Choreographies in Practice

Luís Cruz-Filipe, Fabrizio Montesi

Choreographies in Practice*

Luís Cruz-Filipe, Fabrizio Montesi

University of Southern Denmark, Department of Mathematics and Computer Science
{lcf,fmontesi}@imada.sdu.dk

Abstract. Choreographic Programming is a development methodology for concurrent software that guarantees correctness by construction. The key to this paradigm is to disallow mismatched I/O operations in programs, and mechanically synthesise process implementations.
There is still a lack of practical illustrations of the applicability of choreographies to computational problems with standard concurrent solutions. In this work, we explore the potential of choreographic programming by writing concurrent algorithms for sorting, solving linear equations, and computing Fast Fourier Transforms. The lessons learned from this experiment give directions for future improvements of the paradigm.

1 Introduction

Choreographic Programming is an emerging paradigm for developing concurrent software based on message passing [16]. Its key aspect is that programs are choreographies – global descriptions of communications based on an "Alice and Bob" security protocol notation. Since this notation disallows mismatched I/O actions, choreographies always describe deadlock-free systems by construction. Given a choreography, a distributed implementation can be projected automatically (synthesis) onto terms of a process model – a transformation called End-Point Projection (EPP) [2,3]. A correct definition of EPP yields a correctness-by-construction result: since a choreography cannot describe deadlocks, the generated process implementations are also deadlock-free. Previous works presented formal models capturing different aspects of choreographic programming, e.g., web services [2,12], asynchronous multiparty sessions [3], runtime adaptation [9], modular development [18], protocol compliance [3,4], and computational expressivity [7]. Choreography models have also been investigated in the realms of type theory [14], automata theory [11], formal logics [5], and service contracts [1].

Despite the rising interest in choreographic programming, there is still a lack of evidence about what nontrivial programs can actually be written with this paradigm. This is due to its young age [17]. Indeed, most works on languages for choreographic programming still focus on showcasing representative toy examples (e.g., [2,3,6,12,16,18]), rather than giving a comprehensive practical evaluation based on standard computational problems.

* Supported by *CRC (Choreographies for Reliable and efficient Communication software)*, grant DFF–4005-00304 from the Danish Council for Independent Research.

In this work, we contribute to filling this gap. Our investigation uses the language of Procedural Choreographies (PC) [8], summarised in § 2, which extends previous choreography models with primitives for parameterised procedures. Like other choreography languages (e.g., [3,18]), PC supports implicit parallelism: non-interfering communications can take place in any order. We provide an empirical evaluation of the expressivity of PC, by using it to program some representative and standard concurrent algorithms: Quicksort (§ 3), Gaussian elimination (§ 4), and Fast Fourier Transform (§ 5). As a consequence of using choreographies, all these implementations are guaranteed to be deadlock-free. We also illustrate how implicit parallelism has the surprising effect of automatically giving concurrent behaviour to traditional sequential implementations of these algorithms. Our exploration brings us to the limits of the expressivity of PC, which arise when trying to tackle distributed graph algorithms (§ 6), due to the lack of primitives for accessing the structure of process networks, e.g., broadcasting a message to neighbouring processes.

2 Background

In this section, we recap the language and properties of Procedural Choreographies (PC). We refer the reader to [8] for a more comprehensive presentation.

Procedural Choreographies. The syntax of PC is given below:

$$C ::= \eta; C \mid I; C \mid \mathbf{0} \qquad \eta ::= \mathsf{p}.e \texttt{ -> } \mathsf{q}.f \mid \mathsf{p} \texttt{ -> } \mathsf{q}[l] \mid \mathsf{p} \,\mathsf{start}\, \mathsf{q} \mid \mathsf{p}\texttt{: } \mathsf{q} \texttt{ <-> } \mathsf{r}$$
$$\mathcal{D} ::= X(\tilde{\mathsf{q}}) = C, \mathcal{D} \mid \emptyset \qquad I ::= \mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid X\langle\tilde{\mathsf{p}}\rangle \mid \mathbf{0}$$

A procedural choreography is a pair $\langle \mathcal{D}, C \rangle$, where C is a choreography and $\mathcal{D}$ is a set of procedure definitions. Process names, ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$, identify processes that execute concurrently. Each process p is equipped with a memory cell storing a single value of a fixed type. In the remainder, we omit type information since they can always be inferred using the technique given in [8]. Statements in a choreography can either be communication actions (η) or compound instructions (I), and both can have continuations. Term $\mathbf{0}$ is the terminated choreography, often omitted, and $\mathbf{0}; A$ is used only at runtime.

Processes communicate (synchronously) via direct references (names) to each other. In a value communication $\mathsf{p}.e$ -> $\mathsf{q}.f$, process p evaluates expression e and sends the result to q; e can contain the placeholder c, replaced at runtime with the data stored at p. When q receives the value from p, it applies to it the (total) function f and stores the result. The body of f can also contain c, which is replaced by the contents of q's memory. Expressions and functions are written in a pure functional language, left unspecified.

In a selection p -> $\mathsf{q}[l]$, p communicates to q its choice of label l.

In term $\mathsf{p}\,\mathsf{start}\,\mathsf{q}$, process p spawns the new process q, whose name is bound in the continuation C of $\mathsf{p}\,\mathsf{start}\,\mathsf{q}; C$. After executing $\mathsf{p}\,\mathsf{start}\,\mathsf{q}$, p is the only process who knows the name of q. This knowledge is propagated to other processes by the action p: q <-> r, read "p introduces q and r", where p, q and r are distinct.

In a conditional term $\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, process p evaluates expression e to choose between the possible continuations C_1 and C_2.

The set $\mathcal{D}$ defines global procedures that can be invoked in choreographies. Term $X(\tilde{\mathsf{q}}) = C$ defines a procedure X with body C, which can be used anywhere in $\langle \mathcal{D}, C \rangle$ – in particular, inside the definitions of X and other procedures. The names $\tilde{\mathsf{q}}$ are bound to C, and are assumed to be exactly the free process names in C. The set $\mathcal{D}$ contains at most one definition for each procedure name. Term $X\langle\tilde{\mathsf{p}}\rangle$ invokes procedure X, instantiating its parameters with the processes $\tilde{\mathsf{p}}$.

The semantics of PC, which we do not detail, is a reduction semantics that relies on two extra elements: a total state function that assigns to each process the value it stores, and a connection graph that keeps track of which processes know (are connected to) each other [8]. In particular, processes can only communicate if there is an edge between them in the connection graph. Therefore, choreographies can deadlock because of errors in the programming of communications: if two processes try to communicate but they are not connected, the choreography gets stuck. This issue is addressed by a simple typing discipline, which guarantees that well-typed PC choreographies are deadlock-free [8].

Procedural Processes. Choreographies in PC are compiled into terms of the calculus of Procedural Processes (PP), which has the following syntax:

$$\begin{array}{rl} B ::= & \mathsf{q}!e; B \mid \mathsf{p}?f; B \mid \mathsf{q}!!\mathsf{r}; B \mid \mathsf{p}?\mathsf{r}; B \mid \mathsf{q} \oplus l; B \mid \mathsf{p}\&\{l_i : B_i\}_{i\in I}; B \mid \mathbf{0} \\ & \mid \mathsf{start}\ \mathsf{q} \triangleright B_2; B_1 \mid \mathsf{if}\ e\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2; B \mid X\langle\tilde{\mathsf{p}}\rangle; B \mid \mathbf{0}; B \\ N, M ::= & \mathsf{p} \triangleright_v B \quad \mid \quad N \mid M \quad \mid \quad \mathbf{0} \qquad\qquad \mathcal{B} ::= X(\tilde{\mathsf{q}}) = B, \mathcal{B} \mid \emptyset \end{array}$$

A term $\mathsf{p} \triangleright_v B$ is a process, where p is its name, v is the value it stores, and B is its behaviour. Networks, ranged over by N, M, are parallel compositions of processes, where $\mathbf{0}$ is the inactive network. Finally, $\langle \mathcal{B}, N \rangle$ is a procedural network, where $\mathcal{B}$ defines the procedures that the processes in N may invoke.

We comment on behaviours. A send term $\mathsf{q}!e; B$ sends the evaluation of expression e to process q, and then proceeds as B. Dually, term $\mathsf{p}?f; B$ receives a value from process p, combines it with the value in memory cell of the process executing the behaviour as specified by f, and then proceeds as B. Term $\mathsf{q}!!\mathsf{r}$ sends process name r to q and process name q to r, making q and r "aware" of each other. The dual action is $\mathsf{p}?\mathsf{r}$, which receives a process name from p that replaces the bound variable r in the continuation. Term $\mathsf{q} \oplus l; B$ sends the selection of a label l to process q. Selections are received by the branching term $\mathsf{p}\&\{l_i : B_i\}_{i\in I}$ (I nonempty), which receives a selection for a label l_i and proceeds as B_i. Term $\mathsf{start}\ \mathsf{q} \triangleright B_2; B_1$ starts a new process (with a fresh name) executing B_2, proceeding in parallel as B_1. Other terms (conditionals, procedure calls, and termination) are standard; procedural definitions are stored globally as in PC.

Term $\mathsf{start}\ \mathsf{q} \triangleright B_2; B_1$ binds q in B_1, and $\mathsf{p}?\mathsf{r}; B$ binds r in B. We omit the formal semantics of PP, which follows the intuitions given above.

EndPoint Projection (EPP). In [8] we show how every well-typed choreography can be projected into a PP network by means of an EndPoint Projection (EPP). EPP guarantees a strict operational correspondence: the projection of a

choreography implements exactly the behaviour of the originating choreography. As a consequence, projections of typable PC terms never deadlock.

3 Quicksort

In this section, we illustrate PC's capability of supporting divide-and-conquer algorithms, by providing a detailed implementation of (concurrent) Quicksort.

We begin by defining procedure `split`, which splits the (non-empty) list stored at `p` among three processes: $q_<$, $q_=$ and $q_>$. We assume that all processes store objects of type `List(T)`, where `T` is some type, endowed with the following constant-time operations: get the first element (`fst`); get the second element (`snd`); check that the length of a list is at most 1 (`short`); append an element (`add`); and append another list (`append`). Also, `fst<snd` and `fst>snd` test whether the first element of the list is, respectively, smaller or greater than the second. Procedure `pop2` (omitted) removes the second element from the list.

We write `p -> q1,...,qn[l]` as an abbreviation for the sequence of selections `p -> q1[l]; ...; p -> qn[l]`. We can now define `split`.

```
split(p,q<,q=,q>) =
  if p.short then p -> q<,q=,q>[stop]; p.fst -> q=.add
  else if p.fst<snd then p -> q<[get]; p.snd -> q<.add; p -> q=,q>[skip]
       else if p.fst>snd then p -> q>[get]; p.snd -> q>.add; p -> q<,q=[skip]
            else p -> q=[get]; p.snd -> q=.add; p -> q<,q>[skip]
       ; pop2<p>; split<p,q<,q=,q>>
```

When `split` terminates, we know that all elements in $q_<$ and $q_>$ are respectively smaller or greater than those in $q_=$.[1] Using `split` we can implement a robust version of Quicksort (lists may contain duplicates), the procedure `QS` below. We write `p start q1,..., qn` for `p start q1;...; p start qn`. Note that `split` is only called when `p` stores a non-empty list.

```
QS(p) = if p.short then 0
        else p.start q<,q=,q>;
             split<p,q<,q=,q>>; QS<q<>; QS<q>>;
             q<.c -> p.id; q=.c -> p.append; q>.c -> p.append
```

Procedure `QS` implements Quicksort using its standard recursive structure. Since the created processes $q_<$, $q_=$ and $q_>$ do not have references to each other, they cannot exchange messages, and thus the recursive calls run completely in parallel. Applying EPP, we get the following process procedures (among others).

```
split_p(p,q<,q=,q>) =
  if short then q<⊕stop; q=⊕stop; q>⊕stop; q=!fst
  else if fst<snd then q<⊕get; q<!snd; q=⊕skip; q>⊕skip
       else if fst>snd then q>⊕get; q>!snd; q<⊕skip; q=⊕skip
            else q=⊕get; q=!snd; q<⊕skip; q>⊕skip
       ; pop2<p>; split_p<p,q<,q=,q>>

split_q<(p,q) = p&{stop: 0, get: p?add;split_q<(p,q), skip: split_q<(p,q)}

QS_p(p) = if small then 0
          else (start q< ▷ split_q<<p,q<>; QS_p<q<>; p!c);
               (start q= ▷ split_q=<p,q=>; p!c);
               (start q> ▷ split_q><p,q>>; QS_p<q>>; p!c);
               q<?id; q=?append; q>?append
```

[1] The selections of label `skip` are required for projectability, see [8].

4 Gauss Elimination

Let $A\boldsymbol{x} = \boldsymbol{b}$ be a system of linear equations in matrix form. We define a procedure `gauss` that applies Gaussian elimination to transform it into an equivalent system $U\boldsymbol{x} = \boldsymbol{y}$, with U upper triangular (so this system can be solved by direct substitution). We use parameter processes $\mathtt{a}_{ij}$, with $1 \leq i \leq n$ and $1 \leq j \leq n+1$. For $1 \leq i, j \leq n$, $\mathtt{a}_{ij}$ stores one value from the coefficient matrix; $\mathtt{a}_{i,n+1}$ stores the independent term in one equation. (Including $\boldsymbol{b}$ in the coefficient matrix simplifies the notation.) After execution, each $\mathtt{a}_{ij}$ stores the corresponding term in the new system. We assume A to be non-singular and numerically stable.

This algorithm cannot be implemented in PC directly, as `gauss` takes a variable number of parameters (the $\mathtt{a}_{ij}$). However, it is easy to extend PC so that procedures can also take process *lists* as parameters, as we describe.

Syntax of PC and PP. The arguments of parametric procedures are now lists of process names, all with the same type. These lists can only be used in procedure calls, where they can be manipulated by means of pure functions that take a list as their only argument. Our examples use uppercase letters to identify process lists and lowercase letters for normal process identifiers.

Semantics of PC. We assume that a procedure that is called with an empty list as one of its arguments is equivalent to the terminated process **0**.

Connections. Connections between processes are uniform wrt argument lists, i.e., if `p` and `A` are arguments to some procedure `X`, then `X` requires/guarantees that `p` be connected to none or all of the processes in `A`.

The definition of `gauss` uses: `hd` and `tl` (computing the head and tail of a list of processes); `fst` and `rest` (taking a list of processes representing a matrix and returning the first row of the matrix, or the matrix without its first row); and `minor` (removing the first row and the first column from a matrix). Processes use standard arithmetic operations to combine their value with values received.

```
gauss(A) = solve(fst(A)); eliminate(fst(A),rest(A)); gauss(minor(A))

solve(A) = divide_all(hd(A),tl(A)); set_to_1(hd(A))

divide_all(a,A) = divide(a,hd(A)); divide_all(a,tl(A))
divide(a,b) = a.c -> b.div

eliminate(A,B) = elim_row(A,fst(B)); eliminate(A,rest(B))
elim_row(A,B) = elim_all(tl(A),hd(B),tl(B)); set_to_0(hd(B))
elim_all(A,m,B) = elim1(hd(A),m,hd(B)); elim_all(tl(A),m,tl(B))
elim1(a,m,b) = b start x; b: x <-> a; b: x <-> m;
                  a.c -> x.id; m.c -> x.mult; x.c -> b.minus

set_to_0(a) = a start p; p.0 -> a.id
set_to_1(a) = a start p; p.1 -> a.id
```

Procedure `solve` divides the first equation by the pivot. Then, `eliminate` uses this row to perform an elimination step, setting the first column of the coefficient matrix to zeroes. The auxiliary procedure `elim_row` performs this step at the row level, using `elim_all` to iterate through a single row and `elim1` to perform the actual computations. The first row and the first column of the matrix are then removed in the recursive call, as they will not change further.

This implementation follows the standard sequential algorithm for Gaussian elimination (Algorithm 8.4 in [13]). However, it runs concurrently due to the implicit parallelism in the semantics of choreographies. We explain this behaviour by focusing on a concrete example. Assume that A is a 3×3 matrix, so there are 12 processes in total. For legibility, we will write `b1` for the independent term `a14` etc.; `A=`$\langle$`a11,a12,a13,b1,a21,a22,a23,b2,a31,a32,a33,b3`$\rangle$ for the matrix; `A1=`$\langle$`a11,a12,a13,b1`$\rangle$ for the first row (likewise for `A2` and `A3`); and, `A'2=`$\langle$`a22,a23,b2`$\rangle$ and likewise for `A'3`. Calling `gauss(A)` unfolds to

```
solve(A1); elim_row(A1,A2); elim_row(A1,A3);
solve(A'2); elim_row(A'2,A'3);
solve(⟨a33,b3⟩)
```

Fully expanding the sequence `elim_row(A1,A3); solve(A'2)` yields

```
elim1(a12,a31,a32); elim1(a13,a31,a33); elim1(b1,a31,b3); set_to_0(a31);
a21.c->a22.div; a21.c->a23.div; a21.c->b2.div; a21 start x2; x2.1->a21.id
```

and the semantics of PC allows the communications in the second line to be interleaved with those in the first line in any possible way; in the terminology of [7], the calls to `elim_row(A1,A3)` and `solve(A'2)` run in parallel.

This corresponds to implementing Gaussian elimination with pipelined communication and computation as in § 8.3 of [13]. Indeed, as soon as any row has been reduced by all rows above it, it can apply `solve` to itself and try to begin reducing the rows below. It is a bit surprising that we get such parallel behaviour by straightforwardly implementing an imperative algorithm; the explanation is that EPP encapsulates the part of determining which communications can take place in parallel, removing this burden from the programmer.

5 Fast Fourier Transform

We now present a more complex example: computing the discrete Fourier transform of a vector via the *Fast Fourier Transform* (FFT), as in Algorithm 13.1 of [13]. We assume that n is a power of 2. In the first call, $\omega = e^{2\pi i/n}$.

```
procedure R_FFT(X,Y,n,ω)
if n = 1 then y_0 = x_0
         else R_FFT(⟨x_0, x_2, ..., x_{n-2}⟩,⟨q_0, q_1, ..., q_{n/2}⟩,n/2,ω^2)
              R_FFT(⟨x_1, x_3, ..., x_{n-1}⟩,⟨t_0, t_1, ..., t_{n/2}⟩,n/2,ω^2)
              for j = 0 to n − 1 do    y_j = q_{(j % n/2)} + ω^j t_{(j % n/2)}
```

Implementing this procedure in PC requires two procedures `gsel_then(p,Q)` and `gsel_else(p,Q)`, where `p` broadcasts a selection of label `then` or `else`, respectively, to every process in `Q`.[2] We also use auxiliary procedures `intro(n,m,P)`, where `n` introduces `m` to all processes in `P`, and `power(n,m,nm)`, where at the end `nm` stores the result of exponentiating the value in `m` to the power of the value stored in `n` (see [7] for a possible implementation in a sublanguage of PC).

The one major difference between our implementation of FFT and the algorithm R_FFT reported above is that we cannot create a variable number of

[2] For EPP to work, the merge operator in [8] has to be extended with these procedures.

fresh processes and pass them as arguments to other procedures (the auxiliary vectors $\boldsymbol{q}$ and $\boldsymbol{t}$). Instead, we use $\boldsymbol{y}$ to store the result of the recursive calls, and create two auxiliary processes inside each iteration of the final for loop.

```
fft(X,Y,n,w) = if n.is_one
               then gsel_then(n,join(X,Y)); n -> w[then]; base(hd(X),hd(Y))
               else gsel_else(n,join(X,Y)); n -> w[else];
                    n start n'; n.half -> n'; intro(n,n',Y);
                    w start w'; w.square -> w'; intro(w,w',Y);
                    n: n' <-> w; w: n' <-> w';
                    fft(even(X),half1(Y),n',w');
                    fft(odd(X),half2(Y),n',w');
                    n' start wn; n': w <-> wn; power(n',w,wn);
                    w start wj; w.1 -> wj; intro(w,wj,Y);
                    combine(half1(Y),half2(Y),wn,w,wi)

base(x,y) = x.c -> y

combine(Y1,Y2,wn,w,wj) = combine1(hd(Y1),hd(Y2),wn,wj); w.c -> wj.mult;
                         combine(tl(Y1),tl(Y2),wn,w,wj)

combine1(y1,y2,wn,wj) = y1 start q; y1.c -> q; y1: q <-> y2;
                        y2 start t; y2.c -> t; y2: t <-> y1; y2: t <-> wj;
                        q.c -> y1; wj.c -> t.mult; t.c -> y1.add;
                        q.c -> y2; wn.c -> t.mult; t.c -> y2.add
```

The level of parallelism in this implementation is suboptimal, as both recursive calls to `fft` use `n'` and `w'`. By duplicating these processes, these calls can run in parallel as in the previous example. (We chose the current formulation for simplicity.) Process `n'` is actually the main orchestrator of the whole execution.

6 Graphs

Another prototypical application of distributed algorithms is graph problems. In this section, we focus on a simple example (broadcasting a token to all nodes of a graph) and discuss the limitations of implementing these algorithms in PC.

The idea of broadcasting a token in a graph is very simple: each node receiving the token for the first time should communicate it to all its neighbours. The catch is that, in PC, there are no primitives for accessing the connection graph structure from within the language. Nevertheless, we can implement our simple example of token broadcasting if we assume that the graph structure is statically encoded in the set of available functions over parameters of procedures. To be precise, assume that we have a function `neighb(p,V)`, returning the neighbours of `p` in the set of vertices `V`. (The actual graph is encapsulated in this function.) We also use `++` and `\` for appending two lists and computing the set difference of two lists. We can then write a procedure `broadcast(P,V)`, propagating a token from every element of `P` to every element of `V`, as follows.

```
broadcast(P,V) = bcast(hd(P),neighb(hd(P),V));
                 broadcast(tl(P)++neighb(hd(P),V),V\neighb(hd(P),V))

bcast(p,V) = bcast_one(p,hd(V)); bcast(p,tl(V))

bcast_one(p,v) = p.c -> v.id
```

Calling `broadcast(`$\langle$`p`$\rangle$`,G)`, where `G` is the full set of vertices of the graph and `p` is one vertex, will broadcast `p`'s contents to all the vertices in the connected

component of `G` containing `p`. Implicit parallelism ensures that each node starts broadcasting after it receives the token, independently of the remaining ones.

This approach is not very satisfactory as a graph algorithm, as it requires encoding the whole graph in the definition of `broadcast`, and does not generalise easily to more sophisticated graph algorithms. Adding primitives for accessing the network structure at runtime would however heavily influence EPP and the type system of PC [8]. We leave this as an interesting direction for future work, which we plan to pursue in order to be able to implement more sophisticated graph algorithms, e.g., for computing a minimum spanning tree.

7 Related Work and Conclusions

To the best of our knowledge, this is the first experience report on using choreographic programming for writing real-world, complex computational algorithms.

Related work. The work nearest to ours is the evaluation of the Chor language [16], which implements the choreographic programming model in [3]. Chor supports multiparty sessions (as π-calculus channels [15]) and their mobility, similar to introductions in PC. Chor is evaluated by encoding representative examples from Service-Oriented Computing (e.g. distributed authentication and streaming), but these do not cover interesting algorithms as in here.

Previous works based on Multiparty Session Types (MPST) [14] have explored the use of choreographies as protocol specifications for the coordination of message exchanges in some real-world scenarios [10,19,20]. Differently from our approach, these works fall back to a standard process calculus model for defining implementations. Instead, our programs are choreographies. As a consequence, programming the composition of separate algorithms in PC is done on the level of choreographies, whereas in MPST composition requires using the low-level process calculus. Also, our choreography model is arguably much simpler and more approachable by newcomers, since much of the expressive power of PC comes from allowing parameterised procedures, a standard feature of most programming languages. The key twist in PC is that parameters are process names.

Conclusions. Our main conclusion is that choreographies make it easy to produce simple concurrent implementations of sequential algorithms, by carefully choosing process identifiers and relying on EPP for maximising implicit parallelism. This is distinct from how concurrent algorithms usually differ from their sequential counterparts. Although we do not necessarily get the most efficient possible distributed algorithm, this automatic concurrency is pleasant to observe.

The second interesting realisation is that it is relatively easy to implement nontrivial algorithms in choreographies. This is an important deviation from the typical use of toy examples, of limited practical significance, that characterises previous works in this programming paradigm.

References

1. M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In M. Lumpe and W. Vanderperren, editors, *SC*, volume 4829 of *LNCS*, pages 34–50. Springer, 2007.
2. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
3. 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.
4. M. Carbone, F. Montesi, and C. Schürmann. Choreographies, logically. In P. Baldan and D. Gorla, editors, *CONCUR*, volume 8704 of *LNCS*, pages 47–62. Springer, 2014.
5. M. Carbone, F. Montesi, C. Schürmann, and N. Yoshida. Multiparty session types as coherence proofs. In L. Aceto and D. de Frutos-Escrig, editors, *CONCUR*, volume 42 of *LIPIcs*, pages 412–426. Schloss Dagstuhl, 2015.
6. Chor. Programming Language. `http://www.chor-lang.org/`.
7. L. Cruz-Filipe and F. Montesi. Choreographies, computationally. *CoRR*, abs/1510.03271, 2015. Submitted for publication.
8. L. Cruz-Filipe and F. Montesi. Choreographies, divided and conquered. *CoRR*, abs/1602.03729, 2016. Submitted for publication.
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. P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In T. Ball and M. Sagiv, editors, *POPL*, pages 435–446. ACM, 2011.
11. P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP*, LNCS, pages 194–213. Springer-Verlag, 2012.
12. M. Gabbrielli, S. Giallorenzo, and F. Montesi. Applied choreographies. *CoRR*, abs/1510.03637, 2015.
13. A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Pearson, 2003. 2nd edition.
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. 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.
16. F. Montesi. *Choreographic Programming*. Ph.D. thesis, IT Univ. of Copenhagen, 2013. `http://fabriziomontesi.com/files/choreographic_programming.pdf`.
17. F. Montesi. Kickstarting choreographic programming. *CoRR*, abs/1502.02519, 2015.
18. F. Montesi and N. Yoshida. Compositional choreographies. In P. R. D'Argenio and H. C. Melgratti, editors, *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.
19. N. Ng and N. Yoshida. Pabble: parameterised scribble. *Service Oriented Computing and Applications*, 9(3-4):269–284, 2015.
20. N. Yoshida, P. Deniélou, A. Bejleri, and R. Hu. Parameterised multiparty session types. In C. L. Ong, editor, *FOSSACS*, volume 6014 of *LNCS*, pages 128–145. Springer, 2010.