Lecture Notes on Choreographies, Part 2

Fabrizio Montesi
fmontesi@imada.sdu.dk

November 6, 2017

Abstract

This document contains lecture notes for the course on Concurrency Theory (2017) at the University of Southern Denmark.

1 Towards a correct EPP

In part 1, we defined a simple choreography language and its EPP towards a simple process calculus. However, this framework does not support a key desirable property: the EPP of a choreography should only do what the original choreography prescribes.

The counterexample is simple. Take the following choreography:

$$C_{\mathsf{problem}} \triangleq \mathsf{p} \mathbin{\texttt{->}} \mathsf{q}; \mathsf{r} \mathbin{\texttt{->}} \mathsf{s}; \mathbf{0}$$

. The EPP of this choreography is:

$$[\![C_{\mathsf{problem}}]\!] = \mathsf{p} \triangleright \mathsf{q}!; \mathbf{0} \mid \mathsf{q} \triangleright \mathsf{p}?; \mathbf{0} \mid \mathsf{r} \triangleright \mathsf{s}!; \mathbf{0} \mid \mathsf{s} \triangleright \mathsf{r}?; \mathbf{0}$$

. We have the following reduction for this network, by synchronising r with s:

$$\mathsf{p} \triangleright \mathsf{q}!; \mathbf{0} \mid \mathsf{q} \triangleright \mathsf{p}?; \mathbf{0} \mid \mathsf{r} \triangleright \mathsf{s}!; \mathbf{0} \mid \mathsf{s} \triangleright \mathsf{r}?; \mathbf{0} \quad \to \quad \mathsf{p} \triangleright \mathsf{q}!; \mathbf{0} \mid \mathsf{q} \triangleright \mathsf{p}?; \mathbf{0}$$

. However, this reduction cannot be mimicked by C_{problem}, which can only reduce the first interaction between p and q according to the semantics given in the previous lecture notes.

The example above shows that our framework is not sound yet, because the projection of a choreography can perform "extra" reductions with respect to the choreography. There are two ways to fix this.

Forbidding independent sequences of interactions One way is to say that choreographies like C_{problem} are "forbidden", because the sequence of interactions $\mathsf{p} \texttt{->} \mathsf{q}; \mathsf{r} \texttt{->} \mathsf{s}$ is not enforced by any causality relation between the two interactions. More specifically, since the processes p, q, r, and s are all different, they operate independently—as the reduction for the projection of C_{problem} shows. However, if the process names of the two interactions intersected in any way, this problem would not appear. For example, consider the choreography $\mathsf{p} \texttt{->} \mathsf{q}; \mathsf{r} \texttt{->} \mathsf{q}; \mathbf{0}$. Its projection reduces as expected by the choreography because process q necessarily needs to complete the first interaction before participating in the second.

Exercise 1. *Check that the EPP of* $\mathsf{p} \texttt{->} \mathsf{q}; \mathsf{r} \texttt{->} \mathsf{q}; \mathbf{0}$ *reduces as expected by the choreography (i.e., their respective reduction chains match).*

Detecting sequences of interactions that do not have causal dependencies can be done mechanically. Thus, it is possible to automatically detect whether a choreography respects the condition of not having sequences of independent interactions, as in C_{problem}. Forbidding programmers to write choreographies like C_{problem} was a popular approach (and still is in some works, when useful) in the first studies on choreographies, like [Fu et al., 2005; Qiu et al., 2007; Carbone et al., 2007].

Out of order execution The other way to solve our issue with sequences of independent interactions is to extend the semantics of choreographies to correctly capture the parallel nature of processes. Going back to C_{problem} again, if we could somehow design a semantics that allowed for the following reduction

$$\mathsf{p} \texttt{->} \mathsf{q}; \mathsf{r} \texttt{->} \mathsf{s}; \mathbf{0} \quad \to \quad \mathsf{p} \texttt{->} \mathsf{q}; \mathbf{0}$$

then we would be fine, because that is the reduction that we need to match the problematic one that the EPP of the choreography can do. Observe that this reduction does not respect the order in which instructions are given in the choreography. This kind of semantics is typically called "out-of-order execution". The idea is widespread in many domains. For example, modern CPUs and/or language runtimes may change the order in which instructions are executed to increase performance, when it is safe to do so—a typical example is the single-threaded imperative code `x++; y++;`, where the order in which the two increments are done is ininfluential and the runtime may thus decide to parallelise it.

Since the inception of out-of-order execution for choreographies [Carbone and Montesi, 2013], similar ideas have been adopted in different works [Deniélou and Yoshida, 2013; Honda et al., 2016]. In the next section, where

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

Figure 1: Simple choreographies, syntax.

$$\frac{}{\mathsf{p} \rightarrow \mathsf{q}; C \to C}\ \textsc{Com} \qquad \frac{C \preceq C_1 \quad C_1 \to C_2 \quad C_2 \preceq C'}{C \to C'}\ \textsc{Struct}$$

Figure 2: Simple choreographies, semantics.

we develop the technical details, we borrow the formulation by Cruz-Filipe and Montesi [2016].

2 Concurrent Simple Choreographies

We update our model of simple choreographies to capture concurrent execution of different processes.

The syntax of choreographies remains unchanged. It is displayed in fig. 1.

The semantics of choreographies, instead, needs some updating to capture out-of-order execution of interactions. We obtain this by adding a structural precongruence $\preceq$ for choreographies[1]. The reduction rules are given in fig. 2. The only change is the addition of rule Struct, which closes reductions under $\preceq$. There is only one rule defining $\preceq$, which is displayed in fig. 3.

Recall that $C \equiv C'$ stands for $C \preceq C'$ and $C' \preceq C$. (We will see rules that use $\preceq$ in only one direction later on, so it is still useful to explicitly formulate these rules using precongruences.) Rule Swap states that two interactions $\mathsf{p} \rightarrow \mathsf{q}$ and $\mathsf{r} \rightarrow \mathsf{s}$ can be exchanged in a choreography if the processes p, q, r, and s are distinct (they are all different). This is captured by the premise $\{\mathsf{p}, \mathsf{q}\} \# \{\mathsf{r}, \mathsf{s}\}$, which states that the sets $\{\mathsf{p}, \mathsf{q}\}$ and $\{\mathsf{r}, \mathsf{s}\}$ must be disjoint. Formally, given two sets S and S', $S \# S'$ is a shortcut notation for $S \cap S' = \emptyset$ (empty intersection).

Example 1. *This is the reduction we wished we could do in section 1.*

$$\mathsf{p} \rightarrow \mathsf{q}; \mathsf{r} \rightarrow \mathsf{s}; \mathbf{0} \quad \to \quad \mathsf{p} \rightarrow \mathsf{q}; \mathbf{0}$$

[1]We use the same symbol as for the structural precongruence for networks, since we can easily distinguish them from the context (they operate on different domains, one choreographies and the other networks).

$$\frac{\{\mathsf{p},\mathsf{q}\}\,\#\{\mathsf{r},\mathsf{s}\}}{\mathsf{p}\,\text{->}\,\mathsf{q};\mathsf{r}\,\text{->}\,\mathsf{s} \quad\equiv\quad \mathsf{r}\,\text{->}\,\mathsf{s};\mathsf{p}\,\text{->}\,\mathsf{q}}\ \textsc{Swap}$$

Figure 3: Simple choreographies, structural precongruence.

. *We can now perform it with our new semantics. Here is the derivation:*

$$\frac{\begin{array}{c}\mathsf{p}\,\text{->}\,\mathsf{q};\\ \mathsf{r}\,\text{->}\,\mathsf{s};\mathbf{0}\end{array} \preceq \begin{array}{c}\mathsf{r}\,\text{->}\,\mathsf{s};\\ \mathsf{p}\,\text{->}\,\mathsf{q};\mathbf{0}\end{array} \qquad \dfrac{}{\begin{array}{c}\mathsf{r}\,\text{->}\,\mathsf{s};\\ \mathsf{p}\,\text{->}\,\mathsf{q};\mathbf{0}\end{array} \to \mathsf{p}\,\text{->}\,\mathsf{q};\mathbf{0}}\ \textsc{Com} \qquad \mathsf{p}\,\text{->}\,\mathsf{q};\mathbf{0} \preceq \mathsf{p}\,\text{->}\,\mathsf{q};\mathbf{0}}{\mathsf{p}\,\text{->}\,\mathsf{q};\mathsf{r}\,\text{->}\,\mathsf{s};\mathbf{0} \quad\to\quad \mathsf{p}\,\text{->}\,\mathsf{q};\mathbf{0}}\ \textsc{Struct}$$

.

Exercise 2. *Prove the following statement.*
Let $[\![C]\!] = N$. If $C \preceq C'$ for some C', then $N \preceq [\![C']\!]$.

Exercise 3 (!)**.** *Prove the following statement.*
Let $[\![C]\!] = N$. If $N \to N'$ for some N', then there exists C' such that $C \to C'$ and $N' \preceq [\![C']\!]$.

Suggestion: *proceed by structural induction on C, and then by cases on the possible reductions $N \to N'$.*

Exercise 4. *Prove the following statement.*
Let $[\![C]\!] = N$. If $C \to C'$ for some C', then there exists N' such that $N \to N'$ and $N' \preceq [\![C]\!]$.

We can now formally state that EPP is correct, in the sense that the behaviour implemented by the network projected from a choreography is exactly the one defined in the choreography.

Theorem 1 (Operational Correspondence)**.** *Let $[\![C]\!] = N$. Then,*

Completeness *If $C \to C'$ for some C', then there exists N' such that $N \to N'$ and $N' \preceq [\![C']\!]$.*

Soundness *If $N \to N'$ for some N', then there exists C' such that $C \to C'$ and $N' \preceq [\![C']\!]$.*

Intuitively, the completeness part means that the network generated by the EPP of a choreography does *all* that the choreography says. Conversely, the soundness part means that the network generated by the EPP of a choreography does *only* what the choreography says. The two parts combined give us correctness: the network generated by EPP does exactly what is defined in the originating choreography. So what happens is what we want, finally!

The correctness of EPP gives us a powerful result for free: the EPP of a choreography never gets stuck (for example, there cannot be deadlocks). Intuitively, this works because interactions in choreographies are written atomically, in the sense that they specify both the send and receive actions needed for the communication in a single term. To have a deadlock, one typically needs a language where send and receive actions are defined separately (hence the opportunity for mistakes).

Let us formalise this result. First, we observe that choreographies never get stuck.

Theorem 2 (Progress for choreographies)**.** *Let C be a choreography. Then, either $C = \mathbf{0}$ (C has terminated) or there exists C' such that $C \to C'$.*

Proof. By cases on C. If $C = \mathbf{0}$, the thesis follows immediately. Otherwise, we can just apply rule COM. □

We now combine theorem 2 with the completeness part of theorem 1, which respectively say that "a choreography can always reduce until it terminates" and "the EPP of a choreography can always do what the choreography does". This means that "the EPP of a choreography can always reduce until it terminates", as formalised below.

Corollary 1 (Progress for EPP)**.** *Let $[\![C]\!] = N$. Then, either $N = \mathbf{0}$ (N has terminated) or there exists N' such that $N \to N'$.*

Proof. Direct consequence of theorems 1 and 2. □

3 Local Computation

Now that we know the basic soundness principles of choreographies and EPP, we can play with extending our model such that we can capture more interesting—and realistic—examples.

In this section, we equip processes with the ability to perform local computation. This will enable us to capture our initial scenario from Part 1 precisely, i.e., we want to define the *content* of the messages exchanged by Buyer and Seller.

3.1 Stateful Choreographies

Syntax We augment the syntax of choreographies with functions—ranged over by f, g, ...—which can be used by processes to perform local computation. The new syntax is given in fig. 4.

$$
\begin{aligned}
C &::= \eta; C \mid \mathbf{0} \\
\eta &::= \mathsf{p}.f \;\texttt{->}\; \mathsf{q}.g \mid \mathsf{p}.f
\end{aligned}
$$

Figure 4: Stateful choreographies, syntax.

The key idea is that now processes are equipped with their own local memories, which they can manipulate through computation. The new term $\mathsf{p}.f$ reads "process p stores the result of function f in its memory". The new communication term $\mathsf{p}.f \;\texttt{->}\; \mathsf{q}.g; C$ reads "process p sends the result of computing function f to q; q then computes function g according to the received message and its local memory, and updates its memory with the result".

It will be convenient to reason about the process names used in choreographic statements—ranged over by η. We thus update the definition of procs as follows.

$$
\begin{aligned}
\mathsf{procs}\,(\eta; C) &= \mathsf{procs}(\eta) \cup \mathsf{procs}(C) \\
\mathsf{procs}(\mathbf{0}) &= \emptyset \\
\mathsf{procs}\,(\mathsf{p}.f \;\texttt{->}\; \mathsf{q}.g) &= \{\mathsf{p}, \mathsf{q}\} \\
\mathsf{procs}\,(\mathsf{p}.f) &= \{\mathsf{p}\}
\end{aligned}
$$

Semantics Now that processes have functions that may refer to their memories, the execution of a choreography depends on the state of process memories. To formalise this, we need to formulate reductions on more than just choreographies, but rather pairs of choreographies and memory states.

It is convenient to abstract from how process memory is concretely implemented, since different processes may use different kinds of data structures. Let v, u, ... range over memory states (or values), which we leave unspecified. Also, let σ range over global memory states, mapping process names to values. Intuitively, σ maps each process to its memory state. For example, $\sigma(\mathsf{p}) = v$ means that process p has v as memory. We assume that σs are total functions (they are never undefined).

We define a semantics for stateful choreographies in terms of reductions $\langle C, \sigma\rangle \to \langle C', \sigma'\rangle$, where $\langle C, \sigma\rangle$ is a *runtime configuration.* The rules defining $\to$ are given in fig. 5. It is based as usual on a structural precongruence $\preceq$, defined by the rule in fig. 6.

Let us look at rule LOCAL first, which gives a semantics for local computations. The premise uses the evaluation operator $\downarrow$, which we leave unspecified.

$$\frac{f(\sigma(\mathsf{p}))\downarrow v \quad g(\sigma(\mathsf{q}),v)\downarrow u}{\langle \mathsf{p}.f \texttt{->} \mathsf{q}.g; C, \sigma\rangle \to \langle C, \sigma[\mathsf{q}\mapsto u]\rangle}\ \textsc{Com} \qquad \frac{f(\sigma(\mathsf{p}))\downarrow v}{\langle \mathsf{p}.f; C, \sigma\rangle \to \langle C, \sigma[\mathsf{p}\mapsto v]\rangle}\ \textsc{Local}$$

$$\frac{C \preceq C_1 \quad \langle C_1,\sigma\rangle \to \langle C_2,\sigma'\rangle \quad C_2 \preceq C'}{\langle C,\sigma\rangle \to \langle C',\sigma'\rangle}\ \textsc{Struct}$$

Figure 5: Stateful choreographies, semantics.

$$\frac{\mathsf{procs}(\eta)\ \#\ \mathsf{procs}(\eta')}{\eta;\eta' \quad\equiv\quad \eta';\eta}\ \textsc{Swap}$$

Figure 6: Stateful choreographies, structural precongruence.

More specifically, we write $f(v_1,\ldots,v_n)\downarrow u$ when the evaluation of function f—where its parameters are instantiated with the values $v_1,\ldots,v_n$—returns the value u. In rule LOCAL, we pass the current memory state of p—$\sigma(\mathsf{p})$—to f and get a value v, which then becomes the new state for process p. The notation $\sigma[\mathsf{p}\mapsto v]$ is a mapping update and means "σ, but where p is now mapped to v". Formally:

$$(\sigma[\mathsf{p}\mapsto v])\,(\mathsf{q}) = \begin{cases} v & \text{if } \mathsf{q}=\mathsf{p} \\ \sigma(\mathsf{q}) & \text{otherwise} \end{cases}$$

.

Rule COM is the natural extension of interactions to include internal computation. In the first premise, we evaluate the function f used by the sender p under its memory state, getting a value v. Then, we evaluate the function g used by the receiver under the memory state of the receiver and the value v (the message received from p), obtaining a value u. The memory of the receiver q is then updated to become this value.

We assume that evaluating a function always terminates. In practice, this means that evaluation may yield error values (like empty values), and that infininte computations may be interrupted by timeouts. We abstract from such details, since what we are interested in here is communications, not the algorithmic details of internal computation. It is easy to plug in existing techniques for ensuring that the parameters passed to local functions are always of the right type, as shown in [Cruz-Filipe and Montesi, 2017].

Rule SWAP is updated to deal with all kinds of choreographic statements, be they interactions or internal computations.

Modelling variables Mainstream programming languages typically programmers to manipulate different *variables* whose values reside in memory. Let x, y, z, ... range over variable names (variables for short). A variable mapping h is a total function that maps variables to values. From now on, we adopt the following shortcut notation[2]: $\mathsf{p}.f \text{ -> } \mathsf{q}.x$ stands for $\mathsf{p}.f \text{ -> } \mathsf{q}.set^x$, where set^x is the function that replaces x in the variable mapping of q with the value received from p. Formally, given x, the evaluation of set^x is defined as:

$$set^x(h, v) \downarrow h[x \mapsto v]$$

.

Exercise 5. *Prove the following statement.*

For all $\langle \mathsf{p}.f \text{ -> } \mathsf{q}.x; C, \sigma \rangle$ *such that* $\sigma(\mathsf{q}) = h$ *and* h *is a variable mapping, we have that*

$$\langle \mathsf{p}.f \text{ -> } \mathsf{q}.x; C, \sigma \rangle \quad \to \quad \langle C, \sigma[\mathsf{q} \mapsto h'] \rangle$$

where $f(\sigma(\mathsf{p})) = v$ *and* $h' = h[x \mapsto v]$.

Conversely, it is useful to have a shortcut notation for *sending* the content of a variable. Thus, from now on, we adopt also another shortcut notation: $\mathsf{p}.x \text{ -> } \mathsf{q}.g$ stands for $\mathsf{p}.get^x \text{ -> } \mathsf{q}.g$, where get^x is the function that returns the value of variable x from the variable mapping of p. Formally, given x, the evaluation of get^x is defined as:

$$get^x(h) \downarrow h(x)$$

.

Example 2. *We can finally give a precise choreography for our example introduced in Part 1, including computation and message contents as well. Recall the informal description of the example:*

1. Buyer *sends the title of a book she wishes to buy to* Seller*;*
2. Seller *replies to* Buyer *with the price of the book.*

A corresponding choreography that defines this behaviour is:

$$\mathsf{Buyer}.title \text{ -> } \mathsf{Seller}.x; \mathsf{Seller}.cat(x) \text{ -> } \mathsf{Buyer}.price; \mathbf{0}$$

where title is a variable, cat is a function (cat stands for catalogue, if you like) that given a book title returns the price for it, and price is a variable.

[2]Shortcut notations for programming languages are sometimes called syntactic sugar.

$$
\begin{aligned}
N &::= \mathsf{p} \triangleright_v B \mid N \,|\, N \mid \mathbf{0} \\
B &::= \mathsf{p}!f; B \mid \mathsf{p}?f; B \mid f; B \mid \mathbf{0}
\end{aligned}
$$

Figure 7: Stateful processes, syntax.

Exercise 6. *Let σ be such that $\sigma(\mathsf{p}) = h_{\mathsf{p}}$ and $\sigma(\mathsf{q}) = h_{\mathsf{q}}$ for some variable mappings h_{p} and h_{q}. Also, let $h_{\mathsf{p}}(title) =$ "Flowers for Algernon" and cat be a function such that cat("Flowers for Algernon") $= 100$. Show the reduction chain for the choreography in the previous example:*

$$\mathsf{Buyer}.title \text{ -> } \mathsf{Seller}.x; \mathsf{Seller}.cat(x) \text{ -> } \mathsf{Buyer}.price; \mathbf{0}$$

.

3.2 Stateful Processes

Since we updated our choreography model, we also need to update our process model to describe the implementations of choreographies. This is a straightforward extension of our previous calculus of simple processes, obtained by adding memories to processes. The new syntax is given in fig. 7.

A process term $\mathsf{p} \triangleright_v B$ now holds a value v, representing the memory state of the process. Send and receive actions are now extended to applying functions. A send action $\mathsf{p}!f$ sends the result of computing f in the local state of the process. Conversely, a receive action $\mathsf{p}?f$ computes f by using the value received from the other process and the local process memory, and then stores the result in the local process memory. An action f executes function f and updates the local memory of the process according to the result.

The semantics of stateful processes is also a straightforward extension, which uses the same evaluation function used for choreographies. The rules are given in fig. 8. The rules for the structural precongruence are the same (modulo the addition of values v in process terms, but they are ininfluential), but we report them for the reader's convenience anyway in fig. 9.

3.3 EndPoint Projection

We have to update our definition of EPP to our new language model.

First, as usual, let us gain some intuition. Given any σ, the network implementation of the choreography given in example 2 should look like the

$$\frac{f(v)\downarrow v' \qquad g(u,v')\downarrow u'}{\mathsf{p} \triangleright_v \mathsf{q}!f;B \;\mid\; \mathsf{q} \triangleright_u \mathsf{p}?g;B' \quad\to\quad \mathsf{p} \triangleright_v B \;\mid\; \mathsf{q} \triangleright_{u'} B'}\ \textsc{Com}$$

$$\frac{f(v)\downarrow u}{\mathsf{p} \triangleright_v f;B \;\to\; \mathsf{p} \triangleright_u B}\ \textsc{Local} \qquad \frac{N_1 \to N_1'}{N_1 \mid N_2 \;\to\; N_1' \mid N_2}\ \textsc{Par}$$

$$\frac{N \preceq N_1 \qquad N_1 \to N_2 \qquad N_2 \preceq N'}{N \to N'}\ \textsc{Struct}$$

Figure 8: Stateful processes, semantics.

$$\frac{}{(N_1 \mid N_2) \mid N_3 \;\equiv\; N_1 \mid (N_2 \mid N_3)}\ \text{PA}$$

$$\frac{}{N_1 \mid N_2 \;\equiv\; N_2 \mid N_1}\ \text{PC} \qquad \frac{}{N \mid \mathbf{0} \;\preceq\; N}\ \text{GCN} \qquad \frac{}{\mathsf{p} \triangleright_v \mathbf{0} \;\preceq\; \mathbf{0}}\ \text{GCP}$$

Figure 9: Stateful processes, structural precongruence.

following.

$$\begin{array}{l}\mathsf{Buyer} \triangleright_{\sigma(\mathsf{Buyer})} \mathsf{Seller}!\mathit{title}; \mathsf{Seller}?\mathit{price}; \mathbf{0} \\ \mid \\ \mathsf{Seller} \triangleright_{\sigma(\mathsf{Seller})} \mathsf{Buyer}?x; \mathsf{Buyer}!\mathit{cat}(x); \mathbf{0}\end{array}$$

Exercise 7. *Define EPP for stateful choreographies. EPP should now take a configuration as input—$[\![\langle C, \sigma\rangle]\!]$—since we need to know the memory states of processes to generate a network in stateful processes. As guideline, the choreography in example 2 should be projected to the network above.*

References

M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274. ACM, 2013.

M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In R. D. Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007. doi: 10.1007/978-3-540-71316-6_2.

L. Cruz-Filipe and F. Montesi. A core model for choreographic programming. In O. Kouchnarenko and R. Khosravi, editors, *FACS*, volume 10231 of *LNCS*. Springer, 2016. doi: 10.1007/978-3-319-57666-4_3.

L. Cruz-Filipe and F. Montesi. Procedural choreographic programming. In *FORTE*, LNCS. Springer, 2017.

P. Deniélou and N. Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP (2)*, volume 7966 of *Lecture Notes in Computer Science*, pages 174–186. Springer, 2013.

X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. *International Journal on Web Service Res.*, 2(4):68–93, 2005.

K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. *J. ACM*, 63(1):9, 2016. doi: 10.1145/2827695. URL `http://doi.acm.org/10.1145/2827695`.

Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In *WWW*, pages 973–982. ACM, 2007.