach, such as *deadlock-freedom*. A deadlocked network is one that is not terminated but cannot make any transitions, typically because all processes are waiting for someone else. Even in a simplistic process language such as Simple Processes, we can write deadlocked networks, such as:

$$\mathsf{p}[\mathsf{q}?] \mid \mathsf{q}[\mathsf{p}?].$$

Here, p and q are both waiting for each other, and therefore the network will never be able to proceed.

Since communication terms in choreographies specify simultaneously what sender and receiver processes are involved, choreographies cannot describe deadlocks, a property known as *deadlock-freedom by design* [7]. As a consequence of Theorem 1, the networks generated by EPP can never become deadlocked.

The choreographic language that we consider in the rest of our article is more expressive than Simple Choreographies, as it includes features that are important for modelling realistic protocols. However, the general structure of the development follows the roadmap given in this section, albeit with a much higher level of complexity.

3 Core Choreographies

We introduce Core Choreographies (CC), the choreographic language that we work with, and its formalisation. At the end of this section, we discuss how the formalisation process guided the evolution of the language from its original presentation in [16] to its present form, which is closer to the style of [41].

In CC, processes can perform point-to-point communications and have storage. Communicated messages can be either values, which are computed by evaluating local expressions, or labels (tags, or constants) from a fixed set {`left`, `right`}.[2] Additionally, choreographies can include conditionals based on Boolean expressions and invoke recursive procedures.

3.1 Preliminaries

Choreographies are parameterised by a signature, which defines the types for process names (processes for short) `pid`, local variables `var` (used to access the processes' storage), values `val`, expressions `expr`, Boolean expressions `bexpr`, and procedure names `recvar` (from

[2] Restricting the set of labels to two elements is standard practice [6, 16]. The practical implications were discussed in the Introduction (p. 4); for a discussion on how this simplifies the formalisation, see Sect. 5.5.

recursion variables). Signatures also include types for (user-defined) annotations ann (as discussed in Sect. 1). Since the types of expressions and values are parameters, signatures also need to specify the evaluation functions mapping expressions to values and Boolean expressions to Booleans. We fix a signature Sig and introduce abbreviations Pid:= (pid Sig) and similarly for all other parameters for convenience.

All datatypes except the evaluation functions are equipped with a decidable equality. Since we are targetting extraction, which is not compatible with modules, we reimplemented DecType as a record type consisting of exactly these two components, and reproved the lemmas about decidable equality from the Coq standard library. We also show that the Cartesian product of two DecTypes can be made into a DecType, and we define a two-element decidable type Label whose elements are the two labels left and right.

Evaluation functions are again records. The first element is a function that takes an expression and a mapping from a process's variables to values, and returns a value (possibly of a different type as the one stored locally). The second element is a proof that the value returned by evaluation does not change if the mapping from variables to values is replaced by an extensionally equivalent one.

The type State:= Pid $\rightarrow$ Var $\rightarrow$ Value models the memory state of the set of all processes.[3] We define extensional equality on states, written [==], and prove that it is an equivalence relation. Furthermore, we define an operation s[[p,x $\Rightarrow$ v]] for updating the state s with the assignment of value v to process p's variable x, and prove a number of useful rewriting lemmas.

3.2 Syntax

Choreographies are defined inductively by the following grammar.[4]

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

Here, p,q:Pid are processes, e:Expr is an expression, x:Var is a variable, l:Label is a label, a:Ann is an annotation, b:BExpr is a Boolean expression, X:RecVar is a procedure name, and ps:list Pid is a list of processes.

The terms denoted η are called *interactions*; for many results, it is convenient that they form their own type. Term p#e $\longrightarrow$ q\$x is a value communication, where p communicates the result of evaluating e to q, which stores it in its local variable x. Term p $\longrightarrow$ q[l] is a label selection, where p communicates label l to q.

Label selections are used in conjunction with conditionals. In a conditional If p ?? b Then C1 Else C2, the evolution of the choreography is determined by the outcome of evaluating the Boolean expression b at p. Other processes that need to know which branch was chosen (*knowledge of choice* [9]) can get this information through the reception of label left or right from p.

Interactions are paired with annotations (a), which are ignored by the semantics. They are meant to include additional information that may be needed in subsequent processing steps, such as documentation or the second-stage compilation mentioned in Sect. 1. We omit annotations in all our examples.

[3] In the formalisation, the type Var $\rightarrow$ Value is given the name LState (for "local state"), but since local states are unused elsewhere we do not discuss them here.

[4] Throughout this article, we use the pretty-printing rules defined in the Coq formalisation so that the correspondence between the informal mathematical presentation and the formal results is clear.

Term `Call X` invokes the procedure named `X`. A procedure may involve several processes, and the semantics of CC allows each process to join the procedure only when needed. The *runtime* term `RT_Call X ps C` represents this intermediate situation: execution of procedure `X` has already evolved to `C`, but the processes in `ps` have not yet joined it. Runtime terms are not meant to be written by programmers: they are auxiliary terms generated by the semantics.

The grammar of choreographies is defined as the following inductive types.

```
Inductive Eta : Type :=
| Com : Pid → Expr → Pid → Var → Eta
| Sel : Pid → Pid → Label → Eta.

Inductive Choreography : Type :=
| Interaction : Eta → Ann → Choreography → Choreography
| Cond : Pid → BExpr → Choreography → Choreography → Choreography
| Call : RecVar → Choreography
| RT_Call : RecVar → (list Pid) → Choreography → Choreography
| End : Choreography.
```

A set of procedure definitions, formalised as type `DefSet`, is a mapping assigning to each `RecVar` a list of processes and a choreography; intuitively, the list contains the processes that are used in the procedure. A `Program` is a pair containing a set of procedure definitions and the choreography to be executed at the start, also called the *main* choreography.

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

We write `Procedures P` and `Main P` for, respectively, the set of procedure definitions and the main choreography in a program `P` (so `Procedures` and `Main` are simply aliases for the corresponding projections). Likewise, `Vars P X` and `Procs P X` denote the list of processes and the definition of a particular procedure `X` within `P`. Finally, `Names D` is the function mapping each variable `X` to the set of processes that it uses according to `D:DefSet`.

Example 8 (*Distributed Authentication*) The choreography `C1` below describes a multiparty authentication scenario where an identity provider `ip` authenticates a client `c` to server `s`. (For convenience, we name some of the subterms in the choreography.)

```
 C1 := c#credentials ⟶ ip$x;; If ip ?? (check x) Then C1t Else C1e
C1t := ip ⟶ s[left];; ip ⟶ c[left];; s#token ⟶ c$t;; End
C1e := ip ⟶ s[right];; ip ⟶ c[right];; End
```

`C1` starts with `c` communicating its `credentials` to `ip`, which stores them in `x`. Then, `ip` checks whether the received credentials are valid by evaluating the Boolean expression `check x`, and signals the result to `s` and `c` by selecting `left` when the credentials are valid (`C1t`) and `right` otherwise (`C1e`). In the first case, the server communicates a `token` to `c`, otherwise the choreography simply terminates.

The selections from `ip` to `s` and `c` address knowledge of choice, as previously described.

Well-formedness. There are a number of well-formedness requirements on choreographies, which can be grouped in three categories.

1. Intended use of choreographies. Interactions must have distinct processes (there are no self-communications), e.g., `p#e ⟶ p$x` is disallowed.
2. Intended use of runtime terms. Procedure definitions may not contain runtime terms. `Main P` may include subterms `RT_Call X ps C`, but `ps` must be nonempty and include only process names that occur in `Vars P X`.

3. Design choices in the formalisation. The processes in `Vars X` include all processes that are used in `Procs X`.

Well-formedness is essential in the proof of correctness of EPP (Sect. 6).

We start by formalising the different properties of choreographies separately:

– `initial C` holds if `C` does not contain runtime terms (`RT_Call`);
– `no_self_comm C` holds if `C` contains no self-communications;
– `no_empty_ann C` holds if all runtime terms in `C` have nonempty lists of process names.

These properties are defined recursively over `C` in the natural way. Well-formedness of choreographies `Choreography_WF` is defined as the conjunction of the last two properties.

Well-formedness of programs also takes into account the additional requirements on the lists of processes annotating runtime terms. Specifically, in a program `P`, the choreography `Main P` must be consistently annotated with respect to `Vars P`: in any subterm `RT_Call X ps C'` in `Main P`, the list `ps` only contains processes appearing in `Vars P X`. This property is written as `consistent (Vars P) (Main P)`, where predicate

```
consistent: (RecVar → list Pid) → Choreography → Prop
```

is defined inductively in the expected way. Also, the set of procedure definitions in `P` must be well-annotated: if `Procedures P X=(ps,C)`, then the set of processes used in `C` must be a nonempty subset of `ps`,[5]

```
Definition well_ann (P:Program) (X:RecVar) : Prop :=
 Vars P X ≠ nil ∧ CCC_pn (Procs P X) (Vars P) [C] Vars P X.
```

The last definition uses function `CCC_pn`, which computes the set of processes occurring in a choreography, given the set of processes used in each procedure. It generalises to `CCP_pn`, which computes the set of processes occurring in a well-annotated program.

Using these ingredients, we define well-formedness of programs as follows.

```
Definition Program_WF (P:Program) : Prop :=
 Choreography_WF (Main P) ∧ consistent (Vars P) (Main P) ∧
 ∀ X, no_self_comm (Procs P X) ∧ initial (Procs P X) ∧ well_ann P X.
```

Since `initial` choreographies do not include runtime terms, this definition also implies that all procedure definitions are well-formed.

Example 9 Let `Defs:DefSet` map `FileTransfer` to the pair consisting of the process list `c:: s:: nil` and the following choreography.

```
s.(file, check) ⟶ c.x;;        (* send file and check data *)
If c.(crc(fst(x)) == snd(x))   (* cyclic redundancy check *)
 Then c ⟶ s[left];; End        (* file received correctly, end *)
 Else c ⟶ s[right];; Call FileTransfer (* errors detected, retry *)
```

`FileTransfer` describes a file transfer protocol between a server `s` and a client `c` using Cyclic Redundancy Checks (`crc`) to detect errors from a noisy channel.

Assuming that `Defs` maps all other procedure definitions to `End`, the program `P=(Defs,Call FileTransfer)` satisfies `Program_WF P`.

Recall that our long-term future goal is to apply program extraction to this formalisation, and then use the result in tools. Many of the results that we show later only hold for well-formed programs, and any tool built on our theory should be able to validate that its input is

[5] We defined suggestive notations [C] [U], [\] and [#] for the set operations we use.

well-formed. However, due to the quantification over all procedure names, well-formedness of programs is in general not decidable. In practice, though, choreographic programs only use a finite number of procedures; if these are known, well-formedness becomes decidable.

This observation motivates the definition of a recursive predicate

```
used_procedures_C : Choreography → list RecVar → Prop
```

such that `used_procedures_C C Xs` holds iff `C` only calls procedures in `Xs` (directly). This is generalised to programs by requiring that all procedures in `Xs` also satisfy the same property, and additionally that all procedures not in `Xs` be defined as `End`.

```
Definition used_procedures (P:Program) (Xs:list RecVar) :=
 used_procedures_C (Main P) Xs ∧
 ∀ X, (In X Xs → used_procedures_C (Procs P X) Xs)
   ∧ (~In X Xs → Procs P X = End ∧ Vars P X ≠ nil).
```

The requirement `Vars P X ≠ nil` for procedures not in `Xs` is included to ensure well-formedness. From this, we can prove decidability of well-formedness.

```
Lemma Program_WF_dec : ∀ P Xs, used_procedures P Xs →
  {Program_WF P} + {~Program_WF P}.
```

Applying this lemma in extracted code requires knowing a suitable set `Xs`. While we cannot automatically verify that this set satisfies `used_procedures P Xs`, it is very reasonable to trust that a correct one has been provided: typically, the relevant procedures used in a program are written down explicitly, making it straightforward to list them.

An alternative approach would be requiring the set of procedure names to be finite. This is closer in spirit to the pen-and-paper presentations of choreographic languages—even if procedure names are taken from an infinite set, only a finite number of them can be used in a concrete program [16]. We chose the present approach for simplicity, as working with finite sets in Coq is notoriously cumbersome.

3.3 Semantics

The semantics of CC is defined by means of labelled transition systems, in three layers. At the lowest layer, we define the transitions that a choreography can make (`CCC_To`), parameterised by a set of procedure definitions; then we pack these transitions into the more usual presentation—as a labelled relation `CCP_To` on *configurations* (pairs program/state). Finally, we define multi-step transitions `CCP_ToStar` as the transitive and reflexive closure of the transition relation. This layered approach makes proofs about transitions cleaner, allowing us to separate the different levels of induction.

Transition labels. Each layer of the semantics has its type of transition labels. For the lower level, we define an inductive type `RichLabel` whose constructors reflect the possible actions a choreography can take: value communications, label selections, reducing a conditional, or locally joining a procedure call.

The second layer uses the type `TransitionLabel` of labels corresponding to the observable actions. These types are related `forget:RichLabel → TransitionLabel`. Labels in the third layer are simply lists of `TransitionLabels`.

```
Inductive RichLabel : Type :=
| RL_Com (p:Pid) (v:Value) (q:Pid) (x:Var) : RichLabel
| RL_Sel (p:Pid) (q:Pid) (l:Label) : RichLabel
| RL_Cond (p:Pid) : RichLabel
| RL_Call (X:RecVar) (p:Pid) : RichLabel.
```

```
Inductive TransitionLabel : Type :=
| TL_Com (p:Pid) (v:Value) (q:Pid) : TransitionLabel
| TL_Sel (p:Pid) (q:Pid) (l:Label) : TransitionLabel
| TL_Tau (p:Pid) : TransitionLabel.
```

Pen-and-paper presentations only include `TransitionLabels`, which capture what can be observed in transitions without revealing syntactic information about the choreography. However, in Coq, this information is needed to obtain induction hypotheses that are strong enough for our development, which is why we have introduced `RichLabels`.

The transition relations are defined inductively by the rules in Figs. 5, 6, 7. For readability, we present them in a more standard rule notation – below, we exemplify how they correspond to constructors in the formalisation. We also introduce suggestive notations for all these relations: `≪C,s≫ —[rl,D]⟶ ≪C',s'≫` stands for `(CCC_To D C s rl C's')` (this relation is parameterised by `D:DefSet` for dealing with procedure calls); `c —[tl]⟶ c'` stands for `(CCP_To c tl c')`, where `c,c':Configuration` are pairs containing a `Program` and a `State`; and `c —[ts]⟶* c'` stands for `(CCP_ToStar c ts c')`.

The rules defining `CCC_To` can be divided into three groups, which we describe in the following paragraphs.

Transition rules. Rules `C_Com`, `C_Sel`, `C_Then` and `C_Else` deal with execution of the first action in a choreography.

As an example, rule `C_Sel` corresponds to a constructor

```
C_Sel D p q l a C s s': s [==] s' → CCC_To D (p ⟶ q [l] @a ;; C) s (RL_Sel p q l) C s'
```

Including the requirement `s [==] s'` instead of simply writing `s` in the conclusion is essential for enabling transitions between different intensional representations of the same state, which occur in practice. In particular, confluence (discussed below) does not hold without this formulation. The corresponding more compact rules are proved as lemmas, e.g.,

```
Lemma C_Sel' : ≪p ⟶ q[l] @a;; C,s≫ —[RL_Sel p q l,D]⟶ ≪C,s≫.
```

These formulations can be useful in proofs that use existential tactics to infer a previously uninstantiated target of a transition.

Procedure calls. Rules `C_Call_Local`, `C_Call_Start`, `C_Call_Enter` and `C_Call_Finish` allow a process to enter a procedure call, with different cases according to whether other processes have already entered the procedure and/or whether there are any other processes that still have to join it.

A procedure call is expanded when the first process joins it (rule `C_Call_Start`). The remaining processes and the procedure's definition are stored in a runtime term, from which we can observe transitions either by more processes entering the procedure (rule `C_Call_Enter`) or by out-of-order execution of internal transitions of the procedure (rule `C_Delay_Call`, discussed below). When the last process enters the procedure, the runtime term is consumed (rule `C_Call_Finish`). Rule `C_Call_Local` addresses the edge case of a procedure that only uses one process.

Out-of-order execution. Rules `C_Delay_Eta`, `C_Delay_Cond` and `C_Delay_Call` deal with out-of-order execution (cf. Example 3). These rules require that the processes involved in the transition do not appear in the first term in the choreography; these conditions are specified by auxiliary predicates defined straightforwardly.

Example 10 Consider the program `(D,C1)` where `C1` is the choreography in Example 8 and `D:DefSet` is arbitrary (there are no recursive calls in `C1`).

```
(D, C1, st1) —[L_Com c ip v1]⟶ (D, If ip ?? (check x) Then C1t Else C1e, st2)
```

$$\frac{\mathtt{v\ :=\ eval\_on\_state\ Ev\ e\ s\ p} \qquad \mathtt{s'\ [==]\ s[\![q,x \Rightarrow v]\!]}}{\ll \mathtt{p\#e} \longrightarrow \mathtt{q\$x@a;;\ C,s} \gg\ -\!\![\mathtt{RL\_Com\ p\ v\ q\ x,\ D}]\!\!\longrightarrow\ \ll \mathtt{C,s'} \gg}\ \mathtt{C\_Com}$$

$$\frac{\mathtt{s\ [==]\ s'}}{\ll \mathtt{p} \longrightarrow \mathtt{q[l]@a;;\ C,s} \gg\ -\!\![\mathtt{RL\_Sel\ p\ q\ l,\ D}]\!\!\longrightarrow\ \ll \mathtt{C,s'} \gg}\ \mathtt{C\_Sel}$$

$$\frac{\mathtt{eval\_on\_state\ BEv\ b\ s\ p} = \mathtt{true} \qquad \mathtt{s\ [==]\ s'}}{\ll \mathtt{If\ p??b\ Then\ C1\ Else\ C2,s} \gg\ -\!\![\mathtt{RL\_Cond\ p,\ D}]\!\!\longrightarrow\ \ll \mathtt{C1,s'} \gg}\ \mathtt{C\_Then}$$

$$\frac{\mathtt{eval\_on\_state\ BEv\ b\ s\ p} = \mathtt{false} \qquad \mathtt{s\ [==]\ s'}}{\ll \mathtt{If\ p??b\ Then\ C1\ Else\ C2,s} \gg\ -\!\![\mathtt{RL\_Cond\ p,\ D}]\!\!\longrightarrow\ \ll \mathtt{C2,s'} \gg}\ \mathtt{C\_Else}$$

$$\frac{\mathtt{disjoint\_eta\_rl\ eta\ t} \qquad \ll \mathtt{C,s} \gg\ -\!\![\mathtt{t,D}]\!\!\longrightarrow\ \ll \mathtt{C',s'} \gg}{\ll \mathtt{eta@ann;;\ C,s} \gg\ -\!\![\mathtt{t,D}]\!\!\longrightarrow\ \ll \mathtt{eta@ann;;\ C',s'} \gg}\ \mathtt{C\_Delay\_Eta}$$

$$\frac{\mathtt{disjoint\_p\_rl\ p\ t} \qquad \begin{array}{c}\ll \mathtt{C1,s} \gg\ -\!\![\mathtt{t,D}]\!\!\longrightarrow\ \ll \mathtt{C1',s'} \gg \\ \ll \mathtt{C2,s} \gg\ -\!\![\mathtt{t,D}]\!\!\longrightarrow\ \ll \mathtt{C2',s'} \gg\end{array}}{\ll \mathtt{If\ p??b\ Then\ C1\ Else\ C2,s} \gg\ -\!\![\mathtt{t,D}]\!\!\longrightarrow\ \ll \mathtt{If\ p??b\ Then\ C1'Else\ C2',s'} \gg}\ \mathtt{C\_Delay\_Cond}$$

$$\frac{\mathtt{disjoint\_ps\_rl\ ps\ t} \qquad \ll \mathtt{C,s} \gg\ -\!\![\mathtt{t,D}]\!\!\longrightarrow\ \ll \mathtt{C',s'} \gg}{\ll \mathtt{RT\_Call\ X\ ps\ C,s} \gg\ -\!\![\mathtt{t,D}]\!\!\longrightarrow\ \ll \mathtt{RT\_Call\ X\ ps\ C',s'} \gg}\ \mathtt{C\_Delay\_Call}$$

$$\frac{\mathtt{s\ [==]\ s'} \qquad \mathtt{[\#](fst\ (D\ X))} = 1 \qquad \mathtt{In\ p\ (fst\ (D\ X))}}{\ll \mathtt{Call\ X,s} \gg\ -\!\![\mathtt{RL\_Call\ X\ p,D}]\!\!\longrightarrow\ \ll \mathtt{snd\ (D\ X),s'} \gg}\ \mathtt{C\_Call\_Local}$$

$$\frac{\mathtt{s\ [==]\ s'} \qquad \mathtt{[\#](fst\ (D\ X))} > 1 \qquad \mathtt{In\ p\ (fst\ (D\ X))}}{\ll \mathtt{Call\ X,s} \gg\ -\!\![\mathtt{RL\_Call\ X\ p,D}]\!\!\longrightarrow\ \ll \mathtt{RT\_Call\ X\ (fst\ (D\ X)[\backslash]p)\ (snd\ (D\ X)),s'} \gg}\ \mathtt{C\_Call\_Start}$$

$$\frac{\mathtt{s\ [==]\ s'} \qquad \mathtt{[\#]ps} > 1 \qquad \mathtt{In\ p\ ps}}{\ll \mathtt{RT\_Call\ X\ ps\ C,s} \gg\ -\!\![\mathtt{RL\_Call\ X\ p,D}]\!\!\longrightarrow\ \ll \mathtt{RT\_Call\ X\ (ps[\backslash]p)\ C,s'} \gg}\ \mathtt{C\_Call\_Enter}$$

$$\frac{\mathtt{s\ [==]\ s'} \qquad \mathtt{[\#]ps} = 1 \qquad \mathtt{In\ p\ ps}}{\ll \mathtt{RT\_Call\ X\ ps\ C,s} \gg\ -\!\![\mathtt{RL\_Call\ X\ p,D}]\!\!\longrightarrow\ \ll \mathtt{C,s'} \gg}\ \mathtt{C\_Call\_Finish}$$

Fig. 5 Semantics of choreographies, lower layer (`CCC_To`)

Fig. 6 Semantics of choreographies, middle layer (`CCP_To`)

$$\frac{\ll \mathtt{C,s} \gg\ -\!\![\mathtt{t,D}]\!\!\longrightarrow\ \ll \mathtt{C',s'} \gg}{\mathtt{(D,C,s)}\ \ -\!\![\mathtt{forget\ t}]\!\!\longrightarrow\ \mathtt{(D,C',s')}}\ \mathtt{CCP\_Base}$$

$$\frac{s[==]s'}{\mathtt{(P,s)}\ \ -\!\![\mathtt{nil}]\!\!\longrightarrow^*\ \mathtt{(P,s')}}\ \mathtt{CCT\_Base} \qquad \frac{\mathtt{c1}\ \ -\!\![\mathtt{t}]\!\!\longrightarrow\ \mathtt{c2} \qquad \mathtt{c2}\ \ -\!\![\mathtt{l}]\!\!\longrightarrow^*\ \mathtt{c3}}{\mathtt{c1}\ \ -\!\![\mathtt{t::l}]\!\!\longrightarrow^*\ \mathtt{c3}}\ \mathtt{CCT\_Step}$$

Fig. 7 Semantics of choreographies, top layer (`CCP_ToStar`)

where `v1:= eval_on_state Ev credentials st1 c` is the evaluation of `credentials` at `c` in `st1` according to the evaluation function `Ev`, and `st2:= st1`$[\![$`ip,x` $\Rightarrow$ `v1`$]\!]$. If `check x` evaluates to `true` at `ip` in `st2`, then execution continues as follows.

$$\begin{array}{l}\mathtt{(D,\ If\ ip\ ??\ (check\ x)\ Then\ C1t\ Else\ C1e,\ st2)}\ \ -\!\![\mathtt{L\_Tau\ ip}]\!\!\longrightarrow\ \mathtt{(D,\ C1t,\ st2)} \\ \quad -\!\![\mathtt{L\_Sel\ ip\ s\ left;\ L\_Sel\ ip\ c\ left}]\!\!\longrightarrow^*\ \mathtt{(D,\ s\#token} \longrightarrow \mathtt{c\$t;;\ End,\ st2)} \\ \quad -\!\![\mathtt{L\_Com\ s\ c\ v2}]\!\!\longrightarrow\ \mathtt{(D,\ End,\ st3)}\end{array}$$

where `v2:= eval_on_state Ev token st2 s` and `st3:= st2`$[\![$`c,t` $\Rightarrow$ `v2`$]\!]$.

If the check fails, the choreography instead continues as follows.

```
(D, If ip ?? (check x) Then C1t Else C1e, st2) —[L_Tau ip]⟶ (D, C1e, st2)
  —[L_Sel ip s right; L_Sel ip c right]⟶* (D, End, st2)
```

In the compound transitions in the examples above, the actions in the label are executed in order.

Example 11 Let `Defs` be as in Example 9 and `C` be the body of `FileTransfer`. Consider the program `(Defs,Call FileTransfer)`. The processes in the procedure `FileTransfer` can join it in any order as exemplified by the transitions below.

```
(Defs, Call FileTransfer, st) —[L_Tau c]⟶
  (Defs, RT_Call FileTransfer s::nil C, st) —[L_Tau s]⟶ (Defs, C, st)

(Defs, Call FileTransfer, st) —[L_Tau s]⟶
  (Defs, RT_Call FileTransfer c::nil C, st) —[L_Tau c]⟶ (Defs, C, st)
```

The state `st` is immaterial.

We prove a number of useful low-level properties about transitions. For example, we show that transitions are preserved by state equivalence.

```
Lemma CCC_To_eq : s1 [==] s1' → s2 [==] s2' →
  ≪C,s1≫ —[tl,D]⟶ ≪C',s2≫ → ≪C,s1'≫ —[tl,D]⟶ ≪C',s2'≫.
```

This result generalises to `CCP_To` and `CCP_ToStar`. Likewise, we show that: the set of processes involved in a choreography cannot increase during execution; transitions preserve well-formedness and the set of procedure definitions; well-formed choreographies do not perform self-communications; and terminated choreographies cannot perform transitions.

3.4 Progress, Determinism, and Confluence

The challenging part of formalising CC is establishing the core properties of the language semantics, which are essential for more advanced results and not always proven in full detail in pen-and-paper publications. We discuss some of the issues encountered, as these were also the driving force behind the changes relative to [16].

The first key property of choreographies is that they are deadlock-free by design: any choreography that is not terminated can execute.

This is proved by doing case analysis on the choreography and invoking the rule consuming its first action. Since the only terminated choreography in CC is `End`, this property also implies that any choreography either eventually reaches the terminated choreography `End` or runs infinitely.

```
Theorem progress : ∀ P, Main P ≠ End → Program_WF P →
  ∀ s, ∃ tl c', (P,s) —[tl]⟶ c'.

Theorem deadlock_freedom : ∀ P, Program_WF P →
  ∀ s ts c', (P,s) —[ts]⟶* c' →
  {Main (fst c') = End} + {∃ tl c'', c' —[tl]⟶ c''}.
```

The second property of our semantics is that it is deterministic, in the sense that transitions can be uniquely inferred from their label or the resulting state. These properties are essential for later results, and the need for them was the original motivation for introducing type `RichLabel`—the first group of results does not hold if `TransitionLabels` are used in the definition of `CCC_To`.

```
Lemma CCC_To_deterministic : ≪C,s≫ —[tl1,D]⟶ ≪C1,s1≫ →
  ≪C,s≫ —[tl2,D]⟶ ≪C2,s2≫ → tl1 = tl2 → C1 = C2 ∧ s1 [==] s2.

Lemma CCC_To_deterministic_3 : ≪C,s≫ —[tl1,D]⟶ ≪C',s1≫ →
  ≪C,s≫ —[tl2,D]⟶ ≪C',s2≫ → tl1 = tl2.
```

The third key property is confluence, which has some relevant implications for our calculus: if a choreography has two different transition paths, then these paths either end at the same configuration, or both resulting configurations can reach the same one. This is proved by first showing the diamond property for choreography transitions (considering all possible combinations of independent transitions), then lifting it to one-step transitions, and finally applying induction to show it for multistep transitions.

```
Lemma diamond_Chor :
  ≪C,s≫ —[tl1,D]⟶ ≪C1,s1≫ → ≪C,s≫ —[tl2,D]⟶ ≪C2,s2≫ → tl1 ≠ tl2 →
  ∃ C' s', ≪C1,s1≫ —[tl2,D]⟶ ≪C',s'≫ ∧ ≪C2,s2≫ —[tl1,D]⟶ ≪C',s'≫.

Lemma diamond_1 : c —[tl1 ]⟶ c1 → c —[tl2 ]⟶ c2 → tl1 ≠ tl2 →
  ∃ c', c1 —[tl2 ]⟶ c' ∧ c2 —[tl1 ]⟶ c'.

Lemma diamond_4 : (P,s) —[tl1 ]⟶* (P1,s1) → (P,s) —[tl2 ]⟶* (P2,s2) →
  (∃ P' tl1' tl2' s1' s2',
    (P1,s1) —[tl1' ]⟶* (P',s1') ∧ (P2,s2) —[tl2' ]⟶* (P',s2') ∧ s1' [==] s2').
```

As an important consequence, we get that any two executions of a choreography that end in a terminated choreography must finish in the same state.

```
Lemma termination_unique : c —[tl1]⟶* c1 → c —[tl2]⟶* c2 →
  Main (fst c1) = End → Main (fst c2) = End → snd c1 [==] snd c2.
```

Using these results, we can establish Turing completeness of CC. The structure of the proof closely follows that of [16], and has been described in [21]. We briefly summarise it for completeness of the presentation.

First, we formalise Kleene's partial recursive functions [34] as an inductive type in Coq. Since all functions in Coq are total, this definition only establishes syntax for them. We define an evaluation function separately that takes a partial recursive function f, an input $\vec{n}$ and a number of steps k, and performs k steps of the computation of $f(\vec{n})$—where e.g. base functions evaluate to their value in one step, while unfolding a composition or performing one recursive call takes one step. This allows us to define convergence to a value (the computation finishes in a finite number of steps) and divergence (the computation does not finish in any number of steps).

Next, we define a mapping from partial recursive functions to choreographies, and show that there is a correspondence between the evaluation function defined above and the execution of the choreography. In particular, given a function f and an input $\vec{n}$, if $f(\vec{n})$ converges to a value, then executing the choreography obtained from f from an appropriate state storing the values $\vec{n}$ terminates in a state where a particular process stores the result; if $f(\vec{n})$ is undefined, then execution of the choreography never terminates. The converse implications also hold. The formal proof relies essentially on the definition of the semantics of choreographies and confluence results. The interested reader is referred to the works cited above for additional details.

3.5 Discussion

Formalising the proof of confluence following [16] turned out to be a spiralling process: the pen-and-paper proof assumes some obvious properties, which were not proved; proving these required some additional lower-level lemmas; these in turn generated some even more specific lemmas; and so on. At some point, we realised that the auxiliary lemmas accumulated already accounted for 90% of the formalisation. Worse, these lemmas were extremely specific and detached from the contents of [16]—even though we were far from done. This led us to rethinking the design of CC, and eventually to adopting the language of [41].

In this section, we discuss the features of the original language that turned out to be problematic. These regarded the handling of procedure definitions (syntax) and the treatment of procedure calls and out-of-order execution (semantics).

Syntax. Procedures were initially defined by including a term `def X=CX in C` in the grammar defining choreographies. While this removed the need for a separate notion of program, it introduced several dimensions of complexity. Even the notion of terminated choreography was nontrivial, since `End` could occur arbitrarily deep inside some of these terms. This made it hard to ensure that the Coq definition was an adequate representation of the informal notion in [16], affecting all results regarding termination, progress, and deadlock-freedom. With the current syntax, terminated programs are exactly those whose for which `Main P=End`.

Additionally, the name `X` in `def X=CX in C` acts as a binder, which added all the usual problems of working with binders—in particular, having to deal with capture-avoiding substitutions and α-renaming. In the current language, procedure names are statically determined and fixed, so there is no need to rename them ever, and they can be treated as constants. This constructor also allowed for unintuitive choreographies, e.g., `def X=CX in C` where the choreography `CX` itself contains additional procedure definitions.

Pairing procedure definitions with choreographies in programs yields a cleaner theory, and the overhead of an additional layer is a very small price to pay for the simplicity gained. This approach had been proposed earlier [13], and the two formulations are argued to be equally expressive in [18].

Semantics. Instead of a labelled transition system, the semantics of [16] was a reduction semantics that used a structural precongruence relation to model out-of-order execution and to unfold procedure definitions.

To understand this issue, consider again Example 3, which shows a choreography that has two possible initial transitions. In a framework with reductions and structural precongruence, the out-of-order transition is obtained by first rewriting the choreography as $\mathsf{o}_2 \rightarrow \mathsf{s}_2; \mathsf{o}_1 \rightarrow \mathsf{s}_1$ and then applying rule COM. The set of legal rewritings is formally defined by the structural precongruence relation $\preceq$, and there is a rule in the semantics that closes the transition relation under it.

In any proofs about the semantics, an approach using structural precongruence needs to take into account all the possible ways into which choreographies may be rewritten in a reduction. Concretely, in the proof of confluence, where there are two reductions, there are four possible places where choreographies are rewritten; given the high number of rules defining structural precongruence, this led to an explosion of the number of cases. Furthermore, induction hypotheses typically were not strong enough, requiring us to resort to complicated auxiliary notions such as explicitly measuring the size of the derivation of transitions, and proving that rewritings could be normalised. This process led to a seemingly ever-growing number of auxiliary lemmas that needed to be proved, with no counterpart in the original reference [16], and after several months of work with little progress it became evident that the problem lay in the formalism.

Summary. The current proof of confluence takes about 300 lines of Coq code, including a total of 11 lemmas. This is in stark contrast with the previous attempt, which while still unfinished already included over 30 lemmas with extremely long proofs.

With the current definitions, the theory of CC is formalised in two files. The first file, which defines the preliminaries, contains 24 definitions, 60 lemmas and around 740 lines of code. The second file, defining the syntax and semantics of CC and proving properties about it (including all the ones described herein), contains 32 definitions, 126 lemmas, 2 theorems and around 2300 lines of code.

The formalisation of partial recursive functions contains 22 definitions and 84 lemmas, with a total of 1464 lines of code. The proof of Turing completeness consists of 28 definitions and 65 lemmas, with a total of 2371 lines of code.

4 The Process Language

The second part of our formalisation concerns the process calculus that we use for implementing CC: Stateful Processes (SP). We follow the pen-and-paper design presented in [41]. SP is used to define networks of processes running in parallel, each with its own behaviour, that can interact by direct messaging.

4.1 Syntax

The syntax of SP is structured in three layers: behaviours, which express the local actions performed by individual processes; networks, which combine processes in a system where they can interact; and programs, which pair a network with a set of procedure definitions (which all processes can call). As with CC, we assume an underlying signature.

The constructors for behaviours correspond to those for choreographies, but interactions are now split between the two different roles involved (sender and receiver). The type `Behaviour` is defined inductively from the grammar below.

```
 B := End | p!e @! a; B | p?x @? a; B | p(+)l @+ a; B | p & mB1 // mB2
      | If e Then B1 Else B2 | Call X
mB := None | Some (a,B)
```

Conditionals, procedure calls, and the terminated behaviour are standard and similar to the corresponding constructs in CC.

A term `p!e @! a` represents a send action towards `p`, where `e` is the expression used to compute the value to be sent and `a` is an annotation. Dually, a term `p?x @? a` represents a receive action where a value received from `p` is stored in the local variable `x` (`a` is, again, an annotation).

A selection action `p(+)l @+ a; B` is similar to a send action (label `l` is sent to `p`). The dual action needs to offer a behaviour for `l`, but may also accept other labels. In pen-and-paper presentations, these *branching terms* are typically defined as partial functions from labels to behaviours.

Formalising this informal description is challenging. A natural choice would be to include a constructor `Branching: Pid → (Label → option Behaviour) → Behaviour`. However, this is problematic for defining EPP, which relies on a recursively defined function on pairs of behaviours called *merging* (cf. Sect. 5.1). Defining this function directly in Coq is unwieldy because of the complexity of writing the appropriate term of type `Label → option Behaviour` given the corresponding subterms from the arguments.

Using partial functions also seems like an overkill, considering that there are only two possible labels. Instead, we include a constructor

```
Branching : Pid → option (Ann∗Behaviour) → option (Ann∗Behaviour) → Behaviour
```

that registers explicitly the behaviours offered for each of the two possible labels, in order. This design choice avoids the aforementioned issues, at the cost of making our development harder to generalise to larger sets of labels in the future.

Because of the option types in `Branching`, the induction principles generated automatically for `Behaviour` are not strong enough (they do not include induction hypotheses over the `Behaviours` appearing within branching terms). To overcome this, we define an auxiliary function `depth:Behaviour → nat` measuring the depth of the AST corresponding to a `Behaviour`, use it to prove the expected general induction principle, and define a tactic `BInduction B` that applies it.

Networks. Networks are simply (total) functions from processes to behaviours.

```
Definition Network := Pid → Behaviour.
```

We define extensional equality of networks `N (==) N'` in the expected way and show that it is an equivalence relation. We support the common notation for writing networks by including a function for constructing singleton networks `p[B]`, a parallel composition operator `N | N'`, and a removal operator `N p` (recall the description in Sect. 2).

For simplicity, we do not require disjoint support in parallel composition: if both networks define a nonterminated behaviour for `p`, the result of `(N | N') p` and `(N' | N) p` is different. Although this may seem odd, it has the advantage of making parallel composition total. We show that parallel composition is commutative under the assumption that the two composed networks have disjoint supports.

```
Lemma Behaviour_eq_End_dec : ∀ (b:Behaviour), {b=End} + {b≠ End}.

(* N | N' stands for (Par N N') *)
Definition Par (N N':Network) :=
  fun p ⇒ if (Behaviour_eq_End_dec (N p)) then N' p else N p.

Definition Network_disjoint (N N':Network) := ∀ p, N p = End ∨ N' p = End.

Lemma Par_comm : Network_disjoint N N' → (N | N') (==) (N' | N).
```

Our library includes a number of results to reason about the network operations, including very specific lemmas dealing with networks that appear in the rules defining the semantics of SP, e.g., that updating the behaviours of two distinct processes yields the same result independent of the order of the updates.

Programs and well-formedness. As before, a program is a pair consisting of a set of procedure definitions and a network.

```
Definition DefSetB := RecVar → Behaviour.
Definition Program := DefSetB ∗ Network.
```

Well-formedness is significantly simpler than for choreographies. If `B:Behaviour`, then `B` is well-formed, `Behaviour_WF B`, as long as no process in `B` attempts to communicate with itself. `N:Network` is well-formed, `Network_WF N`, if all processes are mapped to well-formed behaviours. This is not decidable in general, but it is under the assumption that all processes outside a given set `ps` are mapped to `End`—an assumption that holds for all networks that can be written explicitly using parallel composition of singleton networks.

$$\frac{\begin{array}{c}\texttt{N p} = (\texttt{q ! e @!a ; B})\\ \texttt{N q} = (\texttt{p ? x @? a'; B'})\end{array}\quad \begin{array}{c}\texttt{v} := \texttt{eval\_on\_state Ev e s p}\\ \texttt{N'}\ (==)\ (\texttt{N p}\ \ \texttt{q} \mid \texttt{p[B]} \mid \texttt{q[B']})\end{array}\quad \texttt{s'}\ [==]\ (\texttt{s}[\![\texttt{q,x} \Rightarrow \texttt{v}]\!])}{\ll\texttt{N,s}\gg\ \text{—[}\texttt{RL\_Com p v q x,D}\text{]}\!\!\longrightarrow\ \ll\texttt{N',s'}\gg}\ \texttt{S\_Com}$$

$$\frac{\begin{array}{c}\texttt{N p} = (\texttt{q (+) left @+ a ; B})\\ \texttt{N q} = (\texttt{p \& Some (a',Bl) // Br})\end{array}\quad \texttt{N'}\ (==)\ (\texttt{N p}\ \ \texttt{q} \mid \texttt{p[B]} \mid \texttt{q[Bl]})\quad \texttt{s}\ [==]\ \texttt{s'}}{\ll\texttt{N,s}\gg\ \text{—[}\texttt{RL\_Sel p q left,D}\text{]}\!\!\longrightarrow\ \ll\texttt{N',s'}\gg}\ \texttt{S\_LSel}$$

$$\frac{\begin{array}{c}\texttt{N p} = (\texttt{q (+) right @+ a ; B})\\ \texttt{N q} = (\texttt{p \& Bl // Some (a',Br)})\end{array}\quad \texttt{N'}\ (==)\ (\texttt{N p}\ \ \texttt{q} \mid \texttt{p[B]} \mid \texttt{q[Br]})\quad \texttt{s}\ [==]\ \texttt{s'}}{\ll\texttt{N,s}\gg\ \text{—[}\texttt{RL\_Sel p q right,D}\text{]}\!\!\longrightarrow\ \ll\texttt{N',s'}\gg}\ \texttt{S\_RSel}$$

$$\frac{\begin{array}{c}\texttt{N p} = (\texttt{If b Then B1 Else B2})\\ \texttt{eval\_on\_state BEv b s p} = \texttt{true}\end{array}\quad \texttt{N'}\ (==)\ (\texttt{N p} \mid \texttt{p[B1]})\quad \texttt{s}\ [==]\ \texttt{s'}}{\ll\texttt{N,s}\gg\ \text{—[}\texttt{RL\_Cond p,D}\text{]}\!\!\longrightarrow\ \ll\texttt{N',s'}\gg}\ \texttt{S\_Then}$$

$$\frac{\begin{array}{c}\texttt{N p} = (\texttt{If b Then B1 Else B2})\\ \texttt{eval\_on\_state BEv b s p} = \texttt{false}\end{array}\quad \texttt{N'}\ (==)\ (\texttt{N p} \mid \texttt{p[B2]})\quad \texttt{s}\ [==]\ \texttt{s'}}{\ll\texttt{N,s}\gg\ \text{—[}\texttt{RL\_Cond p,D}\text{]}\!\!\longrightarrow\ \ll\texttt{N',s'}\gg}\ \texttt{S\_Else}$$

$$\frac{\texttt{N p} = \texttt{Call X}\quad \texttt{N'}\ (==)\ (\texttt{N p} \mid \texttt{p[D X]})\quad \texttt{s}\ [==]\ \texttt{s'}}{\ll\texttt{N,s}\gg\ \text{—[}\texttt{RL\_Call X p,D}\text{]}\!\!\longrightarrow\ \ll\texttt{N',s'}\gg}\ \texttt{S\_Call}$$

Fig. 8 Semantics of networks, bottom layer (SP_To)

Fig. 9 Semantics of networks, middle layer (SPP_To)

$$\frac{\ll\texttt{N,s}\gg\ \text{—[}\texttt{t,D}\text{]}\!\!\longrightarrow\ \ll\texttt{N',s'}\gg}{(\texttt{D,N,s})\ \ \text{—[}\texttt{forget t}\text{]}\!\!\longrightarrow\ (\texttt{D,N',s'})}\ \texttt{SPP\_Base}$$

Well-formedness of programs does not make sense: well-formedness of a behaviour depends on who is executing it, but a procedure definition has no information about which processes will call it.

Example 12 Consider the network N = c[Bc] | s[Bs] | ip[Bip], where:

```
Bc := ip!credentials; ip & Some (s?t; End) // Some End
Bs := ip & Some (c!token; End) // Some End

Bip := c?x; Bip'
Bip' := If (check x) Then (s(+)left; c(+)left; End) Else (s(+)right; c(+)right; End)
```

This network implements the choreography in Example 8.

4.2 Semantics

The semantics of SP is again defined by a labelled transition system. Transitions for communications match dual actions in two processes, while conditionals and procedure calls simply run locally. There are again three layers of definitions, which are shown in Fig. 8, 9, 10, and two types of transition labels (as in CC). Transitions support suggestive notations: ≪N,s≫ —[tl,D]⟶ ≪N',s'≫ for (SP_To D N s tl N's'), C —[l]⟶ C' for (SPP_To C l C'), and C —[ls]⟶* C' for (SPP_ToStar C ls C').

These definitions warrant similar observations as those for the semantics of CC. Transitions include premises on network equality and state equality, rather than requiring specific values.

$$\frac{\texttt{s [==] s'}}{\texttt{(P,s)}\ \longrightarrow\!\![\texttt{nil}]\!\!\longrightarrow^*\ \texttt{(P,s')}}\ \texttt{SPT\_Base} \qquad \frac{\texttt{c1}\ \longrightarrow\!\![\texttt{t}]\!\!\longrightarrow \texttt{c2} \quad \texttt{c2}\ \longrightarrow\!\![\texttt{l}]\!\!\longrightarrow^* \texttt{c3}}{\texttt{c1}\ \longrightarrow\!\![\texttt{t::l}]\!\!\longrightarrow^* \texttt{c3}}\ \texttt{SPT\_Step}$$

Fig. 10 Semantics of networks, top layer (SPP_ToStar)

We include some lemmas stating the more restricted rules, both as a sanity check and because they can be useful to instantiate variables created by the use of existential tactics in proofs.

Lemma S_LSel' : N p = (q (+) left @+ a ; B) → N q = (p & Some (a',Bl) // Br) →
 ≪N,s≫ —[RL_Sel p q left,D]⟶ ≪N \ p \ q | p[B] | q[Bl],s≫.

There are two rules for reducing selections, one for each label. This is a deviation for standard practice (where there is a single rule and a premise matching the label in both behaviours) stemming from our design choice of avoiding functions in branching terms. Having an extra rule generates additional cases in induction proofs, but this formulation effectively simplifies the formalisation by eliminating one layer of inversion.

Example 13 We illustrate the possible transitions of the network from Example 12. We abbreviate the behaviours of processes that do not change in a reduction to … to make it clearer what parts of the network are changed. Furthermore, we omit trailing Ends in Behaviours.

The network starts by performing the transition

```
(D, c[Bc] | s[Bs] | ip[Bip], st1) —[L_Com c ip v1]⟶
 (D, c[ip & Some (s?t) // Some End] | s[...] | ip[Bip'], st2)
```

where v1 and st2 are as in Example 10.

If eval_on_state (check x) st2 ip=true, execution continues as

```
(D, c[ip & Some (s?t) // Some End] | s[Bs] | ip[Bip'], st2)
  —[L_Tau ip]⟶ (D, c[...] | s[...] | ip[s(+)left;c(+)left], st2)
  —[L_Sel ip s left]⟶ (D, c[...] | s[c!token] | ip[c(+)left], st2)
  —[L_Sel ip c left]⟶ (D, c[s?t] | s[...] | ip[End], st2)
  —[L_Com s c v2]⟶ (D, c[End] | s[End] | ip[End], st3)
```

where v2 and st3 are again as in Example 10. Otherwise, it continues as follows.

```
(D; c[ip & Some (s?t) // Some End] | s[Bs] | ip[Bip'], st2)
  —[L_Tau ip]⟶ (D; c[...] | s[...] | ip[s(+)right;c(+)right], st2)
  —[L_Sel ip s right]⟶ (D; c[...] | s[End] | ip[c(+)right], st2)
  —[L_Sel ip c right]⟶ (D; c[End] | s[End] | ip[End], st2)
```

The labels in these reductions are exactly as in Example 10.

4.3 Determinism and Confluence

As for CC, we prove a number of useful results about the semantics of SP. These can be roughly divided in two groups: results showing that reductions are stable under the extensional equalities on the different types involved, and properties on the actual transitions. While the results in the first category are not surprising, they are useful and show that the definitions make sense.

Lemma SP_To_eq : s1 [==] s1' → s2 [==] s2' →
 ≪N,s1≫ —[tl,D]⟶ ≪N',s2≫ → ≪N,s1'≫ —[tl,D]⟶ ≪N',s2'≫.

Lemma SP_To_Network_eq : N1 (==) N2 →
 ≪N1,s≫ —[tl,D]⟶ ≪N',s'≫ → ≪N2,s≫ —[tl,D]⟶ ≪N',s'≫.

Lemma SP_To_Defs_wd : (∀ X, D X = D' X) →
 ≪N,s≫ —[tl,D]⟶ ≪N',s'≫ → ≪N,s≫ —[tl,D']⟶ ≪N',s'≫.

While determinism and confluence are similar to the corresponding results to CC, they are not as interesting: for networks generated by EPP (which are the ones we are interested in), these results would follow by the same properties for choreographies.

The formalisation of SP consists of 25 definitions, 81 lemmas, 11 simple tactics, and approximately 1960 lines of Coq code.

5 Endpoint Projection

As with the simple language from Sect. 2, the intuition for generating process implementations is that each choreographic action should be projected to the corresponding process action. The prototypical example is the value communication `p#e` ⟶ `q$x @a`, which should be projected to a send action `q!e @!a` for `p`, to a receive action `p?x @?a` for `q`, and skipped for any other processes.

In the presence of conditionals, this intuition is not enough. Projecting a conditional `If p.b Then Ct Else Ce` for any process other than `p`, say `q`, is nontrivial, because `q` has no way of knowing which branch should be executed. Therefore `q`'s behaviour must combine the projections obtained for `Ct` and `Ce`.

This problem is commonly known as *knowledge of choice*, and one of the solutions relies on the usage of label selections [8, 9]. If `q`'s behaviour should depend on the result of `p`'s local evaluation, then the result of this evaluation should be communicated to `q` by means of a label selection. The two possible behaviours can then be combined in a branching term offering two different options.

5.1 Merge

A standard way of combining behaviours to solve the problem above is the *merge* operator [8]: a partial binary operator that returns a behaviour combining all possible executions of its arguments (if possible). In SP, two behaviours can be merged only if they are built from the same constructor with matching parameters. So if `B1` can be merged with `B2` to yield `B`, we can also merge `p?x @? a; B1` with `p?x @? a; B2` to obtain `p?x @? a; B`, but `p?x @? a; B1` can never be merged with `q?x @? a; B2` for `p`≠ `q` (different arguments) or with `q!e @! a; B2` (different constructor).

The only exception is branching terms, where merge can combine offers on different labels. For example, merging `p & Some (a,B) // None` with `p & None // Some (a',B')` yields `p & Some (a,B) // Some (a',B')`. In this way, the prototypical choreographic conditional `If p??b Then (p⟶ q[left];; q.e ⟶ p;; End) Else (p⟶ q[right];; End)` can be projected for `q` as `p & Some (p!e; End) // Some End`.

The partiality of merge again poses a formalisation problem. Our original approach [20] defined an auxiliary type `XBehaviour` that extends the syntax of behaviours with a constructor `XUndefined: XBehaviour`. In this work, instead, we define a ternary relation `merge: Behaviour → Behaviour → Behaviour → Prop`.[6] While this design requires two additional lemmas stating that this relation is functional and computable, it significantly simplified this part of the formalisation (both in size and complexity of the proofs). As an example, [20] reported a number of inversion results, e.g., if merging two behaviours yields

[6] Technically, because `merge` is defined on a separate file, all types defined in the formalisation of SP need the signature as a parameter. Since this signature is fixed, we omit it everywhere.

$$\frac{}{\texttt{End [V] End == End}}\ \texttt{merge\_End} \qquad \frac{\texttt{B1 [V] B2 == B}}{\texttt{p!e @!a; B1 [V] p!e @!a; B2 == p!e @!a; B}}\ \texttt{merge\_Send}$$

$$\frac{\texttt{B1 [V] B2 == B}}{\texttt{p?x @?a; B1 [V] p?x @?a; B2 == p?x @?a; B}}\ \texttt{merge\_Recv}$$

$$\frac{\texttt{B1 [V] B2 == B}}{\texttt{p(+)l @+a; B1 [V] p(+)l @+a; B2 == p(+)l @+a; B}}\ \texttt{merge\_Sel}$$

$$\frac{}{\texttt{p \& None // None [V] p \& None // None == p \& None // None}}\ \texttt{merge\_Branching\_NNNN}$$

$$\frac{}{\begin{array}{r}\texttt{p \& Some (aL,bL) // None [V] p \& None // None}\\ \texttt{== p \& Some (aL,bL) // None}\end{array}}\ \texttt{merge\_Branching\_SNNN}$$

$$\frac{\texttt{bL1 [V] bL2 == bL}}{\begin{array}{r}\texttt{p \& Some (aL,bL1) // None [V] p \& Some(aL,bL2) // None}\\ \texttt{== p \& Some (aL,bL) // None}\end{array}}\ \texttt{merge\_Branching\_SNSN}$$

$$\frac{\texttt{bL1 [V] bL2 == bL}}{\begin{array}{r}\texttt{p \& Some (aL,bL1) // Some (aR,bR) [V] p \& Some(aL,bL2) // None}\\ \texttt{== p \& Some (aL,bL) // Some (aR,bR)}\end{array}}\ \texttt{merge\_Branching\_SSSN}$$

$$\frac{\texttt{bL1 [V] bL2 == bL} \quad \texttt{bR1 [V] bR2 == bR}}{\begin{array}{l}\texttt{p \& Some (aL,bL1) // Some (aR,bR1)}\\ \quad\texttt{[V] p \& Some(aL,bL2) // Some(aR,bR2)}\\ \qquad\texttt{== p \& Some (aL,bL) // Some (aR,bR)}\end{array}}\ \texttt{merge\_Branching\_SSSS}$$

$$\frac{\texttt{Bt1 [V] Bt2 == Bt} \quad \texttt{Be1 [V] Be2 == Be}}{\begin{array}{r}\texttt{If p??e Then Bt1 Else Bt2 [V] If p??e Then Be1 Else Be2}\\ \texttt{== If p??e Then Bt Else Be}\end{array}}\ \texttt{merge\_Cond}$$

$$\frac{}{\texttt{Call X [V] Call X == Call X}}\ \texttt{merge\_Call}$$

Fig. 11 Definition of the merge relation

a behaviour starting with a send action, then both arguments start with that same action. All these results can now be obtained directly by applying inversion on the relevant hypotheses.

The full definition of `merge` includes 22 clauses. Figure 11 lists all representative cases; the missing clauses deal with the remaining combinations of `None` / `Some` subterms in branching terms (see Sect. 5.5 for a discussion on the exponential dependency of the number of clauses on the number of labels, and on the problems with formalising the more general definition from the literature). We also define the suggestive notation `B1 [V] B2 == B` for `merge B1 B2 B`, which reminds us that `merge` is a partial function.

We show that merge is functional, decidable, and preserves well-formedness.

All proofs are simple using induction on behaviours and inversion on the hypotheses on `merge`.

$$\frac{}{\mathtt{End}\ [\gg]\ \mathtt{End}}\ \mathtt{MB\_End} \qquad \frac{\mathtt{B}\ [\gg]\ \mathtt{B'}}{\mathtt{p!e\ @!\ a;\ B}\ [\gg]\ \mathtt{p!e\ @!\ a;\ B'}}\ \mathtt{MB\_Send}$$

$$\frac{\mathtt{B}\ [\gg]\ \mathtt{B'}}{\mathtt{p?x\ @?\ a;\ B}\ [\gg]\ \mathtt{p?x\ @?\ a;\ B'}}\ \mathtt{MB\_Recv} \qquad \frac{\mathtt{B}\ [\gg]\ \mathtt{B'}}{\mathtt{p(+)l\ @+\ a;\ B}\ [\gg]\ \mathtt{p(+)l\ @+\ a;\ B'}}\ \mathtt{MB\_Sel}$$

$$\frac{}{\mathtt{p\ \&\ mBl\ //\ mBr}\ [\gg]\ \mathtt{p\ \&\ None\ //\ None}}\ \mathtt{MB\_Branching\_NN}$$

$$\frac{\mathtt{Br}\ [\gg]\ \mathtt{Br'}}{\mathtt{p\ \&\ mBl\ //\ Some\ (a,Br)}\ [\gg]\ \mathtt{p\ \&\ None\ //\ Some\ (a,Br')}}\ \mathtt{MB\_Branching\_NS}$$

$$\frac{\mathtt{Bl}\ [\gg]\ \mathtt{Bl'}}{\mathtt{p\ \&\ Some\ (a,Bl)\ //\ mbR}\ [\gg]\ \mathtt{p\ \&\ Some\ (a,Bl')\ //\ None}}\ \mathtt{MB\_Branching\_SN}$$

$$\frac{\mathtt{Bl}\ [\gg]\ \mathtt{Bl'} \qquad \mathtt{Br}\ [\gg]\ \mathtt{Br'}}{\mathtt{p\ \&\ Some\ (a,Bl)\ //\ Some\ (a',Br)}\ [\gg]\ \mathtt{p\ \&\ Some\ (a,Bl')\ //\ Some\ (a',Br')}}\ \mathtt{MB\_Branching\_SS}$$

$$\frac{\mathtt{B1}\ [\gg]\ \mathtt{B1'} \qquad \mathtt{B2}\ [\gg]\ \mathtt{B2'}}{\mathtt{If\ p\ Then\ B1\ Else\ B2}\ [\gg]\ \mathtt{If\ p\ Then\ B1'\ Else\ B2'}}\ \mathtt{MB\_Cond} \qquad \frac{}{\mathtt{Call\ X}\ [\gg]\ \mathtt{Call\ X}}\ \mathtt{MB\_Call}$$

Fig. 12 Definition of the branching order

```
Lemma merge_unique : B1 [V] B2 == B → B1 [V] B2 == B' → B = B'.
Lemma merge_dec : { B | B1 [V] B2 == B } + { ~∃ B, B1 [V] B2 == B }.
Lemma merge_WF : B1 [V] B2 == B →
  Behaviour_WF _ p B1 → Behaviour_WF _ p B2 → Behaviour_WF _ p B.
```

Decidability is formulated using the stronger existential quantifier so that we can also obtain the existential witness to use in further definitions.

5.2 Branching Order

In the literature, the arguments of merge and its result, when defined, are in a relation known as the *branching order* [8, 41]. This is formalised as yet another inductive type, defined by the rules in Fig. 12.[7] We call the relation `more_branches`, for which we define the infix notation [≫].

The branching order is reflexive, transitive and antisymmetric. It is pointwise extended to networks by defining `N (≫) N':= ∀ p, N p [≫] N'p` where `(≫)` is infix notation for `more_branches_N: Network → Network → Prop`. This relation is again reflexive, transitive and antisymmetric (with respect to extensional equality).

More interestingly, adding branches to some behaviours in a network does not eliminate any transitions that the network can do.

```
Lemma SP_To_MBN : ≪N1,s≫ —[tl,D]⟶ ≪N2,s'≫ → N1' (≫) N1 →
  (∀ X, D X = D' X) → ∃ N2', ≪N1',s≫ —[tl,D']⟶ ≪N2',s'≫ ∧ N2' (≫) N2.
```

(The quantification on `D'` makes this lemma easier to apply.)

[7] In the formalisation, these definitions precede that of `merge`. This was chosen because the branching relation is more primitive than the notion of merging. For presentation purposes, though, it makes more sense to invert this order.

We can now justify the notation for `merge`: it is the partial join for the branching order, in the sense that if two behaviours have an upper bound, then they are mergeable and their merging is their least upper bound.

```
Lemma MB_merge : B1 [≫] B2 ↔ B1 [V] B2 == B1.

Lemma merge_is_upper_bound : B1 [V] B2 == B → B [≫] B1.

Lemma MB_has_lub : B [≫] B1 → B [≫] B2 → ∃ B', B1 [V] B2 == B' ∧ B [≫] B'.

Lemma merge_is_lub : B [≫] B1 → B [≫] B2 → ∀ B', B1 [V] B2 == B' → B [≫] B'.
```

Another key result is that the branching order is stable under merging:

```
Lemma MB_yields_merge : B1 [≫] B1' → B2 [≫] B2' → B1 [V] B2 == B →
  ∃ B', B1' [V] B2' == B' ∧ B [≫] B'.
```

As we will see, this result is essential for the cases of the EPP theorem dealing with conditionals.

Lastly, we prove the algebraic properties of `merge` – idempotency, commutativity, and associativity—by exploiting the relationship between `merge` and the branching order.

All proofs in this section are again simple induction arguments using inversion on the hypotheses on `merge` and `more_branches`.

5.3 Projection

We can now define the projection of a choreography for an individual process. Since this definition relies on `merge` for the case of the conditionals, it is also a partial function. We define it inductively as a relation

```
bproj : DefSet → Choreography Sig → Pid → Behaviour Sig' → Prop
```

and abbreviate `bproj D C p B` to `[[D,C | p]]== B`.

The type of `bproj` also reveals a new feature of projection when compared to the simple language from Sect. 2: the signature for the target instance of SP is different than that of the source instance of CC. The reason for this lies in the presence of procedure definitions: each procedure yields several projected procedures, one for each process in the choreography.[8] The type of procedure names in the target of projection is thus `RecVar*Pid`; this can be seen in the rule for projecting procedure calls, which is included with the remaining rules in Fig. 13.

We show that `bproj` is functional and decidable, and that it returns well-formed behaviours for choreographies without self-communications.

From `bproj` we obtain several notions of projectability: relative to a process or a set of processes, and projectability of `D:DefSet`—which requires each procedure to be projectable relative to its set of used processes.

These notions complement well-formedness, as being well-formed is not enough to be projectable—well-formedness does not ensure knowledge of choice.

```
Definition projectable_B D C p := ∃ B, [[D,C | p]] == B.
Definition projectable_C D C ps := Forall (fun p ⇒ projectable_B D C p) ps.
Definition projectable_D D := ∀ X, projectable_C D (snd (D X)) (fst (D X)).
```

[8] More precisely, one nontrivial procedure for each process actually involved in it—the remaining ones are all `End`.

$$\frac{}{\llbracket \mathtt{D,End} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{End}}\ \mathtt{bproj\_End}$$

$$\frac{\llbracket \mathtt{D,C} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{B}}{\llbracket \mathtt{D,p\#e} \longrightarrow \mathtt{q\$x\ @a\ ;;\ C} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{q!e\ @!a;\ B}}\ \mathtt{bproj\_Send}$$

$$\frac{\mathtt{p} \neq \mathtt{q} \qquad \llbracket \mathtt{D,C} \mid \mathtt{q}\rrbracket \mathrel{==} \mathtt{B}}{\llbracket \mathtt{D,p\#e} \longrightarrow \mathtt{q\$x\ @a\ ;;\ C} \mid \mathtt{q}\rrbracket \mathrel{==} \mathtt{p?x\ @?a;\ B}}\ \mathtt{bproj\_Recv}$$

$$\frac{\mathtt{p} \neq \mathtt{r} \qquad \mathtt{q} \neq \mathtt{r} \qquad \llbracket \mathtt{D,C} \mid \mathtt{r}\rrbracket \mathrel{==} \mathtt{B}}{\llbracket \mathtt{D,p\#e} \longrightarrow \mathtt{q\$x\ @a\ ;;\ C} \mid \mathtt{r}\rrbracket \mathrel{==} \mathtt{B}}\ \mathtt{bproj\_Com}$$

$$\frac{\llbracket \mathtt{D,C} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{B}}{\llbracket \mathtt{D,p} \longrightarrow \mathtt{q[l]\ @a\ ;;\ C} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{q(+)l\ @+a;\ B}}\ \mathtt{bproj\_Pick}$$

$$\frac{\mathtt{p} \neq \mathtt{q} \qquad \llbracket \mathtt{D,C} \mid \mathtt{q}\rrbracket \mathrel{==} \mathtt{B}}{\llbracket \mathtt{D,p} \longrightarrow \mathtt{q[left]\ @a\ ;;\ C} \mid \mathtt{q}\rrbracket \mathrel{==} \mathtt{p\ \&\ Some\ (a,B)\ //\ None}}\ \mathtt{bproj\_Left}$$

$$\frac{\mathtt{p} \neq \mathtt{q} \qquad \llbracket \mathtt{D,C} \mid \mathtt{q}\rrbracket \mathrel{==} \mathtt{B}}{\llbracket \mathtt{D,p} \longrightarrow \mathtt{q[right]\ @a\ ;;\ C} \mid \mathtt{q}\rrbracket \mathrel{==} \mathtt{p\ \&\ None\ //\ Some\ (a,B)}}\ \mathtt{bproj\_Right}$$

$$\frac{\mathtt{p} \neq \mathtt{r} \qquad \mathtt{q} \neq \mathtt{r} \qquad \llbracket \mathtt{D,C} \mid \mathtt{r}\rrbracket \mathrel{==} \mathtt{B}}{\llbracket \mathtt{D,p} \longrightarrow \mathtt{q[l]\ @a\ ;;\ C} \mid \mathtt{r}\rrbracket \mathrel{==} \mathtt{B}}\ \mathtt{bproj\_Sel}$$

$$\frac{\llbracket \mathtt{D,C1} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{B1} \qquad \llbracket \mathtt{D,C2} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{B2}}{\llbracket \mathtt{D,If\ p??b\ Then\ C1\ Else\ C2} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{If\ b\ Then\ B1\ Else\ B2}}\ \mathtt{bproj\_Cond}$$

$$\frac{\mathtt{p} \neq \mathtt{r} \qquad \llbracket \mathtt{D,C1} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{B1} \qquad \llbracket \mathtt{D,C2} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{B2} \qquad \mathtt{B1\ [V]\ B2} \mathrel{==} \mathtt{B}}{\llbracket \mathtt{D,If\ p??b\ Then\ C1\ Else\ C2} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{B}}\ \mathtt{bproj\_Cond'}$$

$$\frac{\mathtt{In\ p\ (fst\ (D\ X))}}{\llbracket \mathtt{D,Call\ X} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{Call\ (X,p)}}\ \mathtt{bproj\_Call\_in}$$

$$\frac{\sim\mathtt{In\ p\ (fst\ (D\ X))}}{\llbracket \mathtt{D,Call\ X} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{End}}\ \mathtt{bproj\_Call\_out}$$

$$\frac{\mathtt{In\ p\ ps}}{\llbracket \mathtt{D,RT\_Call\ X\ ps\ C} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{Call\ (X,p)}}\ \mathtt{bproj\_RT\_Call\_in}$$

$$\frac{\sim\mathtt{In\ p\ ps} \qquad \llbracket \mathtt{D,C} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{B}}{\llbracket \mathtt{D,RT\_Call\ X\ ps\ C} \mid \mathtt{p}\rrbracket \mathrel{==} \mathtt{B}}\ \mathtt{bproj\_Call\_out}$$

Fig. 13 Rules for projecting a choreography for a given target process. The notations are the ones printed by Coq, but they are not parsable due to the the different signatures

A program is projectable if the main choreography is projectable for all its processes and the set of procedure definitions is projectable.

```
Definition projectable_P P :=
 projectable_C (Procedures P) (Main P) (CCP_pn P) ∧
 projectable_D (Procedures P).
```

Finally, we want to compute projections, which are again partial functions. Since our ultimate goal is to extract a correct implementation of EPP, we need to take a different approach to partiality and define

```
Definition epp_C D ps C : projectable_C D C ps → Network Sig'.
Definition epp_D D : projectable_D D → DefSetB Sig'.
Definition epp P : projectable_P P → Program Sig'.
```

taking a proof term as additional argument (for which we prove proof irrelevance). These definitions are interactive, so we also state and prove lemmas showing that they yield the expected results as in pen-and-paper presentations [16, 41].

```
Lemma epp_C_Com_p : In p ps →
  epp_C D ps (p #e ⟶ q$x @a;;C) HC p = q!e @!a; epp_C D ps C HC' p.
```

Paving the way for the EPP theorem, we prove a number of inversion lemmas for EPP, which cannot be trivially obtained by applying inversion to a hypothesis.

The proofs follow the structure of the interactive definition, possibly combined with induction on the choreography.

```
Lemma epp_C_not_Branching_None_None : epp_C D ps C HC p ≠ q & None // None.

Lemma epp_C_Sel_Branching_l : epp_C D ps C HC p = q(+)left @+a Bp →
  epp_C D ps C HC q = p & Bl // Br → Bl ≠ None ∧ Br = None.

Lemma epp_C_Cond_Send_inv : epp_C D ps (If p ?? b Then C1 Else C2) HC r = q!e @!a B →
  ∃ B1 B2, epp_C D ps C1 HC1 r = q!e @!a B1
  ∧ epp_C D ps C2 HC2 r = q!e @!a B2 ∧ B1 [V] B2 == B.
```

5.4 Strong Projectability

The operational correspondence between choreographies and their projections, which is the topic of Sect. 6, states that a projectable choreography can make a transition iff its projection can make a corresponding transition. Generalising this result to multi-step transitions requires chaining applications of this correspondence. However, projectability is not preserved by transitions, due to how runtime terms are projected: `RT_Call X ps C'` is projected as `Call (X,p)` if `p` is in `ps`, and as the projection of `C'` otherwise. Our definition of projectability allows `C'` to be unprojectable for any process in `ps`, which would make the result of the latter transition unprojectable.

This situation can never arise if one respects the intended usage of runtime terms: initially `C'` is the body of a procedure, and `ps` is the set of processes used in it. Afterwards `ps` only shrinks, while `C'` may change due to execution of actions that involve processes not in `ps` (which keeps `C'` projectable). This assumption is implicit in pen-and-paper presentations. We formalise it in the following definition of strong projectability.

```
Fixpoint str_proj D (C:Choreography Sig) (r:Pid) : Prop :=
match C with
| eta @a;; C' ⇒ str_proj D C' r
| If p ?? b Then C1 Else C2 ⇒ str_proj D C1 r ∧ str_proj D C2 r ∧ projectable_B D C r
| RT_Call X ps C ⇒ str_proj D C r ∧ (∀ p, In p ps →
                    ∀ B B', [[D,snd (D X) | p]] == B → [[D,C | p]] == B' → B [≫] B')
| _ ⇒ True
end.
```

The last conjunct in the case of conditional is needed to guarantee that strong projectability implies projectability. The last conjunct in the case of runtime terms captures the notion that

C' may differ from the original definition of procedure X, but the transitions in the reduction path did not involve processes that still have to execute the procedure call.

Projectability and strong projectability coincide for initial choreographies. Furthermore, we state and prove lemmas that show that `str_proj D C r` implies both `projectable_B D C r` and `str_proj D C'r` for any choreography C' that C can transition to. (This is the reason for including the last conjunct in the clause defining strong projectability of conditionals: without it, we still would not be able to prove that `projectable_B D C'r`.)

Strong projectability for programs requires as expected that all choreographies in the program be strongly projectable. Furthermore, we also require the program to be well-formed. This assumption makes the definition simpler and more manageable, as all procedures will be initial and annotated with the right sets of processes.

```
Definition str_proj_P P := Program_WF P ∧ projectable_D (Procedures P) ∧
  ∀ r, str_proj (Procedures P) (Main P) r.
```

Using these results, we can start relating the semantics of choreographies with the definition of EPP. For example, if C can execute a communication from p to q, then the behaviour of its projection for p starts by sending the corresponding expression to q, while q's behaviour starts by receiving a value from p.

```
Lemma CCC_To_bproj_Com_p :
  str_proj D C p → ≪C,s≫ —[RL_Com p v q x,D]⟶ ≪C',s'≫ →
  ∃ e a Bp, [[D,C | p]] == Send Sig' q e a Bp ∧ [[D,C' | p]] == Bp
            ∧ v = eval_on_state Ev e s p.

Lemma CCC_To_bproj_Com_q :
  str_proj D C q → ≪C,s≫ —[RL_Com p v q x,D]⟶ ≪C',s'≫ →
  p ≠ q → ∃ a Bq, [[D,C | q]] == Recv Sig' p x a Bq ∧ [[D,C' | q]] == Bq.
```

An interesting corner case is what happens for processes not involved in the transition: they may lose some subbehaviours in branching terms due to some branches of conditionals disappearing from the choreography.

```
Lemma CCC_To_bproj_disjoint :
  (∀ X, CCC_pn (snd (D X)) (Names D) [C] fst (D X)) →
  str_proj D C p → disjoint_p_rl p tl → ≪C,s≫ —[tl,D]⟶ ≪C',s'≫ →
  ∃ B B', [[D,C | p]] == B ∧ [[D,C' | p]] == B' ∧ B [≫] B'.
```

The first hypothesis states that the procedures in D are well-annotated.

All these proofs use induction on the choreography. As a consequence of these lemmas, we get that strong projectability is preserved by transitions.

```
Lemma CCC_To_str_proj :
  (∀ p, str_proj D C p) →
  (∀ p Y, str_proj D (snd (D Y)) p) →
  (∀ Y, CCC_pn (snd (D Y)) (Names D) [C] fst (D Y)) →
  ≪C,s≫ —[t,D]⟶ ≪C',s'≫ → ∀ p, str_proj D C' p.

Lemma CCP_To_str_proj : str_proj_P P → (P,s) —[tl]⟶ (P',s') → str_proj_P P'.
```

The hypotheses of the first lemma all hold if (D,C) is a strongly projectable program.

5.5 Discussion

Modelling of partial functions. The definition of `merge` very explicitly considers the $2^4 = 16$ possible combinations of behaviours that can be offered when both arguments are branching terms. This clearly does not scale if the set of labels is larger, and it is the place where our design choice of fixing it to a two-element set is most critical. The same issue, but with a smaller impact, arises in the definition of `more_branches`, which includes $2^2 = 4$ clauses related to branching terms, and in the definition of `bproj`, which includes one clause for each branching term.

We do not think that this issue can be circumvented. Our original approach considered an unspecified type `Label:DecType`, and branching terms had type

`Branching : Pid → (Label → option (Ann*Behaviour)) → Behaviour`

This definition quickly proved unusable in practice: the induction principles generated by Coq were too weak, and most datatypes related to the process calculus had an undecidable equality. Furthermore we ran into problems with definitions that required inspecting the behaviours associated to the labels because of the size restrictions in elimination combinators.

The first time we managed to have a working definition of `merge` was after fixing the set of labels to contain two elements. This was the approach presented in [20], where `merge` is formalised by first defining a total function

`Xmerge : XBehaviour → XBehaviour → XBehaviour`

(where `XBehaviour` is a type including `XUndefined` subterms with the obvious intended meaning), and then defining `merge B1 B2` as `Xmerge (inject B1) (inject B2)`, where `inject` is the trivial injection from `Behaviour` to `XBehaviour`.

Apart from the added complexity of having duplications of types and definitions throughout the formalisation, working with these functions is very cumbersome. The definition of `Xmerge` relies heavily on deciding equalities, so proofs of results about `Xmerge` necessarily had to perform the same eliminations. At the end of the day, the number of cases in proofs was in the same order of magnitude as in the current version—but they were generated in several verbose elimination steps, rather than directly from performing induction/inversion on a hypothesis. Furthermore, the old definition required us to consider a significant number of absurd cases (in some lemmas, around 90% of the total), whereas with the current definition these cases are simply not generated. The only added complexity we noticed while adapting the formalisation was that we occasionally needed to apply lemma `merge_unique` to infer that two behaviours are identical—but the size of this part of the formalisation was reduced by about 80% (from around 3150 lines down to 700 lines).

Taking all these aspects into account, we believe that the current design choices are the best possible compromise at this stage between the full generality given by including an unrestricted set of labels and the benefits of having a fully formalised theory.

Projectability. The lemmas relating projectability to the low-level semantics of choreographies typically include several hypotheses, cf. lemma `CCC_To_str_proj`. For programs, we packaged these properties in a single definition (`str_proj_P`). For the lower-level lemmas, we decided against this because not all these properties are needed in all lemmas—some are only required in results involving procedure calls, others are important for conditionals, and communications require far fewer. By including only the necessary assumptions in each lemma, we obtain more robust results.

Strong projectability. The need for strong projectability was independently identified in the pen-and-paper presentation in [41]. There, the projectability requirement on runtime terms was included in the notion of well-formedness for choreographies. While this option matches the intuition of "intended usage of runtime terms", it requires having defined projection. In our formalisation, we strive for modularity, and we opted for a design where the choreographic calculus is fully decoupled from the target language and the definition of projection. In this way, we allow for future extensions of our development with alternative definitions of EPP.

In the future, it would be interesting to investigate whether there is a syntactic characterisation of "intended usage of runtime terms" that is completely at the level of choreographies. Such a characterisation would yield the benefits of both approaches described above: it would give us a notion of well-formedness closer to intuition, while keeping it decoupled from EPP.

Summary. The definitions of branching order, merge and Endpoint Projection, together with the accompanying lemmas, are divided in three files totaling 14 definitions, 126 lemmas and 15 tactics to automate recurring types of goals. By far the largest bulk is the formalisation of EPP, at over 2200 lines of Coq code (with approximately 100 lemmas), while the branching order and merge require respectively 260 and 440 lines of Coq code (with a total of only 3 results that require longer proofs).

6 The EPP Theorem

The operational correspondence between choreographies and their projections, in languages that include both conditionals and out-of-order execution, is not as straightforward as for the simple language in Sect. 2. In particular, branching terms in networks may linger for a bit longer compared to the choreographies that generated them. This requires referring to the branching order in the EPP theorem:

```
Lemma EPP_Complete : str_proj_P P → (P,s) —[tl]⟶ (P',s') →
  ∃ N tl', (epp P HP,s) —[tl']⟶ (N,s')
  ∧ Procs N = Procs (epp P HP) ∧ ∀ HP', Net N (≫) Net (epp P' HP').
```

```
Lemma EPP_Sound : str_proj_P P → (epp P HP,s) —[tl]⟶ (N',s') →
  ∃ P' tl', (P,s) —[tl']⟶ (P',s') ∧ ∀ HP', Net N' (≫) Net (epp P' HP').
```

(Recall that `epp` takes a proof of projectability as its last argument.)

Completeness is not too hard to prove. As in [16, 41], the result is proven by considering the possible transitions that `Main P` can make; there are four cases, and the results proved earlier about the shape of the projection of `P` suffice to establish the thesis without too much work. The whole proof is 250 lines long, and the generalisation to multi-step transitions requires an additional 40 lines.

The proof of soundness is known to be harder [8, 16, 40, 41]. A common strategy is to proceed by induction on the choreography, and then do case analysis on the possible network transition. The latter is either the first term in the choreography, and we can apply the matching choreography rule; or it is not, and we can apply a delay rule and invoke the induction hypothesis.

Each of these cases is challenging in itself, and they are therefore stated as separate lemmas on transitions. As an example, the transition lemma for communications reads

```
Lemma SP_To_bproj_Com : str_proj_P (D,C) →
  ≪epp_C D ps C HC,s≫ —[RL_Com p v q x,D']⟶ ≪N',s'≫ →
  ∃ C', ≪C,s≫ —[RL_Com p v q x,D]⟶ ≪C',s'≫ ∧ ∀ HC', (N' (==) (epp_C D ps C' HC')).
```

and the corresponding proof script is around 320 lines long. There are five of these lemmas in total, of a similar level of complexity.

Soundness also requires an additional lemma on procedure calls:

```
Lemma SP_To_bproj_Call_name : ≪epp_C D ps C HC,s≫ —[RL_Call X p,D']⟶ ≪N',s'≫ →
  ∃ (Y:RecVar), X = (Y,p) ∧ X_Free _ Y C.
```

which is needed to apply the corresponding transition lemma.

Chaining applications of `EPP_Sound` also requires that extending the projection of a choreography with extra branches does not add transitions.

```
Lemma SP_To_MBN_epp : N1 (≫) epp_C D' ps C HC → ≪N1,s≫ —[tl,D]⟶ ≪N2,s'≫ →
 ∃ N2', ≪epp_C D' ps C HC,s≫ —[tl,D]⟶ ≪N2',s'≫ ∧ N2 (≫) N2'.
```

This result is lifted to `SPP_To` and `SPP_ToStar`. The latter generalisation requires applying `EPP_Sound`. It is then itself used to prove soundness of EPP for multi-step transitions.

The proof of the EPP theorem consists of an additional 2650 lines of Coq code, for only 14 lemmas.

7 Related Work

The need for formalising concurrency theory is identified in [39], where the authors formalised a published article on a process calculus in Coq and discovered several major flaws in the proofs. The authors

> […] feel that it is [the errors'] very presence in a peer-reviewed, state-of-the-art paper that strongly underlines the need for a more precise formal treatment of proofs in this domain. [39, Sect. 6]

Since then, there have been a number of formalisation efforts in this area. We discuss the ones closest to our work.

To the best of our knowledge, our original presentations [20, 21] were the first formalisations of a choreographic language featuring the expected programming constructs that allow for infinite and branching concurrent behaviour. As we discussed, this article presents a substantial improvement of the original development.

More recently, there have been two additions to the family of fully-formalised choreographic programming languages.

Kalas is a certified compiler written in HOL from a choreographic language similar to ours to CakeML [44].

As in our development, the set of selection labels in Kalas is restricted to two. Kalas includes an asynchronous semantics, while ours is synchronous, but the notion of EPP is more restrictive than ours: it is an *ad-hoc* definition that bypasses the need for the merge operator, but does not provide its full flexibility. In particular, processes evaluating conditionals must immediately send selections to the processes that need them, while CC is more faithful to the pen-and-paper literature on choreographies [7, 8, 30].

Example 14 To illustrate the flexibility of our projection, consider the following enhanced version of our distributed authentication choreography from Example 8, where `ip` now immediately communicates whether the authentication attempt was successful to a `logger`.

```
c#credentials ⟶ ip$x;;
If ip ?? (check x) Then
  ip#(valid x) ⟶ logger.y;;
  ip ⟶ s[left];; ip ⟶ c[left];; s#token ⟶ c$t;; End
Else
  ip#(invalid x) ⟶ logger.y;;
  ip ⟶ s[right];; ip ⟶ c[right];; End
```

This choreography is projectable in our framework but not in Kalas, because `ip` performs does not immediately engage in selections in the branches of the conditional. However, doing so would require postponing the logging action, which might be important to do right away because of non-functional requirements.

Pirouette is a functional choreographic programming language formalised in Coq [29]. It supports asynchronous communication and higher-order functions, but at the cost of introducing hidden global synchronisations for *all* processes whenever a function is called. The semantics of CC is, instead, decentralised and all synchronisations are syntactically explicit—but again it only has synchronous semantics and no higher-order features.

Extending CC with asynchronous communication has also been studied [12], but since it was not part of the reference pen-and-paper work that we followed, we postponed its formalisation to future work.

Another line of research connected to choreographic programming is that of multiparty session types [30]. These types are essentially choreographies without computation (e.g., communications only specify sender, receiver, and message type, but not how the message is computed or where it is stored), and are therefore simpler than CC. There are two available formalisations of multiparty session types [10, 33]. Both formalisations include a counterpart to the EPP theorem, but they are even more restrictive than Kalas in how they handle the projection of conditionals.

8 Conclusion

We presented a formalisation of a state-of-the-art article on theory of choreographic programming. The formalisation process unveiled subtle problems in definitions, making a case for a more systematic use of theorem provers to validate results in the field. Even more, it positively impacted the theory itself, showing that formalisation can be valuable tool also in the design phase of the research process.

Our formalisation was done in parallel with the pen-and-paper revision of CC carried out in [41]. There are two interesting observations to make about this parallel development. First, many of the technical aspects that we discuss in this article were also independently discovered during the writing of [41]. Second, the seemingly disparate goals of making the theory more intuitive to students and amenable to formalisation actually converged on the same solution, and sometimes resulted in useful exchanges of feedback. Taken together, these two observations strongly suggest that the current formulation of CC is the “right” one, and offers a suitable basis for future developments.

Our work also provides some valuable lessons about formalising semantics of concurrent systems. While choosing between a reduction semantics with a structural precongruence for dealing with out-of-order execution or a transition semantics based on a labelled transition system was mostly a matter of taste in pen-and-paper presentations, the latter approach is clearly preferable from a formalisation point of view. Since it does not require syntactic manipulation of choreographies for modelling transitions, the derivations corresponding to execution steps are shorter and do not include potential redundancy, which makes it easier to reason about them and to find appropriate induction hypotheses.

Our formalisation further benefits from the design choice of defining all procedures at the top level, which allows us to bypass all the complexity of having to work explicitly with binders and substitution.

We have already started exploring extensions and applications of our formalisation. These include amendment (a procedure that injects appropriate selections to make a choreography projectable), a proof of starvation-freedom, alternative definitions of EPP, and applying program extraction to develop a certified toolchain from choreographies to executable code.

An important tool for future extensions is stronger automation for proofs about choreographies. Our development already includes a few simple tactics that deal with commonly-recurring goals, but it would be worthwhile to extend this library with more powerful tactics, e.g., to reason about multi-step transitions. Furthermore, for many proofs by structural induction, there are strong similarities among their different cases, and it would be interesting to try to automate proof strategies that can capitalise on this.

The appeal of choreographic programming largely depends on its promise of delivering correct implementations, by removing the possibility of human error through EPP. This promise has motivated a proliferation of choreographic programming languages, including features of practical value such as asynchronous communication, nondeterminism, broadcast, dynamic network topologies, and more [2, 26, 31, 41]. The theories of these languages are becoming more and more complex, thus increasing the likelihood of critical mistakes and making the case for more trustworthy developments. We hope that our work can contribute a solid foundation for the development of these features.

Acknowledgements The authors would like to thank the anonymous reviewers for their comments and suggestions, which contributed to increasing the quality of this article. This work was partially supported by Villum Fonden, grant nr 29518, and the Independent Research Fund Denmark, grant nr 0135-00219.

Funding Open access funding provided by Royal Danish Library.

References

1. Albert, E., Lanese, I. (eds.): Formal Techniques for Distributed Objects, Components, and Systems—36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, 6–9 June 2016, Proceedings. Lecture Notes in Computer Science, vol. 9688. Springer, Berlin (2016)
2. Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P., Gay, S.J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E.B., Martins, F., Mascardi, V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V.T., Yoshida, N.: Behavioral types in programming languages. Found. Trends Program. Lang. **3**(2–3), 95–230 (2016)
3. Bravetti, M., Carbone, M., Zavattaro, G.: Undecidability of asynchronous session subtyping. Inf. Comput. **256**, 300–320 (2017). https://doi.org/10.1016/j.ic.2017.07.010
4. Bravetti, M., Carbone, M., Zavattaro, G.: On the boundary between decidability and undecidability of asynchronous session subtyping. Theor. Comput. Sci. **722**, 19–51 (2018). https://doi.org/10.1016/j.tcs.2018.02.010
5. Bravetti, M., Carbone, M., Lange, J., Yoshida, N., Zavattaro, G.: A sound algorithm for asynchronous session subtyping and its implementation. Log. Methods Comput. Sci. (2021). https://lmcs.episciences.org/7238
6. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) Proc. CONCUR. Lecture Notes in Computer Science, vol. 6269, pp. 222–236. Springer, Berlin (2010). https://doi.org/10.1007/978-3-642-15375-4_16
7. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: Giacobazzi, R., Cousot, R. (eds.) Procs. POPL, pp. 263–274. ACM, New York (2013). https://doi.org/10.1145/2429069.2429101

8. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. **34**(2), 8:1-8:78 (2012). https://doi.org/10.1145/2220365.2220367
9. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multi-party sessions. In: Bruni, R., Dingel, J. (eds.) Procs. FORTE. LNCS, vol. 6722, pp. 1–28. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-21461-5_1
10. Castro-Perez, D., Ferreira, F., Gheri, L., Yoshida, N.: Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In: Freund, S.N., Yahav, E. (eds.) Procs. PLDI, pp. 237–251. ACM, New York (2021). https://doi.org/10.1145/3453483.3454041
11. Cruz-Filipe, L., Montesi, F.: Choreographies in practice. In: Albert, E., Lanese, I. (eds.): Formal Techniques for Distributed Objects, Components, and Systems—36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, 6–9 June 2016, Proceedings, pp. 114–123. https://doi.org/10.1007/978-3-319-39570-8_8
12. Cruz-Filipe, L., Montesi, F.: On asynchrony and choreographies. In: Bartoletti, M., Bocchi, L., Henrio, L., Knight, S. (eds.) Procs. ICE, EPTCS, vol. 261, pp. 76–90 (2017). https://doi.org/10.4204/EPTCS.261.8
13. Cruz-Filipe, L., Montesi, F.: Procedural choreographic programming. In: Bouajjani, A., Silva, A. (eds.) Procs. FORTE. Lecture Notes in Computer Science, vol. 10321, pp. 92–107. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-60225-7_7
14. Cruz-Filipe, L., Lugović, L., Montesi, F.: Certified compilation of choreographies with HACC. CoRR. (2023). https://doi.org/10.48550/arXiv.2303.03972
15. Cruz-Filipe, L., Montesi, F., Rasmussen, R.R.: Keep me out of the loop: a more flexible choreographic projection. Submitted for publication
16. Cruz-Filipe, L., Montesi, F.: A core model for choreographic programming. Theor. Comput. Sci. **802**, 38–66 (2020). https://doi.org/10.1016/j.tcs.2019.07.005
17. Cruz-Filipe, L., Montesi, F.: Now it compiles! certified automatic repair of uncompilable protocols. CoRR. (2023). https://doi.org/10.48550/arXiv.2302.14622
18. Cruz-Filipe, L., Larsen, K.S., Montesi, F.: The paths to choreography extraction. In: Foundations of Software Science and Computation Structures—20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings, LNCS, vol. 10203, pp. 424–440. https://doi.org/10.1007/978-3-662-54458-7_25
19. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Choreographies in Coq. In: TYPES 2019, Abstracts (2019). Extended abstract
20. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Certifying choreography compilation. In: Cerone, A., Ölveczky, P.C. (eds.) Procs. ICTAC, LNCS, vol. 12819, pp. 115–133. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-85315-0_8
21. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Formalising a Turing-complete choreographic language in Coq. In: Cohen, L., Kaliszyk, C. (eds.) Procs. ITP, LIPIcs, vol. 193, pp. 15:1–15:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Wadern (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.15
22. Cruz-Filipe, L., Montesi, F., Peressotti, M.: A formal theory of choreographic programming in Coq (2022). https://doi.org/10.5281/zenodo.7773479
23. Dalla Preda, M., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J.: Dynamic choreographies: Theory and implementation. Log. Methods Comput. Sci. (2017). https://doi.org/10.23638/LMCS-13(2:1)2017
24. Esparza, J., Murawski, A.S. (eds.): Foundations of Software Science and Computation Structures—20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, 22–29 April 2017, Proceedings, LNCS, vol. 10203 (2017)
25. Finkel, A., Lozes, É.: Synchronizability of communicating finite state machines is not decidable. In: Chatzigiannakis, I., Indyk, P., Kuhn, F., Muscholl, A. (eds.) Procs. ICALP, LIPIcs, vol. 80, pp. 122:1–122:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern (2017). https://doi.org/10.4230/LIPIcs.ICALP.2017.122
26. Gay, S.J., Vasconcelos, V.T., Wadler, P., Yoshida, N.: Theory and applications of behavioural types (Dagstuhl seminar 17051). Dagstuhl Rep. **7**(1), 158–189 (2017). https://doi.org/10.4230/DagRep.7.1.158
27. Giallorenzo, S., Lanese, I., Russo, D.: Chip: A choreographic integration process. In: Panetto, H., Debruyne, C., Proper, H.A., Ardagna, C.A., Roman, D., Meersman, R. (eds.) Procs. OTM, part II. Lecture Notes in Computer Science, vol. 11230, pp. 22–40. Springer, Berlin (2018). https://doi.org/10.1007/978-3-030-02671-4_2

28. Giallorenzo, S., Montesi, F., Peressotti, M.: Choreographies as objects. CoRR. (2020). https://arxiv.org/abs/2005.09520
29. Hirsch, A.K., Garg, D.: Pirouette: higher-order typed functional choreographies. Proc. ACM Program. Lang. **6**(POPL), 1–27 (2022). https://doi.org/10.1145/3498684
30. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM (2016). https://doi.org/10.1145/2827695. Also: POPL, pp. 273–284 (2008)
31. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1-3:36 (2016). https://doi.org/10.1145/2873052
32. Intl. Telecommunication Union: Recommendation Z.120: Message Sequence Chart (1996)
33. Jacobs, J., Balzer, S., Krebbers, R.: Multiparty GV: functional multiparty session types with certified deadlock freedom. Proc. ACM Program. Lang. **6**(ICFP), 466–495 (2022). https://doi.org/10.1145/3547638
34. Kleene, S.C.: Introduction to Metamathematics, vol. 1. North-Holland, Amsterdam (1952)
35. Lange, J., Yoshida, N.: On the undecidability of asynchronous session subtyping. In: Esparza, J., Murawski, A.S. (eds.) Foundations of Software Science and Computation Structures—20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, 22–29 April 2017, Proceedings, pp. 441–457. https://doi.org/10.1007/978-3-662-54458-7_26
36. Lluch-Lafuente, A., Nielson, F., Nielson, H.R.: Discretionary information flow control for interaction-oriented specifications. In: Martí-Oliet, N., Ölveczky, P.C., Talcott, C.L. (eds.) Logic, Rewriting, and Concurrency. Lecture Notes in Computer Science, vol. 9200, pp. 427–450. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-23165-5_20
37. López, H.A., Heussen, K.: Choreographing cyber-physical distributed control systems for the energy sector. In: Seffah, A., Penzenstadler, B., Alves, C., Peng, X. (eds.) Procs. SAC, pp. 437–443. ACM, New York (2017). https://doi.org/10.1145/3019612.3019656
38. López, H.A., Nielson, F., Nielson, H.R.: Enforcing availability in failure-aware communicating systems. In: Albert, E., Lanese, I. (eds.) Formal Techniques for Distributed Objects, Components, and Systems—36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, 6–9 June 2016, Proceedings, pp. 195–211. https://doi.org/10.1007/978-3-319-39570-8_13
39. Maksimovic, P., Schmitt, A.: HOCore in Coq. In: Urban, C., Zhang, X. (eds.) Procs. ITP. Lecture Notes in Computer Science, vol. 9236, pp. 278–293. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-22102-1_19
40. Montesi, F.: Choreographic programming. Ph.D. Thesis, IT University of Copenhagen (2013). http://www.fabriziomontesi.com/files/choreographic_programming.pdf
41. Montesi, F.: Introduction to Choreographies. Cambridge University Press, Cambridge (2023)
42. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM **21**(12), 993–999 (1978). https://doi.org/10.1145/359657.359659
43. Object Management Group: Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0/ (2011)
44. Pohjola, J.Å., Gómez-Londoño, A., Shaker, J., Norrish, M.: Kalas: a verified, end-to-end compiler for a choreographic language. In: Andronick, J., de Moura, L. (eds.) Procs. ITP, LIPIcs, vol. 237, pp. 27:1–27:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Wadern (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.27
45. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Proc. ACM Program. Lang. **3**(POPL), 30:1-30:29 (2019). https://doi.org/10.1145/3290343
46. W3C: WS Choreography Description Language. http://www.w3.org/TR/ws-cdl-10/ (2004)