# Encoding Asynchrony in Choreographies

Luís Cruz-Filipe and Fabrizio Montesi*
Dept. Mathematics and Computer Science, University of Southern Denmark
Campusvej 55, 5230 Odense M, Denmark
{lcf,fmontesi}@imada.sdu.dk

## ABSTRACT

Choreographies are widely used both for the specification and the programming of concurrent and distributed software architectures. Since many of such architectures use asynchronous communications, it is essential to understand how the behaviour described in a choreography can be correctly implemented in asynchronous settings. So far, this problem has been addressed by relying on additional technical machinery, such as ad-hoc syntactic terms, semantics, or equivalences. In this work, we show that such extensions are not needed for choreography languages that support primitives for process spawning and name mobility. Instead, we can just encode asynchronous communications in choreographies themselves, yielding a simpler approach.

## CCS Concepts

•**Theory of computation** → *Process calculi;*

## Keywords

Asynchrony; Choreography; Concurrency

## 1. INTRODUCTION

Programming concurrent and distributed 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 [7, 8] reveal that most errors are due to: deadlocks; violations of atomicity intentions; or, violations of ordering intentions. The issue is particularly pressing in architectures where hundreds of components may interact via message passing, like microservices [5]. To mitigate this problem, *choreographies* can be used as high-level formal specifications of the intended interactions among components [1, 2].

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. s.*book* -> a;
6. else **0**

*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,* s *sends the book to* a *(Line 5). Otherwise, the choreography terminates.*

In addition to their clarity, choreographies enable new development methodologies. For example, in Choreographic Programming [9, 10], choreographies are compiled to compliant local implementations for the described components. In our example, the implementation inferred for 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.

In most software architectures, communications are asynchronous. Therefore, it is important to prove that the code generated by compiling a choreography implements it correctly under in such a setting. So far, such proofs have been developed by defining ad-hoc extensions to the syntax and semantics of the models used to represent choreographies or their compiled code [2, 6, 11].

In this paper, we show that choreography languages equipped with primitives for process spawning and name mobility are already powerful enough to capture asynchronous communications. The key idea 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 [12]. We present our result (sketch) as an endo-encoding in the new language of Dynamic Minimal Choreographies (DMC), an extension of the representative choreography calculus of Minimal Choreographies [4].

*Montesi was supported by CRC (Choreographies for Reliable and efficient Communication software), grant no. DFF–4005-00304 from the Danish Council for Independent Research.

*SAC'17, April 3-7, 2017, Marrakesh, Morocco*

http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx

$$C ::= \mathbf{0} \mid \eta; C \mid \mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{def}\ X(\tilde{\mathsf{p}}) = C_2\ \mathsf{in}\ C_1 \mid X\langle\tilde{\mathsf{p}}\rangle \qquad \eta ::= \mathsf{p}.e\ \texttt{->}\ \mathsf{q} \mid \mathsf{p.r}\ \texttt{->}\ \mathsf{q} \mid \mathsf{p}\ \mathsf{starts}\ \mathsf{q}$$

$$\frac{\mathsf{p} \xleftrightarrow{G} \mathsf{q} \qquad e[\sigma(\mathsf{p})/*] \downarrow v}{G, \mathsf{p}.e\ \texttt{->}\ \mathsf{q}; C, \sigma \ \to\ G, C, \sigma[\mathsf{q} \mapsto v]}\ \lfloor\mathrm{C|Com}\rceil \qquad \frac{i = 1 \text{ if } \sigma(\mathsf{p}) = \sigma(\mathsf{q}), \quad i = 2 \text{ otherwise}}{G, \mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \sigma \ \to\ G, C_i, \sigma}\ \lfloor\mathrm{C|Cond}\rceil$$

$$\frac{}{G, \mathsf{p}\ \mathsf{starts}\ \mathsf{q}; C, \sigma \ \to\ G \cup \{\mathsf{p} \leftrightarrow \mathsf{q}\}, C, \sigma[\mathsf{q} \mapsto \bot]}\ \lfloor\mathrm{C|Start}\rceil \qquad \frac{\mathsf{p} \xleftrightarrow{G} \mathsf{q} \qquad \mathsf{p} \overset{G}{\to} \mathsf{r}}{G, \mathsf{p.r}\ \texttt{->}\ \mathsf{q}; C, \sigma \ \to\ G \cup \{\mathsf{q} \to \mathsf{r}\}, C, \sigma}\ \lfloor\mathrm{C|Intro}\rceil$$

**Figure 1: Dynamic Minimal Choreographies, Syntax and Semantics.**

## 2. LANGUAGE MODEL

We introduce Dynamic Minimal Choreographies (DMC), an extension of the calculus from [4].

The syntax of DMC is given in Figure 1 (top), 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 that can be read with the expression $*$. Term $\eta; C$ is an interaction between two processes, read "the system may execute $\eta$ and proceed as $C$". In a value communication $\mathsf{p}.e\ \texttt{->}\ \mathsf{q}$, $\mathsf{p}$ sends its local evaluation of expression $e$ (whose syntax we leave undefined) to $\mathsf{q}$, which stores the received value. In term $\mathsf{p}\ \mathsf{starts}\ \mathsf{q}$, process $\mathsf{p}$ starts a new process $\mathsf{q}$, whose name is known only by $\mathsf{p}$. Names can be communicated via term $\mathsf{p.r}\ \texttt{->}\ \mathsf{q}$. In a conditional $\mathsf{if}\ \mathsf{p} \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, $\mathsf{q}$ sends its value to $\mathsf{p}$, 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 all these actions, the two interacting processes must be different. Definitions and invocations of (parametric) recursive procedures ($X$) are standard. The term $\mathbf{0}$ is the terminated choreography.

In the semantics of DMC, we use a graph of connections $G$ [3], keeping track of which pairs of processes are allowed to communicate. This graph is directed, and an edge from $\mathsf{p}$ to $\mathsf{q}$ in $G$ (written $\mathsf{p} \overset{G}{\to} \mathsf{q}$) means that $\mathsf{p}$ knows the name of $\mathsf{q}$. In order for an actual message to flow between $\mathsf{p}$ and $\mathsf{q}$, both processes need to know each other, which we write as $\mathsf{p} \xleftrightarrow{G} \mathsf{q}$. The semantics for DMC uses reductions of the form $G, C, \sigma \to G', C', \sigma'$, where $G$ and $G'$ are the connection graphs before and after executing $C$, respectively, and the total state function $\sigma$ maps each process name to its value (values are denoted $v$, $w$, ...). The complete rules are given in Figure 1 (bottom), closed under a structural precongruence that allows for unfolding of procedure calls, garbage collection, and swapping of independent actions (see [4]).

In the premise of $\lfloor\mathrm{C|Com}\rceil$, $e[\sigma(\mathsf{p})/*]$ denotes replacing $*$ with $\sigma(\mathsf{p})$ in $e$. In the reductum, $\sigma[\mathsf{q} \mapsto v]$ denotes the updated state function $\sigma$ where $\mathsf{q}$ now maps to $v$. In $\lfloor\mathrm{C|Start}\rceil$, the fresh process $\mathsf{q}$ is assigned a default value $\bot$. We write $G \cup G'$ for the graph obtained by merging $G$ with $G'$.

The main limitation of DMC is that its semantics is synchronous. Indeed, in a real-world scenario implementation of Example 1, we would expect $\mathsf{s}$ to proceed immediately to sending its message in Line 3 after having sent the one in Line 2, without waiting for $\mathsf{a}$ 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 DMC

The calculus of Minimal Choreographies (MC) from [4] is the fragment of DMC that does not include process spawning and name mobility. In this fragment, we can omit procedure parameters by assuming that all procedures take all processes as arguments. In this section, we focus on MC and show that any MC choreography can be encoded in DMC 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 $\mathsf{p}.e\ \texttt{->}\ \mathsf{q} \in C \in \mathrm{MC}$ becomes split into a send/receive pair in $\{\!\!\{C\}\!\!\} \in \mathrm{DMC}$, with the properties that: $\mathsf{p}$ can continue executing without waiting for $\mathsf{q}$ to receive its message (and even send further messages to $\mathsf{q}$); and messages from $\mathsf{p}$ to $\mathsf{q}$ are delivered in the same order as they were originally sent.

Let $C$ be a choreography in MC. In order to encode $C$ in DMC, we use a function $M : \mathcal{P}^2 \to \mathbb{N}$, where $\mathcal{P} = \mathsf{pn}(C)$ is the set of process names in $C$. Intuitively, $\{\!\!\{C\}\!\!\}$ use a countable set of auxiliary processes $\{\mathsf{pq}^i \mid \mathsf{p}, \mathsf{q} \in \mathcal{P}, i \in \mathbb{N}\}$, where $\mathsf{pq}^i$ holds the $i$th message from $\mathsf{p}$ to $\mathsf{q}$.

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

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

Here, $M_0(\mathsf{p}, \mathsf{q}) = 0$ for all $\mathsf{p}$ and $\mathsf{q}$. For simplicity, we write $\mathsf{pq}^M$ for $\mathsf{pq}^{M(\mathsf{p},\mathsf{q})}$ and $\mathsf{pq}^{M+}$ for $\mathsf{pq}^{M(\mathsf{p},\mathsf{q})+1}$. The definition of $\{\!\!\{C\}\!\!\}_M$ is given in Figure 2.

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

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

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

**Figure 2: Encoding MC in DMC.**

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

Theorem 1. *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* $\mathsf{p}$ *to* $\mathsf{pq}$*, then there exist* $G'', C_3, \sigma_3, C_4$ *and* $\sigma_4$ *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* $\mathsf{pq}$ *to some process* $\mathsf{q} \in \mathsf{pn}(C)$.

Theorem 2. *If* $G, \{\!\!\{C\}\!\!\}_M, \sigma \to^* G_1, C_1, \sigma_1$*, then there exist* $C'$, $\sigma'$ *and* $\sigma''$ *such that* $G, C, \sigma \to^* G, C', \sigma'$*, and* $G_1, C_1, \sigma_1 \to^* G', \{\!\!\{C'\}\!\!\}_M, \sigma''$*, and* $\sigma'$ *and* $\sigma''$ *coincide on the values stored at* $\mathsf{pn}(C)$.

Example 2. *We show the result of applying this transformation to Lines 1–3 of Example 1. The numbers refer to the line numbers in the original example.*

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

*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 an intermediary (*$\mathsf{as}^0$*,* $\mathsf{sa}^0$ *or* $\mathsf{sb}^0$*, respectively), who eventually delivers it to the recipient. Moreover, causal dependencies are kept: in Step 2,* s *can only send its message to* $\mathsf{sa}^0$ *after receiving the message sent by* a *in Step 1. However, in Step 3* s *can send its message to* $\mathsf{sb}^0$ *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.*

*We briefly illustrate Theorems 1 and 2 in this example. Theorem 1 guarantees that the action* $\mathsf{a}.\mathit{title}\ \texttt{->}\ \mathsf{as}^0$ *is eventually followed by a communication of title from* $\mathsf{as}^0$ *to some other process in the original choreography (in this case,* s*). Theorem 2 implies that if* $\mathsf{a}.\mathit{title}\ \texttt{->}\ \mathsf{as}^0$ *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).*

Our construction can be extended to the whole language of DMC, but we omit this for space constraints.

# 4. REFERENCES

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

[2] 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.

[3] L. Cruz-Filipe and F. Montesi. A language for the declarative composition of concurrent protocols. Submitted for publication.

[4] L. Cruz-Filipe and F. Montesi. A core model for choreographic programming. Accepted for publication at FACS'16. http://arxiv.org/abs/1510.03271.

[5] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. *CoRR*, abs/1606.04036, 2016.

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

[7] 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.

[8] 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.

[9] F. Montesi. *Choreographic Programming.* Ph.D. thesis, IT University of Copenhagen, 2013. http://fabriziomontesi.com/files/choreographic_programming.pdf.

[10] F. Montesi. Kickstarting choreographic programming. In *WS-FM*, volume 9421 of *LNCS*, pages 3–10. Springer, 2016.

[11] F. Montesi and N. Yoshida. Compositional choreographies. In *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.

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