Lecture Notes on Choreographies, Part 5

Fabrizio Montesi
fmontesi@imada.sdu.dk

December 14, 2017

Abstract

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

1 Amendment

Consider again the unprojectable choreography from part 4, below.

$$C_{\mathsf{unproj}} \triangleq (\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p}.\mathtt{true} \to \mathsf{q}.x; \mathbf{0}\ \mathsf{else}\ \mathbf{0})\,;\mathbf{0} \tag{1}$$

Now that we have selections, we can easily fix it by adding more selections until we obtain a projectable choreography.

$$(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p} \to \mathsf{q}[\mathrm{L}]; \mathsf{p}.\mathtt{true} \to \mathsf{q}.x; \mathbf{0}\ \mathsf{else}\ \mathsf{p} \to \mathsf{q}[\mathrm{R}]; \mathbf{0})\,;\mathbf{0} \tag{2}$$

This manual intervention suggests a general principle: if we have an unprojectable choreography, we can try to obtain a projectable one by adding selections inside of conditionals. More precisely, if we add selections with different labels from the process that makes a conditional to the other processes that need to behave differently in the two branches but do not know which branch they are in, then we should obtain a projectable choreography. This intuition is formalised in the following definition, which gives us a mechanical method to "amend" a choreography and obtain a projectable version.

Definition 1 (Amendment [Cruz-Filipe and Montesi, 2016])**.** *Let C be a choreography. The transformation* $\mathsf{Amend}(C)$ *repeatedly applies the following procedure until no longer possible, starting from the innermost subterms in C. For each conditional subterm* $\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ *in C, let* $\{\mathsf{r}_1, \ldots, \mathsf{r}_n\} \subseteq (\mathsf{procs}(C_1) \cup \mathsf{procs}(C_2))$ *be the largest set not containing* p *such that* $[\![C_1]\!]_{\mathsf{r}_i} \sqcup$

$[\![C_2]\!]_{\mathsf{r}_i}$ *is undefined for all* $i \in [1, n]$*; then, the subterm* $\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ *is replaced with:*

$$\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p} \to \mathsf{r}_1[\mathrm{L}]; \cdots ; \mathsf{p} \to \mathsf{r}_n[\mathrm{L}]; C_1\ \mathsf{else}\ \mathsf{p} \to \mathsf{r}_1[\mathrm{R}]; \cdots ; \mathsf{p} \to \mathsf{r}_n[\mathrm{R}]; C_2$$

.

As an example, the result of $\mathsf{Amend}(C_{\mathsf{unproj}})$ (where C_{unproj} is defined in eq. (1)) is the choreography in eq. (2).

Amendment is a complete procedure—it is defined for every choreography C. It also guarantees projectability, as we desired.

Proposition 1. *Let* C *be a choreography and* $\mathsf{Amend}(C) = C'$*. Then, for all* σ*,* $[\![\langle C', \sigma\rangle]\!]$ *is defined.*

Exercise 1. *Prove proposition 1.*

Exercise 2. *Write the result of* $\mathsf{Amend}(C)$*, where* C *is the following choreography.*

$$(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p}.x \to \mathsf{q}.y; \mathbf{0}\ \mathsf{else}\ \mathsf{p}.x \to \mathsf{r}.z; \mathbf{0})\,; \mathbf{0}$$

Exercise 3. *Write the result of* $\mathsf{Amend}(C)$*, where* C *is the following choreography.*

$$(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ \mathsf{p}.x \to \mathsf{q}.y; \mathbf{0}\ \mathsf{else}\ \mathsf{p}.(x+1) \to \mathsf{r}.z; \mathsf{p}.(x+2) \to \mathsf{q}.y; \mathbf{0})\,; \mathbf{0}$$

2 Smart Projection

An alternative to amending choreographies is to change the definition of EPP, such that the auxiliary selections are inserted automatically by the projection procedure. The modification is pretty simple: when we project a conditional, we project a selection towards all other processes involved in the conditional from the process that evaluates the guard, and we project a corresponding branching for all such other processes. Here is the modification:

$$[\![\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C]\!]_{\mathsf{r}} = \begin{cases} (\mathsf{if}\ f\ \mathsf{then}\ B_1; [\![C_1]\!]_{\mathsf{r}}\ \mathsf{else}\ B_2; [\![C_2]\!]_{\mathsf{r}})\,; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ B; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} \neq \mathsf{p} \text{ and } B = [\![C_1]\!]_{\mathsf{r}} \sqcup [\![C_2]\!]_{\mathsf{r}} \\ \mathsf{p}\,\&\{\mathrm{L} : [\![C_1]\!]_{\mathsf{r}}, \mathrm{R} : [\![C_2]\!]_{\mathsf{r}}\}; [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

where $B_1 = \mathsf{r}_1 \oplus \mathrm{L}; \cdots ; \mathsf{r}_n \oplus \mathrm{L}$ and $B_2 = \mathsf{r}_1 \oplus \mathrm{R}; \cdots ; \mathsf{r}_n \oplus \mathrm{R}$
such that $[\![C_1]\!]_{\mathsf{s}} \sqcup [\![C_2]\!]_{\mathsf{s}}$ undefined for all $s \in \{r_1, \ldots, r_n\}$

.

Let $\star\,[\![C]\!]$ be EPP with the rule for projecting conditionals replaced by the rule above. (Notice that we adopt this rule only in this section, and do not use smart EPP in the other sections.)

Proposition 2 (Harmony of amendment and smart projection)**.** *Let C be a choreography and σ be a global memory state. Then, $[\![\langle \mathsf{Amend}(C), \sigma\rangle]\!] = \star\,[\![\langle C, \sigma\rangle]\!]$.*

Exercise 4 (!)**.** *Prove proposition 2.*

3 Recursion

All the choreographies that we have seen so far have finite behaviour, in the sense that all their reduction chains have finite length. Even more, all choreographies in the models that we explored so far must terminate. Let us formalise this property.

Definition 2 (Termination)**.** *We say that $\langle C, \sigma\rangle$ must terminate if: for all C' and σ' such that $\langle C, \sigma\rangle \to^* \langle C', \sigma'\rangle$, there exists σ'' such that $\langle C', \sigma'\rangle \to^* \langle \mathbf{0}, \sigma''\rangle$.*

We say that a choreography C must terminate if $\langle C, \sigma\rangle$ must terminate for all σ.

In other words, a choreography must terminate if its execution will necessarily end in $\mathbf{0}$.

We are now going to extend our choreography model to allow for recursive behaviour—"repetitions" of the same communication structures. In this new model, not all choreographies necessarily terminate, as we are going to see.

3.1 Choreographies

Syntax We extend the syntax our choreography model with two ingredients. The new syntax is displayed in fig. 1.

First, we add a set of procedure definitions, ranged over by $\mathscr{C}$. A set of procedure definitions is a (possibly empty) set of definitions like $X(\tilde{\mathsf{p}}) = C$, read "procedure X has parameters $\tilde{\mathsf{p}}$ and body C". We assume that all procedures defined in a set $\mathscr{C}$ have distinct names. This means that we can abstract from the order in which procedures are defined, and thus write $X(\tilde{\mathsf{p}}) = C \in \mathscr{C}$ to state that the definition $X(\tilde{\mathsf{p}}) = C$ is in $\mathscr{C}$. The notation $\tilde{\mathsf{p}}$ is a shortcut for a sequence $\mathsf{p}_1, \ldots, \mathsf{p}_n$ for some n (a list of process names).

$$
\begin{aligned}
C &::= I; C \mid \mathbf{0} \\
I &::= \mathsf{p}.f \to \mathsf{q}.g \mid \mathsf{p} \to \mathsf{q}[l] \mid \mathsf{p}.f \mid \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid X(\tilde{\mathsf{p}}) \mid \mathbf{0} \\
\mathscr{C} &::= X(\tilde{\mathsf{p}}) = C, \mathscr{C} \mid \emptyset
\end{aligned}
$$

Figure 1: Recursive choreographies, syntax.

We assume that all process names in a list of parameters $\tilde{\mathsf{p}}$ are distinct (they are all different).

Second, we add a new primitive to choreographies for invoking procedures, namely $X(\tilde{\mathsf{p}})$, read "run procedure X with arguments $\tilde{\mathsf{p}}$". Also here, we assume that the list of arguments $\tilde{\mathsf{p}}$ contains distinct process names.

Example 1. *A typical way of transmitting a large file over a network is to split the file into multiple parts (or "chunks", or packets) that are then re-assembled at the destination. We implement this strategy here.*

First, we write a procedure S that streams a series of packets from a server s to a client c.

$$
\begin{aligned}
S(\mathsf{c},\mathsf{s}) = \quad & \mathsf{if}\ \mathsf{s}.(n > 0)\ \mathsf{then} \\
& \qquad \mathsf{s} \to \mathsf{c}[\textsc{next}]; \\
& \qquad \mathsf{s}.next \to \mathsf{c}.recvNext; \\
& \qquad \mathsf{s}.dec(n); \\
& \qquad S(\mathsf{c},\mathsf{s}); \\
& \qquad \mathbf{0} \\
& \mathsf{else} \\
& \qquad \mathsf{s} \to \mathsf{c}[\textsc{end}]; \\
& \qquad \mathbf{0} \\
; \mathbf{0}
\end{aligned}
$$

Procedure S is recursive. The guard of the conditional checks that variable n at s contains a value bigger than 0 (the number zero). If so, s informs the client c that it will receive a packet. Function next returns the current packet to send, and function $recvNext$ stores it appropriately in some data structure (e.g., an array) at the client. Then, we decrement n by 1 at the server s and we recursively invoke S to send the remaining packets. When we have finished the packets to send ($n \leq 0$), s sends the label END *to c to inform it that the procedure will terminate.*

Using S, we can write our choreography for sending a large file as chunks.

$$\mathsf{c}.filename \to \mathsf{s}.x;\ \ \mathsf{s}.buildChunks;\ \ S(\mathsf{c},\mathsf{s});\ \ \mathbf{0}$$

Here, the client sends the filename of the file it wishes to download to s. *Then* s *uses function buildChunks, which we assume sets up the right index* n *and the chunks for next used in procedure* S. *Finally, we invoke* S *to perform the streaming.*

Since we extended the syntax of choreographies, we have to update our definition of procs.

$$\begin{aligned}
\mathsf{procs}\,(I; C) &= \mathsf{procs}(I) \cup \mathsf{procs}(C) \\
\mathsf{procs}(\mathbf{0}) &= \emptyset \\
\mathsf{procs}\,(\mathsf{p}.f \to \mathsf{q}.g) &= \{\mathsf{p}, \mathsf{q}\} \\
\mathsf{procs}\,(\mathsf{p} \to \mathsf{q}[l]) &= \{\mathsf{p}, \mathsf{q}\} \\
\mathsf{procs}\,(\mathsf{p}.f) &= \{\mathsf{p}\} \\
\mathsf{procs}\,(\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) &= \{\mathsf{p}\} \cup \mathsf{procs}(C_1) \cup \mathsf{procs}(C_2) \\
\mathsf{procs}(X(\tilde{\mathsf{p}})) &= \{\tilde{\mathsf{p}}\}
\end{aligned}$$

In the remainder, whenever we consider a procedure definition $X(\tilde{\mathsf{p}}) = C$, we assume that $\mathsf{procs}(C) = \{\tilde{\mathsf{p}}\}$, meaning that the process names used in the body of the procedure are exactly those declared as parameters.

Semantics We extend the semantics of our choreography model to consider also the set of procedure definitions under which we are executing. This means that the definition of our reduction relation now depends on the set of procedure definitions $\mathscr{C}$ that we are considering. We thus denote reductions as $\langle C, \sigma\rangle \to_{\mathscr{C}} \langle C', \sigma'\rangle$, read "the configuration $\langle C, \sigma\rangle$ reduces to $\langle C', \sigma'\rangle$ assuming that procedures are defined as in $\mathscr{C}$".

Remark 1. *An alternative notation for reductions that consider procedure definitions could be* $\langle C, \sigma, \mathscr{C}\rangle \to \langle C', \sigma', \mathscr{C}'\rangle$. *However, this notation makes it look like* $\mathscr{C}$ *is part of the "state of the system", just like* C *(the program to run next) and* σ *(the memory of processes). The state of a system can typically change during execution. Thus the question become: is* $\mathscr{C}$ *ever going to change?*

In most programming systems, the definitions of procedures never do, and this will be the case also in our model. Therefore it makes more sense to put $\mathscr{C}$ *out of the notation for the state of the system, and write* $\langle C, \sigma\rangle \to_{\mathscr{C}} \langle C', \sigma'\rangle$ *as we do.*

There are, however, examples of programming models where the code of procedures might change at runtime. In these settings, the alternative notation makes more sense, since $\mathscr{C}$ *might change. A model where the definitions*

$$\frac{f(\sigma(\mathsf{p}))\downarrow v \quad g(\sigma(\mathsf{q}),v)\downarrow u}{\langle \mathsf{p}.f \twoheadrightarrow \mathsf{q}.g; C, \sigma\rangle \rightarrow_{\mathscr{C}} \langle C, \sigma[\mathsf{q} \mapsto u]\rangle}\ \textsc{Com} \qquad \frac{}{\langle \mathsf{p} \twoheadrightarrow \mathsf{q}[l]; C, \sigma\rangle \rightarrow_{\mathscr{C}} \langle C, \sigma\rangle}\ \textsc{Sel}$$

$$\frac{f(\sigma(\mathsf{p}))\downarrow v}{\langle \mathsf{p}.f; C, \sigma\rangle \rightarrow_{\mathscr{C}} \langle C, \sigma[\mathsf{p} \mapsto v]\rangle}\ \textsc{Local}$$

$$\frac{i = 1 \text{ if } f(\sigma(\mathsf{p}))\downarrow \texttt{true},\ i = 2 \text{ otherwise}}{\langle \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C, \sigma\rangle \rightarrow_{\mathscr{C}} \langle C_i; C, \sigma\rangle}\ \textsc{Cond}$$

$$\frac{C \preceq_{\mathscr{C}} C_1 \quad \langle C_1, \sigma\rangle \rightarrow_{\mathscr{C}} \langle C_2, \sigma'\rangle \quad C_2 \preceq_{\mathscr{C}} C'}{\langle C, \sigma\rangle \rightarrow_{\mathscr{C}} \langle C', \sigma'\rangle}\ \textsc{Struct}$$

Figure 2: Recursive choreographies, semantics.

of choreographies can evolve at runtime was presented by Dalla Preda et al. [2017].

The new rules defining the semantics of choreographies are displayed in figs. 2 and 3. Structural precongruence is also annotated with $\mathscr{C}$ now, since it requires to know the definitions of procedures. The only new rule is Unfold, in fig. 3. It states that an invocation of procedure X can be replaced by the body of the procedure, replacing the parameters of the procedure with the arguments passed by the invocation site—$\tilde{\mathsf{p}}/\tilde{\mathsf{q}}$ is a shortcut for replacing each occurrence of q_i in C with the corresponding p_i (assuming that $\tilde{\mathsf{p}}$ and $\tilde{\mathsf{q}}$ have the same length).

Example 2. *Let $\mathscr{C}$ be the singleton set containing the definition of procedure S in example 1. Let C be the choreography in the same example:*

$$C \triangleq \mathsf{c}.filename \twoheadrightarrow \mathsf{s}.x;\ \mathsf{s}.buildChunks;\ S(\mathsf{c},\mathsf{s});\ \mathbf{0}$$

. By rule Unfold*, we can replace the invocation of S inside of C with the*

$$\frac{\mathsf{procs}(I) \# \mathsf{procs}(I')}{I; I' \quad \equiv_{\mathscr{C}} \quad I'; I} \text{ I-I}$$

$$\frac{\mathsf{p} \notin \mathsf{procs}(I)}{I; \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \quad \equiv_{\mathscr{C}} \quad \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ (I; C_1)\ \mathsf{else}\ (I; C_2)} \text{ I-Cond}$$

$$\frac{\mathsf{p} \notin \mathsf{procs}(I) \quad I \neq \mathbf{0}}{\mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; I \quad \equiv_{\mathscr{C}} \quad \mathsf{if}\ \mathsf{p}.f\ \mathsf{then}\ (C_1; I)\ \mathsf{else}\ (C_2; I)} \text{ Cond-I}$$

$$\frac{}{\mathbf{0}; C \preceq_{\mathscr{C}} C} \text{ GCNil}$$

$$\frac{X(\tilde{\mathsf{q}}) = C \ \in \ \mathscr{C}}{X(\tilde{\mathsf{p}}) \preceq_{\mathscr{C}} C[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]} \text{ Unfold}$$

Figure 3: Recursive choreographies, structural precongruence.

body of S, as follows.

$$\begin{aligned}
C \preceq_D \quad & \mathsf{c}.filename \twoheadrightarrow \mathsf{s}.x; \\
& \quad \mathsf{s}.buildChunks; \\
& \quad \mathsf{if}\ \mathsf{s}.(n > 0)\ \mathsf{then} \\
& \qquad\qquad\qquad\qquad \mathsf{s} \twoheadrightarrow \mathsf{c}[\textsc{next}]; \\
& \qquad\qquad\qquad\qquad \mathsf{s}.next \twoheadrightarrow \mathsf{c}.recvNext; \\
& \qquad\qquad\qquad\qquad \mathsf{s}.dec(n); \\
& \qquad\qquad\qquad\qquad S(\mathsf{c}, \mathsf{s}); \\
& \qquad\qquad\qquad\qquad \mathbf{0} \\
& \qquad\qquad\qquad\ \mathsf{else} \\
& \qquad\qquad\qquad\qquad \mathsf{s} \twoheadrightarrow \mathsf{c}[\textsc{end}]; \\
& \qquad\qquad\qquad\qquad \mathbf{0} \\
& \quad ; \mathbf{0}
\end{aligned}$$

Example 3. *We can now write choreographies that can always continue running, i.e., they never terminate.*

The following procedure implements a ping-pong communication structure between two processes p *and* q*, which take turns in pinging each other. Notice how, when we invoke the procedure, we invert the order of processes to make them take turns.*

$$PingPong(\mathsf{p}, \mathsf{q}) = \ \mathsf{p}.ping \twoheadrightarrow \mathsf{q}.x;\ \mathsf{q}.pong \twoheadrightarrow \mathsf{p}.y;\ PingPong(\mathsf{q}, \mathsf{p});\ \mathbf{0}$$

$$
\begin{aligned}
N &::= \mathsf{p} \triangleright_v B \mid N \mid N \mid \mathbf{0} \\
B &::= \mathsf{p}!f; B \mid \mathsf{p}?f; B \mid \mathsf{p} \oplus l; B \mid \mathsf{p} \,\&\{l_i : B_i\}_{i \in I}; B \\
&\quad \mid \mathsf{if}\ f\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2; B \mid \mathbf{0}; B \mid X(\tilde{\mathsf{p}}) \mid \mathbf{0} \\
\mathscr{B} &::= X(\tilde{\mathsf{p}}) = B, \mathscr{B} \mid \emptyset
\end{aligned}
$$

Figure 4: Recursive processes, syntax.

The turns are evident by looking at the unfolding of PingPong:

$$
\begin{array}{l}
\mathsf{p}.ping \twoheadrightarrow \mathsf{q}.x; \\
\mathsf{q}.pong \twoheadrightarrow \mathsf{p}.y; \\
PingPong(\mathsf{q}, \mathsf{p}); \\
\mathbf{0}
\end{array}
\quad \preceq_D \quad
\begin{array}{l}
\mathsf{p}.ping \twoheadrightarrow \mathsf{q}.x; \\
\mathsf{q}.pong \twoheadrightarrow \mathsf{p}.y; \\
\mathsf{q}.ping \twoheadrightarrow \mathsf{p}.x; \\
\mathsf{p}.pong \twoheadrightarrow \mathsf{q}.y; \\
PingPong(\mathsf{p}, \mathsf{q}); \\
\mathbf{0}; \\
\mathbf{0}
\end{array}
$$

.

Exercise 5. *Prove that the choreography* $PingPong(\mathsf{p}, \mathsf{q}); \mathbf{0}$ *never terminates.*

3.2 Processes

The extension to our process model to include recursion is very similar to that for choreographies. The new syntax and semantics of processes are displayed in figs. 4 to 6. The additions are the new syntax term for invoking procedures and an unfolding rule for structural precongruence.

3.3 Projection

Consider again our procedure $PingPong$ and a choreography C_{pp} that invokes it.

$$
\begin{array}{rcl}
\mathscr{C}_{pp} & \triangleq & PingPong(\mathsf{p}, \mathsf{q}) = \ \mathsf{p}.ping \twoheadrightarrow \mathsf{q}.x;\ \mathsf{q}.pong \twoheadrightarrow \mathsf{p}.y;\ PingPong(\mathsf{q}, \mathsf{p});\ \mathbf{0} \\
C_{pp} & \triangleq & PingPong(\mathsf{p}, \mathsf{q}); \mathbf{0}
\end{array}
$$

How should we project C_{pp}? The idea is that procedure $PingPong$ should be translated to two procedures on the process level: one that describes the behaviour of p inside of its body—let us call this procedure $PingPong_{\mathsf{p}}$—and another that describes the behaviour of q—let us call this procedure

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

$$\frac{j\in I}{\mathsf{p}\triangleright_v \mathsf{q}\oplus l_j;B \;\mid\; \mathsf{q}\triangleright_u \mathsf{p}\,\&\{l_i:B_i\}_{i\in I};B' \quad \to_{\mathscr{B}} \quad \mathsf{p}\triangleright_v B \;\mid\; \mathsf{q}\triangleright_u B_j;B'}\ \textsc{Sel}$$

$$\frac{i=1 \text{ if } f(v)\downarrow \texttt{true},\ i=2 \text{ otherwise}}{\mathsf{p}\triangleright_v (\mathsf{if}\ f\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2)\,;B \to_{\mathscr{B}} \mathsf{p}\triangleright_v B_i;B}\ \textsc{Cond}$$

$$\frac{N_1 \to_{\mathscr{B}} N_1'}{N_1 \mid N_2 \ \to_{\mathscr{B}}\ N_1' \mid N_2}\ \textsc{Par} \qquad \frac{N \preceq_{\mathscr{B}} N_1 \quad N_1 \to_{\mathscr{B}} N_2 \quad N_2 \preceq_{\mathscr{B}} N'}{N \to_{\mathscr{B}} N'}\ \textsc{Struct}$$

Figure 5: Recursive processes, semantics.

$$\frac{}{(N_1 \mid N_2) \mid N_3 \ \equiv_{\mathscr{B}}\ N_1 \mid (N_2 \mid N_3)}\ \text{PA} \qquad \frac{}{\mathbf{0};B \preceq_{\mathscr{B}} B}\ \text{GCB}$$

$$\frac{}{N_1 \mid N_2 \ \equiv_{\mathscr{B}}\ N_2 \mid N_1}\ \text{PC} \qquad \frac{}{N \mid \mathbf{0} \ \preceq_{\mathscr{B}}\ N}\ \text{GCN} \qquad \frac{}{\mathsf{p}\triangleright_v \mathbf{0} \ \preceq_{\mathscr{B}}\ \mathbf{0}}\ \text{GCP}$$

$$\frac{X(\tilde{\mathsf{q}}) = B \ \in\ \mathscr{B}}{X(\tilde{\mathsf{p}}) \preceq_{\mathscr{B}} B[\tilde{\mathsf{p}}/\tilde{\mathsf{q}}]}\ \textsc{Unfold}$$

Figure 6: Recursive processes, structural precongruence.

$PingPong_{\mathsf{q}}$. Thus we obtain the following set of procedure definitions for process behaviours.

$$\mathscr{B}_{pp} \triangleq \begin{array}{ll} PingPong_{\mathsf{p}}(\mathsf{q}) = & \mathsf{q}!ping; \\ & \mathsf{q}?y; \\ & PingPong_{\mathsf{q}}(\mathsf{q}) \\ & \mathbf{0}, \\ PingPong_{\mathsf{q}}(\mathsf{p}) = & \mathsf{p}?x; \\ & \mathsf{p}!pong; \\ & PingPong_{\mathsf{p}}(\mathsf{p}) \\ & \mathbf{0} \end{array}$$

We can then project C_{pp} as follows, for some σ.

$$[\![\langle C_{pp}, \sigma\rangle]\!] = \quad \mathsf{p} \triangleright_{\sigma(\mathsf{p})} PingPong_{\mathsf{p}}(\mathsf{q}); \mathbf{0} \mid \mathsf{p} \triangleright_{\sigma(\mathsf{p})} PingPong_{\mathsf{q}}(\mathsf{p}); \mathbf{0}$$

Exercise 6. *Write down the first four reductions of C_{pp} and its projection above. Do they correspond?*

Our example shows that now we need to be able to project both choreographies and procedure definitions.

For projecting choreographies, the definition is the same as before (but we will have to update behaviour projection, $[\![C]\!]_{\mathsf{p}}$).

Definition 3 (EndPoint Projection (EPP))**.** *The EPP of a configuration $\langle C, \sigma\rangle$, denoted $[\![\langle C, \sigma\rangle]\!]$, is defined as:*

$$[\![\langle C, \sigma\rangle]\!] = \prod_{\mathsf{p} \in \mathsf{procs}(C)} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} [\![C]\!]_{\mathsf{p}}$$

.

To update behaviour projection to deal with procedure invocations, we first update merging. This is done by adding the following rule.

$$X(\tilde{\mathsf{p}}); C \sqcup X(\tilde{\mathsf{p}}); C' \;=\; X(\tilde{\mathsf{p}}); (C \sqcup C')$$

Then, we simply need to define the projection of a procedure invocation.

$$[\![X(\tilde{\mathsf{p}}); C]\!]_{\mathsf{r}} = \begin{cases} X_i(\tilde{\mathsf{p}} \setminus \mathsf{p}_i); [\![C]\!]_{\mathsf{r}} & \text{if } \tilde{\mathsf{p}} = \mathsf{p}_1, \ldots, \mathsf{p}_n \text{ and } \mathsf{r} = \mathsf{p}_i \text{ and } 1 \leq i \leq n \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

The notation $\tilde{\mathsf{p}} \setminus \mathsf{r}$ means "the list obtained by removing r from $\tilde{\mathsf{p}}$". Observe that the procedure invocation that we output for a process involved

in the choreographic invocation is for X_{r}, which we assume implements the behaviour for process r in X.

Now that we know how to project a recursive choreography, we can write the definition of projection for procedure definitions.

Definition 4. *The EPP of a set of procedure definitions $\mathscr{C}$, denoted $[\![\mathscr{C}]\!]$, is defined as:*

$$[\![\mathscr{C}]\!] = \left\{ X_i(\tilde{\mathsf{p}} \setminus \mathsf{p}_i) = [\![C]\!]_{\mathsf{p}} \;\middle|\; \begin{array}{l} X(\tilde{\mathsf{p}}) = C \in \mathscr{C} \text{ and} \\ \tilde{\mathsf{p}} = \mathsf{p}_1, \ldots, \mathsf{p}_n \text{ and } 1 \leq i \leq n \end{array} \right\}$$

.

Exercise 7. *Write the EPP of the procedure and choreography in example 1.*

We can formulate an operational correspondence for recursive choreographies and their EPP as follows.

Theorem 1 (Operational Correspondence)**.** *Let $[\![\langle C, \sigma\rangle]\!] = N$ and $[\![\mathscr{C}]\!] = \mathscr{B}$. Then,*

Completeness *If $\langle C, \sigma\rangle \rightarrow_{\mathscr{C}} \langle C', \sigma'\rangle$ for some C' and σ', then there exists N' such that $N \rightarrow_{\mathscr{B}} N'$ and $N' \preceq_{\mathscr{B}} \sqsupseteq [\![\langle C', \sigma'\rangle]\!]$.*

Soundness *If $N \rightarrow_{\mathscr{B}} N'$ for some N', then there exists C' and σ' such that $\langle C, \sigma\rangle \rightarrow_{\mathscr{C}} \langle C', \sigma'\rangle$ and $N' \preceq_{\mathscr{B}} \sqsupseteq [\![\langle C', \sigma'\rangle]\!]$.*

References

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.

M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), 2017.