On Asynchrony and Choreographies

Luís Cruz-Filipe[a,*], Fabrizio Montesi[a]

[a]*Department of Mathematics and Computer Science, University of Southern Denmark*

Abstract

Choreographic Programming is a paradigm for the development of concurrent software, where deadlocks are prevented syntactically and then correct-by-construction implementations in process models are mechanically generated. The formal semantics of choreography languages are typically based on synchronous communications, in order to achieve a simpler theory. However, many real-world systems have asynchronous communications. Previous attempts at enriching choreographies with asynchrony rely on *ad-hoc* constructions, whose adequacy is only argued informally.

In this work, we present a systematic study of asynchronous communications in choreographies. First, we discuss and formalise the properties that an asynchronous semantics for choreographies should have. Then, we explore how out-of-order execution, previously used in choreographies for modelling concurrency, can be used to endow choreographies with an asynchronous semantics that satisfies our properties. Our development yields a pleasant correspondence with FIFO-based asynchronous messaging, modelled in a process calculus. Finally, we investigate the expressivity of choreography languages with respect to asynchrony. Specifically, we find out that choreography languages equipped with process spawning and name mobility primitives are expressive enough to program asynchronous behaviour over a simple synchronous semantics.

Keywords: choreographic programming; asynchrony; concurrent programming

1. Introduction

Choreographic Programming [26] is a paradigm for developing concurrent software, where an "Alice and Bob" notation is used to prevent mismatched I/O actions syntactically. An EndPoint Projection (EPP) can then be used to synthesise correct-by-construction process implementations. Choreographies are used in different settings, including standards [3, 32], languages [6, 20, 28, 31], specification models [4, 5, 23, 29], and design tools [3, 28, 31, 32].

The key to preventing mismatched I/O actions in choreographies is that interactions between two (or more) processes are specified atomically, using terms such as $\mathsf{p}.e \texttt{ -> } \mathsf{q}; C$ – read "process p sends the evaluation of expression e to process q, and then we proceed as the choreography C". Giving a semantics to such terms is relatively easy if we assume that communications are synchronous: we can just reduce $\mathsf{p}.e \texttt{ -> } \mathsf{q}; C$ to C in a single step (and update q's state with the received value, but this is orthogonal to this discussion). For this reason, most research on choreographic programming focused on systems with a synchronous communications semantics.

However, many real-world systems use asynchronous communications. This motivated the introduction of an ad-hoc reduction rule for modelling asynchrony in choreographies (Rule $\lfloor$C|Async$\rceil$ in [5]). As an example, consider the choreography $\mathsf{p}.1 \texttt{ -> } \mathsf{q}; \mathsf{p}.2 \texttt{ -> } \mathsf{r}$ (where 1 and 2 are just constants). The special rule would allow for consuming the second communication immediately, thus reducing the choreography to $\mathsf{p}.1 \texttt{ -> } \mathsf{q}$. In general, roughly, this rule allows a choreography of the form $\mathsf{p}.e \texttt{ -> } \mathsf{q}; C$ to execute an action in C if this action involves p but not q (sends are non-blocking, receives are blocking). This approach was later

[*]Corresponding author; tel. +45 6550 2387, fax +45 6550 2373

adopted in other works (see Section 6). Unfortunately, it also comes with a problem: it yields an unintuitive semantics, since a choreography can now reduce to a state that would normally not be reachable in the real world. In our example, specifically, in the real world p would have to send its first message to q before it could proceed to sending its other message to r. This information is lost in the choreography reduction, where it appears that p can just send its messages in any order. In [5], this also translates to a misalignment between the structures of choreographies and their process implementations generated by EPP, since the latter use a standard asynchronous semantics with message buffers; see Section 6 for a detailed discussion of this aspect. Previous work [17] uses intermediate runtime terms in choreographies to represent asynchronous messages in transit, in an attempt at overcoming this problem. However, the adequacy of this approach has never been formally demonstrated.

In this work, we are interested in studying asynchrony for choreographies in a systematic way. Thus, we first analyse the properties that an asynchronous choreography semantics should have. Assuming assuming standard FIFO duplex channels between each pair of processes, messages can be sent without the intended receiver being ready, and sent messages are eventually received. We formulate these properties precisely in a representative choreography language.

Our study leads naturally to the construction of a new choreography model that supports asynchronous communications, by capitalising on the characteristic feature of out-of-order execution found in choreographic programming. We formally establish the adequacy of our asynchronous model, by proving that it respects the formal definitions of our properties. Then, we define an EPP from our new model to an asynchronous process calculus. Thanks to the accurate asynchronous semantics of our choreography model, we prove that the code generated by our EPP and the originating choreography are lockstep operationally equivalent. As a corollary, our generated processes are deadlock-free by construction. Our development also has the pleasant property that programmers do not need to reason about asynchrony: they can just program thinking in the usual terms of synchronous communications, and assume that adopting asynchronous communications will not lead to errors.

Finally, we leverage our new knowledge on asynchronous choreography semantics to study the expressivity of choreography languages with respect to asynchrony. In other words, we want to know when a choreography model is expressive enough to encode asynchronous behaviour using its existing semantics. We find out that choreography languages equipped with process spawning and name mobility enjoy this property, by presenting an endo-encoding that rewrites a choreography to one where all communications are implemented asynchronously. We conclude by linking our two developments, showing that the encoding produces correct asynchronous behaviour according to our formal notion of asynchrony for choreographies.

Contributions.. This article presents four main contributions. First, we give an abstract characterisation of asynchronous semantics for choreography languages, which is formalised for a minimal choreography calculus. Secondly, we propose an asynchronous semantics for this minimal language, show that it is an instance of our characterisation, and discuss how it can be applied to other choreography calculi. Thirdly, we prove a lockstep operational correspondence between choreographies and their process implementations, when asynchronous semantics for both systems are considered. Finally, we show that choreography languages equipped with the features of process spawning and name mobility (which our minimal language does not possess) are expressive enough to simulate asynchronous communications as we formalised.

Publication history.. This article combines and extends material and results from previous publications. The abstract characterization of asynchrony (Section 3) and the asynchronous semantics for choreographies (Section 4) were originally introduced in [13]. The encoding of asynchronous communications using process spawning was described in [10], while the full extension to the calculi DMC and DCC (Section 5) was first presented in [12]. Theorems 9 and 10, which connect this encoding to the abstract characterization from Section 3, are new, as well as the material in Section 5.4. The discussion on the impact of our results and comparison to related work has also been expanded from the previous presentations.

Structure.. We present the representative choreography language MC in which we develop our work, together with its associated process calculus and EPP, in Section 2. In Section 3, we motivate and introduce

$$C ::= \mathsf{p}.e \;\texttt{->}\; \mathsf{q}; C \mid \mathsf{p} \;\texttt{->}\; \mathsf{q}[\ell]; C \mid \mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1 \mid X \mid \mathbf{0}$$

Figure 1: Core Choreographies, syntax.

the properties we would expect of an asynchronous choreography semantics. Section 4 introduces a semantics for MC that satisfies these properties. We also show that we can define an asynchronous variant of the target process calculus and extend the definition of EPP towards it, preserving the precise operational correspondence from the synchronous case. In Section 5, we present a syntactic approach to asynchrony, showing how to encode asynchronous communications in choreography languages that support process spawning and name mobility. We relate our development to other approaches for asynchrony in choreographies in Section 6, before concluding in Section 7 with a discussion on the implications of our work and possible future directions.

2. Core Choreographies and Stateful Processes

We review the choreography model of Core Choreographies (CC) and its target calculus of Stateful Processes (SP), originally introduced in [9].

2.1. Core Choreographies

The language of CC is defined inductively in Figure 1. Processes, ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$, are assumed to run concurrently. Processes interact through two types of communication actions, which we collectively denote by η. In value communications $\mathsf{p}.e \;\texttt{->}\; \mathsf{q}$, process p evaluates expression e and sends the result to q. The precise syntax of expressions is immaterial for our presentation; in particular, expressions can access values stored in p's memory. In label selections $\mathsf{p} \;\texttt{->}\; \mathsf{q}[\ell]$ process p selects, from several possible behaviours of q, the one identified by label ℓ. We abstract over the set of labels L, requiring only that it be finite and contain at least two elements.

The remaining choreography terms denote conditionals, recursion, and termination. In the conditional $\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, process p evaluates expression e to decide whether the choreography should proceed as C_1 or as C_2. In $\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1$, we define variable X to be the choreography term C_2, which then can be called (as X) inside both C_1 and C_2. Term $\mathbf{0}$ is the terminated choreography, which we sometimes omit. For a more detailed discussion of these primitives, we refer the reader to [9].[1]

The (synchronous) semantics of CC is a reduction semantics that uses a total state function σ to represent the memory state at each process p. Since our development is orthogonal to the details of the memory implementation, we say that $\sigma(\mathsf{p})$ is a representation of the memory state of p (left unspecified) and write $\mathsf{upd}(\sigma, \mathsf{p}, v)$ to denote the (uniquely defined) updated memory state of p after receiving a value v. Term $e \downarrow^{\mathsf{p}}_{\sigma} v$ denotes that locally evaluating expression e at p, with memory state σ, evaluates to v. (This formulation captures the essence of previous memory models for choreographies, cf. [5, 8].)

The transitions are defined over pairs $\langle C, \sigma \rangle$, given by the rules in Figure 2. These rules are mostly standard, and we summarise their intuition. In $\lfloor$C|Com$\rceil$, the state of q is updated with the value received from p (which results from the evaluation of expression e at that process). Label selections are no-ops (rule $\lfloor$C|Sel$\rceil$), but play a role in the implementations as we discuss later. Rule $\lfloor$C|Cond$\rceil$ is as expected, while rule $\lfloor$C|Ctx$\rceil$ allows reductions under recursive definitions. Finally, rule $\lfloor$C|Struct$\rceil$ uses a structural precongruence $\preceq$, defined in Figure 3, which essentially allows (i) independent communications to be swapped (rule $\lfloor$C|Eta-Eta$\rceil$), (ii) recursive definitions to be unfolded (rule $\lfloor$C|Unfold$\rceil$), and (iii) garbage collection (rule $\lfloor$C|ProcEnd$\rceil$). These rules, taken together, endow the semantics of CC with out-of-order execution: interactions not at the top level may be brought to the top and executed if they do not interfere with other

[1]We relaxed the syntax of CC slightly with respect to [9] by leaving the syntax of expressions unspecified, which allows for the simpler conditional $\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ in line with typical choreography languages. This minor change simplifies our presentation.

$$\frac{e \downarrow^{\mathsf{p}}_{\sigma} v}{\mathsf{p}.e \;\texttt{->}\; \mathsf{q}; C, \sigma \to C, \mathsf{upd}(\sigma, \mathsf{q}, v)} \;\lfloor \mathrm{C|Com} \rceil$$

$$\frac{}{\mathsf{p} \;\texttt{->}\; \mathsf{q}[\ell]; C, \sigma \to C, \sigma} \;\lfloor \mathrm{C|Sel} \rceil$$

$$\frac{i = 1 \text{ if } e \downarrow^{\mathsf{p}}_{\sigma} \mathsf{true} \qquad i = 2 \text{ o.w.}}{\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \sigma \to C_i, \sigma} \;\lfloor \mathrm{C|Cond} \rceil$$

$$\frac{C_1, \sigma \to C_1', \sigma'}{\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1, \sigma \to \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1', \sigma'} \;\lfloor \mathrm{C|Ctx} \rceil$$

$$\frac{C_1 \preceq C_2 \qquad C_2, \sigma \to C_2', \sigma' \qquad C_2' \preceq C_1'}{C_1, \sigma \to C_1', \sigma'} \;\lfloor \mathrm{C|Struct} \rceil$$

Figure 2: Core Choreographies, synchronous semantics.

$$\frac{\mathsf{pn}(\eta) \cap \mathsf{pn}(\eta') = \emptyset}{\eta; \eta' \equiv \eta'; \eta} \;\lfloor \mathrm{C|Eta\text{-}Eta} \rceil \qquad \frac{\mathsf{pn}(C_i) \cap \mathsf{pn}(\eta) = \emptyset}{\mathsf{def}\ X = C_2\ \mathsf{in}\ (\eta; C_1) \equiv \eta; \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1} \;\lfloor \mathrm{C|Eta\text{-}Rec} \rceil$$

$$\frac{}{\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1[X] \preceq \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1[C_2]} \;\lfloor \mathrm{C|Unfold} \rceil \qquad \frac{}{\mathsf{def}\ X = C\ \mathsf{in}\ \mathbf{0} \preceq \mathbf{0}} \;\lfloor \mathrm{C|ProcEnd} \rceil$$

$$\frac{\{\mathsf{p}, \mathsf{q}\} \cap \mathsf{pn}(\eta) = \emptyset}{\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ (\eta; C_1)\ \mathsf{else}\ (\eta; C_2) \equiv \eta; \mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2} \;\lfloor \mathrm{C|Eta\text{-}Cond} \rceil$$

$$\frac{\mathsf{p} \neq \mathsf{q}}{\begin{array}{c}\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ (\mathsf{if}\ \mathsf{q}.e'\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2)\ \mathsf{else}\ (\mathsf{if}\ \mathsf{q}.e'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2') \\ \equiv \\ \mathsf{if}\ \mathsf{q}.e'\ \mathsf{then}\ (\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_1')\ \mathsf{else}\ (\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_2\ \mathsf{else}\ C_2')\end{array}} \;\lfloor \mathrm{C|Cond\text{-}Cond} \rceil$$

Figure 3: Core Choreographies, structural precongruence.

$$B ::= \mathsf{q}!e;B \mid \mathsf{p}?;B \mid \mathsf{q}\oplus\ell;B \mid \mathsf{p}\&\{\ell_i : B_i\}_{i\in I};B \mid \mathsf{if}\, e\, \mathsf{then}\, B_1\, \mathsf{else}\, B_2;B \mid \mathsf{def}\ X = B_2\ \mathsf{in}\ B_1 \mid X \mid \mathbf{0}$$

$$N, M ::= \mathsf{p} \triangleright_{\sigma_p} B \mid (N \mid M) \mid \mathbf{0}$$

Figure 4: Stateful Processes, syntax.

interactions that precede them in the choreography's abstract syntax tree. We write $C \equiv C'$ for $C \preceq C'$ and $C' \preceq C$, and denote the set of process names in a choreography C by $\mathsf{pn}(C)$.

Example 1. *We introduce the running example we use throughout this article. In the following choreography where a buyer, Alice (*a*), purchases a product from a seller (*s*) through her bank (*b*).*

```
1.  a.title -> s;
2.  s.price -> a;
3.  s.price -> b;
4.  if b.happy then
5.      b -> s[ok];  b -> a[ok];
6.      s.book -> a
7.  else  b -> s[ko];  b -> a[ko]
```

In Line 1, the term a.*title* -> s *denotes an interaction whereby* a *communicates the title of the book that Alice wishes to buy to* s*. The seller then sends the price of the book to both* a *and* b*. In Line 4,* a *sends the price she expects to pay to* b*, which confirms that it is the same amount requested by* s *(stored internally at* b*). If so,* b *notifies both* s *and* a *of the successful transaction (Line 5) and* s *sends the book to* a *(Line 6). Otherwise,* b *notifies* s *and* a *of the failure (Line 7) and the choreography terminates.*

Unsurprisingly, choreographies in CC are always deadlock-free. We use this property later on, to prove that the process code generated from choreographies is also deadlock-free.

Theorem 1 (Deadlock-freedom [9])**.** *Given a choreography C, either $C \preceq \mathbf{0}$ (termination) or, for every state σ, there exist a choreography C' and a state σ' such that $C, \sigma \to C', \sigma'$.*

2.2. Stateful Processes and EndPoint Projection

Core Choreographies are meant to be implemented in a minimalistic process calculus, also introduced in [9], called Stateful Processes (SP). We summarize this calculus, noting that we make the same conventions and changes regarding expressions, labels, and states as above.

The syntax of SP is reported in Figure 4. A term $\mathsf{p} \triangleright_{\sigma_\mathsf{p}} B$ is a process with name p, memory state σ_p and behaviour B. Networks, ranged over by N, M, are parallel compositions of processes, with $\mathbf{0}$ being the inactive network.

Behaviours correspond to the local views of choreography actions. The process executing a send term $\mathsf{q}!e;B$ evaluates expression e and sends the result to process q, proceeding as B. The dual receiving behaviour $\mathsf{p}?;B$ expects a value from process p, stores it in its memory and proceeds as B. Term $\mathsf{q} \oplus l;B$ sends a label ℓ to process q. Selections are received by the branching term $\mathsf{p}\&\{\ell_i : B_i\}_{i\in I}$, which upon receiving one of the labels ℓ_i proceeds according to B_i. Branching terms must offer at least one branch. The other terms are as in CC.

These intuitions are formalized in the synchronous semantics of SP, which is defined by the rules in Figure 5. Rule $\lfloor$P|Com$\rceil$ models synchronous value communication: a process p wishing to send a value to q can synchronise with a receive-from-p action at q, and the state of q is updated accordingly. Rule $\lfloor$P|Sel$\rceil$ is standard selection, with p selecting one of the branches offered by q. The remaining rules are standard.

$$\frac{e \downarrow_{\sigma_p} v}{\mathsf{p} \triangleright_{\sigma_\mathsf{p}} \mathsf{q}!e; B_1 \mid \mathsf{q} \triangleright_{\sigma_\mathsf{q}} \mathsf{p}?; B_2 \to \mathsf{p} \triangleright_{\sigma_\mathsf{p}} B_1 \mid \mathsf{q} \triangleright_{\mathsf{upd}(\sigma_\mathsf{q}, v)} B_2} \lfloor \text{P}|\text{Com} \rceil$$

$$\frac{j \in I}{\mathsf{p} \triangleright_{\sigma_\mathsf{p}} \mathsf{q} \oplus \ell_j; P \mid \mathsf{q} \triangleright_{\sigma_\mathsf{q}} \mathsf{p}\&\{\ell_i : Q_i\}_{i \in I} \to \mathsf{p} \triangleright_{\sigma_\mathsf{p}} P \mid \mathsf{q} \triangleright_{\sigma_\mathsf{q}} Q_j} \lfloor \text{P}|\text{Sel} \rceil$$

$$\frac{i = 1 \text{ if } e \downarrow_{\sigma_\mathsf{p}} \mathsf{true} \qquad i = 2 \text{ o.w.}}{\mathsf{p} \triangleright_{\sigma_\mathsf{p}} \mathsf{if}\, e\, \mathsf{then}\, P_1\, \mathsf{else}\, P_2 \to \mathsf{p} \triangleright_{\sigma_\mathsf{p}} P_i} \lfloor \text{P}|\text{Cond} \rceil \qquad \frac{N \preceq M \qquad M \to M' \qquad M' \preceq N'}{N \to N'} \lfloor \text{P}|\text{Struct} \rceil$$

$$\frac{P \mid N \to P' \mid N'}{\mathsf{p} \triangleright_v \mathsf{def}\, X = Q\, \mathsf{in}\, P \mid N \to \mathsf{p} \triangleright_v \mathsf{def}\, X = Q\, \mathsf{in}\, P' \mid N'} \lfloor \text{P}|\text{Ctx} \rceil \qquad \frac{N \to N'}{N \mid M \to N' \mid M} \lfloor \text{P}|\text{Par} \rceil$$

Figure 5: Stateful Processes, synchronous semantics.

$$\frac{}{\mathsf{def}\, X = B_2\, \mathsf{in}\, B_1[X] \preceq \mathsf{def}\, X = B_2\, \mathsf{in}\, B_1[B_2]} \lfloor \text{S}|\text{Unfold} \rceil$$

$$\frac{}{\mathsf{p} \triangleright_v \mathbf{0} \preceq \mathbf{0}} \lfloor \text{S}|\text{PZero} \rceil \qquad \frac{}{N \mid \mathbf{0} \preceq N} \lfloor \text{S}|\text{NZero} \rceil \qquad \frac{}{\mathsf{def}\, X = B\, \mathsf{in}\, \mathbf{0} \preceq \mathbf{0}} \lfloor \text{S}|\text{ProcEnd} \rceil$$

Figure 6: Stateful Processes, structural precongruence.

This calculus once again includes a structural precongruence relation $\preceq$, defined in Figure 6, which allows unfolding of recursive definition and garbage collection (removal of terminated processes).

Networks in SP do not enjoy a counterpart to Theorem 1, as send/receive actions may be unmatched or wrongly ordered. To avoid this happening, we generate networks automatically from choreographies in a faithful way, by EndPoint Projection (EPP).

EPP is defined first at the process level, by means of a partial function $[\![C]\!]_\mathsf{p}$. The rules defining behaviour projection are given in Figure 7. Each choreography term is projected to the local action of the process that we are projecting. For example, a communication term $\mathsf{p}.e$ `->` q is projected into a send action for the sender process p, a receive action for the receiver process q, or nothing otherwise. The rule for projecting a conditional uses the standard (and partial) merging operator $\sqcup$: $B \sqcup B'$ is isomorphic to B and B' up to branching, where the branches of B or B' with distinct labels are also included (see [9] for details). The definition of merge is given in Figure 8. This aspect is found repeatedly in most choreography models [2, 4, 5, 15].

Definition 1 (EPP from CC to SP)**.** *Given a choreography C and a state σ, the endpoint projection $[\![C, \sigma]\!]$*

$$[\![\mathsf{p}.e \texttt{ -> } \mathsf{q}; C]\!]_\mathsf{r} = \begin{cases} \mathsf{q}!e; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}?; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases} \qquad [\![\mathsf{p} \texttt{ -> } \mathsf{q}[\ell]; C]\!]_\mathsf{r} = \begin{cases} \mathsf{q} \oplus \ell; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}\&\{\ell : [\![C]\!]_\mathsf{r}\} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{if}\, \mathsf{p}.e\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2; C]\!]_\mathsf{r} = \begin{cases} \mathsf{if}\, e\, \mathsf{then}\, [\![C_1]\!]_\mathsf{r}\, \mathsf{else}\, [\![C_2]\!]_\mathsf{r}; [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ ([\![C_1]\!]_\mathsf{r} \sqcup [\![C_2]\!]_\mathsf{r}); [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{def}\, X = C_2\, \mathsf{in}\, C_1]\!]_\mathsf{r} = \mathsf{def}\, X = [\![C_2]\!]_\mathsf{r}\, \mathsf{in}\, [\![C_1]\!]_\mathsf{r} \qquad [\![\mathbf{0}]\!]_\mathsf{r} = \mathbf{0} \qquad [\![X]\!]_\mathsf{r} = X$$

Figure 7: Core Choreographies, behaviour projection.

$$
\begin{array}{rcl}
(\mathsf{q}!e;B) \sqcup (\mathsf{q}!e;B') &=& \mathsf{q}!e;(B \sqcup B') \\
(\mathsf{p}?;B) \sqcup (\mathsf{p}?;B') &=& \mathsf{p}?;(B \sqcup B') \\
(\mathsf{q} \oplus l;B) \sqcup (\mathsf{q} \oplus l;B') &=& \mathsf{q} \oplus l;(B \sqcup B') \\
\mathsf{p}\&\{l_i : B_i\}_{i\in J} \sqcup \mathsf{p}\&\{l_i : B'_i\}_{i\in K} &=& \\
\multicolumn{3}{c}{\mathsf{p}\&\big(\{l_i : (B_i \sqcup B'_i)\}_{i\in J\cap K} \cup \{l_i : B_i\}_{i\in J\setminus K} \cup \{l_i : B'_i\}_{i\in K\setminus J}\big)} \\
(\mathsf{if}\, e\, \mathsf{then}\, B_1\, \mathsf{else}\, B_2) \sqcup (\mathsf{if}\, e\, \mathsf{then}\, B'_1\, \mathsf{else}\, B'_2) &=& (\mathsf{if}\, e\, \mathsf{then}\,(B_1 \sqcup B'_1)\,\mathsf{else}\,(B_2 \sqcup B'_2)) \\
X \sqcup X &=& X \\
(\mathsf{def}\, X = B_2\, \mathsf{in}\, B_1) \sqcup (\mathsf{def}\, X = B'_2\, \mathsf{in}\, B'_1) &=& (\mathsf{def}\, X = (B_2 \sqcup B'_2)\,\mathsf{in}\,(B_1 \sqcup B'_1)) \\
B_1 \sqcup B_2 &=& B'_1 \sqcup B'_2 \quad (\text{if } B_1 \preceq B'_1 \text{ and } B_2 \preceq B'_2)
\end{array}
$$

Figure 8: Core Choreographies, merge operator in behaviour projection.

is the parallel composition of the EPPs of C wrt all processes in $\mathsf{pn}(C)$*:*

$$[\![C,\sigma]\!] = \prod_{\mathsf{p}\in\mathsf{pn}(C)} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} [\![C]\!]_{\mathsf{p}}\,.$$

When $[\![C,\sigma]\!] = N$ is defined for any σ, we say that C is *projectable* and that N is the projection of C,σ.

Example 2. *The projection of the choreography in Example 1 is the parallel composition of its projections relative to each of the three intervening processes.*

$$
\begin{array}{l}
\mathsf{a} \triangleright_{\sigma(\mathsf{a})} \mathsf{s}!\mathit{title};\ \mathsf{s}?;\ \mathsf{b}\&\{\mathit{ok} : \mathsf{s}?,\ \mathit{ko} : \mathbf{0}\} \\
\mathsf{s} \triangleright_{\sigma(\mathsf{s})} \mathsf{a}?;\ \mathsf{a}!\mathit{price};\ \mathsf{b}!\mathit{price};\ \mathsf{b}\&\{\mathit{ok} : \mathsf{a}!\mathit{book},\ \mathit{ko} : \mathbf{0}\} \\
\mathsf{b} \triangleright_{\sigma(\mathsf{b})} \mathsf{s}?;\ \mathsf{if}\ \mathit{happy}\ \mathsf{then}\ (\mathsf{s} \oplus \mathit{ok};\ \mathsf{a} \oplus \mathit{ok})\ \mathsf{else}\ (\mathsf{s} \oplus \mathit{ko};\ \mathsf{a} \oplus \mathit{ko})
\end{array}
$$

When these processes execute concurrently, they precisely follow the specification from the original choreography.

Theorem 2 (EPP Theorem [9])**.** *Let σ be a choreography and σ be a state. If $[\![C,\sigma]\!]$ is defined, then:*

- *(Completeness) if $C,\sigma \to C',\sigma'$ for some choreography C' and state σ', then $[\![C,\sigma]\!] \to\succ [\![C',\sigma']\!]$;*
- *(Soundness) if $[\![C,\sigma]\!] \to N$ for some process network N, then there exist a choreography C' and a state σ' such that $C,\sigma \to C',\sigma'$ and $[\![C',\sigma']\!] \prec N$.*

The *pruning relation* $\prec$ (see [4, 5]) eliminates branches introduced by the merging operator $\sqcup$ when they are not needed anymore to follow the originating choreography. Pruning does not alter reductions, since the eliminated branches are never selected, cf. [4, 15, 23]. Combining Theorem 2 with Theorem 1 we get that the projections of choreographies never deadlock.

Corollary 1 (Deadlock-freedom by construction [9])**.** *Let N be a process network such that $N = [\![C,\sigma]\!]$ for some choreography C and state σ. Then either $N \preceq \mathbf{0}$ (termination) or there exists a network N' such that $N \to N'$.*

Our final definition in this section, also from [9], provides us with a way to compare the behaviour of two choreographies with respect to a given set of processes.

Definition 2 (Computational equivalence)**.** *Two states σ_1,σ_2 are* equivalent *wrt a set of process names $\tilde{\mathsf{p}}$, written $\sigma_1 \equiv_{\tilde{\mathsf{p}}} \sigma_2$, if $\sigma_1(\mathsf{p}) = \sigma_2(\mathsf{p})$ for every $\mathsf{p} \in \tilde{\mathsf{p}}$.*

Two choreographies C_1 and C_2 are equivalent *wrt a set of process names $\tilde{\mathsf{p}}$ if: whenever $\sigma_1 \equiv_{\tilde{\mathsf{p}}} \sigma_2$, if $C_1,\sigma_1 \to^* \mathbf{0},\sigma'_1$ then $C_2,\sigma_2 \to^* \mathbf{0},\sigma'_2$ with $\sigma'_1 \equiv_{\tilde{\mathsf{p}}} \sigma'_2$, and conversely.*

$$\mathscr{C}[\bullet] ::= \bullet; C \mid \eta; \mathscr{C}[\bullet] \mid \mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ \mathscr{C}_1[\bullet]\ \mathsf{else}\ \mathscr{C}_2[\bullet] \mid \mathsf{def}\ X = C\ \mathsf{in}\ \mathscr{C}[\bullet] \mid X \mid \mathbf{0}$$

Figure 9: Contexts for asynchrony.

3. Asynchrony in Choreographies

In order to define an asynchronous semantics for CC and SP and prove their equivalence to the synchronous semantics, we need to define precisely what we understand by "asynchronous semantics" and by "equivalence". Intuitively, communications from p to q are asynchronous if: they occur in two steps (send and receive), the send step does not require q to be ready, and if the send step is executed, then the corresponding receive step is also eventually executed.

We begin by defining asynchrony in CC. We extend the syntax with contexts, which can contain holes denoted as $\bullet$, formally defined in Figure 9. Structural precongruence is defined for contexts as for choreographies. We do not consider holes to be interactions, so they cannot be swapped with any other action.

Using contexts, we can define a function that identifies the type of the next action in context $\mathscr{C}[\bullet]$ involving a process r: a communication (comm), a conditional (cond), or dependent on how $\bullet$ is instantiated ($\bullet$). This function is partial – in particular, it is undefined if $\mathscr{C}[\bullet]$ does not contain either r or $\bullet$.

Definition 3. *The* next *action for process* r *in a context* $\mathscr{C}[\bullet]$*, denoted* $\mathsf{next}_\mathsf{r}(\mathscr{C}[\bullet])$*, is defined as follows.*

$$\mathsf{next}_\mathsf{r}(\bullet; C) = \bullet \qquad \mathsf{next}_\mathsf{r}(\mathbf{0}) = \mathsf{next}_\mathsf{r}(X) = \textit{undefined}$$

$$\mathsf{next}_\mathsf{r}(\eta; \mathscr{C}[\bullet]) = \begin{cases} \mathsf{comm} & \textit{if } \mathsf{r} \in \mathsf{pn}(\eta) \\ \mathsf{next}_\mathsf{r}(\mathscr{C}[\bullet]) & \textit{otherwise} \end{cases}$$

$$\mathsf{next}_\mathsf{r}(\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ \mathscr{C}_1[\bullet]\ \mathsf{else}\ \mathscr{C}_2[\bullet]) = \begin{cases} \mathsf{cond} & \textit{if } \mathsf{p} = \mathsf{r} \\ \mathsf{next}_\mathsf{r}(\mathscr{C}_1[\bullet]) & \textit{if } \mathsf{next}_\mathsf{r}(\mathscr{C}_1[\bullet]) = \mathsf{next}_\mathsf{r}(\mathscr{C}_2[\bullet]) \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

$$\mathsf{next}_\mathsf{r}(\mathsf{def}\ X = C\ \mathsf{in}\ \mathscr{C}[\bullet]) = \mathsf{next}_\mathsf{r}((\mathscr{C}[\bullet])[C/X])$$

The proviso in the second case of the definition of $\mathsf{next}_\mathsf{r}(\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ \mathscr{C}_1[\bullet]\ \mathsf{else}\ \mathscr{C}_2[\bullet])$ ensures that the enabled action for r is uniquely defined. Note that recursive definitions are only unfolded once.

This notion is well-defined, since structural precongruence cannot swap actions involving the same process. We denote by $\mathscr{C}[\eta]$ the result of replacing every occurrence of $\bullet$ in $\mathscr{C}[\bullet]$ by η. If η is an interaction and $\mathscr{C}[\eta]$ is projectable, then $\mathsf{next}_\mathsf{r}(\mathscr{C}[\eta])$ describes the top action in the EPP of $[\![\mathscr{C}[\eta]]\!]_\mathsf{r}$.

Since we expect asynchronous communications to occur in two steps, it is reasonable to expect CC to be extended with additional actions. By an *extension* of CC, we understand a choreography language that has all the primitives of CC, an arbitrarily larger set of statements η, and an adequately extended function pn. Our definitions of contexts and next extend automatically to these languages.

We can now formalise the intuition given at the beginning of this section.

Definition 4. *A semantics* $\to_a$ *for an extension of CC is* asynchronous *if the following conditions hold for any state* σ *and context* $\mathscr{C}[\bullet]$*, where we assume that* $\mathbf{0}; C$ *stands for* C.

- *If* $\mathsf{next}_\mathsf{p}(\mathscr{C}[\bullet]) = \bullet$*, then* $\mathscr{C}[\mathsf{p}.e \texttt{ -> } \mathsf{q}], \sigma \to_a \mathscr{C}[*^v_\mathsf{q}], \sigma$*, where* $e \downarrow^\mathsf{p}_\sigma v$ *and* $*^v_\mathsf{q}$ *is a statement that depends on* v *and* q.
- *If* $\mathsf{next}_\mathsf{p}(\mathscr{C}[\bullet]) = \bullet$*, then* $\mathscr{C}[\mathsf{p} \texttt{ -> } \mathsf{q}[\ell]], \sigma \to_a \mathscr{C}[*^\ell_\mathsf{q}], \sigma$*, with* $*^\ell_\mathsf{q}$ *as above.*
- *If* $\mathsf{next}_\mathsf{q}(\mathscr{C}[\bullet]) = \bullet$*, then* $\mathscr{C}[*^v_\mathsf{q}], \sigma \to_a \mathscr{C}[\mathbf{0}], \mathsf{upd}(\sigma, \mathsf{q}, v)$.
- *If* $\mathsf{next}_\mathsf{q}(\mathscr{C}[\bullet]) = \bullet$*, then* $\mathscr{C}[*^\ell_\mathsf{q}], \sigma \to_a \mathscr{C}[\mathbf{0}], \sigma$.

$$\eta ::= \ldots \mid \mathsf{p}.e \overset{x}{\texttt{->}} \bullet_{\mathsf{q}} \mid \bullet_{\mathsf{p}} \overset{\hat{v}}{\texttt{->}} \mathsf{q} \mid \mathsf{p} \overset{x}{\texttt{->}} \bullet_{\mathsf{q}}[\ell] \mid \bullet_{\mathsf{p}} \overset{\hat{\ell}}{\texttt{->}} \mathsf{q}[\ell] \qquad \hat{v} ::= x \mid v \qquad \hat{\ell} ::= x \mid \ell$$

Figure 10: Asynchronous Core Choreographies, runtime terms.

The second ingredient we need is a formal definition of equivalence. Our notion is similar to the standard notion of operational correspondence from [19], but stronger, since we require that any additional term introduced by asynchronous reductions can always be consumed.

Definition 5. *An asynchronous semantics $\to_a$ for an extension of CC is asynchronously equivalent to the semantics $\to$ of CC if, for all choreographies C and C' and states σ and σ':*

- *if $C, \sigma \to C', \sigma'$, then $C, \sigma \to_a^* C', \sigma'$;*
- *if $C, \sigma \to_a^* C', \sigma'$, then there exist a choreography C'' and a state σ'' such that $C, \sigma \to^* C'', \sigma''$ and $C', \sigma' \to_a^* C'', \sigma''$.*

Note that, in the last point of the above definition, the choreography C' might include terms from the extended choreography language.

4. Asynchrony by Design

Now we are ready to present our extension of CC and define its asynchronous semantics.

4.1. Asynchronous Core Choreographies

We extend the syntax of choreographies with the runtime terms in Figure 10. We call the resulting calculus aCC (for asynchronous CC). The key idea is that, at runtime, a communication is expanded into multiple actions. For example, a communication $\mathsf{p}.e \texttt{ -> } \mathsf{q}$ expands in $\mathsf{p}.e \overset{x}{\texttt{->}} \bullet_{\mathsf{q}}$ – a send action from p – and $\bullet_{\mathsf{p}} \overset{x}{\texttt{->}} \mathsf{q}$ – a receive action by q. The process subscripts at $\bullet$ are immaterial for the semantics, but are used later to define EPP. The tag x is used to specify that the original intention by the programmer was for that message from p to reach that receive action at q. Thus, executing $\mathsf{p}.e \overset{x}{\texttt{->}} \bullet_{\mathsf{q}}$ replaces x in the corresponding receive action with the actual value v computed from e at p, yielding $\bullet_{\mathsf{p}} \overset{v}{\texttt{->}} \mathsf{q}$. Finally, executing $\bullet_{\mathsf{p}} \overset{v}{\texttt{->}} \mathsf{q}$ updates the state of q. The tags in the actions for label selections are not essential for propagating values as above, but they make the semantics uniform. Process names for the new runtime terms are defined as follows, ignoring process names in tags and in $\bullet$ subscripts.

$$\mathsf{pn}(\mathsf{p}.e \overset{x}{\texttt{->}} \bullet_{\mathsf{q}}) = \mathsf{pn}(\mathsf{p} \overset{x}{\texttt{->}} \bullet_{\mathsf{q}}[\ell]) = \{\mathsf{p}\} \qquad \mathsf{pn}(\bullet_{\mathsf{p}} \overset{\hat{v}}{\texttt{->}} \mathsf{q}) = \mathsf{pn}(\bullet_{\mathsf{p}} \overset{\hat{\ell}}{\texttt{->}} \mathsf{q}[\ell]) = \{\mathsf{q}\}$$

The semantics for aCC includes:

- the rules from Figure 11, replacing $\lfloor$C|Com$\rceil$ and $\lfloor$C|Sel$\rceil$, and from Figure 12, extending $\preceq$;
- rules $\lfloor$C|Cond$\rceil$ and $\lfloor$C|Struct$\rceil$ from Figure 2;
- the rules defining $\preceq$ (Figure 2), with η now ranging also over the new runtime terms.

In line with the Barendregt convention, we require the tags introduced by unfolding a value communication or selection to be globally fresh. This allows us to use these tags to maintain the correspondence between the value being sent and that being received.

The key to the asynchronous semantics of aCC lies in the new swaps allowed by $\preceq$, due to the definition of pn for the new terms.

$$\frac{e \downarrow^{\mathsf{p}}_{\sigma} v}{\mathsf{p}.e \overset{x}{\texttt{->}} \bullet_{\mathsf{q}}; C, \sigma \to_a C[v/x], \sigma}\ \lfloor\text{C}|\text{Com-S}\rceil \qquad \frac{}{\bullet_{\mathsf{p}} \overset{v}{\texttt{->}} \mathsf{q}; C, \sigma \to_a C, \mathsf{upd}(\sigma, \mathsf{q}, v)}\ \lfloor\text{C}|\text{Com-R}\rceil$$

$$\frac{}{\mathsf{p} \overset{x}{\texttt{->}} \bullet_{\mathsf{q}}[\ell]; C, \sigma \to_a G, C[\ell/x], \sigma}\ \lfloor\text{C}|\text{Sel-S}\rceil \qquad \frac{}{\bullet_{\mathsf{p}} \overset{\ell}{\texttt{->}} \mathsf{q}[\ell]; C, \sigma \to_a C, \sigma}\ \lfloor\text{C}|\text{Sel-R}\rceil$$

Figure 11: Asynchronous Core Choreographies, semantics (runtime terms).

$$\frac{}{\mathsf{p}.e \texttt{ -> } \mathsf{q} \preceq \mathsf{p}.e \overset{x}{\texttt{->}} \bullet_{\mathsf{q}}; \bullet_{\mathsf{p}} \overset{x}{\texttt{->}} \mathsf{q}}\ \lfloor\text{C}|\text{Com-Unfold}\rceil \qquad \frac{}{\mathsf{p} \texttt{ -> } \mathsf{q}[\ell] \preceq \mathsf{p} \overset{x}{\texttt{->}} \bullet_{\mathsf{q}}[\ell]; \bullet_{\mathsf{p}} \overset{x}{\texttt{->}} \mathsf{q}[\ell]}\ \lfloor\text{C}|\text{Sel-Unfold}\rceil$$

Figure 12: Asynchronous Core Choreographies, new rule for structural precongruence.

Example 3. *Consider again the choreography from Example 1. We focus on the first three communications. To execute them asynchronously, we must first expand them:*

$$\begin{aligned}&\mathsf{a}.\mathit{title} \texttt{ -> } \mathsf{s};\ \mathsf{s}.\mathit{price} \texttt{ -> } \mathsf{a};\ \mathsf{s}.\mathit{price} \texttt{ -> } \mathsf{b}\\ &\qquad\preceq \mathsf{a}.\mathit{title} \overset{x}{\texttt{->}} \bullet_{\mathsf{s}};\ \bullet_{\mathsf{a}} \overset{x}{\texttt{->}} \mathsf{s};\ \mathsf{s}.\mathit{price} \overset{y}{\texttt{->}} \bullet_{\mathsf{a}};\ \bullet_{\mathsf{s}} \overset{y}{\texttt{->}} \mathsf{a};\ \mathsf{s}.\mathit{price} \overset{z}{\texttt{->}} \bullet_{\mathsf{b}};\ \bullet_{\mathsf{s}} \overset{z}{\texttt{->}} \mathsf{b}\end{aligned}$$

The first two communications must be executed in order, since the first action blocks the second one and the second action blocks all subsequent sends. However, the last four actions can be interchanged using $\preceq$:

$$\mathsf{s}.\mathit{price} \overset{y}{\texttt{->}} \bullet_{\mathsf{a}}; \bullet_{\mathsf{s}} \overset{y}{\texttt{->}} \mathsf{a}; \mathsf{s}.\mathit{price} \overset{z}{\texttt{->}} \bullet_{\mathsf{b}}; \bullet_{\mathsf{s}} \overset{z}{\texttt{->}} \mathsf{b} \preceq \mathsf{s}.\mathit{price} \overset{y}{\texttt{->}} \bullet_{\mathsf{a}}; \mathsf{s}.\mathit{price} \overset{z}{\texttt{->}} \bullet_{\mathsf{b}}; \bullet_{\mathsf{s}} \overset{z}{\texttt{->}} \mathsf{b}; \bullet_{\mathsf{s}} \overset{y}{\texttt{->}} \mathsf{a}$$

So s *can send both its messages at once, while* b *can receive its message before* a*.*

The semantics of aCC also enjoys deadlock-freedom.

Theorem 3 (Deadlock-freedom)**.** *Given a choreography C, either $C \preceq \mathbf{0}$ (termination) or, for every σ, there exist a choreography C' and a state σ' such that $C, \sigma \to_a C', \sigma'$.*

Proof. Straightforward by structural induction on C. □

Theorem 4. *The relation $\to_a$ is an asynchronous semantics for aCC.*

Proof. For the first case, we are only interested in a situation when $\mathsf{next}_{\mathsf{p}}(\mathscr{C}[\bullet]) = \bullet$. A straightforward induction on the definition of next, using the rules in Figure 3, shows that in this case $\mathscr{C}[\mathsf{p}.e \texttt{ -> } \mathsf{q}] \equiv \mathscr{C}[\mathsf{p}.e \overset{x}{\texttt{->}} \bullet_{\mathsf{q}}; \bullet_{\mathsf{p}} \overset{x}{\texttt{->}} \mathsf{q}] \preceq \mathsf{p}.e \overset{x}{\texttt{->}} \bullet_{\mathsf{q}}; \mathscr{C}[\bullet_{\mathsf{p}} \overset{x}{\texttt{->}} \mathsf{q}]$, as p does not occur in any action encountered during the computation of $\mathsf{next}_{\mathsf{p}}(\mathscr{C}[\bullet])$. The thesis then follows by rule $\lfloor$C|Com-S$\rceil$. The second case is nearly identical. The remaining two cases are similar, arguing over $\mathsf{next}_{\mathsf{q}}(\mathscr{C}[\bullet])$. □

Theorem 5. *The semantics $\to_a$ is asynchronously equivalent to the semantics of CC.*

Proof. Proving the first property is straightforward. The only non-trivial case is when the transition $C, \sigma \to C', \sigma'$ uses rule $\lfloor$C|Com$\rceil$, and in this case in the asynchronous semantics we can unfold the corresponding communication and reduce twice using $\lfloor$C|Com-S$\rceil$ and $\lfloor$C|Com-R$\rceil$.

For the second property, we make two observations.

(1) The relation $\to_a$ has the diamond property, namely: if $C, \sigma \to_a C', \sigma'$ and $C, \sigma \to_a C'', \sigma''$, then there exist C''' and σ''' such that $C', \sigma' \to_a C''', \sigma'''$ and $C'', \sigma'' \to_a C''', \sigma'''$ (see Figure 13, left). This is directly proven by case analysis on the structure of a choreography C with two distinct possible transitions.

(2) If $C, \sigma \rightarrow_a C', \sigma'$, then there exist C'' and σ'' such that $C, \sigma \rightarrow^* C'', \sigma''$ and $C', \sigma' \rightarrow_a^* C'', \sigma''$. This is similar to the proof for the first property. If the transition uses rule $\lfloor$C$|$Com-S$\rceil$, then the choreography C'' is obtained by firing rule $\lfloor$C$|$Com-R$\rceil$, but it might be necessary to execute additional actions in case the receiver is not ready for that transition yet. This is always possible due to Theorem 3.

The property now follows by induction on the length of the reduction chain $C, \sigma \rightarrow_a^* C', \sigma'$. When this length is 0, the result is trivial, and when the length is 1, the result is simply (2) above. Otherwise, we follow the reasoning displayed in Figure 13, right. By assumption, we have that $C, \sigma \rightarrow_a^* C_1, \sigma_1 \rightarrow_a C', \sigma'$ (top of the picture). By induction hypothesis, there exist C_1' and σ_1' such that $C, \sigma \rightarrow^* C_1', \sigma_1'$ and $C_1, \sigma_1 \rightarrow_a^* C_1', \sigma_1'$ (left triangle). By iterated application of (1) above, there exist C_1'' and σ_1'' such that $C_1', \sigma_1' \rightarrow_a^? C_1'', \sigma_1''$ and $C', \sigma' \rightarrow_a^* C_1'', \sigma_1''$ (right square). The reduction $C_1', \sigma_1' \rightarrow_a^? C_1'', \sigma_1''$ is optional because it may be the case that the action in the reduction $C_1, \sigma_1 \rightarrow_a C', \sigma'$ is already included in the reduction chain $C_1, \sigma_1 \rightarrow_a^* C_1', \sigma_1'$. In this case, the thesis follows, taking $C'' = C_1'$ and $\sigma'' = \sigma_1'$. Otherwise, we apply (2) above (lower right triangle) to obtain C'' and σ'' such that $C_1', \sigma_1' \rightarrow^* C'', \sigma''$ and $C_1'', \sigma_1'' \rightarrow_a^* C'', \sigma''$, and the thesis follows. $\square$

$$\begin{tikzcd}
C, \sigma \arrow[r, "a"'] \arrow[d, "a"'] & C', \sigma' \arrow[d, dashed, "a"'] \\
C'', \sigma'' \arrow[r, dashed, "a"'] & C''', \sigma'''
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
C, \sigma \arrow[r, "*"] \arrow[r, phantom, "a"'] \arrow[dr, dashed, "*"'] & C_1, \sigma_1 \arrow[r, "a"'] \arrow[d, dashed, "a"', "*"] & C', \sigma' \arrow[d, dashed, "a"', "*"] \\
& C_1', \sigma_1' \arrow[r, dashed, "?"] \arrow[r, phantom, "a"'] \arrow[dr, dashed, "*"'] & C_1'', \sigma_1'' \arrow[d, dashed, "a"', "*"] \\
& & C'', \sigma''
\end{tikzcd}$$

Figure 13: The diamond property for $\rightarrow_a$ (left) and the reasoning in the proof of the second property in Theorem 5 (right).

Remark 1. *Reasoning on asynchronous behaviour is known to be hard for programmers. That is why our terms for modelling asynchronous communications are* runtime terms*: they are not intended to be part of the source language used by developers to program. In general, programmers do not need to be aware of the development in this section, thanks to Theorem 5. Instead, they can just write a choreography thinking in terms of synchronous communications as usual (syntax and semantics), and then just assume that adopting an asynchronous communications will not lead to any bad behaviour. In the next subsection, we show that this abstraction carries over to asynchronous process code generated by EPP. Since EPP is automatic, developers do not need to worry about asynchrony in process code either.*

4.2. Asynchronous Stateful Processes

In the development above, we characterized asynchrony at an abstract level, and showed that our choreography model satisfies the properties we identified. In this section, we take a more down-to-earth approach: we define directly an asynchronous variant for SP, extend EPP to the runtime terms introduced earlier, and show that the asynchronous semantics for both calculi again satisfy the EPP Theorem (Theorem 2).

Syntactically, we need to make a slight change to our processes: each process is now also equipped with a queue of incoming messages. We name this version of the calculus aSP. So actors in the asynchronous model have the form $\mathsf{p} \triangleright_v^\rho B$, where ρ is a queue of incoming messages. We sometimes omit ρ when it is empty; in particular, this allows us to view SP networks as special cases of aSP networks. A message is a pair $\langle \mathsf{q}, m \rangle$, where q is the sender process and m is a value, label, or process identifier.

The semantics of aSP consists of rules $\lfloor$P$|$Cond$\rceil$, $\lfloor$P$|$Par$\rceil$, $\lfloor$P$|$Ctx$\rceil$ and $\lfloor$P$|$Struct$\rceil$ from SP (Figure 5) together with the new rules given in Figure 14. Structural precongruence for aSP is defined exactly as for SP. We write $\rho \cdot \langle \mathsf{q}, m \rangle$ to denote the queue obtained by appending message $\langle \mathsf{q}, m \rangle$ to ρ, and $\langle \mathsf{q}, m \rangle \cdot \rho$ for the queue with $\langle \mathsf{q}, m \rangle$ at the head and ρ as tail. We simulate having one separate FIFO queue for each other

$$\frac{e \downarrow^{\mathsf{p}}_{\sigma} v \qquad \rho'_{\mathsf{q}} = \rho_{\mathsf{q}} \cdot \langle \mathsf{p}, v\rangle}{\mathsf{p} \triangleright^{\rho_\mathsf{p}}_{\sigma_\mathsf{p}} \mathsf{q}!e; B_\mathsf{p} \mid \mathsf{q} \triangleright^{\rho_\mathsf{q}}_{\sigma_\mathsf{q}} B_\mathsf{q} \to \mathsf{p} \triangleright^{\rho_\mathsf{p}}_{\sigma_\mathsf{p}} B_\mathsf{p} \mid \mathsf{q} \triangleright^{\rho'_\mathsf{q}}_{\sigma_\mathsf{q}} B_\mathsf{q}} \lfloor \text{P}|\text{Com-S} \rceil \qquad \frac{\rho_\mathsf{q} \preceq \langle \mathsf{p}, v\rangle \cdot \rho'_\mathsf{q}}{\mathsf{q} \triangleright^{\rho_\mathsf{q}}_{\sigma_\mathsf{q}} \mathsf{p}?; B \to \mathsf{q} \triangleright^{\rho'_\mathsf{q}}_{\mathsf{upd}(\sigma_\mathsf{q}, v)} B} \lfloor \text{P}|\text{Com-R} \rceil$$

$$\frac{\rho'_{\mathsf{q}} = \rho_{\mathsf{q}} \cdot \langle \mathsf{p}, \ell\rangle}{\mathsf{p} \triangleright^{\rho_\mathsf{p}}_{\sigma_\mathsf{p}} \mathsf{q} \oplus \ell; B_\mathsf{p} \mid \mathsf{q} \triangleright^{\rho_\mathsf{q}}_{\sigma_\mathsf{q}} B_\mathsf{q} \to \mathsf{p} \triangleright^{\rho_\mathsf{p}}_{\sigma_\mathsf{p}} B_\mathsf{p} \mid \mathsf{q} \triangleright^{\rho'_\mathsf{q}}_{\sigma_\mathsf{q}} B_\mathsf{q}} \lfloor \text{P}|\text{Sel-S} \rceil \qquad \frac{j \in I \qquad \rho_\mathsf{q} \preceq \langle \mathsf{p}, v\rangle \cdot \rho'_\mathsf{q}}{\mathsf{q} \triangleright^{\rho_\mathsf{q}}_{\sigma_\mathsf{q}} \mathsf{p}\&\{\ell_i : B_i\}_{i \in I} \to \mathsf{q} \triangleright^{\rho'_\mathsf{q}}_{\sigma_\mathsf{q}} B_j} \lfloor \text{P}|\text{Sel-R} \rceil$$

Figure 14: Asynchronous Stateful Processes, semantics (new rules).

process by allowing incoming messages from different senders to be exchanged, which we represent using the congruence $\rho \preceq \rho'$ defined by the rule $\langle \mathsf{p}, m\rangle \cdot \langle \mathsf{q}, m'\rangle \preceq \langle \mathsf{q}, m'\rangle \cdot \langle \mathsf{p}, m\rangle$ if $\mathsf{p} \neq \mathsf{q}$.

All behaviours of SP are valid also in aSP.

Theorem 6. *Let N be a process network in SP. If $N \to N'$ for some network N', then $N_{[]} \to^* N'_{[]}$ in aSP, where $N_{[]}$ denotes the asynchronous network obtained by adding an empty queue to each process.*

Proof. Straightforward by case analysis: the only reduction rules in SP that are not present in aSP are those for value communications and label selections, which can be simulated by applying rules $\lfloor$P|Com-S$\rceil$ and $\lfloor$P|Com-R$\rceil$ (respectively, $\lfloor$P|Sel-S$\rceil$ and $\lfloor$P|Sel-R$\rceil$) in sequence. □

The converse is not true, so the relation between SP and aSP is not so strong as that between CC and aCC (stated in Theorem 5). This is because of deadlocks: in SP, a communication action can only take place when the sender and receiver are ready to synchronize; in aSP, a process can send a message to another process, even though the intended recipient is not yet able to receive it. For example, the network $\mathsf{p} \triangleright_{\sigma_\mathsf{p}} \mathsf{q}!1 \mid \mathsf{q} \triangleright_{\sigma_\mathsf{q}} \mathbf{0}$ is deadlocked in SP, but its counterpart in aSP (obtained by adding empty queues at each process) reduces to $\mathsf{q} \triangleright^{\langle \mathsf{p},1\rangle}_{\sigma_\mathsf{q}} \mathbf{0}$. (This network is not equivalent to $\mathbf{0}$, since there is a non-empty queue.)

More interestingly, the network

$$\mathsf{p} \triangleright_{\sigma_\mathsf{p}} \mathsf{q}!e_1; \mathsf{r}? \mid \mathsf{q} \triangleright_{\sigma_\mathsf{q}} \mathsf{r}!e_2; \mathsf{p}? \mid \mathsf{r} \triangleright_{\sigma_\mathsf{r}} \mathsf{p}!e_3; \mathsf{q}? \tag{1}$$

is deadlocked in SP, but reduces to $\mathbf{0}$ in aSP (all queues are eventually emptied). It is possible to extend CC with communication primitives that capture this type of behaviour, as discussed in [7]; we briefly discuss this point in Section 7.

4.3. Asynchronous EndPoint Projection

Defining an EPP from aCC to aSP requires extending the previous definition with clauses for the new runtime terms, which populate the local queues in the projections. Intuitively, when compiling, e.g., $\bullet_\mathsf{p} \overset{\mathsf{q}}{\texttt{->}} v$, we add a message from containing v at the top of q's queue.

There is one problem with this approach: we can write choreographies that use runtime terms in a "wrong" way, for which Theorem 2 no longer holds.

Example 4. *Consider the choreography $C = \mathsf{p}.1 \texttt{ -> } \mathsf{q}; \bullet_\mathsf{p} \overset{\mathsf{q}}{\texttt{->}} 2$. If we naively project it as we described informally, we obtain $\mathsf{p} \triangleright^{[]}_{\sigma(\mathsf{p})} \mathsf{q}!1 \mid \mathsf{q} \triangleright^{\langle \mathsf{p},2\rangle}_{\sigma(\mathsf{q})} \mathsf{p}?; \mathsf{p}?$, where $[]$ is the empty queue, and q will receive 2 before it receives 1.*

To avoid this undesired behaviour, we restrict ourselves to *well-formed* choreographies: those that can arise from executing a choreography that does not contain runtime terms (i.e., a program). Since runtime terms are supposed to be hidden from the programmer anyway, this restriction does not make us lose any generality in practice.

Definition 6 (Well-formedness)**.** *A choreography C in aCC containing runtime terms is* well-formed *if $\eta_1; \dots; \eta_n; C^{CC} \preceq^- C$, where:*

$$[\![\bullet_{\mathsf{p}} \overset{\mathsf{q}}{\texttt{->}} v; C]\!]_{\mathsf{r}} = \begin{cases} \mathsf{p}?; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases} \qquad [\![\bullet_{\mathsf{p}} \overset{\ell}{\texttt{->}} \mathsf{q}[\ell]; C]\!]_{\mathsf{r}} = \begin{cases} \mathsf{p}\&\{\ell : [\![C]\!]_{\mathsf{r}}\} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

Figure 15: Core Choreographies, asynchronous behaviour projection (new rules).

$$(\![\bullet_{\mathsf{p}} \overset{\mathsf{q}}{\texttt{->}} v; C]\!)_{\mathsf{r}} = \begin{cases} \langle \mathsf{p}, v\rangle \cdot (\![C]\!)_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ (\![C]\!)_{\mathsf{r}} & \text{otherwise} \end{cases} \qquad (\![\bullet_{\mathsf{p}} \overset{\ell}{\texttt{->}} \mathsf{q}[\ell]; C]\!)_{\mathsf{r}} = \begin{cases} \langle \mathsf{p}, \ell\rangle \cdot (\![C]\!)_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ (\![C]\!)_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$(\![\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C]\!)_{\mathsf{r}} = (\![C_1]\!)_{\mathsf{r}} \cdot (\![C]\!)_{\mathsf{r}} \qquad (\![\eta; C]\!)_{\mathsf{r}} = (\![C]\!)_{\mathsf{r}}$$

Figure 16: Core Choreographies, projection of messages in transit.

- *$\preceq^-$ is structural precongruence without rule $\lfloor$C|Unfold$\rceil$;*
- *each η_i is an instantiated receive action of the form $\bullet_{\mathsf{p}} \overset{\mathsf{q}}{\texttt{->}} v$ or $\bullet_{\mathsf{p}} \overset{\ell}{\texttt{->}} \mathsf{q}[\ell]$;*
- *C^{CC} is an CC choreography (i.e., a choreography without runtime terms).*

Well-formedness is decidable, since the set of choreographies equivalent up to $\preceq^-$ is decidable. More efficiently, one can check that C is well-formed by swapping all runtime actions to the beginning and folding all paired send/receive terms. Furthermore, choreography execution preserves well-formedness. Note that the problematic choreography from Example 4 is not well-formed. More generally, we can use well-formedness in the remainder of this section to reason about EPP.

Definition 7 (Asynchronous EPP from aCC to aSP). *Let C be a well-formed aCC choreography and σ be a state. Without loss of generality, we assume that C does not contain $\mathsf{p}.e \overset{x}{\texttt{->}} \bullet_{\mathsf{q}}$ actions.*[2] *The EPP of C and σ is defined as*

$$[\![C, \sigma]\!] = \textstyle\prod_{\mathsf{p}\in\mathsf{pn}(C)} \mathsf{p} \triangleright^{(\![C]\!)_{\mathsf{p}}}_{\sigma(\mathsf{p})} [\![C]\!]_{\mathsf{p}}$$

where $[\![C]\!]_{\mathsf{p}}$ is defined as in Figure 7 with the extra rule in Figure 15 and $(\![C]\!)_{\mathsf{p}}$ is defined by the rules in Figure 16.

In the last case of the definition of $(\![C]\!)_{\mathsf{p}}$ (bottom-right), η ranges over all cases that are not covered previously. The rule for the conditional may seem a bit surprising: in the case of projectable choreographies, mergeability and well-formedness together imply that unmatched receive actions at a process must occur in the same order in both branches. We could alternatively define projection only for well-formed choreographies in the "canonical form" implicit in the definition of well-formedness.

With this definition, we can state and prove an asynchronous variant of Theorem 2.

Theorem 7 (Asynchronous EPP Theorem). *If C is a projectable and well-formed CC choreography, then, for all states σ:*

- *(Completeness) if $C, \sigma \to C', \sigma'$ for some choreography C' and state σ', then $[\![C, \sigma]\!] \to\succ [\![C', \sigma']\!]$;*
- *(Soundness) if $[\![C, \sigma]\!] \to N$ for some network N, then there exist a choreography C' and a state σ' such that $C, \sigma \to C', \sigma'$ and $[\![C', \sigma']\!] \prec N$.*

[2]By well-formedness, we can always rewrite C to an equivalent choreography satisfying this condition. Such a choreography is also guaranteed not to contain $\bullet_{\mathsf{p}} \overset{x}{\texttt{->}} \mathsf{q}$ or $\mathsf{p} \overset{x}{\texttt{->}} \mathsf{q}$ actions.

As a consequence, Corollary 1 applies also to the asynchronous case: the processes projected from CC into aSP are deadlock-free, even when they contain runtime terms.

As usual, the hypotheses in Theorem 7 are not necessary, and this result also holds for some choreographies that are not well-formed. For example, the network in (1) is the projection of

$$\mathsf{p}.e_1 \overset{x}{\texttt{->}} \bullet_{\mathsf{q}}; \mathsf{q}.e_2 \overset{y}{\texttt{->}} \bullet_{\mathsf{r}}; \mathsf{r}.e_3 \overset{z}{\texttt{->}} \bullet_{\mathsf{p}}; \bullet_{\mathsf{p}} \overset{\mathsf{r}}{\texttt{->}} x; \bullet_{\mathsf{q}} \overset{\mathsf{p}}{\texttt{->}} y; \bullet_{\mathsf{r}} \overset{\mathsf{q}}{\texttt{->}} z;$$

with an adequately defined state σ, but this choreography is not structurally precongruent to any choreography obtained from execution of an CC choreography.

5. Asynchrony by Encoding

We now show an alternative implementation of asynchrony, by encoding communications in a synchronous semantics in another extension of CC. We take inspiration from how asynchrony is modelled in foundational process models, specifically the π-calculus [25]. The key idea there is to use processes to represent messages in transit, allowing the sender to proceed immediately after having sent a message without having to synchronise with the receiver [30]. In an asynchronous system, there is no bound to the number of messages that could be transiting in the network; this means that CC is not powerful enough for our purposes, because it can only capture a finite number of processes. For this reason, we extend CC with two standard notions, borrowed from process calculi and previous choreography models: process spawning – the ability to create new processes at runtime – and name mobility – the ability to send process references, or names. We call this extended language Dynamic Core Choreographies (DCC).

For simplicity, we first present our development for the computational fragments of CC and DCC, called Minimal Choreographies (MC) and Dynamic Minimal Choreographies (DMC), respectively. The relationships between the different choreography calculi discussed in this section are summarized in Figure 17. Strict inclusions between calculi are denoted as $\hookrightarrow$; in particular, MC is a strict sublanguage of DMC and CC is a strict sublanguage of DCC.

The dotted arrow (1) in Figure 17 is the cornerstone of the development in this section: every choreography in MC can be encoded in an asynchronous implementation in DMC, by using auxiliary processes to represent messages in transit. Since DMC extends MC with new primitives, it makes sense to extend this encoding to the whole language of DMC (2). Due to the syntactic minimality of MC, our results imply that asynchronous communications can be modelled in choreographies using well-known notions, i.e., process spawning and name mobility (studied, e.g., in [5, 11]), without the need for *ad-hoc* constructions.

The fact that our encoding can be extended from MC to DMC is evidence that our approach is robust, and the simplicity of DMC makes it a convenient foundational calculus to use in future developments of choreographies. However, one of the expected advantages of using a foundational theory such as DMC for capturing asynchrony is indeed that we can reuse existing formal techniques based on standard primitives for choreographies. (This is a common scenario in π-calculus, where many techniques apply to its sublanguages [30].) An example of such reuses is the extension of our results to CC. An important property of CC, shown in [9], is that selections can be encoded as value communications preserving projectability, yielding a map from CC to MC – the dashed arrow (a) in Figure 17. What happens if we consider DCC, which extends DMC with label selections? Ideally, this calculus should *both* have an asynchronous interpretation through the techniques introduced in this section, and still possess the property that selections are encodable in the simpler language DMC. This is indeed the case. We extend our encoding to yield an interpretation of asynchronous selections, yielding arrows (3) and (4) in Figure 17. Encodability of selections in DMC follows immediately from the fact that CC is a sublanguage of DCC, giving us (b) for free.

5.1. Dynamic Core Choreographies

We begin by describing the mapping (1) in Figure 17. This mapping $\{\!\{\cdot\}\!\}$: MC $\to$ DMC transforms every communication action $\mathsf{p}.e \texttt{ -> } \mathsf{q} \in C \in$ MC into a send/receive pair in $\{\!\{C\}\!\} \in$ DMC, with the properties that: p can continue executing without waiting for q to receive its message (and even send further messages to

$$\begin{tikzcd}
\mathrm{MC} \arrow[r, hook] \arrow[d, dotted, bend right, "1"'] \arrow[d, bend left] & \mathrm{CC} \arrow[l, dashed, bend right, "a"'] \arrow[d, dotted, bend left, "4"] \arrow[d, bend right] \\
\mathrm{DMC} \arrow[r, hook] \arrow[loop below, dotted, "2"] & \mathrm{DCC} \arrow[l, dashed, bend left, "b"] \arrow[loop below, dotted, "3"]
\end{tikzcd}$$

Figure 17: Choreography calculi and encodings.

$$C ::= \cdots \mid \mathsf{def}\, X(\tilde{\mathsf{p}}) = C_2 \,\mathsf{in}\, C_1 \mid X\langle\tilde{\mathsf{p}}\rangle$$
$$\eta ::= \cdots \mid \mathsf{p\,start\,q}^T \qquad e ::= \mathsf{p} \mid \cdots$$

Figure 18: Dynamic Core Choreographies, syntax (new terms).

q); and messages from p to q are delivered in the same order as they were originally sent. These properties essentially reflect the requirements concerning value communications in Definition 4.

In MC, there is a bound on the number of values that can be stored at any given time by the system: since each process can hold a single value, the maximum number of values the system can know is equal to the number of processes in the choreography, which is fixed. However, in an asynchronous setting, the number of values that need to be stored is unbounded: a process p may loop forever sending values to q, and q may wait an arbitrary long time before receiving any of them. Therefore, we need to extend MC with the capability to generate new processes. As discussed in [11], this requires enriching the language with two additional abilities: parameters to recursive procedures (in order to be able to use a potentially unbounded number of processes at the same time) and a new action to communicate process names.

Since the extensions from MC to DMC and from CC to DCC are similar, and we later extend our results to CC/DCC (mapping (4) in Figure 17, we first describe the full calculus DCC, which extends CC with process spawning and name mobility.

Formally, the differences between the syntaxes of CC and DCC are highlighted in Figure 18: procedure definitions and calls now have parameters; there is a new term for generating processes; and the expressions sent by processes can also be process names. The possibility of communicating a process name (p.q -> r) ensures name mobility. We use the abbreviation p : r <-> q as shorthand for p.q -> r; p.r -> q.

The semantics for DCC includes an additional ingredient, borrowed from [11]: a graph of connections G, keeping track of which pairs of processes are allowed to communicate. This graph is directed, and an edge from p to q in G (written $\mathsf{p} \overset{G}{\to} \mathsf{q}$) means that p knows the name of q. In order for an actual message to flow between p and q, both processes need to know each other, which we write as $\mathsf{p} \overset{G}{\longleftrightarrow} \mathsf{q}$.[3] The reduction relation now has the form $G, C, \sigma \to G', C', \sigma'$, where G and G' are the connection graphs before and after executing C, respectively. The complete rules are given in Figure 19, with $\preceq$ defined as in CC.[4] In rule $\lfloor$C|Start$\rceil$, the fresh process q is assigned a default value $\perp$.

The proof of Theorem 1 can be generalised to DCC, but this requires an extra ingredient: a simple type system (which we do not detail, as it is a subsystem of that presented in [11]). This type system checks that all processes that attempt at communicating are connected in the communication graph, e.g., by being properly introduced using name mobility. Furthermore, we can define a target process calculus for

[3]In some process calculi, the weaker condition $\mathsf{p} \overset{G}{\to} \mathsf{q}$ is typically sufficient for p to send a message to q. Our condition is equivalent to that found in the standard model of Multiparty Session Types [22]. This choice is orthogonal to our development.

[4]The function $\mathsf{pn}(\eta)$ is extended to communications of process names by the clause $\mathsf{pn}(\mathsf{p.r \text{ -> } q}) = \{\mathsf{p}, \mathsf{q}, \mathsf{r}\}$.

$$\frac{\mathsf{p} \xleftrightarrow{G} \mathsf{q} \qquad \downarrow^{v}_{e[\sigma(\mathsf{p})/\mathsf{c}]}}{G, \mathsf{p}.e \;\texttt{->}\; \mathsf{q}; C, \sigma \;\to\; G, C, \mathsf{upd}(\sigma, \mathsf{q}, v)} \;\lfloor\mathrm{C|Com}\rceil \qquad \frac{\mathsf{p} \xleftrightarrow{G} \mathsf{q}}{G, \mathsf{p} \;\texttt{->}\; \mathsf{q}[\ell]; C, \sigma \;\to\; G, C, \sigma} \;\lfloor\mathrm{C|Sel}\rceil$$

$$\frac{}{G, \mathsf{p}\,\mathsf{start}\,\mathsf{q}^{T}; C, \sigma \;\to\; \mathsf{upd}(\mathsf{p}, \mathsf{q}), C, \mathsf{upd}(\sigma, \mathsf{q}, \bot)} \;\lfloor\mathrm{C|Start}\rceil$$

$$\frac{\mathsf{p} \xleftrightarrow{G} \mathsf{q} \qquad \mathsf{p} \overset{G}{\to} \mathsf{r}}{G, \mathsf{p}.\mathsf{r} \;\texttt{->}\; \mathsf{q}; C, \sigma \;\to\; G \cup \{\mathsf{q} \to \mathsf{r}\}, C, \sigma} \;\lfloor\mathrm{C|Intro}\rceil \qquad \frac{i = 1 \text{ if } \sigma(\mathsf{p}) = \sigma(\mathsf{q}), \qquad i = 2 \text{ otherwise}}{G, \mathsf{if}\; \mathsf{p}.e\; \mathsf{then}\; C_1\; \mathsf{else}\; C_2, \sigma \;\to\; G, C_i, \sigma} \;\lfloor\mathrm{C|Cond}\rceil$$

$$\frac{C_1 \preceq C_2 \qquad G, C_2, \sigma \;\to\; G', C_2', \sigma' \qquad C_2' \preceq C_1'}{G, C_1, \sigma \;\to\; G', C_1', \sigma'} \;\lfloor\mathrm{C|Struct}\rceil$$

Figure 19: Dynamic Core Choreographies, semantics.

DCC and an EndPoint Projection that will automatically synthetise correct-by-construction deadlock-free implementations of (projectable) choreographies, using techniques from [11]. Although these constructions are not technically challenging, we omit them at this point, since they are immaterial for our results. In Section 5.4 we briefly explain how the EndPoint Projection in [11] applies to DCC, and how the results related to it can be combined with our development.

The fragment of DCC that does not contain label selections is called Dynamic Minimal Choreographies (DMC). Amendment and label selection elimination, described in [9], hold for DMC and DCC just as for MC and CC, so that, for any DMC choreography C, we have the following results.

Lemma 1 (Amendment Lemma)**.** *There is a function* $\mathsf{Amend} : CC \to CC$ *such that, for every choreography* C*:*

Completeness: $\mathsf{Amend}(C)$ *is defined;*

Projectability: *for every state* σ*,* $[\![\mathsf{Amend}(C), \sigma]\!]$ *is defined;*

Correspondence: *for every other choreography* C' *and states* σ *and* σ'*,* $C, \sigma \to^* C', \sigma'$ *iff* $\mathsf{Amend}(C), \sigma \to^* \mathsf{Amend}(C'), \sigma'$.

Eliminating selections can be done for any CC choreography; however, the interesting application (which also suffices for our development) is when it is applied to choreographies obtained by amendment.

Theorem 8 (Selection elimination)**.** *There is a function* $(\![\cdot]\!) : CC \to MC$ *such that, for every choreography* $C \in MC$*,* $(\![\mathsf{Amend}(C)]\!)$ *is projectable.*

The function $(\![\cdot]\!)$ essentially replaces all label selections by a pair value communication/conditional. Intuitively, instead of p sending a label to q from a possible set of labels, it sends this label as a value, and q tests the value received in order to decide on its behaviour. To adapt the construction in [9] to the variant of MC we consider in this work, the translation must also change the type of the value stored at each process p from T_{p} to $T_{\mathsf{p}} \times L$, where L is the set of possible labels; the state update function $\mathsf{upd}(,\mathsf{t},\mathsf{h})$en changes either component of the pair depending on the type of the value communicated.[5] The following lemmas state that selection elimination does not change the semantics of the original choreography in an essential way. In these lemmas, we write σ^{+} for a state obtained from σ by adding some label to the value stored at each process.

Lemma 2. *Choreographies* C *and* $(\![\mathsf{Amend}(C)]\!)^{+}$ *are equivalent wrt* $\mathsf{pn}(C)$.

[5]In the context of the original work [9] there were only two possible labels, making the encoding slightly simpler.

$$
\begin{aligned}
\{\!\{\mathsf{p}.e \texttt{ -> } \mathsf{q}; C\}\!\}_M &= \mathsf{p}.e \texttt{ -> } \mathsf{pq}^M;\ \mathsf{p}\,\mathsf{start}\,\mathsf{pq}^{M+};\ \mathsf{p} : \mathsf{pq}^M \texttt{ <-> } \mathsf{pq}^{M+};\\
&\quad \mathsf{pq}^M.\mathsf{q}\texttt{->}\mathsf{pq}^{M+};\ \mathsf{pq}^M.\mathsf{pq}^{M+}\texttt{->}\mathsf{q};\ \mathsf{pq}^M.\mathsf{c} \texttt{ -> } \mathsf{q};\\
&\quad \{\!\{C\}\!\}_{M[(\mathsf{p},\mathsf{q})\mapsto M(\mathsf{p},\mathsf{q})+1]}\\
\{\!\{\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\}\!\}_M &= \mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ \{\!\{C_1\}\!\}_M\ \mathsf{else}\ \{\!\{C_2\}\!\}_M\\
\{\!\{\mathbf{0}\}\!\}_M &= \mathbf{0}\\
\{\!\{X\}\!\}_M &= X\langle \bar{M}\rangle\\
\{\!\{\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1\}\!\}_M &= \mathsf{def}\ X(\overline{M_0}) = \{\!\{C_2\}\!\}_{M_0}\ \mathsf{in}\ \{\!\{C_1\}\!\}_M
\end{aligned}
$$

Figure 20: Encoding MC in DMC

Lemma 3. *Let C and C' be MC choreographies and σ and σ' be states. If $C, \sigma \to C', \sigma'$, then $(\![\mathsf{Amend}(C)]\!), \sigma^+ \to^* (\![C']\!), \sigma'^+$ for some σ^+ and σ'^+. Furthermore, the latter reduction consists of only one step except for the case when the former uses rule $\lfloor C|Cond\rceil$.*

Conversely, if $(\![\mathsf{Amend}(C)]\!), \sigma^+ \to C', \sigma'$, then there exist a choreography C'' and a state σ'' such that $C, \sigma \to C'', \sigma''$ and $C', \sigma' \to^? (\![\mathsf{Amend}(C'')]\!), \sigma''^+$ for some σ''^+. Furthermore, the latter reduction is non-empty only in the case when the former uses rule $\lfloor C|Cond\rceil$.

The proofs of all these results are straightforward adaptations of the corresponding proofs in [9]: all proofs either by induction on the structure of the choreography or by case analysis on its possible transitions, and the new cases that arise due to the presence of new constructors are trivially dealt with, since both Amend and $(\![\cdot]\!)$ treat these constructors homomorphically.

5.2. Asynchronously encoding MC in DMC

We focus now on the mapping (1) from Figure 17, as this is the key ingredient to establish the remaining connections in that figure. Let C be a choreography in MC. In order to encode C in DMC, we use a function $M_C : \mathscr{P}^2 \to \mathbb{N}$, where $\mathscr{P} = \mathsf{pn}(C)$ is the set of process names in C. We omit the subscript C when the choreography is clear. The countable set of auxiliary processes $\{\mathsf{pq}^i \mid \mathsf{p},\mathsf{q} \in \mathscr{P}, i \in \mathbb{N}\}$ is used to store messages in transit, with pq^i holding the ith message from p to q.

First, we setup initial channels for communications between all processes occurring in C.

$$\{\!\{C\}\!\} = \{\mathsf{p}\,\mathsf{start}\,\mathsf{pq}^0;\ \mathsf{p} : \mathsf{q} \texttt{<->} \mathsf{pq}^0\}_{p,q\in\mathscr{P},\mathsf{p}\neq\mathsf{q}}\,;\ \{\!\{C\}\!\}_{M_0}$$

Here, $M_0(\mathsf{p},\mathsf{q}) = 0$ for all p and q. For simplicity, we write pq^M for $\mathsf{pq}^{M(\mathsf{p},\mathsf{q})}$ and pq^{M+} for $\mathsf{pq}^{M(\mathsf{p},\mathsf{q})+1}$. The definition of $\{\!\{C\}\!\}_M$ is given in Figure 20. We write $\bar{M}$ for $\{\mathsf{pq}^M \mid \mathsf{p},\mathsf{q} \in \mathscr{P}, \mathsf{p} \neq \mathsf{q}\}$, where we assume that the order of the values of M is fixed. In recursive definitions, we reset M to M_0; note that the parameter declarations act as binders, so these process names are still fresh. We can use α-renaming on $\{\!\{C\}\!\}$ to make all bound names distinct.

In order to encode $\mathsf{p}.e$ -> q, p uses the auxiliary process pq^M to store the value it wants to send to q. Then, p creates a fresh process (to use in the next communication) and sends its name to pq^M. Afterwards, p is free to proceed with execution. In turn, pq^M communicates q's name to the new process, which now is ready to receive the next message from p. Finally, pq^M waits for q to be ready to receive both the value being communicated and the name of the process that will store the next value.

The encoding from C to $\{\!\{C\}\!\}$ does not constitute an asynchronous semantics for MC, since it expands communication actions syntactically, rather than semantically. We support our claim that $\{\!\{C\}\!\}$ is an asynchronous encoding of C by showing that, nevertheless, the two properties in Definition 5 hold. In these lemmas, we abuse notation and, given a state σ, we also write σ to denote an unspecified extension of σ to the auxiliary processes introduced by the encoding.[6]

[6]This imprecision significantly simplifies the statements and proofs, which do not depend on the precise values assigned to those spurious states.

Theorem 9. *Let C be an MC choreography and σ be a state, and assume that $C, \sigma \to C', \sigma'$ for some C' and σ'. For each labeling function M, let G_M be the graph such that $\mathsf{p} \xleftrightarrow{G} \mathsf{q}$, $\mathsf{p} \xleftrightarrow{G} \mathsf{pq}^M$ and $\mathsf{pq}^M \xleftrightarrow{G} \mathsf{q}$ for all $\mathsf{p}, \mathsf{q} \in \mathsf{pn}(C)$. Then for every labeling function M there exists M' such that $G_M, \{\!\!\{C\}\!\!\}_M, \sigma \to^* G', \{\!\!\{C'\}\!\!\}_{M'}, \sigma'$ for some graph G' containing $G_{M'}$.*

Proof. By induction on the derivation of the transition $C, \sigma \to C', \sigma'$.

- If the transition is justified directly by rule $\lfloor$C|Com$\rceil$, then C is of the form $\mathsf{p}.e$ -> $\mathsf{q}; C'$, $e \downarrow^{\mathsf{p}}_{\sigma} v$ and $\sigma' = \mathsf{upd}(\sigma, \mathsf{q}, v)$. In this case,

$$\begin{aligned}\{\!\!\{C\}\!\!\}_M = \{\!\!\{\mathsf{p}.e \texttt{ -> } \mathsf{q}; C\}\!\!\}_M = {} & \mathsf{p}\, \mathsf{start}\, \mathsf{pq}^{M+};\ \mathsf{p} : \mathsf{pq}^M \texttt{<->} \mathsf{pq}^{M+};\\ & \mathsf{pq}^M.\mathsf{q}\texttt{->}\mathsf{pq}^{M+};\ \mathsf{pq}^M.\mathsf{pq}^{M+}\texttt{->}\mathsf{q};\ \mathsf{pq}^M.\mathsf{c} \texttt{ -> } \mathsf{q};\\ & \{\!\!\{C'\}\!\!\}_{M[(\mathsf{p},\mathsf{q}) \mapsto M(\mathsf{p},\mathsf{q})+1]}\end{aligned}$$

  and therefore $G_M, \{\!\!\{C\}\!\!\}_M, \sigma$ reduces in six steps to $G', \{\!\!\{C'\}\!\!\}_{M[(\mathsf{p},\mathsf{q}) \mapsto M(\mathsf{p},\mathsf{q})+1]}, \mathsf{upd}(\sigma, \mathsf{p}, v)$, where $G' = G_M \cup \{(\mathsf{p}, \mathsf{pq}^{M+}), (\mathsf{pq}^M, \mathsf{pq}^{M+}), (\mathsf{q}, \mathsf{pq}^{M+})\}$.

- If the transition is justified directly by rule $\lfloor$C|Cond$\rceil$ and $e \downarrow^{\mathsf{p}}_{\sigma} \mathsf{true}$, then C is of the form $\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, $\sigma' = \sigma$, $C' = C_1$, and $\{\!\!\{C\}\!\!\}_M = \mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ \{\!\!\{C_1\}\!\!\}_M\ \mathsf{else}\ \{\!\!\{C_2\}\!\!\}_M$. In this case, $G_M, \{\!\!\{C\}\!\!\}_M, \sigma \to G_M, \{\!\!\{C_1\}\!\!\}_M, \sigma$. The case where $e \downarrow^{\mathsf{p}}_{\sigma} \mathsf{false}$ is analogous.

- If the transition is justified by rule $\lfloor$C|Ctx$\rceil$, then C is of the form $\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1$, C' is of the form $\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1'$ and $C_1, \sigma \to C_1', \sigma'$. Furthermore, $\{\!\!\{C\}\!\!\}_M = \mathsf{def}\ X(\overline{M_0}) = \{\!\!\{C_2\}\!\!\}_{M_0}\ \mathsf{in}\ \{\!\!\{C_1\}\!\!\}_M$.

  By induction hypothesis, $G_M, \{\!\!\{C_1\}\!\!\}_M, \sigma \to^* G', \{\!\!\{C_1'\}\!\!\}_{M'}, \sigma'$ for some G' and M' satisfying the thesis, from which we deduce that $G_M, \{\!\!\{\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1\}\!\!\}_M, \sigma \to G', \mathsf{def}\ X(\overline{M_0}) = \{\!\!\{C_2\}\!\!\}_{M_0}\ \mathsf{in}\ \{\!\!\{C_1'\}\!\!\}_{M'}, \sigma'$, and the thesis follows by observing that $\mathsf{def}\ X(\overline{M_0}) = \{\!\!\{C_2\}\!\!\}_{M_0}\ \mathsf{in}\ \{\!\!\{C_1'\}\!\!\}_{M'} = \{\!\!\{\mathsf{def}\ X(\overline{M_0}) = C_2\ \mathsf{in}\ C_1'\}\!\!\}_{M'}$.

- If the transition is justified by rule $\lfloor$C|Struct$\rceil$, then there exist C_0 and C_0' such that $C \preceq C_0$, $C_0' \preceq C'$ and $C_0, \sigma \to C_0', \sigma'$.

  By induction hypothesis, $G_M, \{\!\!\{C_0\}\!\!\}_M, \sigma \to^* G', \{\!\!\{C_0'\}\!\!\}_{M'}, \sigma'$ for some G' and M' in the conditions of the thesis. By induction, it is straightforward to prove that $C_1 \preceq C_2$ implies $\{\!\!\{C_1\}\!\!\}_{M_1} \preceq C_2 M_1$ for any M_1, from which it follows that $\{\!\!\{C\}\!\!\}_M \preceq \{\!\!\{C_0\}\!\!\}_M$ and $\{\!\!\{C_0'\}\!\!\}_{M'} \preceq \{\!\!\{C'\}\!\!\}_{M'}$. By applying rule $\lfloor$C|Struct$\rceil$ to the first and last transitions in the sequence $G_M, \{\!\!\{C_0\}\!\!\}_M, \sigma \to^* G', \{\!\!\{C_0'\}\!\!\}_{M'}, \sigma'$, we conclude that $G_M, \{\!\!\{C\}\!\!\}_M, \sigma \to^* G', \{\!\!\{C'\}\!\!\}_{M'}, \sigma'$.

□

Since the semantics of DMC is confluent, Theorem 9 states that messages sent from p to q are eventually received by q in every terminating computation.

Theorem 10. *Let C be an MC choreography, G be a connection graph, M be a labeling function and σ be a state, and assume that $G, \{\!\!\{C\}\!\!\}_M, \sigma \to^* G', C', \sigma'$ for some graph G', choreography C' and state σ'. Then there exist a graph G'', a choreography C'' and a state σ'' such that $C, \sigma \to^* C'', \sigma''$, and $G', C', \sigma' \to^* G'', \{\!\!\{C'\}\!\!\}_M, \sigma''$.*

Sketch. This proof is very similar to the proof of Theorem 5, except that in the second property the relevant case is when $G, \{\!\!\{C\}\!\!\}_M, \sigma$ reduces by application of rule $\lfloor$C|Start$\rceil$. Then C must be of the form $\mathsf{p}.e$ -> $\mathsf{q}; C_0$, and we can model the reduction $C, \sigma \to C_0, \sigma_0$ by consuming the five additional communication actions obtained by encoding $\mathsf{p}.e$ -> q. □

Theorem 10 states that the encoding does not add any additional behaviour to the original choreography, aside from expanding communications into several actions.

Example 5. *We again return to the setting of Example 1, and partially show the result of applying this transformation to the first three communication actions. We only include the initializations of the channels that are used in this fragment; the numbers indicated refer to the line numbers in the original choreography.*

$$\begin{array}{ll}
 & \mathsf{a}\ \mathsf{start}\ \mathsf{as}^0;\ \mathsf{a} : \mathsf{as}^0 \texttt{<->} \mathsf{s}; \\
 & \mathsf{s}\ \mathsf{start}\ \mathsf{sa}^0;\ \mathsf{s} : \mathsf{sa}^0 \texttt{<->} \mathsf{a}; \\
 & \mathsf{s}\ \mathsf{start}\ \mathsf{sb}^0;\ \mathsf{s} : \mathsf{sb}^0 \texttt{<->} \mathsf{b}; \\
1. & \mathsf{a}.\mathit{title} \texttt{->} \mathsf{as}^0;\ \mathsf{a}\ \mathsf{start}\ \mathsf{as}^1;\ \mathsf{a} : \mathsf{as}^1 \texttt{<->} \mathsf{as}^0; \\
 & \mathsf{as}^0.\mathsf{as}^1 \texttt{->} \mathsf{s};\ \mathsf{as}^0.\mathsf{s} \texttt{->} \mathsf{as}^1;\ \mathsf{as}^0.\mathsf{c} \texttt{->} \mathsf{s}; \\
2. & \mathsf{s}.\mathit{price} \texttt{->} \mathsf{sa}^0;\ \mathsf{s}\ \mathsf{start}\ \mathsf{sa}^1;\ \mathsf{s} : \mathsf{sa}^1 \texttt{<->} \mathsf{sa}^0; \\
 & \mathsf{sa}^0.\mathsf{sa}^1 \texttt{->} \mathsf{a};\ \mathsf{sa}^0.\mathsf{a} \texttt{->} \mathsf{sa}^1;\ \mathsf{sa}^0.\mathsf{c} \texttt{->} \mathsf{a}; \\
3. & \mathsf{s}.\mathit{price} \texttt{->} \mathsf{sb}^0;\ \mathsf{s}\ \mathsf{start}\ \mathsf{sb}^1;\ \mathsf{s} : \mathsf{sb}^1 \texttt{<->} \mathsf{sb}^0; \\
 & \mathsf{sb}^0.\mathsf{sb}^1 \texttt{->} \mathsf{b};\ \mathsf{sb}^0.\mathsf{b} \texttt{->} \mathsf{sb}^1;\ \mathsf{sb}^0.\mathsf{c} \texttt{->} \mathsf{b}; \\
 & \ldots
\end{array}$$

The first three lines initialize three channels: from a *to* s*; from* s *to* a*; and from* s *to* b*. Then one message is passed in each of these channels, as dictated by the encoding. All communications are asynchronous in the sense explained above, as in each case the main sender process sends its message to a dedicated intermediary (*as^0*,* sa^0 *or* sb^0*, respectively), who will eventually deliver it to the recipient. Moreover, causal dependencies are kept: in Step 2,* s *can only send its message to* sa^0 *after receiving the message sent by* a *in Step 1. However, in Step 3* s *can send its message to* sb^0 *without waiting for* a *to receive the previous message, as the action* $\mathsf{s}.\mathit{price} \texttt{->} \mathsf{sa}^0$ *can swap with the three actions immediately preceding it. (This is similar to the behaviour previously observed in Example 3.)*

We briefly illustrate Theorems 9 and 10 in this setting. From Theorem 9 we obtain that there is a reduction path of the encoded choreography that passes through the encodings of each reductum of the original choreography. Furthermore, since the original choreography terminates, we also know that, e.g., the action $\mathsf{a}.\mathit{title} \texttt{->} \mathsf{as}^0$ *is eventually followed by a communication of title from* as^0 *to* s*. Theorem 10 implies that if, e.g.,* $\mathsf{a}.\mathit{title} \texttt{->} \mathsf{as}^0$ *is executed, then it must be "part" of an action in the original choreography (in this case,* $\mathsf{a}.\mathit{title} \texttt{->} \mathsf{s}$*), and furthermore it is possible to find an execution path that will execute the remaining actions generated from that one (the remaining five actions in Step 1).*

5.3. Asynchrony in DMC and DCC

The calculus DMC is in itself synchronous, just like MC. We now show that we can extend $\{\!\{\cdot\}\!\}$ to the full language of DMC – arrow (2) in Figure 17 – thereby obtaining a systematic way to write asynchronous communications in DMC. By further marking which communications we want to treat as synchronous (so that they are untouched by $\{\!\{\cdot\}\!\}$) we obtain a calculus in which we can have both synchronous and asynchronous communication, compiled in itself. This is similar (albeit dual) to the situation in asynchronous π-calculus, where we can also encode synchronous communication without extending the language.

The main challenge is dealing with M, as the source choreography can now include process spawning. This means that the domain of M can be dynamically extended throughout the computation of $\{\!\{C\}\!\}_M$, which renders our parameter-passing in recursive calls invalid (since the number of parameters in the procedures generated by our encoding is fixed). However, since each procedure $X(\mathsf{p}_1, \ldots, \mathsf{p}_n)$ in DMC can only use (by convention) the processes $\mathsf{p}_1, \ldots, \mathsf{p}_n$ in its body, we can restrict the additional parameters introduced by the encoding to the $n(n-1)$ auxiliary processes currently assigned by M to communications between the p_is. For example, $\{\!\{\mathsf{def}\, X(\mathsf{p},\mathsf{q}) = C_2\ \mathsf{in}\, C_1\}\!\}_M$ becomes $\mathsf{def}\, X(\mathsf{p},\mathsf{q},\mathsf{pq}^0,\mathsf{qp}^0) = \{\!\{C_2\}\!\}_{M_0}\ \mathsf{in}\ \{\!\{C_1\}\!\}_M$. We assume the convention that these additional arguments are given in lexicographic ordering (where processes are ordered by their position in the original definition or call).

With this in mind, we can easily define the new cases for $\{\!\!\{C\}\!\!\}_M$.

$$\begin{aligned}
\{\!\!\{\mathsf{p\ start\ q}; C\}\!\!\}_M &= \mathsf{p\ start\ q};\ \mathsf{p\ start\ pq}^0;\ \mathsf{q\ start\ qp}^0;\\
&\quad \mathsf{p : q <\!\!-\!\!> pq}^0;\ \mathsf{q : p <\!\!-\!\!> qp}^0;\\
&\quad \{\!\!\{C\}\!\!\}_{M[(\mathsf{p},\mathsf{q})\mapsto 0,(\mathsf{q},\mathsf{p})\mapsto 0]}\\
\{\!\!\{\mathsf{p.q \to r}; C\}\!\!\}_M &= \mathsf{p\ start\ qr}^0;\ \mathsf{p.qr}^0 \mathsf{\to pq}^M;\ \mathsf{p.qr}^0 \mathsf{\to pr}^M;\\
&\quad \mathsf{p\ start\ pq}^{M+};\ \mathsf{p : pq}^M \mathsf{<\!\!-\!\!> pq}^{M+};\\
&\quad \mathsf{p\ start\ pr}^{M+};\ \mathsf{p : pr}^M \mathsf{<\!\!-\!\!> pr}^{M+};\\
&\quad \mathsf{pq}^M\mathsf{.q \to pq}^{M+};\ \mathsf{pr}^M\mathsf{.r \to pr}^{M+};\\
&\quad \mathsf{pq}^M\mathsf{.pq}^{M+} \mathsf{\to q};\ \mathsf{pq}^M\mathsf{.qr}^0 \mathsf{\to q};\\
&\quad \mathsf{pr}^M\mathsf{.pr}^{M+} \mathsf{\to r};\ \mathsf{pr}^M\mathsf{.qr}^0 \mathsf{\to r};\\
&\quad \{\!\!\{C\}\!\!\}_{M[(\mathsf{p},\mathsf{q})\mapsto M(\mathsf{p},\mathsf{q})+1,(\mathsf{p},\mathsf{r})\mapsto M(\mathsf{p},\mathsf{r})+1,(\mathsf{q},\mathsf{r})\mapsto 0]}
\end{aligned}$$

In $\{\!\!\{\mathsf{p\ start\ q}; C\}\!\!\}_M$, we simply create the asynchronous communication channels between p and q – the only step where these process will need to synchronize – and extend M in the continuation. The encoding of $\mathsf{p.q \to r}$ is better understood by reading it as a composition: first, p creates the new asynchronous communication channel from q to r, then uses its own channels to send this name to these processes. Note that the auxiliary channels do not communicate, so this encoding will introduce asymmetries in the graph of communications.

It is easy to verify the new cases of the proofs of Theorems 9 and 10, establishing that they still hold for this extended encoding.

Finally, we extend this encoding to the whole language of DCC – yielding arrow (4) in Figure 17 – by adding the clause

$$\begin{aligned}
\{\!\!\{\mathsf{p \to q}[\ell]; C\}\!\!\}_M &= \mathsf{p \to pq}^M[l];\ \mathsf{p\ start\ pq}^{M+};\\
&\quad \mathsf{p : pq}^M \mathsf{<\!\!-\!\!> pq}^{M+};\ \mathsf{pq}^M\mathsf{.q \to pq}^{M+};\\
&\quad \mathsf{pq}^M\mathsf{.pq}^{M+} \mathsf{\to q};\ \mathsf{pq}^M \mathsf{\to q}[l];\\
&\quad \{\!\!\{C\}\!\!\}_{M[(\mathsf{p},\mathsf{q})\mapsto M(\mathsf{p},\mathsf{q})+1]}
\end{aligned}$$

to the definition of $\{\!\!\{C\}\!\!\}_M$. Restricting this encoding to the language of CC yields arrow (3) in Figure 17.

Example 6. *Again in the setting of Example 1, we now assume that communications with* b *are supposed to be asynchronous, while the remaining ones are synchronous. The corresponding implementation would* *thus leave the communications in Lines 1, 2 and 6 unaffected, and only transform the remaining ones.*

```
    a start ab$^0$; a : ab$^0$ <-> b;
    b start ba$^0$; b : ba$^0$ <-> a;
    b start bs$^0$; b : bs$^0$ <-> s;
    s start sb$^0$; s : sb$^0$ <-> b;
1.  a.$title$ -> s
2.  s.$price$ -> a
3.  s.$price$ -> sb$^0$; s start sb$^1$; s : sb$^1$ <-> sb$^0$;
    sb$^0$.sb$^1$ -> b; sb$^0$.b -> sb$^1$; sb$^0$.c -> b;
4.  a.c -> ab$^0$; a start ab$^1$; a : ab$^1$ <-> ab$^0$;
    ab$^0$.ab$^1$ -> b; ab$^0$.b -> ab$^1$;
    if b.$happy$ then
5.      s.$price$ -> sb$^0$; s start sb$^1$; s : sb$^1$ <-> sb$^0$;
        sb$^0$.sb$^1$ -> b; sb$^0$.b -> sb$^1$; sb$^0$.c -> b;
    ...
```

5.4. Projectability

In general, DCC choreographies may deadlock because of attempted communications between processes that are not joined in the connection graph. We have not discussed this aspect yet, as it is orthogonal to the results in this section: Theorems 9 and 10 guarantee that a choreography deadlocks if and only if the choreography obtained by encoding its communications asynchronously deadlocks.

In order to regain properties such as Theorem 1 for DCC and DMC, we need to have some syntactic criteria to identify deadlock-free choreographies. A standard way to do this in choreography models is by means of a type system. We omit a formal discussion of a type system for DCC, since it can be constructed by restricting the type system for the choreography model of [11] to the constructors present in this language. From the results in that work, this type system enjoys type inference and decidability of type checking. Typable choreographies are deadlock-free. Furthermore, we can define an EndPoint Projection to a process calculus of Dynamic Stateful Processes (DSP) such that the correspondence in Theorem 2 holds. As a consequence, projections of typable choreographies are again deadlock-free by construction.

More interestingly, our encoding for asynchronous communications is designed to ensure that typability and projectability are preserved, i.e., if C is typable/projectable, then $\{\!\!\{C\}\!\!\}$ is also typable/projectable. As a result, the correspondences described in Theorems 9 and 10 for the different choreography models also hold at the process level.

5.5. Overview of correspondences

We now summarize the operational correspondences between the main calculi in this article (see Figure 21). Similar relationships exist between the restricted calculi without label selections, as they are all subcalculi of the languages included here.

The solid arrows in Figure 21 refer to the one-to-one correspondence described by the EPP theorems for each of the calculi (stated in Theorem 2 for CC and SP, in Theorem 7 for aCC and aSP, and discussed informally in the previous section, for DCC and DSP).

The dashed arrows between choreography languages express the expansion simulation obtained by either endowing CC with asynchronous semantics (arrow from CC to aCC, formalized as Theorem 5, first property) or by encoding communications asynchronously (arrow from CC to DCC, Theorem 9). The arrow from aCC to DCC can be obtained by extending the encoding from CC to DCC to runtime terms in aCC, using ideas similar to those in Definition 7: messages in transit need to be stored at the right auxiliary processes, and

$$\begin{tikzcd}
 & & \mathrm{SP} \arrow[d, dash] & & \\
 & & \mathrm{CC} \arrow[dl, dash] \arrow[dr, dash] & & \\
 & \mathrm{aCC} \arrow[rr, dash] \arrow[dl, dash] & & \mathrm{DCC} \arrow[dr, dash] & \\
\mathrm{aSP} & & & & \mathrm{DSP}
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & \llbracket C, \sigma \rrbracket \arrow[d, leftrightarrow] \arrow[ddll, dashed, bend right, "*"'] \arrow[ddrr, dashed, bend left, "*"] & & \\
 & & C, \sigma \arrow[dl, dashed, bend right] \arrow[dr, dashed, bend left] & & \\
 & C, \sigma \arrow[ur, dotted] \arrow[rr, dashed, bend right] \arrow[dl, leftrightarrow] & & \{\!\{ C \}\!\}_M, \sigma \arrow[ul, dotted] \arrow[ll, dotted] \arrow[dr, leftrightarrow] & \\
\llbracket C, \sigma \rrbracket_a \arrow[uurr, dotted] \arrow[rrrr, dashed, bend right, "*"'] & & & & \llbracket \{\!\{ C \}\!\}_M, \sigma \rrbracket_D \arrow[uull, dotted] \arrow[llll, dotted, bend right, "*"']
\end{tikzcd}$$

Figure 21: The diagram on the right summarizes the simulation relationships between the different choreography and process calculi: one-to-one (solid), one-to-many (dashed) or many-to-one (dotted). The correspondences marked $(*)$ also hold for arbitrary networks, as explained in the main text. For reference, the diagram on the left indicates the choreography/process language each term lives in.

some assumptions need to be made regarding the connection graphs. We do not formalize this translation, as it is a tedious but relatively straightforward extension of our development.

The correspondence between SP and aSP was stated explicitly as Theorem 6. In order to obtain similar results for DSP, we need to define translations from SP/aSP networks into DSP networks. Again, this is a simple extension of our constructions, since we can use the EPP to infer how these translations should be defined. The results in this paper imply that these translations give expansion simulations for networks in the fragments of SP and aSP obtained by projecting choreographies in CC or aCC, but the correspondences can be shown to hold for the entire language in both cases.

The last correspondences are the dotted arrows, which state many-to-one correspondences: if, for example, a choreography in aCC makes a reduction, then the reductum may need to make some more reductions in order to reach a reductum of the original choreography in CC (property (2) in the proof of Theorem 5). The corresponding relationship between DCC and CC is also stated as part of the proof of Theorem 10. The dotted arrow from DCC to aCC is similar, and states that the asynchronous encoding is more fine-grained than the asynchronous semantics: this is not surprising, since a communication action in CC becomes two actions in aCC, but six actions in DCC.

These correspondences do not propagate automatically to the process level, since the EPP theorems only guarantee that similar correspondences hold for projections of choreographies (see the discussion after Theorem 6: a deadlocked SP choreography may reduce under asynchronous semantics). Therefore, we can only state them for networks that are obtained as choreography projections. The exception is the correspondence between aSP and DSP: since both these calculi are inherently asynchronous, it is still the case that encoding a network from aSP into DSP by making its communications pass through intermediate auxiliary processes does not add new possible reductions.

6. Related Work

Asynchronous semantics for choreographies. The first work that introduced an asynchronous semantics to choreographies is [5], as described in the introduction. The same approach was later adapted to compositional choreographies [27] and multiparty session types [22]. The models presented in these works all suffer from the shortcoming discussed in the introduction: choreographies can reduce to states that would normally not be reachable in the real world.

We can now give a formal example of the consequences of this discrepancy.

Example 7. *Let C be the choreography from the introduction: $C \triangleq \mathsf{p.1} \texttt{ -> } \mathsf{q}; \mathsf{p.2} \texttt{ -> } \mathsf{r}$. If we adopted the* *asynchronous semantics from [5], then the reduction*

$$C, \sigma \to \mathsf{p}.1 \text{ -> } \mathsf{q}, \mathsf{upd}(\sigma, \mathsf{r}, 2)$$

would be possible for any σ. This shows that Theorem 7 (the completeness direction) does not hold in this semantics, since $[\![C, \sigma]\!]$ cannot reduce to any N such that $[\![\mathsf{p}.1 \text{ -> } \mathsf{q}, \mathsf{upd}(\sigma, \mathsf{r}, 2)]\!] \preceq N$ (we have to consume the first output by p *first).*

Therefore, if we want to formalise the correspondence between a choreography and its EPP in this setting, we need to consider multiple steps. In this example, specifically, we can observe that

$$\mathsf{p}.1 \text{ -> } \mathsf{q}, \mathsf{upd}(\sigma, \mathsf{r}, 2) \to \mathbf{0}, \mathsf{upd}(\mathsf{upd}(\sigma, \mathsf{r}, 2), \mathsf{q}, 1)$$

and that $[\![C, \sigma]\!] \to^* [\![\mathbf{0}, \mathsf{upd}(\mathsf{upd}(\sigma, \mathsf{r}, 2), \mathsf{q}, 1)]\!]$.

In general, the EPP Theorem with this kind of asynchronous semantics is weaker and more complicated. We report it here by adapting the formulation from [26, Chapter 2] (the full version of [5]) to our notation:

Theorem 11. *If C is a projectable choreography, then, for all σ:*

- *(Completeness) if $C, \sigma \to C', \sigma'$, then $C', \sigma' \to^* C'', \sigma''$ for some C'', σ'' and $[\![C, \sigma]\!] \to^* [\![C'', \sigma'']\!]$;*
- *(Soundness) if $[\![C, \sigma]\!] \to^* N$, then $N \to^* N'$ for some N', and $C, \sigma \to^* C', \sigma'$ for some σ' and C' such that $[\![C', \sigma']\!] \preceq N'$.*

Compared to our Theorem 7 above, Theorem 11 is missing the step-by-step correspondence between choreographies and their projections, and therefore does not say much about the intermediate steps.

In [17], multiparty session types are equipped with runtime terms that represent messages in transit in asynchronous communications, similarly to our approach. Differently from our model, the semantics in [17] uses a labelled transition system (instead of out-of-order execution) to identify when a communication can be executed asynchronously. The difference between these two systems seems to be mostly a matter of presentation, but their expressive powers cannot be directly compared because a counterpart to Theorem 5 is not provided for the model of [17].

A more recent development that is nearer to ours is the model of Applied Choreographies [18], where out-of-order execution (modelled as structural equivalences) is used to swap independent partial choreographic actions, which contain terms that are similar to our runtime asynchronous terms. However, the asynchronous terms in Applied Choreographies are not enough to ensure that the semantics is sound; therefore, these choreographies also need to include queues to store messages in transit. This makes their development substantially more complicated than the one we propose. Furthermore, the work in [18] focuses on implementation models, and does not provide a formal definition of asynchrony for choreographies as in this paper. There is also no correspondence result relating Applied Choreographies to a standard synchronous choreography semantics, although we conjecture that it is possible to map a fragment of that language into an asynchronous extension of CC in the sense of our definition.

Our approach relies on out-of-order execution for choreographic interactions, which was first introduced in [5] to capture parallel execution. We extended it to support the swapping of interactions supported by asynchronous message passing. Out-of-order execution does not introduce any burden on the programmer, since only non-interfering interactions can be swapped and all swappings are thus safe by design (more precisely, swaps correspond to the parallel semantics of typical process calculi [26]); instead, they just model different safe ways of executing the same choreography.

Encoding asynchrony in choreographies. To the best of our knowledge, this is the first work that presenting an interpretation of asynchronous communications in choreographies based solely on the expressive power of primitives for the creation of processes and their connections, via name mobility.

Our development in Section 5 recalls the development of the asynchronous π-calculus [21] (Aπ for short, using the terminology from [30]). Aπ has a synchronous semantics, in the sense that two processes can communicate when they are both ready to, respectively, perform compatible input and output actions.

However, an output action can have no sequential continuation, but can instead only be composed in parallel with other behaviour. Thus, the interpretation of communications in Aπ is asynchronous, since outputs can be seen as messages in transit over a network. The synchronisation between (the process holding) a message in transit and the intended receiver models then the extraction of the message from the medium by the receiver. Differently from our work, Aπ is obtained from the standard π-calculus by *restricting* the syntax of processes such that all communications necessarily conform to this asynchronous interpretation. It is then shown that Aπ is expressive enough to encode the synchronous communications from standard π-calculus, by using acknowledgement messages. DMC and DCC exhibit the dual behaviour: communications are naturally synchronous, but we can always encode them to be asynchronous by passing them through intermediary processes.

Process spawning and name mobility are the key additions to DCC and DMC, from CC and MC, that yield the expressive power to represent asynchronous communications. Process spawning in choreographies has been studied also in the works [4, 5, 27], but in a different form where processes have to synchronise over a shared channel to proceed. Name mobility in choreographies was introduced in [5], but for channel rather than process names. Our process spawning and name mobility primitives are simplifications of those presented in [11], which makes all results from that work applicable to DCC (and thus DMC).

7. Conclusions

This article discusses asynchrony in choreographic programming. Within the context of the minimal choreography language CC, we first identified abstractly the properties that an asynchronous semantics should have. Then, we extended CC with runtime terms that store messages in transit in order to endow the resulting calculus aCC with a semantics that enjoys these properties (Theorem 4) and is equivalent to the original semantics for CC in a precisely defined sense (Theorem 5). We showed how the EndPoint projection for CC can be extended to aCC, yielding terms in the same target process calculus SP, and that this extension still enjoys a correspondence theorem (Theorem 7) with respect to an asynchronous semantics for SP that uses queues of incoming messages at each process. In comparison with the traditional approach to asynchrony in choreographies [5], the correspondence guaranteed by the EPP theorem is more precise in our case, and the proof of this theorem is significantly simpler.

Our construction was deliberately made in a modular way: it changes only the rules for communications, leaving the rest untouched, so that it can be easily applied to more expressive languages. In particular, our semantics can be orthogonally applied to the models in [4, 5, 7, 8, 16, 24, 27], since it only changes the behaviour of communication actions. Furthermore, other types of communication (e.g. name mobility [5, 11]) can be treated in a similar way.

A second approach to asynchrony relies on encoding it syntactically, rather than semantically, by relying on auxiliary processes to store messages in transit. We showed that CC is not expressive enough for this purpose, given that the number of messages in transit is in general unbounded. Therefore, we extended CC with a primitive for spawning new processes at runtime. Extending CC with only this primitive would yield a minimal calculus for our purposes, but in order to obtain a realisable model we also argued that we need a primitive for communicating process names along the network. The resulting calculus DCC is essentially a subcalculus of the choreography model from [11]. We showed that we could simulate asynchronous communications by having senders spawn a new process for each message, storing the message until the receiver is ready to consume it, and thus unblocking the sender for further actions. We showed that this encoding also does not fundamentally change the behaviour of the original choreography (Theorems 9 and 10). Furthermore, it can be extended to process spawning and name mobility to an encoding of the whole language of DCC in itself.

As before, this construction can be applied in any calculus that contains the primitives in DCC, such as the language of Procedural Choreographies [11].

The extension of CC to DCC consists of two primitives: one for spawning new processes, such that we can have an unbounded number of processes at runtime, and another for name mobility, such that the spawned processes can be discovered. Intuitively, this is a minimal extension, in the sense that we strictly need both primitives to simulate asynchronous behaviour in CC. In particular, spawning new processes is necessary for concurrency reasons. To understand why, consider the choreography snippet $\mathsf{p}.e$ -> q; $\mathsf{p}.e'$ -> q. In an asynchronous setting, there is no order restriction between p sending e' and q receiving e. If we relied on a single process (acting as a channel) to relay messages from p to q, then this process would need the ability to offer both those capabilities after receiving message e from p. This requires changing the semantics of both choreographies and processes to allow for swapping communication actions performed by the same process, as in [14]. Such a change, however, is much more intrusive than adding a new primitive (process spawn), because it alters fundamentally the semantics of communications themselves. As a result, this alternative development would not be modular and would not be readily applicable to other existing choreography models.

Instead of out-of-order execution, some choreography models also include explicit parallel composition, e.g., $C \mid C'$ [4, 29]. Most behaviours of $C \mid C'$ are captured by out-of-order execution, for example $\mathsf{p}.e$ -> q | $\mathsf{r}.e'$ -> s is equivalent to $\mathsf{p}.e$ -> q; $\mathsf{r}.e'$ -> s in CC (due to rule $\lfloor$C|Eta-Eta$\rceil$ in Figure 2) – see [5] for a deeper discussion. Generalising our construction to supporting also an explicit parallel operator is straightforward, since structural precongruence can be extended homomorphically without surprises.

A more interesting extension is multicom [7], a primitive that considerably extends the expressivity of choreographies by allowing for general criss-cross communications. A prime example is the asynchronous exchange $\{\mathsf{p}.e$ -> $\mathsf{q}, \mathsf{q}.e'$ -> $\mathsf{p}\}$, which allows two processes to exchange message without waiting for each other (this is the building block, e.g., of the alternating 2-bit protocol given in [7]). Multicoms crucially depend on asynchrony, making them an interesting case study in our context. Our development applies immediately to multicoms, since we can just apply the expansion rule $\lfloor$C|Com-Unfold$\rceil$ to all communications in a multicom simultaneously.

Acknowledgements. This work was partially supported by CRC (Choreographies for Reliable and efficient Communication software), grant no. DFF–4005-00304 from the Danish Council for Independent Research, the Open Data Framework project at the University of Southern Denmark, and by the Independent Research Fund Denmark, Natural Sciences, grant DFF-7014-00041.

References

[1] Elvira Albert and Ivan Lanese, editors. *Formal Techniques for Distributed Objects, Components, and Systems – 36th IFIP WG 6.1 International Conference, FORTE 2016*, volume 9688 of *LNCS*. Springer, 2016.

[2] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.

[3] Business Process Model and Notation. `http://www.omg.org/spec/BPMN/2.0/`.

[4] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.

[5] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274. ACM, 2013.

[6] Chor. Programming Language. `http://www.chor-lang.org/`.

[7] Luís Cruz-Filipe, Kim S. Larsen, and Fabrizio Montesi. The paths to choreography extraction. In Javier Esparza and Andrzej S. Murawski, editors, *FoSSaCS*, volume 10203 of *LNCS*, pages 424–440. Springer, 2017.

[8] Luís Cruz-Filipe and Fabrizio Montesi. Choreographies in practice. In Albert and Lanese [1], pages 114–123.

[9] Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. In Olga Kouchnarenko and Ramtin Khosravi, editors, *FACS*, volume 10231 of *LNCS*, pages 17–35. Springer, 2017.

[10] Luís Cruz-Filipe and Fabrizio Montesi. Encoding asynchrony in choreographies. In D. Shin and M. Lencastre, editors, *Proceedings of SAC 2017*, pages 1175–1177, 2017.

[11] Luís Cruz-Filipe and Fabrizio Montesi. Procedural choreographic programming. In Ahmed Bouajjani and Alexandra Silva, editors, *Proceedings of FORTE 2017*, volume 10321 of *LNCS*, pages 92–107. Springer, 2017.

[12] Luís Cruz-Filipe and Fabrizio Montesi. That's enough: Asynchrony with standard choreography primitives. In Carlos Caleiro, Francisco Dionísio, Paula Gouveia, Paulo Mateus, and João Rasga, editors, *Logic and Computation: Essays in Honour of Amílcar Sernadas*, volume 33 of *Tributes*, pages 125–142. College Publications, 2017.

[13] Luís Cruz-Filipe and Fabrizio Montesi. On asynchrony and choreographies. In *Proceedings of ICE 2017*, accepted for publication.

[14] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Communications in choreographies, revisited. In *Proceedings of SAC 2018*, 2018. Accepted for publication.

[15] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Deadlock freedom by construction for distributed adaptive applications. *CoRR*, abs/1407.0970, 2014.

[16] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies. In *COORDINATION*, LNCS, pages 67–82. Springer, 2015.

[17] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP (2)*, pages 174–186, 2013.

[18] Maurizio Gabbrielli, Saverio Giallorenzo, and Fabrizio Montesi. Applied choreographies. *CoRR*, abs/1510.03637, 2015.

[19] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010.

[20] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In Raja Natarajan and Adegboyega K. Ojo, editors, *Proceedings of ICDCIT 2011*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.

[21] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of ECOOP'91*, volume 512 of *LNCS*, pages 133–147. Springer, 1991.

[22] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *J. ACM*, 63(1):9, 2016.

[23] Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *SEFM*, pages 323–332, 2008.

[24] Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. Enforcing availability in failure-aware communicating systems. In Albert and Lanese [1], pages 195–211.

[25] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

[26] Fabrizio Montesi. *Choreographic Programming.* Ph.D. Thesis, IT University of Copenhagen, 2013. `http://fabriziomontesi.com/files/choreographic_programming.pdf`.

[27] Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.

[28] PI4SOA. http://www.pi4soa.org, 2008.

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

[30] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

[31] Savara. JBoss Community. `http://www.jboss.org/savara/`.

[32] W3C WS-CDL Working Group. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/, 2004.