That's Enough: Asynchrony with Standard Choreography Primitives

Luís Cruz-Filipe and Fabrizio Montesi
Dept. Mathematics and Computer Science
University of Southern Denmark
{lcf,fmontesi}@imada.sdu.dk

Abstract

Choreographies are widely used for the specification of concurrent and distributed software architectures. Since asynchronous communications are ubiquitous in real-world systems, previous works have proposed different approaches for the formal modelling of asynchrony in choreographies. Such approaches typically rely on ad-hoc syntactic terms or semantics for capturing the concept of messages in transit, yielding different formalisms that have to be studied separately.

In this work, we take a different approach, and show that such extensions are not needed to reason about asynchronous communications in choreographies. Rather, we demonstrate how a standard choreography calculus already has all the needed expressive power to encode messages in transit (and thus asynchronous communications) through the primitives of process spawning and name mobility. The practical consequence of our results is that we can reason about real-world systems within a choreography formalism that is simpler than those hitherto proposed.

1 Introduction

Today, concurrent and distributed systems are widespread. Multi-core hardware and large-scale networks represent the norm rather than the exception. However, programming such systems is challenging, because it is difficult to program correctly the intended interactions among components executed concurrently (e.g., services). Empirical investigations of bugs in concurrent and distributed software [17, 18] reveal that most errors are due to: deadlocks (e.g., a component that was supposed to be ready for interaction at a given time is actually not); violations of atomicity intentions (e.g., a component is performing some action when not intended to); or, violations of ordering intentions (some components perform the right actions, but not when intended). If the design and implementation of a concurrent system are initially difficult, they get even harder as the system evolves and has to be maintained. Without proper tool support, introducing new actions at components may have unexpected effects due to side-effects.

To mitigate this problem, *choreographies* can be used as high-level formal specifications of the intended interactions among components [1, 2, 3, 4, 15, 16, 24, 26].

Example 1. *We use a choreography to define a scenario 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 $\overset{\leftarrow}{=}$ a 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.*

Choreographies are the foundations of an emerging development paradigm, called Choreographic Programming [20, 21], where an automatic projection procedure is used to synthesise a set of compliant local implementations (the implementations of the single components) from a choreography [4, 16, 24]. This procedure is formally proven to be correct, preventing deadlocks, ordering errors, and atomicity violations. This ensures, critically, that updates to either the choreography or the local implementations do not introduce bugs and that developers always know what communications their systems will enact (by looking at the choreography). In the previous example, the implementation inferred for, e.g., Alice (a), would be: send the book title to s; receive the price from s; send the price to b for confirmation; await the success/failure notification from b; in case of success, receive the book from s.

Choreography languages come in all sizes and flavours, with different sets of primitives inspired by practical applications, such as adaptation [9, 10], channel mobility [5, 6], or web services [2, 4, 26]. However, this multiplicity makes it increasingly difficult to reuse available theory and tools, because of the differences and redundancies among these models. For this reason, we previously introduced the model of Core Choreographies (CC) [8], a minimal and representative theoretical model of Choreographic Programming. In CC, components are modelled as concurrent processes that run independently and possess own memory, inspired by process calculi [25]. Example 1 is written in the syntax of CC described in § 2.

In this paper, we are interested in studying asynchronous communications in choreographies. As a motivation, consider the two communications in Lines
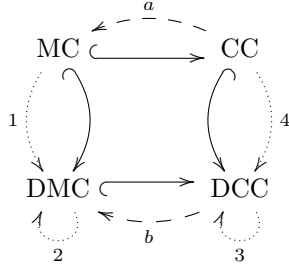
Figure 1: Choreography calculi and encodings.

2 and 3 of Example 1: typically, in a realistic system, we would expect s to send the price to a and then immediately proceed to sending it also to b, without waiting for a to receive its message. Typically, asynchronous communications are formalised in choreography models by defining ad-hoc extensions to their syntax and semantics [5, 11, 15, 16, 22, 23], causing a substantial amount of duplication in their technical developments (many of which are even incompatible with each other).

Unfortunately, there are still no foundational studies that provide an elegant and general understanding of asynchrony in choreographies. Here, we pursue such a study in the context of CC. We depict our overall development in Figure 1, and describe it in the following.

We first present our development for the computational fragment of CC, called Minimal Choreographies (MC) [8]. We take inspiration from how asynchrony is modelled in foundational process models, specifically the π-calculus [19]. 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 [25]. In an asynchronous system, there is no bound to the number of messages that could be transiting in the network; this means that MC is not powerful enough for our purposes, because it can only capture a finite number of processes (the same holds for CC). For this reason, we extend MC 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 new language Dynamic Minimal Choreographies (DMC). MC is a strict sub-language of DMC, denoted by the arrow $\hookrightarrow$ on the left-hand side of Figure 1. In general, all arrows of shape $\hookrightarrow$ in that figure denote (strict) language inclusion.

The dotted arrow (1) in Figure 1 is the cornerstone of our development: 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). This syntactic interpretation of asynchrony in choreographies is our main contribution. Specifically, our results show that asynchronous communications can be modelled in choreographies using well-known notions, i.e., process spawning and name mobility (studied, e.g., in [5,

3

$$C ::= \mathbf{0} \mid \eta; C \mid \text{if } \mathsf{p} \overset{\leftarrow}{=} \mathsf{q} \text{ then } C_1 \text{ else } C_2 \mid \text{def } X = C_2 \text{ in } C_1 \mid X$$
$$\eta ::= \mathsf{p}.e \mathrel{\text{->}} \mathsf{q} \mid \mathsf{p} \mathrel{\text{->}} \mathsf{q}[l] \qquad\qquad e ::= v \mid * \mid \ldots$$

Figure 2: Core Choreographies, Syntax.

7]), without the need for ad-hoc constructions. Coming back to the title: we already have enough.

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 sub-languages [25].) We show an example of such reuses. Core Choreographies (CC) [8] is MC with the addition of a primitive for communicating choices explicitly as messages, called selection [4, 13, 15, 27] (the terms in Lines 5 and 7 in Example 1 are selections). An important property of CC is that selections can be encoded in the simpler language MC – the dashed arrow (a) in Figure 1. What happens if we add selections to DMC? Ideally, the resulting calculus (called Dynamic Core Choreographies, or DCC) should *both* have an asynchronous interpretation through the techniques introduced in this paper and still possess the property that selections are encodable using the simpler language DMC. This is indeed the case. We extend our encoding to yield an interpretation of asynchronous selections, yielding (3) and (4). The second property (encodability of selections in DCC) follows immediately from language inclusion, giving us (b) for free.

2 Background

We briefly introduce CC and MC, from [8], and summarise their key properties.

The syntax of CC is given in Figure 2, where C ranges over choreographies. Processes ($\mathsf{p}, \mathsf{q}, \ldots$) run in parallel, and each process stores a value in a local memory cell.[1] Each process can access its own value using the syntactic construct $*$, but it cannot read the contents of another process. Term $\eta; C$ is an interaction between two processes, read "the system may execute η and proceed as C". In a value communication $\mathsf{p}.e \mathrel{\text{->}} \mathsf{q}$, p sends its local evaluation of expression e to q, which stores the received value. In a label selection $\mathsf{p} \mathrel{\text{->}} \mathsf{q}[l]$, p communicates label l to q. The set of labels is immaterial, as long as it contains at least two elements. In a conditional if $\mathsf{p} \overset{\leftarrow}{=} \mathsf{q}$ then C_1 else C_2, q sends its value to p, which checks if the received value is equal to its own; the choreography proceeds as C_1, if that is the case, or as C_2, otherwise. In

---

[1] In the original presentation, values were restricted to natural numbers; we drop this restriction here since it is orthogonal to our development.

4

$$\frac{e[\sigma(\mathsf{p})/*]\downarrow v}{\mathsf{p}.e \mathbin{\text{->}} \mathsf{q}; C, \sigma \to C, \sigma[\mathsf{q} \mapsto v]} \ \lfloor \mathrm{C}|\mathrm{Com} \rfloor$$

$$\frac{}{\mathsf{p} \mathbin{\text{->}} \mathsf{q}[l]; C, \sigma \to C, \sigma} \ \lfloor \mathrm{C}|\mathrm{Sel} \rfloor$$

$$\frac{i = 1 \text{ if } \sigma(\mathsf{p}) = \sigma(\mathsf{q}), \ i = 2 \text{ o.w.}}{\text{if } \mathsf{p} \overset{\Leftarrow}{=} \mathsf{q} \text{ then } C_1 \text{ else } C_2, \sigma \to C_i, \sigma} \ \lfloor \mathrm{C}|\mathrm{Cond} \rfloor$$

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

$$\frac{C_1 \preceq C_2 \quad C_2, \sigma \to C_2', \sigma' \quad C_2' \preceq C_1'}{C_1, \sigma \to C_1', \sigma'} \ \lfloor \mathrm{C}|\mathrm{Struct} \rfloor$$

Figure 3: Core Choreographies, Semantics.

all these actions, the two interacting processes must be different. Definitions and invocations of recursive procedures (X) are standard. The term $\mathbf{0}$ is the terminated choreography.

The semantics of CC uses reductions of the form $C, \sigma \to C', \sigma'$, where the total state function σ maps each process name to its value. We use v, w, ... to range over values. The reduction relation $\to$ is defined by the rules given in Figure 3.

These rules formalise the intuition presented earlier. In the premise of $\lfloor \mathrm{C}|\mathrm{Com} \rfloor$, we write $e[\sigma(\mathsf{p})/*]$ for the result of replacing $*$ with $\sigma(\mathsf{p})$ in e. In the reductum, $\sigma[\mathsf{q} \mapsto v]$ denotes the updated state function σ where q now maps to v.

Rule $\lfloor \mathrm{C}|\mathrm{Struct} \rfloor$ uses the structural precongruence relation $\preceq$, which gives a concurrent interpretation to choreographies by allowing non-interfering actions to be executed in any order. The key rule defining $\preceq$ is

$$\frac{\mathsf{pn}(\eta) \cap \mathsf{pn}(\eta') = \emptyset}{\eta; \eta' \ \equiv \ \eta'; \eta} \ \lfloor \mathrm{C}|\mathrm{Eta\text{-}Eta} \rfloor$$

where $C \equiv C'$ stands for $C \preceq C'$ and $C' \preceq C$ and $\mathsf{pn}(C)$ returns the set of all process names occurring in C. The other rules for $\preceq$ are standard, and support recursion unfolding and garbage collection of unused definitions.

CC was designed as a core choreography language, in which in particular it is possible to implement any computable function. Furthermore, CC choreographies can always progress until they terminate.

Theorem 1. *If C is a choreography, then either $C \preceq \mathbf{0}$ (C has terminated) or, for all σ, $C, \sigma \to C', \sigma'$ for some C' and σ' (C can reduce).*

Label selections are not required for Turing completeness, and thus the simpler fragment MC obtained from CC by omitting them is interesting as an intermediate language for compilers and, also, for theoretical analysis. One of

the reasons for having label selection is to make choice propagation explicit in choreographies; in a system implementation, this allows, e.g., to monitor distributed choices without having to inspect the message payload. Another reason is *projectability*: the possibility of automatically generating processes implementations that satisfy the choreographic specification. In Example 1, the label selections in Lines 5 and 7 are important in order for b to let a and s know whether or not they should communicate.

Choices communicated by label selections can also be encoded as data in value communications, by sending a boolean value to determine which one of two branches was selected. This is the key idea behind the encoding presented in [8] – arrow (a) in Figure 1 – which transforms a choreography in CC to one in MC by encoding selections as value communications and nested conditionals.

We do not need to concern ourselves with projectability in this work, and we will thus omit its details. This is because CC and MC enjoy a projectability property that is not altered by our development. Formally, there exists a procedure $\mathsf{Amend}(\cdot)$ that, given any choreography, returns a choreography in CC that is projectable. Then, given a projectable CC choreography, the encoding $(\![\cdot]\!)$ transforms it into a choreography in MC, by encoding selections as value communications and conditionals. These transformations preserve the computational meaning of choreographies, as formally stated in the following theorem ($\cdot^+$ extends a state function to the auxiliary processes introduced by the transformations in a systematic way).

Theorem 2. *Let C, C' be MC choreographies and σ, σ' be states. If $C, \sigma \to^* C', \sigma'$, then $(\![\mathsf{Amend}(C)]\!)^+, \sigma^+ \to^* (\![\mathsf{Amend}(C')]\!), \sigma'^+$.*

The main limitation of CC is that its semantics is synchronous. Indeed, in a real-world scenario implementation of Example 1, we would expect s to proceed immediately to sending its message in Line 3 after having sent the one in Line 2, without waiting for a to receive the latter. Capturing this kind of asynchronous behaviour is the main objective of our development in the remainder of this paper.

3   Asynchrony in MC

In this section, we extend CC with primitives to implement asynchronous communication, obtaining a calculus of Dynamic Core Choreographies (DCC). We focus on MC and first show that any MC choreography can be encoded in DMC – the fragment of DCC that does not use label selection – in such a way that communication becomes asynchronous.

More precisely, we provide a mapping $\{\![\cdot]\!\} : \mathrm{MC} \to \mathrm{DMC}$ such that every communication action $\mathsf{p}.e \to \mathsf{q} \in C \in \mathrm{MC}$ becomes split into a send/receive pair in $\{\![C]\!\} \in \mathrm{DMC}$, with the properties that: p can continue executing without waiting for q to receive its message (and even send further messages to q); and messages from p to q are delivered in the same order as they were originally sent.

The system DCC. We briefly motivate DCC. In CC, there is a bound on the number of values that can be stored at any given time by the system:

$$C ::= \cdots \mid \mathsf{def}\, X(\tilde{\mathsf{p}}) = C_2 \,\mathsf{in}\, C_1 \mid X\langle\tilde{\mathsf{p}}\rangle$$

$$\eta ::= \cdots \mid \mathsf{p}\,\mathsf{starts}\,\mathsf{q} \qquad e ::= \mathsf{p} \mid \cdots$$

Figure 4: Dynamic Core Choreographies, Syntax.

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 CC with the capability to generate new processes. As discussed in [7], 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 action to communicate process names.

Formally, the differences between the syntax of CC and that of DCC are highlighted in Figure 4: 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 ($\mathsf{p.q} \rightarrow \mathsf{r}$) ensures name mobility. We will use the abbreviation $\mathsf{p} : \mathsf{r} \mathrel{<->} \mathsf{q}$ as shorthand for $\mathsf{p.q} \rightarrow \mathsf{r}; \mathsf{p.r} \rightarrow \mathsf{q}$.

The semantics for DCC includes an additional ingredient, borrowed from [7]: 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 $\mathsf{p} \xrightarrow{G} \mathsf{q}$) 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 $\mathsf{p} \xleftrightarrow{G} \mathsf{q}$.[2] The reduction relation now has the form $G, C, \sigma \rightarrow G', C', \sigma'$, where G and G' are the connection graphs before and after executing C, respectively. The complete rules are given in Figure 5, with $\preceq$ defined similarly to CC. In rule $\lfloor\mathsf{C|Start}\rfloor$, the fresh process q is assigned a default value $\bot$.

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 [7]). 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 DCC and an EndPoint Projection that will automatically synthetise correct-by-construction deadlock-free implementations of (projectable) choreographies, using techniques from [8] and [7]. Although these constructions are not technically challenging, we omit them for brevity, since they are immaterial for our results.

The fragment of DCC that does not contain label selections is called Dy-

---

[2]In some process calculi, the weaker condition $\mathsf{p} \xrightarrow{G} \mathsf{q}$ 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 [15]. This choice is orthogonal to our development.

$$\frac{\mathsf{p} \xleftrightarrow{G} \mathsf{q} \quad e[\sigma(\mathsf{p})/*] \downarrow v}{G, \mathsf{p}.e \rightarrow \mathsf{q}; C, \sigma \ \rightarrow \ G, C, \sigma[\mathsf{q} \mapsto v]} \ \lfloor\mathrm{C}|\mathrm{Com}\rfloor$$

$$\frac{\mathsf{p} \xleftrightarrow{G} \mathsf{q}}{G, \mathsf{p} \rightarrow \mathsf{q}[l]; C, \sigma \ \rightarrow \ G, C, \sigma} \ \lfloor\mathrm{C}|\mathrm{Sel}\rfloor$$

$$\frac{}{G, \mathsf{p \ starts \ q}; C, \sigma \ \rightarrow \ G \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\}, C, \sigma[\mathsf{q} \mapsto \bot]} \ \lfloor\mathrm{C}|\mathrm{Start}\rfloor$$

$$\frac{\mathsf{p} \xleftrightarrow{G} \mathsf{q} \quad \mathsf{p} \xrightarrow{G} \mathsf{r}}{G, \mathsf{p}.\mathsf{r} \rightarrow \mathsf{q}; C, \sigma \ \rightarrow \ G \cup \{\mathsf{q} \rightarrow \mathsf{r}\}, C, \sigma} \ \lfloor\mathrm{C}|\mathrm{Intro}\rfloor$$

$$\frac{i = 1 \text{ if } \sigma(\mathsf{p}) = \sigma(\mathsf{q}), \quad i = 2 \text{ otherwise}}{G, \text{if } \mathsf{p} \overset{\leftarrow}{=} \mathsf{q} \text{ then } C_1 \text{ else } C_2, \sigma \ \rightarrow \ G, C_i, \sigma} \ \lfloor\mathrm{C}|\mathrm{Cond}\rfloor$$

$$\frac{C_1 \preceq C_2 \quad G, C_2, \sigma \ \rightarrow \ G', C_2', \sigma' \quad C_2' \preceq C_1'}{G, C_1, \sigma \ \rightarrow \ G', C_1', \sigma'} \ \lfloor\mathrm{C}|\mathrm{Struct}\rfloor$$

Figure 5: Dynamic Core Choreographies, Semantics.

namic Minimal Choreographies (DMC). Amendment and label selection elimination hold for DMC and DCC just as for MC and CC, so that, for any DMC choreography C, $(\!|\mathsf{Amend}(C)|\!)$ is always projectable, and Theorem 2 also holds for these calculi (arrow (b) in Figure 1).

The encoding. We focus now on the mapping (1) from Figure 1, 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 $M_C :\ \mathcal{P}^2 \rightarrow \mathbb{N}$, where $\mathcal{P} = \mathsf{pn}(C)$ is the set of process names in C. Intuitively, $\{\!\{C\}\!\}$ will use a countable set of auxiliary processes $\{\mathsf{pq}^i \mid \mathsf{p}, \mathsf{q} \in \mathcal{P}, i \in \mathbb{N}\}$, where pq^i will hold the ith message from p to q.

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

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

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 6.

We write $\bar{M}$ for $\left\{\mathsf{pq}^M \mid \mathsf{p}, \mathsf{q} \in \mathcal{P}, \mathsf{p} \neq \mathsf{q}\right\}$, 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 $\mathsf{p}.e \rightarrow \mathsf{q}$, 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

$$\{\!\!\{\, \mathsf{p}.e \text{ -> } \mathsf{q}; C \,\}\!\!\}_M = \mathsf{p}.e \text{ -> } \mathsf{pq}^M; \ \mathsf{p\, start\, pq}^{M+}; \ \mathsf{p} : \mathsf{pq}^M \text{ <-> } \mathsf{pq}^{M+};$$
$$\mathsf{pq}^M.\mathsf{q} \text{->} \mathsf{pq}^{M+}; \ \mathsf{pq}^M.\mathsf{pq}^{M+} \text{->} \mathsf{q}; \ \mathsf{pq}^M. * \text{ -> } \mathsf{q};$$
$$\{\!\!\{\, C \,\}\!\!\}_{M[(\mathsf{p},\mathsf{q})\mapsto M(\mathsf{p},\mathsf{q})+1]}$$
$$\{\!\!\{\, \text{if } \mathsf{p} \overset{\Leftarrow}{=} \mathsf{q} \text{ then } C_1 \text{ else } C_2 \,\}\!\!\}_M = \mathsf{q}. * \text{ -> } \mathsf{qp}^M; \ \mathsf{q\, start\, qp}^{M+}; \ \mathsf{q} : \mathsf{qp}^M \text{ <-> } \mathsf{qp}^{M+};$$
$$\mathsf{qp}^M.\mathsf{p} \text{ -> } \mathsf{qp}^{M+}; \ \mathsf{qp}^M.\mathsf{qp}^{M+} \text{ -> } \mathsf{p};$$
$$\text{if } \mathsf{p} \overset{\Leftarrow}{=} \mathsf{qp}^M \text{ then } \{\!\!\{\, C_1 \,\}\!\!\}_{M[(\mathsf{p},\mathsf{q})\mapsto M(\mathsf{p},\mathsf{q})+1]}$$
$$\text{else } \{\!\!\{\, C_2 \,\}\!\!\}_{M[(\mathsf{p},\mathsf{q})\mapsto M(\mathsf{p},\mathsf{q})+1]}$$
$$\{\!\!\{\, \mathbf{0} \,\}\!\!\}_M = \mathbf{0}$$
$$\{\!\!\{\, X \,\}\!\!\}_M = X\langle \bar{M} \rangle$$
$$\{\!\!\{\, \text{def } X = C_2 \text{ in } C_1 \,\}\!\!\}_M = \text{def } X(\overline{M_0}) = \{\!\!\{\, C_2 \,\}\!\!\}_{M_0} \text{ in } \{\!\!\{\, C_1 \,\}\!\!\}_M$$

Figure 6: Encoding MC in DMC

be ready to receive both the value being communicated and the name of the process that will store the next value.

The behaviours of the choreographies C and $\{\!\!\{\, C \,\}\!\!\}$ are closely related, as formalised in the following theorems.

Theorem 3. *Let* $\mathsf{p} \in \mathsf{pn}(C)$ *and* $\mathsf{pq} \in \mathsf{pn}(\{\!\!\{\, C \,\}\!\!\}) \setminus \mathsf{pn}(C)$. *If* $G, \{\!\!\{\, C \,\}\!\!\}, \sigma \to^* G', C_1, \sigma_1 \to G', C_2, \sigma_2$ *where in the last transition a value* v *is sent from* p *to* pq, *then there exist* G'', C_3, σ_3, C_4 *and* σ_4 *such that* $G', C_2, \sigma_2 \to^* G'', C_3, \sigma_3 \to G'', C_4, \sigma_4$ *and in the last transition the same value* v *is sent from* pq *to some process* $\mathsf{q} \in \mathsf{pn}(C)$.

Theorem 3 states that messages sent from p to q are eventually received by q.

Theorem 4. *If* $G, \{\!\!\{\, C \,\}\!\!\}_M, \sigma \to^* G_1, C_1, \sigma_1$, *then there exist* G'', C', σ' *and* σ'' *such that* $C, \sigma \to^* C', \sigma'$, *and* $G_1, C_1, \sigma_1 \to^* G'', \{\!\!\{\, C' \,\}\!\!\}_M, \sigma''$, *and* σ' *and* σ'' *coincide on the values stored at* $\mathsf{pn}(C)$.

Theorem 4 states that the encoding does not add any additional behaviour to the original choreography, aside from expanding communications into several actions.

Example 2. *We partially show the result of applying this transformation to Lines 1–3 of Example 1. We only include the initializations of the channels that are used in this fragment; the numbers indicated refer to the line numbers*

9

in the original example.

$$a \text{ start } as^0; \ a : as^0 \ \text{<->} \ s;$$
$$s \text{ start } sa^0; \ s : sa^0 \ \text{<->} \ a;$$
$$s \text{ start } sb^0; \ s : sb^0 \ \text{<->} \ b;$$

1. $a.title \ \text{->} \ as^0; \ a \text{ start } as^1; \ a : as^1 \ \text{<->} \ as^0;$
$$as^0.as^1 \ \text{->} \ s; \ as^0.s \ \text{->} \ as^1; \ as^0. * \ \text{->} \ s;$$

2. $s.price \ \text{->} \ sa^0; \ s \text{ start } sa^1; \ s : sa^1 \ \text{<->} \ sa^0;$
$$sa^0.sa^1 \ \text{->} \ a; \ sa^0.a \ \text{->} \ sa^1; \ sa^0. * \ \text{->} \ a;$$

3. $s.price \ \text{->} \ sb^0; \ s \text{ start } sb^1; \ s : sb^1 \ \text{<->} \ sb^0;$
$$sb^0.sb^1 \ \text{->} \ b; \ sb^0.b \ \text{->} \ sb^1; \ sb^0. * \ \text{->} \ b;$$

$$\dots$$

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* s.price -> sa^0 *can swap with the three actions immediately preceding it.*

We briefly illustrate Theorems 3 and 4 in this setting. Theorem 3 states that, e.g., the action a.title -> as^0 *is eventually followed by a communication of title from* as^0 *to some other process in the original choreography (in this case,* s*). Theorem 4 implies that if, e.g.,* a.title -> as^0 *is executed, then it must be "part" of an action in the original choreography (in this case,* a.title -> 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).*

4 The General Case

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 1 – 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, $\{\!\!\{\mathsf{def}\,X(\mathsf{p},\mathsf{q}) = C_2\,\mathsf{in}\,C_1\}\!\!\}_M$ would be $\mathsf{def}\,X(\mathsf{p},\mathsf{q},\mathsf{pq}^0,\mathsf{qp}^0) = \{\!\!\{C_2\}\!\!\}_{M_0}\,\mathsf{in}\,\{\!\!\{C_1\}\!\!\}_M$. We will not write this definition formally.

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

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

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 $\mathsf{p.q\,\text{->}\,r}$ 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.

Theorems 3 and 4 still hold for this extended encoding.

Projections. Finally, we extend this encoding to the whole language of DCC – arrow (4) in Figure 1 – by adding the clause

$$\{\!\!\{\mathsf{p\,\text{->}\,q}[l]; C\}\!\!\}_M = \mathsf{p\,\text{->}\,pq}^M[l];\ \mathsf{p\,start\,pq}^{M+};$$
$$\mathsf{p}:\mathsf{pq}^M <\!\!-\!\!> \mathsf{pq}^{M+};\ \mathsf{pq}^M.\mathsf{q}\,\text{->}\,\mathsf{pq}^{M+};$$
$$\mathsf{pq}^M.\mathsf{pq}^{M+}\,\text{->}\,\mathsf{q};\ \mathsf{pq}^M\,\text{->}\,\mathsf{q}[l];$$
$$\{\!\!\{C\}\!\!\}_{M[(\mathsf{p},\mathsf{q})\mapsto M(\mathsf{p},\mathsf{q})+1]}$$

to the definition of $\{\!\!\{C\}\!\!\}_M$. Restricting this encoding to the language of CC yields arrow (3) in Figure 1.

We finish this section with a brief informal note on projectability. As we discussed in § 2, a formal presentation of projection for DMC and DCC is beyond the scope of this paper. However, we point out that our encoding for asynchronous communications preserves projectability, i.e., if C is projectable, then so is $\{\!\!\{C\}\!\!\}$.

5   Related Work

To the best of our knowledge, this is the first work 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 work recalls the development of the asynchronous π-calculus [12] (Aπ for short, using the terminology from [25]). 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.

Other studies have investigated asynchronous communications in choreographies. The distinctive feature of our work is that it does not rely on any ad-hoc syntax or semantics for capturing asynchrony. In [14], choreographies are used as types for communication protocols and are related to asynchronous communications by encoding choreographies in types for terms in a variant of the π-calculus. However, asynchrony can only be observed in the semantics of processes, not at the level of choreographies, and the syntax of processes is equipped with ad-hoc runtime terms[3] that represent messages in transit. The first work defining an asynchronous semantics for choreographies is [5], by defining an ad-hoc rule in the semantics of choreographies that allows nested (not appearing at the top level) communications to be executed if, among other conditions: the sender is the same as the one in the communication at the top level of the choreography, the receiver is not involved in the nested communication. This technique has been later adopted also in [22] – for defining the composition of asynchronous choreographies with legacy process code – and in [15] (the journal version of [14]) – to formulate a semantics for communication protocols represented as choreographies. In [11], choreographies (not processes, for example as in [14]) are equipped with runtime terms to represent messages in transit.

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 stud-

---

[3]Runtime terms are assumed never to be used by the programmer, but only produced as the result of execution.

ied also in the works [4, 5, 22], 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 [7], which makes all results from that work applicable to DCC (and thus DMC).

6 Conclusions

Choreographies are widely used in the context of concurrent and distributed software architectures, in order to specify precisely how the different components of a system should interact [2, 26]. Previous formalisations of asynchronous communications in choreographies exchange expressivity for simplicity, yielding ad-hoc models with unclear connections. In this work, we showed that a choreography calculus with process spawning and process name mobility can capture asynchronous communications. Therefore, all such calculi with similar primitives have the same power. Our development is conservative wrt previous work, allowing us to import existing techniques developed for previous calculi. For example, the techniques shown in [7, 8] could be reapplied to DCC to synthesise deadlock-free process implementations. Here, we showed how to import the result of selection elimination from [8]. In conclusion, we now have a setting where we can reason about asynchronous communications in choreographies by considering a simple synchronous semantics, just like it can be done in the seminal model of π-calculus for mobile processes.

References

[1] S. Basu and T. Bultan. Choreography conformance via synchronizability. In *WWW*, pages 795–804, 2011.

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

[3] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *COORDINATION*, volume 4038 of *LNCS*, pages 63–81. Springer, 2006.

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

[5] M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 263–274. ACM, 2013.

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

[7] L. Cruz-Filipe and F. Montesi. A language for the declarative composition of concurrent protocols. *CoRR*, abs/1602.03729, 2016. Submitted for publication.

[8] L. Cruz-Filipe and F. Montesi. A core model for choreographic programming. In *FACS*, LNCS. Springer, Accepted for publication. Available at `http://arxiv.org/abs/1510.03271`.

[9] M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic choreographies - safe runtime updates of distributed applications. In T. Holvoet and M. Viroli, editors, *COORDINATION*, volume 9037 of *LNCS*, pages 67–82. Springer, 2015.

[10] M. Dalla Preda, S. Giallorenzo, I. Lanese, J. Mauro, and M. Gabbrielli. AIOCJ: A choreographic framework for safe adaptive distributed applications. In B. Combemale, D. Pearce, O. Barais, and J. Vinju, editors, *SLE*, volume 8706 of *LNCS*, pages 161–170. Springer, 2014.

[11] P.-M. Deniélou and N. Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In F. Fomin, R. Freivalds, M. Kwiatkowska, and D. Peleg, editors, *ICALP (II)*, volume 7966 of *LNCS*, pages 174–186. Springer, 2013.

[12] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP*, volume 512 of *LNCS*, pages 133–147, Berlin, Heidelberg, New York, Tokyo, 1991. Springer.

[13] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In C. Hankin, editor, *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[14] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In G. C. Necula and P. Wadler, editors, *POPL*, pages 273–284. ACM, 2008.

[15] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016.

[16] I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In A. Cerone and S. Gruner, editors, *SEFM*, pages 323–332. IEEE, 2008.

[17] T. Leesatapornwongsa, J. Lukman, S. Lu, and H. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS*, pages 517–530. ACM, 2016.

[18] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *ACM SIGARCH Computer Architecture News*, 36(1):329–339, 2008.

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

[20] F. Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. `http://fabriziomontesi.com/files/choreographic\_programming.pdf`.

[21] F. Montesi. Kickstarting choreographic programming. In T. Hildebrandt, A. Ravara, J. van der Werf, and M. Weidlich, editors, *WS-FM*, volume 9421 of *LNCS*, pages 3–10. Springer, 2016.

[22] F. Montesi and N. Yoshida. Compositional choreographies. In P. D'Argenio and H. Melgratti, editors, *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.

[23] D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP*, pages 316–332, 2009.

[24] Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In C. Williamson, M. Zurko, P. Patel-Schneider, and P. Shenoy, editors, *WWW*, pages 973–982. ACM, 2007.

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

[26] 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.

[27] N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble protocol language. In *TGC*, pages 22–41, 2013.